CockroachDB: The Definitive Guide

Distributed Data at Scale

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jesse Seldess, Ben Darnell, and Guy Harrison

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

CockroachDB: The Definitive Guide
by Jesse Seldess and Ben Darnell

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Angela Rufino and Andy Kwan
Production Editor: Deborah Baker
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

November 2021: First Edition

Revision History for the First Edition
YYYY-MM-DD: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098100247* for release details.

978-1-098-10024-7

[FILL IN]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Chapter 1: Introduction to CockroachDB

CockroachDB is a distributed, transactional, relational, cloud-native SQL database system. That's quite a mouthful! But in short, CockroachDB leverages both the strengths of the previous generation of relational database systems - strong consistency, the power of SQL, and the relational data model - and the strengths of modern distributed cloud principles. The result is a database system that is broadly compatible with other SQL-based transactional databases but delivers much greater scalability and availability.

In this chapter, we'll review the history of Database Management Systems and discover how CockroachDB exploits technology advances of the last few decades to deliver on its ambitious goals.

A Brief History of Databases

Data storage and data processing are the "killer apps" of human civilization. Verbal language gave us an enormous advantage in co-operating as a community. Still, it was only when we developed data storage – e.g., written language – that each generation could build on the lessons of preceding generations.

The earliest written records - dating back almost 10,000 years - are agricultural accounting records. These cuneiform records, recorded on clay tablets Figure 1-1, serve the same purpose as the databases that support modern accounting systems.

Figure 1-1. Cuneiform tablet circa 3000BC[1]

Information storage technologies over thousands of years progressed only slowly. The use of cheap, portable, and reasonably durable paper media organized in libraries and cabinets represented best practice for almost a millennia.

The emergence of digital data processing has truly resulted in an information revolution. Within a single human lifespan, digital information systems have resulted in exponential growth in the velocity and volumes of information storage. Today, the vast bulk of human information is stored in digital formats, much of it within Data-Base Management Systems.

1 *https://commons.wikimedia.org/wiki/File:Cuneiform_tablet-_administrative_account_of_barley_distribution_with_cylinder_seal_impression_of_a_male_figure,_hunting_dogs,_and_boars_MET_DT847.jpg*

Pre-relational Databases

The first digital computers had negligible storage capacities and were used primarily for computation — for instance, the generation of ballistic tables, decryption of codes, and scientific calculation. However, as magnetic tape and disks became mainstream in the 1950s, it became increasingly possible to use computers to store and process volumes of information that would be unwieldy by other means.

Early applications used simple flat files for data storage. But it soon became obvious that the complexities of reliably and efficiently dealing with large amounts of data required specialized and dedicated software platforms – and these became the first Database Management Systems (DBMS).

Early DBMS systems ran within monolithic mainframe computers, which also were responsible for the application code. The applications were tightly coupled with the database management system and processed data directly using procedural language directives. By the 1970s, two models of database systems were vying for dominance - the **Network** model and the **CODASYL** standard. These models were represented by the major databases of the day **IMS** (Information Management System) and **IDMS** (Integrated Database Management System).

These systems were great advances on their predecessors but had significant drawbacks. Queries needed to be anticipated in advance of implementation, and only record-at-a-time processing was supported. Even the simplest report required programming resources to implement, and all IT departments suffered from a huge backlog of reporting requests.

The Relational Model

In 1970, IBM computer scientist Edgar Codd wrote his seminal paper "A Relational Model of Data for Large Shared Data Banks"[2]. This paper outlined what Codd saw as fundamental issues in the design of existing DBMS systems:

- Existing DBMS systems merged physical and logical representations of data in a way that often complicated requests for data and created difficulties in satisfying requests that were not anticipated during database design.
- There was no formal standard for data representation. As a mathematician, Codd was familiar with theoretical models for representing data – he believed these principles should be applied to database systems.
- Existing DBMS systems were too hard to use. Only programmers were able to retrieve data from these systems, and the process of retrieving data was need-

2 *http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf*

lessly complex. Codd felt that there needed to be an easy access method for data retrieval.

Codd's relational model described a means of logically representing data that was independent of the underlying storage mechanism. It required a *query language* that could be used to answer any question that could be satisfied by the data.

The relational model defines the fundamental building blocks of a relational database:

- **Tuples** are a set of **attribute** values. Attributes are named scalar values. A tuple might be thought of as an individual "record" or "row".
- A **relation** is a collection of distinct tuples of the same form. A relation represents a two-dimensional dataset with a fixed number of attributes and an arbitrary number of tuples. A table in a database is an example of a relation.
- **Constraints** enforce consistency and define relationships between tuples.
- Various **Operations** are defined, such as joins, projections, unions. Operations on relations always return relations. For instance, when you join two relations, the result is itself a relation.

The relational model furthermore defined a series of "normal forms" that represent reducing levels of redundancy in the model. A relation is in **third normal form** if all the data in each tuple is dependent on the entire primary key of that tuple and on no other attributes. We generally remember this by the adage, "The key, the whole key and nothing but the key (so help me, Codd)". Third normal form generally represents the starting point for the construction of an efficient and performant data model. We will come back to Third Normal Form in Chapter 5. Figure 1-2 illustrates data in Third Normal Form.

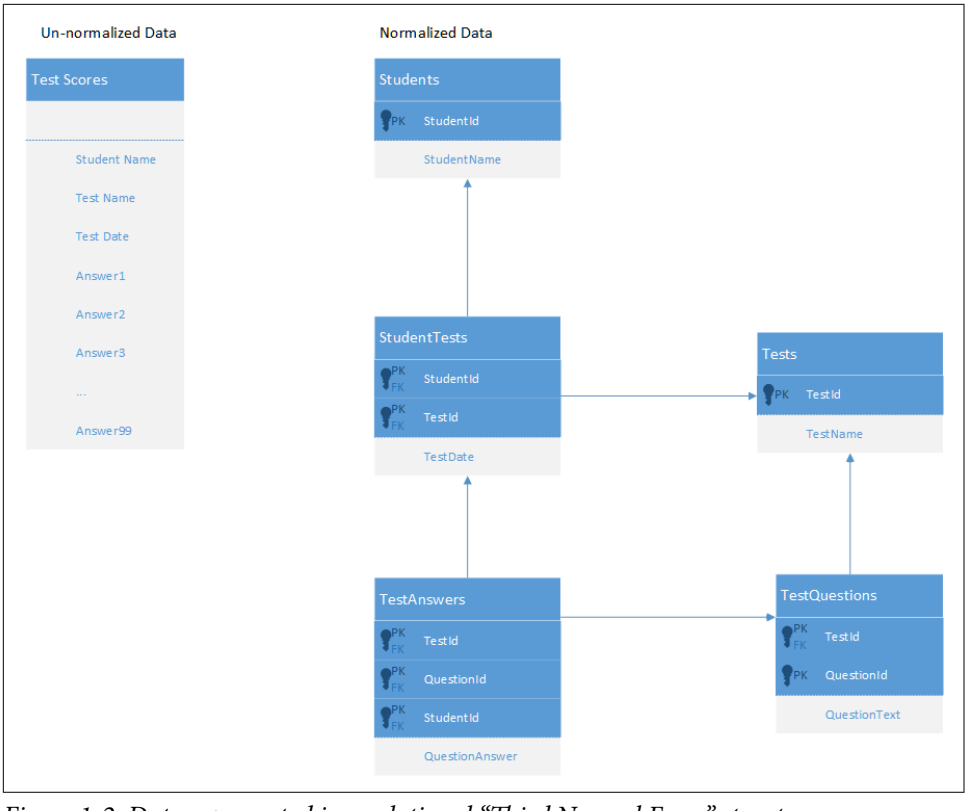

Figure 1-2. Data represented in a relational "Third Normal Form" structure.

Implementing the relational model

The relational model served as the foundation for the familiar structures present in all relational databases today. Tuples are represented by **rows** and relations as **tables**.

A table is a relation that has been given physical storage. The underlying storage may take different forms. In addition to the physical representation of the data, indexing and clustering schemes were introduced to facilitate efficient data processing and implement constraints.

Indexes and clustered storage were not an invention of the relational databases, but in relational databases, these structures were not required for data navigation; they transparently enhanced query execution rather than defining the queries that could be performed. The logical representation of the data as presented to the application was independent of the underlying physical model.

Indeed, in some relational implementations, a table might be implemented by multiple indexed structures allowing different access paths to the data.

Transactions

A transaction is a logical unit of work that must succeed or fail as a unit. Transactions predated the relational model, but in pre-relational systems transactions were often the responsibility of the application layer. In Codd's relational model, the database took formal responsibility for transactional processing. In Codd's formulation, a relational system would provide explicit support for commencing a transaction and either committing or aborting that transaction.

The use of transactions to maintain consistency in application data was also used internally to maintain consistency between the various physical structures that represented tables. For instance, when a table is represented in multiple indexes, all of those indexes must be kept synchronized in a transactional manner.

Codd's relational model did not define all the aspects of transactional behavior that became common to most relational database systems. In 1981 Jim Gray articulated the core principles of transaction processing that we still use today[3]. These principles later became known as **ACID** – Atomic, Consistent, Isolated and Durable – transactions.

As Gray put it, "A transaction is a transformation of state which has the properties of **atomicity** (all or nothing), **durability** (effects survive failures) and **consistency** (a correct transformation)." The principle of **isolation** – added shortly after - required that one transaction should not be able to see the effects of other in-progress transactions.

Perfect isolation between transactions – **serializable** isolation – creates some restrictions on concurrent data processing. Many databases adopted lower levels of isolation or allowed applications to choose from various isolation levels. These implications will be discussed further in Chapter 2.

The SQL Language

Codd had specified that a relational system should support a "Database Sublanguage" to navigate and modify relational data. He proposed the **Alpha** language in 1971, which influenced the **QUEL** language designed by the creators of Ingres – an early relational database system developed at the University of California, which influenced the open-source PostgreSQL database.

Meanwhile, researchers at IBM were developing **System R**, a prototype DBMS based on Codd's relational model. They developed the **SEQUEL** language as the data sublanguage for the project. SEQUEL eventually was renamed **SQL** and was adopted in commercial IBM databases, including IBM DB2.

3 *https://jimgray.azurewebsites.net/papers/theTransactionConcept.pdf*

Oracle chose SQL as the query language for their pioneering Oracle RDBMS, and by the end of the 1970s, SQL had won out over QUEL as the relational query language and became an ANSI standard language in 1986.

SQL needs very little introduction. Today it's one of the most widely used computer languages in the world. We will devote Chapter 4 to the CockroachDB SQL implementation.

The RDBMS hegemony

The combination of the relational model, SQL language and ACID transactions became the dominant model for new database systems from the early 1980s through to the early 2000s. These systems became known generically as Relational Database Management Systems (**RDBMS**).

The RDBMS came into prevalence at around the same time as a seismic paradigm shift in application architectures. The world of Mainframe applications was giving way to the **client-server** model. In the client-server model, application code ran on microcomputers (PCs) while the Database ran on a minicomputer, increasingly running the UNIX operating system. During the migration to client-server, mainframe-based pre-relational databases were largely abandoned in favor of the new breed of RDBMS.

By the end of the 20th century, the RDBMS reigned supreme. The leading commercial databases of the day – Oracle, Sybase, SQL Server, Informix, and DB2 competed on performance, functionality or price, but all were virtually identical in their adoption of the relational model, SQL and ACID transactions. As open-source software grew in popularity, open-source RDBMS systems such as MySQL and PostgreSQL gained significant and growing traction.

Enter the Internet

Around the turn of the century, an even more important shift in application architectures occurred. That shift was, of course, the Internet. Initially, Internet applications ran on a software stack not dissimilar to a client-server application. A single large server hosted the application's Database, while application code ran on a "middle tier" server and end-users interacted with the application through web browsers.

In the early internet, this architecture sufficed – though often just barely. The monolithic database servers were often a performance bottleneck, and although standby databases were routinely deployed, a database failure was one of the most common causes of application failure.

As the web grew, the limitations of the centralized RDBMS became untenable. The emerging "Web 2.0" social network and e-commerce sites had two characteristics that were increasingly difficult to support:

- These systems had a global or near-global scale. Users in multiple continents needed simultaneous access to the application.
- Any level of downtime was undesirable. The old model of "weekend upgrades" was no longer acceptable. There was no maintenance window that did not involve significant business disruption.

All parties agreed that the monolithic single database system would have to give way if the demands of the new breed of internet applications were to be realized. But it became recognized that a very significant and potentially immovable obstacle stood in the way: **CAP Theorem**.

CAP – or Brewer's – theorem [4] states that you can only have at most two of three desirable characteristics in a distributed system (typically illustrated as in Figure 1-3):

- **Consistency**: every user sees a consistent view of the database state.
- **Availability**: the Database remains available unless all elements of the distributed system fail.
- **Partition Tolerance**: the system runs in an environment in which a network partition might divide the distributed system in two.

4 *https://dl.acm.org/doi/10.1145/564585.564601*

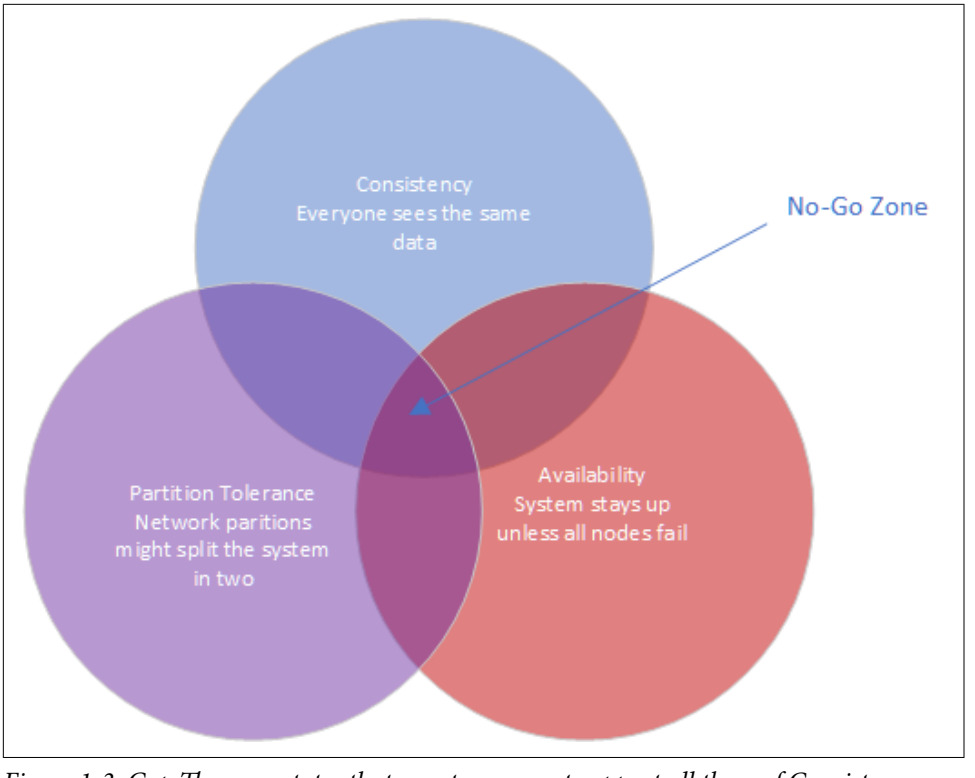

Figure 1-3. Cap Theorem states that a system cannot support all three of Consistency, Availability and Partition Tolerance

For instance, consider the case of a global e-commerce system with users in North America and Europe. If the network between the two continents fails (a network partition), then you must choose one of the following outcomes:

- Users in Europe and North America may see different versions of the Database: **sacrificing consistency**.
- One of the two regions needs to shutdown (or go read-only): **sacrificing availability.**

Clustered RDBMS systems of the day would generally sacrifice availability. For instance, in Oracle's RAC clustered Database, a network partition between nodes would cause all nodes in one of the partitions to shut down.

Internet pioneers such as Amazon, however, believed that availability was more important than strict consistency. Amazon developed a database system – **Dynamo** – that implemented "**eventual consistency**". In the event of a partition, all zones would

continue to have access to the system, but when the partition was resolved, inconsistencies would be reconciled – possibly losing data in the process.

The NoSQL movement

Between 2008-2010 dozens of new database systems emerged, all of which abandoned the three pillars of the RDBMS – the relational data model, SQL language and ACID transactions. Some of these new systems – Cassandra, Riak, Project Voldemort, HBase, for example – were directly influenced by non-relational technologies developed at Amazon and Google.

Many of these systems were essentially "schema-free" – requiring or even supporting no specific structure for the data they stored. In particular, in **key-value databases**, an arbitrary key provides programmatic access to an arbitrary structured "value". The Database knows nothing about what is in this value. From the Database's view, the value is just a set of unstructured bits. Other non-relational systems represented data in semi-tabular formats or as **JSON** (JavaScript Object Notation) documents. However, none of these new databases implemented the principles of the relational model.

These systems were initially referred to as Distributed Non-Relational Database Systems (DNRDBMS), but – because they did not include the SQL language – rapidly become known by the far catchier term "NoSQL" databases.

NoSQL was always a very questionable term. It defined what the class of systems discarded, rather than their unique distinguishing features. Nevertheless, the NoSQL term stuck, and in the following decade, "NoSQL" databases such as Cassandra, DynamoDB and MongoDB became established as a distinct and important segment of the database landscape.

The emergence of distributed SQL

The challenges of implementing distributed transactions at a web-scale, more than anything else, led to the schism in modern database management systems. With the rise of global applications with extremely high uptime requirements, it became unthinkable to sacrifice availability for perfect consistency. Almost in unison, the leading web 2.0 companies such as Amazon, Google, and Facebook introduced new database services that were only "eventually" or "weakly" consistent but globally and highly available, and the open-source community responded with databases based on these principles.

However, NoSQL databases had their own severe limitations. The SQL language was extremely widely understood and was the basis for almost all Business Intelligence tools. NoSQL databases found that they had to offer some SQL-compatibility, and so many added some SQL-like dialect – leading to the redefinition of NoSQL as "Not Only SQL". In many cases, these SQL implementations were query only and intended

only to support Business Intelligence features. In other cases, SQL-like language supported transactional processing but provided only the most limited subset of SQL functionality.

However, the problems caused by weakened consistency were harder to ignore. Consistency and correctness in data are very often non-negotiable for mission-critical applications. While in some circumstances – social media, for instance – it might be acceptable for different users to see slightly different views of the same topic, in other contexts – such as finance – any inconsistency is unacceptable. Advanced non-relational databases adopted tunable consistency and sophisticated conflict resolution algorithms to mitigate data inconsistency. However, any database that abandons strict consistency must accept scenarios in which data can be lost or corrupted during the reconciliation of network partitions or from ambiguously timed competing transactions.

Google had pioneered many of the technologies behind important open-source NoSQL systems. For instance, the Google File System and MapReduce technologies led directly to Apache Hadoop, and Google BigTable led to Apache HBase. As such, Google was well aware of the limitations of these new data stores.

The Spanner project was initiated as an attempt to build a distributed database, similar to Google's existing BigTable system, that could support both strict consistency and high availability.

Spanner benefitted from Google's highly redundant network, which reduced the probability of network-based availability issues, but the really novel feature of Spanner was its **TrueTime** system. Distributed databases go to a lot of effort to return consistent information from replicas maintained across the system. Locks are the primary mechanism to prevent inconsistent information from being created in the Database, while snapshots are the primary mechanism for returning consistent information. Queries don't see changes to data that occur while they are executing because they read from a consistent "snapshot" of data. Maintaining snapshots in distributed databases can be tricky: usually, there is a large amount of inter-node communication required to create agreement on the ordering of transactions and queries.

Google Spanner simplifies the snapshot mechanism by using GPS receivers and atomic clocks installed in each datacenter. GPS provides an externally validated timestamp while the atomic clock provides high-resolution time between GPS "fixes". The result is that every Spanner server across the world has very close to the same clock time. This allows Spanner to order transactions and queries precisely without requiring excessive inter-node communication.

Spanner is highly dependent on Google's redundant network and specialized server hardware. Spanner can't operate independently of the Google network.

The initial version of Spanner pushed the boundaries of the CAP theorem as far as technology allowed. It represented a distributed database system in which consistency was guaranteed, availability maximized, and network partitions avoided as much as possible. Over time, Google added relational features to the data model of Spanner and SQL language support. By 2017, Spanner had evolved to a distributed database that supported all three pillars of the RDBMS – the SQL language, relational data model and ACID transactions.

The Advent of CockroachDB

With Spanner, Google persuasively demonstrated the utility of a highly consistent distributed database. However, Spanner was tightly coupled to the Google Cloud platform and – at least initially – not publicly available.

There was an obvious need for the technologies pioneered by Spanner to be made more widely available. In 2015 a trio of Google alumni - Spencer Kimball, Peter Mattis, and Ben Darnell -founded Cockroach Labs with the intention of creating an open-source, geo-scalable ACID compliant database.

Spencer, Peter and Ben chose the name "CockroachDB" in honor of the humble Cockroach who, it is told, is so resilient that it would survive even a nuclear war Figure 1-4.

Figure 1-4. The original CockroachDB logo

CockroachDB design goals

CockroachDB was designed to support the following attributes:

- **Scalability**: the CockroachDB distributed architecture allows a cluster to scale seamlessly as workload increases or decreases. Nodes can be added to a cluster

without any manual rebalancing, and performance will scale predictably as the number of nodes increase.

- **High Availability**: A CockroachDB cluster has no single point of failure. CockroachDB can continue operating if a node, zone or region fails without compromising availability.
- **Consistency:** CockroachDB provides the highest practical level of transactional isolation and consistency. Transactions operate independently of each other and, once committed, transactions are guaranteed to be durable and visible to all sessions.
- **Performance**: The CockroachDB architecture is designed to support low latency and high-throughput transactional workloads. Every effort has been made to adopt Database best practices with regards to indexing, caching, and other database optimization strategies.
- **Geo-partitioning**: CockroachDB allows data to be physically located in specific localities to enhance performance for "localized" applications and to respect data sovereignty requirements.
- **Compatibility:** CockroachDB implements ANSI-standard SQL and is wire-protocol compatible with PostgreSQL. This means that the vast majority of database drivers and frameworks that work with PostgreSQL will also work with CockroachDB. Many PostgreSQL applications can be ported to CockroachDB without requiring significant coding changes.
- **Portability:** CockroachDB is offered as a fully-managed database service which in many cases is the easiest and most cost-effective deployment mode. But it's also capable of running on pretty much any platform you can imagine, from a developer's laptop to a massive cloud deployment. In particular, the CockroachDB architecture is very well aligned with containerized deployment options, and in particular with Kubernetes. CockroachDB provides a Kubernetes operator that eliminates much of the complexity involved in a Kubernetes deployment.

You may be thinking, "this thing can do everything!". However, it's worth pointing out that CockroachDB was not intended to be all things to all people. In particular:

- **CockroachDB prioritizes consistency over availability.** We saw earlier how CAP theorem states that you have to choose either Consistency or Availability when faced with a network partition. Unlike "eventually" consistent databases like DynamoDB or Cassandra, CockroachDB guarantees consistency at all costs. This means that there are circumstances in which a CockroachDB node will refuse to service requests if it is cut off from its peers. A Cassandra node in similar circumstances might accept a request even if there is a chance that the data in the request will later have to be discarded.

- **The CockroachDB architecture prioritizes transactional workloads.** CockroachDB includes the SQL constructs for issuing aggregations and the SQL 2003 Analytic "Windowing" functions, and CockroachDB is certainly capable of integrating with popular Business Intelligence tools such as Tableau. There's no specific reason why CockroachDB could not be used for analytic applications. However, the unique features of CockroachDB are targeted more at transactional workloads. For analytic-only workloads that do not require transactions, other database platforms might provide better performance.

It is important to remember that while CockroachDB was inspired by Spanner, it is in no way a "Spanner clone". The CockroachDB team has leveraged many of the Spanner team's concepts but has diverged from Spanner in several important ways.

Firstly, Spanner was designed to run on very specific hardware. Spanner nodes have access to an atomic clock and GPS device, allowing incredibly accurate timestamps. CockroachDB is designed to run well on commodity hardware and within containerized environments (such as Kubernetes) and therefore cannot rely on atomic clock synchronization. As we will see in Chapter 2, CockroachDB does rely on decent clock synchronization between nodes but is far more tolerant of clock skew than Spanner. As a result, CockroachDB can run anywhere, including any cloud provider or on-premise datacenter (and one CockroachDB cluster can even span multiple cloud environments).

Secondly, while the distributed storage engine of CockroachDB is inspired by Spanner, the SQL engine and APIs are designed to be PostgreSQL compatible. PostgreSQL is one of the most implemented RDBMS systems today and is supported by an extensive ecosystem of drivers and frameworks. The "wire protocol" of CockroachDB is completely compatible with PostgreSQL, which means that any driver that works with Postgres will work with CockroachDB. At the SQL language layer, there will always be differences between PostgreSQL and CockroachDB because of differences in the underlying storage and transaction models. But the vast majority of commonly used SQL syntax are shared between the two databases.

Thirdly, Spanner has evolved to satisfy the needs of its community and has introduced many features never envisaged by the Spanner project. Today CockroachDB is a thriving database platform whose connection to Spanner is only of historical interest.

CockroachDB Releases

The first production release of CockroachDB appeared in May 2017. This release introduced the core capabilities of the distributed transactional SQL databases, albeit with some limitations of performance and scale.

Version 2.0 – released in 2018 – included massive improvements in performance and added support for JSON data.

In 2019, CockroachDB courageously leaped from version 2 to version 19! This was not because of 17 failed versions between 2 and 19 but instead reflects a change in numbering strategy from sequential numbering to associating each major release with its release year.

Version 19 included security features such as encryption at rest and LDAP integration, the Change Data Capture facility described in chapter?? and multi-region optimizations.

2020s version 20 included enhancements to indexing and query optimization, the introduction of the fully managed CockroachDB Cloud and many relatively minor but important new features and optimizations.

(We will add something accurate about version 21 here as the book approaches final production)

CockroachDB in action

CockroachDB has gained strong and growing traction in a crowded database market. Users who have been frustrated with the scalability of traditional relational databases such as PostgreSQL and MySQL are attracted by the greater scalability of CockroachDB. Those who have been using distributed NoSQL solutions such as Cassandra are attracted by the greater transactional consistency and SQL compatibility offered by CockroachDB. And those who are transforming towards modern containerized and cloud-native architectures appreciate the cloud and container readiness of the platform.

Today, CockroachDB can boast of significant adoption at scale across multiple industries. Let's look at a few of these case studies[5]!

CockroachDB at Baidu

Beijing-headquartered Baidu is one of the largest technology companies in the world. Baidu search is the most popular Chinese language web search platform, and Baidu offer many other consumer and business-oriented internet services. Before adopting CockroachDB, the Baidu standard database platform involved sharded clusters of MySQL servers. Although single-node MySQL is a transactional SQL RDBMS, in a sharded deployment secondary indexes, transactions, joins, and other familiar DBMS constructs become enormously complex.

5 Cockroach labs maintains a growing list of CockroachDB case studies at *https://resources.cockroachlabs.com/customers*.

Baidu has implemented several new applications using CockroachDB rather than MySQL. These applications access 40TB of data with 100,000 queries per second across 20 clusters.

Compared with the sharded MySQL solution, CockroachDB reduces complexity for both application developers and database administrators. Developers no longer need to route database requests through the sharding middleware and can take advantage of distributed transactions and SQL operations. Administrators benefit from CockroachDB's automated scalability and high availability features.

Cockroach at MyWorld

MyWorld is a next-generation virtual world company. They are developing a framework to provide developers with a modern platform providing fast, scalable and extensible services for MMOGs (Massive Multiplayer Online Games) and other virtual world applications.

Initially, MyWorld employed Cassandra as the primary persistence layer. Cassandra's scalability and high-availability were a good fit for MyWorld. However, MyWorld found that Cassandra's weaker consistency model and non-relational data model were creating constraints on My World's software architecture. As founder Daniel Perano explained[6]:

> Using Cassandra was unduly influencing the model, restricting our higher-level design choices, and forcing us to maintain certain areas of data consistency at the application level instead of in the database. Some design trade-offs always have to be made in a distributed environment, but Cassandra was influencing higher-level design choices in ways a database shouldn't.
>
> —Daniel Perano

Switching to CockroachDB allowed MyWorld to model data more naturally and use multi-table transactions and constraints to maintain data consistency. CockroachDB's PostgreSQL compatibility was another benefit, allowing the company to use familiar PostgreSQL compatible drivers and development frameworks.

CockroachDB at Bose

Bose is a world-leading consumer technology company particularly well known as a leading provider of high-fidelity audio equipment.

Bose's customer base spans the globe, and Bose aims to provide those customers with best-in-class cloud-based support solutions.

6 *https://www.cockroachlabs.com/blog/cassandra-to-cockroachdb/*

Bose has embraced modern, microservices-based software architecture. The backbone of the Bose platform is Kubernetes, which allows applications to access low-level services – containerized compute – and to higher-level services such ElasticSearch, Kafka, Redis, and so on. CockroachDB became the foundation of the database platform for this Containerized Microservice platform. Aside from the resiliency and scalability of CockroachDB, CockroachDB's ability to be hosted within a Kubernetes environment was decisive.

By running CockroachDB in a Kubernetes environment, Bose has empowered Developers by providing a self-service, Database on-demand capability. Developers can spin up CockroachDB clusters for development or testing simply and quickly within a Kubernetes environment. In production, CockroachDB running with Kubernetes provides full-stack scalability, redundancy and high-availability.

Summary

In this chapter, we've placed CockroachDB in a historical context and introduced the goals and capabilities of the CockroachDB database.

The Relational Database Management Systems (RDBMS) that emerged in the 1970s and 1980s were a triumph of software engineering that powered software applications from client-server through to the early internet. But the demands of globally scalable, always available internet applications were inconsistent with the monolithic, strictly consistent RDBMS architectures of the day. Consequently, a variety of NoSQL distributed, "eventually consistent" systems emerged about ten years ago to support the needs of a new generation of internal applications.

However, while these NoSQL solutions have their advantages, they are a step backward for many or most applications. The inability to guarantee data correctness and the loss of the highly familiar and productive SQL language was a regression in many respects. CockroachDB was designed as a highly consistent and highly available SQL-based transactional database that provides a better compromise between availability and consistency.

CockroachDB is a highly available, transactionally consistent SQL database compatible with existing development frameworks and with increasingly important containerized deployment models and cloud architectures. CockroachDB has been deployed at scale across a wide range of verticals and circumstances.

In the next chapter, we'll examine the architecture of CockroachDB and see exactly how it achieves its ambitious design goals.

CHAPTER 2

Chapter 2: CockroachDB architecture

The architecture of a software system defines the high-level design decisions that enable the goals of that system. As you may recall from Chapter 1, the goals of CockroachDB are to provide a scalable, highly available, highly performant, strongly consistent, geo-distrusted, SQL-powered relational database system capable of running across a wide variety of hardware platforms. The architecture of CockroachDB is aligned to those objectives.

There are multiple ways of looking at the CockroachDB architecture. At the cluster level, a CockroachDB deployment consists of one or more shared-nothing, masterless nodes that collaborate to present a single logical view of the distributed database system. Within each node, we can observe the CockroachDB architecture as a series of layers that provide essential database services, including SQL processing, transaction processing, replication, distribution and storage.

In this chapter, we'll endeavor to give you a comprehensive overview of the CockroachDB architecture. The aim of the chapter is to provide you with the fundamental concepts that will help you make sensible decisions regarding schema design, performance optimization, cluster deployment and other topics.

The CockroachDB architecture is sophisticated: it incorporates decades of database engineering best practice designs together with several unique innovations. However, CockroachDB doesn't require that you understand its internals in order to get things done. If you are in a hurry to get started with CockroachDB, you can skip forward to the next chapter and return to this chapter later as necessary. We will, however, assume you are broadly familiar with the key concepts in this chapter when we consider advanced topics later in the book.

The CockroachDB Cluster Architecture

From a distance, a CockroachDB deployment consists of one or more database server processes. Each server has its own dedicated storage – the familiar "**shared nothing**" database cluster pattern. The nodes in a CockroachDB cluster are symmetrical – there are no "special" or "master nodes". This storage is often directly attached to the machine on which the CockroachDB server runs, though it's also possible for that data to be physically located on a shared storage subsystem.

Data is distributed across the cluster based on **key ranges**. Each range is replicated to at least three members of the cluster.

Database clients – applications, administrative consoles, the CockroachDB shell and so on – connect to a CockroachDB server within the cluster.

The communications between a database server and database client occur over the PostgreSQL **wire protocol** format. This protocol describes how SQL requests and responses are transmitted between a PostgreSQL client and a PostgreSQL server. Because CockroachDB uses the PostgreSQL wire protocol, any PostgreSQL driver can be used to communicate with a CockroachDB server.

In a more complex deployment, one or more **load balancer** processes will be responsible for ensuring that these connections are evenly and sensibly distributed across nodes. The load balancer will connect the client with one of the nodes within the cluster, which will become the **gateway server** for the connection.

The client request might involve reading and writing data to a single node or to multiple nodes within the cluster. For any given range of key values, a **Leaseholder node** will be responsible for controlling reads and writes to that range. The Leaseholder is also usually the **Raft leader**, which has the responsibility to make sure that replicas of the data are maintained correctly.

Figure 2-1 illustrates some of these concepts. A Database client connects to a Load Balancer (1) that serves as a proxy for the CockroachDB cluster. The Load Balancer directs requests to an available CockroachDB node (2). This node becomes the Gateway node for this connection. The request requires data in Range 4, so the Gateway node communicates with the Leaseholder node for this range (3), which returns data to the gateway, which in turn returns the required data to the database client (4).

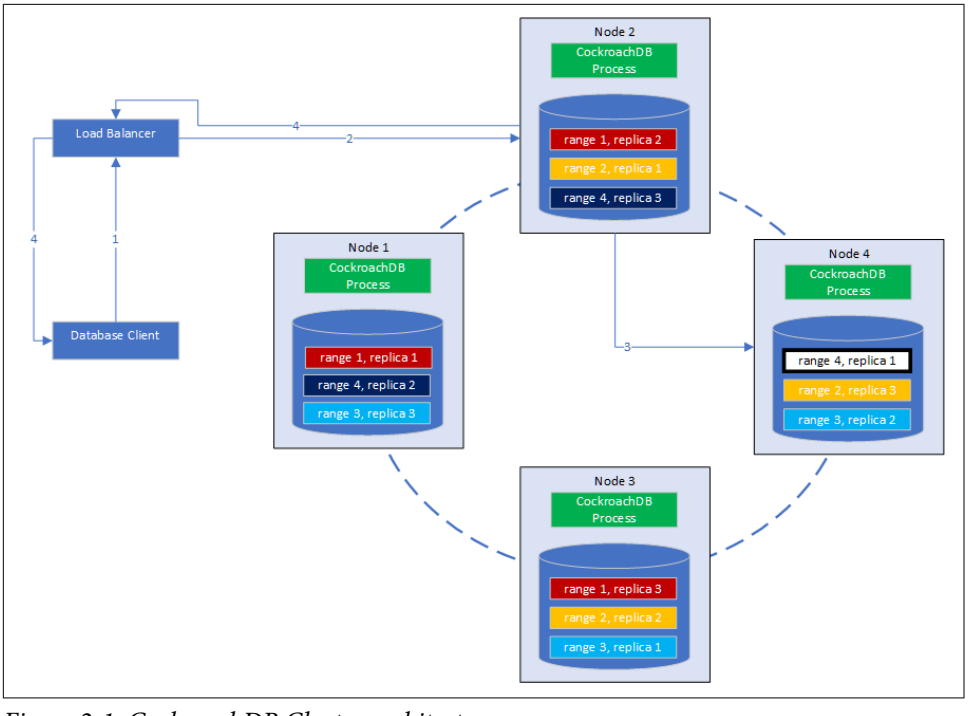

Figure 2-1. CockroachDB Cluster architecture

This architecture distributes load evenly across the nodes of the cluster. Gateway duties are distributed evenly across the nodes of the cluster by the load balancer; leaseholder duties are similarly distributed by ranges across all the nodes.

In the case of a query that requires data from multiple ranges or where data must be changed (and therefore replicated), the workflow involves more steps. We will work through a more complex example towards the end of the chapter.

Ranges and Replicas

We'll examine the nuances of CockroachDB distribution and replication later in this chapter. But for now, there are a few concepts we need to understand.

Under the hood, data in a CockroachDB table is organized in a **Key-Value** (KV) storage system. The Key for the KV store is the table's Primary Key. The Value in the KV store is a binary representation of the values for all the columns in that row.

Indexes are also stored in the KV system. In the case of a non-unique index, the Key is the index key concatenated to the table's Primary Key. In the case of a unique index, the Key is the index key, with the primary key appearing as the corresponding Value for that key.

Ranges store contiguous spans of key values. Figure 2-2 illustrates how a "dogs" table might be segmented into ranges.

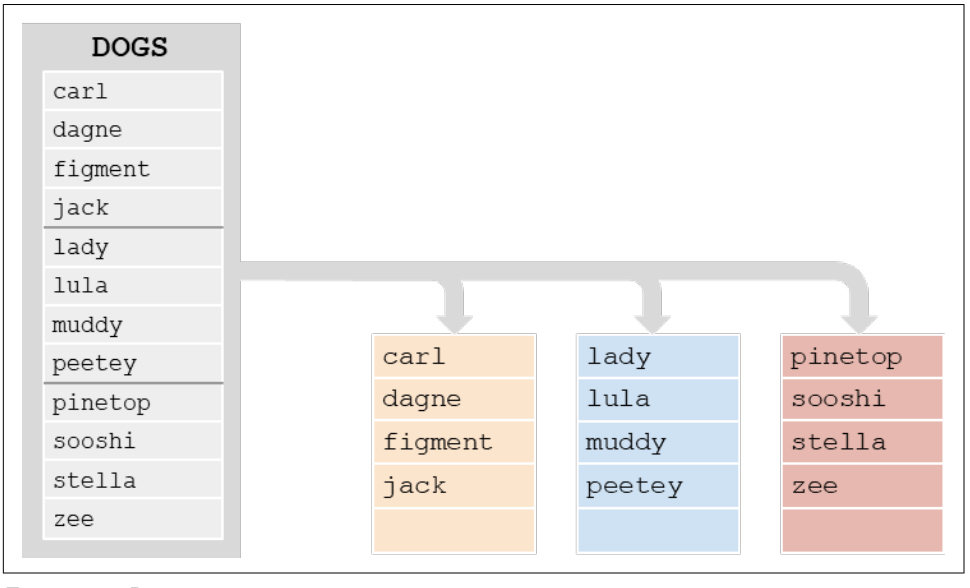

Figure 2-2. Ranges

As mentioned earlier, **Leases** are granted to a node giving it responsibility for managing reads and writes to a range. The node holding the lease is known as the **Leaseholder**. The same node is generally also the **Raft leader,** who is responsible for ensuring that replicas of the node are correctly maintained across multiple nodes.

The CockroachDB software stack

Each CockroachDB node runs a copy of the CockroachDB software, which is a single multi-threaded process. From the Operating System perspective, the CockroachDB process might seem like a black-box, but internally it is organized into multiple logical layers, as shown in Figure 2-3.

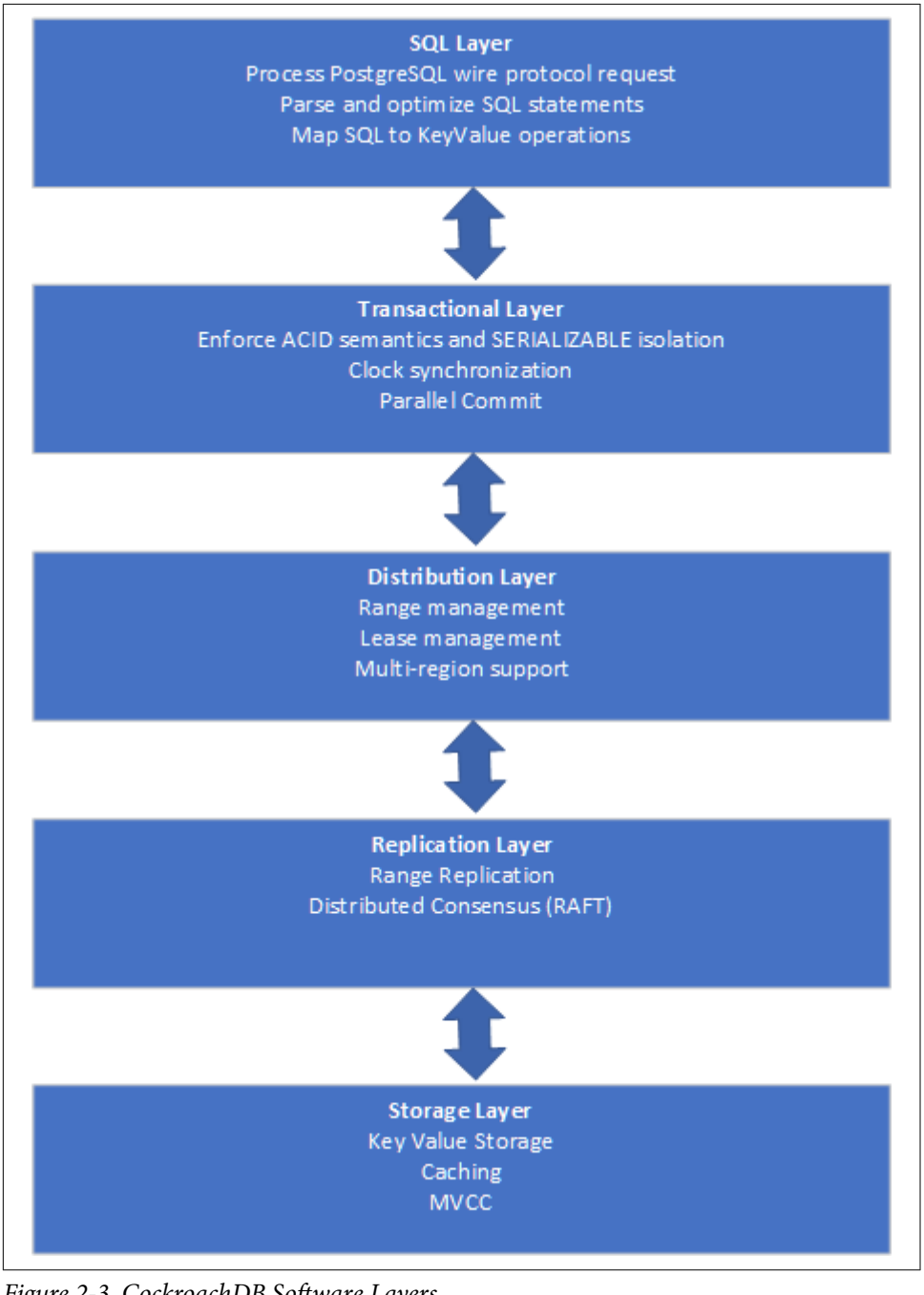

Figure 2-3. CockroachDB Software Layers

We'll discuss each of these layers in turn as we proceed through the chapter.

The CockroachDB SQL layer

The CockroachDB SQL layer is the part of the CockroachDB software stack that is responsible for handling SQL requests. Since CockroachDB is a SQL database, you would be forgiven for thinking that the SQL layer does pretty much everything. However, the core responsibility of the SQL layer is actually to turn SQL requests into requests that can be serviced by the Key-Value subsystem. Other layers handle transactions, distribution and replication of ranges and physical storage to disk.

The SQL layer receives requests from **database clients** over the **PostgreSQL wire protocol**.

A database client is any program that is using a database driver to communicate with the server and includes the CockroachDB command-line SQL processor, GUI tools such as DBEaver or Tableau or applications written in Java, Go, NodeJS, Python or any other language that has a compatible driver.

The PostgreSQL wire protocol describes the format of network packets that are used to send requests and receive results from a database client and server. The wire protocol layers on top of a transport medium such as TCP/IP or Unix-style sockets. The use of the PostgreSQL wire protocol allows CockroachDB to take advantage of the large ecosystem of compatible language drivers and tools that support the PostgreSQL database.

SQL Optimization

The SQL layer parses the SQL request, checking it for syntactical accuracy and ensuring that the connection has privileges to perform the requested task.

CockroachDB then creates an execution plan for the SQL statement and proceeds to **optimize** that plan.

SQL is a declarative language: You define the data you want, not how to get it. Although the non-procedural nature of SQL results in improvements in programmer productivity, the database server must support a set of sophisticated algorithms to determine the optimal method of executing the SQL. These algorithms are collectively referred to as **the optimizer**.

For almost all SQL statements, there will be more than one way for CockroachDB to retrieve the rows required. For instance, given a SQL with JOIN and WHERE clauses, there may be multiple join orders and multiple access paths (table scans, index lookups, etc.) available to retrieve data. It's the goal of the optimizer to determine the best access path. CockroachDB's SQL optimizer has some unique features relating to its distributed architecture, but broadly speaking, the Cost-based optimizer is similar to that found in other SQL databases such as Oracle or PostgreSQL.

The optimizer uses both heuristics – rules – and cost-based algorithms to perform its work.

The first stage of the SQL optimization process is to transform the SQL into a normalized form suitable for further optimization. This transformation removes any redundancies in the SQL statement and performs rule-based transformations to improve performance. The transformation takes into account the distribution of data for the table, adding predicates to direct parts of the queries to specific ranges or adding predicates that allow the use of indexed retrieval paths.

The optimization of the SQL statement proceeds in two stages – expansion and ranking. The SQL statement is transformed into an initial plan. Then the optimizer expands that plan into a set of equivalent candidate plans which involve alternative execution paths such as join orders or indexes.

The optimizer then ranks the plans by calculating the relative cost of each operation, leveraging statistics that supply the size and distribution of data within each table. The plan with the lowest cost is then selected.

CockroachDB also supports a **vectorized execution** engine that can speed up the processing of batches of data. This engine translates data from a row-oriented format (where sets of data contain data from the same row) to a column-oriented format (where every set of data contains information from the same column).

We'll return to the optimizer in Chapter 8 when we look in detail at SQL tuning.

From SQL to Key-Values

As we mentioned earlier, CockroachDB data ends up stored in a Key-Value storage system that is distributed across multiple nodes in ranges. We'll look at the details of this storage system towards the end of the chapter, but since the outputs of the SQL layer are in fact, Key-Value (KV) operations, the mapping of data from tables and indexes to Key-Value representation is part of the SQL layer. The output of the SQL layer are Key-Value operations.

This translation means that only the SQL layer needs to be concerned with SQL syntax – all the subsequent layers are blissfully unaware of the SQL language.

Tables as represented in the KV store

Each entry in the KV store has a Key based on the following structure:

```
/<tableID>/<indexID>/<IndexKeyValues>/<ColumnFamily>
```

We'll discuss ColumnFamilies in the next section. By default, all columns are included in a single default ColumnFamily.

For a base table, the default indexID is “primary”.

Figure 2-4 shows a simplified version of this mapping, omitting the ColumnFamily identifier.

Figure 2-4. Key-Value to column mappings

Figure 2-4 shows the table name and index name (“primary”) as text, but within the KV store, these are represented as compact table and index identifiers.

Column Families

In the above example, all the columns for a table are aggregated together in the Value section of a single KV entry. However, it’s possible to direct CockroachDB to store groups of columns in separate KV entries using **Column Families**. Each column family in a table will be allocated its own KV entry. ??? on page 34 illustrates this concept – if a table has two column families, then each row in the table will be represented by two KV entries.

image::images/Fig 02-05 Column Families.png[Column Families in the KV store]

Column Families can have a number of advantages. If infrequently accessed large columns are separated, then they will not be retrieved during row lookups which can improve the efficiency of the Key-Value store cache. Furthermore, concurrent operations on columns in separate column families will not interfere with each other.

Indexes in the KV store

Indexes are represented by a similar KV structure. For instance, the representation of a non-unique index is shown in Figure 2-5.

```
CREATE TABLE inventory (
      id INT PRIMARY KEY,
      name STRING,
      price FLOAT,
      INDEX name_idx (name)
)
```

ID	Name	Price
1	Bat	1.11
2	Ball	2.22
3	Glove	3.33
4	Bat	4.44

Key	Value
/inventory/name_idx/"Bat"/1	∅
/inventory/name_idx/"Ball"/2	∅
/inventory/name_idx/"Glove"/3	∅
/inventory/name_idx/"Bat"/4	∅

Figure 2-5. Non-unique index KV store representation

The key for a non-unique index includes the table and index name, the key value and the primary key value. For a non-unique index there is no "Value" by default.

For a unique index, the KV Value defaults to the value of the primary key. So, if `name` was unique in the `inventory` table used in previous examples, a unique index on name could be represented as shown in Figure 2-6.

```
CREATE TABLE inventory (
      id INT PRIMARY KEY,
      name STRING,
      price FLOAT,
      UNIQUE INDEX name_idx (name)
)
```

ID	Name	Price
1	Bat Acme	1.11
2	Ball	2.22
3	Glove	3.33
4	Bat Sears	4.44

Key	Value
/inventory/name_idx/"Bat Acme"	/1
/inventory/name_idx/"Ball"	/2
/inventory/name_idx/"Glove"	/3
/inventory/name_idx/"Bat Sears"	/4

Figure 2-6. Unique index KV store representation

Inverted Indexes

Inverted indexes allow indexed searches into values included in JSON documents. In this case, the key values include the JSON path and value together with the primary key - as shown in Figure 2-7.

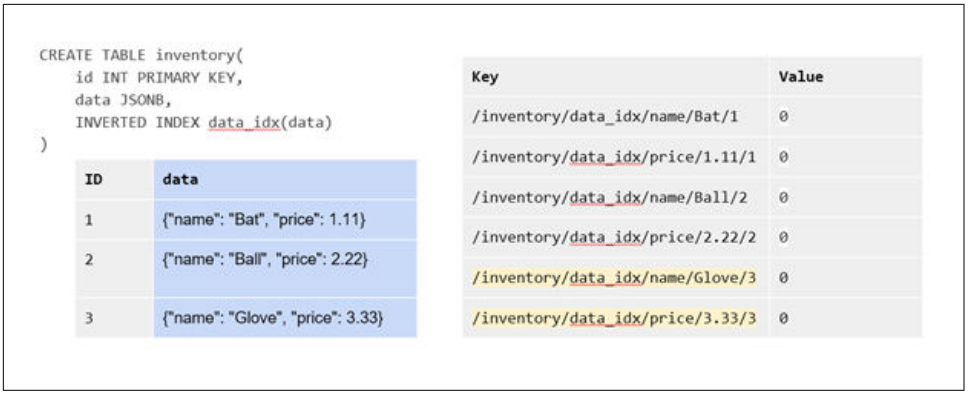

Figure 2-7. Inverted Index KV representation

Inverted indexes are also used Spatial indexes – see Chapter 8 for more details.

Inverted indexes can be larger and more expensive to maintain than other indexes since a single JSON document in a row will generate one index entry for each unique attribute. For very complex JSON documents, this might result in dozens of index entries for each document. We'll also discuss this further – and consider some alternatives - in Chapter 8.

The STORING clause

The `STORING` clause of `CREATE INDEX` allows us to add additional columns to the Value portion of the KV index structure. These additional columns can streamline a query that contains a projection that includes only those columns and the index keys. For instance, in Figure 9, we see a non-unique index on name and date of birth that uses the STORING clause to add the phone number to the KV Value. Queries that seek to find the phone number using name and date of birth can now be resolved by the index alone without reference to the base table.

Figure 2-8. STORING clause of CREATE INDEX

Table Definitions and schema changes

The schema definitions for tables (and its associated indexes) are stored in a special keyspace called a **tableDescriptor**. For performance reasons, tableDescriptors are replicated on every node. The tableDescriptor is used to parse and optimize SQL and to correctly construct Key Value operations for a table.

CockroachDB support online Schema changes using `ALTER TABLE`, `CREATE INDEX`, `TRUNCATE` and other commands. The schema is changed in discrete stages that allow the new schema to be rolled out while the previous version is still in use. Schema changes run as background tasks.

The node initiating the schema change will acquire a write lease on the relevant `table Descriptor`. Nodes which are performing DML on a table will have a lease on the relevant tableDescriptor. When node holding the write lease modifies the definition, it is broadcast to all nodes in the cluster who will – when it becomes possible - release their lease on the old schema.

The schema change may involve transactional changes to table data (removing or adding columns) and or creating new index structures. When all of the instances of the table are stored according to the requirements of the new schema, then all nodes will switch over to the new schema, and will allow reads and writes of the table using the new schema.

The CockroachDB Transactional layer

The transaction layer is responsible for maintaining the atomicity of transactions by ensuring that all operations in a transaction are committed or aborted.

Additionally, the transactional layer maintains serializable isolation between transactions – which means that transactions are completely isolated from the effects of other transactions. Although multiple transactions may be in progress at the same time, the experience of each transaction is as if the transactions were run one at a time – the **SERIALIZABLE** isolation level.

Isolation Levels

Transaction "isolation levels" define to what extent transactions are isolated from the effects of other transactions. ANSI SQL defines four isolation levels which are, from weakest to strongest: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` and `SERIALIZABLE`. Additionally, an isolation level of `SNAPSHOT` is used by many databases as an alternative "strong" isolation level.

In some databases, users may choose a lower level of isolation in order to achieve improved concurrency at the expense of consistency.

However, CockroachDB supports only the `SERIALIZABLE` level of isolation. This means that CockroachDB transactions must exhibit absolute independence from all other transactions. The results of a set of concurrent transactions must be the same as if they had all been performed one after the other.

Even `SERIALIZABLE` is arguably a compromise between performance and correctness. `LINEARIZABLE` or `STRICT SERIALIZABLE + isolation levels provide even stron ger guarantees that transactions will be sequenced in the exact order they occurred in the real world. However, in practice, +STRICT SERIALIZA BLE` isolation requires either specialized hardware (as in Spanner) or extreme limits on concurrency.

The transactional layer processes key-value operations generated by the SQL layer. A transaction consists of multiple Key-Value operations, some of which may be the result of a single SQL statement. In addition to updating table entries, index entries must also be updated. Maintaining perfect consistency under all circumstances involves multiple sophisticated algorithms, not all of which can be covered in this chapter. For deep details, you may wish to consult the CockroachDB 2020 SIGMOD paper[1] , which covers many of these principles in more detail.

1 *https://resources.cockroachlabs.com/guides/cockroachdb-the-resilient-geo-distributed-sql-database-sigmod-2020*

MVCC principles

Like most transactional database systems, CockroachDB implements the MultiVersion Concurrency Control (MVCC) pattern. MVCC allows readers to obtain a consistent view of information, even while that information is being modified. Without MVCC, consistent reads of a data item need to block simultaneous writes of that item and vice-versa. With MVCC, readers can obtain a consistent view of information even while the information is being modified by a concurrent transaction.

Figure 2-9 illustrates the basic principles of MVCC. At time t1, session s1 commences a transaction (1). At timestamp t2, s1 updates row r2 (2), creating a new version of that row (3). Also at timestamp t1, another database session s2 commences a transaction (4). When s2 attempts to read row r2 at time t2, it reads from the original version of the row - v1 (5). After both transactions commit, (5 & 6) session s2 will read from version v2 of the row (7).

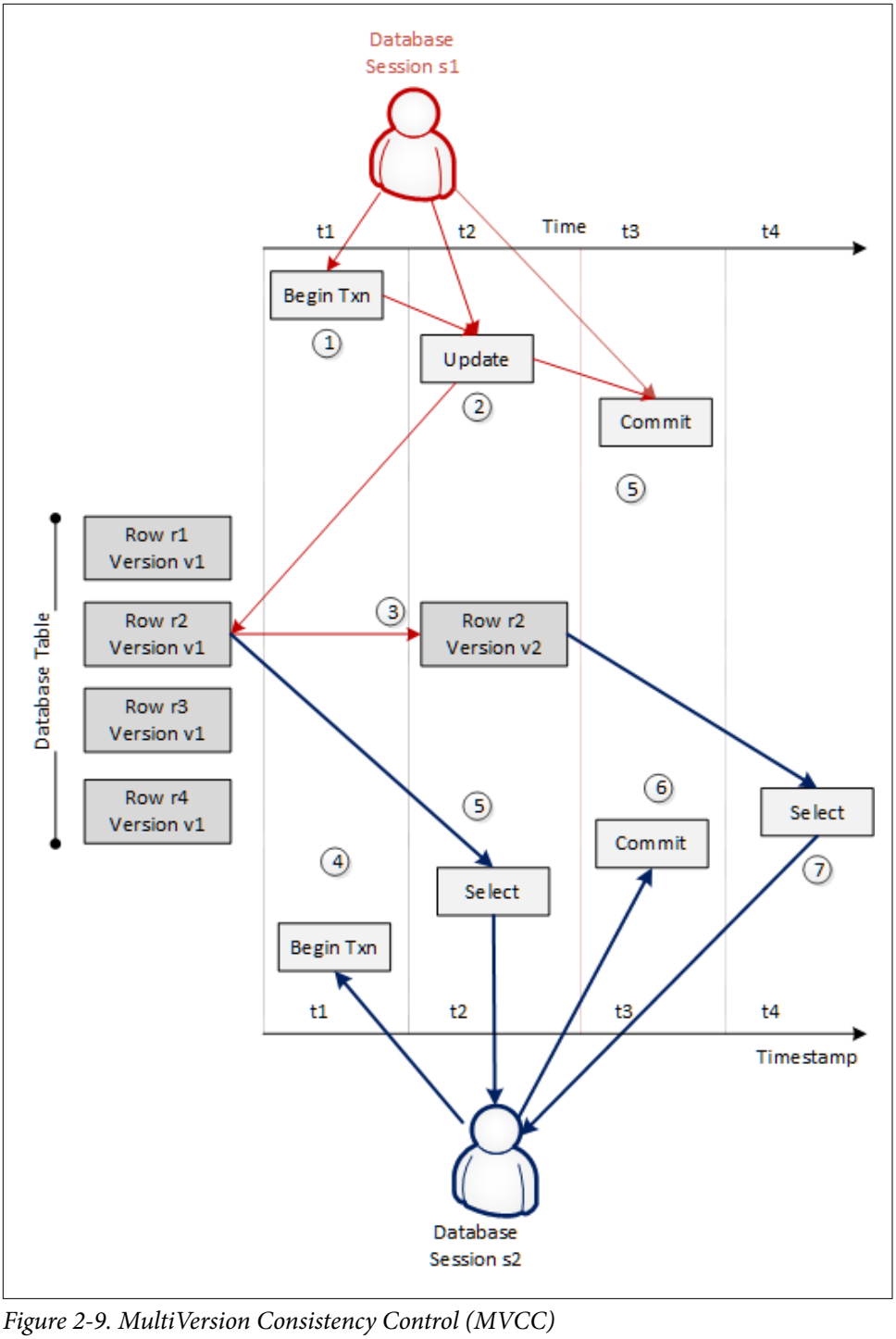

Figure 2-9. MultiVersion Consistency Control (MVCC)

The constraints of `SERIALIZABLE` isolation limit the ability of transactions to read from previous versions. For instance, if a read transaction commences after a write transaction has commenced, it may not be able to read the original version of the row because it might be inconsistent with other data already read or which will be read later in the transaction. This may result in the read transaction "blocking" until the write transaction commits or aborts.

We'll see later on how the storage engine implements MVCC, but for now, the important concept is that multiple versions of any row are maintained by the system, and transactions can determine which version of the row to read depending on their timestamp and the timestamp of any concurrent transactions.

Transaction workflow

Distributed transactions must proceed in multiple stages. Simplistically, each node in the distributed system must lay the groundwork for the transaction and only if all nodes report that the transaction can be performed will the transaction be finalized.

Figure 2-10 illustrates a highly simplified flow of a transaction preparation. In this case, a two-statement transaction is sent to the CockroachDB gateway node (1). The first statement involves a change to range 2, so that request is sent to the Leaseholder for that range, which creates a new tentative version of the range. The second statement affects range 4, so the transaction coordinator sends that request to the appropriate Leaseholder. When all changes have correctly propagated, the transaction completes, and the client is notified of success (9).

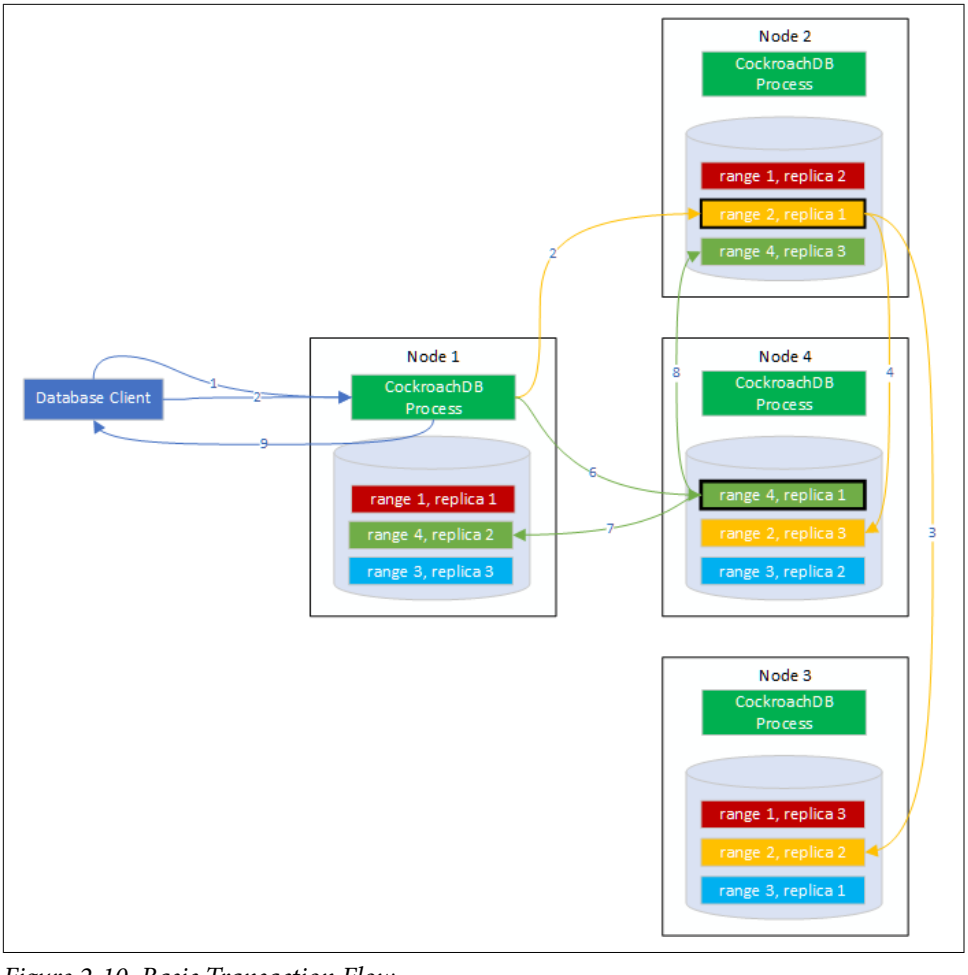

Figure 2-10. Basic Transaction Flow

Behind the scenes, these changes are propagated to replicas by the distribution layer.

Write intents

During the initial stages of transaction processing, when it is not yet known whether the transaction will succeed, the Leaseholder writes tentative modifications to modified values known as **write intents** . Write intents are specially constructed MVCC-compliant versions of the records, which are marked as provisional. They serve both as tentative transaction outcomes and as locks that prevent any concurrent attempts to update the same record.

Inside the first key range to be modified by the transaction, CockroachDB writes a special **transaction record**. This transaction record records the definitive status of the

transaction. In the example shown in Figure 2-10, this transaction record would be stored in range 2 since that is the first range to be modified in the transaction.

This transaction record will record the transaction state as one of the following:

- **PENDING**: Indicates that the write intent's transaction is still in progress.
- **STAGING**: All transaction writes have been performed, but the transaction is not yet guaranteed to commit.
- **COMMITTED**: The transaction has been successfully completed.
- **ABORTED**: Indicates that the transaction was aborted and its values should be discarded.

Parallel Commit

In a distributed database, the number of network round trips is often the dominant factor in latency. In general, committing a distributed transaction requires at least two round trips (indeed, one of the classic algorithms for this is called Two-Phase Commit). CockroachDB uses an innovative protocol called **Parallel Commits** to hide one of these round trips from the latency as perceived by the client.

The key insight behind Parallel Commits is that the gateway can return success to the client as soon as it becomes impossible for the transaction to abort, even if it is not yet fully committed. The remaining work can be done after returning as long as its outcome is certain. This is done by transitioning the transaction to the STAGING state in parallel with the transaction's last round of writes. The keys of all of these writes are recorded in the transaction record. A STAGING transaction must be committed if and only if all of those writes succeeded.

In the normal case, the gateway learns the status of these writes as soon as they complete and returns control to the client before beginning the final resolution of the transaction in the background. If the gateway fails, the next node to encounter the staging transaction record is responsible for querying the status of each write and determining whether the transaction must be committed or aborted (but because the transaction record and each write intent have been written durably, the outcome is guaranteed to be the same whether the transaction is resolved by its original gateway or by another node).

Note that any locks held by the transaction are not released until after this resolution process has been completed. Therefore, the duration of a transaction from the perspective of another transaction waiting for its locks is still at least two round trips (just as in Two-Phase Commit). However, from the point of view of the session issuing the transaction, the elapsed time is significantly reduced.

Transaction clean up

As discussed in the previous section, a COMMIT operation "flips a switch" in the transaction record to mark the transaction as committed, minimizing any delays that would otherwise occur when a transaction is committed. After the transaction has reached the COMMIT stage, then it will asynchronously resolve the write intents by modifying the write intents, which then become normal MVCC records representing the new record values.

However, as with any asynchronous operation, there may be a delay in performing this cleanup. Furthermore, since a committed write intent looks just the same as a pending write intent, transactions that encounter a write intent record when reading a key will need to determine if the write intent is committed.

If another transaction encounters a write intent that has not yet been cleaned up by the transaction coordinator, then it can perform the write intent cleanup by checking the transaction record. The write intent contains a pointer to the transaction records, which can reveal if the transaction is committed.

Overview of transaction flow

Figure 2-11 illustrates the flow of a successful two-statement transaction. A client issues a DML statement (1). This creates a transaction coordinator which maintains a transaction record in PENDING state. Write intent commands are issued to the Leaseholder for the range concerned (2). The Leaseholder writes the intent markers to its copy of the data. It returns success to the Transaction coordinator without waiting for the replica intents to be acknowledged (3).

Subsequent modifications in the transaction are processed in the same manner.

The client issues a COMMIT (3). The transaction co-coordinator marks the transaction status as STAGING. When all write intents are confirmed, the initiating client is advised of success, and then the transaction status is set to COMMITTED (4).

After a successful commit, the transaction coordinator resolves the write intents in affected ranges, which become normal MVCC records (5). At this point, the transaction has released all its locks, and other transactions on the same records are free to proceed.

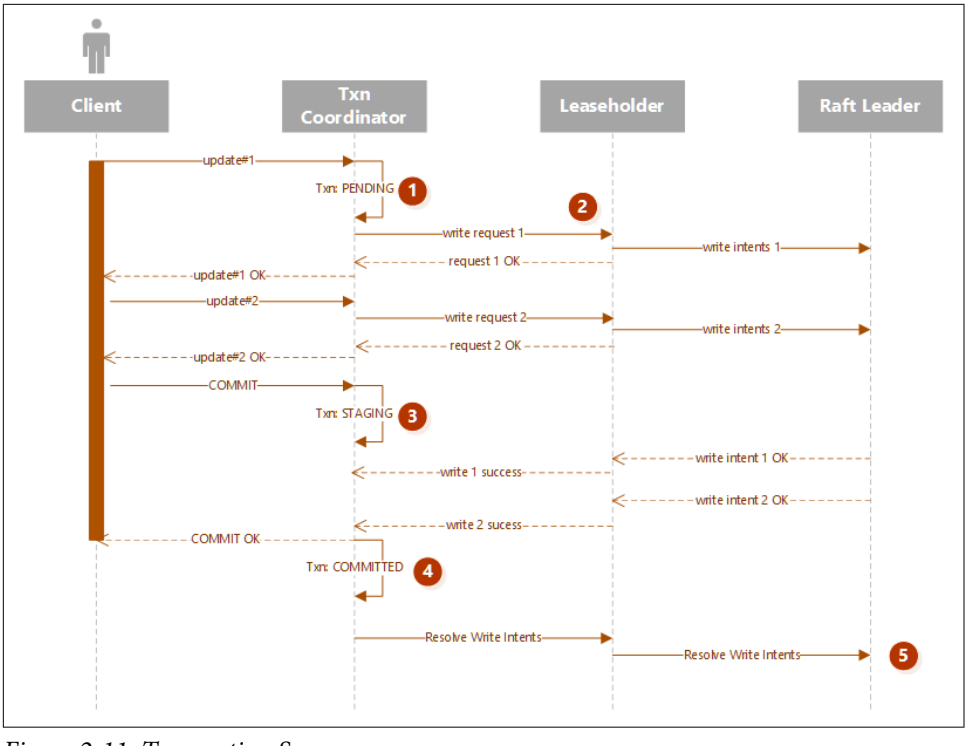

Figure 2-11. Transaction Sequence

Read/Write conflicts

So far, we've looked at the processing of successful transactions. It would be great if all transactions succeeded, but in all but the most trivial scenarios, concurrent transactions create conflicts that must be resolved.

The most obvious case is when two transactions attempt to update the same record. There cannot be two write intents active against the same Key, so either one of the transactions will wait for the other to complete, or one of the transactions will be aborted. If the transactions are of the same priority, then the second transaction – the one that has not yet created a write intent – will wait. However, if the second transaction has a high priority, then the original transaction will be aborted and will have to retry.

The **TxnWaitQueue** object tracks the transactions that are waiting and the transactions that they are waiting on. This structure is maintained within the Raft leader of the range associated with the transaction. When a transaction commits or aborts, the TxnWaitQueue is updated, and any waiting transactions are notified.

A **Deadlock** can occur if two transactions are both waiting on write intents created by the other transaction. In this case, one of the transactions will be randomly aborted.

Transaction conflicts can also occur between readers and writers. If a reader encounters an uncommitted write intent that has a lower timestamp than the consistent read timestamp for the read, then a consistent read cannot be completed. This can occur if a modification occurs between the time a read transaction starts and the time it attempts to read the key concerned. In this case, the read will need to wait until the write either commits or aborts. However, if the read has a high priority, CockroachDB may "push" the lower-priority write's timestamp to a higher value, allowing the read to complete. The "pushed" transaction may need to restart if the push invalidates any previous work in the transaction.

Many transaction conflicts are managed automatically, and while these have performance implications, they don't impact functionality or code design. However, there are multiple scenarios in which an application may need to handle an aborted transaction. We'll look at these scenarios and discuss best practices for transaction retries in Chapter 6.

Clock synchronization and clock skew

You may have noticed in previous sections that CockroachDB must compare timestamps of operations frequently to determine if a transaction is in conflict. Simplistically, we might imagine that every node in the system can agree on the time of each operation and make these comparisons easily. In reality, every system is likely to have a slightly different system clock time, and this discrepancy is likely to be greater the more geographically distributed a system is. The difference in clock times is referred to as **clock skew**. Consequently, in widely distributed systems with very high transaction rates, getting nodes to agree on the exact sequence of transactions is problematic.

As you might remember, Spanner attacked this problem by using specialized hardware – atomic clocks and GPS - to reduce the inconsistency between system clocks. As a result, Spanner can keep the clock skew within 7ms and simply adds a 7ms sleep to every transaction to ensure that no transactions complete out of order.

Since CockroachDB must run reliably on generic hardware, it synchronizes time using the venerable and ubiquitous internet Network Time Protocol (NTP). NTP produces accurate timestamps but nowhere near as accurate as Spanners GPS and atomic clocks.

By default, CockroachDB will tolerate a clock skew as high as 500ms. Adding half a second to every transaction in the Spanner manner would be untenable, so CockroachDB takes a different approach for dealing with transactions that appear within the 500ms uncertainty interval. Put simply, while Spanner always waits after writes, CockroachDB sometimes retries reads.

If a reader can't say for certain whether a value being read was committed before the read transaction started, then it pushes its own provisional timestamp just above the timestamp of the uncertain value. Transactions reading constantly updated data from many nodes may be forced to restart multiple times, though never for longer than the uncertainty interval, nor more than once per node.

The CockroachDB time synchronization strategy allows CockroachDB to deliver true `SERIALIZABLE` consistency. However, there are still some anomalies that can occur. Two transactions that operate on unrelated key values that still have some real-world sequencing dependency might appear to be committed in reverse order – the **causal reverse** anomaly. This is not a violation of `SERIALIZABLE` isolation because the transactions are not actually logically dependent. Nevertheless, it is possible in CockroachDB for transactions to have timestamps that do not reflect their real-world ordering.

The distribution layer

Logically, a table is represented in CockroachDB as a monolithic Key-Value structure, in which the Key is a concatenation of the primary keys of the table, and the value is a concatenation of all of the remaining columns in the table. We introduced this structure back in Figure 2-2.

The distribution layer breaks this monolithic structure into contiguous chunks of approximately 512MB. The 512MB chunk size is sized so as to keep the number of ranges per node manageable.

Meta Ranges

The distribution of ranges is stored in global keyspaces `meta1` and `meta2`. `meta1` can be thought of as a "range of ranges" lookup, which then allows a node to find the location of the node holding the `meta2` record, which in turn points to the nodes holding copies of every range within the "range of ranges". Figure 2-12 illustrates this two-level lookup structure.

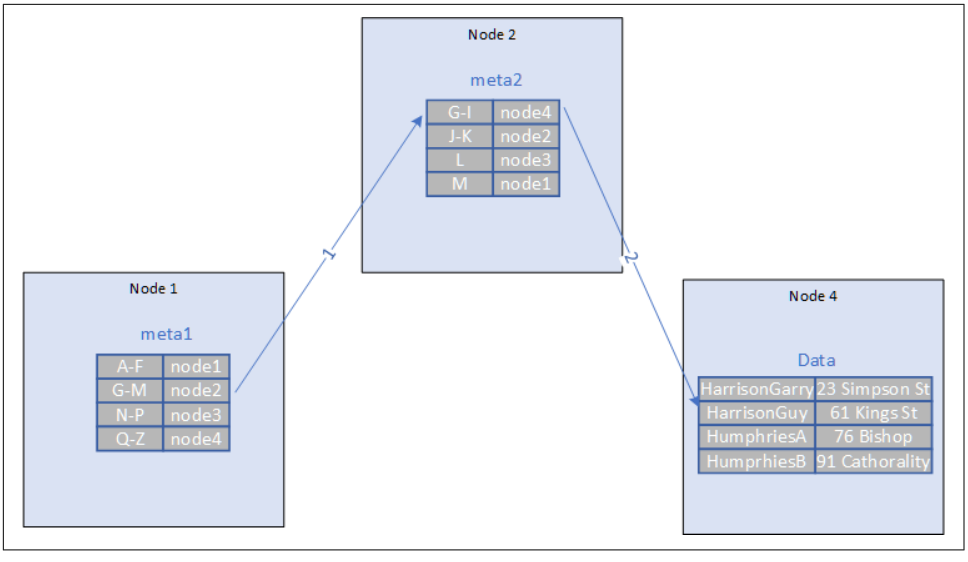

Figure 2-12. Meta Ranges

In Figure 2-12, Node1 needs to get data for the key "HarrisonGuy". It looks in its copy of `meta1`, which tells it that node2 contains the `meta2` information for the range G-M. It accesses the `meta2` data concerned from node2, which indicates that node4 is the Leaseholder for the range G-I, and therefore the Leaseholder for the range concerned.

Gossip

CockroachDB uses the Gossip protocol to share ephemeral information between nodes. Gossip is a widely used protocol in distributed systems in which nodes propagate information virally through the network.

Gossip maintains an eventually consistent key-value map maintained on all the CockroachDB nodes. It is used primarily for bootstrapping: it contains a "meta0" record that tells the cluster where the meta1 range can be found, as well as mappings from the node IDs stored in meta records to network addresses. Gossip is also used for certain operations that do not require strong consistency, such as maintaining information about the available storage space on each node for rebalancing purposes.

Leaseholders

The Leaseholder is the CockroachDB node responsible for serving reads and coordinating writes for a specific range of keys. We discussed some of the responsibilities of the Leaseholder in the transaction section. When a transaction coordinator or gateway node wants to initiate a read or write against a range, it finds that range's Lease-

holder (using the meta ranges structure discussed in the previous section) and forwards the request to the Leaseholder.

Leaseholders are assigned using the Raft protocol, which we will discuss in the Replication layer section below.

Range Splits

CockroachDB will attempt to keep a range at less than 512MB. When a range exceeds that size, the range will be split into two smaller contiguous ranges.

Ranges can also be split if they exceed a load threshold. If the parameter `kv.range_split.by_load_enabled` is true and the number of queries per second to range exceeds the value of `kv.range_split.load_qps_threshold`, then a range may be split even if it is below the normal size threshold for range splitting. Other factors will determine if a split actually occurs, including whether the resulting split would actually split the load between the two new ranges and the impact on queries that might now have to span the new ranges.

When splitting based on load, the two new ranges might not be of equal sizes. By default, the range will be split at the point at which the load on the two new ranges will be roughly equal.

Ranges can also be split manually using the `SPLIT AT` clause of the `ALTER TABLE` and +ALTER INDEX_ statements.

Figure 2-13 illustrates a basic range split when an insert causes a range to exceed the 512MB threshold. Two ranges are created as a consequence.

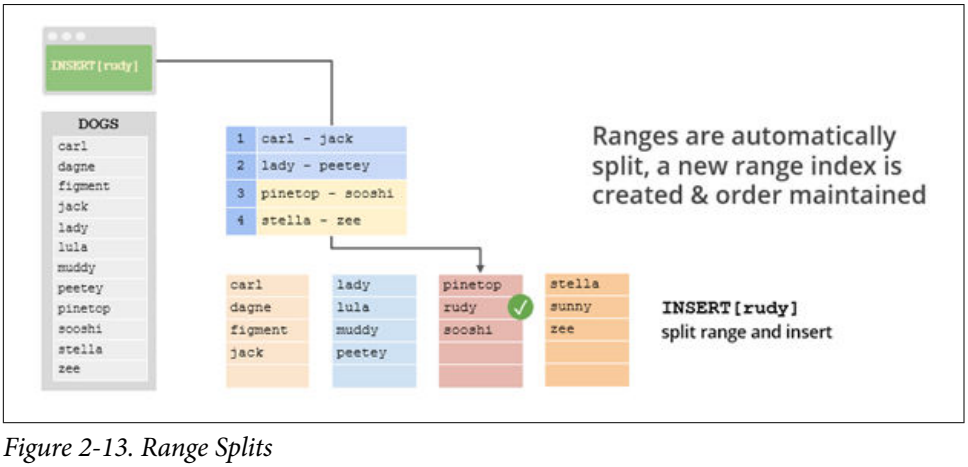

Figure 2-13. Range Splits

Multi-region distribution

Geo-partitioning is a special feature of CockroachDB Enterprise that allows data to be located within a specific geographic region. This might be desirable from a performance point of view – reducing latencies for queries from a region about that region – or from a data sovereignty perspective – keeping data within a specific geographic region for legal or regulatory reasons.

CockroachDB supports a multi-region configuration that controls how data should be distributed across regions. The following core concepts are relevant:

- **Cluster Regions** are geographic regions that a user specifies at node start time.
- **Regions** may have multiple zones
- Databases within the cluster are assigned to one or more regions: one of these regions is the **primary** region.
- Tables within a database may have specific **locality rules** (global, regional by table, regional by row), which determine how its data will be distributed across zones.
- **Survival Goals** dictate how many simultaneous failures a database can survive.

With the **zone-level survival goal**, the database will remain fully available for reads and writes, even if a zone goes down. However, the database may not remain fully available if multiple zones fail in the same region. Surviving zone failures is the default setting for multi-region databases.

The **region-level survival goal** has the property that the database will remain fully available for reads and writes, even if an entire region goes down. This, of course, means that copies of data will need to be maintained in other regions, magnifying write time.

By default, all tables in a multi-region database are **regional tables** — that is, CockroachDB optimizes access to the table's data from a single region (by default, the database's primary region).

Regional by row tables provide low-latency reads and writes for one or more rows of a table from a single region. Different rows in the table can be optimized for access from different regions.

Global tables are optimized for low-latency reads from all regions.

Replication layer

High availability requires that data not be lost or made unavailable should a node fail. This, of course, requires that multiple copies of data be maintained.

The two most commonly used high availability designs are:

- **Active-passive**, in which a single node is a "master" or "active" node whose changes are propagated to passive "secondary" or "slave" nodes.
- **Active-active** in which all nodes run identical services. Typically, active-active database systems are of the eventually consistent variety. Since there is no "master", conflicting updates can be processed by different nodes. These will need to be resolved, possibly by discarding one of the conflicting updates.

CockroachDB implements a **distributed consensus** mechanism that is called Multi-active. Like Active-active, all replicas can handle traffic, but for an update to be accepted, it must be confirmed by a majority of replicas.

This architecture ensures that there is no data loss in the event of a node failure, and the system remains available, providing at least a majority of nodes remain active.

CockroachDB implements replication at the range level: each range is replicated independently of other ranges. At any given moment, a single node is responsible for changes to a single node, but there is no overall master within the cluster.

Raft

CockroachDB employs the widely used **Raft protocol**[2] as its distributed consensus mechanism. In CockroachDB, each range is a distinct Raft group – the consensus for each range is determined independently of other ranges.

In Raft and in most distributed consensus mechanisms, we need a minimum of 3 nodes. This is because a majority of nodes (a quorum) must always agree on the state. In the event of a network partition, the only side of the partition with the majority of nodes can continue.

In a Raft group, one of the nodes is elected as **leader** by a majority of nodes in the group. The other nodes are known as **followers**. The Raft leader controls changes to the raft group.

Changes sent to the Raft leader are written to its **Raft log** and propagated to the followers. When a majority of nodes accept the change, then the change is committed by the leader. Note that in CockroachDB, each range has its own Raft log since every range is replicated separately.

Leader elections occur regularly or may be triggered when a node fails to receive a heartbeat message from the leader. In the latter case, a follower who cannot communicate with the leader will declare itself a candidate and initiate an election. Raft

2 *https://en.wikipedia.org/wiki/Raft_(algorithm)*

includes a set of safety rules that prevent any data loss during the election process. In particular, a candidate cannot win an election unless its log contains all committed entries.

Nodes that are temporarily disconnected from the cluster can be sent relevant sections of the Raft log to re-synchronize or – if necessary – a point in time snapshot of the state followed by a catch-up via Raft logs.

Raft and Leaseholders

The CockroachDB Leaseholder and the Raft leader responsibilities serve very similar purposes. The Leaseholder controls access to a range for the purposes of transactional integrity and isolation, while the Raft Leaseholder controls access to a range for the purposes of replication and data safety.

The Leaseholder is the only node that can propose writes to the Raft leader. CockroachDB will attempt to elect a Leaseholder who is also the Raft leader so that these communications can be streamlined. The Leaseholder serves all writes and most reads, so it is able to maintain the in-memory data structures necessary to mediate read/write conflicts for the transaction layer.

Closed timestamps and follower reads

Periodically the Leaseholder will "close" a timestamp in the recent past, which guarantees that no new writes with lower timestamps will be accepted.

This mechanism also allows for **follower reads.** Normally, reads have to be serviced by a replica's Leaseholder. This can be slow since the Leaseholder may be geographically distant from the gateway node that is issuing the query. A follower read is a read taken from the closest replica, regardless of the replica's leaseholder status. This can result in much better latency in geo-distributed, multi-region deployments.

If the cluster setting + kv.closed_timestamp.follower_reads_enabled+ is `TRUE`, and a query uses the `AS OF SYSTEM TIME` clause, then the gatekeeper forwards the request to the closest node that contains a replica of the data–– whether it be a follower or the Leaseholder. The timestamp provided in the query (i.e., the AS OF SYSTEM TIME value) must be less or equal to the node's closed timestamp.

The Storage layer

We touched upon the logical structure of the Key-Value store earlier in the chapter when we discussed the Key-Value store. However, we have not yet looked at the physical implementation of the Key-Value storage engine.

As of CockroachDB version 20, CockroachDB uses the **PebbleDB** storage engine – an open-source Key-Value store based inspired by the LevelDB and RocksDB storage

engines. PebbleDB is primarily maintained by the CockroachDB team and is optimized specifically for CockroachDB use cases. Older versions of CockroachDB use the RocksDB storage engine.

Log Structured Merge (LSM) Trees

PebbleDB implements the Log Structured Merge Tree (LSM) architecture.

LSM is a structure that seeks to optimize storage and support extremely high insert rates, while still supporting efficient random read access.

The simplest possible LSM tree consists of two indexed "trees":

- An in-memory tree that is the recipient of all new record inserts - the **MemTable**.
- A number of on-disk trees representing copies of in–memory trees that have been flushed to disk. These are referred to as **Static Sorted Tables (SSTables).**

SSTables exist at multiple levels, numbered L0 to L6 (L6 is also called the base level). L0 contains an unordered set of SSTables, each of which is simply a copy of an in-memory MemTable that has been flushed to disk. Periodically, SSTables are periodically compacted into larger consolidated stores in the lower levels. In levels other than L0, SSTables are ordered and non-overlapping so that only one SSTable per level could possibly hold a given key.

SSTables are internally sorted and indexed, so lookups within an SSTable are fast.

The basic LSM architecture ensures that writes are always fast since they primarily operate at memory speed, although there is often also a sequential Write Ahead Log on disk. The transfer to on disk SSTables is also fast since it occurs in append-only batches using fast sequential writes. Reads occur either from the in-memory tree or from the disk tree; in either case, reads are facilitated by an index and are relatively swift.

Of course, if a node fails while data is in the in-memory store, then it could be lost. For this reason, database implementations of the LSM pattern include a **Write Ahead Log** (WAL) that persists transactions to disk. The WAL is written via fast sequential writes.

Figure 2-14 illustrates LSM writes. Writes from higher CockroachDB layers are first applied to the Write Ahead Log (WAL) (1) and then to the MemTable (2). Once the MemTable reaches a certain size, it is flushed to disk to create a new SSTable (3). Once the flush completes, WAL records may be purged (4). Periodically multiple SSTables are merged (compacted) into larger SSTables (5).

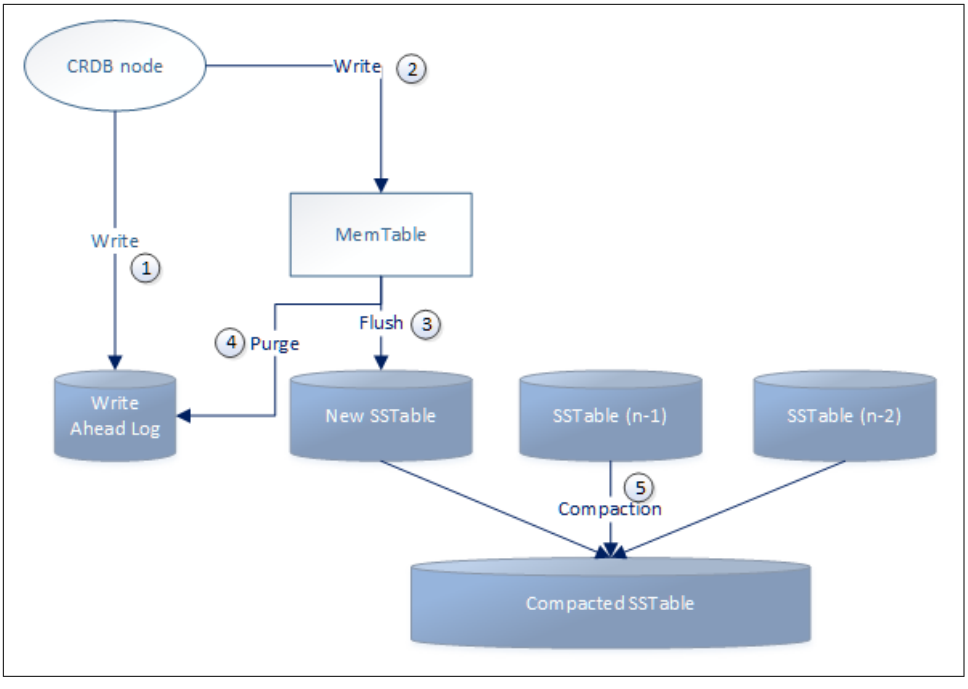

Figure 2-14. LSM Writes

SSTables and Bloom Filters

Each SSTable is indexed. However, there may be many SSTables on disk, and this creates a multiplier effect on index lookups since we might theoretically have to examine every index for every SSTable in order to find our desired row.

To reduce the overhead of multiple index lookups, **Bloom filters** are used to reduce the number of lookups that must be performed. A Bloom filter is a very compact and quick to maintain structure that can quickly tell you if a given SSTable "might" contain a value. CockroachDB uses Bloom filters to quickly determine which SSTables have a version of a key. Bloom filters are compact enough to fit in memory and are very quick to navigate. However, to achieve this compression, bloom filters are "fuzzy" and may return false positives. If you get a positive result from a bloom filter, it means only that the file *may* contain the value. However, the bloom filter will never incorrectly advise you that a value is not present. So, if a bloom filter tells us that a key is not included in a specific SSTable, then we can safely omit that SSTable from our look-up.

Figure 2-15 shows the read pattern for an LSM. A database request first reads from the MemTable (1). If the required value is not found it will consult the Bloom filters for all SSTables in L0 (2). If the bloom filter indicates that no matching value is

present, it will examine the SSTable in each subsequent level which covers the given key (3). If the Bloom filter indicates a matching key value may be present in the SSTable, then the process will use the SSTable index (4) to search for the value within the SSTable (5). Once a matching value is found, no older SSTables need be examined.

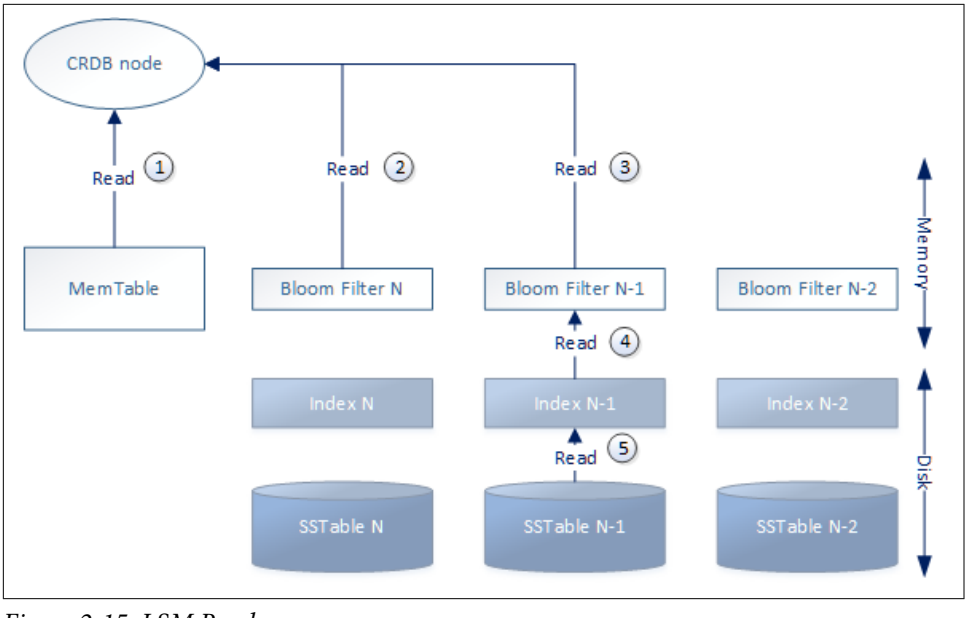

Figure 2-15. LSM Reads

Deletes and updates

SSTables are immutable - once the MemTable is flushed to disk and becomes an SSTable, no further modifications to the SSTable can be performed. If a value is modified repeatedly over a period of time, the modifications will build up across multiple SSTables. When retrieving a value, the system will read SSTables from youngest to oldest to find the most recent value for a key. Therefore, to update a value we only need to insert the new value, since the older values will not be examined when a newer version exists.

Deletions are implemented by writing tombstone markers into the MemTable, which eventually propagate to SSTables. Once a tombstone marker for a row is encountered, the system stops examining older entries and reports “not found” to the application.

As SSTables multiply, read performance and storage will degrade as the number of bloom filters, indexes and obsolete values increases. During compaction, rows that are fragmented across multiple SSTables will be consolidated and deleted rows removed. Tombstones are retained until they are compacted to the base level L6.

MultiVersion Concurrency Control

We introduced MVCC as a logical element of the transaction layer earlier in the chapter – see, for instance, Figure 10.

CockroachDB encodes the MVCC timestamp into the each key so that multiple MVCC versions of a key are stored as distinct keys within PebbleDB. However, the Bloom filters which we introduced above exclude the MVCC timestamp so that a query does not need to know the exact timestamp in order to lookup a record.

CockroachDB removes records older than the configuration variable `gc.ttlseconds`, but will not remove any records covered by **protected timestamps**. Protected timestamps are created by long-running jobs such as backups which need to be able to obtain a consistent view of data.

The Block cache

PebbleDB implements a block cache providing fast access to frequently accessed data items. This block cache is separate from the in-memory indexes, bloom filters and MemTables. The block cache operates on a Least Recently Used (LRU) basis – when a new data entry is added to the cache, the entry that was least recently access will be evicted from the cache.

Reading from the blockchain bypasses the need to scan multiple SSTables and associated bloom filters. We'll speak more about the cache in chapter 14 when we discuss cluster optimization.

Summary

In this chapter, we've tried to give you an overview of the essential architectural elements of CockroachDB.

Although having a strong grasp of the CockroachDB architecture is advantageous when performing advanced systems optimization or configuration, it's by no means a pre-requisite for working with a CockroachDB system. CockroachDB includes many sophisticated design elements, but its internal complexity is not reflected in its user interface – you can happily develop a CockroachDB application without mastering the architectural concepts in this chapter.

At a cluster level, a CRDB deployment consists of three or more symmetrical nodes, each of which carries a complete copy of the CRDB software stack and each of which can service any database client requests. Data in a CRDB table is broken up into ranges of 512MB in size and distributed across the nodes of the cluster. Each range is replicated at least three times.

The CRDB software stack consists of five major layers:

- The SQL layer accepts SQL requests in the PostgreSQL wire protocol. It parses and optimizes the SQL requests and translates the requests into Key-Value operations that can be processed by lower layers.
- The Transactional layer is responsible for ensuring ACID transactions and SERIALIZABLE isolation. It ensures that transactions see a consistent view of data and that modifications occur as if they had been executed one at a time.
- The distribution layer is responsible for the partitioning of data into ranges and the distribution of those ranges across the cluster. It is responsible for managing Range leases and assigning Leaseholders.
- The Replication layer ensures that data is correctly replicated across the cluster to allow high availability in the event of a node failure. It implements a distributed consensus mechanism to ensure that all nodes agree on the current state of any data item.
- The storage layer is responsible for the persistence of data to local disk and the processing of low-level queries and updates on that data.

In the next chapter, we'll gleefully abandon the complexities and sophisticated CockroachDB architecture and focus on the far simpler task of getting started with the CockroachDB system.

About the Authors

Jesse Seldess is the VP of Education at Cockroach Labs, where he leads the documentation and training teams. He has nearly 20 years of experience in technical documentation, and has built teams from the ground up at Cockroach Labs and AppNexus (now Xander).

Ben Darnell is the cofounder and chief architect at Cockroach Labss, where he built the distributed consensus protocols that underpin CockroachDB's transactional model. He started his career at Google and then went on to a series of startups where he saw firsthand the need for better scalable storage systems.

Guy Harrison Guy Harrison is CTO at ProvenDB.com, a partner at Toba Capital and a software professional with more than 20 years' experience in database design, development, administration, and optimization. He is author of "Next Generation Databases", "Oracle Performance Survival Guide", "MySQL Stored Procedure programming" as well as many other books and articles on database technology