# 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 the ability to successfully compete in the wild, but it was only when we developed data storage – e.g., written language – that each generation was able to 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), are genuinely analogous to the databases that support modern accounting systems.



Figure 1-1 Cuneiform tablet circa 3000BC [[1]](#footnote-2)

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 .

### 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 system 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]](#footnote-3). 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 needlessly 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.

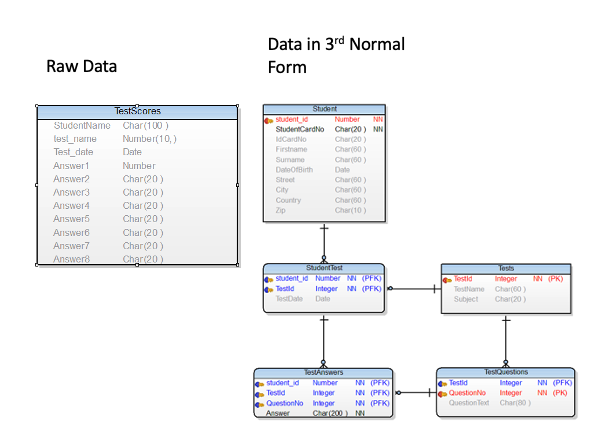


Figure 1-2 Data represented in a relational structure.

### 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 performance 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 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 behaviour 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]](#footnote-4). 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.

**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 sever 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.

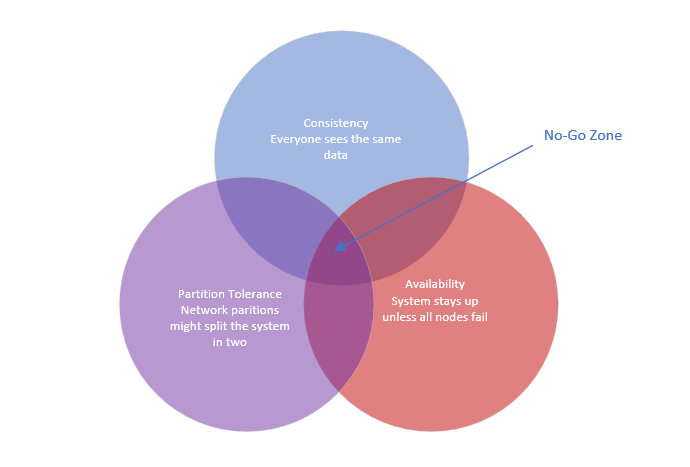
As the web grew, the limitations of the centralized RDBMS became untenable. The emerging "Web 2.0" social network and e-commerce sites had two characteristics that were increasingly difficult to support:

* 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]](#footnote-5) states that you can only have at most two of three desirable characteristics in a distributed system:

* **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.

Figure 1-4 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 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 lead directly to Apache Hadoop, and Google BigTable lead directly 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.

[NOTE]

====

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, availably 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 honour of the humble Cockroach who, it is told, is so resilient that it would survive even a nuclear war (Figure 1-5).



Figure 1-5 The original CockroachDB logo

### The CockroachDB advantage

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 workloads. 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 have diverged from Spanner in several important ways.

Firstly, Spanner was designed to run on very specific hardware. Spanner nodes include 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 leapt all the way 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.

2020's 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 accurage 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]](#footnote-6)!

### 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.

Baidu have 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 reduced complexity for both application developers and database administrators. Developers no longer needed to route database requests through the sharding middleware and could take advantage of distributed transactions and SQL operations. Administrators benefited 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 was 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]](#footnote-7):

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.

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.

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. Asides from the resiliency and scalability of CockroachDB, the ability of CockroachDB servers 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, for many or most applications, they are a step backwards. 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.

It is a highly available, transactionally consistent SQL database that is 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 design goals.

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> [↑](#footnote-ref-2)
2. <http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> [↑](#footnote-ref-3)
3. http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf [↑](#footnote-ref-4)
4. <https://dl.acm.org/doi/10.1145/564585.564601> [↑](#footnote-ref-5)
5. Cockroach labs maintains a growing list of CockroachDB case studies at <https://resources.cockroachlabs.com/customers>. [↑](#footnote-ref-6)
6. <https://www.cockroachlabs.com/blog/cassandra-to-cockroachdb/> [↑](#footnote-ref-7)