Introduction to CockroachDB

# Introduction to CockroachDB

CockroachDB is an open-source, distributed, transactional, relational SQL database system. That's quite a mouthful! But one way to look at it is that 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 systems that allow CockroachDB to achieve global scale and continual availability.

To understand the strengths of CockroachDB, it's worth revisiting the evolution of database systems. We'll see that CockroachDB is the latest in a succession of technology advances that make it a compelling addition to the database ecosystem.

## A Brief History of Databases

Data storage and data processing is a core feature of human civilization. The earliest written records - dating back 10,000 years - represented agricultural accounting records. These cuneiform records, recorded on clay tablets, are genuinely analogous to the databases that support modern accounting systems such as Xero.



Cuniform table circa 3000BC [[1]](#footnote-1)

However, today we generally use the term database to refer to a collection of information stored using digital computing technology - specifically a DataBase Management System (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 application of its own. Consequently, the first DBMS systems emerged.

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-2). This paper outlined what Codd saw as fundamental issues in the design of existing DBMS systems:

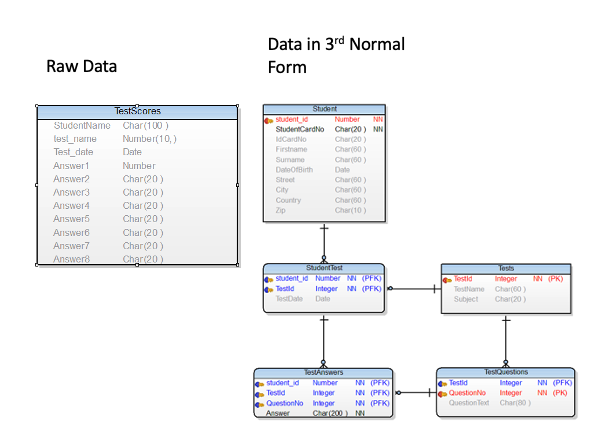
* Existing DBMS systems merged physical and logical representations of data in a way that often complicated requests for data and created difficulties in satisfying requests that were not anticipated during the physical design.
* There was no formal standard for data representation. As a mathematician, Codd was well versed in formal data structures; he felt these structures had a role in Data management systems.
* Existing DBMS systems were too hard to use. To fulfil their potential, DBMS systems needed to be accessible to those without advanced programming skills.

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. In an actual database system, a tuple corresponds to a **row** and an attribute to a column **value**.
* A **relation** is a collection of distinct tuples and corresponds to a **table** in relational database implementations.
* **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. In practice, this means that the output of a SQL query returns data in a table-like structure.

The relational model furthermore defines a series of "normal forms" that represent reducing levels of redundancy in the model. A table is in third normal form if all data in a row is dependent on the entire primary key of that row 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 performance data model. We'll come back to Third Normal Form in Chapter 5.



Data represented in a relational structure

### 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'll devote Chapter 4 to the CockroachDB SQL implementation.

### ACID transactions

The relational model and the SQL language represented two important foundations for the emerging databases of the early 1980s. The **ACID transaction model** represented the final piece of the puzzle.

All databases have to handle concurrent data change requests in a way that balances **consistency** with **concurrency**. In 1981 Jim Gray articulated the core principles of transaction processing that we still use today[[3]](#footnote-3). These principles later became known as ACID – atomic, consistent, isolated and durable – transaction processing.

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 required that one transaction should not be able to see the effects of other in-progress transactions.

### 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), though it's important to realize that the relational model is just one of the three important pillars of almost all RDBMS systems.

By the end of the 20th century, the RDBMS reigned supreme. The leading databases of the day – Oracle, Sybase, SQL Server, Informix and DB2 all competed around performance, functionality or price, but all were virtually identical in their adoption of the relational model, SQL and ACID transactions.

It helped that 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 favour of the new breed of RDBMS.

### Enter the Internet

During the first decade of the 21st 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.

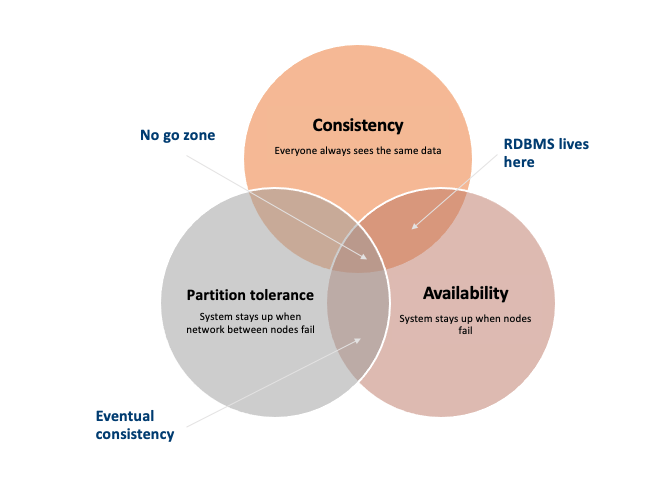
As the web grew, the limitations of the centralized RDBMS became untenable. The emerging "Web 2.0" social network and e-commerce sites had two characteristics that were increasingly difficult to support:

* These systems had a global or near-global scale. Users in multiple continents would 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 go 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 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 even if some elements of the distributed system fail.
* **Partition Tolerance**: the system remains available even if a network partition divides 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 have to 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 one of the partitions to shutdown.

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.

It's worth noting, however, that there is nothing about the SQL language or the relational model that is affected by the CAP theorem. It was the ACID transaction model – not the relational model or SQL - that needed to be sacrificed in order to achieve perfect availability at a global scale.

### The NoSQL movement

Between 2008-2010 dozens of new database systems emerged, all of which abandoned the three pillars of the RDBMS – 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.

Some 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 the "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 more catchy term "NoSQL" databases.

NoSQL was always a very questionable term. It defined what a class of systems discarded, rather than their unique distinguishing features. Furthermore, it created a focus on the SQL language as a problem, when it was generally the interaction between ACID transactions and the CAP theorem that these new breeds of systems sought to overcome.

[NOTE]

====

It was not problems with SQL that lead to most NoSQL databases. Rather it was the inability of ACID transactions to cope with global scale and availability that broke the dominance of the RDBMS model.

====

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 implications of the CAP theorem, 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 fairly 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 many provided some SQL-like dialect – leading to the redefinition of NoSQL as "Not Only SQL".

However, the problems caused by eventual consistency were harder to ignore. Consistency and correctness in data are usually far greater more than "nice to have". 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 was unthinkable. Advanced non-relational databases adopted tunable consistency and sophisticated conflict resolution algorithms to mitigate data inconsistency. However, any database that abandoned strict consistency could produce scenarios in which data could be lost or corrupted during the reconciliation of network partitions or ambiguously timed transactions.

Google had pioneered many of the technologies behind important NoSQL technologies such as Hadoop and HBase and was as aware as anyone of the limitations of these new data stores. Engineers at Google wondered if there was not a way to overcome the CAP theorem; perhaps there was a way to "stretch" it far enough to provide Good Enough Availability without sacrificing consistency. The result was the Spanner project.

CAP theorem assumes that network partitions are inevitable in a wide area network. And in the universal wide area network of the internet, this is undoubtedly true – you simply can't assume network availability when the network is constructed of so many varied service providers. But Google had the advantage of a highly redundant internal global network. Google's network had sufficient redundancy to eliminate hardware failure as a likely cause of a network partition. Spanner was designed in such a way as to reduce the chance of a network partition as close to zero as was practically possible.

Another other novel feature of Spanner is 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 antennas and atomic clocks physically installed in each server. 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 the same clock time. This allows Spanner to order transactions and queries precisely without requiring inter-node communication.

[NOTE]

====

Spanner is highly dependent on Google's redundant network and specialized server hardware. Spanner can't operate idependently 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 evolved to a distributed database that supported all three pillars of the RDBMS – the SQL language, relational data model and ACID transactions.

## Hello 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.

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's 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 can't rely on atomic clock synchronization. As we will see in Chapter 2, Spanner does rely on decent clock synchronization between nodes but is far more tolerant of clock skew that Spanner.

Secondly, Spanner relies on Google's highly redundant network to avoid network partitions. CockroachDB is network-agnostic so needs to be able to respond to partitions as they occur. The CockroachDB architecture is designed to distribute data across nodes in such a way to survive most network partitions.

Thirdly, while the distributed storage engine of CockroachDB is inspired by Spanner, the SQL engine and APIs are designed to be PostgreSQL compatible. 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 – not surprisingly called 1.0 - 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 wasn't 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.

The 21 release included

## CockroachDB advantages and deployments

CockroachDB has been designed to offer the following desirable 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 requiring 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.
* **Performant**: CockroachDB
* Compatible
* Global
* Portable

The above might seem a bit like an advertisement and maybe too good to be true. However, all of the features above are demonstrably present in the CockroachDB database, which does indeed provide a lot of advanced functionality. In the interests of balance, let's look at what CockroachDB is not:

* **CockroachDB is not completely immune to network partitions.** 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.
* **CockroachDB does not aspire to be an analytics platform.**  CockroachDB

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-1)
2. <http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> [↑](#footnote-ref-2)
3. http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf [↑](#footnote-ref-3)