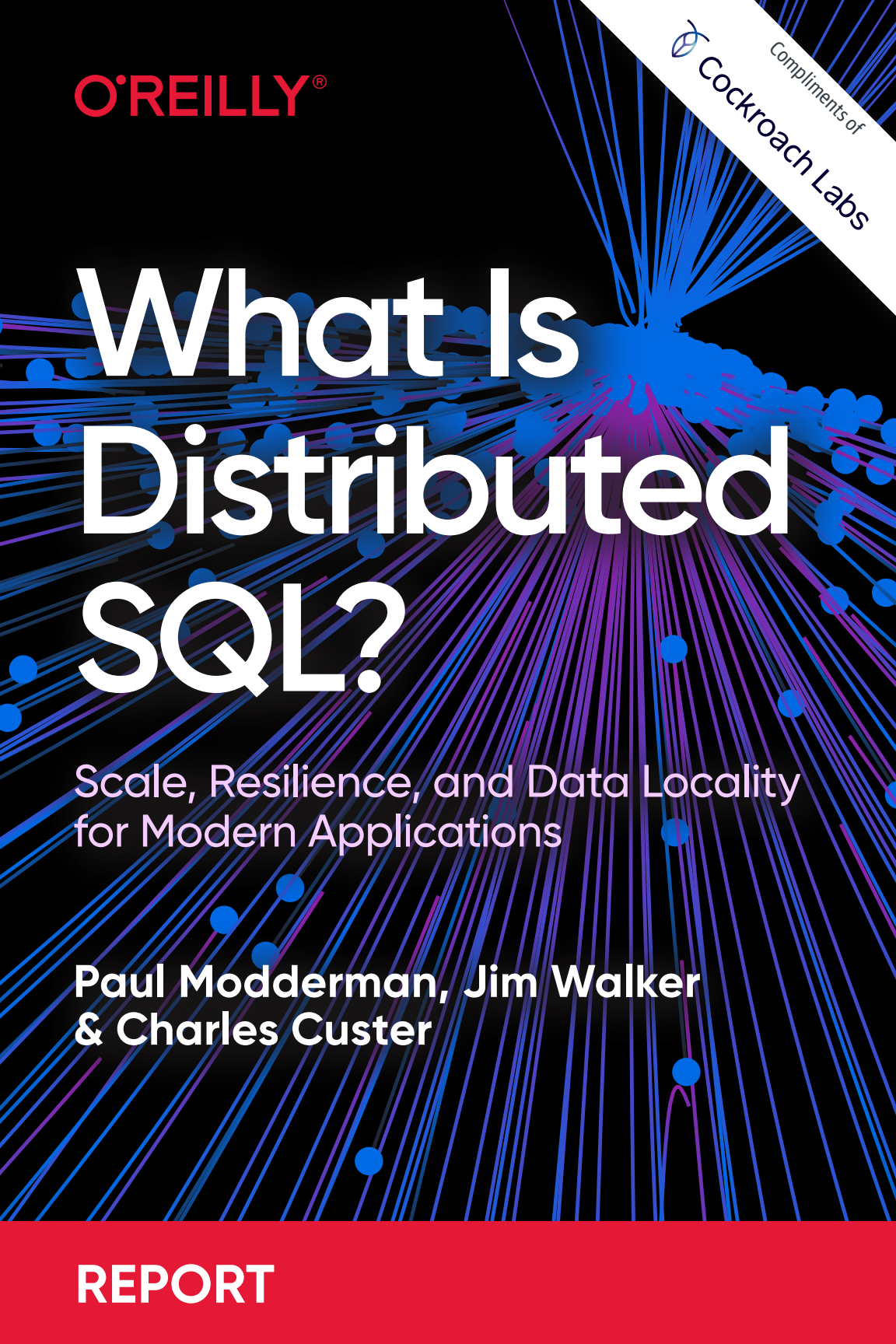
O'REILLY®
Compliments of
Cockroach Labs
What Is Distributed SQL?
Scale, Resilience, and Data Locality for Modern Applications
Paul Modderman, Jim Walker & Charles Custer
REPORT

CockroachDB

Build what you dream.

Never worry about your database again.

```
/* Give all of your apps effortless scale, bulletproof resilience,
and low latency performance for users everywhere. */
```

A hassle-free cloud SQL database

Elastic scale that won't break your budget

Compatible with PostgreSQL

Get started for free, instantly.

cockroachlabs.com/product

Trusted by innovators

COMCAST

MYTHICAL™

BOSE

Kami

ALLSAINTS

Shipt

What Is Distributed SQL?

Scale, Resilience, and Data Locality for Modern Applications

Paul Modderman, Jim Walker, and Charles Custer

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

What Is Distributed SQL?
by Paul Modderman, Jim Walker, and Charles Custer

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Andy Kwan
Development Editor: Gary O'Brien
Production Editor: Katherine Tozer
Copyeditor: nSight, Inc.

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

February 2022: First Edition

Revision History for the First Edition
2022-02-16: First Release

This work is part of a collaboration between O'Reilly and Cockroach Labs. See our statement of editorial independence.

978-1-098-11645-3

[LSI]

Table of Contents

The Distributed Mindset

Modern developers procure exactly the amount of storage, compute, and memory they need for a given application; choose from easily applied container solutions with prebuilt components; and can deploy them anywhere and everywhere on earth in a matter of hours (and sometimes minutes). Cloud storage and compute can easily bend to accommodate spikes in application needs. A disaster isn't a disaster because a managed restore from a managed backup is just a click away. Enterprise architects can retire to fulfilling lives of leisure.

But this life of relative comfort is recent. Not long ago, application development meant procuring physical hardware, finding rack space, and painstakingly installing OS, system software, and application software one server at a time. And—often most painful of all—working through backup solutions and finding safe off-site locations for boxes full of tapes.

Developers of today naturally want to avoid this pain and move on to the ease and elegance of distributed cloud services. And to do so, they are adopting a distributed mindset. This paradigm, however, requires a shift in the way we think and a focus on three key challenges:

Scale
: Workloads that need lots of resources get them on demand, and when demand decreases, the extra resources are released to other work.

Resilience
: It doesn't matter if one computer or one datacenter disappears. Applications exist in computing fabric and simply survive failures.

Locality
: One must put the computing resource where it needs to go, no matter where that is.

The distributed mindset and cloud computing concepts are understood and accepted in the areas of storage, compute, development, deployment, and analytical data—but there is a straggler: the transactional database. However, in recent years, our traditional SQL (structured query language) databases have been reimagined to incorporate this distributed mindset, and an emerging category known as *distributed SQL* has been established.

Distributed SQL fits all applications and eliminates complex challenges like sharding from traditional RDBMS systems. Distributed SQL databases can scale to become global without introducing the consistency trade-offs found in NoSQL solutions. Distributed SQL comes to life through cloud computing, where legacy databases simply can't rise to meet this elastic and ubiquitous paradigm.

This report will take you through why the distributed SQL category has emerged, what it consists of, how it can be (and is) used, and what the future holds for it.

Let's start with the motivation.

CHAPTER 1

Why a New Type of Database?

In this chapter, we examine two major forces compelling a new type of database: application development and business needs.

Development has evolved to fit the distributed mindset. Applications are global, developers expect the best tools, and consumers have higher expectations of resilience. But the databases backing global applications sometimes hold them back with centrally managed transactions.

The distributed mindset also serves modern businesses. Enterprise IT needs ways to comply with increasingly complex legal requirements for data. Businesses expect better availability for large workloads. But the databases that run in enterprises don't scale gracefully.

The next two sections go into more detail on the application development and business needs. The result is clear: the old paradigm for databases can't keep up with cloud infrastructure, so we need to rebuild from the ground up.

Application Development Has Evolved

Developers build wherever they like and have a world of valuable services at their fingertips. Sometimes development is just a little dollop of cloud services; sometimes it's a bunch of things. For instance, with a service like Google's Firebase, it is possible to set up data persistence and synchronization with Firebase Realtime Database, host web assets with Firebase Hosting, and ensure secure

authentication with easily integrated Firebase Authentication. Pretty simple.

The application toolkit can run entirely on small, cloud-hosted development systems. There, developers run isolated versions of their system and unit-test feature builds. When they're ready, they initiate a full suite of cloud-based automated tests. If those tests pass, code and assets automatically deploy to a global content distribution network, and automatically update a set of virtual machines or containers. Google is not alone in the distributed application development game. See Table 1-1 for an incomplete list of products offered by major cloud vendors in this same space.

Table 1-1. Cloud application development offerings from major cloud vendors

Cloud vendor	Cloud application development products
Amazon Web Services (AWS)	AWS CloudShell, AWS CodeBuild, AWS CodeCommit, AWS CodeDeploy, AWS CodePipeline, AWS Elastic Beanstalk, AWS Lambda, AWS X-Ray
Google Cloud Platform (GCP)	App Engine, Cloud Build, Cloud Code, Cloud Monitoring, Cloud Run, Firebase suite, Google Kubernetes Engine
Microsoft Azure	Azure App Service, Azure DevOps, Azure DevTest Labs, Azure Functions, Azure Monitor, Azure Pipelines

Globally available resources have become the status quo. They're accessible, distributed, and resilient. Cloud computing is not just the infrastructure that runs this work; it is the mindset that's now required for doing fast, impactful development work.

Our traditional SQL database options haven't kept up. Centralized SQL databases, even those with read replicas in the cloud, put all the transactional load on a central system. The further away a transaction happens from the user, the more the user experience suffers. Application resources like app store binaries or web assets can use content distribution networks to make downloading and running apps seem fast. But if the transactional data powering the application is greatly slowed down, fast-loading web pages mean nothing.

Distributed SQL databases solve this problem with easy scalability and data locality. They deliver a single logical database across disparate, widely distributed hardware. Anywhere on earth that has a datacenter can host additional resources for distributed SQL databases. And they support consistent transactions everywhere using

advanced consensus algorithms, thus avoiding the bottleneck of reliance on a centralized node for transaction consistency.

Some distributed SQL databases even allow for data relevant to a particular region to be tightly bound there. Relevant user data is close to where it is needed, which speeds up the user experience and eases the challenges associated with scale. Chapter 2 goes more in depth on the particulars of scale and consensus.

Business IT Needs Have Evolved

Consumers have three expectations of our applications: they must be fast; they must be correct; and they must be always on and available. These are critical expectations since consumers' patience has waned and the ability to move on to another option is radically easy. If you opened an account and your balance was wrong, would you have confidence in your bank?

People also expect their apps and services to work everywhere, without delay. If you take a trip from London to Sydney, you'll still have quick access to your Gmail and Instagram pictures, and those services will adjust to your new presence.

Finally, they don't lose availability just because of some backend infrastructure problem—they naturally survive and can service any and all requests. If you run a business with a customer-facing application, your customers' app experience must be as good as using Facebook or Instagram and must be at the top of your priority list.

The cloud helps meet these requirements, and businesses know it. See Table 1-2 for Gartner's recorded and projected cloud revenue forecast. Infrastructure as a service (IaaS) and platform as a service (PaaS) will more than double over just four years! The business IT environment is increasingly cloud-dependent and is seeing enormous productivity growth from adopting cloud-first approaches to solutions. IT departments that learn to think from a distributed perspective for their data, operations, and compute workloads far outpace their on-premises-centered competitors.

Table 1-2. Projected worldwide public cloud service revenue forecast into 2022 (billions of US dollars)[a]

	2018	2019	2020	2021	2022
Cloud business process services (BPaaS)	41.7	43.7	46.9	50.2	53.8
Cloud application infrastructure services (PaaS)	26.4	32.2	39.7	48.3	58.0
Cloud application services (SaaS)	85.7	99.5	116.0	133.0	151.1
Cloud management and security services	10.5	12.0	13.8	15.7	17.6
Cloud system infrastructure services (IaaS)	32.4	40.3	50.0	61.3	74.1
Total market	**196.7**	**227.8**	**266.4**	**308.5**	**354.6**

[a] BPaaS stands for business process as a service; IaaS stands for infrastructure as a service; PaaS stands for platform as a service; Saas stands for software as a service. Note the totals may not add up due to rounding. Source: Gartner (November 2019).

Again, existing SQL databases haven't kept up. They are not particularly resilient to outages in infrastructure zones, especially when the zone that goes offline contains the main transactional instance. And while NoSQL alternatives help with this particular challenge, they can't promise transactional consistency and present developers complexity around a document model that lacks the elegance and power of the relational model (for example, normalization, referential integrity, secondary indexes, and joins).

Distributed SQL databases fill in the gaps. They are highly resilient to any type of outage. They deliver the time-tested and familiar SQL query syntax that developers know and love and promise truly consistent ACID transactions. And most importantly, they are aligned with the elastic scale and ubiquitous nature of our cloud infrastructure.

The Evolution of the Database

In the cloud, the distributed mindset should permeate everything we do. This will allow us to take advantage of the elastic scale, resilience, and ubiquity of cloud infrastructure. And this especially applies to the database.

Cloud databases have iterated to try to meet these needs but continue to fall a bit short of delivering on these requirements. While we have moved the ball forward, we have not completely closed the gap. Let's explore our progress to date:

Lift and shift
: The first approach to transactional databases in the cloud involved simply lifting traditional RDBMS instances and shifting into hyperscaler datacenters on virtual machines. Amazon Relational Database Service (RDS) and Google Cloud SQL are examples of this. Both are great options for some workloads and wildly popular with developers. But they don't leverage cloud distributed thinking; their value lies in ease of access/deployment (and capex/opex trade-offs). Transactions against these databases are solid but locked to a single instance. Scale is achieved through either deploying on a more high-powered instance or using manual sharding to gain horizontal scale. While useful, we haven't quite cracked the elastic scale promise and must rely on active/passive configuration for resilience. We no longer have to deal with complex operations to deploy and manage, but we haven't taken advantage of cloud infrastructure with this approach.

Move and improve
: Some have chosen to augment existing databases with distributed technologies that automate sharding and enhance the database's survivability by reworking a single layer of the database. This class of cloud database has moved a legacy database to the cloud and improved part of it to meet some cloud requirements. Amazon Aurora is a good example. Its major innovation is a distributed storage system that acts as a lake of data under many read-only Postgres instances. It improves availability, is simple to scale for reads, and looks and feels like a traditional SQL database. However, it uses a single node for writes, which can create bottlenecks and limit the ability to scale this database beyond a single region. Additionally, there are several new databases that automate sharding, but they struggle with anything beyond a single region.

NoSQL
: NoSQL options are often considered as a cloud database and they definitely deliver value in this environment. Most notably, Apache Cassandra, DynamoDB, and MongoDB provide

lightning speed with often schemaless, flexible structures. They scale easily. For data that does not require tight global consistency, like social media feeds, NoSQL is an excellent choice. But when developers need tight consistency for important things like financial transactions, NoSQL can't meet those demands. Distributed transactions are not guaranteed to be correct and can only get you eventual consistency. Further, the document model limits their ability to deliver the elegance of the relational model—NoSQL databases are missing critical concepts like referential integrity, secondary indexes, joins, and normalization.

The Way Forward

These approaches cover a lot of ground, but the gap is clear. With application development and business needs both adopting the distributed mindset, databases need a new set of features to keep up:

Ease of scale
: It should be easy to have as little or as much cloud power behind your database as you need.

Always-on resilience
: Downtime should be eliminated.

Data locality
: For both compliance and performance, you need to be able to tie certain data to certain geographic locations.

SQL
: Developers love the SQL language for its expressive data features, and since they're already familiar, many tasks are made easier.

ACID compliance
: If you commit a record, you should be able to trust that subsequent queries will match that new state.

Doing globe-spanning database transactions consistently across a physical universe limited by the speed of light is *an extremely difficult software engineering challenge*. But it's possible. A new set of players has emerged who rise to meet the distributed SQL mindset.

We've got the motivation down. Next, we'll dive deep on those features, and how these new players are getting them done.

CHAPTER 2

A Definition of Distributed SQL

In Chapter 1, we established the development and business motivation for distributed SQL solutions. We identified the core feature set: ease of scale, resilience, data locality, SQL language support, and ACID compliance. The distributed mindset of scale, resilience, and locality is at the heart of distributed SQL solutions that fit these features.

To define distributed SQL, we'll highlight the technical components with a brief overview of three key distributed database concepts and Google's seminal white paper "Spanner: Google's Globally Distributed Database." Then we'll investigate how emerging players take distributed SQL to polished product readiness by incorporating enterprise requirements. By the end of this chapter, we'll have a definition that includes both nerd-friendly tech functions and state-of-the-art product features currently on the market.

The Key Concepts

Before we define distributed SQL, we should describe a few key concepts that drive this new technology. The first is CAP theorem, which identifies the functional trade-offs that distributed databases will need to make. The second is distributed consensus, which ensures that when data is stored on multiple machines, those machines reach agreement on a single version of the truth. In distributed SQL databases, this is most often implemented through the Raft algorithm. The third key concept is multiversion concurrency control (MVCC), ensuring that transactions in the database don't

overlap each other and create strange read or write scenarios. After we lay out the key concepts, we'll see Google's Spanner project as a pioneer in distributed SQL.

CAP Theorem

Computer scientist Eric Brewer published the foundational pieces of the CAP theorem in the late 1990s. It lays out the boundaries for what a distributed data store can guarantee about three desirable read and write capabilities:

Consistency
: Every data node provides the latest state of requested data. If it doesn't have the latest state, it won't provide a state.

Availability
: Every data node can always read and write data in the system.

Partition tolerance
: The data system works even if partitions occur in the network. *Partitions* mean network functionality has broken between two or more data nodes in a distributed system.

These three capabilities relate in such a way that a distributed data store can only guarantee two of them during a partition event. Looking at the Venn diagram (Figure 2-1), CAP theorem could be restated to mean that there is no central three-circle overlap—no distributed data store can guarantee all three at the same time.

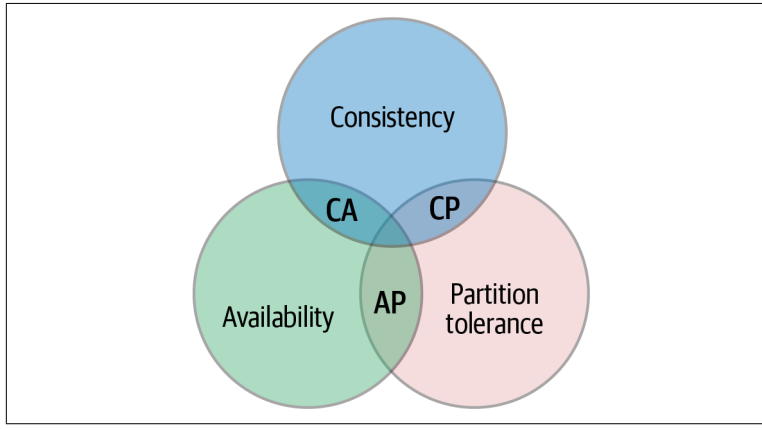

Figure 2-1. CAP theorem

Since CAP theorem puts a "pick two and do your best with the third" restriction on distributed systems, it's possible to group databases into three groups, which represent different intersections of the C, A, and P circles—and thus different strategies for overcoming the challenge of the circle they can't guarantee:

- The CA group chooses consistency and availability as the guarantees. Since this eliminates the "partition" requirement, this does not describe a distributed database. Our traditional relational databases such as Microsoft SQL Server, Oracle, Db2, MySQL, and PostgreSQL land in this category.
- The AP group chooses availability and partition tolerance. MongoDB, Apache Cassandra, and Amazon DynamoDB go this route. They offer *eventual consistency*, allowing applications to write to any node at any time with a chance of inconsistencies, and then detect and reconcile those inconsistencies.
- The CP group chooses consistency and partition tolerance. Google Cloud Spanner, CockroachDB, and YugabyteDB fall into this corner of the chart. Distributed databases in this group mitigate their sacrifice on availability by spreading nodes to different availability zones and regions. This is the most difficult combination to implement.

Datacenter and network reliability have matured to such a degree that making the CP choice can still result in five nines of availability—99.999% uptime of distributed SQL databases running on cloud systems. CAP theorem author Eric Brewer wrote in a 2017 Google paper: "Does this mean that Spanner is a CA system as defined by CAP? The short answer is 'no' technically, but 'yes' in effect and its users can and do assume CA."

Distributed Consensus with Raft

Fault tolerance requires that each individual data object will exist in multiple places at one time. How do a distributed SQL solution's many nodes keep their stories straight among one another? Raft and Paxos are consensus algorithms designed with this problem in mind. For this paper, we will focus on Raft.

Raft allows a system to manage consistency across multiple copies of data through consensus and not synchronization. Members of a Raft group agree on the state of data through some key concepts we

will define below, but most importantly, they gain quorum across the members, so that a majority agree on state. This requires an odd number of members.

The design of Raft splits consensus into three areas: leader election, log replication, and safety. Let's examine each area to understand how consensus happens in distributed systems through Raft. (We will explore Raft here, but there is also a graphical version online.)

Leader election

Systems implementing Raft split nodes into groups. Each group has a leader node, which handles incoming data requests from clients and sends its log to other nodes. The leader is the authoritative source of truth within the replica group. The leader election process provides Raft-implementing systems a way to determine which node is the leader, and when to redetermine leadership. A node's duration of being leader is called its *term*, with a monotonically increasing number identifier.

A leader node sends *heartbeat* messages to its followers. When a follower node doesn't receive a heartbeat from the leader after a certain interval, it makes itself a *candidate* node. A candidate node assigns itself a new term number and sends messages to other nodes to request their vote. If the candidate node receives votes from a majority of other nodes, it becomes the leader and starts sending heartbeat messages.

Other nodes in the group may have undergone the follower-to-candidate process as well. Those nodes have chosen their own term number and are still waiting for election results. Recall that object data flow goes one way in the Raft group: the leader system sends log data messages to followers. When the newly elected leader node sends log data messages, it also sends the term number it was elected under. When candidate nodes receive log data messages with a higher term number than their own, they accept that log entry and revert to follower state.

It's possible for an election cycle to come away with no clear winner node. Raft minimizes this by setting randomized heartbeat timeouts on each node as it enters a new election cycle, so that not all nodes enter the candidate state at one time.

Leader election guarantees that the group of Raft nodes continues to remain available when an existing leader fails. Electing leaders and handling term numbers ensures that our next key distributed consensus concept, log replication, is possible.

Log replication

The leader of a Raft group maintains a log of data activity. As the data comes from the leader and moves on to the followers, the leader needs to keep all the node logs in sync with its own. It does this through the Raft log replication process.

A client-initiated request to operate on data is added to the leader's log. This log entry includes an incremented index marking the log entry's position in the log, and the term number that the leader has been elected with. Then the leader sends all the follower nodes a message containing the log entry, index, election term, and previous index, until all the followers replicate it. When a majority of followers respond in the affirmative, the leader executes the data change in the log and responds to the client with a success message.

The key to maintaining log consistency is that followers reject log entries if they cannot find a log entry matching the previous index. The leader then begins a process of moving backward through the follower's indices and finding where the two logs agree; then the leader forces the follower to duplicate logs from there.

Safety

In election and replication processes, Raft ensures that the following five rules apply, to ensure that all nodes take the same logs in the same order. Ongaro and Ousterhout from Stanford briefly lay out the safety mechanisms in their original Raft paper:[1]

Election safety
: At most one leader can be elected in a given term.

Leader append-only
: A leader never overwrites or deletes entries in its log; it only appends new entries.

1 Ongaro, Diego, and John Ousterhout. "In Search of an Understandable Consensus Algorithm". Paper included in the Proceedings of USENIX ATC '14, Philadelphia, PA, June 2014, 305–20.

Log matching
: If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Leader completeness
: If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.

State machine safety
: If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

Taking care during leader elections helps these rules apply. Candidate nodes give voter nodes information about their log. If the voter node finds that the candidate has an older log, it refuses to cast a vote for the candidate.

Now that we understand Raft's key concepts of leader election, log replication, and safety keeping consensus across systems, let's review multiversion concurrency control for a window into how distributed SQL systems keep queries and updates isolated for maximum reliability.

Multiversion Concurrency Control

With multiple machines in multiple physical locations powering a distributed SQL database, the likelihood of overlapping writes and reads happening to the same data objects is high. Imagine a bank employee needing access to a large amount of data that takes several seconds to fully query. During that time, transactions may occur against the data being queried. Without MVCC, the employee may see an inconsistent set of results. In the case of writing data, without MVCC, a transaction initiated earlier in time might actually overwrite a transaction initiated later in time. It's hard to rely on your data if you can't depend on actions being recorded in the order they happen in the real world.

How does MVCC help? By keeping actions isolated from one another in time. To achieve the isolation required to preserve read or write integrity, the distributed SQL engine appends a reliable timestamp to every value stored. (See "Google Spanner" for more on its TrueTime API—the provider of the reliable timestamp for

Spanner.) With the timestamp of a transaction appended to the key, the database is storing multiple versions of data.

This makes it possible for the storage engine to provide data at a snapshot in time. A query initiated at a point in time will not retrieve the records whose timestamps exceed the query's initiation time. A transaction initiated at a certain time cannot complete if there are outstanding transactions that depend on earlier versions of the same object.

We wrap the MVCC discussion with a recommendation: application development with distributed SQL must involve try-catch and retry mechanisms for transactions that fail due to version snapshot control.

Google Spanner

In the implementation of distributed SQL, everything started with Spanner. It takes the key elements we've described (distributed consensus and multiversion concurrency control) and gives distributed SQL its first steps as a real-life database product. Spanner reached the public eye with the publication of a paper describing it in 2012.[2] The paper's abstract describes it well: "Spanner is Google's scalable, multiversion, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions."

Google had data needs that stretched the boundaries of existing database systems. Very large datasets like Google Ads had grown to enormous size, spread all over the world, in a highly sharded MySQL system. Having many shards makes management, maintenance, and upgrades incredibly difficult. And Ads wasn't the only system facing this problem. It was time to make it easier for programmers to build applications that fit the growing need for global transactional data. The Google team invented Spanner and put into practice the previously discussed consensus and MVCC concepts, using a different consensus algorithm called Paxos.

2 Corbett, James C. et al. "Spanner: Google's Globally-Distributed Database". Paper included in the Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12). USENIX Association, Hollywood, California, October 2012, 251–64.

As we discussed in "Multiversion Concurrency Control", timestamps are a key factor in distributed SQL systems. For Spanner, accurate timestamping is achieved by the TrueTime API. TrueTime is a global network of time servers that use multiple factors such as GPS and hardware atomic clocks to achieve incredibly accurate synchronized timestamping. By using TrueTime to timestamp transactions all across the global network of data that is constantly replicating, Spanner can ensure correct ordering, even when not all members of a replica group are present.

Since Spanner can achieve consistency even when data is not stored geographically close to a client, it opens the door to choosing the best places to store data. Keeping certain sets of data geographically close to common usage points achieves the *data locality* our distributed SQL systems need.

In a separate paper published in 2017,[3] Google authors described updating Spanner to include standard SQL language support and ACID compliance in transactions. Those are the last two of our distributed SQL feature requirements. We now have the technical backbone to make distributed SQL databases work.

Completing the Picture

Walking through CAP theorem, Raft, and the amazing Spanner technology gives us the technical bones of distributed SQL. But the most complete definition should encompass more than just the tech. Vendors must implement it and meet the needs of the market. Distributed SQL databases mature into products by taking this technical wizardry and adding features that meet enterprise requirements. In this section, we'll break down the four most fundamental parts of these requirements.

Management

This could be said for any distributed system: if it's mission critical, it needs to be managed. Distributed SQL is no different. Enterprises need a variety of ways to visualize and control database resources.

3 See David A. Bacon et al. "Spanner: Becoming a SQL System." Paper included in the Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17). Association for Computing Machinery, New York, NY, May 2017, 331–43. *https://doi.org/10.1145/3035918.3056103*.

They need web-based tools that can quickly and visually display key database metrics. Even if a key feature of distributed SQL is resilience to node loss, administrators need to be informed. Common information available in web consoles includes replication status, uptime, and active sessions. Many distributed SQL systems include RESTful APIs that allow access to these metrics by web service. Google Cloud Spanner has a comprehensive set of visual management and API-driven information tools.

Enterprises need command-line interface (CLI) tools. Orchestrating computing resources doesn't happen in isolation; tasks like starting and stopping databases need to be scriptable because they need to happen in concert with other systems. For example, CockroachDB includes CLI tools that, in addition to starting and stopping, allow SQL statements.

Security

Beyond monitoring and control, enterprises need security. Distributed SQL databases support a variety of authentication mechanisms. Username and password are a given for any system, but many others are available. For the data protection side of security, almost all distributed SQL products support row-level permissions and in-motion and at-rest encryption.

Integration

These days, no enterprise runs just one large-scale system. They need to integrate many systems. CockroachDB and YugabyteDB have wire compatibility with PostgreSQL, so integration tools that connect to databases are likely to have drivers that already work. Google Cloud Spanner, while not wire-compatible with other standard database drivers, has deep integration with other Google Cloud Platform products and a growing set of open source connectors.

Databases are always part of a larger (and often quite complex) enterprise data architecture that will allow for streaming analytics, data warehousing, AI/ML, and other data operations. The database should fit into this landscape and be able to use shared storage or change data capture (CDC) to integrate and work with these systems.

Everything as a Service

If a company does not sell technology as a core product, it may not wish to spend time and resources on complex technology management. Distributed SQL products are available as managed services—in some cases like Google Cloud Spanner, they are *only* available as a service. Others offer self-hosted and even open source versions of their core technology. In today's cloud-focused distributed mindset, having distributed SQL available as a service is the bare minimum. Every major distributed SQL vendor offers a managed service option.

One Definition to Rule Them All

Based on the distributed mindset of scale, resilience, locality, and the market needs of enterprises who use distributed SQL solutions, here is our broad definition:

> A distributed SQL database is a relational database that maintains data consistency across a large number of machines and geographic locations. Its core technical features are ease of scale, resilience, data locality, SQL language support, and ACID compliance. Distributed SQL solutions provide enterprise-grade management tools, security, and integration capabilities. Applications that use distributed SQL solutions often choose vendor-managed solutions, getting global relational databases as a service on the same level as storage or compute as a service.

In Chapter 3, we'll take distributed SQL out of theory and into practice to see how enterprises and products are using it today.

CHAPTER 3

Distributed SQL in Action

In Chapter 1 we established the *why* of distributed SQL. In Chapter 2 we put some rigor around the *what*. In this chapter, we'll look at examples of distributed SQL in action across three industries: telecom, retail, and gaming. Base features laid out earlier in the report will see real-world action. There's a lot of nuance here; for example, you'll notice how high availability is key to both retail and telecom, but for very different business reasons. Similarly, sometimes scale is valuable because a company may not know when spikes occur; other times, scale is a primary value because it can be predicted. Each value that distributed SQL brings to the table varies in each real-world application, and each industry requires a unique mix of them.

After we look at some industry examples, we'll briefly discuss a couple of themes that run across them.

Industry Example: Telecom

In our examples, *telecom* means both internet service provider (ISP) and Voice over Internet Protocol (VoIP) provider. For today's world, both are absolutely vital. Companies in both subspaces are choosing distributed SQL solutions to give their customers dependable service.

Usage

A major US telecommunications company uses a virtual customer support agent as the triage point for customer requests. This virtual agent was originally built based on a standard RDBMS—which caused major pain when a cloud provider connectivity issue made the database unavailable. To mitigate this in the future, this telecom reengineered the virtual agent's backend to run on a distributed SQL solution with high availability.

A Greek telecommunications company provides businesses with VoIP telephony. A build-out of their new Session Initiation Protocol (SIP) platform required a global database that can live anywhere—on-premises or in the cloud—and provide high performance on both volume and latency. Providing VoIP services to businesses means ensuring that those businesses never miss a call from their customers, so the database solution also required extreme resilience to regional outages.

Core Needs Addressed

These telecom companies focused on two of the major distributed SQL features: *resilience* and *speed*. In particular for virtual agents, losing the state of a customer chat session is a major impact to serving that customer. Telecom companies' customers—both consumer and business—value resilient service from their providers. If getting support causes headaches, those telecoms can quickly see customer churn. Data locality provides the performance and compliance that cross-border communications require.

Deployment

Hybrid deployment of distributed SQL means that some servers are hosted in private datacenters and some are hosted in public clouds. In both cases here, hybrid deployment was key. Telecoms often have their own datacenters, so their distributed computing landscape includes both on-premises and cloud computing.

Industry Example: Retail

Distributed SQL in retail includes both directly serving retailers in their sales efforts and in the ecosystem of vendors who crunch data to support upselling initiatives. Selling to anyone in the world means having data capabilities all over the world: the distributed mindset at work.

Usage

A global retailer with hundreds of physical stores in dozens of countries faced significant "distributed" challenges. A set of legacy systems based on MySQL limited their ability to scale, and availability was becoming an issue more and more often. They moved to using a set of managed instances of a distributed SQL system to process orders for wholesale partners. Success of this application has laid plans for further electronic data interchange (EDI) integration and later moving all product data to the distributed SQL system.

A data platform that helps retailers match promotions and tactics to customers' usage habits chose to consolidate several types of existing database systems into one system. This simplified their data infrastructure and enabled extremely responsive, fast query operations. Moving to a single database spread across multiple availability zones made their data more highly available than ever.

A business-to business (B2B) company creates white-label experiences for its retail customers, tuned for before and after a purchase. A growing set of retailers and brands onboarding to its platform made scaling on AWS databases very expensive, especially considering large-impact scaling events like Black Friday and Cyber Monday. Switching to multicloud distributed SQL saved money versus AWS solutions, kept them in compliance with customers' EU General Data Protection Regulation (GDPR) requirements, and simplified operations.

Core Needs Addressed

Retailers and their ecosystem partners rely on many of the features the distributed SQL players now provide. Two of them from our earlier list stand out in particular: *scale* and *resilience*. Major retail events like Black Friday create the need for scale-up and scale-down

operations. When it comes to always-on resilience, all you have to do is consider just how vital promotions, transactions, and customer experience are to the retail world. A poor experience or a missed promotion can mean lots of sales left on the table.

Deployment

Two styles of deployment are common in the retail sector: *multicloud* and *managed cloud*. A multicloud deployment means hosting replicas of the distributed SQL system in datacenters managed by different cloud providers. For example, one CockroachDB system addressable from one entry point might have hosts in DigitalOcean, AWS, and GCP. Multicloud is particularly helpful in minimizing latency while maintaining availability.

Managed cloud means letting a service provider do the heavy lifting of operations for your distributed SQL database. Often, you still have a limited choice in which cloud provider hosts your database. A managed cloud deployment is especially powerful when you have a small engineering team or your technical resources do not focus on system administration, but your solutions still require distributed SQL features.

Industry Example: Gaming

Gaming is global by default. App stores, transactions, massively multiplayer environments, and more demand data that crosses the globe. With millions of gamers and hundreds of billions of dollars in the market, gaming has a huge ecosystem that expands far beyond game development.

Usage

A leading gaming company processes billions of financial transactions per year. Forecasting continued growth and new markets, they chose a distributed SQL solution to manage customer wallet data, so that georeplication could enhance responsiveness and compliance for customers no matter their location.

A game development studio has created successful games for years. As they built on the success of their intellectual property, they began to see scalability issues with their Aurora MySQL databases. They migrated to a new distributed SQL solution with automated

horizontal scaling to support heavy workloads that can suddenly spike. And since gaming is all about fast in-game experiences, georeplication keeps data located close to customers as new regional content rolls out.

A virtual world startup is creating a platform for online games that have huge player bases. They began their journey on Apache Cassandra, a distributed NoSQL database that performed well in scaling clear, simple designs. But as their query needs grew, so did their need for a more expressive and capable database design. By switching to a distributed SQL solution, they gained the modeling flexibility that comes with tried and true SQL: designing performant joins, updating entity relationships, and enabling normalized data.

Core Needs Addressed

Speed is vital to the gaming industry, and *georeplication* delivers it, both for responsive performance and for compliance across geopolitical boundaries. Billions of players and hundreds of billions of dollars in transactions can't help but be cross-border. *Scale* also comes through in many stories. In contrast to retail, in the gaming industry needing scale can be less predictable. It's easier to know when Black Friday is coming than it is to know when any particular game might become a massive viral hit.

Deployment

Multicloud deployments are common in gaming, which makes sense in light of georeplication and scaling needs. For gaming companies with lots of in-house administration skill, Kubernetes is a popular choice for orchestrating distributed SQL clusters. Vendors such as Cockroach Labs and Yugabyte offer Kubernetes operators to simplify operations.

Themes

In examining use cases, a few other themes pop up alongside industry-specific use cases. Understanding real-world use wouldn't be complete without these.

Internet of Things (IoT) use cases abound across industries. Logistics, social, gaming, and others all have needs to capture events as they happen on connected devices. For example, with the ability to scale

to enormous size and keep data localized, distributed SQL is a great way to keep a multinational company in tune with its plant operations. Or when trucks, ships, and containers all move constantly—distributed SQL presents a single source of truth on tracking and load data. Gaming consoles and peripherals, social groups, and education all benefit from global-yet-local data.

Cost is, of course, a huge factor for any company considering technology solutions. Across industries and use cases, customers report cost reduction as a major benefit of standardizing on distributed SQL. Scale is a well-known cost killer in cloud platform-as-a-service solutions like AWS DynamoDB or Azure SQL. Taking tight control of when and how to scale with Kubernetes and multicloud deployments makes distributed SQL an effective way to reduce total cost of ownership of database solutions.

Distributed SQL Is Out There

In this chapter we reviewed in-the-wild uses of distributed SQL at three industries, highlighted the benefits from the standpoint of our Chapter 1 wish list, and saw varying deployment models. Many organizations around the world are growing up from local to global, and therefore need database solutions that look at the world through those eyes. Still others start day one knowing they'll be global and choose distributed SQL out of the gate. In both cases, the argument for distributed SQL doesn't come from consensus algorithm implementation complexity—it comes from meeting a business where it needs to go.

Other sections of the book have looked at the way things are—Chapter 4 will look at what's to come.

CHAPTER 4
Looking Forward

The final chapter of this report is your crystal ball, looking future-ward at distributed SQL and other distributed-computing-enabled capabilities.

Distributed SQL: The Enabler

The distributed mindset *influenced* the creation of the distributed SQL category, but that doesn't mean it's *done*. Distributed SQL is a foundation to extrapolate *from*. As we saw in Chapter 3, businesses are choosing distributed SQL because it gives them value that wasn't possible before, and they can now extend that to other offerings.

Technologists are building capabilities into this category that impact your strategic plans. Here's a short overview on a new distributed SQL capability, to help you plot the course to the next stage of your product vision. We finish up this report with an exhortation about the distributed mindset and using it to look into the future.

Serverless

Since the mid-2010s, a distributed paradigm known as *serverless* has risen in popularity. Its most common form is code functions that can be invoked remotely from anywhere, with the distinguishing feature that while they execute on computers, they don't need to be managed in any way as physical machines (see Table 4-1 for hyperscaler serverless function solutions).

For example, in AWS Lambda a developer can create a utility function in whatever language they like and, when the function executes the hosting environment, provides the environment to execute that function. If there are no running instances of the function code, one is quickly started and responds to the request. The developer does not know or care about details at the machine level, and the hosting environment automatically scales up or down to as much computing power as needed to respond to multiple invocations of the function.

Table 4-1. Just some of the serverless function compute solutions available in the market today

Vendor	Serverless function product
Alibaba Cloud	Function compute
Amazon Web Services	AWS Lambda
Cloudflare	Cloudflare Workers
Google Cloud Platform	Cloud functions, Firebase functions
Microsoft Azure	Azure functions
Netlify	Netlify functions

Serverless adds simplicity to the distributed paradigm. A serverless architecture means that the resources are always available. Rather than planning for how many computers to buy or servers to provision, customers are billed based on their consumption of serverless resources: if you use less, you pay less.

Functions aren't the sole focus for the serverless paradigm. Low-level object storage solutions like AWS S3, Azure Blob Storage, or Google Cloud Platform Cloud Storage also fit. You can probably see where we're going: if application logic in functions can be serverless, and low-level storage can be serverless, why can't the database? Distributed SQL already automates a lot of the key capabilities required for serverless.

With some small addition, distributed SQL can become the perfect fit for serverless functions and any application. There is a possibility that, with serverless capabilities embedded into a distributed SQL database, we can completely remove dependency on operations *and reduce a database down to a simple SQL API in the cloud*.

In this new world, developers will be able to create models of data and let the database handle scale, replication, and locality requirements. As SQL statements are sent to the database, monitoring processes can automatically handle necessary physical instance operations. Developers will be able to have an attitude of "I don't care how it's accomplished machine-wise, just do the things I ask of my distributed SQL database." This will naturally increase time-to-value and greatly reduce the cost, time, and complexity of operating the database.

It's also the last piece of the puzzle for making many applications truly distributed. Store nonrelational data in object storage solutions that can replicate anywhere, make transactional SQL data distributed, automatically scalable and available, and run application code in functions that execute ephemerally without being managed on dedicated servers. Serverless distributed SQL completes the picture for always-on, autoscaled, fully managed applications.

Cockroach Labs has already introduced a beta serverless option for CockroachDB. They make CockroachDB serverless by giving their database multitenant architecture, making it so they can run a tenant-isolated SQL layer depending on a core, shared, distributed key value store that automatically includes tenant IDs in each query (Figure 4-1). By separating the storage from the SQL computation, CockroachDB can autoscale its tenant SQL processes using Kubernetes pods, giving customers the ability to quickly scale their database processing power up or down.

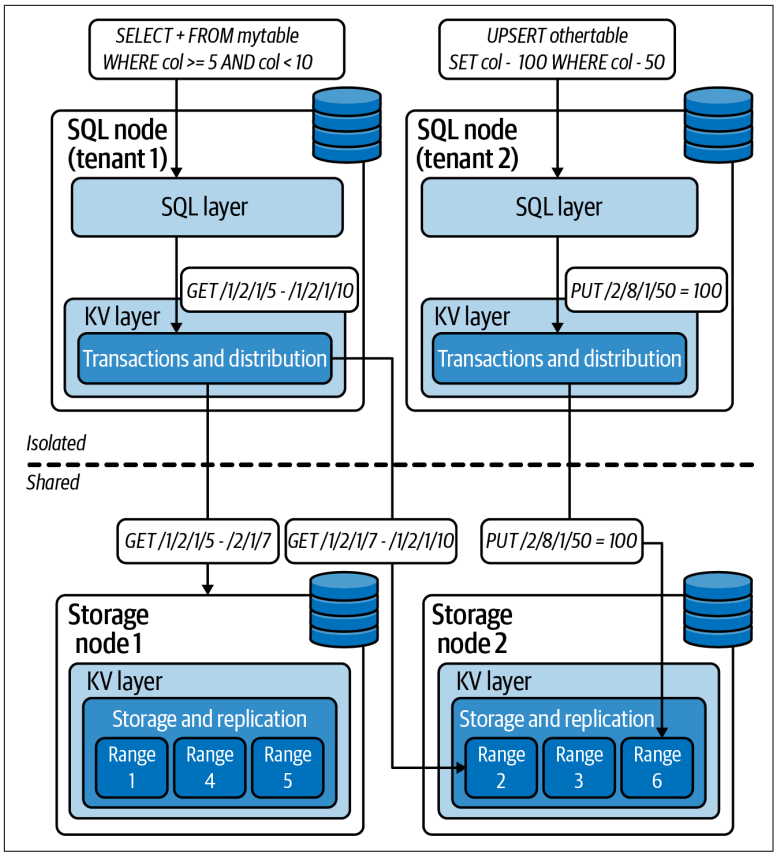

Figure 4-1. The CockroachDB approach to multitenancy with serverless distributed SQL

The Distributed Mindset: An Exhortation

We've discussed the vital parts of the distributed mindset: scale, resilience, and locality. We've also discovered the crucial feature list of distributed SQL: ease of scaling, always-on resilience, data locality, SQL, and ACID compliance. And we've reviewed the ways distributed SQL cozies up to real-world situations.

We're just at the beginning. The next five years will shake the foundations of our traditional, nondistributed databases. Here are three complementary, short ideas about the future to help you craft a vision of this future.

The Future Must Be Physical

Creating software has always been a logic-based activity: make a flowchart, diagram classes, design data relationships. Architect a proof that the application concept will work, and then apply that to a rack of servers in the nearest datacenter. With distributed SQL, you can apply the physical world to all these plans: what do you want the experience of your application to be for people in Stockholm? How do your customers in Mexico City need their regulatory data stored?

The paradigm shift looks like this: create things with an entity relationship diagram in one hand, and a globe in the other. The physical world used to be a map of limitations. Now it's a map of possibilities.

The Future Must Be Familiar

Distributed SQL owes its success to understanding where it came from. The SQL language is simply the best way to track and understand the data that runs our critical business applications. Vendors in this category understand this and have enabled many ways of retaining the language while massively updating the capabilities. Apart from keeping the SQL language itself, many vendors have made their solutions wire-compatible with PostgreSQL. This ensures that drivers and tools that already understand Postgres will be able to connect and understand distributed solutions.

Architects and leaders, when you choose distributed SQL solutions, keep this in mind: empower technical people with familiar tools. Many object-relational mapping (ORM) tools now have extensions that work with distributed solutions, including Hibernate for Java and the ORM in Django for Python. If you bring people along with you on the journey, they'll enhance their own skills and come ready to continually shift their solutions to a distributed mindset.

The Future Must Be Fantastic

The distributed mindset applied to software creation will continue to allow developers to dream and build breakthrough applications. Soon, any business can have the audacity to believe they are making something Google-sized, and they will be able to have the distributed computing tools right in front of them to go ahead and

try. It will be cheaper and easier every year to make your creation available to the world, without compromising experience.

In a distributed world, you can create something that delights hearts no matter where they beat.

About the Authors

Paul Modderman loves creating things and sharing them. His tech career has spanned web applications with technologies like .NET, Java, Python, and React to SAP solutions in ABAP, OData, and SAPUI5, to cloud technologies in Google Cloud Platform, Amazon Web Services, and Microsoft Azure. He was principal technical architect on Mindset's certified solutions CloudSimple and Analytics for BW. He's currently the chief consultant at Bowdark Consulting.

Jim Walker is a recovering developer turned product marketer and has spent his career in emerging tech. During his career, he has brought multiple products to market in a variety of fields, including data loss prevention, master data management, Hadoop, Predictive Analytics, Kubernetes, and Distributed SQL. Jim specializes in open source business models and is focused on accelerating the emergence of categories and broad developer movements. Jim is currently the VP of product marketing at Cockroach Labs.

Charles Custer is a content strategist and creator of all types, but especially: writer, editor, and motion designer/animator. He is currently a senior technical content marketer at Cockroach Labs.