# 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 last generation of database systems - strong consistency, the power of SQL, and the relational 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 higher scalability and availability.

In this chapter we’ll review the history of Database Management Systems and understand how CockroachDB pulls together the technology advances of the last few decades to deliver on its ambitious goals.

## A Brief History of Databases

Data storage and data processing is one of 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, are genuinely analogous to the databases that support modern accounting systems.



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 computers has truly resulted in a 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 (DBMS).

### 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 a specialized and dedicated software platform – the first Digital Databases.

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** and **IDMS**.

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, 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:

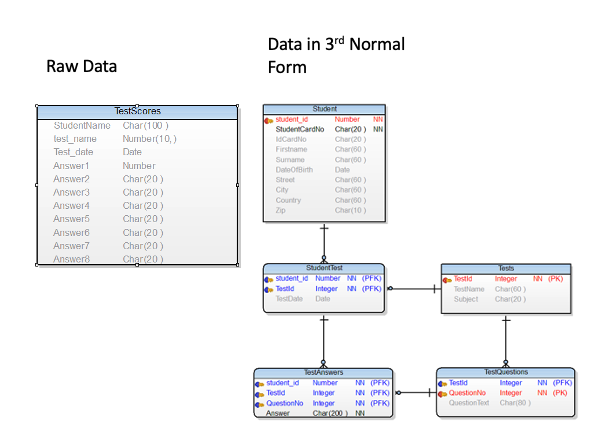
* Existing DBMS systems merged physical and logical representations of data in a way that often complicated requests for data and created difficulties in satisfying requests that were not anticipated during the physical design.
* There was no formal standard for data representation. As a mathematician, 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 path for data retrieval.

The 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 therefore represents a two dimensional dataset with a fixed number of attributes and arbitrary number of tuples.
* **Constraints** enforce consistency and define relationships between tuples.
* Various **Operations** are defined, such as joins, projections, unions, and so on. Operations on relations always return relations. 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.



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 hardly 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 physical layout of the data is independent from the logical model.

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

### Transactions

A transaction is a logical unit of work that must succeed to fail as a unit. Transactions predated the relational model, but in Codd’s relational model, the database took formal repsonbility for transactional processing. In Codd’s formulation a relational system would provide explicit support for commencing a transaction and either committing, or aborting a transaction.

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

Codd’s relational model did not define all the aspects of transactional behavior that became common to most relational database systems.

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.

### 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.

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**).

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.

The RDBMS came into prevalence at around the same time as another seismic paradigm shift. 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.

### 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 traditional 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 to simultaneously access 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[[3]](#footnote-6) 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 unelss 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.



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 challanges 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" 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 the SQL supported transactional processing but provided only the most limited subset of SQL functionality. In no circumstance was a full set of SQL features provided.

However, the problems caused by eventual 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 result in 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 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 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 therefore 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 honor of the humble Cockroach who, it is told, is so resilient that it would survive even a nuclear war.



Figure 1 The original CockroachDB logo

* **Geo-partitioning**: CockroachDB allows data to be physically located in specific localities to enhance performance for “localized” applications and to respect data sovereignty requirements.
* **prioritizes consistency over availability**

### CockroachDB Architecture

Like Spanner, initial versions of CockroachDB were based on Key-Value data and access models, but the Cockroach team rapidly transitioned to a SQL/Relational model (the Key-Value store acts as a storage engine for the SQL layer).

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 that 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.

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.

… Version 21 and maybe 22 ?

### CockroachDB in action

CockroachDB immediately gained traction in the crowded database market. User who had been frustrated with the scalability of existing relational databases such as PostgreSQL and MySQL were attracted by the greater scalability of CockroachDB. Those who were using distributed NoSQL solutions such as Cassandra where attracted by the greater transactional consistency offered by CockroachDB. And those who were transforming towards modern containerized and cloud-native architectures appreciated 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 to see what CockroachDB can do!

#### Cockroach at MyWorld

MyWorld is a next-generation virtual world company. They are developing a framework to provide developers with modern platform providing fast scalable and extensible services for MMOGs s (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 implementation. As founder Daniel Perano explained[[4]](#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 the data more naturally, and rely on multi-table transactions and constraints to maintain data consistency. CockroachDB’s Postgres compatibility was another benefit.

#### CockroachDB at Baidu

##### Example of ShardedSQL -> CockroachDB

They were relying on MySQL to do the job with multiple shards and middleware to support critical applications. Baidu’s DBA team, however, wanted to try a different approach for a new application that needed to store increasing amounts of data while supporting continuous inserts with highly concurrent and real-time access. This application also needed secondary indexes to speed up queries, as well as support for basic real-time analytics to extract insights from existing data. Their existing MySQL deployment would require application developers to transform

and modify data at the application layer, while NoSQL databases that sacrificed secondary indexes, aggregations, and transactions would similarly introduce complexity for application developers. For applications that needed scalable SQL, Baidu’s DBA team had to stick with a relational database. It was time to invest in a different database.

<https://content.cdntwrk.com/files/aT0xMjEwNTAzJnY9MSZpc3N1ZU5hbWU9YmFpZHUtY2FzZS1zdHVkeS1jb2Nrcm9hY2hkYiZjbWQ9ZCZzaWc9MWFiNjcyOWY1ZDA0Njg2ODg5YWZjYTNmZWMxMjRkZTg%253D>

#### CockroachDB at Bose

Example of K8S and microsevices

<https://resources.cockroachlabs.com/case-study-video/bose>

#### CockroachDB at DoorDash

Example of adopter seeking to push performance envelope

Postgres compatibility made CDB “familiar”

Looking for growth and geo-region

Needed transactions and serializable isolation

## Summary

In this chapter we’ve placed CockroachDB in an historical context and introduced the goals and capabilities of the CockroachDB database.

The Relational Database Management Systems (RDBMS) that emerged in the 1970 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 step backwards 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 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 how it that architecture allows it to deliver on 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. <https://dl.acm.org/doi/10.1145/564585.564601> [↑](#footnote-ref-6)
4. <https://www.cockroachlabs.com/blog/cassandra-to-cockroachdb/> [↑](#footnote-ref-7)