CockroachDB: The Definitive Guide

Distributed Data at Scale

Jesse Seldess, Ben Darnell, and Guy Harrison

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

CockroachDB: The Definitive Guide
by Jesse Seldess and Ben Darnell and Guy Harrison

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

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

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 DataBase 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.

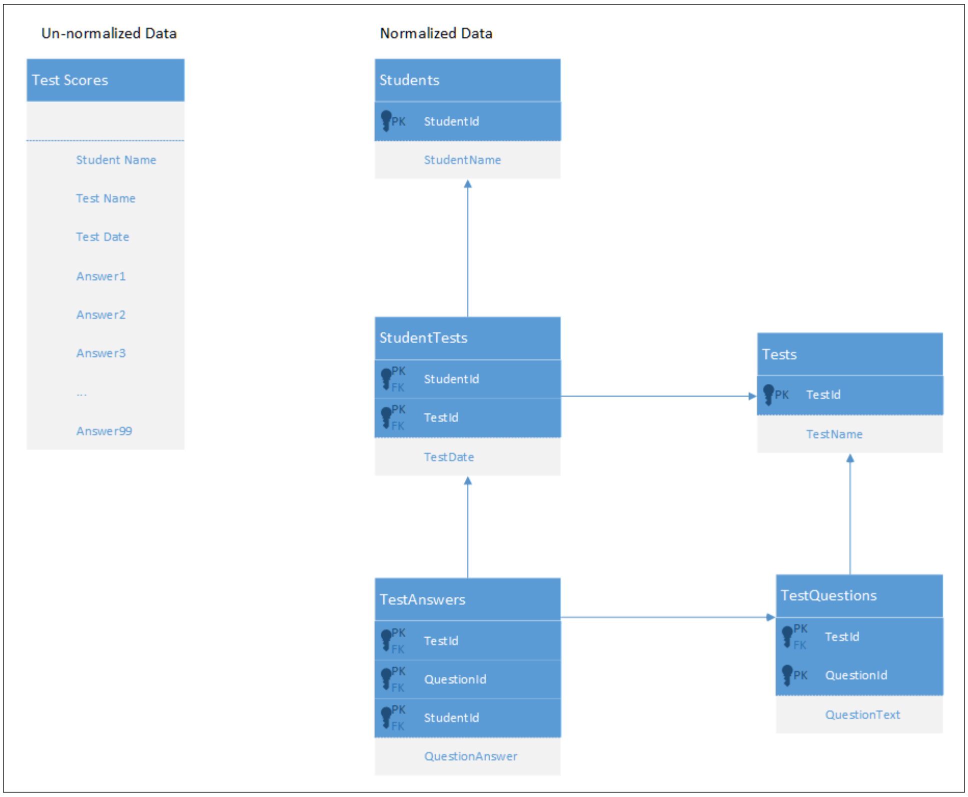


Figure 1-2. Data represented in a relational "Third Normal Form" structure.

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*

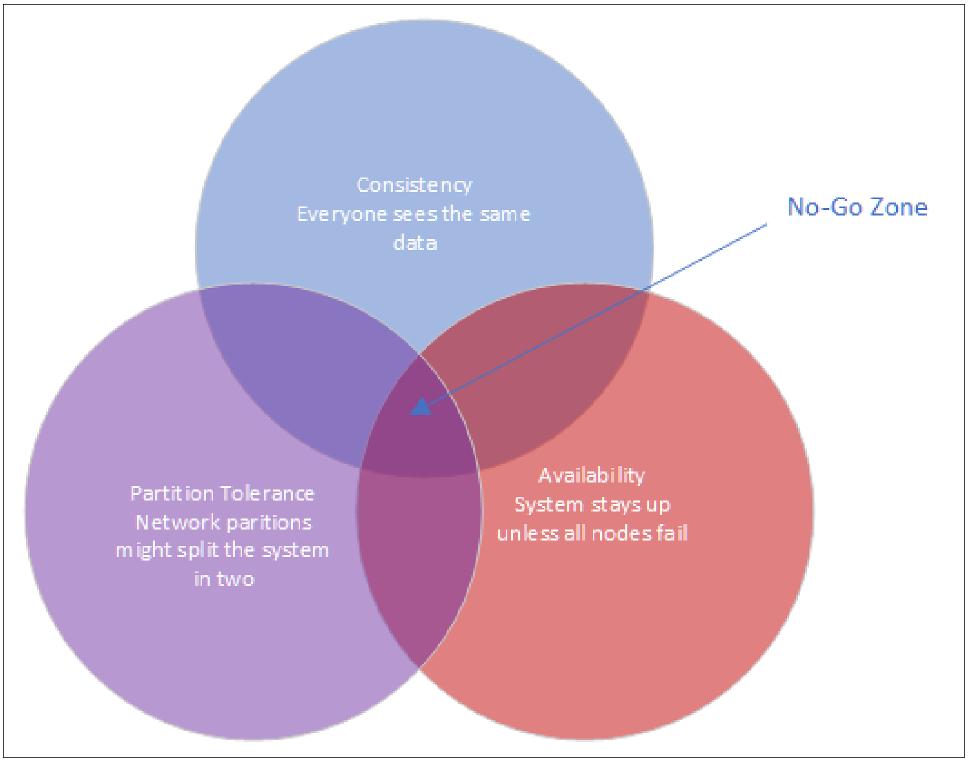

Consistency
Everyone sees the same data
No-Go Zone
Partition Tolerance
Network paritions might split the system in two
Availability
System stays up unless all nodes fail

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).

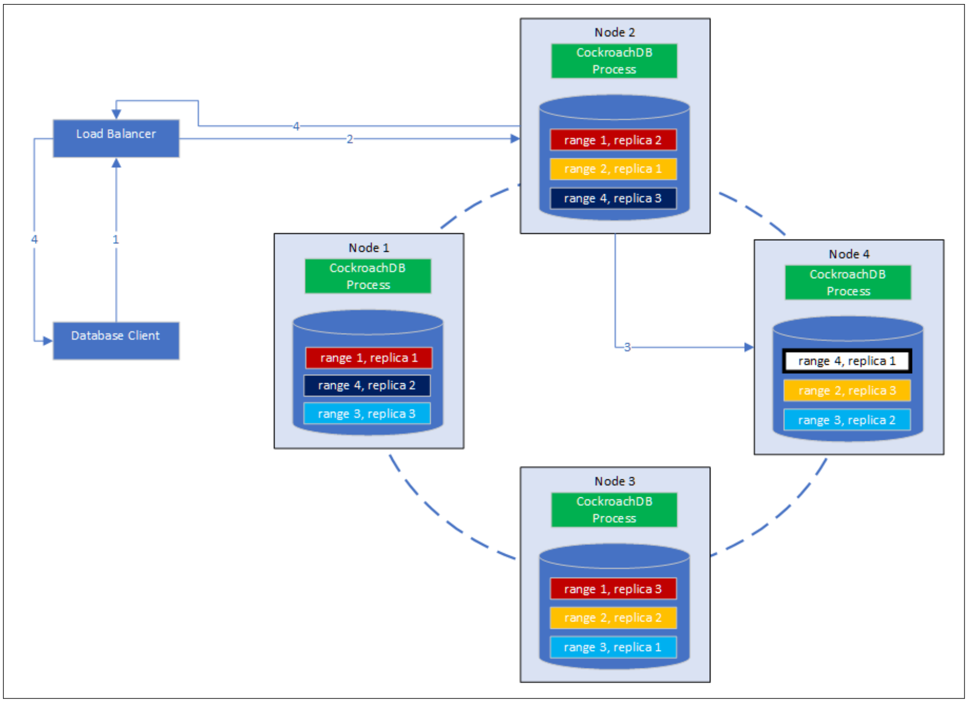


Figure 2-1. CockroachDB Cluster architecture

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.

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.

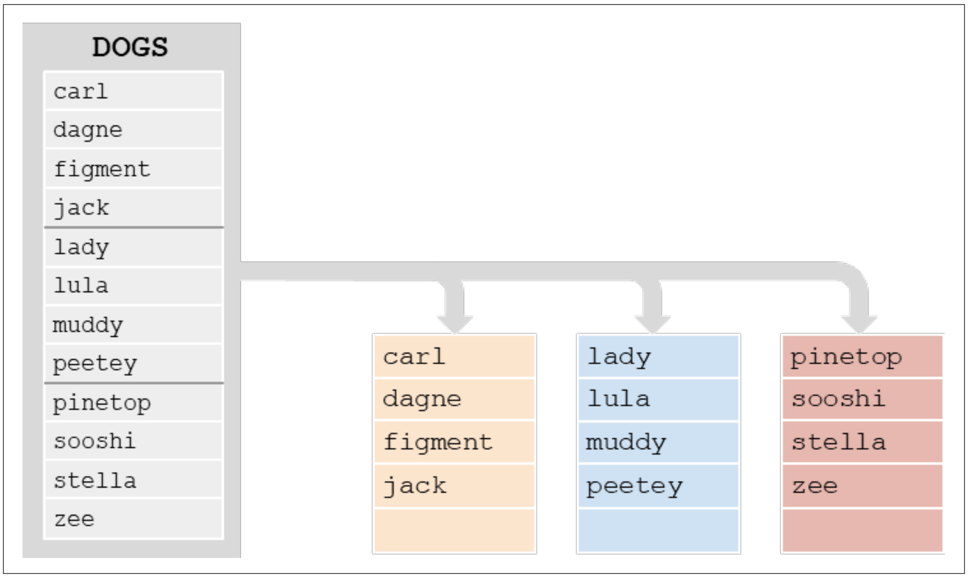


Figure 2-2. Ranges

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.

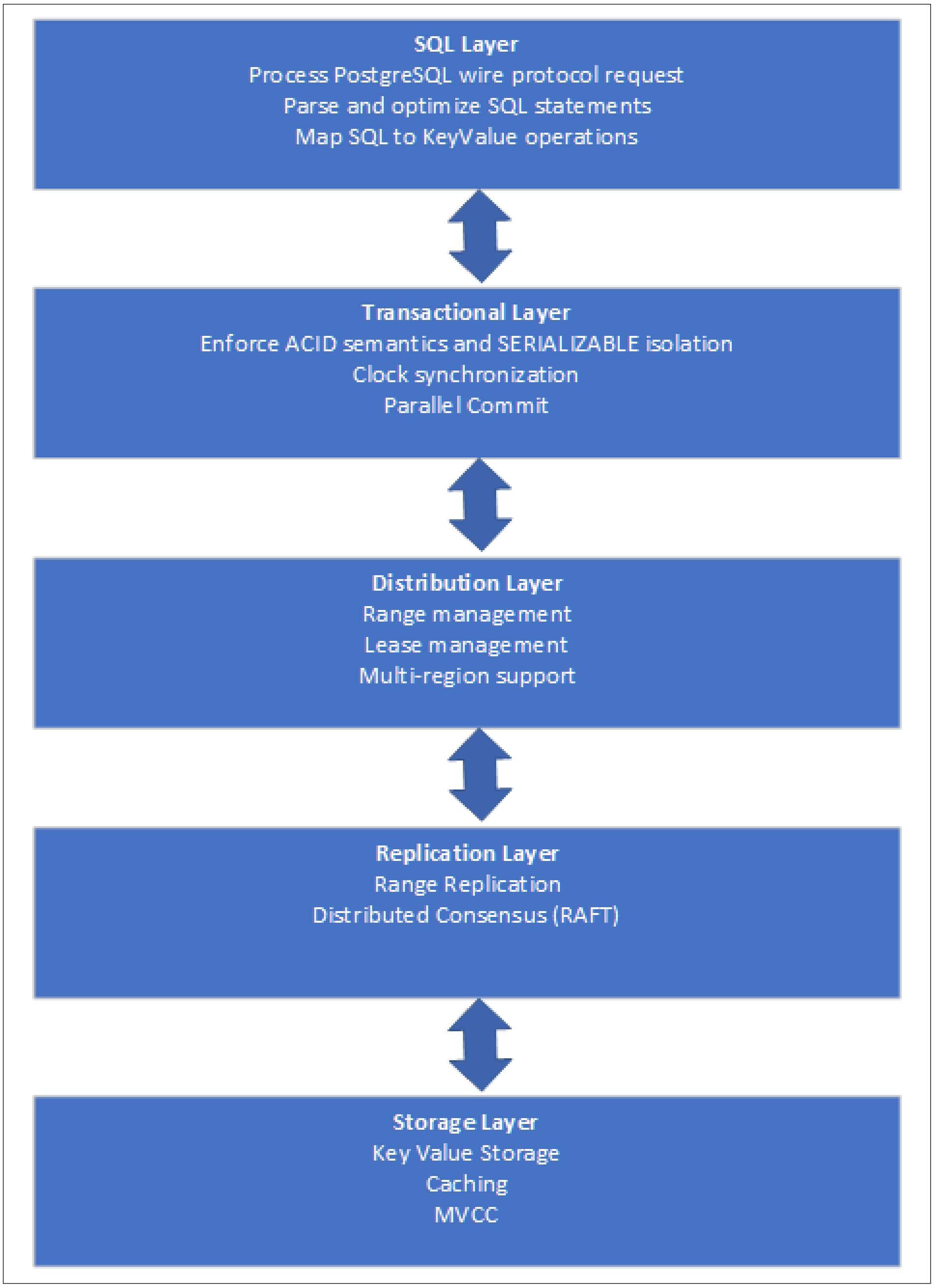


Figure 2-3. CockroachDB Software Layers

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

The CockroachDB SQL layer

The 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. It 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

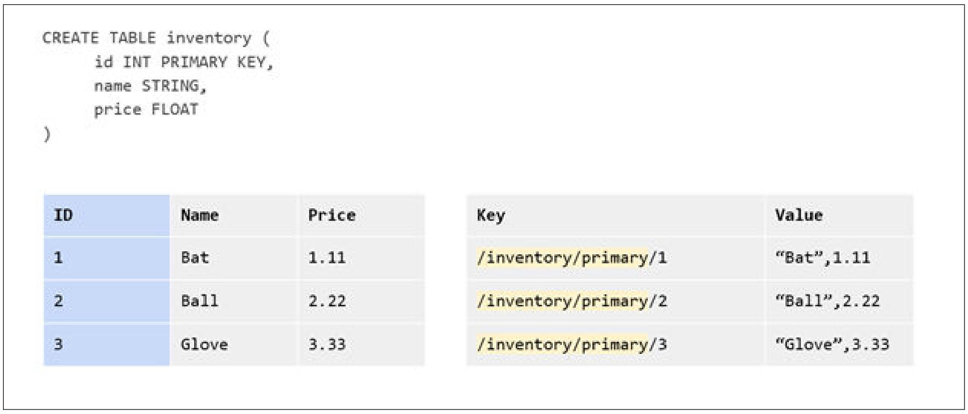
Each entry in the KV store has a Key based on the following structure:

```
/<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.



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

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

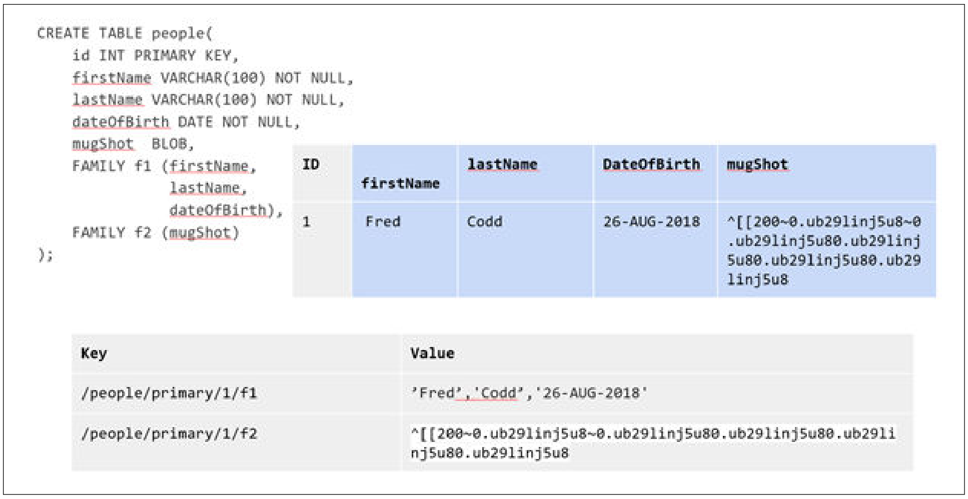
Key	Value
/inventory/primary/1	"Bat",1.11
/inventory/primary/2	"Ball",2.22
/inventory/primary/3	"Glove",3.33

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. Figure 2-5 illustrates this concept – if a table has two column families, then each row in the table will be represented by two KV entries.



```
CREATE TABLE people(
    id INT PRIMARY KEY,
    firstName VARCHAR(100) NOT NULL,
    lastName VARCHAR(100) NOT NULL,
    dateOfBirth DATE NOT NULL,
    mugShot  BLOB,
    FAMILY f1 (firstName,
               lastName,
               dateOfBirth),
    FAMILY f2 (mugShot)
);
```

ID	firstName	lastName	DateOfBirth	mugShot
1	Fred	Codd	26-AUG-2018	^[[200~0.ub29linj5u8~0.ub29linj5u80.ub29linj5u80.ub29linj5u80.ub29linj5u8

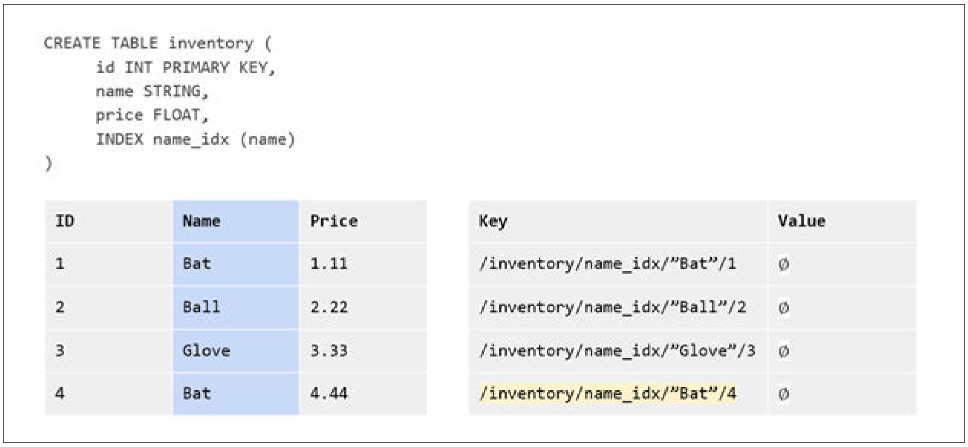
Key	Value
/people/primary/1/f1	'Fred','Codd','26-AUG-2018'
/people/primary/1/f2	^[[200~0.ub29linj5u8~0.ub29linj5u80.ub29linj5u80.ub29linj5u80.ub29linj5u8

Figure 2-5. 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 2-6.



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

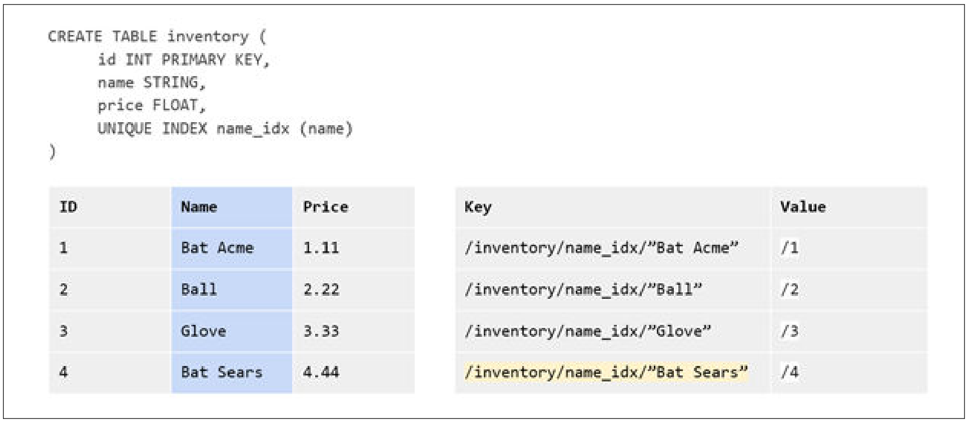
ID	Name	Price
1	Bat	1.11
2	Ball	2.22
3	Glove	3.33
4	Bat	4.44

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

Figure 2-6. 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 2-7.



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

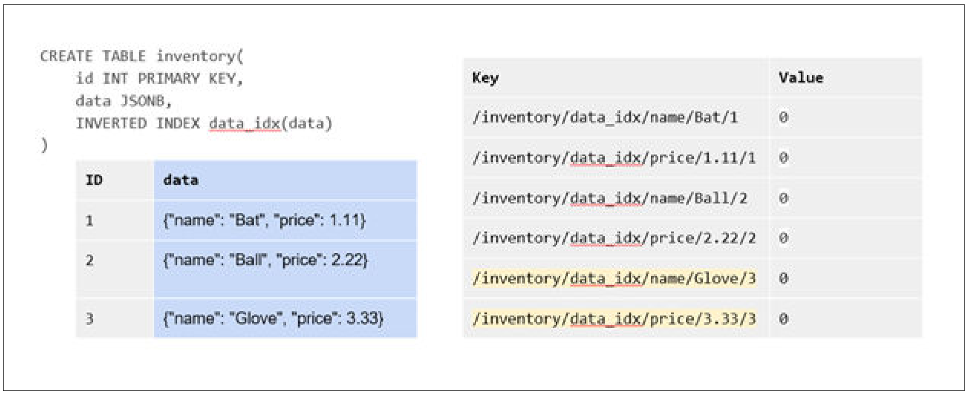
ID	Name	Price
1	Bat Acme	1.11
2	Ball	2.22
3	Glove	3.33
4	Bat Sears	4.44

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 2-7. 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 2-8.



```
CREATE TABLE inventory(
    id INT PRIMARY KEY,
    data JSONB,
    INVERTED INDEX data_idx(data)
)
```

ID	data
1	{"name": "Bat", "price": 1.11}
2	{"name": "Ball", "price": 2.22}
3	{"name": "Glove", "price": 3.33}

Key	Value
/inventory/data_idx/name/Bat/1	0
/inventory/data_idx/price/1.11/1	0
/inventory/data_idx/name/Ball/2	0
/inventory/data_idx/price/2.22/2	0
/inventory/data_idx/name/Glove/3	0
/inventory/data_idx/price/3.33/3	0

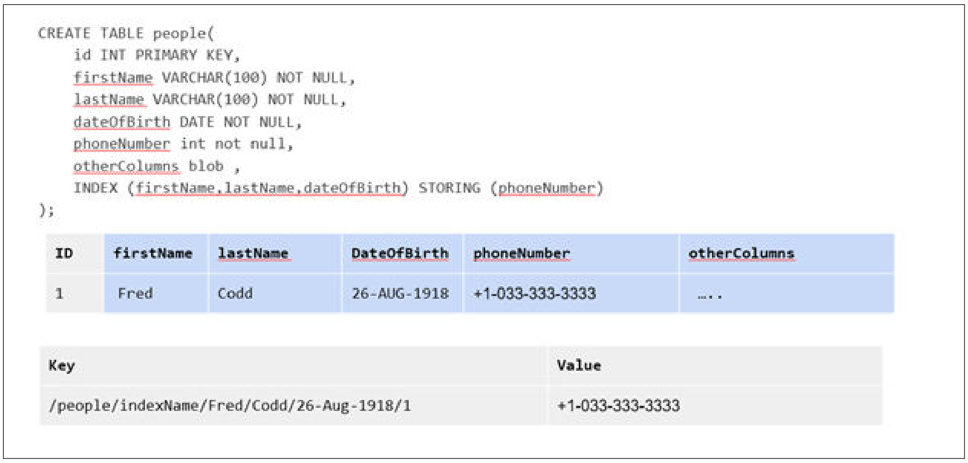
Figure 2-8. Inverted Index KV representation

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 2-9 , 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.



```
CREATE TABLE people(
    id INT PRIMARY KEY,
    firstName VARCHAR(100) NOT NULL,
    lastName VARCHAR(100) NOT NULL,
    dateOfBirth DATE NOT NULL,
    phoneNumber int not null,
    otherColumns blob ,
    INDEX (firstName,lastName,dateOfBirth) STORING (phoneNumber)
);
```

ID	firstName	lastName	DateOfBirth	phoneNumber	otherColumns
1	Fred	Codd	26-AUG-1918	+1-033-333-3333	.....

Key	Value
/people/indexName/Fred/Codd/26-Aug-1918/1	+1-033-333-3333

Figure 2-9. 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`, giving it license to be the only one guiding the schema change. 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 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 transactional 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.

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 2-10 illustrates the basic principles of MVCC. At time t1, session s1 reads from row r2 and accesses version v1 of that row (1). At timestamp t2, another database session s2 updates the row (2), creating version v2 of that row (3). At t3, session s1 reads the row again, but – because s2 has not yet committed it's change – continues to read from version v1 (4). After s2 commits (5) session s1 issues another select and now reads from the new v2 version of the row (6).

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

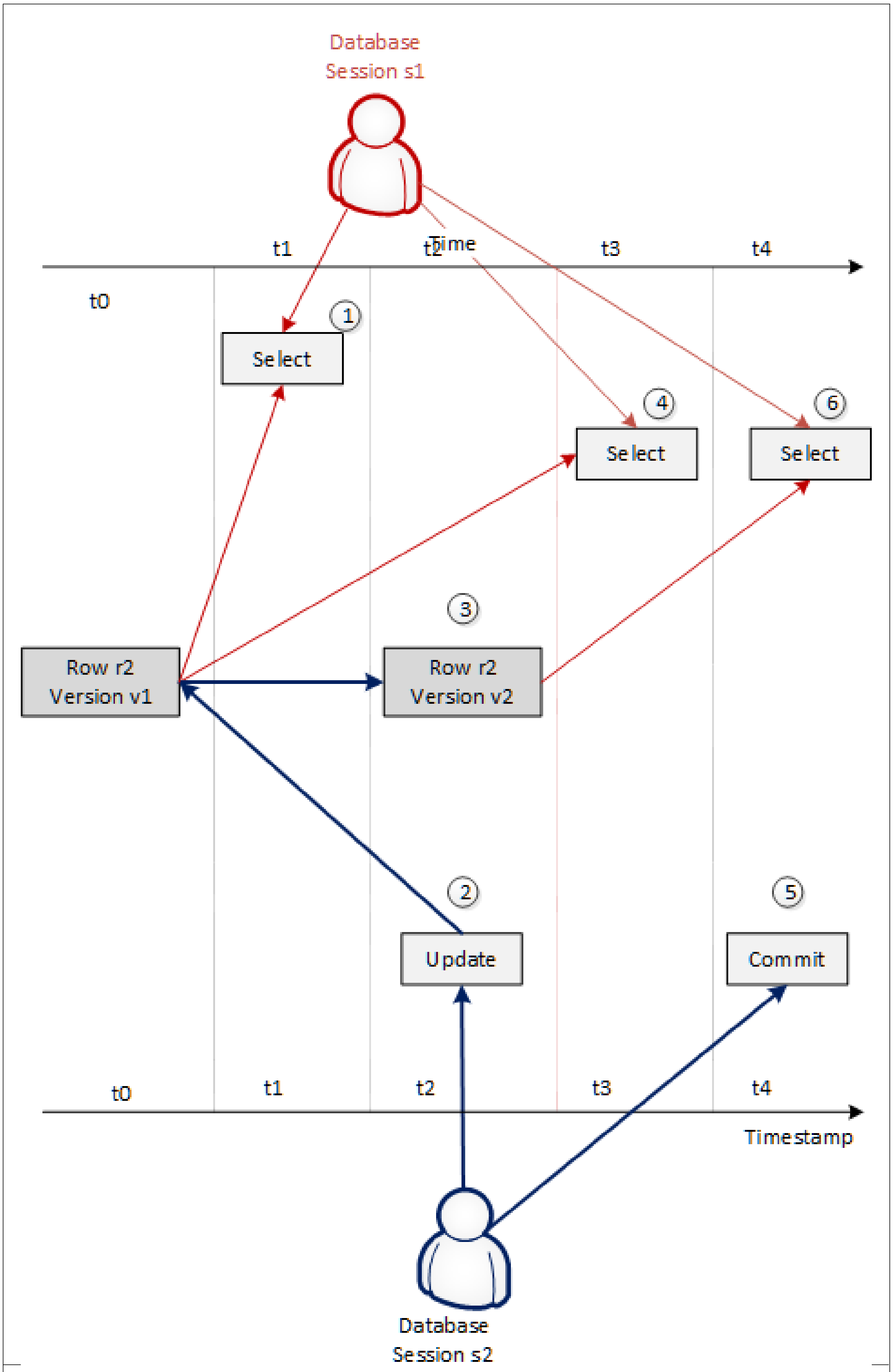


Figure 2-10. MultiVersion Consistency Control (MVCC)

The CockroachDB implemention limits 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 2-11 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 row. 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).

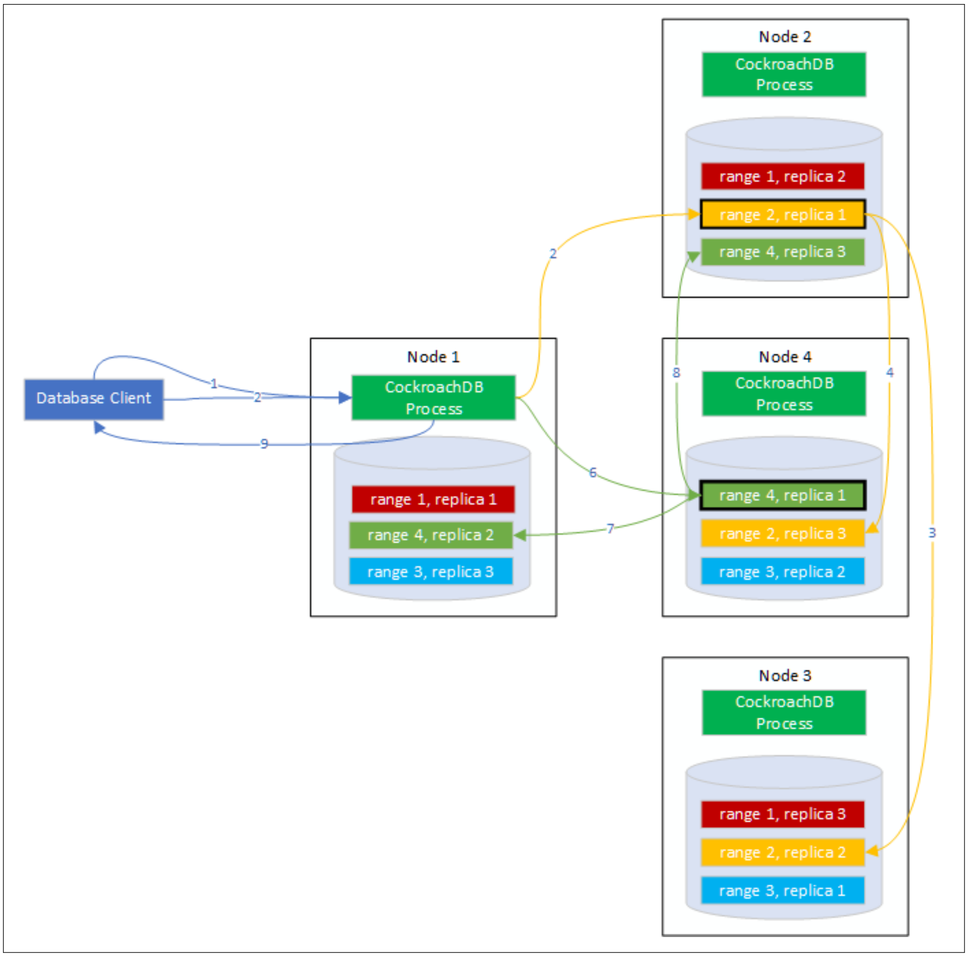


Figure 2-11. Basic Transaction Flow

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 2-11, 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

The flow of a successful two statement transaction is illustrated in Figure 2-12. 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.

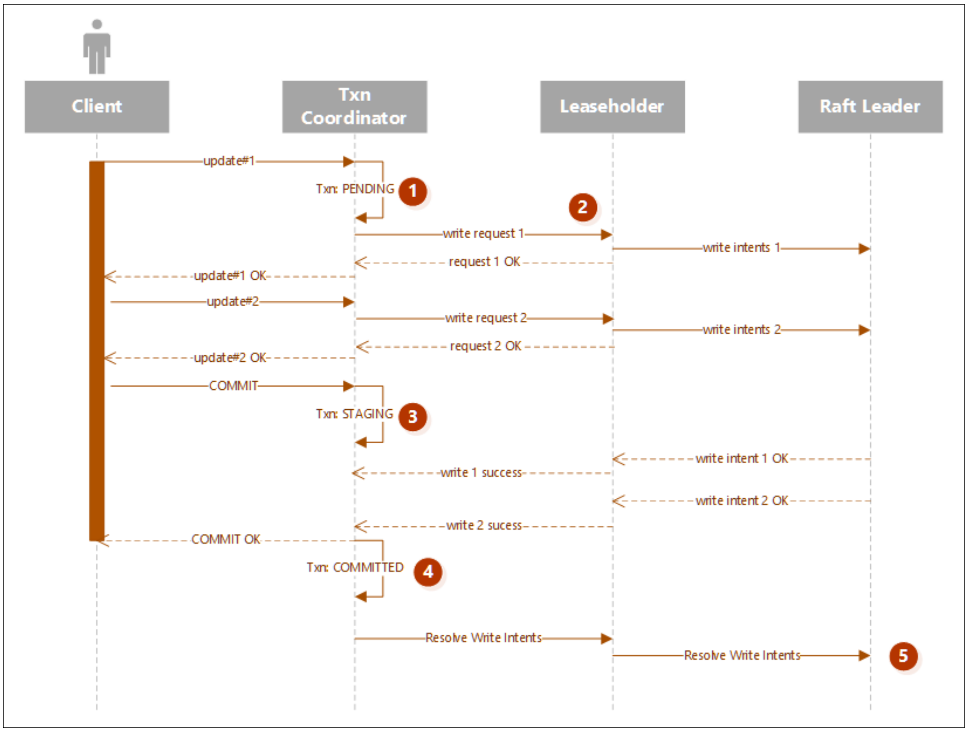


Figure 2-12. Transaction Sequence

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 CockroachDB 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 2-13 illustrates this two-level lookup structure.

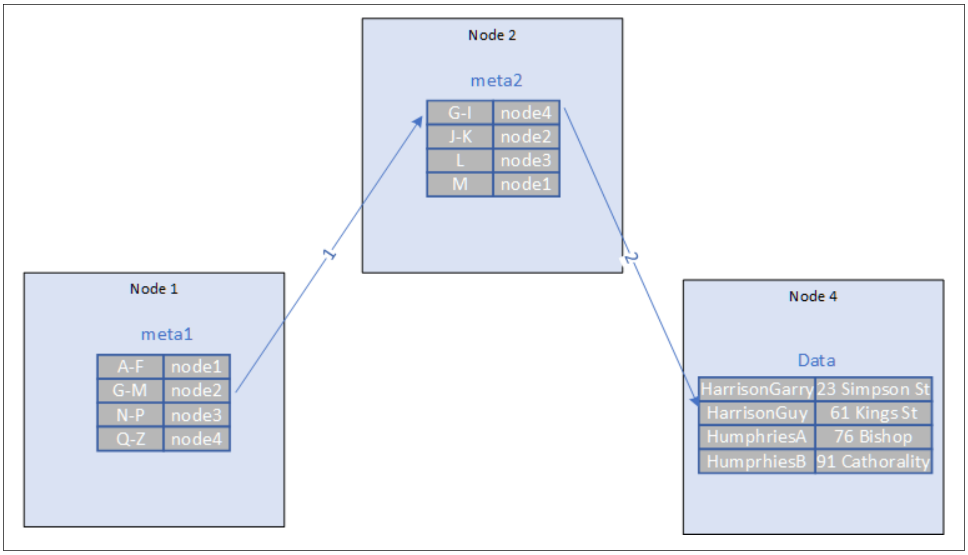


Figure 2-13. Meta Ranges

In Figure 2-13, 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 Leaseholder (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 2-14 illustrates a basic range split when an insert causes a range to exceed the 512MB threshold. Two ranges are created as a consequence.

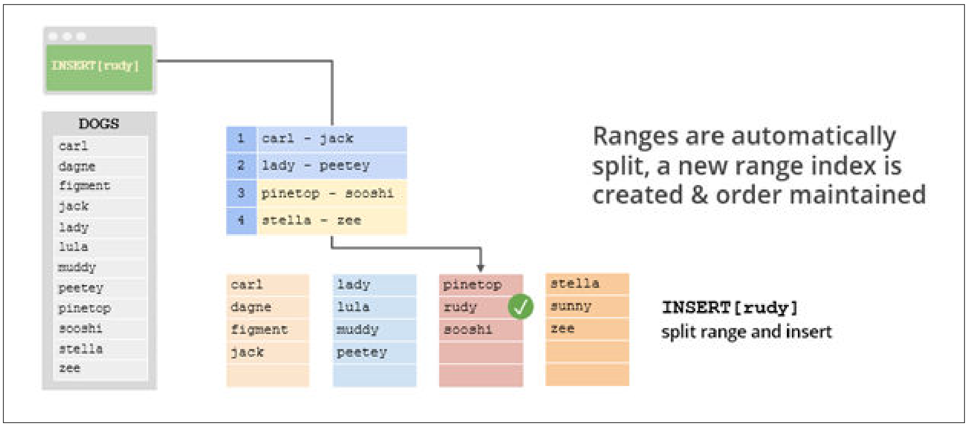


Figure 2-14. Range Splits

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.

The CockroachDB 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.

It's possible to configure non-voting or "read-only" replicas that do not participate in elections and cannot become leaders. These replicas are primarily intended for multi-region deployments and will be discussed in detail in Chapter 10.

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 transactional 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 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. This allows followers to server consistent reads in the recent past, I.e. several seconds ago.

Global tables in a multi-region database use a special variation of the transaction protocol called **non-blocking transactions** which is optimized for reads (from any replica) at the expense of writes. Writes to tables in this mode are assigned timestamps in the future, and timestamps in the future may be closed. This makes it possible for followers to serve consistent reads at the present time.

The CockroachDB 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 **Pebble** storage engine – an open-source Key-Value store based inspired by the LevelDB and RocksDB storage engines. Pebble is primarily maintained by the CockroachDB team and is optimized specifically for CockroachDB use cases. Older versions of CockroachDB use the RocksDB storage engine.

Let's look under the hood of the PebbleDB storage engine so that we can fully appreciate how CockroachDB stores and manipulates data at it's foundational layer.

Log Structured Merge (LSM) Trees

Pebble implements the Log Structured Merge Tree (LSM) architecture. LSM in an widely implemented and battle-tested architecture 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 2-15 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).

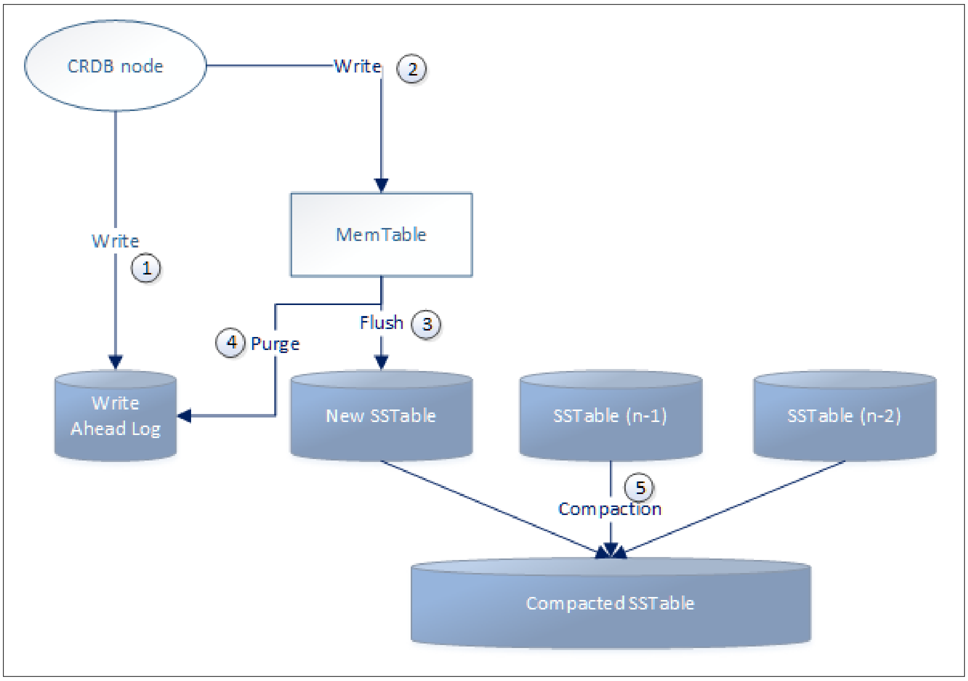


Figure 2-15. LSM Writes

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 2-16 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.

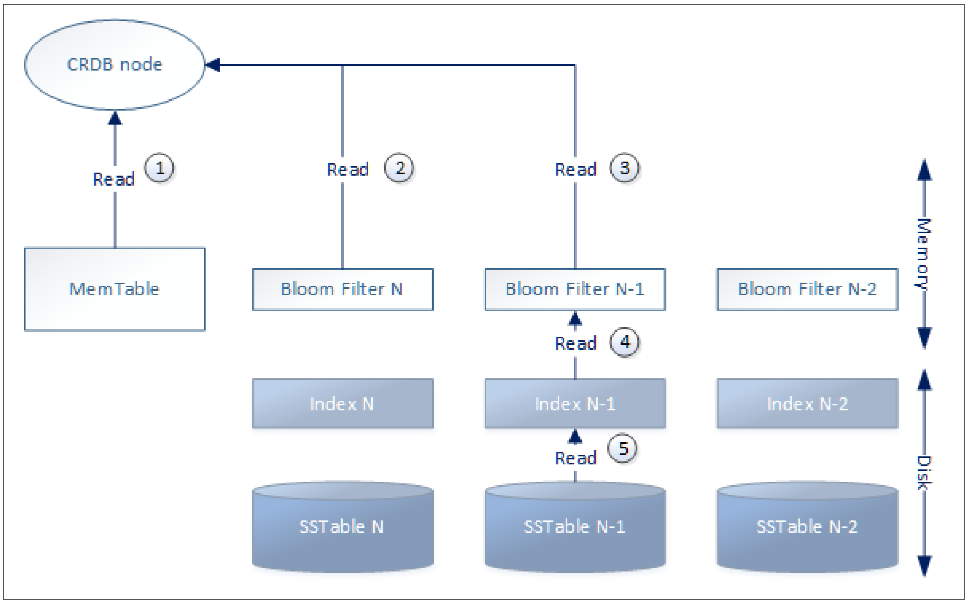


Figure 2-16. LSM Reads

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 transactional layer earlier in the chapter – see, for instance, Figure 2-10.

CockroachDB encodes the MVCC timestamp into the each key so that multiple MVCC versions of a key are stored as distinct keys within Pebble. 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

Pebble 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.

[[Ch03 – Getting Started]]

CHAPTER 3

Getting Started

CockroachDB has a sophisticated and modern architecture and is designed for a global scale. However, that complexity and scalability don't imply a steep learning curve or barrier to entry. In this chapter we'll help you get started with a CockroachDB installation and introduce you to the basics of working with a CockroachDB system.

Installation

CockroachDB can be installed on virtually any flavor of desktop operating system within a few minutes. Alternatively, you can create a free CockroachCloud database or run CockroachDB within a Docker container or Kubernetes cluster.

Installing CockroachDB software

In most scenarios, you'll want to have the CockroachDB software installed on your desktop computer, so let's start with that. You'll find a full list of CockroachDB binaries at *https://www.cockroachlabs.com/docs/releases/*. From there, you can pick your operating system and download the most recent version or pick a previous version.

Installation on MacOS

If you have the `brew` package manager installed, then that is probably the easiest way to get started installing CRDB on Mac. In fact, even if you don't have `brew` installed, its probably easier to install it and then install `CRDB` than to install CRDB manually.

To install `brew`, issue the following command from a terminal window:

```
/bin/bash -c "$(curl -fsSL \
    https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once brew is installed, you can install CRDB with the following command:

```
guyharrison@macos ~ % brew install cockroachdb/tap/cockroach
==> Tapping cockroachdb/tap
Cloning into '/usr/local/Homebrew/Library/Taps/cockroachdb/homebrew-tap'...
 …
==> Installing cockroach from cockroachdb/tap
….
To have launchd start cockroachdb/tap/cockroach now and restart at login:
  brew services start cockroachdb/tap/cockroach
Or, if you don't want/need a background service you can just run:
  cockroach start-single-node --insecure
==> Summary
   /usr/local/Cellar/cockroach/21.1.1: 134 files, 184.8MB, built in 8 seconds
```

One of the great things about brew is that it sets up CRDB as a service, so you can issue `brew services start cockroach` to start a background instance of CRDB.

However, If you don't want to use brew, then you can download the CockroachDB binary directly and copy the binary into your path. Visit *https://www.cockroachlabs.com/docs/releases/?filters=mac* to determine the path for the release you want, then use `curl` or `wget` to copy and decompress that binary:

```
$ curl https://binaries.cockroachdb.com/cockroach-v21.1.1.darwin-10.9-amd64.tgz \
    | tar -xJ
```

You can then copy the binary into your PATH so you can execute cockroach commands from any shell:

```
$ sudo cp -i cockroach-v21.1.1.darwin-10.9-amd64/cockroach /usr/local/bin/
```

Note that copying the binary directly into your path may not install some of the ancillary libraries which support geospatial functionality. Consult the CockroachDB web site (*https://www.cockroachlabs.com/docs/stable/install-cockroachdb-mac.html*) for further details.

Once you've installed CRDB either manually, or via brew, run the `cockroach demo` command to start a demo instance and confirm that it is running:

```
guyharrison@macos ~ % cockroach demo
#
# Welcome to the CockroachDB demo database!
#
# You are connected to a temporary, in-memory CockroachDB cluster of 1 node.
#
# Enter \? for a brief introduction.
#
root@127.0.0.1:49418/movr> show databases;
  database_name | owner
----------------+--------
  defaultdb     | root
  movr          | root
  postgres      | root
  system        | node
(4 rows)

Time: 1ms total (execution 1ms / network 0ms)

root@127.0.0.1:49418/movr>
```

Installation on Linux

To perform a basic installation on Linux, visit *https://www.cockroachlabs.com/docs/releases/* To obtain the latest release (or a specific version you are interested in), download and unpack it. Of course, you can use `curl` or `wget` to obtain the tarball once you have determined its path:

```
$ wget https://binaries.cockroachdb.com/cockroach-v21.1.1.linux-amd64.tgz

2021-04-17 16:10:35  - 'cockroach-v21.1.1.linux-amd64.tgz' saved
$ sudo cp -ir cockroach-v21.1.1.linux-amd64/cockroach  /usr/local/bin
```

Once installed, run the `cockroach demo` command to start a temporary local instance of CRDB and verify the installation.

```
$ cockroach demo
#
# Welcome to the CockroachDB demo database!
#
# You are connected to a temporary, in-memory CockroachDB cluster of 1 node.
#
…
#
# Enter \? for a brief introduction.
#
root@127.0.0.1:44913/movr> show databases;
  database_name | owner
----------------+--------
  defaultdb     | root
  movr          | root
  postgres      | root
  system        | node
(4 rows)

Time: 1ms total (execution 1ms / network 0ms)

root@127.0.0.1:44913/movr>
```

Installation on Microsoft Windows

Microsoft Windows is not a fully supported platform for running a CockroachDB server; certain features - such as spatial - is not available. However, Windows is completely supported for CockroachDB clients and the server runs well enough for experimentation and most developent.

From *https://www.cockroachlabs.com/docs/releases/?filters=windows*, click on the link for the release you'd like to download. Once downloaded, unzip the archive into a directory and add the subdirectory containing the `cockroach.exe` to your PATH

Alternatively, you can download the file directly from a PowerShell prompt. Here, we download the zip fie for version 21.1.0, unzip it to a directory `c:\tools\cockroachdb`:

```
PS > wget wget https://binaries.cockroachdb.com/cockroach-v21.1.0.windows-6.2-amd64.zip
     -OutFile crdb.zip
PS > mkdir c:\tools\cockroachdb
PS > Expand-Archive -Path crdb.zip -DestinationPath C:\tools\cockroachdb\
```

Now you can test your installation by issuing the `cockroach demo` command:

```
PS > cockroach demo
#
# Welcome to the CockroachDB demo database!
#
# You are connected to a temporary, in-memory CockroachDB cluster of 1 node.
#
#
# Enter \? for a brief introduction.
#
root@127.0.0.1:57574/movr> show databases;
  database_name | owner
----------------+--------
  defaultdb     | root
  movr          | root
  postgres      | root
  system        | node
(4 rows)

Time: 2ms total (execution 1ms / network 0ms)
```

Connection URLs

When connecting to a CockroachDB cluster, we need to identify the location and credentials with which we wish to connect. When connecting to a local server using `cockroach demo` or `cockroach sql`, the CockroachDB client will default to a local server on the default port, but as we will see, more complex installations require quite a bit more information.

The most common way to connect is to use a PostgreSQL compatible URL. This URL is of the following format:

```
postgresql://[user[:passwd]@]host[:port]/[db][?parameters...]
```

The simplest possible URL – for a local cluster running without authentication would look something like this:

```
$ cockroach sql --url 'postgres://root@localhost:26257?sslmode=disable'
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.1
# Cluster ID: 072189bb-3970-4f37-afe4-55bc37cdf76e
#
# Enter \? for a brief introduction.
#
root@localhost:26257/defaultdb>
```

This is equivalent to running the command `cockroach sql -insecure`.

The beauty of the URL is that it can be accepted by most PostgreSQL-compatible programs or drivers. For instance, if we have the PostgreSQL client installed, we can use it to connect to CockroachDB:

```
$ psql 'postgres://root@localhost:26257?sslmode=disable'
psql (13.2, server 9.5.0)
Type "help" for help.
root=#
```

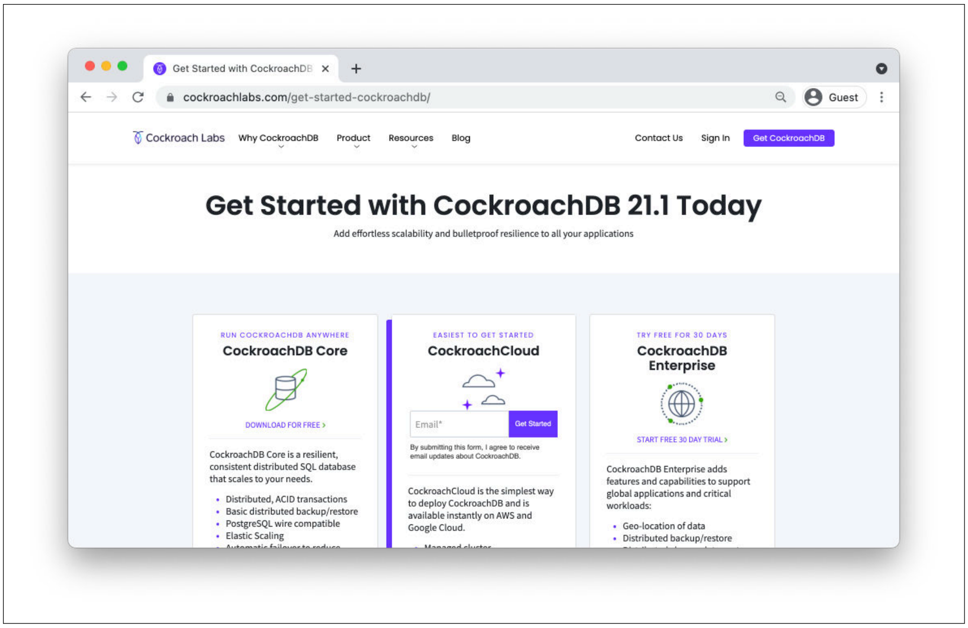
Creating a CockroachCloud cluster

The `cockroach demo` command is a handy way for playing with the CockroachDB server, but the easiest way to get a fully functional CockroachDB server with persistent storage is to take advantage of the CockroachCloud free cloud database service. This service grants you access to a fully functional multi-tenant cloud service with 5GB of storage.

The CockroachCloud has a number of advantages compared with a desktop deployment:

- It's automatically configured for high availability and backup. You don't have to worry about losing your data in the event of a hard drive failure on your desktop.
- It's fully secured using encryption at rest and in transit
- It's available from anywhere, so it can be used for team development purposes.

To create a CockroachCloud server, navigate to *https://www.cockroachlabs.com/get-started-cockroachdb/* and select the CockroachCloud option. Enter your email as shown in Figure 3-1.



Get Started with CockroachDB 21.1 Today

Add effortless scalability and bulletproof resilience to all your applications

CockroachDB Core is a resilient, consistent distributed SQL database that scales to your needs.

- Distributed, ACID transactions
- Basic distributed backup/restore
- PostgreSQL wire compatible
- Elastic Scaling

By submitting this form, I agree to receive email updates about CockroachDB.

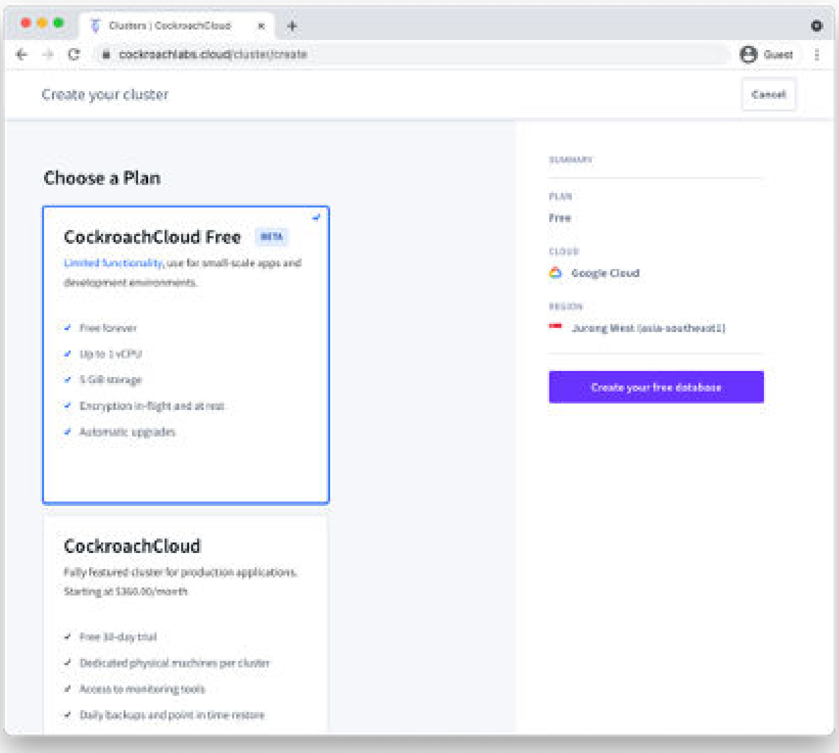
CockroachCloud is the simplest way to deploy CockroachDB and is available instantly on AWS and Google Cloud.

CockroachDB Enterprise adds features and capabilities to support global applications and critical workloads:

- Geo-location of data
- Distributed backup/restore

Figure 3-1. Signing up for CockroachCloud

After entering your details and validating your email address, you'll be given the option to create your free cluster as shown in Figure 3-2.



Clusters | CockroachCloud

cockroachlabs.cloud/cluster/create

Create your cluster

Cancel

Choose a Plan

CockroachCloud Free BETA

Limited functionality, use for small-scale apps and development environments.

- ✓ Free forever
- ✓ Up to 1 vCPU
- ✓ 5 GiB storage
- ✓ Encryption in-flight and at rest
- ✓ Automatic upgrades

CockroachCloud

Fully featured cluster for production applications. Starting at $360.00/month

- ✓ Free 30-day trial
- ✓ Dedicated physical machines per cluster
- ✓ Access to monitoring tools
- ✓ Daily backups and point in time restore

SUMMARY

PLAN
Free

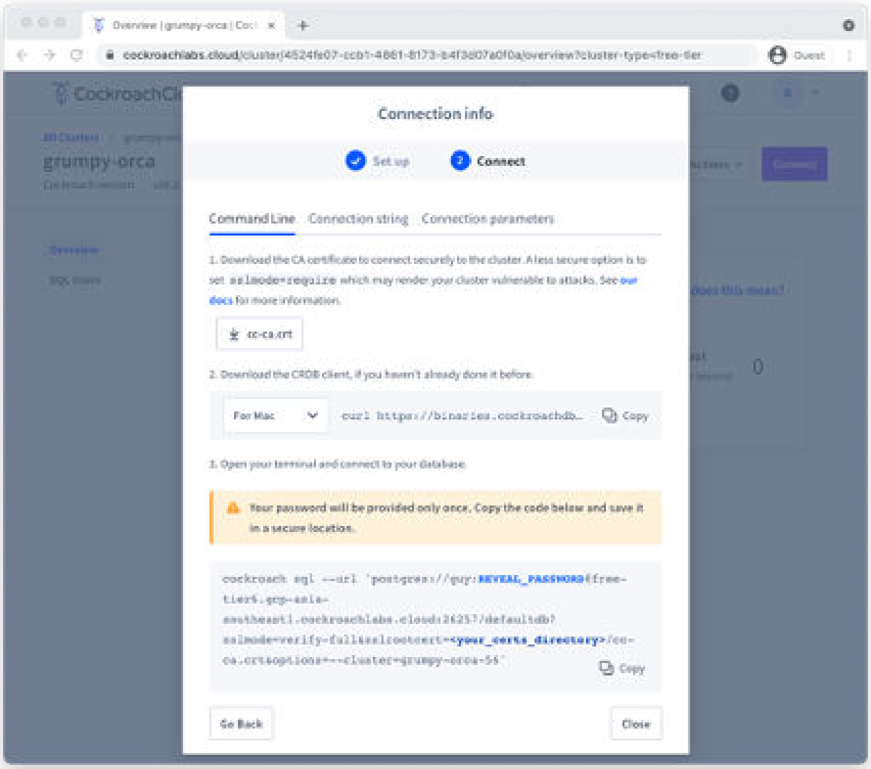
CLOUD
Google Cloud

REGION
Jurong West (asia-southeast1)

Create your free database

Figure 3-2. Creating a free CockroachCloud Database

Once created, the **Connection info** dialogue should appear, with information on how to connect to your new cluster – see Figure 3-3. Download the CA certificate `cc-ca.crt`+ by clicking on the first link in the dialogue and store it on your desktop.



Connection info

1. Download the CA certificate to connect securely to the cluster. A less secure option is to set `sslmode=require` which may render your cluster vulnerable to attacks. See our docs for more information.

2. Download the CRDB client, if you haven't already done it before.

```
curl https://binaries.cockroachdb…
```

3. Open your terminal and connect to your database.

Your password will be provided only once. Copy the code below and save it in a secure location.

```
cockroach sql --url 'postgres://guy:REVEAL_PASSWORD@free-
tier6.gcp-asia-
southeast1.cockroachlabs.cloud:26257/defaultdb?
sslmode=verify-full&sslrootcert=<your_certs_directory>/cc-
ca.crt&options=--cluster=grumpy-orca-56'
```

Figure 3-3. Connecting to CockroachCloud

Once the certificate is stored on the desktop, you can use the connection string provided to establish a connection. Below we copy the `cc-ca.crt` file into a +`~/.cockroach-certs/ca.crt + folder, and then connect to the newly created cloud database.

```
$ mkdir $HOME/.cockroach-certs

$ cp $HOME/Downloads/cc-ca.crt $HOME/.cockroach-certs

$ cockroach sql --url 'postgres://guy:myPassword@
   free-tier6.gcp-asia-southeast1.cockroachlabs.cloud:26257/defaultdb?
  sslmode=verify-full&sslrootcert=/Users/guyharrison/.cockroach-certs
   /cc-ca.crt&options=--cluster=grumpy-orca-56'

#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
```

```
#
# Client version: CockroachDB CCL v21.1.0
# Server version: CockroachDB CCL v20.2.8
# Cluster ID: 45851b67-5277-4795-aab9-390c70a78786
#
# Enter \? for a brief introduction.
#
guy@free-tier6.gcp-asia-southeast1.cockroachlabs.cloud:26257/defaultdb>
   show databases;

  database_name | owner
----------------+--------
  defaultdb     | root
  postgres      | root
  system        | node
(3 rows)

Time: 106ms total (execution 7ms / network 100ms)
```

CockroachCloud passwords

Note that the password in the connection string is **not** the password you provided to connect to your CockroachCloud account. Your CockroachCloud account might be associated with many databases, each of which has its own password.

The password shown in the connection dialogue in Figure 3-3 will be shown only if you hover over the REVEAL_PASSWORD link and will only be shown at this point in the database creation. It's up to you to save that password and keep it safe.

Starting a local single-node server

As we've seen above, you can use the cockroach demo command to start a temporary demo cluster, and we can quickly create a free CockroachCloud server. But if you want to start a single-node CockroachDB with persistent storage on your own hardware, you can use the start=single-node option:

```
$ cockroach start-single-node --insecure
*
* WARNING: ALL SECURITY CONTROLS HAVE BEEN DISABLED!
*
* This mode is intended for non-production testing only.
*
* In this mode:
* - Your cluster is open to any client that can access any of your IP addresses.
* - Intruders with access to your machine or network can observe client-server traffic.
* - Intruders can log in without password and read or write any data in the cluster.
* - Intruders can consume all your server's resources and cause unavailability.
*
*
```

```
* INFO: To start a secure server without mandating TLS for clients,
* consider --accept-sql-without-tls instead. For other options, see:
*
* - https://go.crdb.dev/issue-v/53404/v20.2
* - https://www.cockroachlabs.com/docs/v20.2/secure-a-cluster.html
*
*
* WARNING: neither --listen-addr nor --advertise-addr was specified.
* The server will advertise "mubuntu" to other nodes, is this routable?
*
* Consider using:
* - for local-only servers:  --listen-addr=localhost
* - for multi-node clusters: --advertise-addr=<host/IP addr>
```

This will start a single node CockroachDB cluster with no security controls. To connect to this server we can use the `cockroach sql` command with the default connection string:

```
$ cockroach sql --insecure
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.1
# Cluster ID: 848d8b85-4000-484a-b4ad-8f2c76c68221
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb> show databases;
  database_name | owner
----------------+--------
  defaultdb     | root
  postgres      | root
  system        | node
(3 rows)

Time: 3ms total (execution 2ms / network 0ms)

root@:26257/defaultdb>
```

Insecure mode

The use of the `insecure` flag when starting a CockroachDB server is convenient for quickly starting a CockroachDB server, but it is absolutely not appropriate for a production system. Please see Chapter 12 for instructions on setting up a properly secured production system.

Starting up CockroachDB in a docker container

If you have docker, you can quickly start a CockroachDB single node instance inside a docker container.

You'll need a persistent volume for data, so let's create that first:

```
$ docker volume create crdb1
```

Then, we invoke `docker run` to pull and start the latest CockroachDB docker image and start the server in single-node, insecure mode:

```
$ docker run -d \
> --name=crdb1 \
> --hostname=crdb1 \
> -p 26257:26257 -p 8080:8080  \
> -v "crdb1:/cockroach/cockroach-data"  \
> cockroachdb/cockroach:latest start-single-node  \
> --insecure \
>
Unable to find image 'cockroachdb/cockroach:latest' locally
latest: Pulling from cockroachdb/cockroach
a591faa84ab0: Pull complete
…
6913e7a5719b8cb705c32540523885f6592270cf091ac1013cca66914b1aafe8
```

The output of the docker run command is the container identifier for the CockroachDB container. Using that containerId, we can connect to that container using the `cockroach sql` command.

```
$ docker exec \
    -it 6913e7a5719b8cb705c32540523885f6592270cf091ac1013cca66914b1aafe8 \
    cockroach sql --insecure
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.1
# Cluster ID: 8fcbb9bb-ec7c-40dc-afe0-90306c87f5d7
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb> show databases;
  database_name | owner
----------------+--------
  defaultdb     | root
  postgres      | root
  system        | node
(3 rows)

Time: 3ms total (execution 3ms / network 0ms)
```

We don't need to have the CockroachDB software installed on our local host to connect using the above method, since we are using the cockroachdb client installed within the docker container. However, since we've forwarded port 26257 from the docker container, we can attach from the desktop using the default connection:

```
$  ~ cockroach sql --insecure
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Client version: CockroachDB CCL v21.1.0
# Server version: CockroachDB CCL v21.1.1
# Cluster ID: d070609f-58a7-4aea-aa27-92bc4a1e5406
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb>
```

Note that this port forwarding can only work if there's not already a CockroachDB server listening on that port.

Starting up a secure server

In the previous examples, we've used the `--insecure` mode to start the server without needing to configure secure communications. This is a quick way to set up a test server but is catastrophically dangerous for anything that contains valuable data.

We'll cover CockroachDB security in-depth within Chapter 12, but for now, to set up a secure server, we need to create security certificates to encrypt the communications channel and authenticate the client and server.

The following commands create the certificates. The Certificate Authority key will be held in `my-safe-directory`; the certificates themselves will be held in the `certs` directory:

```
$ mkdir certs my-safe-directory

$ # CA certificate and keypair

$ cockroach cert create-ca \
>     --certs-dir=certs \
>     --ca-key=my-safe-directory/ca.key


$ # certificate and keypair for localhost
$ cockroach cert create-node localhost `hostname` --certs-dir=certs \
>     --ca-key=my-safe-directory/ca.key

$ # certificate for the root user
$ cockroach cert create-client root \
>     --certs-dir=certs \
>     --ca-key=my-safe-directory/ca.key
```

We can now start the server and specify the directory containing the certificates:

```
$ cockroach start-single-node --certs-dir=certs --background
*
* WARNING: neither --listen-addr nor --advertise-addr was specified.
* The server will advertise "mubuntu" to other nodes, is this routable?
*
* Consider using:
* - for local-only servers:  --listen-addr=localhost
* - for multi-node clusters: --advertise-addr=<host/IP addr>
*
*
$ *
* INFO: Replication was disabled for this cluster.
* When/if adding nodes in the future, update zone
  configurations to increase the replication factor.
```

Now when connecting, we must specify the certificates directory. If we are connecting from a remote host then we would need to copy the certificates to that host.

```
$  cockroach sql --certs-dir=certs --certs-dir=certs
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.0
# Cluster ID: f908d29e-1fb6-40b8-9e1f-a2a0a3763603
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb>
```

Certificates directory

On Linux or MacOS systems, CockroachDB will look for certificates in the `~/.cockroach-certs directory. So if your certificates are placed there, then you won't need to specify the – -certs-dir argument. However, if you have multiple CockroachDB servers then you may need to maintain distinct certificates for each, possibly in their own directories.

Remote connection

In the previous examples, we've connected to a server running on the same host as our client. This is pretty unusual in the real world, where we would normally be connecting to a server on another machine. Typically, we'd specify the URL parameter to identify the server concerned. For instance, to connect to a server on the mubuntu server on the default port, we could issue the following command:

```
$ cockroach sql --certs-dir=certs --url postgresql://root@mubuntu:26257/defaultdb
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.0
# Cluster ID: f908d29e-1fb6-40b8-9e1f-a2a0a3763603
#
# Enter \? for a brief introduction.
#
root@mubuntu:26257/defaultdb>
```

Creating a Kubernetes cluster

In the above examples, we've created single-node clusters and connected to a free CockroachCloud database which is a shared region of a multi-tenant cluster. If you want to start with a dedicated multi-node cluster, then the easiest way is to install a CockroachDB cluster in a Kubernetes environment using the CockroachDB Kubernetes operator.

Kubernetes is an increasingly ubiquitous framework that coordinates – orchestrates – the management of the components of a distributed system. The CockroachDB Kubernetes operator contains the configuration and utilities that allow CockroachDB to be deployed in Kubernetes.

We'll come back to production deployment options for Kubernetes later in the book. For now, we will deploy CockroachDB in a Kubernetes Minikube cluster, which implements a local Kubernetes cluster on a desktop system.

For this example, we are using a minikube cluster running on macOS with 6 CPUs and 12GB of memory.

The first step is to deploy the operator, and it's manifest:

```
$ kubectl apply -f \
   https://raw.githubusercontent.com/cockroachdb/cockroach-operator/master
     /config/crd/bases/crdb.cockroachlabs.com_crdbclusters.yaml

customresourcedefinition.apiextensions.k8s.io/crdbclusters.crdb.cockroachlabs.com created

$ kubectl apply \
  -f https://raw.githubusercontent.com/cockroachdb/cockroach-operator/
     master/manifests/operator.yaml

clusterrole.rbac.authorization.k8s.io/cockroach-database-role created
serviceaccount/cockroach-database-sa created
clusterrolebinding.rbac.authorization.k8s.io/cockroach-database-rolebinding created
```

```
role.rbac.authorization.k8s.io/cockroach-operator-role created
clusterrolebinding.rbac.authorization.k8s.io/cockroach-operator-rolebinding created
clusterrole.rbac.authorization.k8s.io/cockroach-operator-role created
serviceaccount/cockroach-operator-sa created
rolebinding.rbac.authorization.k8s.io/cockroach-operator-default created
deployment.apps/cockroach-operator created
```

Once this is done, a `kubectl get pods` command should show the CockroachDB Kubernetes operator running inside the cluster:

```
$ kubectl get pods
NAME                                  READY   STATUS              RESTARTS   AGE
cockroach-operator-84bf588dbb-65m8k   0/1     ContainerCreating   0          9s
```

We then retrieve the example configuration file that is included in the operators repository.

```
$ curl -O \
  https://raw.githubusercontent.com/cockroachdb/cockroach-operator/master/
      examples/example.yaml

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1098  100  1098    0     0   3399      0 --:--:-- --:--:-- --:--:--  3399
```

This file contains definitions for the cluster to be configured, such as the number of nodes to be created and the memory and CPU required by each node. The configuration is tilted towards a production deployment, so you might want to trim down the requirements. For instance, below we see that the default configuration file specifies a 60GB storage requirement. We might want to change this to a lower value for a simple test system (or increase it for a bigger deployment):

```
apiVersion: crdb.cockroachlabs.com/v1alpha1
kind: CrdbCluster
metadata:
  name: cockroachdb
spec:
  dataStore:
    pvc:
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: "60Gi"
        volumeMode: Filesystem
```

You could edit other elements of the configuration file, such as the number of nodes to be created or the version of CockroachDB to be used.

We now apply the configuration file to the operator, which will perform the necessary tasks to create the cluster:

```
$ kubectl apply -f myconfig.yaml

crdbcluster.crdb.cockroachlabs.com/cockroachdb created
```

The cluster creation process can take some time. We'll know it's complete when a `kubectl get pods` command shows all nodes in Running state:

```
$ kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
cockroach-operator-84bf588dbb-65m8k   1/1     Running   0          6m59s
cockroachdb-0                         1/1     Running   0          87s
cockroachdb-1                         1/1     Running   0          71s
cockroachdb-2
```

We can connect to the cluster by invoking the `cockroach sql + command from within any of the CockroachDB nodes. For instance, here we connect to +cockroachdb-2` and connect to the cluster:

```
$ kubectl exec -it cockroachdb-2 -- ./cockroach sql --certs-dir cockroach-certs
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.1
# Cluster ID: cb78255b-befa-4447-9fa8-c06b7a353564
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb> show databases;
  database_name | owner
----------------+--------
  defaultdb     | root
  postgres      | root
  system        | node
(3 rows)

Time: 7ms total (execution 6ms / network 1ms)
```

Connecting to the cluster using the method above requires a very high level of access to the cluster. In a production environment, we would probably setup a load balancer in to securely handle incoming requests to the cluster. We'll look at these sorts of configurations in chapter 10.

Meanwhile, to connect to the simple cluster we just created from outside the cluster, we need to first retrieve the client certificates that the operator created when the cluster was established:

```
mkdir certs
kubectl exec cockroachdb-0 -it \
   -- cat cockroach-certs/ca.crt >certs/ca.crt
kubectl exec cockroachdb-0 -it \
   -- cat cockroach-certs/client.root.key >certs/client.root.key
kubectl exec cockroachdb-0 -it \
   -- cat cockroach-certs/client.root.crt >certs/client.root.crt
chmod 600 certs/*
```

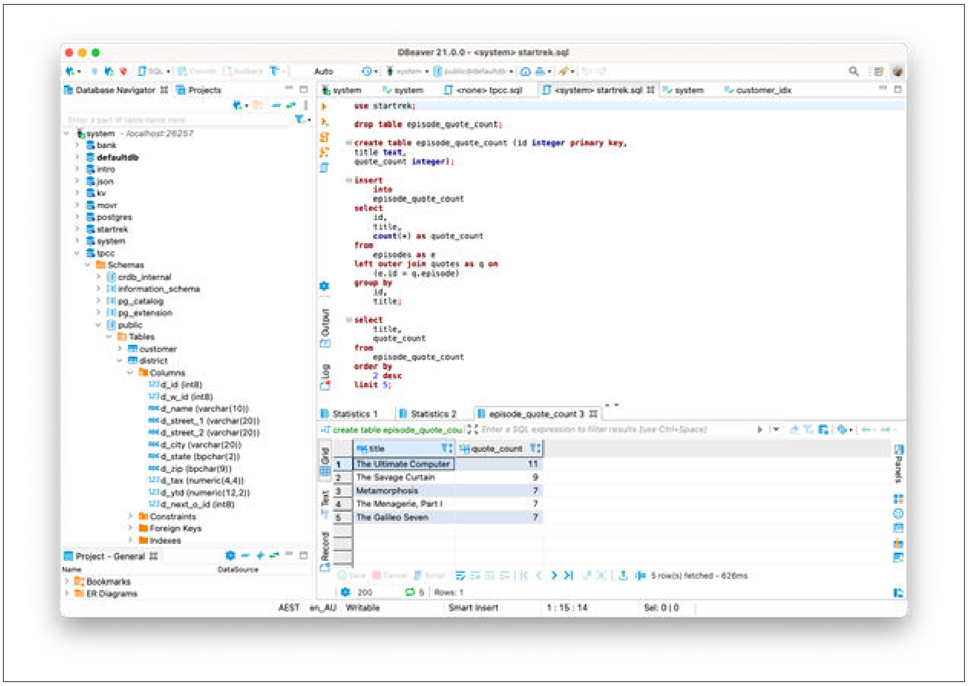
Now we can forward one of the CockroachDB nodes ports to our local machine and connect using the `cockroach sql` command:

```
$ kubectl port-forward services/cockroachdb-public 26257:26257 -n default   &
[1] 22643
$ Forwarding from [::1]:26257 -> 26257

$ cockroach sql --port 26257 --certs-dir certs
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
Handling connection for 26257
# Client version: CockroachDB CCL v21.1.0
# Server version: CockroachDB CCL v21.1.1
# Cluster ID: cb78255b-befa-4447-9fa8-c06b7a353564
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb>
```

Using a GUI client

While some are more than happy to use only a command-line client to interact with a database, some of us prefer a Graphical User Interface (GUI). Many GUI applications for PostgreSQL exist and most of these will work with CockroachDB. However, DBeaver Community edition is a free database GUI that has dedicated support for CockroachDB.

You can get DBeaver from *https://DBeaver.io*. Figure 3-4 shows the DBeaver GUI client.



```sql
use startrek;

drop table episode_quote_count;

create table episode_quote_count (id integer primary key,
title text,
quote_count integer);

insert
    into
    episode_quote_count
select
    id,
    title,
    count(*) as quote_count
from
    episodes as e
left outer join quotes as q on
    (e.id = q.episode)
group by
    id,
    title;

select
    title,
    quote_count
from
    episode_quote_count
order by
    2 desc
limit 5;
```

	title	quote_count
1	The Ultimate Computer	11
2	The Savage Curtain	9
3	Metamorphosis	7
4	The Menagerie, Part I	7
5	The Galileo Seven	7

Figure 3-4. The DBeaver GUI

Exploring CockroachDB

Now that we've got access to a CockroachDB cluster and have the client ready to connect let's take CockroachDB for a drive!

Adding some data

As we say in Australia, "A database without data is like a Pub with no Beer!". So let's get some data into the database so that we have something to look at.

The CockroachDB software includes a number of demonstration databases that you can quickly add to your CockroachDB installation. In some cases, these databases are pre-populated with data; in other cases, you create the schemas then add data afterward.

To initialize the schemas, we use the `cockroach workload init [schema]` command. To run a workload against the schema, we use the `cockroach workload run [schema]` command.

The schemas include:

- **Bank**, which models a set of accounts with currency balances. After the initializing the schema, use `workload run` to generate a workload against the database.
- **Intro**, a simple single-table database.
- **kv**, a simple key-value schema. After the initializing the schema, use run to generate a workload that will be evenly distributed across the cluster.
- **Movr**, a schema for the MovR example application. This schema can be used with the `workload run` command to generate load against the databases.
- **Startrek**, A startrek database, with two tables, `episodes` and `quotes`.
- **Tpcc**, a transaction processing schema for the TPCC standard benchmark. This schema can be used with the `workload run` command to generate load against the databases.
- **Ycsb**, the Yahoo Cloud Serving Benchmark schema. This schema can be used with the `workload run` command to generate load against the databases.

For the `intro` and `startrek` databases, we create the tables and data using the `workload init` command. For instance, in the following example, we create the `startrek` schema and look at some data:

```
root@crdb1 cockroach]# cockroach workload init startrek \
    postgres://localhost:26257?sslmode=disable
I210501 04:29:29.694340 1 workload/workloadsql/dataload.go:140  imported episodes (0s, 79 rows)
I210501 04:29:29.898945 1 workload/workloadsql/dataload.go:140  imported quotes (0s, 200 rows)
[root@crdb1 cockroach]# cockroach sql --insecure
#
# Welcome to the CockroachDB SQL shell.
# All statements must be terminated by a semicolon.
# To exit, type: \q.
#
# Server version: CockroachDB CCL v21.1.1   (same version as client)
# Cluster ID: d070609f-58a7-4aea-aa27-92bc4a1e5406
#
# Enter \? for a brief introduction.
#
root@:26257/defaultdb> show databases;

  database_name | owner
----------------+--------
  defaultdb     | root
  postgres      | root
  startrek      | root
  system        | node
(4 rows)

Time: 2ms total (execution 2ms / network 0ms)

root@:26257/defaultdb> use startrek;
```

```
SET

Time: 1ms total (execution 0ms / network 0ms)

root@:26257/startrek> show tables;
  schema_name | table_name | type  | owner | estimated_row_count
--------------+------------+-------+-------+----------------------
  public      | episodes   | table | root  |                   0
  public      | quotes     | table | root  |                   0
(2 rows)

Time: 56ms total (execution 56ms / network 0ms)

root@:26257/startrek> select * from episodes limit 1;
  id | season | num |     title      | stardate
-----+--------+-----+----------------+-----------
   1 |      1 |   1 | The Man Trap |   1531.1
(1 row)

Time: 1ms total (execution 1ms / network 0ms)
```

In this example, we create the bank schema:

```
$ cockroach workload init bank postgres://localhost:26257?sslmode=disable
I210501 04:31:41.214008 1 imported bank (0s, 1000 rows)
I210501 04:31:41.221478 1 starting 9 splits
```

And then run a workload simulation for 60 seconds:

```
$ cockroach workload run  bank postgres://localhost:26257?sslmode=disable --duration 60s
I210501 04:33:52.340852 1 creating load generator...
I210501 04:33:52.344074 1 reating load generator... done (took 3.220303ms)
_elapsed___errors__ops/sec(inst)___ops/sec(cum)__p50(ms)__p99(ms)_pMax(ms)
    1.0s        0          187.3          187.9     16.8     65.0    121.6 transfer
    2.0s        0          295.0          241.5     11.0     52.4     79.7 transfer
    3.0s        0          260.9          248.0     13.1     54.5     83.9 transfer
    4.0s        0          203.1          236.7     17.8     54.5     79.7 <snip>

_elapsed___errors_____ops(total)___ops/sec(cum)__p50(ms)__p99(ms)_pMax(ms)__result
   60.0s        0          14230          237.2     13.6     65.0    192.9
```

The run command is primarily meant to generated data for load testing purposes but is useful to generate data for query purposes as well.

Databases and tables

As we've seen already, data in a CockroachDB deployment is organized into specific namespaces called databases. Database is a fairly loosely used and overloaded term – it's quite common for a CockroachDB cluster to be referred to as a database or for a database within a cluster to be referred to as a schema. However, in CockroachDB, as in most other SQL databases, a database cluster contains one or more databases.

Within a database, one or more schemas may be defined, though it's common for each database to contain only one schema.

We can list the databases in the cluster using the `SHOW DATABASES` command:

```
root@:26257/defaultdb> show databases;
  database_name | owner
----------------+--------
  bank          | root
  defaultdb     | root
  postgres      | root
  startrek      | root
  system        | node
(5 rows)
```

We can set our current database with the `use` command:

```
root@:26257/defaultdb> use startrek;
SET

Time: 1ms total (execution 0ms / network 0ms)
```

We list tables within a database with the `show tables` command:

```
root@:26257/startrek> show tables;
  schema_name | table_name | type  | owner | estimated_row_count
--------------+------------+-------+-------+----------------------
  public      | episodes   | table | root  |                  79
  public      | quotes     | table | root  |                 200
(2 rows)

Time: 16ms total (execution 16ms / network 0ms)
```

We can describe a table using the `\d` command:

```
root@:26257/startrek> \d quotes;
  column_name | data_type | is_nullable | column_default | g|           indices            |
--------------+-----------+-------------+----------------+--+------------------------------+-
  quote       | STRING    |    true     | NULL           |  | {}                           |
  characters  | STRING    |    true     | NULL           |  | {}                           |
  stardate    | DECIMAL   |    true     | NULL           |  | {}                           |
  episode     | INT8      |    true     | NULL           |  | {quotes_episode_idx}         |
  rowid       | INT8      |    false    | unique_rowid() |  | {primary,quotes_episode_idx} |
(5 rows)

Time: 13ms total (execution 12ms / network 1ms)
```

Issuing SQL

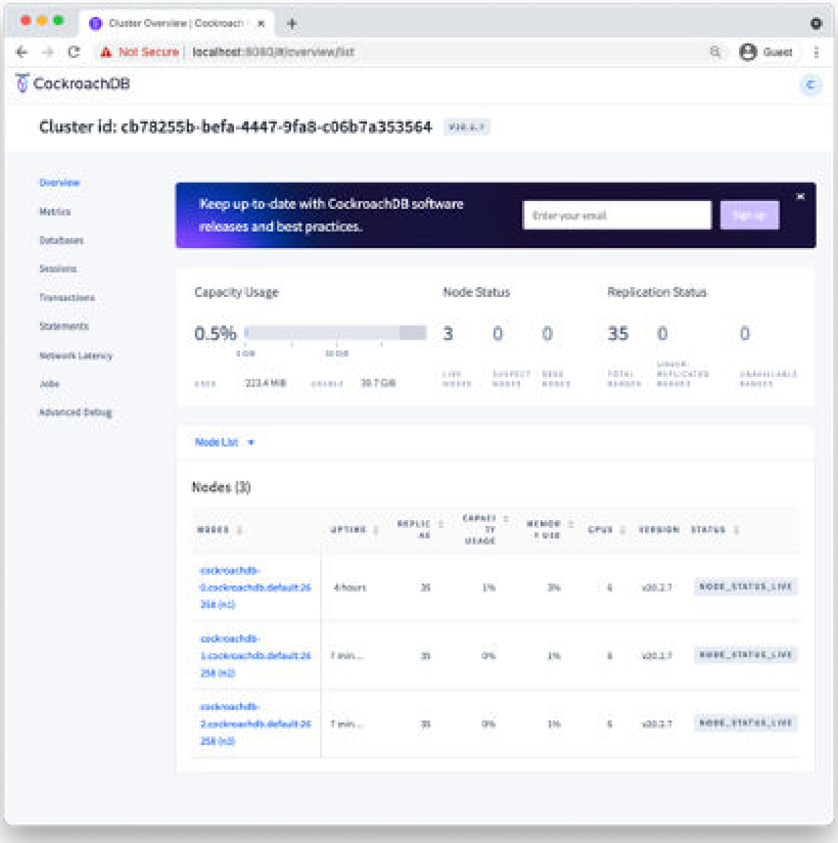
From the CockroachDB client, we can issue any SQL commands for which we are authorized.

Here we create a table within the startrek database, load it with some derived data and issue a query:

```
root@localhost:26257/startrek> CREATE TABLE episode_quote_count (
    id integer PRIMARY KEY,
    title TEXT,
    quote_count integer
);
CREATE TABLE

Time: 135ms total (execution 135ms / network 0ms)

root@localhost:26257/startrek> INSERT INTO episode_quote_count
SELECT id,
    title,
    count(*) AS quote_count
FROM episodes AS e
    LEFT OUTER JOIN quotes AS q ON (e.id = q.episode)
GROUP BY id,
    title;
INSERT 79

Time: 90ms total (execution 90ms / network 0ms)

root@localhost:26257/startrek> SELECT title,
    quote_count
FROM startrek.episode_quote_count
ORDER BY 2 DESC
LIMIT 5;
          title          | quote_count
-------------------------+-------------
  The Ultimate Computer |          11
  The Savage Curtain    |           9
  Metamorphosis         |           7
  The Menagerie, Part I |           7
  The Galileo Seven     |           7
(5 rows)

Time: 3ms total (execution 2ms / network 0ms)
```

The console

The CockroachDB server exposes a web-based client that shows the status of the cluster and useful performance metrics. The webserver is usually exposed on port 8080, though this can be changed using the `--http-addr` setting when starting the server. shows an example of the console, in this case from the Kubernetes cluster that we started earlier in this chapter (we forwarded port 8080 from one of the pods in the cluster).



Cluster id: cb78255b-befa-4447-9fa8-c06b7a353564

Keep up-to-date with CockroachDB software releases and best practices.

Capacity Usage		Node Status			Replication Status		
0.5%		3	0	0	35	0	0
USED 223.4 MiB	USABLE 39.7 GiB	LIVE NODES	SUSPECT NODES	DEAD NODES	TOTAL RANGES	UNDER-REPLICATED RANGES	UNAVAILABLE RANGES

Nodes (3)

NODES	UPTIME	REPLICAS	CAPACITY USAGE	MEMORY USE	CPUS	VERSION	STATUS
cockroachdb-0.cockroachdb.default:26258 (n1)	4 hours	35	1%	3%	6	v20.2.7	NODE_STATUS_LIVE
cockroachdb-1.cockroachdb.default:26258 (n2)	7 min...	35	0%	1%	6	v20.2.7	NODE_STATUS_LIVE
cockroachdb-2.cockroachdb.default:26258 (n3)	7 min...	35	0%	1%	6	v20.2.7	NODE_STATUS_LIVE

Figure 3-5. The CockroachDB console

Working with programming languages

Working with the CockroachDB shell is useful for experimentation, but eventually, most databases interact with application code written in languages such as Javascript, Java, Go or Python.

Because CockroachDB is wire compatible with Postgres, most Postgres compatible drivers will work with CockroachDB. Indeed, there are no CockroachDB-specific drivers on the market because the Postgres drivers work so well. In this section, we'll get you up to speed with "hello world" programs in Java, GoLang, Python and JavaScript that connect to and queries a CockroachDB cluster.

Connecting to CockroachDB from NodeJS

Server-side Javascript using the NodeJS platform is an increasingly popular choice for application development because it allows the same Javascript language to be used for both front-end web presentation code and server-side application logic.

Assuming that you have nodeJS and the Node Package Manager (npm) installed, we'll use the `node-postgres` driver to connect to CockroachDB. We can install this driver with the following command:

```
npm install pg
```

Once pg is installed, then the following example should connect to any CockroachDB database using a connection URI:

```
/// Example of connecting to CockroachDB using NodeJS

const CrClient = require('pg').Client; //load pg client

async function main() {
    try {
        // Check parameters
        if (process.argv.length != 3) {
            console.log('Usage: node helloWorld.js CONNECTION_URI');
            process.exit(1);
        }
        // Establish a connection using the command line URI
        const connectionString = process.argv[2];
        const crClient = new CrClient(connectionString);
        await crClient.connect();

        // Issue a SELECT
        const data = await crClient.query(
            `SELECT CONCAT('Hello from CockroachDB at ',
                        CAST (NOW() as STRING)) as hello`
        );
        // Print out the error message
        console.log(data.rows[0].hello);
    } catch (error) {
        console.log(error.stack);
    }
    // Exit
    process.exit(0);
}

main();
```

This program expects the connection string to be provided as the first argument to the program. The process.argv array contains the full command line including "node" and "helloWorld.js", so the URI actually shows up as the third element in the array.

We then attempt to establish a connection using that connection string, then issue a SELECT statement that retrieves the time as known to the server.

Here we connect to the cockroachCloud server that we setup earlier in this chapter:

```
$ node helloWorld.js "postgres://guy:guysPassword@free-tier6.
             gcp-asia-southeast1.cockroachlabs.cloud:26257/defaultdb?sslmode=verify-full&
             sslrootcert=$HOME/CRDBKeys/cc-ca.crt&options=--cluster=grumpy-orca-56"

Hello from CockroachDB at 2021-05-02 00:17:40.835834+00:00
```

And here we connect to a local CockroachDB running in insecure mode

```
$ node helloWorld.js 'postgres://root@localhost:26257?sslmode=disable'

Hello from CockroachDB at 2021-05-02 00:32:39.125419+00:00
```

Connecting to CockroachDB from Java

Java is the workhorse of millions of applications across all industries and contexts.

In this example, we will use the official PostgresSQL JDBC driver to connect to a CockroachDB server.

Download the JDBC driver from here: *https://jdbc.postgresql.org/download.html* and place it in your CLASSPATH or configure it as a dependency in your IDE.

The following program accepts a URL, username and password as arguments on the command line and connects to the CockroachDB cluster concerned, and issues a SELECT statement:

```
package helloCRDB;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class HelloCRDB {


                public static void main(String[] args) {
            Connection cdb = null;
            try {
                    Class.forName("org.postgresql.Driver");
                    String connectionURL="jdbc:"+args[0];
                    String userName=args[1];
                    String passWord=args[2];

                    cdb = DriverManager.getConnection(connectionURL,userName,passWord);
                    Statement stmt = cdb.createStatement();
                    ResultSet rs = stmt
                         .executeQuery("SELECT CONCAT('Hello from CockroachDB at ',"
                            + "CAST (NOW() as STRING)) AS hello");
                    rs.next();
                    System.out.println(rs.getString("hello"));

            } catch (Exception e) {
                    e.printStackTrace();
                    System.err.println(e.getClass().getName() + ": " + e.getMessage());
                    System.exit(0);
            }


        }

}
```

If we wanted to connect to the CockroachCloud server we created earlier, we'd issue the following command:

```
$ node helloWorld.js "postgres://guy:myPassword@free-tier6.
      gcp-asia-southeast1.cockroachlabs.cloud:26257/defaultdb?sslmode=verify-full&sslrootcert=
      $HOME/CRDBKeys/cc-ca.crt&options=--cluster=grumpy-orca-56"

Hello from CockroachDB at 2021-05-05 15:38:56.691009+10:00
```

And here we connect to a local CockroachDB cluster in insecure mode:

```
$ java -m helloCRDB/helloCRDB.HelloCRDB postgresql://localhost:26257/?sslmode=disable root ‘’
```

Connecting to CockroachDB from Python

Python is a widely used scripting language as well as the tool of choice for many data scientists and data wranglers. In this example We'll use the psycopg python-postgresql package to connect to CockroachDB.

To install the psycopg package, issue the following command:

```
$ pip3 install psycopg2Collecting psycopg2
  Using cached psycopg2-2.8.6.tar.gz (383 kB)
Building wheels for collected packages: psycopg2
  Building wheel for psycopg2 (setup.py) ... done
  Created wheel for psycopg2: sha256=0c386372a9a001b321...
  Stored in directory: /User...
Successfully built psycopg2
Installing collected packages: psycopg2
Successfully installed psycopg2-2.8.6
```

Now the following short program will connect to CockroachDB using a URL provided on the command line and issue a SELECT statement:

```
#!/usr/bin/env python3

import psycopg2
import sys
```

```
def main():

  if ((len(sys.argv)) !=2):
    sys.exit("Error:No URL provided on command line")
  uri=sys.argv[1]

  conn = psycopg2.connect(uri)
  with conn.cursor() as cur:
    cur.execute("""SELECT CONCAT('Hello from CockroachDB at ',
                  CAST (NOW() as STRING))""")
    data=cur.fetchone()
    print("%s" % data[0])

main()
```

Here we connect to a local CockroachDB cluster running in insecure mode:

```
$ python helloCRDB.py 'postgres://root@localhost:26257?sslmode=disable'

Hello from CockroachDB at 2021-05-02 02:33:00.755359+00:00
```

And here we connect to the CockroachCloud database we established earlier in the chapter[1]:

```
$ python helloCRDB.py 'postgres://guy:guysPassword@free-tier6.
         gcp-asia-southeast1.cockroachlabs.cloud:26257/defaultdb?sslmode=verify-full&
         sslrootcert=/Users/guyharrison/CRDBKeys/cc-ca.crt&options=--cluster%3dgrumpy-orca-56'

Hello from CockroachDB at 2021-05-02 02:39:55.859734+00:00
```

Connecting to CockroachDB from Go

The GO language is one of the fastest-growing programming languages, which offers high performance, modern programming paradigms and a low footprint. Much of the CockroachDB database platform is written in Go, so Go is a great choice for CockroachDB development.

In this example, we are going to use the pgx PostgreSQL driver for Go to connect to a CockroachDB cluster. First, we need to install the driver:

```
$ go env -w GO111MODULE=auto
$ go get github.com/jackc/pgx
go: downloading github.com/jackc/pgx v3.6.2+incompatible
go: downloading golang.org/x/text v0.3.6
go: downloading golang.org/x/crypto v0.0.0-20210421170649-83a5a9bb288b
```

1 Note that because of limitations in the psycopg2 driver, we need to replace the final "=" in the URL with "%3d. Instead of + cluster=grumpy-orca-56+ we use + cluster%3dgrumpy-orca-56+

This short program connects to CockroachDB using the URL provided on the command line, and issues a SELECT statement:

```
package main

import (
        "context"
        "fmt"
        "os"

        "github.com/jackc/pgx"
)

func main() {
        if len(os.Args) < 2 {
                fmt.Fprintln(os.Stderr, "Missing URL argument")
                os.Exit(1)
        }
        uri := os.Args[1]
        conn, err := pgx.Connect(context.Background(), uri)
        if err != nil {
                fmt.Fprintf(os.Stderr, "Unable to connect to database: %v\n", err)
                os.Exit(1)
        }
        var text string
        err = conn.QueryRow(context.Background(),
                "SELECT CONCAT('Hello from CockroachDB at ',
             CAST (NOW() as STRING))").Scan(&text)
        if err != nil {
                fmt.Fprintf(os.Stderr, "QueryRow failed: %v\n", err)
                os.Exit(1)
        }

        fmt.Println(text)
}
```

Here, we connect to the CockroachCloud cluster we created earlier in the chapter:

```
  go run helloCRDB.go "postgres://guy:guysPassword@free-tier6.
      gcp-asia-southeast1.cockroachlabs.cloud:26257/defaultdb?sslmode=verify-full&
      sslrootcert=$HOME/CRDBKeys/cc-ca.crt&options=--cluster=grumpy-orca-56"

Hello from CockroachDB at 2021-05-02 02:24:13.930662+00:00
```

And here we run the program to connect to a local CockroachDB cluster in insecure mode:

```
  go run helloCRDB.go 'postgres://root@localhost:26257?sslmode=disable'

Hello from CockroachDB at 2021-05-02 02:21:59.179171+00:00
```

Summary

In this chapter, we've shown you how to install CockroachDB software on a local computer, how to create a CockroachDB cluster in a variety of configurations and how to work with CockroachDB from the command line or a programming language.

It's easy to install CockroachDB software on a desktop and, in most cases, necessary if you want to work with a CockroachDB server from the command line. You can also install CockroachDB software using Docker or Kubernetes.

While a single-node test server can be a useful tool for learning CockroachDB, the CockroachCloud offers a free 5GB server that provides backup and security. You can also install CockroachDB in a Kubernetes cluster to experiment with a full cluster in a local environment.

Because CockroachDB is PostgreSQL compatible, you can use any Postgres compatible driver to connect to CockroachDB. We also provided simple examples of connecting to CockroachDB using the PostgreSQL drivers for Java, Python, GoLang and NodeJS.

[[Ch04 – CockroachDB SQL]]

CHAPTER 4

COCKROACHDB SQL

The language of CockroachDB is SQL. While there are some command-line utilities, all interactions between an application and the database are mediated by SQL language commands.

SQL is a rich language with a long history – we touched upon some of that history in Chapter 1. A full definition of all SQL language features would require a book in its own right and would be almost instantly out of date since the SQL language evolves with each release.

Therefore, this chapter aims to provide you with a broad overview of the SQL language used in CockroachDB without attempting to be a complete reference. We'll take a task-oriented approach to the SQL language, covering the most common SQL language tasks with particular reference to unique features of the CockroachDB SQL implementation.

A complete reference for the CockroachDB SQL language can be found in the CockroachDB documentation set[1]. A broader review of the SQL language can be found in the O'Reilly book "SQL in a Nutshell".

We'll mainly use the MOVR sample dataset in this chapter to illustrate various SQL language features. We learned how to install sample data in Chapter 2; to recap, we create the MOVR database by issuing the command + cockroach workload init movr + command:

```
$ cockroach workload init movr
I210510 01:31 1  imported users (0s, 50 rows)
I210510 01:31 2  imported vehicles (0s, 15 rows)
I210510 01:31 3  imported rides (0s, 500 rows)
```

1 *https://www.cockroachlabs.com/docs/stable/sql-feature-support.html*

```
I210510 01:31 4  imported vehicle_location_histories (0s, 1000 rows)
I210510 01:31 5  imported promo_codes (0s, 1000 rows)
I210510 01:31 1 6  starting 8 splits
I210510 01:31 1 7  starting 8 splits
I210510 01:31 1 8  starting 8 splits
```

You may also wish to run the command `cockroach workload run movr` to generate ride data against the MOVR base tables.

SQL LANGUAGE COMPATIBILITY

CockroachDB is broadly compatible with the PostgreSQL implementation of the SQL:2016 standard. The SQL:2016 standard contains a number of independent modules, and no major database implements all of the standard. However, the PostgreSQL implementation of SQL is arguably as close to "standard" as exists in the database community.

CockroachDB varies from PostgreSQL in a couple of areas:

- CockroachDB does not currently support stored procedures, events or triggers. In PostgreSQL, these stored procedures allow for the execution of program logic within the database server, either on-demand or in response to some triggering event.
- CockroachDB does not currently support User Defined Functions
- CockroachDB does not support PostgreSQL XML functions.
- CockroachDB does not support PostgreSQL FullText indexes and functions.

QUERYING DATA WITH SELECT

Although we need to create and populate tables before querying them, it is logical to start with the SELECT statement since many features of the SELECT statement appear in other types of SQL – subqueries in UPDATEs for instance – and for data scientists and analysts, the SELECT statement is often the only SQL statement they ever need to learn.

The SELECT statement (Figure 4-1) is the workhorse of relational query and has a complex and rich syntax. The CockroachDB SELECT statement implements the standard features of the standard SELECT, with just a few CockroachDB-specific features.

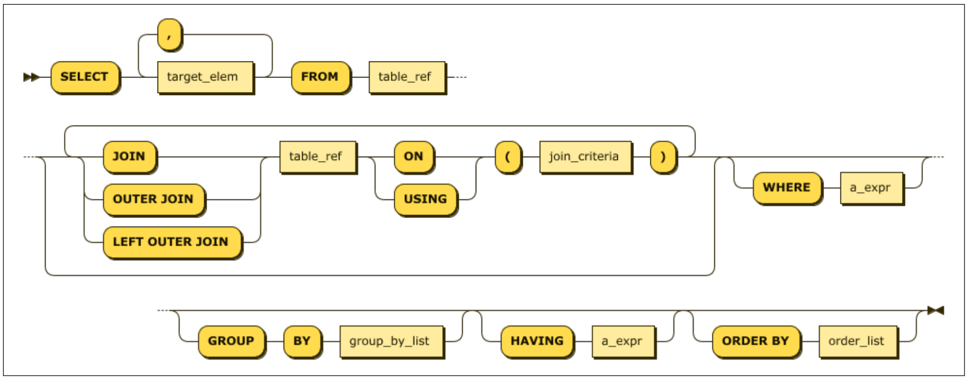


Figure 4-1. Select Statement

In the following sections, we'll examine each of the major elements of the SELECT statement as well as the functions and operators that can be included in a SELECT statement. ==== The SELECT list

A very simple SQL statement consists of nothing but a SELECT statement together with scalar expressions. For instance:

```
SELECT CONCAT('Hello from CockroachDB at ',
              CAST (NOW() as STRING)) as hello
```

The SELECT list includes a comma-separated list of expressions that can contain combinations of constants, functions, and operators. The CockroachDB SQL language supports all the familiar SQL operators. A complete list of functions and operators can be found in the CockroachDB documentation set[2].

The FROM clause

The FROM clause is the primary method of attaching table data to the SELECT statement. In its most simple incarnation, all rows and columns from a table can be fetched:

```
SELECT * FROM rides
```

Table names may be aliased using the `AS` clause or simply by following the table name with an alias. That alias can then be used anywhere in the query to refer to the table. Column names can also be aliased. For instance, the following are all equivalent:

```
SELECT name FROM users;
SELECT u.name FROM users u;
SELECT users.name FROM users u;
```

2 *https://www.cockroachlabs.com/docs/stable/functions-and-operators.html*

```
SELECT users.name AS user_name FROM users;
SELECT u.name FROM users AS u;
```

JOINS

Joins allow the results from two or more tables to be merged based on some common column values.

The **inner join** is the default join operation. In this join, rows from one table are joined to rows from another table based on some common ("key") values. Rows that have no match in both tables are not included in the results. For instance, the following query links vehicle and ride information in the MOVR database:

```
SELECT v.id,v.ext,r.start_time r.start_address
  FROM vehicles v
  LEFT OUTER JOIN rides r
    ON (r.vehicle_id=v.id);
```

Note that a vehicle that had not been involved in a ride would not be included in the result set.

The ON clause specifies the conditions which join the two tables – in the above query, the columns `vehicle_id` in the `rider` table were matched with the `id` column in the vehicles table. If the join is on an identically named column in both tables, then the USING clause provides a handy shortcut. Here we join `users` and `user_ride_counts` using the common `name` column:

```
SELECT *
  FROM users u
  JOIN user_ride_counts urc
 USING (name)
```

The **Outer join** allows rows to be included even if they have no match in the other table. Rows that are not found in the outer join table are represented by NULL values. LEFT and RIGHT determine which table may have missing values. For instance, the following query prints all the users in the `users` table, even if some are not associated with a promo code:

```
SELECT u.name , upc.code
  FROM USERS u
  LEFT OUTER JOIN user_promo_codes upc
    ON (u.id=upc.user_id);
```

The RIGHT anti-join reverses the outer join. So, this query is identical to the previous query since the `users` table is now the "right" table in the join:

```
SELECT DISTINCT u.name , upc.code
  FROM user_promo_codes upc
  RIGHT OUTER JOIN USERS u
    ON (u.id=upc.user_id);
```

Anti-joins

It is often required to select all rows from a table that do not have a matching row in some other result set. This is called an anti-join, and while there is no SQL syntax for this concept, it is typically implemented using a subquery and the IN or EXISTS clause. The query planner recognizes this pattern and executes the query using a version of the JOIN machinery. The following examples illustrate the anti-join using the EXISTS and IN operators.

Each example selects employees who are not also customers.

```
SELECT *
  FROM users
 WHERE id NOT IN
       (SELECT id FROM employees)
```

This query returns the same results but using a correlated sub-query (we'll discuss subqueries in more detail in an upcoming section):

```
SELECT *
   FROM users u
  WHERE NOT EXISTS
        (SELECT id
           FROM employees e
          WHERE e.id=u.id)
```

CROSS JOINS

CROSS JOIN indicates that every row in the left table should be joined to every row in the right table. Usually, this is a recipe for disaster unless one of the tables has only one row or is a laterally correlated subquery (see the section on correlated subqueries later in this chapter).

Set operations

SQL implements a number of operations that deal directly with result sets. These operations collectively referred to as "set operations" allow result sets to be concatenated, subtracted, or overlaid.

The most common of these operations is the UNION operator, which returns the sum of two result sets. By default, duplicates in each result set are eliminated. By contrast, the UNION ALL operation will return the sum of the two result sets, including any duplicates. The following example returns a list of customers and employees. Employees who are also customers will only be listed once:

```
SELECT name, address
  FROM customers
 UNION
```

```
SELECT name,address
  FROM employees;
```

INTERSECT returns those rows which are in both result sets. This query returns customers who are also employees:

```
SELECT name, address
  FROM customers
 INTERSECT
SELECT name,address
  FROM employees;
```

EXCEPT returns rows in the first result set which are not present in the second. This query returns customers who are not also employees:

```
SELECT name, address
  FROM customers
 EXCEPT
SELECT name,address
  FROM employees;
```

All set operations require that the component queries return the same number of columns and that those columns are of a compatible data type.

Group operations

Aggregate operations allow for summary information to be generated, typically upon groupings of rows. Rows can be grouped using the GROUP BY operator. If this is done, the select list must consist only of columns contained within the GROUP BY clause and aggregate functions.

The most common aggregate functions are shown in Table 4-1.

Table 4-1. Aggregate Functions

AVG	Calculate the average value for the group.
COUNT	Return the number of rows in the group.
MAX	Return the maximum value in the group.
MIN	Return the minimum value in the group.
STDDEV	Return the standard deviation for the group.
SUM	-Return the total of all values for the group.

The following example generates summary ride information for each city:

```
SELECT u.city,SUM(urc.rides),AVG(urc.rides),max(urc.rides)
  FROM users u
  JOIN user_ride_counts urc
 USING (name)
 GROUP BY u.city
```

Subqueries

A subquery is a SELECT statement that occurs within another SQL statement. Such a "nested" SELECT statement can be used in a wide variety of SQL contexts, including SELECT, DELETE, UPDATE, and INSERT statements.

The following statement uses a subquery to count the number of rides that share the maximum ride length:

```
SELECT COUNT(*) FROM rides
 WHERE (end_time-start_time)=
  (SELECT MAX(end_time-start_time) FROM rides );
```

Subqueries may also be used in the FROM clause wherever a table or view definition could appear. This query generates a result which compares each ride with the average ride duration for the city:

```
SELECT id, city,(end_time-start_time) ride_duration, avg_ride_duration
FROM rides
JOIN (SELECT city, AVG(end_time-start_time) avg_ride_duration
                FROM rides
                GROUP BY city)
        USING(city) ;
```

Correlated subquery

A correlated subquery is one in which the subquery refers to values in the parent query or operation. The subquery returns a potentially different result for each row in the parent result set. We saw an example of a correlated sub-query when performing an "anti-join" earlier in the chapter.

```
SELECT *
   FROM users u
  WHERE NOT EXISTS
        (SELECT id
           FROM employees e
          WHERE e.id=u.id)
```

Subqueries can often be used to perform an operation that is functionally equivalent to a join. In many cases, the query optimizer will transform these statements to joins so as to streamline the optimization process. ==== Lateral subquery

When a subquery is used in a join, the LATERAL keyword indicates that the subquery may access columns generated in preceding FROM table expressions. For instance, in the following query, the LATERAL keyword allows the subquery to access columns from the USERS table:

```
  SELECT name, address, start_time
   FROM USERS CROSS JOIN
        LATERAL (SELECT *
                   FROM rides
                  WHERE rides.start_address = users.address ) r;
```

This example is a bit contrived, and clearly, we could construct a simple join that performed this query more naturally. Where LATERAL joins really shine is in allowing subqueries to access computed columns in other subqueries within a FROM clause. See the CockroachDB documentation[3] for a more serious example of lateral subqueries.

The WHERE clause

The WHERE clause is common to SELECT, UPDATE, and DELETE statements. It specifies a set of logical conditions that must evaluate to TRUE for all rows to be returned or processed by the SQL statement concerned.

Common Table expressions

SQL statements with a lot of subqueries can be hard to read and maintain, especially if the same subquery is needed in multiple contexts within the query. For this reason, SQL supports **Common Table Expressions** using the WITH clause. Figure 4-2 shows the syntax of a Common Table Expression.

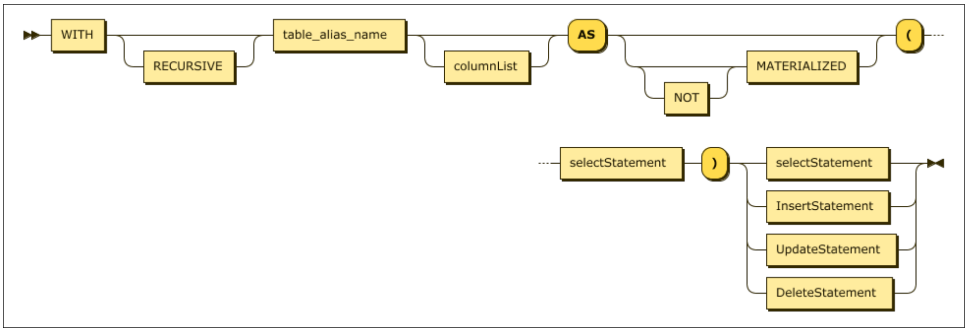


Figure 4-2. Common Table Expression

In its simplest form, a Common Table Expression is simply a named query block that can be applied wherever a table expression can be used. For instance, here we create a Common Table Expression `riderRevenue` with the WITH clause, then refer to it in the FROM clause of the main query:

```
WITH riderRevenue AS (
          SELECT u.id, SUM(r.revenue) AS sumRevenue
            FROM rides r JOIN "users" u
            ON (r.rider_id=u.id)
```

3 *https://www.cockroachlabs.com/blog/using-lateral-joins-in-the-cockroachdb-20-1-alpha/*

```
              GROUP BY u.id)
    SELECT * FROM "users" u2
             JOIN riderRevenue rr USING (id)
     ORDER BY sumrevenue DESC
```

The RECURSIVE clause allows the Common Table Expression to refer to itself, potentially allowing for a query to return an arbitrarily high (or even infinite) set of results. For instance, if the `employees` table contained a `manager_id` column which referred to the 'manager's row in the same table, then we could print a hierarchy of employees and managers as follows:

```
    WITH RECURSIVE employeeMgr AS (
      SELECT id,manager_id, name , NULL AS manager_name, 1 AS level
        FROM employees managers
       WHERE manager_id IS NULL
      UNION ALL
      SELECT subordinates.id,subordinates.manager_id,
             subordinates.name, managers.name ,managers.LEVEL+1
        FROM employeeMgr managers
        JOIN employees subordinates
          ON (subordinates.manager_id=managers.id)
    )
    SELECT * FROM employeeMgr
```

The MATERIALIZED clause forces CockroachDB to store the results of the Common Table Expression as a temporary table rather than re-executing it on each occurrence. This can be useful if the Common Table Expression is referenced multiple times in the query.

ORDER BY

The ORDER BY clause allows query results to be returned in sorted order. Figure 4-3 shows the ORDER BY syntax.

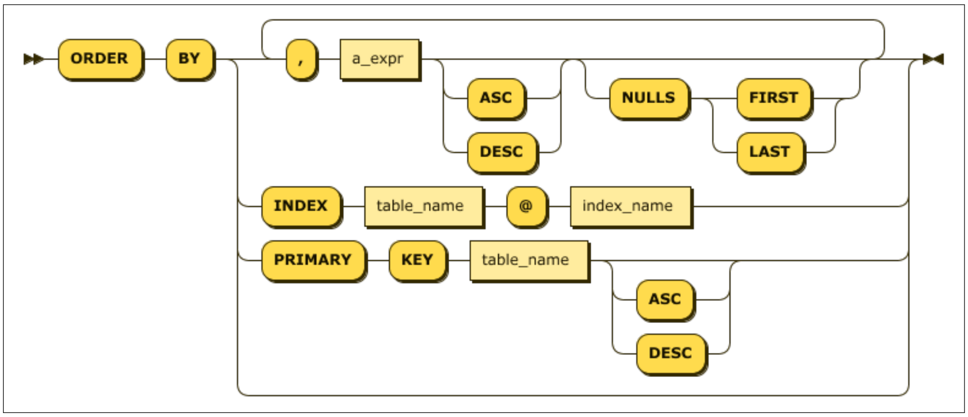


Figure 4-3. Order By

In the simplest form, ORDER BY takes one or more column expressions or column numbers from the SELECT list.

In this example, we sort by column numbers:

```
SELECT city,start_time, (end_time-start_time) duration
   FROM rides r
  ORDER BY 1,3 DESC
```

And in this case by column expressions:

```
SELECT city,start_time, (end_time-start_time) duration
   FROM rides r
  ORDER BY city,(end_time-start_time)  DESC
```

You can also order by an index. In the following example, rows will be ordered by city and start_time, since those are the columns specified in the index:

```
CREATE INDEX rides_start_time ON rides (city ,start_time);


SELECT city,start_time, (end_time-start_time) duration
  FROM rides
 ORDER BY INDEX rides@rides_start_time;
```

The use of ORDER BY INDEX guarantees that the index will be used to directly return rows in sorted order, rather than having to perform a sort operation on rows returned otherwise. See Chapter 8 for more advice on optimizing statements that contain an ORDER BY.

Window functions

Window functions are functions that operate over a subset – a "window" of the complete set of the results. Figure 4-4 shows the syntax of a Window function.

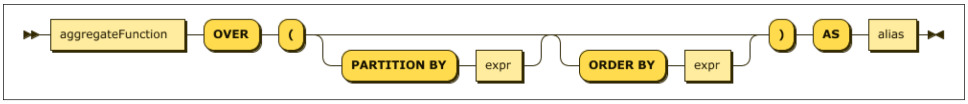


Figure 4-4. Window Function Syntax

PARTITION BY and ORDER BY create a sort of "virtual table" that the function works with. For instance, this query lists the top ten rides in terms of revenue, with the percentage of the total revenue and city revenue displayed:

```
SELECT city, r.start_time ,revenue,
       revenue*100/SUM(revenue) OVER () AS pct_total_revenue,
       revenue*100/SUM(revenue) OVER (PARTITION BY city) AS pct_city_revenue
  FROM rides r
 ORDER BY 5 DESC
 LIMIT 10
```

There are some aggregation functions that are specific to Windowing functions. RANK() ranks the existing row within the relevant window, and DENSE_RANK() does the same while allowing no “"missing"” ranks. LEAD and LAG provide access to functions in adjacent partitions.

For instance, this query returns the top 10 rides, with each ‘ride’s overall rank and rank within the city displayed:

```
SELECT city, r.start_time ,revenue,
       RANK() OVER
           (ORDER BY revenue DESC) AS total_revenue_rank,
       RANK() OVER
            (PARTITION BY city ORDER BY revenue DESC) AS city_revenue_rank
  FROM rides r
 ORDER BY revenue DESC
 LIMIT 10;
```

Other SELECT clauses

The LIMIT clause limits the number of rows returned by a SELECT while the OFFSET clause “jumps ahead” a certain number of rows. This can be handy to paginate through a result set though it is almost always more efficient to use a filter condition to navigate to the next subset of results since otherwise each request will need to reread and discard an increasing number of rows.

CockroachDB arrays

The ARRAY type allows a column to a one dimentional array of elements, each of which shares a common data type. We’ll talk about arrays in the context of data modelling in the next chapter. Although they can be useful, they are strictly speaking a violation of the relational model and should be used carefully.

An ARRAY variable is defined by adding “[]” or the word “ARRAY” to the datatype of a column. For instance:

```
CREATE TABLE a (b STRING[]);
CREATE TABLE c (d INT ARRAY);
```

The ARRAY function allows us to insert multiple items into the ARRAY:

```
INSERT INTO a VALUES (ARRAY['sky', 'road', 'car']);
SELECT * FROM a;
        b
------------------
  {sky,road,car}
```

We can access an individual element of an array with the familiar array element notation:

```
SELECT arrayColumn[2] FROM arrayTable;
  arraycolumn
---------------
  Road
```

The “@>” operator can be used to find arrays that contain one or more elements:

```
SELECT * FROM arrayTable WHERE arrayColumn @>ARRAY['road'];
   arraycolumn
------------------
  {sky,road,car}
```

We can add elements to an existing array using the ARRAY_APPEND function and remove elements using ARRAY_REMOVE.

```
UPDATE  arrayTable
   SET arrayColumn=array_append(arrayColumn,'cat')
  WHERE arrayColumn @>ARRAY['car']
 RETURNING arrayColumn ;

     arraycolumn
----------------------
  {sky,road,car,cat}

 UPDATE  arrayTable
   SET arrayColumn=array_remove(arrayColumn,'car')
  WHERE arrayColumn @>ARRAY['car']
  RETURNING arrayColumn;

   arraycolumn
------------------
  {sky,road,cat}
```

Finally, the UNNEST function transforms an array into a tabular result – one row for each element of the array. This can be used to “join” the contents of an array with data held in relational form elsewhere in the database. We show an example of this in the next chapter:

```
SELECT unnest(arrayColumn) FROM arrayTable;
  unnest
----------
  sky
  road
  cat
```

Working with JSON

The JSONB datatype allows us to store JSON documents into a column, and CockroachDB provides operators and functions to help us work with JSON.

For these examples, we've created a table with a primary key CUSTOMERID and all data in a JSONB column JSONDATA. We can use the `jsonb_pretty` function to retrieve the JSON in a nicely formatted manner:

```
root@:26257/json> SELECT jsonb_pretty(jsondata) FROM customersjson WHERE customerid=1;
                     jsonb_pretty
-------------------------------------------------------
  {
      "Address": "1913 Hanoi Way",
      "City": "Sasebo",
      "Country": "Japan",
      "District": "Nagasaki",
      "FirstName": "MARY",
      "LastName": "Smith",
      "Phone": 886780309,
      "_id": "5a0518aa5a4e1c8bf9a53761",
      "dateOfBirth": "1982-02-20T13:00:00.000Z",
      "dob": "1982-02-20T13:00:00.000Z",
      "randValue": 0.47025846594884335,
      "views": [
          {
              "filmId": 611,
              "title": "MUSKETEERS WAIT",
              "viewDate": "2013-03-02T05:26:17.645Z"
          },
          {
              "filmId": 308,
              "title": "FERRIS MOTHER",
              "viewDate": "2015-07-05T20:06:58.891Z"
          },
          {
              "filmId": 159,
              "title": "CLOSER BANG",
              "viewDate": "2012-08-04T19:31:51.698Z"
          },
          /* Some data removed */
      ]
  }
```

Each JSON document contains some top level attributes and a nested array of document that contains details of films that they have streamed.

We can reference specific JSON attributes in the SELECT clause using the "→" operator:

```
root@:26257/json> SELECT jsondata->'City' AS City
  FROM customersjson WHERE customerid=1;

    city
------------
  "Sasebo"
```

The "→>" operator is similar, but returns the data formatted as text, not JSON.

If we want to search inside a JSONB column, we can use the "@>" operator:

```
SELECT COUNT(*) FROM customersjson
 WHERE jsondata @> '{"City": "London"}';

  count
---------
      3
```

We often get same result using the '→>' operator:

```
SELECT COUNT(*) FROM customersjson
 WHERE jsondata->>'City' = 'London';

  count
---------
      3
```

We can interrogate the structure of the JSON document using the JSONB_EACH and JSONB_OBJECT_KEYS functions. JSONB_EACH returns one row per attribute in the JSON document, while JSONB_OBJECT_KEYS returns just the attribute keys. This is useful if you don't know what is stored inside the JSONB column.

JSONB_ARRAY_ELEMENTS returns one row for each element in a JSON array. For instance, here we expand out the VIEWS array for a specific customer, counting the number of movies that they have seen:

```
SELECT COUNT(jsonb_array_elements(jsondata->'views'))
  FROM customersjson
 WHERE customerid =1;

  count
---------
     37
(1 row)
```

Summary of SELECT

The SELECT statement is probably the most widely used statement in database programming and offers a very wide range of functionality. Even after decades of working in the field, the three of us don't know every nuance SELECT functionality. However, here we've tried to provide you with the most important aspects of the language. For more depth, view the CockroachDB documentation set[4].

4 *https://www.cockroachlabs.com/docs/stable/postgresql-compatibility.html*

Although some database professionals use SELECT almost exclusively, the majority will be creating and manipulating data as well. In the following sections we'll look at the language features that support those activities.

Other SELECT clauses

The LIMIT clause limits the number of rows returned by a SELECT while the OFF-SET clause "jumps ahead" a certain number of rows. This can be handy to paginate through a result set though it is almost always more efficient to use a filter condition to navigate to the next subset of results since otherwise, each request will need to reread and discard an increasing number of rows.

Now that we've seen how to query data in a CockroachDB database using SELECT, let's look at how to create and manipulate that data by Defining tables and indexes using Data Definition Language (DDL) and Data Manipulation Language (DML).

CREATING TABLES AND INDEXES

In a relational database, data can only be added to pre-defined tables. These tables are created by the CREATE TABLE statement. Indexes can be created to enforce unique constraints or to provide a fast access path to the data. Indexes can be defined within the CREATE TABLE statement or by a separate CREATE INDEX statement.

The structure of a database schema forms a critical constraint on database performance and also on the maintainability and utility of the database. We'll discuss the key considerations for database design in Chapter 5. For now, let's create a few simple tables.

We use CREATE TABLE to create a table within a database. Figure 4-5 provides a simplified syntax for the CREATE TABLE statement.

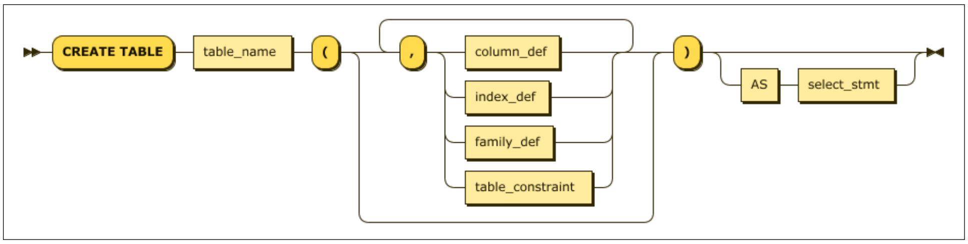


Figure 4-5. Create Table Statement

A very simple CREATE TABLE is shown below. It creates a table `mytable` with a single column `mycolumn`. The `mycolumn` column can store only integer values.

```
CREATE TABLE mytable
(
        mycolumn int
    )
```

The CREATE TABLE statement must define the columns that occur within the table and can optionally define indexes, column families, constraints, and partitions associated with the table. For instance, the CREATE TABLE statement for the `rides` table in the `movr` database would look something like this:

```
    CREATE TABLE public.rides (
            id UUID NOT NULL,
            city VARCHAR NOT NULL,
            vehicle_city VARCHAR NULL,
            rider_id UUID NULL,
            vehicle_id UUID NULL,
            start_address VARCHAR NULL,
            end_address VARCHAR NULL,
            start_time TIMESTAMP NULL,
            end_time TIMESTAMP NULL,
            revenue DECIMAL(10,2) NULL,
            CONSTRAINT "primary" PRIMARY KEY (city ASC, id ASC),
            CONSTRAINT fk_city_ref_users
                    FOREIGN KEY (city, rider_id)
                    REFERENCES public.users(city, id),
            CONSTRAINT fk_vehicle_city_ref_vehicles
                     FOREIGN KEY (vehicle_city, vehicle_id)
                     REFERENCES public.vehicles(city, id),
            INDEX rides_auto_index_fk_city_ref_users
                  (city ASC, rider_id ASC),
            INDEX rides_auto_index_fk_vehicle_city_ref_vehicles
              vehicle_city ASC, vehicle_id ASC),
            CONSTRAINT check_vehicle_city_city
               CHECK (vehicle_city = city)
    );
```

This CREATE TABLE statement specified additional columns, their nullability, primary and foreign keys, indexes, and constraints upon table values.

The relevent clauses in Figure 4-5 are listed in Table 4-2.

Table 4-2. Create Table options

column_def	The definition of a column. This includes the column name, datatype, and nullability. Constraints specific to the column can also be included here, though it's better practice to list all constraints separately.
Index_def	Definition of an index to be created on the table. Same as CREATE INDEX but without the leading `CREATE` verb.
table_contraint	A constraint on the table, such as PRIMARY KEY, FOREIGN KEY, or CHECK. See below for constraint syntax.
family_def	Assigns columns to a column family. See Chapter 2 for more information about column families.

Let's now look at each of these CREATE TABLE options. ==== Column Definitions

A column definition consists of a column name, datatype, nullability status, default value, and possibly column-level constraint definitions. At a minimum, the name and data type must be specified. Figure 4-6 shows the syntax for a column definition.

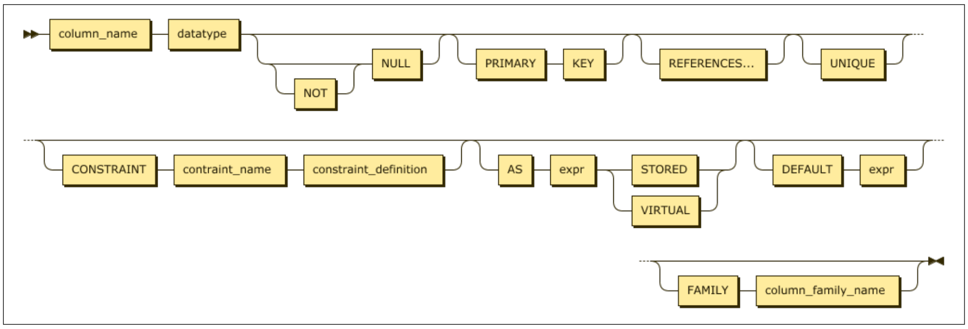


Figure 4-6. Column Definition

Although constraints may be specified directly against column definitions, they may also be independently listed below the column definitions. Many practitioners prefer to list the constraints separately in this manner because it allows all constraints, including multi-column constraints, to be located together.

Computed Columns

CockroachDB allows tables to include **computed columns** that in some other databases would require a view definition.

```
column_name AS expression [STORED|VIRTUAL]
```

A VIRTUAL computed column is evaluated whenever it is referenced. A STORED expression is stored in the database when created and need not always be recomputed.

For instance, this table definition has the firstName and lastName concatenated into a fullName column:

```
CREATE TABLE people
 (
    id INT PRIMARY KEY,
    firstName VARCHAR NOT NULL,
    lastName VARCHAR NOT NULL,
    dateOfBirth DATE NOT NULL,
    fullName STRING AS (CONCAT(firstName,' ',lastName) ) STORED,

 )
```

Computed columns cannot be context-dependent. That is, the computed value must not change over time or be otherwise non-deterministic. For instance, the computed column would not work since the `age` column would be static rather than recalculated every time. While it might be nice to stop aging in real life, we probably want the age column to increase as time goes on.

```
CREATE TABLE people
 (
    id INT PRIMARY KEY,
    firstName VARCHAR NOT NULL,
    lastName VARCHAR NOT NULL,
    dateOfBirth timestamp NOT NULL,
    fullName STRING AS (CONCAT(firstName,' ',lastName) ) STORED,
    age int AS (now()-dateOfBirth) STORED
 );
```

Datatypes

The base CockroachDB datatypes are shown in Table 4-3 . See *https://www.cockroachlabs.com/docs/stable/data-types.html* for more details.

Table 4-3. CockroachDBDatatypes

Type	Description	Example
ARRAY	A 1-dimensional, 1-indexed, homogeneous array of any non-array data type.	{"sky","road","car"}
BIT	A string of binary digits (bits).	B'10010101'
BOOL	A Boolean value.	true
BYTES	A string of binary characters.	b'\141\061\142\062\143\063'
COLLATE	The COLLATE feature lets you sort STRING values according to language- and country-specific rules, known as collations.	*a1b2c3* COLLATE en
DATE	A date.	DATE *2016-01-25*
ENUM	New in v20.2: A user-defined data type comprised of a set of static values.	ENUM (*club, 'diamond*, *heart*, *spade*)
DECIMAL	An exact, fixed-point number.	1.2345
FLOAT	A 64-bit, inexact, floating-point number.	1.2345
INET	An IPv4 or IPv6 address.	192.168.0.1
INT	A signed integer, up to 64 bits.	12345
INTERVAL	A span of time.	INTERVAL *2h30m30s*
JSONB	JSON (JavaScript Object Notation) data.	*{"first_name": "Lola", "last_name": "Dog", "location": "NYC", "online" : true, "friends" : 547}*
SERIAL	A pseudo-type that combines an integer type with a DEFAULT expression.	148591304110702593
STRING	A string of Unicode characters.	*a1b2c3*
TIME TIMETZ	TIME stores a time of day in UTC. TIMETZ converts TIME values with a specified time zone offset from UTC.	TIME *01:23:45.123456* TIMETZ *01:23:45.123456-5:00*

Type	Description	Example
TIMESTAMP TIMESTAMPTZ	TIMESTAMP stores a date and time pairing in UTC. TIMESTAMPTZ converts TIMESTAMP values with a specified time zone offset from UTC.	TIMESTAMP *2016-01-25 10:10:10* TIMESTAMPTZ *2016-01-25 10:10:10-05:00*
UUID	A 128-bit hexadecimal value.	7f9c24e8-3b12-4fef-91e0-56a2d5a246ec

Note that other datatype names may be aliased against these CockroachDB base types. For instance, the PostgreSQL types BIGINT and SMALLINT are aliased against the CockroachDB type INT.

In CockroachDB, datatypes may be cast – or converted – by appending the datatype to an expression using "::". For instance:

```
SELECT revenue::int FROM rides
```

The CAST function can also be used to convert data types and is more broadly compatible with other databases and SQL standards. For instance:

```
SELECT CAST(revenue AS int) FROM rides
```

Primary Keys

As we know, a primary key uniquely defines a row within a table. In CockroachDB, a primary key is mandatory since all tables are distributed across the cluster based on the ranges of their primary key. If you 'don't specify a primary key, a key will be automatically generated for you.

'It's common practice to define an auto-generating primary key using clauses such as AUTOINCREMENT. The generation of primary keys in distributed databases is a significant issue since 'it's the primary key that is used to distribute data across nodes in the cluster. We'll discuss the options for primary key generation in the next chapter, but for now, we'll simply note that you can generate randomized primary key values using the `UUID` datatype with the `gen_random_uuid()` function as the default value:

```
CREATE TABLE people (
        id UUID NOT NULL DEFAULT gen_random_uuid(),
        firstName VARCHAR NOT NULL,
        lastName VARCHAR NOT NULL,
        dateOfBirth DATE NOT NULL
 );
```

This pattern is considered best practice to ensure even distribution of keys across the cluster. Other options for autogenerating primary keys will be discussed in Chapter 5.

Constraints

The `CONSTRAINT` clause specifies conditions that must be satisfied by all rows within a table. In some circumstances, the `CONSTRAINT` keyword may be omitted, for instance, when defining a column constraint or specific constraint types such as PRIMARY KEY or FOREIGN KEY. Figure 4-7 shows the general form of a constraint definition.

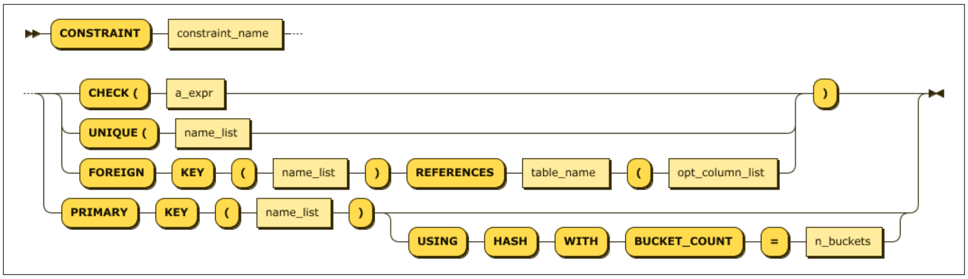


Figure 4-7. Contraint Statement

A UNIQUE constraint requires that all values for the column or column_list be unique.

PRIMARY KEY implements a set of columns that must be unique and which can also be the subject of a FOREIGN KEY constraint in another table. Both PRIMARY KEY and UNIQUE constraints require the creation of an implicit index. If desired, the physical storage characteristics of the index can be specified in the `USING`. The options of the USING INDEX clause have the same usages as in the CREATE INDEX statement.

NOT NULL indicates that the column in question may not be NULL. This option is only available for column constraints, but the same effect can be obtained with a table `CHECK` constraint.

CHECK defines an expression that must evaluate to true for every row in the table. We'll discuss best practices for creating constraints in Chapter 5.

Sensible use of constraints can help ensure data quality and can provide the database with a certain degree of self-documentation. However, some constraints have significant performance implications; we'll discuss these implications in Chapter 5.

INDEXES

Indexes can be created by the CREATE INDEX statement, or an INDEX definition can be included within the CREATE TABLE statement.

We talked a lot about indexes in Chapter 2, and we'll keep discussing indexes in the schema design and performance tuning chapters. Effective indexing is one of the most important success factors for a performance CockroachDB implementation.

Figure 4-8 illustrates a simplistic syntax for the CockroachDB CREATE INDEX statement.

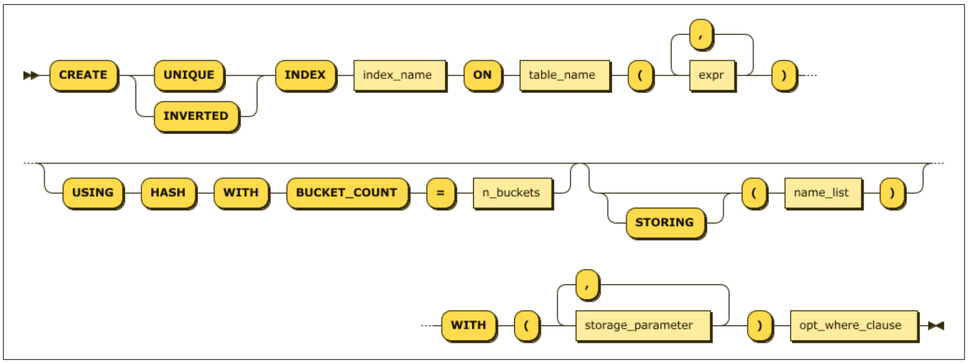


Figure 4-8. Create Index Statement

We looked at the internals of CockroachDB indexes in Chapter 2. From a performance point of view, CockroachDB indexes behave much as indexes in other databases – providing a fast access method for locating rows with a particular set of non-primary key values. For instance, if we simply want to locate a row name and date of birth, we might create the following multi-column index:

```
CREATE INDEX people_namedob_ix ON people
 (lastName,firstName,dateOfBirth);
```

If we furthermore wanted to ensure that no two rows could have the same value for name and date of birth, we might create a unique index:

```
CREATE UNIQUE INDEX people_namedob_ix ON people
 (lastName,firstName,dateOfBirth);
```

The STORING clause allows us to store additional data in the index, which can allow us to satisfy queries using the index alone. For instance, this index can satisfy queries that retrieve phone numbers for a given name and date of birth:

```
CREATE UNIQUE INDEX people_namedob_ix ON people
 (lastName,firstName,dateOfBirth) STORING (phoneNumber);
```

Inverted Indexes

An inverted index can be used to index the elements within an array or the attributes within a JSON document. We looked at the internals of inverted indexes in Chapter 2.

For example, suppose our `people` table used a JSON document to store the attributes for a person:

```
CREATE TABLE people
 ( id UUID NOT NULL DEFAULT gen_random_uuid(),
   personData JSONB );
```

```
INSERT INTO people (personData)
VALUES('{
                "firstName":"Guy",
        "lastName":"Harrison",
        "dob":"21-Jun-1960",
        "phone":"0419533988",
        "photo":"eyJhbGciOiJIUzI1NiIsI..."
     }');
```

We might create an inverted index as follows:

```
CREATE INVERTED INDEX people_inv_idx ON
people(personData);
```

Which would support queries into the JSON document such as this:

```
SELECT *
FROM people
WHERE personData @> '{"phone":"0419533988"}';
```

Bear in mind that inverted indexes index every attribute in the JSON document, not just those that you want to search on. This potentially results in a very large index. Therefore, you might find it more useful to create a calculated column on the JSON attribute and then index on that computed column:

```
ALTER TABLE people ADD phone STRING AS (personData->>'phone') VIRTUAL;

CREATE INDEX people_phone_idx ON people(phone);
```

Hash sharded indexes

If you are working with a table that must be indexed on sequential keys, you should use hash-sharded indexes. Hash-sharded indexes distribute sequential traffic uniformly across ranges, eliminating single-range hotspots and improving write performance on sequentially-keyed indexes at a small cost to read performance

```
CREATE TABLE people
( id INT PRIMARY KEY,
  firstName VARCHAR NOT NULL,
  lastName VARCHAR NOT NULL,
  dateOfBirth timestamp NOT NULL,
  phoneNumber VARCHAR NOT NULL,
  serialNo SERIAL ,
  INDEX serialNo_idx (serialNo) USING HASH WITH BUCKET_COUNT=4);
```

CREATE TABLE AS SELECT

The `AS select` clause of CREATE TABLE allows us to create a new table that has the data and attributes of a SQL SELECT statement. Columns, constraints and indexes can be specified as for an existing table but must align with the data types and number of columns returned by the SELECT statement. For instance, here we create a table based on a join and aggregate of two tables in the MOVR database:

```
CREATE TABLE user_ride_counts AS
SELECT u.name, COUNT(u.name) AS rides
  FROM "users" AS u JOIN "rides" AS r
    ON (u.id=r.rider_id)
 GROUP BY u.name
```

Note that while CREATE TABLE as SELECT can be used to create summary tables and the like, CREATE MATERIALIZED VIEW offers a more functional alternative.

ALTERING TABLES

The `ALTER TABLE` statement allows table columns or constraints to be added, modified, renamed or removed, as well as allowing for constraint validation and partitioning. Figure 4-9 shows the syntax

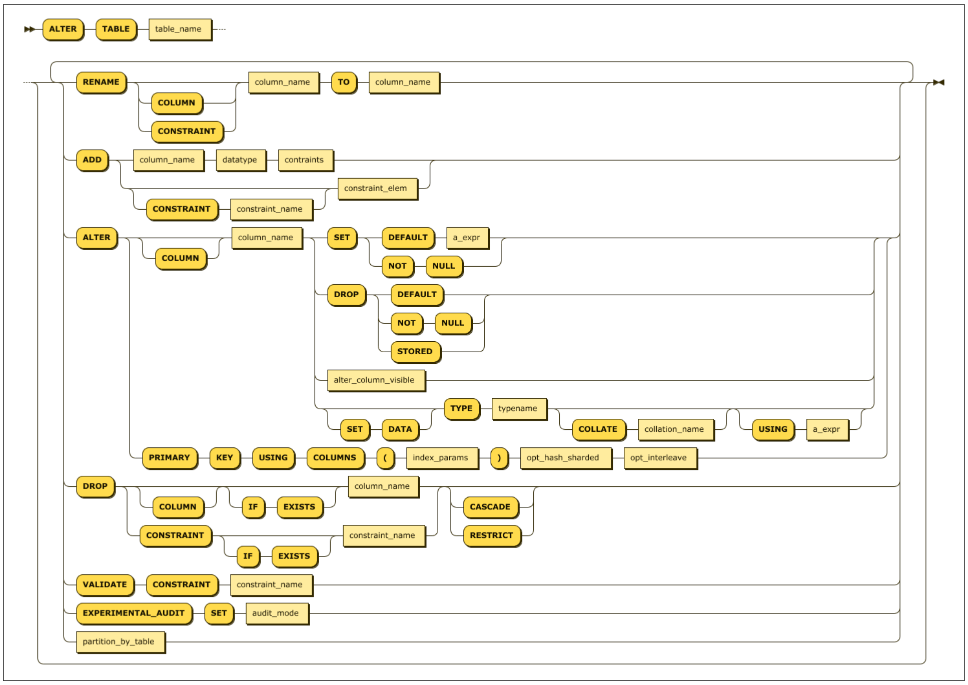


Figure 4-9. Alter Table Statement

Altering table structures online is not something to be undertaken lightly, although CockroachDB provides highly advanced mechanisms for propagating such changes without impacting availability and with minimal impact on performance[5]

DROPPING TABLES

Tables can be dropped using the DROP TABLE statement. Figure 4-10 shows the syntax.

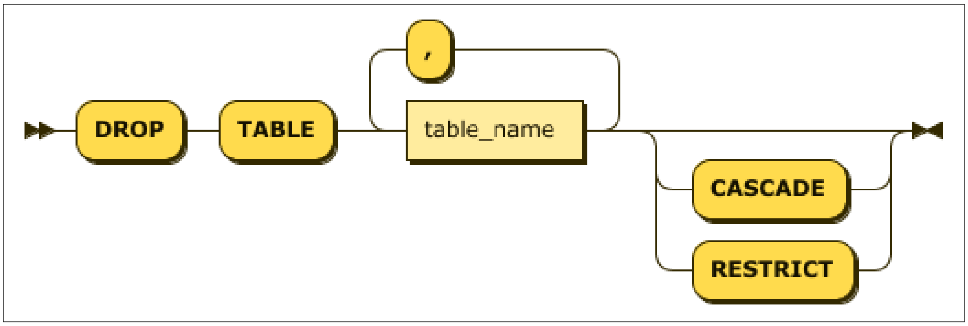


Figure 4-10. Drop Table Statement

More than one table can be removed with a single DROP TABLE statement. The CASCADE keyword causes dependent objects such as views or foreign key constraints to be dropped as well. RESTRICT – the default – has the opposite effect; if there are any dependent objects, then the table will not be dropped.

DROP CASCADE and Foreign Keys

DROP TABLE … CASCADE will drop any foreign key constraints which reference the table but will not drop the tables or rows which contain those foreign keys. The end result will be "dangling" references in these tables.

Because of this incompleteness, and since it can be hard to be certain exactly what CASCADE will do, 'it's usually better to manually remove all dependencies on a table before dropping it.

VIEWS

A standard view is a query definition stored in the database which defines a virtual table. This virtual table can be referenced in the same way as a regular table. Common Table Expressions can be thought of as a way of creating a temporary view for a

5 See *https://www.cockroachlabs.com/docs/stable/online-schema-changes*

single SQL. If you had a Common Table Expression that you wanted to share amongst SQLs, then a view would be a logical solution.

A Materialised view stores the results of the view definition into the database so that the view need not be re-executed whenever encountered. This improves performance but may result in stale results. If you think of a view as a stored query, then a materialized view can be thought of as a stored result.

Figure 4-11 shows the syntax of the CREATE VIEW statement.

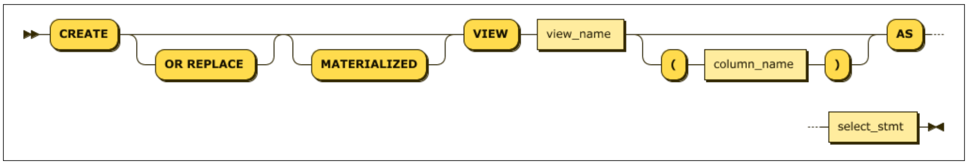


Figure 4-11. CREATE VIEW Statement

The REFRESH MATERIALIZED VIEW statement can be used to refresh the data underlying a materialized view.

INSERTING DATA

We can load data into a new table using the `CREATE TABLE AS select` statement discussed earlier, using the INSERT statement inside a program or from the command line shell, or by loading external data using the `IMPORT` statement. There are also non-SQL utilities that insert data – ' ‘we’ll look at these in Chapter 7.

The venerable INSERT statement adds data to an existing table. Figure 4-12 illustrates a simplified syntax for the INSERT statement.

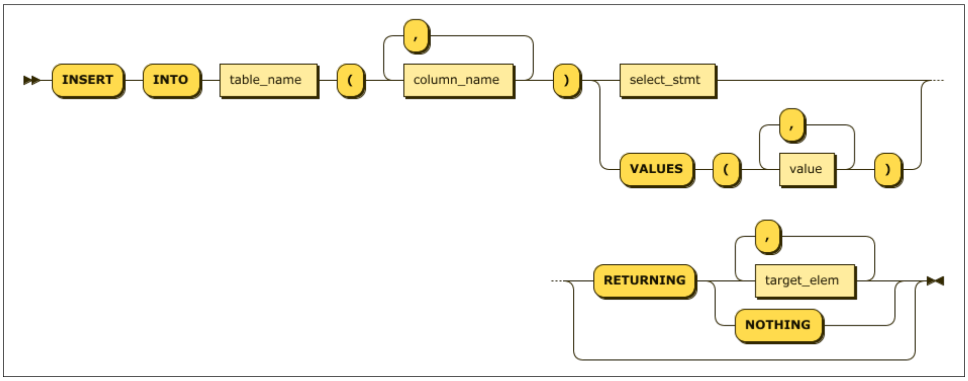


Figure 4-12. Insert Statement

INSERT takes either a set of values or a select statement. For instance, in the following example, we insert a single row into the `people` table:

```
INSERT INTO people (firstName, lastName, dateOfBirth)
VALUES('Guy', 'Harrison', '21-JUN-1960');
```

The VALUES clause of the INSERT statement can accept array values, inserting more than one row in a single execution.

```
INSERT INTO people (firstName, lastName, dateOfBirth)
VALUES ('Guy', 'Harrison', '21-JUN-1960'),
       ('Michael', 'Harrison', '19-APR-1994'),
        ('Oriana', 'Harrison', '18-JUN-2020');
```

There are alternative ways to insert batches in the various program language drivers, and we'll show some examples in Chapter 7.

A SELECT statement can be specified as the source of the inserted data:

```
INSERT INTO people (firstName, lastName, dateOfBirth)
SELECT firstName, lastName, dateOfBirth
  FROM peopleStagingData;
```

The RETURNING clause allows the data inserted to be returned to the user. The data returned will include not just the variables that were inserted but any auto-generated data. For instance, in this case, we INSERT data without specifying an ID value and have the ID values that were created returned to us:

```
INSERT INTO people (firstName, lastName, dateOfBirth)
VALUES ('Guy', 'Harrison', '21-JUN-1960'),
       ('Michael', 'Harrison', '19-APR-1994'),
        ('Oriana', 'Harrison', '18-JUN-2020')
  RETURNING id;
```

The ON CONFLICT clause allows you to control what happens if an INSERT violates a uniqueness constraint. Figure 4-13 shows the syntax.

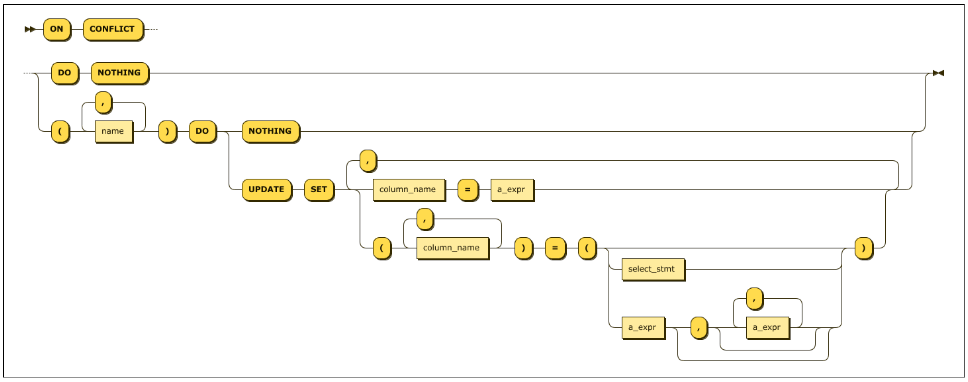


Figure 4-13. On Conflict Clause

Without an ON CONFLICT clause, a uniqueness constraint violation will cause the entire INSERT statement to abort. DO NOTHING allows the INSERT statement as a whole to succeed but ignoring any inserts that violate the uniqueness clause. The DO UPDATE clause allows you to specify an UPDATE statement that executes instead of the INSERT. The DO UPDATE functionality is similar in functionality to the UPSERT statement discussed later in this chapter.

IMPORT/IMPORT INTO

The IMPORT statement imports the following types of data into CockroachDB:

• Avro • CSV/TSV • Postgres dump files • MySQL dump files • CockroachDB dump files

IMPORT creates a new table, while `IMPORT INTO` allows an import into an existing table.

The files to be imported should exist either in a cloud storage bucket – Google Cloud Storage, Amazon S3 or Azure Blob storage – from a HTTP address or from the local filesystem (“nodelocal”).

We’ll discuss the various options for loading data into CockroachDB in Chapter 7. However, for now, let’s create a new table `CUSTOMERS` from a CSV file:

```
root@localhost:26257/defaultdb> IMPORT TABLE customers (
        id INT PRIMARY KEY,
        name STRING,
        INDEX name_idx (name)
)
CSV DATA ('nodelocal://1/customers.csv');
        job_id       |  status   | fra | rows | index_entries | bytes
---------------------+-----------+-----+------+---------------+--------
  659162639684534273 | succeeded |   1 |    1 |             1 |    47
(1 row)

Time: 934ms total (execution 933ms / network 1ms)
```

For a single node demo cluster, the `nodelocal` location will be somewhat dependent on your installation but will often be in an `extern` directory beneath the CockroachDB installation directory.

UPDATE

The UPDATE statement changes existing data in a table. Figure 4-14 shows a simplified syntax for the UPDATE statement.

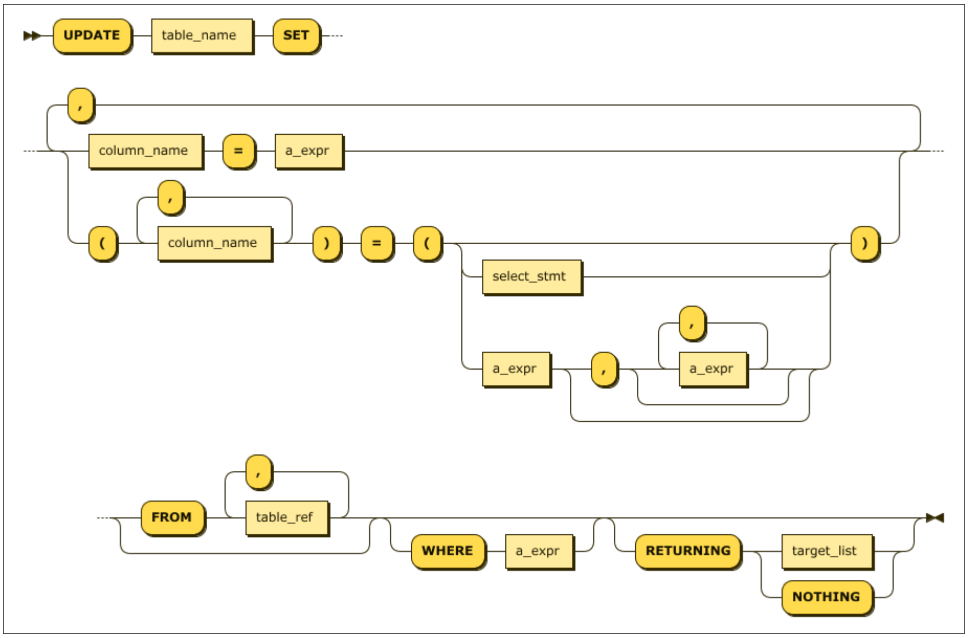


Figure 4-14. Update Statement

An update statement can specify static values as in the following example:

```
UPDATE users
  SET address = '201 E Randolph St',
     city='amsterdam'
 WHERE name='Maria Weber'
```

Alternatively, the values may be an expression referencing existing values:

```
UPDATE user_promo_codes
   SET usage_count=usage_count+1
  WHERE user_id='297fcb80-b67a-4c8b-bf9f-72c404f97fe8'
```

Or the UPDATE can use a subquery to obtain the values.

```
UPDATE rides SET (revenue, start_address) =
   (SELECT revenue, end_address FROM rides
     WHERE id = '94fdf3b6-45a1-4800-8000-000000000123')
WHERE id = '851eb851-eb85-4000-8000-000000000104';
```

The RETURNING clause can be used to view the modified columns. This is particularly useful if a column is being updated by a function, and we want to return the modified value to the application.

```
UPDATE user_promo_codes
   SET usage_count=usage_count+1
  WHERE user_id='297fcb80-b67a-4c8b-bf9f-72c404f97fe8'
 RETURNING (usage_count);
```

UPSERT

UPSERT can insert new data and update existing data in a table in a single operation. If the input data does not violate any uniqueness constraints, it is inserted. If an input matches an existing primary key, then the values of that row are updated.

In CockroachDB, the ON CONFLICT clause of INSERT provides a similar – though more flexible – mechanism. When this flexibility is not needed, UPSERT is likely to be faster than a similar INSERT…ON CONFLICT DO UPDATE statement.

Figure 4-15 shows the syntax of the UPSERT statement.

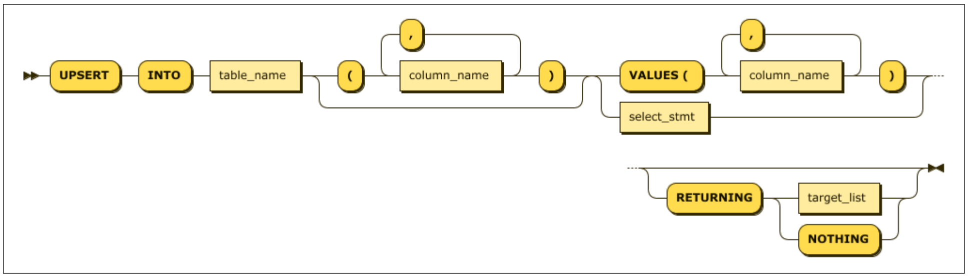


Figure 4-15. Upsert Statement

The UPSERT compares the primary key value of each row provided. If the primary key is not found in the existing table, then a new row is created. Otherwise, the existing row is updated with the new values provided.

The RETURNING clause can be used to return a list of updated or inserted rows.

In this example, the primary key of `user_promo_codes` is `(city, user_id, code)`. If a user already has an entry for that combination in the table, then that row is updated with a user_count of 0. Otherwise, a new row with those values is created.

```
UPSERT INTO user_promo_codes
  (user_id,city,code,timestamp,usage_count)
SELECT id,city,'NewPromo',now(),0
  FROM "users"
```

DELETE

DELETE allows data to be removed from a table. Figure 4-16 shows a simplified syntax for the DELETE statement.

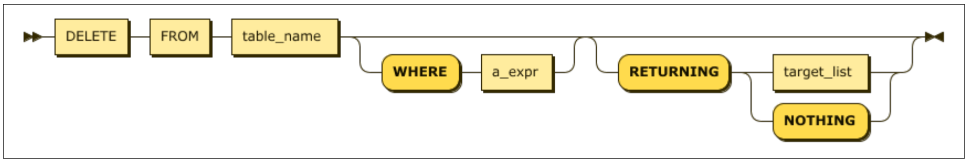


Figure 4-16. Delete Statement

Most of the time, a DELETE statement accepts a `WHERE` clause and not much else. For instance, here we delete a single row in the `people` table:

```
DELETE FROM people
 WHERE firstName='Guy'
   AND lastName='Harrison';
```

The RETURNING clause can return details of the rows which were removed. For instance:

```
DELETE FROM user_promo_codes
 WHERE code='NewPromo'
RETURNING(user_id);
```

TRUNCATE

TRUNCATE provides a quick mechanism for removing all rows from a table. Internally it is implemented as a DROP TABLE followed by a CREATE TABLE. TRUNCATE is not transactional – you cannot ROLLBACK a TRUNCATE.

TRANSACTIONAL STATEMENTS

We talked a lot about CockroachDB transactions in Chapter 2, so review that chapter if you need a refresher on how CockroachDB transactions work. From the SQL language point of view, CockroachDB supports the standard SQL transactional control statements.

BEGIN TRANSACTION

The BEGIN statement commences a transaction and sets its properties. Figure 4-17 shows the syntax.

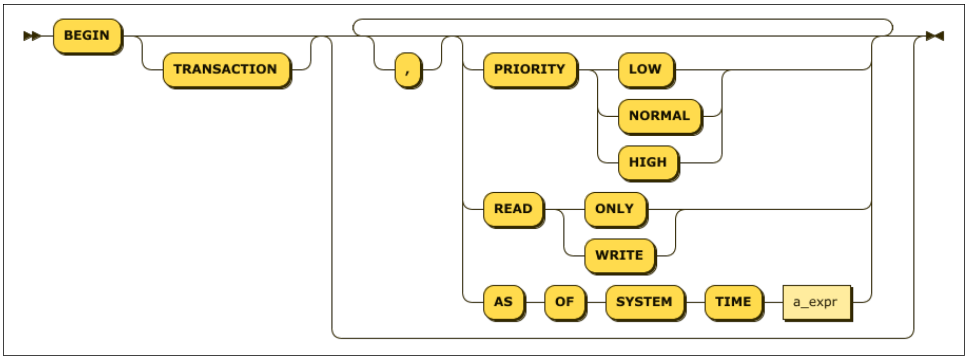


Figure 4-17. Begin Transaction

PRIORITY sets the transaction priority. In the event of a conflict, HIGH priority transactions are less likely to be retried.

READ ONLY specifies that the transaction is read-only and will not modify data.

AS OF SYSTEM TIME allows a READ ONLY transaction to view data from a snapshot of database history. We'll come back to this in a few pages.

SAVEPOINT

SAVEPOINT creates a named rollback point that can be used as the target of a ROLLBACK statement. This allows a portion of a transaction to be discarded without discarding all of the transactions' work. See ROLLBACK for more details.

COMMIT

The COMMIT statement commits the current transactions, making changes permanent.

Note that some transactions may require client-side intervention to handle retry scenarios. These patterns will be explored in Chapter 6.

ROLLBACK

ROLLBACK aborts the current transaction. Optionally, we can ROLLBACK to a savepoint, which rollbacks only the statements issued after the SAVEPOINT.

For instance, in the following example, the insert of the misspelled number *"tree* is rollbacked and corrected without abandoning the transaction as a whole:

```
BEGIN ;

INSERT INTO numbers VALUES(1,'one');
INSERT INTO numbers VALUES(2,'two');
```

```
SAVEPOINT two;

INSERT INTO numbers VALUES(3,'tree');
ROLLBACK TO SAVEPOINT two;

INSERT INTO numbers VALUES(3,'three');
COMMIT;
```

The special savepoint `cockroach_restart` can be used to more efficiently retry a transaction under certain circumstances. We'll review this procedure in Chapter 6.

SELECT FOR UPDATE

The FOR UPDATE clause of a SELECT statement locks the rows returned by a query, ensuring that they cannot be modified by another transaction between the time they are read and when the transaction ends. This is typically used to implement the pessimistic locking pattern that we'll discuss in Chapter 6.

A FOR UPDATE query should be executed within a transaction. Otherwise, the locks are released on completion of the SELECT statement.

A FOR UPDATE issued within a transaction will, by default, block other FOR UPDATE statements on the same rows or other transactions that seek to update those rows from completing until a COMMIT or ROLLBACK is issued. However, if a higher priority transaction attempts to update the rows or attempt to issue a FOR UPDATE, then the lower priority transaction will be aborted and will need to retry.

We'll discuss the mechanics of transaction retries in Chapter 6.

Table 4-4 illustrates two FOR UPDATE statements executing concurrently. The first FOR UPDATE holds locks on the affected rows, preventing the second session from obtaining those locks until the first session completes its transaction.

Table 4-4. For Update Example

Session1 SQL	Session2 SQL	Comment
BEGIN TRANSACTION	BEGIN TRANSACTION	
SELECT * FROM numbers WHERE n=1 FOR UPDATE		Data returned to Session1 and rows locked
	SELECT * FROM numbers WHERE n=1 FOR UPDATE	Session2 waits
COMMIT		Data returned to Session2 and Session2 acquires locks
	COMMIT	All locks released

AS OF SYSTEM TIME

The AS OF SYSTEM TIME clause can be applied to SELECT and BEGIN TRANSACTION statements as well as in BACKUP and RESTORE operations. AS OF SYSTEM TIME specifies that a SELECT statement or all the statements in a READ ONLY transaction should execute on a snapshot of the database at that system time. These snapshots are made available by the Multi-Version Concurrency Control (MVCC) architecture described in Chapter 2.

The time can be specified as an offset, or an absolute timestamp, as in the following two examples:

```
SELECT * FROM rides r
   AS OF SYSTEM TIME '-1d';

 SELECT * FROM rides r
   AS OF SYSTEM TIME '2021-5-22 18:02:52.0+00:00'
```

The time specified cannot be older in seconds than the replication zone configuration parameter `ttlseconds`, which controls the maximum age of MVCC snapshots.

OTHER DATA DEFINITION LANGUAGE TARGETS

So far, we've looked at SQL to create, alter and manipulate data in tables and indexes. These objects represent the core of database functionality in CockroachDB as in other SQL databases. However, the CockroachDB Data Definition Language provides support for a large variety of other, less frequently utilized objects. A full reference for all these objects would take more space than we have available here – see *https://www.cockroachlabs.com/docs/stable/sql-feature-support.html* for a complete list of CockroachDB SQL.

Table 4-5 lists some of the other objects that can be manipulated in CREATE, ALTER and DROP statements.

Table 4-5. Other CockroachDB schema objects

Object	Description
Database	A database is a namespace within a CockroachDB cluster containing schemas, tables, indexes and other objects. Databases are typically used to separate objects that have distinct application responsibilities or security policies.
Schema	A schema is a collection of tables and indexes that belong to the same relational model. In most databases, tables are created in the PUBLIC schema by default
Sequence	A sequence allows for the generation of unique numbers without requiring an explicit transaction. Sequences are often used to create primary key values, though, in CockroachDB, there are often better alternatives. See Chapter 5 for more guidance on primary key generation.
Role	A role is used to group database and schema privileges which can then be granted to users as a unit. See Chapter 12 for more details on CockroachDB security practices.

Object	Description
Type	In CockroachDB, a Type is an enumerated set of values that can be applied to a column in a CREATE or ALTER table statement.
User	A user is an account that can be used to log in to the database and can be assigned specific privileges. See Chapter 12 for more details on CockroachDB security practices.
Statistics	Statistics consist of information about the data within a specified table which the SQL optimizer uses to work out the best possible execution plan for a SQL statement. See Chapter 8 for more information on query tuning.
ChangeFeed	A change feed streams row-level changes for nominated tables to a client program. See Chapter 7 for more information on Changefeed implementation.
Schedule	A schedule controls the periodic execution of backups. See Chapter 11 for guidance on backup policies.

ADMINISTRATIVE COMMANDS

CockroachDB supports commands to maintain authentication of users and their authorities to perform database operations. It also has a job scheduler that can be used to schedule backup and restore and timed Data definition changes. Other commands support the maintenance of the cluster topology.

These commands are generally tightly coupled with specific administrative operations, which we'll discuss in subsequent chapters, so we'll refrain from defining them in detail here. You can always see the full documentation for them at *https://www.cockroachlabs.com/docs/stable/sql-feature-support.html*.

Table 4-6. CRDB Administrative Commands

Command	Description
CANCEL JOB	Cancel long-running jobs such as backups, schema changes or statistics collections
CANCEL QUERY	Cancel a currently running query
CANCEL SESSION	Cancel and disconnect a currently connected session
CONFIGURE ZONE	CONFIGURE ZONE can be used to modify replication zones for tables, databases, ranges or partitions. See Chapter 10 for more information on Zone configuration.
SET CLUSTER SETTING	Change a cluster configuration parameter
EXPLAIN	Show an execution plan for a SQL statement. We'll look at EXPLAIN in detail in Chapter 8.
EXPORT	Dump SQL output to CSV files
SHOW/CANCEL/PAUSE JOBS	Manage background jobs -imports, backups, schema changes, etc. - in the database
SET LOCALITY	Change the locality of a table in a multi-region database. See Chapter 10 for more information
SET TRACING	Enable tracing for a session. We'll discuss this in Chapter 8.
SHOW RANGES	Show how a table, index or databases are segmented into ranges. See Chapter 2 for a discussion on how CockroachDB splits data into ranges.
SPLIT AT	Force a range split at the specified row in a table or index.
BACKUP	Create a consistent backup for a table or database. See Chapter 11 for guidance on backups and high availability.

Command	Description
SHOW STATISTICS	Show optimizer statistics for a table.
SHOW TRACE FOR SESSION	Show tracing information for a session as created by the SET TRACING command.
SHOW TRANSACTIONS	Show currently running transactions
SHOW SESSION	Show sessions on the local node or across the cluster.

THE INFORMATION_SCHEMA

The INFORMATION_SCHEMA is a special schema in each database that contains metadata about the other objects in the database. You can use the information schema to discover the names and types of objects in the database. For instance you can use information_schema, to list all the information schema objects:

```
SELECT * FROM information_schema."tables"
 WHERE table_schema='information_schema';
```

Or you can use information schema to show the columns in a table:

```
SELECT column_name,data_type, is_nullable,column_default
 FROM information_schema.COLUMNS WHERE TABLE_NAME='customers';
```

The information_schema is particularly useful when writing applications against an unknown data model. For instance, GUI tools such as DBEaver use the information_schema to populate the database tree and display information about tables and indexes.

SUMMARY

In this chapter 'we've reviewed the basics of the SQL language for creating, querying and modifying data within the CockroachDB database.

A full definition of all syntax elements of CockroachDB SQL would take an entire book, so 'we've focused primarily on the core features of the SQL language with some emphasis on CockroachDB-specific features. For detailed syntax and for details of CockroachDB administrative commands, see the CockroachDB online documentation.

SQL is the language of CockroachDB, so of course, we'll continue to elaborate on the CockroachDB SQL language as we delve deeper into the world of CockroachDB.

[[Ch05 – CockroachDB Schema Design]]

CHAPTER 5
CockroachDB Schema Design

A sound data model is the foundation of a highly performant and maintainable application. In this chapter, we'll review the fundamentals of relational schema design, with a particular focus on aspects of schema design that bear on distributed database operations and on advanced CockroachDB features such as column families and JSONB support. We'll cover the creation of tables, indexes and other schema objects that support a well-designed CockroachDB application.

Although CockroachDB supports mechanisms for efficiently altering schemas online, schema changes to production applications are nevertheless high impact changes, typically involving coordinated changes to application code and production database configuration. If done poorly, there's the risk of loss of application functionality, availability of performance. Therefore although it's quite possible to alter CockroachDB schemas in production, it is far better to get the schema right during application design.

Relational database design is a big topic and has been the subject of many books and continuing debate. We don't want to try to cover advanced design principles here, nor do we want to engage in any debates about the purity of various design patterns. Most database models are a compromise between the mathematical purity of the relational model and the practicalities imposed by the physical database system. Therefore, in this chapter, we'll attempt to only briefly cover the theoretical side of the relational model but dive quite deep into the practicalities of designing a model that will work well with a CockroachDB implementation.

LOGICAL DATA MODELLING

Application data models are commonly created in two phases. Establishing the logical data model involves modeling the information that will be stored and processed by the application and ensuring that all necessary data is correctly, completely and unambiguously represented. The logical data model is then mapped to a physical data model. The physical data model describes the tables, indexes, views that are created in the DBMS.

The logical data model typically satisfies only the functional requirements of the application. The physical data model must also satisfy non-functional requirements, most particularly performance requirements.

In practice, these two phases are often blurred together, especially in agile and other iterative development environments. Nevertheless, whether done explicitly or not, there is definitely a difference between the analysis required to determine **what** data an application might process and **how** that data is best represented in a specific database system.

We introduced some of the core concepts of the relational model in Chapter 1. Theoretically, during logical data modeling we deal with relations, tuples and attributes, while in physical design, we deal with tables, rows and columns. However, outside of academia, these distinctions are often ignored, and in practice, it's commonplace to develop a logical model using the language of tables and columns.

Mea Culpa

Relational data modeling has generated an immense volume of research and debate over the past four decades. It's almost impossible to say anything sensible about relational data modeling without oversimplification or misrepresentation.

This is a book about CockroachDB, not about the relational theory, and so we have tried to avoid getting bogged down in debates about the correct way to perform relational design. Our purpose here is to provide enough quick background on relational modeling to allow for us to sensibly talk about CockroachDB specific physical design principles. Our apologies to anyone who feels we've failed to adequately cover this complex and nuanced topic.

Normalization

A normalized data model is one in which any data redundancy has been eliminated and in which all data is completely identifiable by primary and foreign keys. Although the normalized data model is rarely the final destination from a performance point of view, the normalized data model is almost always the best initial representation of a schema since it minimizes redundancy and ambiguity.

The relational theory defines multiple "levels" of normalization. The third normal form is the generally accepted standard of an adequately normalized data model.

In the third normal form, every attribute (column) in a tuple (row) are dependent only on the entire primary key of that tuple and not on any other attribute or key. We sometimes remember this as "The key, the whole key and nothing but the key".

For example consider the data shown in Figure 5-1.

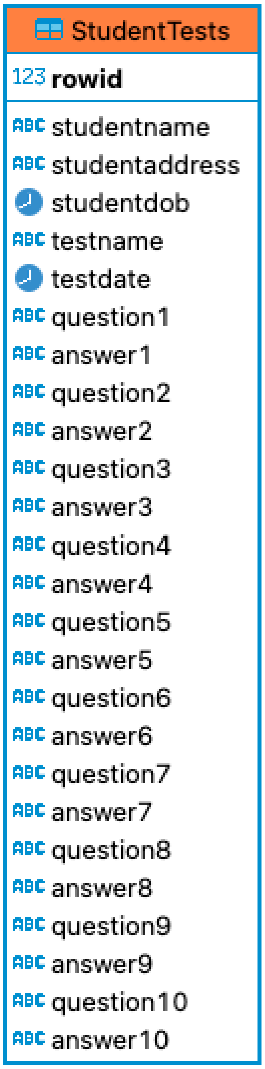


Figure 5-1. Student Test Data (Denormalized)

Even if we created a primary key on STUDENTNAME, TESTNAME and TEST-DATE, we would still be a long way from the third normal form. Attributes such as STUDENTDOB are dependent only on part of the key (STUDENTNAME), and the repeating "answer" columns are dependent on a non-key attribute (the corresponding question).

A normalized version of this data is shown in Figure 5-2. Students take tests, and tests have questions, and students answer those questions. All attributes in each relation are now fully dependent on the primary keys for that relation.

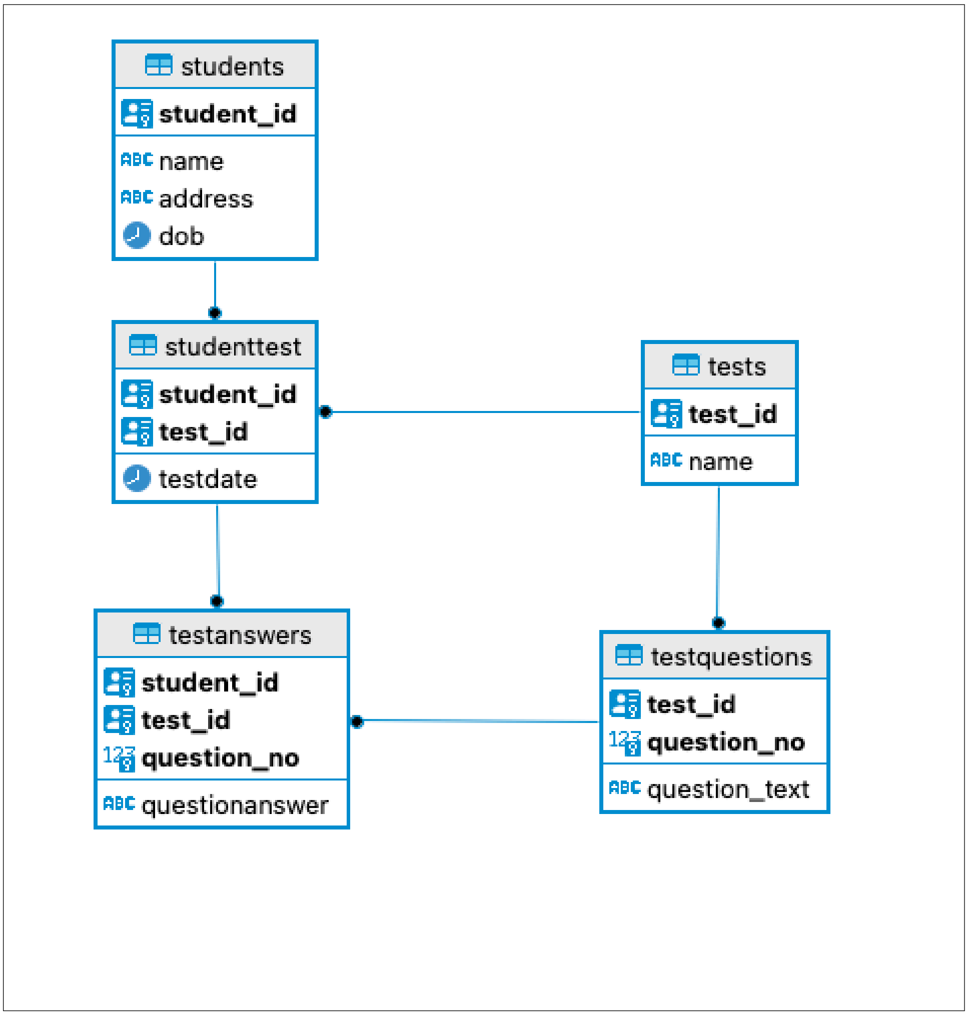


Figure 5-2. Student Test Data (Normalized)

Don't go too far

You'll generally recognize a well-normalized data model by the absence of any redundant information. For instance, in Figure 5-2, you'll notice that student names, test names, question texts, etc. are never repeated across multiple entries. There is one and only one entry for each attribute. The only thing that is repeated in a well-normalized model should be foreign key references.

That having been said, it's often a mistake to take the normalization process too far. In a real-world database, each new table in the model adds complexity to program code and overhead in joining information during data retrieval.

For example, from time to time we see addresses "normalized" as in Figure 5-3.

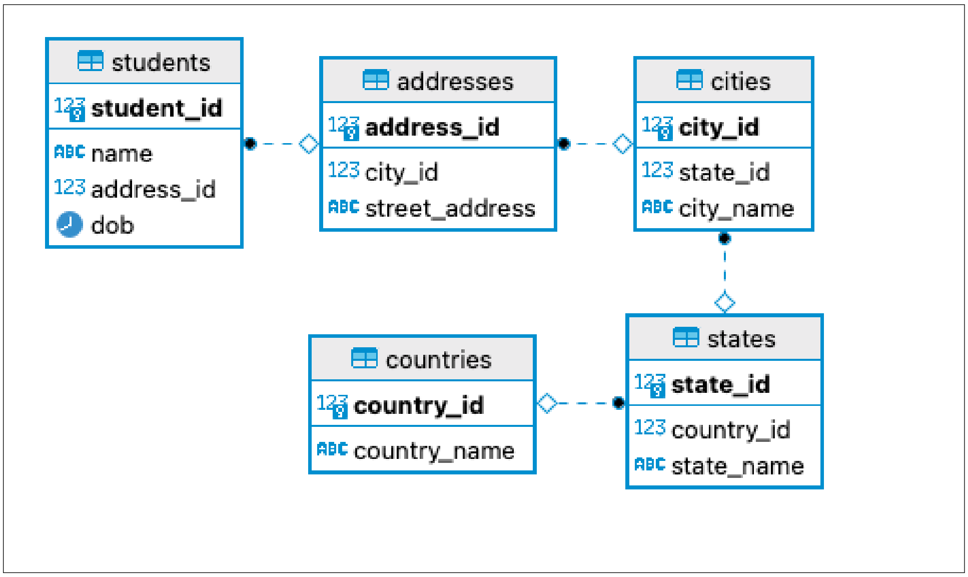


Figure 5-3. Student Address model (Normalized)

There's nothing theoretically wrong with this model. Two students could share a flat, and the relationships between cities, states and countries are very real. We could even throw in continents, solar systems and galaxies entities into the model without violating Third Normal Form.

However, in practice, this model will require a five-way join whenever a student's address needs to be retrieved. Since this is a pretty common operation, the cost of the join across the life of the application will be high. It's probably better just to leave the address fully embedded in the STUDENTS table as shown in Figure 5-2.

Purists will definitely argue that this sort of denormalization should be performed in the physical data modeling stage. And it's also worth noting that there may be good reasons for having a CITY or STATE table in a production system – each might be associated with specific attributes of relevance. But we'd suggest that you be pragmatic when contemplating such extended relationships. There's a lot of wasted effort in modeling logical relationships that are inevitably going to be collapsed in the physical design phase.

Primary Key choices

In CockroachDB, the choice of primary keys is critical to performance because it is the primary key that will determine the distribution of data across the nodes in the system. We'll talk extensively about this in the next major section.

However, even from the logical modeling point of view, there are some factors to consider.

The third normal form requires that each relation have a primary key. However, it does not specify whether that key should be artificial or synthetic. A *natural key* is one constructed from unique attributes that normally occur within the entity. An *artificial key* is one that contains no meaningful column information and which exists only to uniquely identify the row. There is a continual debate within the database community regarding the merits of "artificial" primary keys versus the "natural" key.

In general, we are of the opinion that most fundamental entities should use an artificial key. Artificial keys are generally superior from a performance point of view and eliminate some of the awkwardness involved when a natural primary key is changed. Furthermore, in CockroachDB, the use of an artificial key provides us with methods for ensuring an even distribution of keys throughout the cluster.

Special purpose Designs

For any given set of data, there usually exists more than one way to create a nearly correct relational model. Within the universe of possible models, there exist some patterns that are particularly applicable to certain workloads. To of the most common are:

- **Data warehousing** designs such as the **star** and **snowflake** schemas. These models have a large central "fact" table with foreign keys to multiple "dimension" tables. CockroachDB is not primarily intended as a data warehousing database, and so these models are not typical of a CockroachDB deployment.
- **Time-Series** designs in which the time of origin of data is part of each data element's key and in which data accumulates primarily as continual inserts. We'll briefly consider some of the considerations for time-series in the next section.

PHYSICAL DESIGN

Physical design involves modifying the logical design so as to improve its performance or maintainability. Some of these changes are driven by workload considerations. For instance, if a table is only ever accessed in a JOIN with another table, we might replicate some columns from the second table into the first to avoid the join.

The other primary physical design driver is the capabilities and performance characteristics of the database engine. For instance, in CockroachDB, ascending primary keys cause hotspots on certain nodes and should be avoided, while in a non-distributed SQL database such as PostgreSQL, ascending keys are fine.

Entities to tables

The major output of the logical design process are entities, attributes and keys. To convert the logical model to a physical model, we need to convert entities to tables and attributes to columns.

Depending on your logical model, this conversion may be close to a one-to-one mapping. However, be aware that in some cases, a single entity may map to multiple tables or vice-versa. For instance, we might decide that the logical model shown in Figure 5-3 should be collapsed to a single table, folding all address attributes into the Student table. Or we might collapse the addresses entity into students and collapse states and countries into cities.

In some cases, a logical model may include **subtypes** in which an entity is defined that includes multiple "types" of tuples. For instance, a PEOPLE entity might be defined as shown in Figure 5-4.

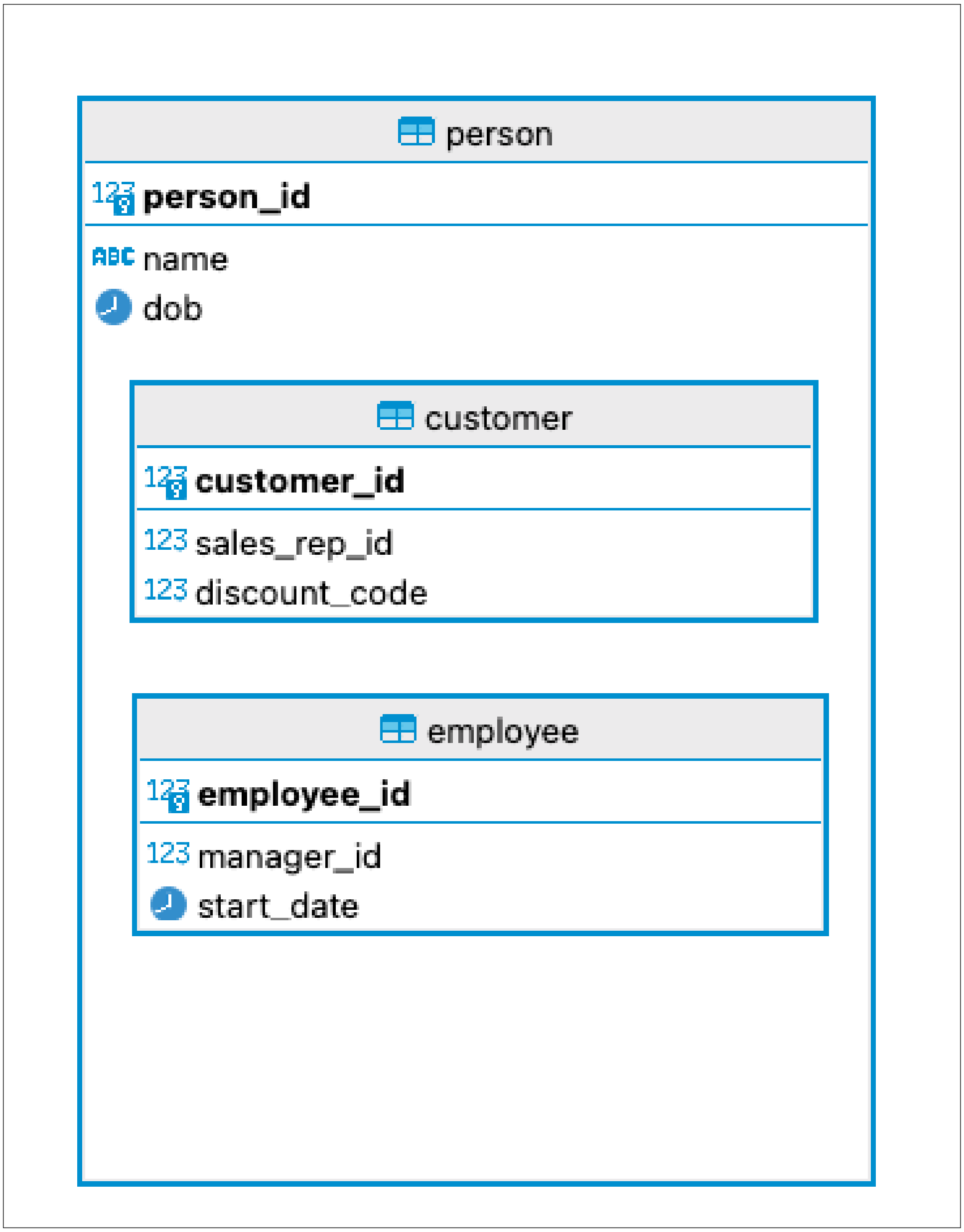


Figure 5-4. Logical Model with Subtypes

People can be customers or employees (or both). So should there be a single PEOPLE table, a CUSTOMER and an EMPLOYEE table or even three tables: a PEOPLE table with attributes common to customers and employees and CUSTOMERS and EMPLOYEES tables with attributes unique to each type?

The answer depends on your workload and performance requirements. Each of the above solutions has a performance advantage for a certain class of query; you'll need to think through the operations that are most important to your application. However, the solution that we've seen most often is the two-table model (CUSTOMERS and EMPLOYEES).

Attributes to Columns

When mapping attributes to columns, we are mainly concerned with selecting the best datatype for the column and defining its nullability correctly.

Null values are an important concept in relational databases – they distinguish between data that has a known value and data which is unknown or missing. Three valued logic - TRUE/FALSE/NULL - is at the heart of SQL operations, such as WHERE.

In some systems, the use of NULLS in indexed columns is discouraged, and it is recommended to use NOT NULL with a DEFAULT value. This is because, in some databases (PostgreSQL, for instance), NULL values are not included in indexes. However, CockroachDB doe store NULL values in indexes, and you can use an index to evaluate a IS NULL where condition.

CockroachDB datatypes generally map easily to logical datatypes. The following considerations may be considered:

- All these CockroachDB string datatypes are equivalent: TEXT, CHAR, VARCHAR, CHARACTER VARYING and STRING.
- All of the integer datatypes: INT, INT2,I NT4, INT8, BIGINT, SMALLINT, etc., are all stored in the same manner in the database. A BIGINT and a SMALLINT consume the same storage (providing they hold the same value). The types only serve to constrain the ranges of values that can be stored. The INT type can hold any allowable integer value (a 64-bit signed integer).
- Similarly, FLOAT, FLOAT4, FLOAT8 and REAL datatypes all store 64-bit signed floating-point numbers).
- DECIMAL stores exact fixed-point numbers and should be used when it's important to preserve precision, such as for monetary values.
- BYTES, BYTEA and BLOB store binary strings of variable length. The data is stored in line with other row data, and therefore, this datatype is not suitable for very large objects (a maximum of 1MB is suggested).

- TIME stores a time value in UTC, while TIMETZ stores a time value with a time-zone offset from UTZ. TIMESTAMP and TIMESTAMPZ are similar but include both date and time in the value.

Some of the other CockroachDB data types – such as ARRAYS and JSON - will be discussed later in the chapter.

Primary key design

We've touched upon the importance of properly defining CockroachDB primary keys in preceding chapters: now it's time to get serious about this very important topic.

The primary key of a table is used to distribute the ranges of that table's data across the cluster. If the primary key value is monotonically increasing, then all new data will go into a new range and will be sent to a specific node in the cluster. Most likely, this node will become a hotspot and limit the insert throughput for the cluster. This becomes particularly significant as your cluster grows: adding new nodes to a cluster may fail to result in higher throughput.

The same phenomenon can be encountered in a time-series database in which the primary key is prefixed with a timestamp. All "new" data will hit a single node, and your cluster scalability will be compromised.

For instance, consider this implementation of the ORDERS table:

```
CREATE SEQUENCE order_seq;

CREATE TABLE orders  (
        salesorderid INT NOT NULL PRIMARY KEY DEFAULT nextval('order_seq'),
  orderdate DATE NOT NULL DEFAULT now() ,
        duedate DATE NOT NULL,
        shipdate DATE NULL,
  customerid INT NOT NULL,
        salespersonid INT NULL,
  totaldue DECIMAL NULL
);
```

The orders_seq sequence generator generates numbers that are guaranteed to be incrementing and – most of the time – without gaps[1]. Since every value of ORDERID is one higher than the preceding value, new orders will be inserted into a single range which will be located on a single node. Consequently, that node will bear the burden of all INSERT operations. As new ranges are created, the responsibility of handling inserts will shift to new nodes, but at any given point in time, just a single node will be handling all of the inserts.

1 Sequence generators are provided mainly for compatibility with other databases and are not recommended for most CockroachDB applications

In the next few sections we'll look at ways of avoiding this primary key "hot spot" anti-pattern.

UUID based primary keys

If your application doesn't need primary key values to be a continuously increasing value, then a UUID (Universally Unique IDentifier) primary key is the recommended solution.

A UUID is a value that is guaranteed to be unique across all systems. A UUID combines host-specific data, random numbers and timestamp data to generate an identifier that will be unique across systems and times.

The + gen_random_uuid()+ function generates UUIDs and can be used as the default value for a primary key, as in the example below.

```
CREATE TABLE orders  (
        orderid uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
  orderdate DATE NOT NULL DEFAULT now() ,
        duedate DATE NOT NULL,
        shipdate DATE NULL,
  customerid INT NOT NULL,
        salespersonid INT NULL,
  totaldue DECIMAL NULL
);
```

UUIDs are unique, selective and guaranteed to be evenly distributed across all nodes of a cluster. They are, therefore the preferred mechanism for CockroachDB primary keys.

The SERIAL Datatype

In PostgreSQL the SERIAL datatype is typically used to create auto-incrementing key values. It's a handy alternative to creating a sequence, as shown in an earlier example.

However, in CockroachDB the SERIAL datatype by default generated unique identifiers using the `unique_rowid()` function. `unique_rowid()` generates unique numbers that combine nodeid and timestamp. While the numbers are generally ascending, the order is not absolutely guaranteed, and large gaps will occur.

You can change the behavior of SERIAL to a more PostgreSQL compatible behavior using the session variable `serial_normalization`. However, as with PostgreSQL, gaps in sequence numbers generated in this manner may still occur and the performance overhead is significant. The CockroachDB team recommends against using SERIAL data types unless compatibility with PostgreSQL is required.

Avoiding hotspots with a composite key

It may be that your application requirements really require a monotonically increasing key value. One way to avoid a hotspot, in this case, is to create a composite primary key that leads with a non-monotonically increasing value. For instance, in this implementation, the CUSTOMERID is prefixed to the order number in the primary key:

```
     CREATE TABLE orders  (
            orderid INT NOT NULL DEFAULT nextval('order_seq'),
      orderdate DATE NOT NULL DEFAULT now() ,
            duedate DATE NOT NULL,
            shipdate DATE NULL,
      customerid INT NOT NULL,
            salespersonid INT NULL,
      totaldue DECIMAL NULL,
      PRIMARY KEY  (customerid,orderid)
    );
```

This implementation tends to send orders for a specific customer into the same ranges, but sequential orders from multiple customers should be distributed across the cluster.

There may be some upside in "clustering" customer data in this way, but the clear downside is that we now need to know the customer id when searching for an order. We've probably all experienced the irritation of having to provide both a customer identifier AND an order identifier to a sales associate, so this downside is potentially significant. Of course, we could create a secondary index just on ORDERID, but then we'd create a secondary index with a hotspot.

What we need is a way to index monotonically increasing key values without creating unscalable insert hotspots. The solution is **hash sharded indexes**.

[[.hashSharding]] ===== Hash Sharded Primary Keys

Hash-sharded indexes add a hashed value to the prefix of a primary key. These hash values are unique but non-sequential. Consequently, if the primary key of a table is based on a hash-sharded index, then its values will be distributed evenly across all the ranges in the cluster. The result should be (statistically) a perfect distribution of writes across nodes.

```
At the time of writing, hash-sharded indexes required the +experimental_enable_hash_sharded_indexe
```

```
SET experimental_enable_hash_sharded_indexes=on;

CREATE TABLE orders  (
        orderid INT NOT NULL DEFAULT nextval('order_seq'),
  orderdate DATE NOT NULL DEFAULT now() ,
        duedate DATE NOT NULL,
        shipdate DATE NULL,
  customerid INT NOT NULL,
        salespersonid INT NULL,
    totaldue DECIMAL NULL ,
    PRIMARY KEY (orderid) USING HASH WITH BUCKET_COUNT=6
);
```

The hash sharding is transparent to the application: you'll never see the hashed values, and all filters against existing primary keys will work as normal. However, the new index cannot be used to find ranges of primary keys or to sort the output by primary key. For instance, with a traditional primary key, the query below would be resolved efficiently by a range scan of the primary key index:

```
SELECT * FROM orders
 WHERE ORDERID>0
 ORDER BY ORDERID;
```

With a hash-sharded index, a full scan and a sort operation would be required.

The WITH BUCKET_COUNT clause determines how many "shards" of the index are created. CockroachDB creates a computed column for each bucket, creates an index shard for each bucket and then stores that index into the Key-Value store the appropriate computed column's hash as its prefix. Setting the number of buckets to twice the number of nodes as are in the cluster is a sensible default.

Gaps in sequential keys

Although sequences provide for guaranteed ascending key values, they cannot guarantee that there will be no missing values in the ordered sequence. For performance reasons, sequence number increments are not within the scope of an application transaction. Therefore, if a transaction issues a ROLLBACK after a sequence number is consumed, that number is lost. To achive anything like scalable distributed performance, you would using the CACHE option to give each node it's own unique set of ranges – which will result in keys being inserted out of order across nodes. Furthermore, cached sequence numbers may be lost on a cluster restart.

If an application needs absolutely gap-free numbers ("no missing orders," for instance), then the application will need to implement its own sequence generating logic. Balancing performance and functionality in this case is not trivial – we'll look at this more in Chapter 6.

Ordering of primary key attributes

For a multi-column primary key, the order of attributes has significant implications for performance. You should follow the guidelines for composite indexes which we'll outline later in the chapter. Generally, the more often a column is used independently of other columns, the more you'll want to place that column first in the primary key.

Likewise, the appearance of primary key columns in ORDER BY clauses should also influence the sequencing.

Summary of Primary key performance

We've spent a fair bit of time on Primary key mechanisms in CockroachDB for good reason. The effect on scalability can be dramatic, and practices that worked fine in traditional monolithic SQL databases can backfire in CockroachDB.

Figure 5-5 shows just how significant these effects can be. In Figure 5-5 we see that insert throughput is severely diminished when SERIAL or Sequence generated keys are used. UUIDs are preferred, but if you need ascending primary keys, then a hash sharded primary key index is recommended.

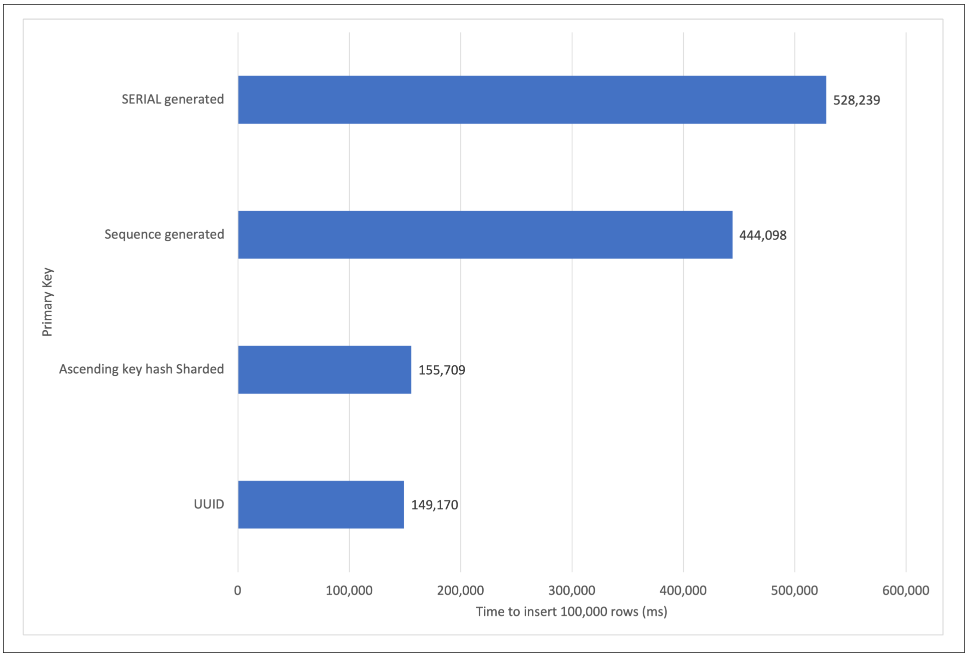


Figure 5-5. Insert performance with various Primary key schemes

The data in Figure 5-5 came from a 9-node CockroachCloud cluster. The performance penalty from ascending primary keys is proportional to the size of the cluster: the more nodes in the cluster the larger the relative penalty. So your milage may vary depending on your cluster size.

It is possible to greatly improve the performance of sequenced by creating them with the CACHE option. This avoids the blocking wait involved in aquiring the "next" sequence number. However, in a distributed system like CockroachDB, using CACHE defeats the purpose of the sequence generator. Because each node in the cluster has it's own cache, sequence numbers will be generated out of order across the cluster as a whole.

Foreign key constraints

Foreign key constraints help ensure data integrity and provide internal documentation of the data model, which can be leveraged by query generators and diagramming tools. However, during DML operations – particularly inserts, the validity of the foreign key must be checked by performing primary key lookups on the referenced table. These lookups can significantly increase the overhead of the operations and reduce throughput.

For the table that includes the Foreign key constraint, this shows up mostly in insert performance since it is somewhat unusual for a foreign key to be updated, and the foreign key references do not need to be validated during deletes.

For the tables that are referenced within in the foreign key constraint (e.g. the "parent" table), the overhead is felt most critically during deletes, where all child tables must be checked for "dangling" references.

The ON DELETE CASCADE clause of a CONTRAINT definition will automatically delete any child rows during the delete of a parent row. ON UPDATE CASCADE has a similar effect when a primary key is updated (which in most applications is a rare event).

Because of the overhead of foreign key constraints, it is not unusual for them to be removed in a production system. They may be left enabled only in test and development environments to catch any data anomalies.

```
    Dropping foreign key constraints can potentially lead to data anomalies, but in some circumstances
```

Denormalization

One of the outcomes in the development of a normalized data model is the removal of redundancies in data representation. In a well-normalized model, a data element is represented in just one place within the model. This eliminates the possibility of inconsistent information within the database.

De-normalization is the process of reintroducing redundant, repeating or otherwise non-normalized structures into the physical model—almost always with the intention of improving performance.

Denormalizing data is a common practice and one that you should not feel guilty about. However, do remember that denormalization has potential downsides:

- Denormalized data can create inconsistencies. These might be transitory (waiting for a materialized view to refresh) or permanent (a derived value is not updated due to a program error). You need to be sure that you have robust mechanisms in place to preserve data integrity.

- Denormalization has a performance overhead. Although denormalization exists to improve performance, most denormalizations have overhead. Typically, you improve query performance at the expense of DML performance. Make sure you understand and accept these trade-offs.

The best types of denormalizations are those that can be maintained by the database system automatically and transparently. For instance, in some databases, you might be tempted to vertically partition a table so that you can separate frequently accessed and rarely accessed columns. In CockroachDB column, families provide this capability without the need to change application code.

Replicating columns to avoid joins

JOIN operations magnify the overhead of retrieving data. Over-enthusiastic normalization can often result in even the most trivial SELECT operations requiring multi-table joins. For instance, consider the partial schema shown in Figure 5-6[2]

2 This is part of the Microsoft Adventureland sample database. You can find a version of Adventureland ported to CockroachDB at ???

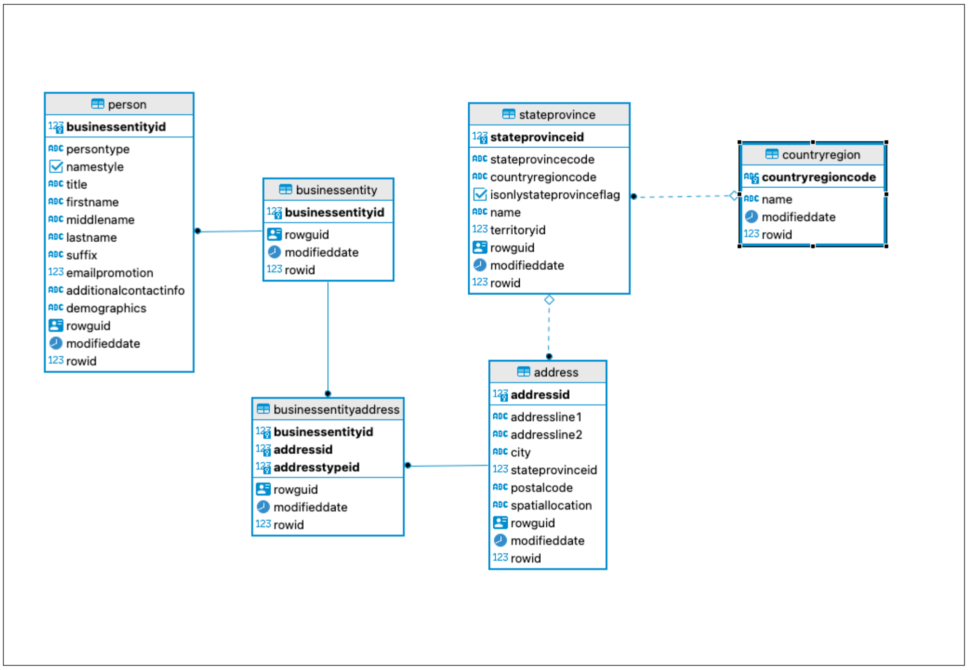


Figure 5-6. Overnormalized Address data model

To retrieve the address for a person (something we presumably do a lot), we need a five-table join:

```
SELECT p.firstname,p.lastname, a.addressline1,a.city, s2.name,c2.name
  FROM person p
  JOIN businessentity b2 ON (p.businessentityid=b2.businessentityid )
  JOIN businessentityaddress b3 ON (b3.businessentityid=b2.businessentityid)
  JOIN address a ON (b3.addressid=a.addressid)
  JOIN stateprovince s2 ON (s2.stateprovinceid=a.stateprovinceid)
  JOIN countryregion c2 ON (c2.countryregioncode=s2.countryregioncode)
 WHERE p.businessentityid =1
```

Because this join follows primary key values, it's going to be reasonably efficient, but it's still clearly going to involve five times as many lookup operations as would occur if all the columns were in the base table. So the solution is obvious: replicate the address directly into the person table. When a person's address changes, you may need to perform two UPDATE (one to ADDRESS, one to PERSON), but you will not have to perform a five-way join every time you want an address.

As with many design decisions, there are many options between the two extreems. If you want to preserve the multiple-address-per-person design of << adventurelan-dAddress>> you could still consider collapsing the STATE and COUNTRY tables into the ADDRESS table to reduce the number of tables involved in the join.

Summary tables

A summary table typically contains aggregated information that is expensive to compute on the fly. For instance, in the MOVR application, we might have a dashboard that shows revenue trends by city based on the following query:

```
SELECT cast(r.start_time AS date) AS ride_date,u.city,SUM(r.revenue)
  FROM rides r
  JOIN users u ON (u.id=r.rider_id)
 GROUP BY 1,2
```

Since revenue for previous days rarely changes, it's wasteful to continually reissue this expensive query every time the dashboard requests. Instead, we create a summary table from this data and reload the data at regular intervals (perhaps once an hour).

We can create such a summary table manually, but **materialized views** exist for this very purpose. We'd create a materialized view as follows:

```
CREATE MATERIALIZED VIEW ride_revenue_by_date_city AS
 SELECT cast(r.start_time AS date) AS ride_date,u.city,SUM(r.revenue)
  FROM rides r
  JOIN users u ON (u.id=r.rider_id)
 GROUP BY 1,2;
```

The resulting table is far smaller than the source tables, and we can manually refresh it whenever we like. One of the advantages of a materialized view is that we can also update it whenever we like with the REFRESH command:

```
REFRESH MATERIALIZED VIEW ride_revenue_by_date_city;
```

Vertical partitioning

Vertical partitioning involves breaking up a table into multiple tables, each of which contains a different set of rows. This is typically done to reduce the amount of work that needs to be done when updating a row, and can also reduce the conflicts that occur when two columns are subject to high concurrent update activity.

For instance, consider an IoT application in which a cities current temperature and air pressure are updated multiple times a second by weather sensor devices across the city:

```
CREATE TABLE cityWeather  (
        city_id uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
        city_name varchar NOT NULL,
        currentTemp float NOT NULL,
        currentAirPressure float NOT NULL);
```

The temperature values and airPressure readings come from different systems, and we're concerned that they will cause transaction conflicts when they attempt to change the same row simultaneously. We could partition the table into two tables to avoid this conflict:

```
CREATE TABLE cityTemp  (
        city_id uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
        city_name varchar NOT NULL,
        currentTemp float NOT NULL);

CREATE TABLE cityPressure (
        city_id uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
        city_name varchar NOT NULL,
        currentTemp float NOT NULL,
        currentAirPressure float NOT NULL);
```

However, CockroachDB **Column Families** provide a solution that does not require us to modify our data model. We simply add each measurement to its own family:

```
CREATE TABLE cityWeather  (
        city_id uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
        city_name varchar NOT NULL,
        currentTemp float NOT NULL,
        currentAirPressure float NOT NULL,
        FAMILY f1 (city_id,city_name),
        FAMILY f2 (currentTemp),
        FAMILY f3 (currentAirPressure)
);
```

Horizontal Partitioning

Horizontal partitioning (usually just referred to as partitioning) allows a table or index to be comprised of multiple segments.

• Queries can read only the partitions that contain relevant data, reducing the number of logical reads required for a particular query. This technique – known as partition elimination – is particularly suitable for queries that read too great a portion of the table to be able to leverage an index, but still do not need to read the entire table. • By splitting tables and indexes into multiple segments, parallel processing can be significantly improved since operations can be performed on partitions concurrently. • Deleting old data can sometimes be achieved by deleting old partitions rather than needing to perform expensive delete operations on large numbers of individual rows. Partitioning is not available in all CockroachDB editions: The partitioning option is currently available only in the CockroachDB Enterprise Edition.

Partitioning can be performed by range or list:

• Range partitioning allows rows to be allocated to partitions based on contiguous ranges of the partition key. Range partitioning on a time-based column is common since it allows us to purge older data by dropping a partition • List partitioning allows rows to be allocated to partitions based on nominated lists of values. This is similar but more flexible than range partitioning and allows non-adjacent partition key rows to be stored in the same partition.

If you don't have an Enterprise license, you can achieve a similar effect by manually creating multiple tables for the ranges or lists of values selected. However, the application logic required to manually manage such a DIY partitioning scheme is cumbersome.

CockroachDB's multi-region capabilities eliminate one of the possible motivations for partitioning data. **Regional by row tables** are effectively transparently partitioned in such a way as to optimize access to those rows from a particular region. In other databases, explicit partitioning might be required to realize this goal. We'll look at Multi-region topologies in Chapter 10.

Repeating groups

The relational model abhors repeating groups since, in any such repeating group, the attributes are not fully dependent on the primary key alone. For instance, an array element is identified by the primary key and the array index.

However, it can be extremely tedious to perform joins to retrieve small groups of data elements of the same type. For instance, at the beginning of the chapter (see Figure 5-2), we defined a TESTANSWERS entity that contains one row for each answer on a test. If a test has 100 questions, we need to access 100 rows to see all the results.

The ARRAY type provides an alternative mechanism. CockroachDB arrays are one-dimensional collections of data of the same datatype. For instance, we could store the answers to the test in such an array:

```
CREATE TABLE  studentTest (
        student_id uuid NOT NULL ,
        test_id uuid NOT NULL,
        testDate date NOT NULL,
        testAnswers varchar[] NOT NULL
   );
```

We can set the results in a single update as follows:

```
UPDATE studenttest s
   SET testAnswers=array['a','b','c','d']
 WHERE student_id='2fdaadf5-ff3e-45c4-bc92-cc0d566e1ad9'
   AND test_id='dca69ac4-6c53-4efb-8c7e-bca9f412e2ee'
```

Now we only need to access a single row to get all the test results, which is a significant reduction in logical IO.

Array datatypes do have some downsides – the query syntax is awkward, and it can be hard to perform analytic queries. For instance, finding the sum or average of all elements in an array is not directly supported.

Inverted indexes and allow us to directly efficiently retrieve data from an array datatype: we'll elaborate on inverted indexes later in the chapter.

JSON DOCUMENT MODELS

The biggest challenge to relational databases over the past decade has come from "document databases" such as MongoDB and Couchbase. These databases store all data in the form of JavaScript Object Notation (JSON) documents. JSON documents are self-describing, so there need be no formal implementation of a schema in the DBMS. One simply retrieves the JSON from the database and examines the JSON to decode the structure.

Without entering into any sort of religious debate about the obvious heresy involved in abandoning the relational model in favor of JSON documents, it's worth pointing out that document databases do offer significant conveniences for the developer:

- Modern object-oriented programming practices involve the modeling of program objects that have an internal structure that allows for inheritance, polymorphism and subclassing, all of which are core features of modern programming languages. These program objects are typically highly denormalized and, when stored in an RDBMS, must be unpacked. Object-oriented programmers used to say, "A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers". In contrast, a document database allows the objects to be stored directly.
- Modern DevOps practices involve continuous integration in which the entire application can be built directly from code and tested upon any significant change. RDBMS makes this difficult because a code change and a database change will need to be coordinated – ALTER TABLE statement and code commits need to be synchronously applied. Document databases avoid this issue.
- Document databases allow you to shoot yourself in the foot without the assistance of a DBA, and most developers want to shoot themselves in the foot whenever they like.

If these document database advantages are attractive to you, then you'll probably be drawn to the idea of storing all or some of your data in a JSONB datatype.

JSON objects are self-describing and can contain nested JSON objects and arrays. It's commonplace in document databases to embed child data within parent objects to avoid the need to perform joins. For instance, a video streaming database might include all of the titles a customer has viewed within a nested array:

```
{
        "_id" : ObjectId("5a0518aa5a4e1c8bf9a53761"),
        "Address" : "1913 Hanoi Way",
        "City" : "Sasebo",
        "Country" : "Japan",
        "District" : "Nagasaki",
        "FirstName" : "MARY",
        "LastName" : "Smith",
        "Phone" : 886780309,
        "dob" : ISODate("1982-02-20T13:00:00Z"),
        "views" : [
                {
                        "viewDate" : ISODate("2013-03-02T05:26:17.645Z"),
                        "filmId" : 611,
                        "title" : "MUSKETEERS WAIT"
                },
                {
                        "viewDate" : ISODate("2015-07-05T20:06:58.891Z"),
                        "filmId" : 308,
                        "title" : "FERRIS MOTHER"
                },
                {
                        "viewDate" : ISODate("2012-08-04T19:31:51.698Z"),
                        "filmId" : 159,
                        "title" : "CLOSER BANG"
                }

        ],
        "dateOfBirth" : ISODate("1982-02-20T13:00:00Z")
}
```

JSON document anti-patterns

In CockroachDB, implementing one to many relationships in JSON documents is inadvisable. Because the JSON data is stored in-line within the row data within the underlying Key-Value store, CockroachDB recommends that you keep the size of the JSON documents fairly small – under 1MB.

For instance, in the video streaming JSON model that we introduced in the previous section we embedded all the films that a customer had viewed within a JSON array. Given all the video streaming that has been going on lately, it's quite likely that, at least for some users, the 1MB limit would be exceeded.

You should also keep primary and foreign keys outside of JSONB. You can't create primary or secondary keys on JSONB data or on virtual columns derived from JSONB data. So at a minimum, your primary and foriegn keys should be explicitly defined in your CREATE TABLE statement.

Indexing JSON attributes

As mentioned in previous chapters, you can create inverted indexes on JSONB columns. These inverted indexes allow you to search for attribute value matches within the JSON object. However, inverted indexes index every attribute in the JSON object and so can have many more entries in the index than rows in the table. A better alternative is to create computed columns on the JSONB attributes and create an index on those attributes.

So let's say we have decided to store our customer details as a JSONB document. The basic customer details look like this:

```
{ "Address" : "1913 Hanoi Way",
        "City" : "Sasebo",
        "Country" : "Japan",
        "District" : "Nagasaki",
        "FirstName" : "Mary",
        "LastName" : "Smith",
        "Phone" : "886780309",
        "dob" :  "1982-02-20T13:00:00Z",
    "likes": ["Dinosaurs","Dogs","People"] }
```

We know that we want to search on LastName, Firstname, and we also know we want to have a foreign key out to an existing CITIES table. Our CREATE TABLE might look like this:

```
CREATE TABLE people (
        personId UUID PRIMARY KEY NOT NULL default gen_random_uuid(),
        cityId UUID ,
   personData JSONB,
   FirstName STRING AS (personData->>'FirstName') VIRTUAL,
   LastName STRING AS (personData->>'LastName') VIRTUAL,
   FOREIGN KEY (cityId) REFERENCES cities(cityid),
   INDEX  (LastName,Firstname)
);
```

This design allows us to perform index searches on LastName and FirstName, and to retrieve those attributes from the JSONB without the awkward JSON dereferencing syntax that we introduced in the last chapter. We can, however, add attributes without needing to issue an ALTER TABLE statement, and programmers can load the personData JSON type directly into a JSON object in their application code.

Using JSON or Arrays to avoid joins

We said before that "one to many" relationships should not be modeled in JSONB columns. The same is true of ARRAY columns. We want to avoid storing more data in these columns that the key-value store can process in a single operation.

However, it can be quite effective to model "one to few" relationships in JSONB or ARRAYS. For instance, consider the Students Tests schema we modeled way back in Figure 5-2. We know that there can be at most only a couple of hundred questions in a test. In the normalized solution, we always have to join tables in order to get the answers for a specific test:

```
SELECT s.student_id,s.test_id,question_no,questionanswer
  FROM studenttest s
  JOIN testanswers t on(t.student_id=s.student_id AND t.test_id=s.test_id)
 WHERE s.student_id=:student_id
   AND s.test_id=:test_id
```

We know that there can only be a couple of hundred questions in a test, and the answers can at most be only a few kilobytes. So being sure that the 1MB limit will not be exceeded, we could collapse the test answers into a JSON document:

```
CREATE TABLE  studentTest (
        student_id uuid NOT NULL ,
        test_id uuid NOT NULL,
        testDate date NOT NULL,
        answers JSONB
);

INSERT INTO studentTest (student_id,test_id,testDate,answers)
 VALUES ('2fdaadf5-ff3e-45c4-bc92-cc0d566e1ad9',
        'dca69ac4-6c53-4efb-8c7e-bca9f412e2ee',
         now(),
        '{"answers":[
                        {"questionNumber":1,"Answer":5},
                        {"questionNumber":2,"Answer":25},
                        {"questionNumber":3,"Answer":58},
                        {"questionNumber":4,"Answer":3425},
                        {"questionNumber":5,"Answer":432},
                        {"questionNumber":6,"Answer":0},
                        {"questionNumber":7,"Answer":673}
                ]}');
```

We can also use ARRAYS or JSON repeating groups to avoid joins where there is a "many to many" relationship between two tables. For instance, consider the relationship between students and classes as shown in Figure 5-7.

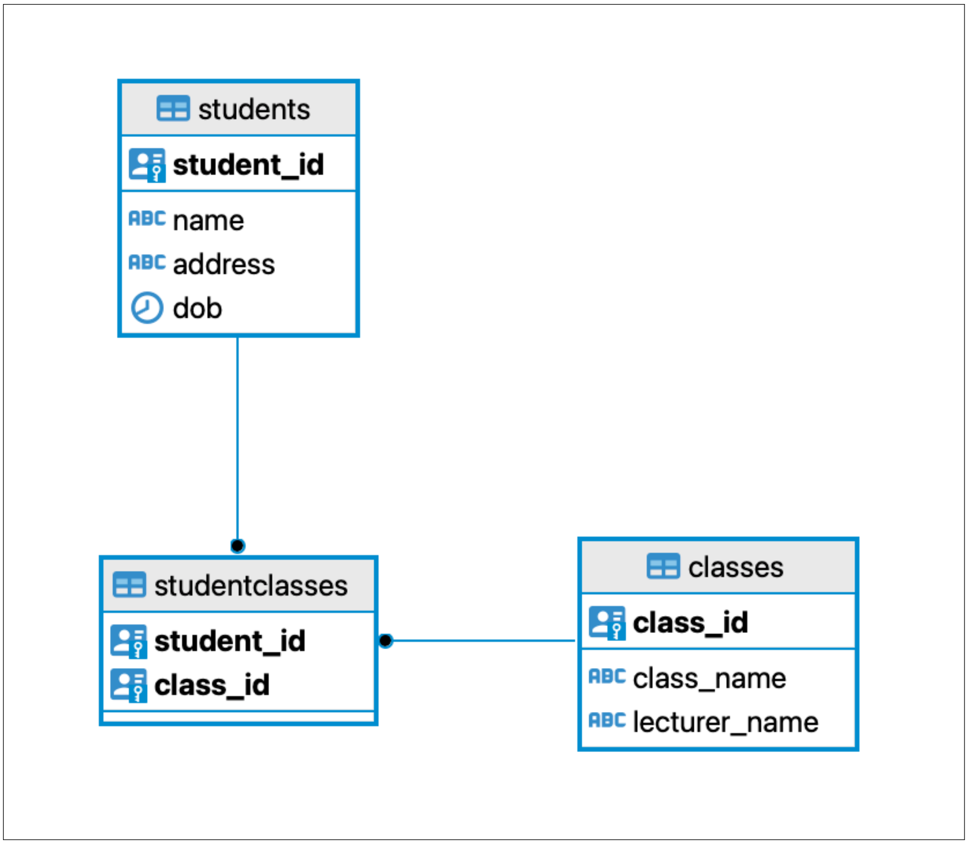


Figure 5-7. Students and Classes

Whenever we want to get a list of a student's classes, we are forced to perform a three-way join:

```
SELECT class_name FROM students
  JOIN studentClasses USING(student_id)
  JOIN classes USING(class_id)
 WHERE student_id='000390a6-4e1d-4bc1-aad7-66b645131d54';
```

The table STUDENTCLASSES exists only to join the STUDENTS and CLASSES tables – it contains no independent information.

We could instead store foreign keys for all the classes in an ARRAY type:

```
ALTER TABLE students ADD COLUMN classes UUID[];

UPDATE students s SET classes= (
       SELECT array_agg(class_id)
        FROM studentClasses sc
       WHERE s.student_id=sc.student_id);
```

The ARRAY_AGG function takes all the columns in a result set and converts them to an array. So in the above example, we copied all the CLASS_ID values for each student into the CLASSES array.

Now when we want to get the classes for a particular student we can "unnest" the array and join the resulting CLASS_ID values directly to the CLASSES table:

```
WITH students_classes AS (
        SELECT student_id , UNNEST(classes) class_id
          FROM students
)
SELECT class_name FROM classes
  JOIN students_classes USING(class_id)
 WHERE student_id='000390a6-4e1d-4bc1-aad7-66b645131d54';
```

This might all seem to be a bit convoluted, but in a high performance workload, reducing a three-way join to a two-way join might be necessary to achieve performance objectives, even if it complicates application code a little.

Of course, we can avoid the joins altogether if we embed all the information about a student's classes into a JSONB column, just as we did earlier for test answers. However, the array solution above doesn't duplicate any information from the CLASSES table into the STUDENTS table, so if the name of a class changes, we only have one update to perform.

Do bear in mind that by embedding foreign keys in this way, we lose the capability of defining FOREIGN KEY constraints and create some opportunities for data inconsistencies.

INDEXES

An index is a database object that provides a fast-path to specific data within a table.

We looked at the structure of indexes in Chapter 2. You might recall that in CockroachDB, indexes and tables share a common fundamental storage structure. A base table is essentially a relation indexed by the primary key. Secondary indexes are also relations but are indexed by the index key, with the payload typically containing the primary key values associated with that secondary key.

Indexes exist to optimize performance and enforce uniqueness. Indexes can generally be added to a system without requiring any change to application code, so compared with other options for physical implementation, they are fairly easy to modify. Creating an optimal set of indexes is one of the most important factors in ensuring optimal database performance.

Index selectivity

The _selectivity_of a column or group of columns is a common measure of the usefulness of an index on those columns. Columns or indexes are selective if they have a large number of unique values and few duplicate values. For instance, a Date_of_birth column will be quite selective, while a Gender column will be not at all selective.

Selective indexes are more efficient than non-selective indexes because they point more directly to specific values. The CockroachDB optimizer will determine the selectivity of the various indexes available to it and will generally try and use the most selective index.

Index break-even point

When you want to look up just a few things in a textbook, you go to the index. When you want to assimilate all or most of the content, you bypass the index and go directly to the text. It's the same with database indexes – we generally only want to use them when we are retrieving a relatively small amount of a table's data.

An non-covering index – one that includes the filter conditions but not all the columns in the SELECT list – is generally effective only when we are retrieving a small percentage of a table's data. Beyond that, the overhead of going backwards and forwards from index to base table will be slower than simply reading all the rows in the table.

However, when we create a covering index using the STORING clause, the situation is very different, In this case, the index can outperform the table access even if very large proportions of data are accessed.

The optimizer will attempt to determine how much data is being accessed and choose an index or a table scan as appropriate. However, you don't want to create an index that will never be used, so it's important to understand the "cut off" point between an index and a table scan.

For instance, let's say we have timeseries data where a measurement (say a temperature) was recorded every minute over the past year. The application is often asked to determine the average measurement over some recent time period. The query looks something like this:

```
SELECT AVG(measurement)
  FROM timeseries_data
 WHERE measurement_timestamp > ((date '20220101')- INTERVAL '$dayFilter days'
```

The variable `${dayFilter}` can take low or high values. We can create a non-covering index on the table as follows:

```
CREATE INDEX timeseries_timestamp_i1
    ON timeseries_data(measurement_timestamp)
```

However, this index will only be effective when the number of days selected is very small – probably less than a week. Alternatively, we could create a covering index that includes the MEASUREMENT column:

```
CREATE INDEX timeseries_covering
    ON timeseries_data(measurement_timestamp) STORING (measurement)
```

This index can be used effectively for any spans of data – from one day to the entire years data.

Figure 5-8 compares the performance of an index scan with a table scan, against the number of days of information being retrived. The table scan must do the same amount of work regardless of the amount of data being processed, while an index scan increases in overhead the larger the amount of data being processed. For a non-covering index, a table scan is better if there's more than about a weeks worth of data retrieved (about 2% of the total data). However, a covering index can perform well even if we are retrieving …..

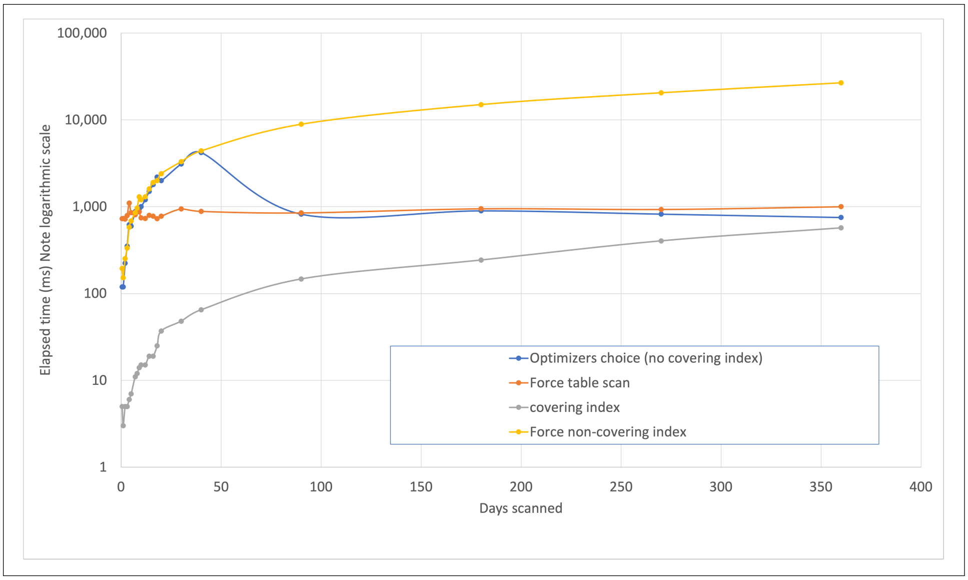


Figure 5-8. Comparison of table scan vs index scan performance

There are a few lessons to be drawn from Figure 5-8:

- The optimizer switches from an index scan to table scan when the amount of data hits about 10-15% of total. The optimizer is a sophisticated piece of software but it isn't magic, and it can't always work out which access path is better. In some circumstances, creating an non-covering index will actually degrade performance.

- A covering index is far superior in performace to a non-covering index, and can be used effectively even if all or most of the table is being accessed. Whenever possible, use a covering index.
- Remember that in CockroachDB, indexes and tables have the same storage format: a covering index is not just a fast access mechanism – it's also a compact representation of a subset of table columns that can be scanned far faster than the base table.

We'll come back to index performance and tuning queries in Chapter 8.

Index overhead

Although indexes can dramatically improve query performance, they do reduce the performance of DML. All of a table's indexes must normally be updated when a row is inserted or deleted, and an index must also be amended when an update changes any column which appears in the index.

It is, therefore, important that all our indexes contribute to query performance since these indexes will otherwise needlessly degrade DML performance. In particular, you should be especially careful when creating indexes on frequently updated columns. A row can only be inserted or deleted once but may be updated many times. Indexes on heavily updated columns or on tables that have a very high insert/delete rate will therefore exact a particularly high cost. Figure 5-9 illustrates the overhead on DML that occurs as more indexes are added to a table.

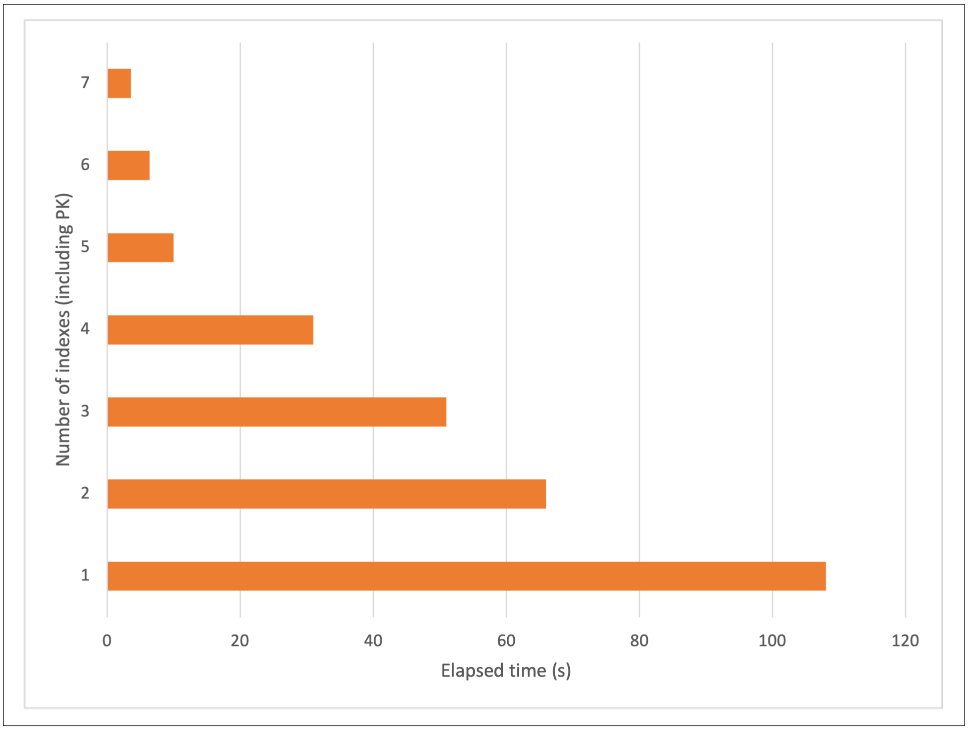


Figure 5-9. Index overhead – time to insert 120,000 rows

Composite indexes

A composite index is simply an index created from more than one column. The advantage of a composite key is that it is often more selective than a single key index. The combination of columns will point to a smaller number of rows than indexes composed of the individual columns.

For instance, if we know that we frequently perform searches on FIRSTNAME and LASTNAME, then it makes sense to create an index on both of those columns:

```
CREATE INDEX flname_idx ON person (lastname,firstname);
```

Such an index will be far more effective than an index on LASTNAME alone, or separate indexes on LASTNAME and FIRSTNAME. We'll provide some performance comparisons for composite indexes a bit later in the chapter.

If a composite index could only be used when all of its keys appeared in the WHERE clause, then composite indexes would probably be of pretty limited use. Luckily, a composite index can be used effectively, providing any of the initial or leading columns are used. Leading columns are those that are specified earliest in the index definition.

So, for instance, the index on (LASTNAME, FIRSTNAME) that we just created can optimize this query:

```
SELECT * FROM person WHERE lastname='Wood';
```

But not this query:

```
SELECT * FROM person WHERE firstname='John' ;
```

Covering indexes

A covering index is one that is capable of satisfying a query without reference to the base table. For instance, in the following query:

```
SELECT phonenumber
  FROM people
 WHERE lastname='Smith'
   AND firstname='Samantha'
   AND state='California' ;
```

An index on LASTNAME, FIRSTNAME, STATE and PHONENUMBER would not only be able to *find* the data requested but would also be able to *return* the PHONE-NUMBER. Only a single index access – and no base table read - would be needed.

In CockroachDB, we can use the STORING clause to store data elements that we might use in the SELECT clause, but not in the WHERE clause. This provides a more efficient mechanism for implementing a covering index. So for the above query, this index would be optimal:

```
CREATE INDEX people_lastfirststatephone_ix ON people
   (lastname,firstname,state)
    STORING (phonenumber);
```

Composite and Covering index performance

Figure 5-10 illustrates the performance advantages offered by composite and covering indexes. The chart shows the number of KeyValue store options necessary to satisfy this query under various indexing scenarios:

```
SELECT phonenumber
  FROM people
 WHERE lastname='Smith'
   AND firstname='Samantha'
   AND state='California' ;
```

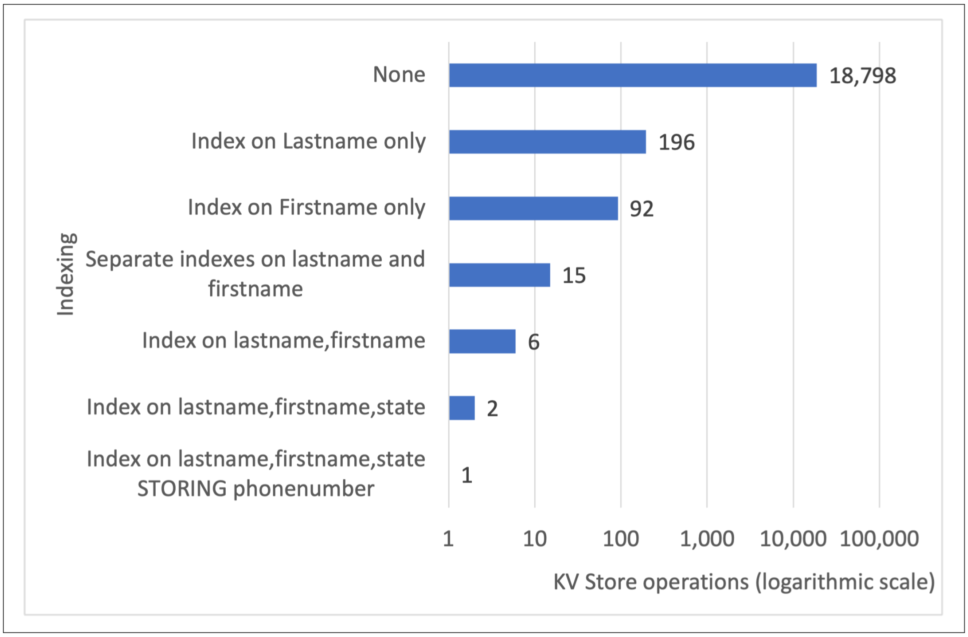


Figure 5-10. CompositeIndex Performance

Figure 5-10 shows that without indexing, the query requires 18,798 KV operations – we have to read every row in the table. Single indexes on LASTNAME or FIRSTNAME improve the performance somewhat, and having both a LASTNAME and FIRSTNAME index is better than just having either of the two indexes alone.

However, it's not until we use a composite index that we see truly efficient indexing. Only six KV reads are needed if we have an index on LASTNAME and FIRSTNAME, and only two KV operations are needed for an index on LASTNAME, FIRSTNAME and STATE. If we STORE the PHONENUMBER in the index, then only a single KV operation is needed.

Guidelines for composite indexes

As we saw earlier, the performance improvements from indexes don't come without a cost – each index ads overhead to DML operations, so we can't usually create every possible index that we might like.

The best strategy is to create composite indexes that cover the broadest possible ranges of queries. Since a composite index can be used if any of its leading columns are included in a WHERE clause, the ordering of columns in composite indexes is very important.

The following guidelines might help when deciding which indexes to create.

- Create composite indexes for columns that appear together in the WHERE clause.
- If columns sometimes appear on their own in a WHERE clause, place them at the start of the index.
- The more *selective* a column is, the more useful it will be at the leading end of the index.
- A composite index is more useful if it also supports queries where not all columns are specified. For instance, LASTNAME, FIRSTNAME is more useful than FIRSTNAME,LASTNAME because queries against LASTNAME only are more likely to occur than queries against FIRSTNAME only.

Indexes and null values

In many relational databases, NULL values are not included in indexes, and consequently, in these systems, it is often recommended not to use NULL values if you might want to search for those values. However, in CockroachDB, null values are included in indexes and can be found using an index in the normal way.

Inverted indexes

We discussed inverted indexes in Chapter 2 and earlier in this chapter in the "JSON DOCUMENT MODELS" on page 149 section. An inverted index creates an index for all elements in an array and for all attributes in a JSONB column. As useful and flexible as these inverted indexes may be, they are expensive from a storage and maintenance point of view. We recommend, whenever possible, creating a computed column from the JSONB attribute concerned and indexing on that column. See "JSON DOCUMENT MODELS" on page 149 for an example.

Partial indexes

A partial index can be created on only only a subset of rows in the table. A partial index is created by adding a WHERE clause to the CREATE INDEX statement.

Partial indexes can have a lower maintenance overhead, require less storage in the database, and be more efficient for suitable queries. They are, therefore a very useful type of index.

The key limitation with a partial index is that it can only be used when CockroachDB can be certain that the partial index contains all the necessary entries to satisfy the query. In practice, this means that a partial index is normally used to optimize queries that contain the same WHERE clause filter condition that was included in the index definition.

Sort-optimizing indexes

Indexes can be used to optimize ORDER BY operations in certain circumstances. When CockroachDB is asked to return data in sorted order, it must retrieve all the rows to be sorted and perform a sort operation on those rows before returning any of the data. However, if an index exists on the ORDER BY columns, then CockroachDB can read the index and retrieve the rows directly from the index in sorted order.

Using the index to retrieve data in sorted order is usually only worthwhile if you are optimizing for some small number of "top" rows. If you read the entire table in sorted order from the index, then you'll be reading all the index entries as well as all the table entries, and the total number of IO operations will be excessive. However, if you are just getting the first "page" of data or a "top ten" then the index will be much faster since you never have to read the rest of the table rows at all.

Figure 5-11 illustrates the effect of an index to optimize a sort like this:

```
SELECT *
  FROM orderdetails
 ORDER BY modifieddate;
```

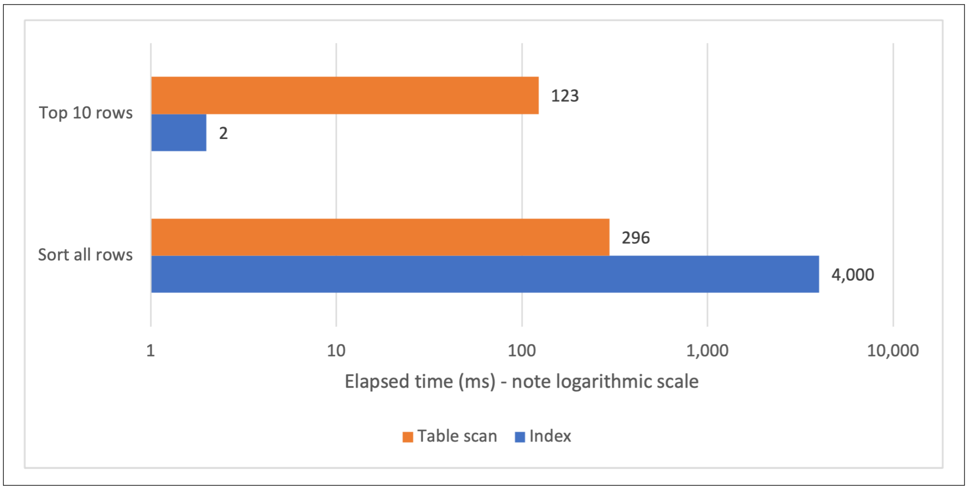


Figure 5-11. Indexes and sort performance

When a LIMIT clause was added to the query, then the index reduced execution time from 123ms to just 2ms – a fantastic improvement. However, if we force CockroachDB to use the index to retrieve all rows (something it won't do by default), then execution time increased from 296ms to 4,000 ms. ==== Spatial indexes

A spatial index is a special type of inverted index that supports operations on the GEOMETRY and GEOGRAPHY two-dimensional spatial datatypes.

Spatial indexing is a complex topic, and we aim only to introduce you to some key concepts here. For more details consult the CockroachDB documentation set[3].

To create a spatial index, we add the USING GIST(geom) clause:

```
CREATE INDEX geom_idx_1 ON some_spatial_table USING GIST(geom);
```

We can furthermore fine-tune the index using various spatial index tuning parameters[4]:

```
CREATE INDEX geom_idx_1 ON geo_table1 USING GIST(geom) WITH (s2_level_mod=3);
CREATE INDEX geom_idx_2 ON geo_table2 USING GIST(geom) WITH (geometry_min_x=0, s2_max_level=15)
CREATE INDEX geom_idx_3 ON geo_table3 USING GIST(geom) WITH (s2_max_level=10)
CREATE INDEX geom_idx_4 ON geo_table4 USING GIST(geom) WITH (geometry_min_x=0, s2_max_level=15);
```

We don't recommend that you change these default tuning parameters lightly; the default values will generally provide the best performance.

HASH Sharded indexes

Earlier in this chapter (???), we showed how in a distributed database, monotonically increasing primary keys can lead to "hot spots" in a distributed database. We recommended using Hash Sharded indexes as a way of avoiding such an issue for monotonically increasing primary keys.

These sorts of hot spots don't occur just in primary keys. Any indexed column that is monotonically increasing might end up with all new values in a single range, and thus creating a scalability and throughput issue.

If you have indexed columns where the value is constantly increasing (timestamps are a good example) and you want to avoid such an insert hotspot, then you should consider hash sharding the index. The syntax is the same as for the primary key example we showed in ???. For instance, to create a hash sharded index on the MODIFIEDDATE column, we might do the following:

```
SET experimental_enable_hash_sharded_indexes=on;

CREATE INDEX orderdetails_hash_ix
    ON orderdetails(modifieddate)
 USING HASH WITH BUCKET_COUNT=6;
```

Note that while CockroachDB will not usually optimize a sort with a hash sharded index, it still might provide good enough performance for a "top 10" type of query. We can force the use of the hash sharded index to perform an ORDER BY using an **index hint** (more on this in chapter 8):

3 *https://www.cockroachlabs.com/docs/stable/spatial-indexes.html*

4 *https://www.cockroachlabs.com/docs/latest/spatial-indexes.html#index-tuning-parameters*

```
SELECT *
  FROM orderdetails@orderdetails_hash_ix
 ORDER BY modifieddate LIMIT 10;
```

CockroachDB will retrieve the top 10 from each “bucket” and amalgamate the results on the gateway node. The result might still be a marked improvement over a full scan.

Measuring Index effectiveness

Having created an index, we'd like to be sure that an index is being used to optimize our query and discover exactly how much benefit we have achieved. We can do this using the EXPLAIN and EXPLAIN ANALYZE commands.

EXPLAIN reveals to us the CockroachDB optimizer “plan” for an SQL statement. We'll dig into EXPLAIN in detail within Chapter 8, but for now, let's just quickly see how they work to explain your query performance.

EXPLAIN reveals the optimizer's plan for resolving a query. For instance, if we created an index on PEOPLE and wanted to see if the query would use it, we could issue the following command:

```
EXPLAIN
SELECT phonenumber
  FROM people
 WHERE lastname='Smith'
   AND firstname='Samantha'
   AND state='California';
                                                                                      info
-----------------------------------------------------------------------------
  distribution: local
  vectorized: true

  • filter
  │ estimated row count: 63
  │ filter: state = 'California'
  │
  └── • index join
      │ estimated row count: 5
      │ table: people@primary
      │
      └── • scan
          estimated row count: 5 (0.02% of the table;)
          table: people@people_lastfirst_ix
          spans: [/'Smith'/'Samantha' - /'Smith'/'Samantha']
```

We can see that the PEOPLE_LASTFIRST_IX will be used to resolve the query.

However, in some cases, we might still not be sure if the index improved execution time. If we use EXPLAIN ANALYZE, then CockroachDB will execute the operation and will report on the amount of IO and other operations that occurred:

```
EXPLAIN analyze
SELECT phonenumber
  FROM people
 WHERE lastname='Smith'
   AND firstname='Samantha'
   AND state='California';
                                          info
-----------------------------------------------------------------------------
  planning time: 2ms
  execution time: 4ms
  distribution: local
  vectorized: true
  rows read from KV: 6 (598 B)
  cumulative time spent in KV: 3ms
  maximum memory usage: 30 KiB
  network usage: 0 B (0 messages)

  • filter
  │ cluster nodes: n1
  │ actual row count: 1
  │ estimated row count: 63
  │ filter: state = 'California'
  │
  └── • index join
      │ cluster nodes: n1
      │ actual row count: 3
      │ KV rows read: 3
      │ KV bytes read: 430 B
      │ estimated row count: 5
      │ table: people@primary
      │
      └── • scan
          cluster nodes: n1
          actual row count: 3
          KV rows read: 3
          KV bytes read: 168 B
          estimated row count: 5 (0.02% of the table)
          table: people@people_lastfirst_ix
          spans: [/'Smith'/'Samantha' - /'Smith'/'Samantha']
```

EXPLAIN has some additional advanced features that we'll learn about in Chapter 8. However, you can see how easy it is to use EXPLAIN to simply determine index utilization and effectiveness.

SUMMARY

In this chapter, we looked at design principles for a CockroachDB database schema. A sound data model right is an essential foundation for a performant and maintainable CockroachDB database.

Database modeling typically proceeds in two stages: logical modeling followed by physical modeling. The aim of the Logical modeling phase is to identify the data required for application functionality. The physical modeling phase attempts to construct a data model that can meet functional requirements together with performance and availability requirements. The physical model should almost never be a direct copy of the logical model.

Database design for a distributed SQL database like CockroachDB creates some unique challenges when compared with a traditional monolithic database. In particular, primary keys should be constructed so that new rows are distributed equitably across the nodes in the cluster. The UUID datatype can achieve this, but if an ascending primary key is required, then using hash sharded primary key indexes are indicated.

We also looked at indexing choices for a CockroachDB database design. Creating the least number of composite indexes to support common filter conditions is our objective. We may also want to create some indexes to support sort operations.

Now that we've learned how to create a data model, we are in a position to start writing application code. We've already introduced CockroachDB SQL: in the next chapter, we'll see how to use CockroachDB SQL in application development frameworks.

[[Ch06 – Application design and implementation]]

CHAPTER 6

Application design and implementation

Like all databases, CockroachDB responds to requests from application code. How an application requests and uses data has a huge bearing on application performance and scalability. In this chapter, we'll review how an application should work with CockroachDB – including best practices for coding CockroachDB requests and transactional models.

Because CockroachDB is PostgreSQL protocol-compatible, any language that supports PostgresSQL can be used with CockroachDB. And in general, the programming idioms and best practices of PostgreSQL apply to CockroachDB. However, because CockroachDB behaves differently at a server level than PostgreSQL, there are some differences in programming styles between CockroachDB and PostgreSQL.

Although you can work with CockroachDB using pretty much any programming language in common use, in this chapter, we'll constrain our discussion to these four languages: Go, Java, Python and JavaScript.

In Chapter 3, we showed how to install language drivers for each of these languages. Please refer back to Chapter 3 for instructions on driver installation, or refer to the CockroachDB documentation [1] for more detailed guidelines, including guidance on how to install drivers for other languages or for alternative drivers.

1 *https://www.cockroachlabs.com/docs/stable/hello-world-example-apps*

COCKROACHDB PROGRAMMING

Performing CRUD operations

We provided basic “Hello world” examples for each language back in Chapter32. Let’s extend those examples to perform some non-trivial “CRUD” operations – Create, Read, Update, Delete.

Programming drivers differ in terms of vocabulary, but they generally adopt a similar grammar. The fundamental operations in a database program are:

- The driver establishes a **connection** object representing a connection to the database server.
- The connection object is used to create **statements**, that represent commands that can be submitted to the databases
- Some statements return **Result Sets** that can be used to iterate through tabular output returned by SELECT statements, DML statements that include a RETURNING clause and some other statements that return results.

Here we see this basic pattern in Java:

```
package helloCRDB;

import java.sql.*;

public class example1 {

  public static void main(String[] args) {

    try {
      Class.forName("org.postgresql.Driver");
      String connectionURL = "jdbc:" + args[0];
      String userName = args[1];
      String passWord = args[2];

      Connection connection = DriverManager.getConnection(connectionURL,
userName, passWord);
      Statement stmt = connection.createStatement();
      stmt.execute("DROP TABLE IF EXISTS names");
      stmt.execute("CREATE TABLE names (name String NOT NULL)");
      stmt.execute("INSERT INTO names (name) VALUES('Ben'),('Jesse'),('Guy')");

      ResultSet results = stmt.executeQuery("SELECT name FROM names");
      while (results.next()) {
        System.out.println(results.getString(1));
        System.out.println(results.getString("NAME"));

      }
```

```
        results.close();
        stmt.close();
        connection.close();

      } catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
      }

    }

  }
```

We create a single **Connection** object and a single **Statement** object, then the statement those to execute multiple SQL commands. When we execute a query, we create a **ResultSet** object that we can use to iterate through results. Finally, we close all these objects.

Note that we can retrieve column values from the ResultSet object by position or by name – both styles are illustrated in the above example.

Below we see similar logic for Python. The **cursor()** method of the connection object creates a cursor object that can be used to execute a statement or navigate through a result set.

```
import psycopg2
import sys

def main():

  if ((len(sys.argv)) !=2):
    sys.exit("Error:No URL provided on command line")
  uri=sys.argv[1]

  connection = psycopg2.connect(uri)
  cursor=connection.cursor()
  cursor.execute("DROP TABLE IF EXISTS names")
  cursor.execute("CREATE TABLE names (name String NOT NULL)")
  cursor.execute("INSERT INTO names (name) VALUES('Ben'),('Jesse'),('Guy')")
  cursor.execute("SELECT name FROM names")
  for row in cursor:
    print(row[0])
  cursor.close()
  connection.close()

main()
```

Here we do the same thing in a NodeJS JavaScript program:

```
const CrClient = require('pg').Client;

async function main() {
    try {
        if (process.argv.length != 3) {
            console.log(`Usage: node ${process.argv[1]} CONNECTION_URI`);
            process.exit(1);
        }

        const connection = new CrClient(process.argv[2]);
        await connection.connect();

        await connection.query('DROP TABLE IF EXISTS names');
        await connection.query('CREATE TABLE names (name String NOT NULL)');
        await connection.query(`INSERT INTO names (name)
                                VALUES('Ben'),('Jesse'),('Guy')`);

        const data = await connection.query('SELECT name from names');
        data.rows.forEach((row) => {
            console.log(row.name);
        });
    } catch (error) {
        console.error(error.stack);
    }
    process.exit(0);
}

main();
```

We've used the "async/await" style for handling asynchronous database requests. You can also use callbacks or promises if that is your programming style. The node-postgres driver documentation[2] contains examples of using each of these programming styles.

Finally, let us look at how we'd perform the same task in go:

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/jackc/pgx"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Fprintln(os.Stderr, "Missing URL argument")
        os.Exit(1)
    }
    uri := os.Args[1]
```

2 *https://node-postgres.com/features/queries*

```
    conn, err := pgx.Connect(context.Background(), uri)
    if err != nil {
        fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)
    }
    execSQL(*conn, "DROP TABLE IF EXISTS names")
    execSQL(*conn, "CREATE TABLE names (name String NOT NULL)")
    execSQL(*conn, "INSERT INTO names (name) VALUES('Ben'),('Jesse'),('Guy')")

    rows, err := conn.Query(context.Background(), "SELECT name FROM names")
    if err != nil {
        fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)
    }
    defer rows.Close()
    for rows.Next() {
        var name string
        err = rows.Scan(&name)
        fmt.Println(name)
    }
}

func execSQL(conn pgx.Conn, sql string) {
    result, err := conn.Exec(context.Background(), sql)
    if err != nil {
        fmt.Fprintf(os.Stderr, "CockroachDB error: %v\n", err)
        os.Exit(1)
    }
    fmt.Fprintf(os.Stdout, "%v rows affected\n", result.RowsAffected())
}
```

We created the `execSQL` function in the Go example to modularize the repetitive error checking involved in the initial SQL statements, though in production code, we would perform error checking independently for each query.

Cursors

A **cursor** is an object that allows you to scroll through the results of a query rather than retrieving all the data in one hit. Cursors are a preferred means of dealing with large amounts of data since they avoid the necessity of holding the complete result set in memory and allow you to abort query processing if you actually only want the first few rows.

For instance, let's say we have a web application that displays blog posts ordered by time. The key query might look something like this:

```sql
SELECT post_timestamp, summary
  FROM blog_posts ORDER BY post_timestamp DESC
```

We have a covering index on POST_TIMESTAMP, and this index stores the SUMMARY column, so we can retrieve rows efficiently in order. We display our data a page at a time, so we might code a nodeJs routine something like this (to render the first page with ten items):

```
const sql = `SELECT post_timestamp, summary
            FROM blog_posts ORDER BY post_timestamp DESC`;
// let rows=await connection.query(sql);
const data = await connection.query(sql);
for (let i = 0; i < 10; i++) {
    console.log(data.rows[i]);
}
```

The problem with this code is that we need to pull all of the contents of the table across the network just to display the first ten rows.

A cursor lets us pull data on demand so that we don't need to pull the second and subsequent pages of data until or unless needed.

To use cursors in NodeJS pg driver, we have to install the associated **pg-cursor** package:

```
const Cursor = require('pg-cursor');
```

Having done that, we can then define a cursor object and pull rows from it on-demand:

```
const cursor = await connection.query(new Cursor(sql));

 cursor.read(10, (err, rows) => {
     console.log(rows);
     if (err) {
         throw err;
     }
 });
```

The response time implications are significant: for a 5 million row table, the cursor implementation returned the first ten rows in 13ms, compared to 14,556 ms for the vanilla query implementation. Figure 6-1 compares the two approaches.

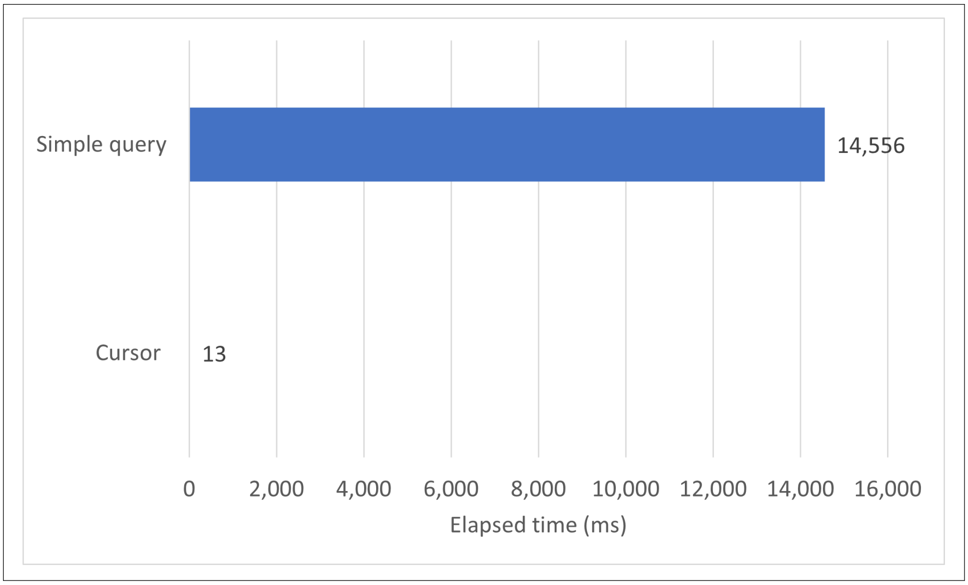


Figure 6-1. Comparison of NodeJs fetch options

Cursors vs. LIMIT

While cursors allow you to pull data from a query on-demand, LIMIT restricts the absolute amount of data requested from the database on the server-side.

The two implementations can result in an identical outcome and often similar performance characteristics. However, if you know for certain that the number of rows to be returned is limited, then using LIMIT is by far the more effective strategy. With LIMIT, the optimizer knows in advance that only a subset of rows will be returned and can pick a better plan. For instance, the optimizer might choose to use an index to avoid a sort **only** if it knows that not all of the data in the result set will be returned.

However, you can't "paginate" with LIMIT – there's no efficient way to request second and subsequent pages of information. For this reason, the use of cursors is preferred when possible.

JDBC Result set objects do not give you the option of fetching all rows in one operation (eg, there's no equivalent to the Python **fetchall()** function. However, that doesn't mean that the Java driver is pulling rows across the network one at a time. Under the hood, the JDBC driver retrieves rows in batches whose size is controlled by the **setFetchSize()** method of the Statement object. By default, fetchSize is set to 0, which results in **all** the rows being pulled into the application before the first row can be processed.

We can adjust the fetchSize if we want to pull only a few rows in each batch as follows[3]:

```
Statement stmt = connection.createStatement();
stmt.setFetchSize(100);
 results = stmt
     .executeQuery("SELECT post_timestamp, summary "
                 + "  FROM blog_posts "
                + "  ORDER BY post_timestamp DESC ");
 for (int ri = 0; ri < 10; ri++) {
   if (results.next())
     System.out.println(results.getString("SUMMARY"));
 }
```

You don't have to change your loop logic when you change setFetchSize, but under the hood, the PostgreSQL Driver will pull rows in batches of **setFetchSize()** size. Figure 6-2 shows that this can be very effective if we want to optimize for fetching the first few rows.

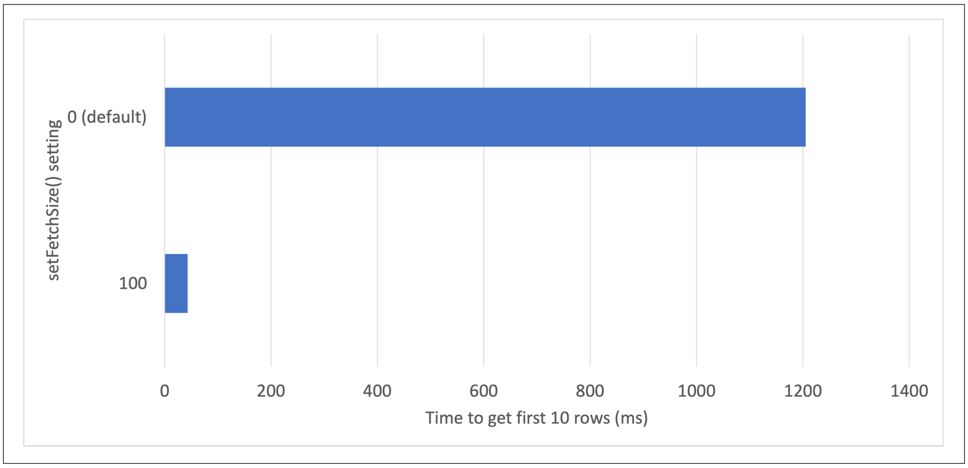


Figure 6-2. Reducing setFetchSize to improve fetch time for first rows

Cursors in Go and Python

In the pgx Go driver, rows are accessed sequentially as the Next() function is called on the results. However, all the rows are moved into program memory when the query() method is called, so even if you only access the first row, you will pull all rows across the network.

[3] Note that the setFetchSize() call has no effect if setAutoCommit is set to true

The Python psycopg2 driver provides methods access to the entire result set (fetchall()), a selection of rows (fetchmany()) or a single row (fetchone()). However, regardless of the method called, the entire result set is always transferred from the database to the application.

These restrictions are characteristics of the drivers themselves and are not specific to CockroachDB – they apply equally to PostgreSQL applications.

For these drivers, you may want to use LIMIT or otherwise prevent massive result sets being transferred unnecessarily. The CRDB team is working with the authors of these packages with an aim to improve their efficiency .

Prepared and parameterized statements

Most SQL operations are parameterized – the same statement is run multiple times with different input parameters. For instance, we might have a lookup program that retrieves rider names for a specified ride id as follows:

```
SELECT u.name FROM movr.rides r
  JOIN movr.users u ON (r.rider_id=u.id)
 WHERE r.id='ffc3c373-63ec-43fe-98ff-311f29424d8b'
```

We would of course, execute this SQL many times, each time specifying a different value for the Rider Id.

When coding a generic lookup function, it seems natural enough to append the parameter to the SQL statement using string concatenation. For instance, in Java we might be tempted to do something like this:

```
private static String getRiderName(String riderId) throws SQLException {
Statement stmt = connection.createStatement();
        String sql = " SELECT u.name FROM movr.rides r "
                   + "  JOIN movr.users u ON (r.rider_id=u.id) "
                   + " WHERE r.id='"
+ riderId + "'";
        ResultSet rs = stmt.executeQuery(sql);
        rs.next();
        return (rs.getString("name"));
        }
```

However, as natural as this might seem, it represents an extremely poor practice that has both performance and security downsides.

Most significantly, this code is vulnerable to **SQL Injection**. For instance, imagine the application could somehow be persuaded to pass the following string to the function:

```
riderName = getRiderName(
"ffc3c373-63ec-43fe-98ff-311f29424d8b' UNION select credit_card from movr.users order by 1,name 'r
```

The resulting SQL statement would become:

```
SELECT u.name FROM movr.rides r
  JOIN movr.users u ON (r.rider_id=u.id)
 WHERE r.id='ffc3c373-63ec-43fe-98ff-311f29424d8b'
 UNION select credit_card from movr.users order by 1,name
```

And the function would now return credit card numbers as well as rider names.

Of course, the application should prevent such a string from being entered at the user interface layer, but creating the vulnerability in the database code is very poor practice.

The solution is to use **prepared** or **parameterized** statements. For instance, in the Java example above, we would declare a **preparedStatement** as follows:

```
getRiderStmt = connection.prepareStatement(
   "SELECT u.name FROM movr.rides r "
 + " JOIN movr.users u ON (r.rider_id=u.id) "
 + " WHERE r.id=?");
```

The "?" indicates a placeholder for a parameter (sometimes called a *bind variable*). We can call the prepared statement by setting the parameter and executing the statement:

```
getRiderStmt.setString(1, riderId);
ResultSet rs = getRiderStmt.executeQuery();
rs.next();
return (rs.getString("name"));
```

As well as avoiding SQL injection, preparedStatements generally execute faster because CockroachDB can more easily recognize the SQL as one that has already been parsed and can avoid some of the overhead involved with examining what would otherwise appear to be a brand new statement.

Formally "preparing" statements is a Java practice. In other languages, it's sufficient to simply call a SQL statement with placeholders and provide the values in the call. For instance, in JavaScript:

```
const sql = `SELECT u.name FROM movr.rides r
              JOIN movr.users u ON (r.rider_id=u.id)
             WHERE r.id=$1`;
const results = await connection.query(sql, ['ffc3c373-63ec-43fe-98ff-311f29424d8b']);
console.log(results.rows[0].name);
```

In Python:

```
sql = """SELECT u.name FROM movr.rides r
          JOIN movr.users u ON (r.rider_id=u.id)
         WHERE r.id=%s"""
cursor.execute(sql,('ffc3c373-63ec-43fe-98ff-311f29424d8b',))
row=cursor.fetchone()
print(row[0])
```

And in Go:

```
    sql := `SELECT u.name FROM movr.rides r
             JOIN movr.users u ON (r.rider_id=u.id)
            WHERE r.id=$1`
    rows, err := conn.Query(ctx, sql, "ffc3c373-63ec-43fe-98ff-311f29424d8b")
    rows.Next()
    var name string
    err = rows.Scan(&name)
    fmt.Println(name)
```

Connection Pools

In a microservices architecture, we create small routines to perform small tasks. If the service requires database access, then it might seem natural to supply each microservice request with a dedicated connection. This has a clear advantage over a single shared connection since it allows for concurrent requests. For instance, imagine that we have a simple web service that we call whenever a new ride is commenced in our Uber-busting ride-sharing app. We might code the database logic for it as follows:

```
async function newRide(city, riderId, vehicleId, startAddress) {
    const connection = new pg.Client(connectionString);
    await connection.connect();
    const sql = `INSERT INTO movr.rides
    (id, city,rider_id,vehicle_id,start_address,start_time)
    VALUES(gen_random_uuid(), $1,$2,$3,$4,now())`;
    await connection.query(sql, [city, riderId, vehicleId, startAddress]);
    await connection.end();
}
```

We don't want to single thread these requests, so we've given each call its own connection. Unfortunately, creating a connection has a non-trivial overhead. When the database access is very simple, the time taken to create and dispose of the connection might dominate overall response time. But we can't run every request through the same connection because that would restrict concurrent queries.

The solution is to use **Connection Pools**. A connection pool is a set of connections that can be reused by the application. You avoid the overhead of constantly creating and destroying connections, and you can control the maximum amount of concurrency hitting the database.

In NodeJS, we'd create the pool as follows:

```
const pool = new pg.Pool({
    connectionString,
    max: 40
});
```

We now can change our routine so that it gets connections from the pool:

```
async function newRidePool(city, riderId, vehicleId, startAddress) {
    const connection = await pool.connect();
    const sql = `INSERT INTO movr.rides
```

```
                (id, city,rider_id,vehicle_id,start_address,start_time)
                VALUES(gen_random_uuid(), $1,$2,$3,$4,now())`;
    await connection.query(sql, [city, riderId, vehicleId, startAddress]);
    await connection.release();
}
```

Figure 6-3 illustrates how the two approaches compare for performance. With 40 concurrent requests, a connection pool implementation outperformed the unique connection approach by about 700%.

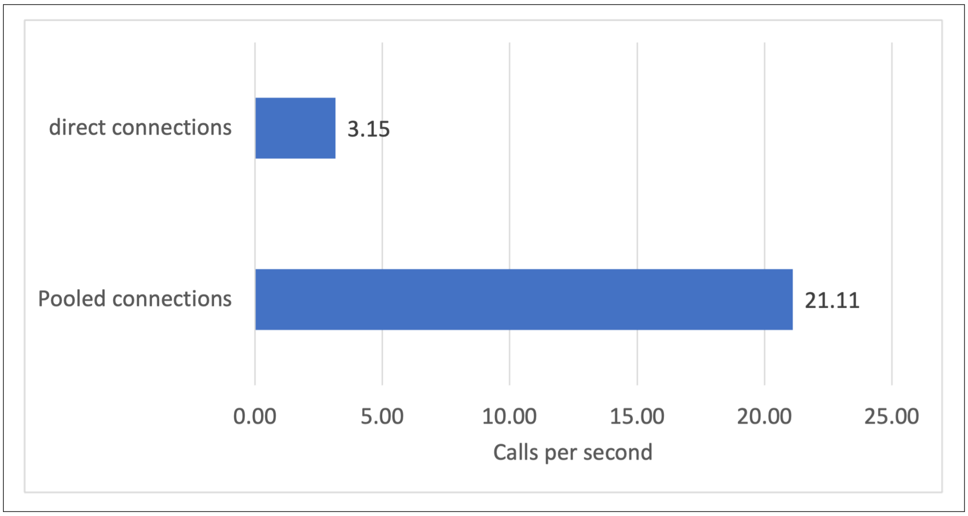


Figure 6-3. Using connection pools to improve concurrency

The amount of benefit you get from connection pools will vary depending on the amount of work performed in each connection and the amount of concurrent activity the application issues. However, it's almost always advisable to use a connection pool in preference to a single connection used by all threads or allocating each thread with its own transitory connection.

Connection Pools and blocked connections

Most connection pool implementations will block requests for new connections if all the pooled connections are in use. Therefore, it's important to configure a sufficient number of connections in the pool for the anticipated concurrency. A common rule of thumb is to configure four connections for every core in the entire cluster. For instance, if you have a three-node cluster with eight cores in each node, you might configure 3*8*4=96 connections. However, bear in mind that this is just a guideline – the optimal number will depend heavily on the duration of each connection and the amount of idle time each connection experiences as the application performs non-database tasks.

It's also critically important to release connections when not in use. For instance, in the NodeJS example, the *connection.release()* statement at the end of our function is very important.

In Java, there are a variety of connection Pool options[4]. Here's an example using the Hikari framework[5]:

```
import com.zaxxer.hikari.*;
import java.sql.*;

public class ConnectionPoolDemo {

  public static void main(String[] args) {
    try {
      Class.forName("org.postgresql.Driver");
      String connectionURL = "jdbc:" + args[0];
      String userName = args[1];
      String passWord = args[2];

      HikariConfig config = new HikariConfig();
      config.setJdbcUrl(connectionURL);
      config.setUsername(userName);
      config.setPassword(passWord);
      config.addDataSourceProperty("ssl", "true");
      config.addDataSourceProperty("sslMode", "require");
      config.addDataSourceProperty("reWriteBatchedInserts", "true");
      config.setAutoCommit(false);
      config.setMaximumPoolSize(40);
      config.setIdleTimeout(3000);


      HikariDataSource hikariPool = new HikariDataSource(config);
```

This example creates a connection pool with 40 connections using arguments passed in on the command line. Once the pool is created, a connection can be obtained from the pool as follows:

```
Connection connection = hikariPool.getConnection();
```

In the Go pgx driver, we can use the pgxpool package to create and use a connection pool:

```
ctx := context.Background()
config, err := pgxpool.ParseConfig(uri)
config.MaxConns = 40
```

4 For instance, see *https://www.baeldung.com/java-connection-pooling*

5 *https://github.com/brettwooldridge/HikariCP*

```
	pool, err := pgxpool.ConnectConfig(ctx, config)
	defer pool.Close()
```

We can acquire a connection from the pool as follows:

```
	connection, err := pool.Acquire(ctx)
```

Psycopg2 for Python includes a built-in connection pool which we can easily configure as follows:

```
import psycopg2
from psycopg2 import pool

def main():

  if ((len(sys.argv)) !=2):
    sys.exit("Error:No URL provided on command line")
  uri=sys.argv[1]

  pool= psycopg2.pool.ThreadedConnectionPool(10, 40, uri)
  # min connection=10, max=40
```

And we can connect to the pool as follows:

```
connection = pool.getconn()
```

Bulk inserts

It's very common for an application to insert multiple rows of data into a single logical operation.

When you have an array of values to insert, it can seem natural to simply insert the values in a loop, as in this python example:

```
for value in arrayValues:
    cursor.execute("INSERT INTO insertTestP1(id,x,y) VALUES ($1,$2,$3)",value)
```

It's very inefficient to insert large amounts of data one at a time – each insert will require a network round trip, and there may be transactional implications if we want the entire batch to be inserted in a single truncation (by taking longer, the chance of a transaction conflict and subsequent retry will be magnified).

SQL allows multiple VALUES to be included in a single operation, for instance:

```
INSERT INTO insertTest(id,x,y)
VALUES (3,'x',1) ,
       (4,'y',2) ,
       (5,'x',5)
```

So we could, if necessary, dynamically construct an INSERT statement to insert an array of data in a single operation. For instance, in Python, the following code will generate and execute an INSERT statement to insert an array of arbitrary length:

```
sql="INSERT INTO insertTestP(id,x,y) VALUES"
valueCount=0
for value in arrayValues:
    if valueCount>0:
        sql=sql+","
    sql=sql+"(%d,'%s',%d)" % value
    valueCount+=1
cursor.execute(sql)
```

In the psycopg2 extras package, there is an execute_extras helper function that simplifies the coding required:

```
from psycopg2 import extras

<snip>

extras.execute_values(cursor,
 "INSERT INTO insertTestP1(id,x,y) VALUES %s",
 arrayValues)
```

The performance improvements obtained with array inserts are dramatic.

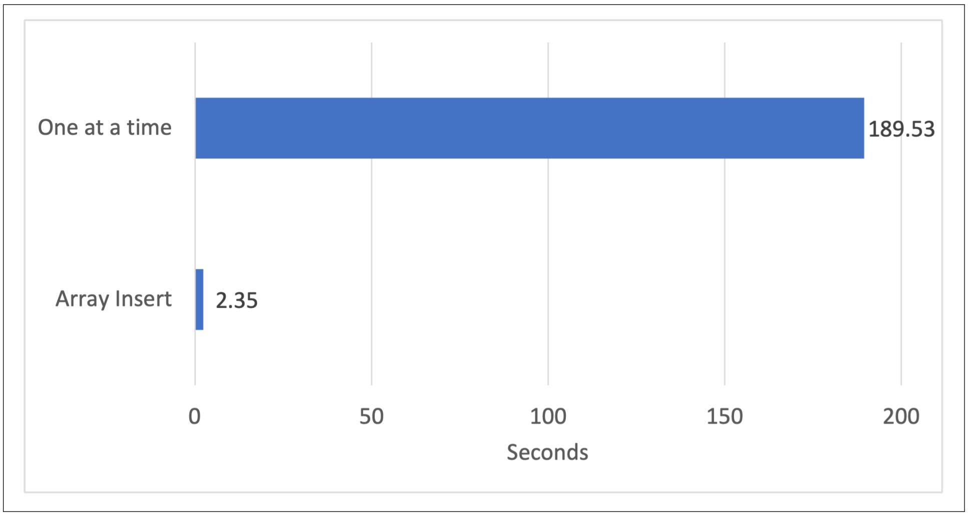


Figure 6-4. Improvement obtained by inserting rows in an array

JDBC includes **addBatch** and **executeBatch** methods that allow you to prepare inserts one at a time and then submit all the inserted values in a single operation. This avoids the need to concatenate a huge VALUES list and allows us to use formal parameters.

Here's an example of the **addBatch** and **executeBatch** methods:

```
        String sql="INSERT INTO insertTest(id,x,y) VALUES (?,?,?)";
        PreparedStatement InsertStmt = connection.prepareStatement(sql);
        for (int arrayIdx = 1; arrayIdx < arrayCount; arrayIdx++) {
            InsertStmt.setInt(1, idArray.get(arrayIdx));
            InsertStmt.setString(2, xArray.get(arrayIdx));
            InsertStmt.setInt(3, yArray.get(arrayIdx));
            InsertStmt.addBatch();
            }

        InsertStmt.executeBatch();
```

We use **setInt** and **setString** methods to supply values to the prepared statement as usual, but instead of executing, we use **addBatch** to add them to the batch of rows to be inserted. When we are ready, we call **executeBatch** to add all the rows in a single operation.

The pg NodeJS library does not include any direct support for batch inserts. However, we can use the **pg-format** package to create SQL statements that contain multiple VALUES from an array:

```
const pg = require('pg');
const format = require('pg-format');

async function main() {
    const connection = new pg.Client(connectionString);

    const sql = format('INSERT INTO insertTestP2(id,x,y) VALUES  %L,
arrayData);

    await connection.query(sql);
```

The Go pgx library does not currently support bulk inserts directly. You would need to construct dynamic SQL with multiple VALUES entries, as shown earlier for Python. However, some users have written helper functions to expedite dynamic SQL generation for bulk inserts[6].

Projections

In relational database parlance, "projection" refers to the selection of a subset of columns from a table (or *attributes* from an *entity*). In practice, a projection is represented by the list of columns in a SELECT clause.

While SELECT accepts a wildcard projection "**", this should almost never be used in production code since it results in unnecessary transport of columns from the database to the application. Using "**" can seem like a handy programming shortcut, but it can have severe performance penalties when processing large result sets.

6 *https://github.com/jackc/pgx/issues/764#issuecomment-685249471*

For instance, let's say we are retrieving a list of user ids and blog post dates to populate a dashboard or to perform some other real-time diagnostic. The following code might seem acceptable:

```
        ResultSet results = stmt.executeQuery(
  "SELECT * FROM blog_posts");
        while (results.next()) {
          java.sql.Timestamp postTimestamp =
  results.getTimestamp("POST_TIMESTAMP");
          Integer userid = results.getInt("USERID");
          plotPost(userid, postTimestamp);

        }
```

However, a couple of coding seconds saved in omitting the column names costs the application dearly. Every time this code is executed, it retrieves not only the user id and timestamp, but also the potentially very large blog post text. As a result, each network packet can hold less data, and the number of network round trips is magnified. If we add a projection:

```
        results = stmt.executeQuery(
  "SELECT userid,post_timestamp FROM blog_posts");
```

Then elapsed time is reduced dramatically. Figure 6-5 illustrates the elapsed time savings for a ten million row result set from a remote cluster.

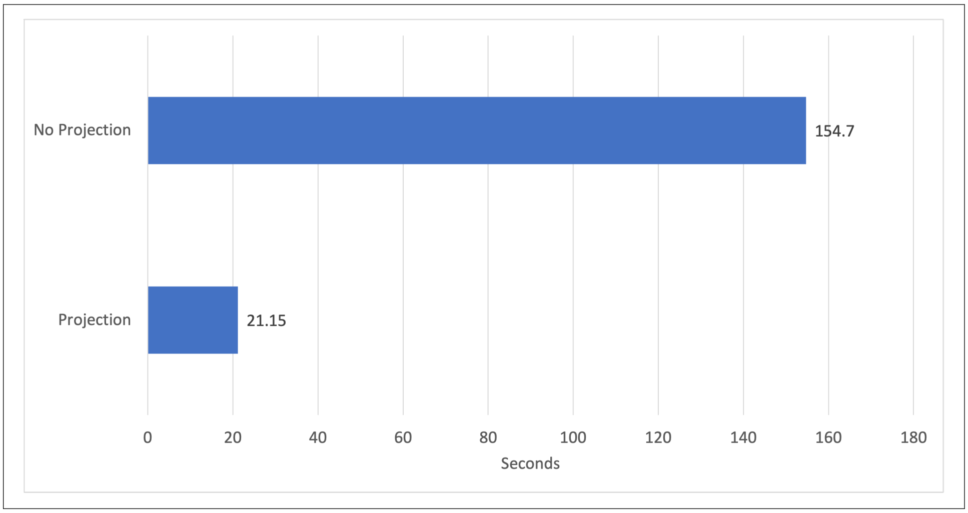


Figure 6-5. Improvement obtained by adding a projection to a query

Of course, the absolute time saved will depend on the total row size versus the size of the projection and the network latency between the application and the server. Furthermore, this degradation only kicks when we pull more rows from the database that can fit in a single network packet. For single-row retrievals, the overhead is negligible.

Client side caching

The best way to optimize a database request is not to send it all. No matter how carefully we optimize the database – adding indexes, memory, fast disks, etc. –database requests are blocking operations that can never be made as fast as local computation. For most applications, database accesses are the slowest operations performed and the most critical component of application response time.

One of the most effective ways of avoiding unnecessary database calls is to cache frequently accessed static data in application code. Avoid asking the database over and over again for the same data unless there's a chance that the data will change.

Golang example

MANAGING TRANSACTIONS

Transactions provide an important mechanism for ensuring that related modifications succeed or fail as a unit. We discussed the internals of CockroachDB transactions back in Chapter 2.

The basics of programming transactions are common across a wide variety of SQL databases and even some non-SQL systems. A transaction is commenced with a BEGIN statement. Multiple SQL statements are executed within the transaction scope, and then all the changes are made permanent with the COMMIT statement. If an error is encountered during the transaction, all of the transaction's work can be abandoned with a ROLLBACK statement.

Here's a relatively simple transaction consisting of an INSERT followed by an UPDATE – implemented in NodeJS JavaScript:

```
async function takeMeasurement(locationId, measurement) {
    let success = false;
    const measurementTime = new Date();
    const connection = await pool.connect();
    try {
        await connection.query('BEGIN TRANSACTION');
        await connection.query(
            `INSERT INTO measurements
                                (locationId,measurement)
                                VALUES($1,$2)`,
            [locationId, measurement]
        );
        await connection.query(`UPDATE locations
                                   SET last_measurement=$1, last_timestamp=$2
                                 WHERE id=$3`,
            [measurement, measurementTime, locationId]);
        await connection.query('COMMIT');
        success = true;
    } catch (error) {
        console.error(error.message);
        connection.query('ROLLBACK');
        success = false;
    }
    connection.release();
    return (success);
}
```

Transaction Retry errors

In databases that implement lower levels of transaction isolation (MySQL, for instance), this transaction would almost always succeed, perhaps failing only if there was a database outage. However, in a database like CockroachDB that implements SERIALIZABLE transaction isolation, there is a very good chance of transaction failure. If a concurrent transaction modifies the same LOCATION table row between the time our transaction commences and the time we attempt to modify that row, then we will encounter a *TransactionRetryWithProtoRefreshError: WriteTooOldError* (We'll call this a **TransactionRetry** error for the sake of brefity brevity).

Figure 6-6 illustrates a typical sequence of events in two concurrent transactions that would lead to a transactionRetry error.

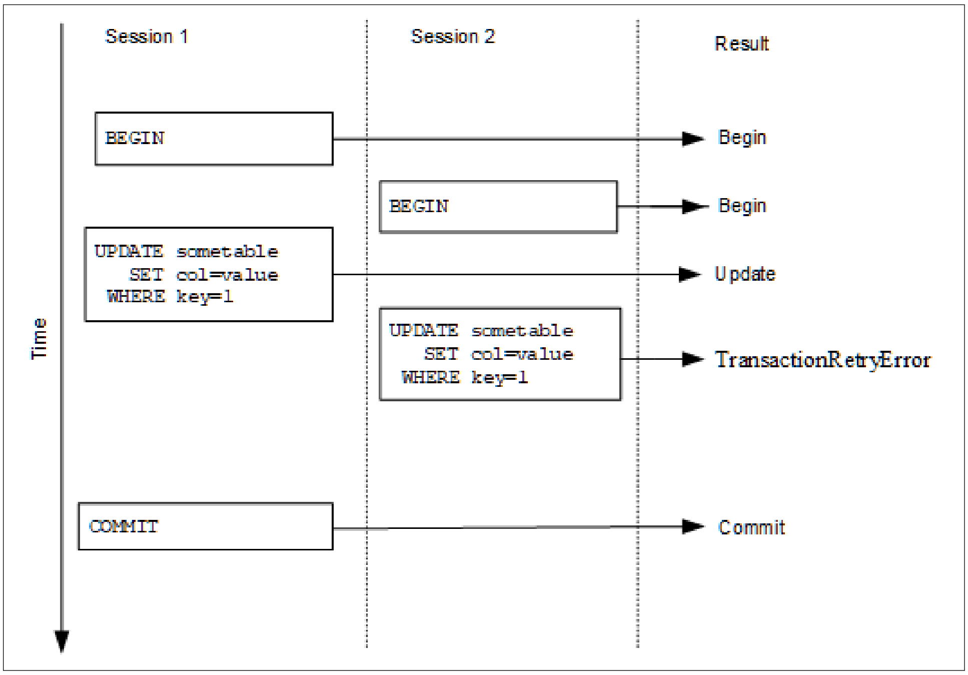


Figure 6-6. Transaction Retry error scenario

The chance of receiving a TransactionRetry error depends on the chance of two transactions colliding on the same row. For our above example, the percentage of retries varied from less than 1% if there were 10,000 distinct locations to more than 75% when there were just ten locations[7].

However, whatever the possibility of encountering a transaction Retry error, the possibility exists, and your application code should be able to cope with these expected error scenarios.

Implementing transaction retries

The relatively obvious way to handle retries errors is to do exactly what the error code suggests – retry the transaction. When the TransactionRetry error is encountered, issue a ROLLBACK command to discard the work done so far in the transaction and try the transaction again.

Here are some modifications to our JavaScript NodeJS method to retry the transaction when necessary:

```
async function takeMeasurementWithRetry(locationId, measurement, maxRetries) {
    const connection = await pool.connect();
    const measurementTime = new Date();
    let retryCount = 0;
    let transactionEnd = false;
    while (!transactionEnd) {
        retryCount += 1;
        if (retryCount >= maxRetries) {
            throw Error('Maximum retry count exceeded');
        } else {
            try {
                await connection.query('BEGIN TRANSACTION');
                await connection.query(
                    `INSERT INTO measurements
                                        (locationId,measurement)
                                        VALUES($1,$2)`,
                    [locationId, measurement]
                );
                await connection.query(`UPDATE locations
                                   SET last_measurement=$1, last_timestamp=$2
                                 WHERE id=$3`,
                    [measurement, measurementTime, locationId]);
                await connection.query('COMMIT');
                transactionEnd = true;
            } catch (error) {
                if (error.code == '40001') { // Rollback and retry
                    connection.query('ROLLBACK');
                    const sleepTime = (2 ** retryCount) * 100
```

7 The simulation ran 100 concurrent threads randomly executing a transaction 10 times a second

```
                                + Math.ceil(Math.random() * 100);
                        console.warn('Sleeping for ', sleepTime);
                        await sleep(sleepTime);
                    } else {
                        console.log('aborted ', error.message);
                        transactionEnd = true;
                    }
                }
            }
        }
        connection.release();
        return (retryCount);
    }
```

If this method encounters an error *40001* – the retry transaction code – it issues a ROLLBACK, waits for a short time and then tries the transaction again.

In this implementation, the sleep time increases exponentially as the number of retries increases. This is done to avoid a situation in which transactions "thrash" on a resource. This exponential backoff strategy tends to reduce the load on a busy system, but it can result in some very high transaction waits for "unlucky" transactions. Furthermore, when we retry transactions, there is no guarantee that updates will succeed in the order in which they are originally submitted. Transactions that are submitted first may only succeed after transactions submitted at a later date.

Automatic transaction retries

The logic shown in the previous section can be implemented in any language[8]. However, some drivers implement this logic for you transparently:

- The GoLang dbtools library includes a transaction retry handler for Go transactions. You pass a set of operations to the transaction handler which will automatically retry transactions with a configurable retry limit and delay[9].
- Many Object Relational Mapping frameworks – SQLAlchemy for Python, for instance – will automatically retry transactions for you transparently.

Avoiding transaction retry errors with FOR UPDATE

Performing transaction retries has some significant downsides. Firstly, they are wasteful since work in the transaction that is done prior to the retry is discarded. Secondly, they introduce a delay in transaction processing that is unpredictable or even unnecessary. It's hard to know how long to sleep between transaction retries, and exponen-

8 See *https://www.cockroachlabs.com/docs/stable/transactions#client-side-intervention-example* for a generic implementation

9 *https://pkg.go.dev/github.com/arsham/dbtools*

tial backoffs can lead to some extreme waits. Finally, transaction retries result in non-deterministic behaviors. Transactions will not necessarily be applied to the database in the order in which they are submitted by the application, and even under identical workloads, differences in outcomes will be observed.

The alternative to the transaction retry approach is to "lock" the rows required at the beginning of the transaction with a FOR UPDATE statement. FOR UPDATE is a blocking statement, and once it returns, your transaction has the update rights over the rows concerned.

Here's our sample code with the FOR UPDATE logic:

```
async function takeMeasurementForUpdate(locationId, measurement) {
    let success = false;
    const connection = await pool.connect();
    const measurementTime = new Date();
    try {
        await connection.query('BEGIN TRANSACTION');
        await connection.query(`SELECT id FROM locations
                                 WHERE id=$1
                                   FOR UPDATE`, [locationId]);
        const insertReturn = await connection.query(
            `INSERT INTO measurements
                                (locationId,measurement)
                                VALUES($1,$2) RETURNING (measurement_timestamp)`,
            [locationId, measurement]
        );
        await connection.query(`UPDATE locations
                                   SET last_measurement=$1, last_timestamp=$2
                                 WHERE id=$3`,
            [measurement, measurementTime, locationId]);

        await connection.query('COMMIT');
        success = true;
    } catch (error) {
        console.error(error.message);
        connection.query('ROLLBACK');
    }
    connection.release();
    return (success);
}
```

By locking the LOCATIONS row as soon as the transaction begins, we avoid any chance of a transaction retry being issued. However, in a production implementation, it is probably advisable to include a transactionRetry error handler in any transaction .

Optimistic vs. pessimistic transaction design

The two patterns for transactions we've looked at here – retry handling versus FOR UPDATE locking – have historically been referred to as **optimistic** versus **pessimistic** transaction models.

In the optimistic transaction model, we feel it is unlikely that there will be a conflicting update that will cause a transaction to abort. Therefore, we don't "pre-lock" data and rely on transaction retries to handle any conflicts that might occur.

In the pessimistic model, we are quite worried about transaction conflicts, so we preemptively lock rows that might come into conflict.

Neither model is superior – it really does depend on how likely row-level transaction conflicts are. Don't choose one or the other based on your emotional disposition – think carefully about the likelihood of conflicts – benchmark if necessary - and act accordingly.

Time travel queries

Deadlocks

Nested transactions

Transaction priorities

Other options for avoiding conflicts

Both transaction retries and FOR UPDATE locking introduce delays into your application which are undesirable. A transaction retry burns up time during the rollback and sleep following a retry error, while FOR UPDATE will block other transactions that seek to update the contended resource.

The core problem to be avoided is contention on a single row. These "hot spots" should be avoided in database design. If the contention exists for multiple columns then placing each column in it's own **column family** for instance. We might also want to consider splitting up

• Keep transactions short • Update highly contentious rows first • Partition the data •

tracking retries

Working with ORM Frameworks

While many applications work directly with CockroachDB via SQL statements embedded in application code, other applications use frameworks that avoid the direct use of SQL and instead leverage automated mapping of database tables to program objects. In this section, we'll introduce some of the most popular and provide an example of their use.

SQLAlchemy for Python

Django for Python

Java Hibernate

Java JOOQ

GORM for GoLang

TypeORM for NodeJS

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 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