White Paper

Migrating From Monolithic To Cloud Native Operational Databases

How YugabyteDB Provides
Multi-Cloud Mobility in the Data Layer

yugabyteDB

Table of Contents

Introduction

Enterprises large and small are increasingly migrating their mission-critical applications from legacy, monolithic architectures running inside on-prem data centers to modern microservices architectures running on public, private, or hybrid cloud infrastructure. Application development and technical operations teams leverage a multitude of cloud native infrastructure technologies, such as Docker and Kubernetes, to make this migration possible.

However, one major roadblock to a more accelerated migration to microservices and cloud infrastructure is an inability to move the stateful data layer with the stateless application layer at the same velocity. This stateful data layer consists of a persistent database and an in-memory cache that fronts the database. In the context of databases, it is well known that SQL-based relational databases are not suitable for dynamic cloud infrastructure. On the other hand, NoSQL and NewSQL databases look cloud ready from a distance, but they, too, present significant operational challenges when run at scale in production.

In this white paper, we highlight the causes and effects of these industry defining trends, especially in the context of operational databases. We also introduce YugabyteDB, a cloud native, strongly consistent, distributed SQL database built from the ground-up for today's mission-critical applications. YugabyteDB provides the much desired cross-cloud operational mobility in the data layer while simultaneously making it extremely easy for enterprise developers to write applications.

The Evolution Of Operational Databases

While application architectures and infrastructures underwent massive shifts, operational databases powering mission-critical Online Transaction Processing (OLTP) applications also experienced significant changes, albeit more slowly than the application and infrastructure side.

Monolithic To Sharded And Distributed

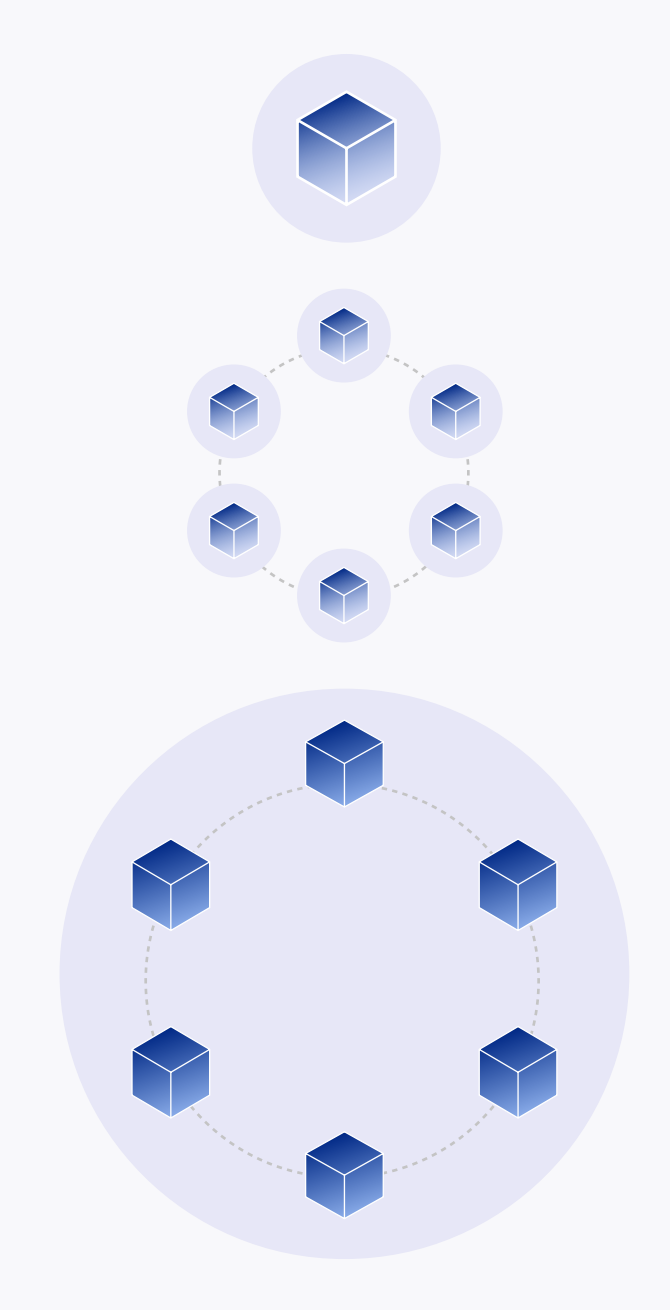
Relational databases, also known as **Structured Query Language (SQL)** databases, have been the de-facto OLTP database standard for the last 40 years. They are architecturally monolithic in nature and run on a single node backed by specialized, high cost hardware with several problematic consequences. They require painful vertical hardware scaling as the application's read/write volume increases.

Adding high availability entails adding a new independent database instance and then copying the data over via either async or sync replication. Async replication, also known as master-standby, involves replicating committed data from master to standby periodically. There can be some data loss when the master fails and the standby hasn't received some of the writes yet.

Availability is also not 100% since there is usually downtime involved in failing over to the standby. Sync replication guarantees zero data loss using atomic write operations implemented via protocols such as 2-phase commit. However, such a system becomes unavailable whenever one of the two instances is down or there is a network problem leading to loss of connectivity between the two instances.

Simple sharding, where data is partitioned across independent databases, is an architectural enhancement that is added to achieve linear scalability. The application must become responsible for identifying which shard has what data. The resulting complexity is often underestimated, especially as the number of shards grows with changing business needs. High availability of each shard is still maintained through async or sync replication, thus adding another layer of operational complexity. Such sharded relational databases are often referred to as **NewSQL.**

On the other hand, **NoSQL**, which stands for Not Only SQL, refers to a class of databases that are distributed, horizontally scalable, sharded and multi- copy replicated by default. The distributed, multi-copy replicated approach provides more write and read availability than both SQL and NewSQL databases. This is because the number of failures tolerated can be tuned by simply increasing or decreasing the replication factor, the number of replicas per shard. NewSQL databases have adopted NoSQL-like distributed architectures with the goal of providing higher availability than older generation databases.



Strong Consistency To Eventual Consistency

The famous **CAP (Consistency, Availability, and Partition-tolerance) theorem** states that network partitions cannot be avoided in distributed systems. Therefore, whenever such network partitions arise, the choice is only between 100% consistency and 100% availability (there is no compromise needed during normal operating conditions). Since partitions are a non-issue in monolithic relational databases, there is no trade-off to be made between consistency and availability. The practical trade-off is lack of scalability. Sharded NewSQL and distributed NoSQL databases solve the scalability problem but now are forced to make a choice between 100% consistency and 100% availability during network partitions. In practice, this choice is based on the primary workload type the database is optimized for.

NewSQL databases are used for traditional "system-of-record" OLTP applications such as a product catalog where there are a smaller number of entities being worked on with a narrow context (such as online checkout). Hence the motivation for strong consistency, also called immediate consistency, where data viewed immediately after an update is consistent for all observers of the entity.

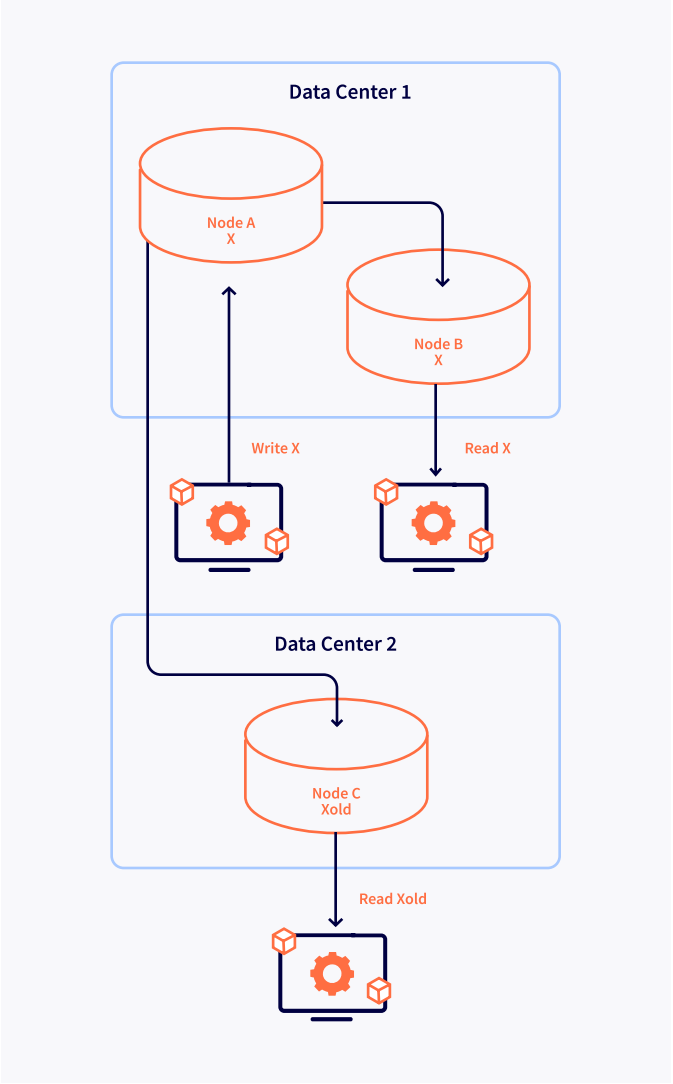
The underlying implementation of strong consistency relies on various kinds of locks and concurrency control for brief periods of time. This allows intermediate and incorrect states of the data that are either avoided altogether or become invisible to client applications. As expected, these databases suffer from lower availability and higher latency during network partitions. NewSQL databases aim for much higher availability while maintaining strong consistency.

On the other hand, traditional NoSQL databases are typically used for one of two kinds of applications: Hybrid Transactional/Analytics Processing (HTAP) and Online Analytical Processing (OLAP).

Both application types require high write availability and can tolerate some degree of stale reads, hence the motivation for eventual consistency. Eventual consistency is a theoretical guarantee that, provided no new updates to an entity are made, all reads of the entity will eventually return the last updated value. In these cases, the inclusion or exclusion of a set of entities is not expected to impact user experience since the overall number of entities is large.

Given this fundamental unpredictability associated with eventual consistency, NoSQL databases are not recommended for mission-critical OLTP and HTAP applications where guaranteed Service Level Agreements (SLAs) are mandatory. Critics consider eventual consistency to be "hopeful" consistency given the hopeful and arguably unrealistic requirement of avoiding new updates altogether.

Traditional NoSQL databases with their eventually consistent cores are increasingly tuned for strong consistency using quorum-based approaches. However, these band-aid solutions come with a clear decrease in performance as well as a significant increase in operational and end user pain. Even the large technology companies that originally served as early NoSQL adopters are now forced to rethink their ever-increasing engineering investment.



ACID To BASE

The consistency and replication architecture of a database's server-side has a direct impact on how client transactions are served. Arguably the most important reason behind the massive adoption of strongly consistent relational databases is the notion of Atomicity, Consistency, Isolation and Durablity (ACID) guarantees with respect to client transactions. These transactions make it simple for developers to reason about and control (by tuning isolation levels) the write/read behaviors expected in their applications.

Applying the lock-based implementation approach behind ACID transactions to distributed NoSQL databases would make them unavailable for extended periods of time. This is unacceptable given the primary need of high availability.

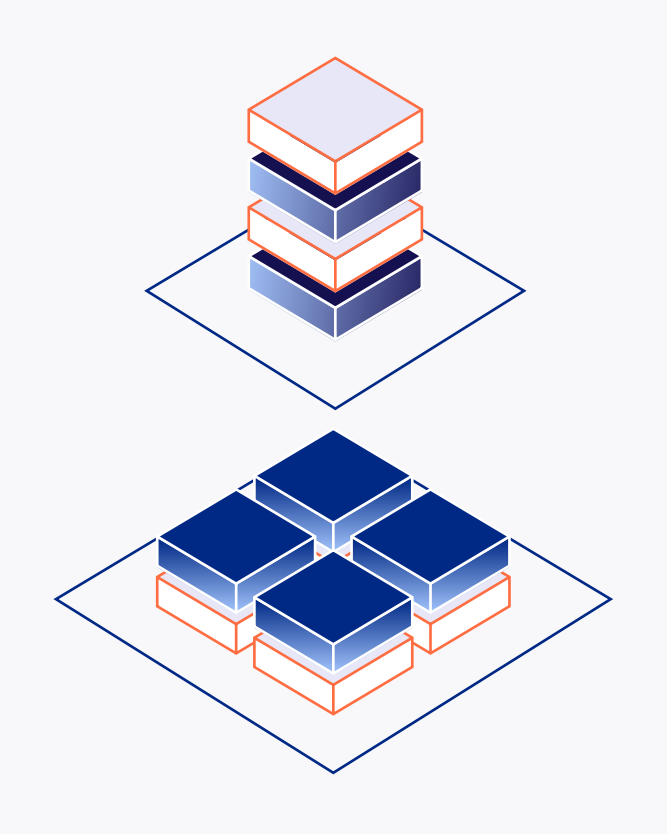
But NoSQL databases instead support **Basically Available Soft-state Eventually Consistent (BASE)** client transactions where higher read and write availability is favored even if the data is stale or dirty. BASE transactions, with isolation levels worse than READ UNCOMMITTED, are unfit to be used in system-of-record applications since it's impossible to say what's the correct value of a record at a given instant of time.

SQL To NewSQL And NoSQL

The above server-side and client-side architectural differences mean that NoSQL databases cannot simply adopt SQL as their client interaction language. SQL allows for multiple rows to be updated and queried (sometimes using JOINs) at the same time.

Unfortunately, those rows are not guaranteed to be in the same shard in NoSQL databases, and reducing availability to coordinate atomicity in multi-shard transactions is not an option. NoSQL databases thus do not support JOINs and have a client language of their own. For example, Apache Cassandra has a SQL variant called Cassandra Query Language, and MongoDB has its proprietary CRUD operations.

NewSQL databases aim to support as much of the traditional SQL syntax as possible including JOINs. These databases are essentially abstraction layers running on top of multiple independent monolithic relational databases. There are also a few use case specific databases for streaming data analytics. The choice of SQL for such non-OLTP use cases is primarily driven by the need to take advantage of SQL's popularity among developers.



Relational To Multi-Model

Relational databases work well for structured data that gets organized as rows in tables with each column having a strongly typed value. Such a rigid schema approach does not work well for applications with semi-structured or unstructured data needs. NoSQL databases solve this problem by providing either a more flexible schema (e.g., column-oriented databases such as Apache Cassandra and Apache HBase) or going schema-less (e.g., document databases such as MongoDB). There are also key-value stores (e.g., Redis) with in-memory data structures but no concept of a persistent store or schema as such.

On-Premises To Cloud Hosted

The distributed architecture and horizontal scaling aspects of NoSQL databases make them well suited for hosted deployments in the cloud where instances can be dynamically provisioned. On the other hand, the monolithic architecture and vertical scaling requirements of relational databases mean that their infrastructure needs are less dynamic. Although they can be hosted in the cloud, on-premises data centers with static capacity work just fine for such cases.

Rise Of Database-As-A-Service (DBaaS)

In the last few years, both leading cloud providers and major database vendors have introduced Database-as-a-Service (DBaaS) offerings. The common theme is that the operational complexity of managing and monitoring the database is no longer the customer's concern. Operations are now the provider's responsibility and the cost is included in the price of the offering. In many cases, the offering is simply a hosted version of an open source database such as Amazon RDS/Aurora or MongoDB Atlas. In a select few cases, the cloud providers also offer proprietary databases as a service, such as Amazon DynamoDB.

Defining "Cloud Native" For Databases

Cloud native is fast becoming a reality for the application layer. However, the same cannot be said about the data layer powering the application. Cloud hosted NoSQL and NewSQL databases can be horizontally deployed on cloud infrastructure but they do not abstract away the various cloud providers while keeping the core database functionality uniform and reliable. Below are the five defining characteristics of a cloud native database.

Extreme elasticity: Scale clusters up and down speedily and reliably.

Geo-redundant, always-on availability: Easily create multi-AZ/multi-region clusters and then expand or shrink availability zones or regions anytime, while remaining resilient to unplanned failures and planned upgrades.

Hardware flexibility: Seamlessly move from one type of compute and storage to another for cost and performance reasons.

Multi-cloud mobility: Avoid cloud lock-in by moving to or co-existing on multiple cloud providers.

Data placement policies: Define and enforce geo-specific data residency controls without impacting the app

Note that DBaaS offerings abstract out the underlying cloud infrastructure and are being marketed as "cloud native." However, enterprises do not consider these services truly cloud native given the inherent cloud vendor lock-in and loss of control on data and servers that comes along with any such service. Additionally, deploying traditional NoSQL or NewSQL databases on stateful containers (such as Kubernetes StatefulSets) solves the cloud provider abstraction problem, but it does not serve the desired end goal of agile development and operations. This is because these traditional databases are not built on a cloud native foundation. The data storage and replication layers of these databases lack the availability and mobility necessary to exploit the full potential of orchestration-ready stateful containers.

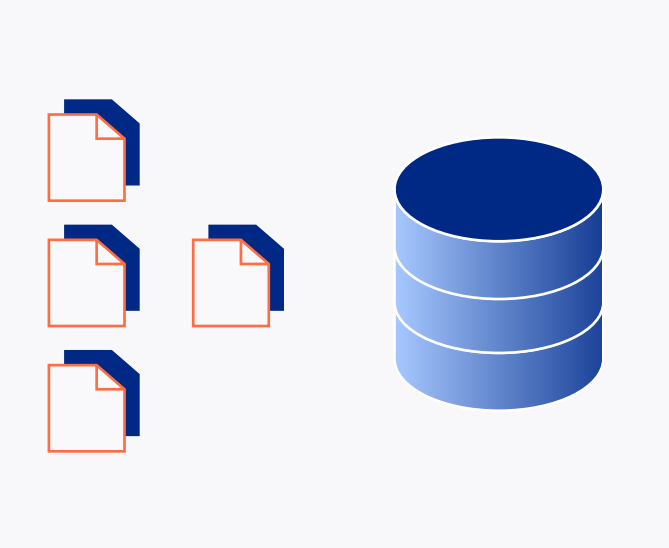
Three Challenges Of NoSQL Databases

In this section, we walk through the three challenges of using current generation NoSQL databases: operational complexity, frustrating application development, and inconsistent customer experiences.

Operational Complexity

As noted in the previous section, databases have evolved to become cloud hosted but are far from being truly cloud native. The current operational complexity is painful for organizations looking to exploit the elasticity and geo-redundancy of modern cloud infrastructure to its maximum potential. On top of this, the eventually consistent core of NoSQL adds hidden costs such as performance-killing background repairs and unpredictable, memory-intensive compaction storms. The fact that most enterprises also run an independent caching layer (such as Redis or Memcache) alongside their persistent database simply makes all operational challenges twice as hard.

In context of the original need to move the data layer through the same phases and at the same velocity as the application layer, the above operational challenges simply make such moves next to impossible. If such a move is forced on operations teams, then it means business loss manifested as both unpredictable downtimes and manual, error-prone war rooms.

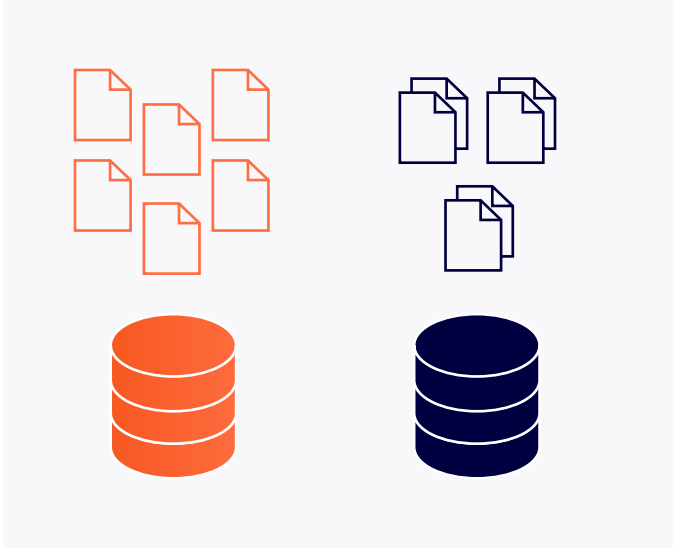


Frustrating Application Development

Application developers desire the simplicity of ACID transactions so that they can easily reason about the read/write behavior of their database client code. Relational databases support multi-row ACID where multiple related rows can be updated or read in an all-or-nothing and consistent manner. However, most NoSQL databases do not even support single-row ACID transactions, since eventual consistency leads to the “C” getting compromised at the remote replicas.

The net result is extremely long development and test cycles to handle all the corner cases necessary to achieve desired read/write behavior. For example, the answer to “what value will be read after a failed write?” varies wildly among NoSQL databases. Additionally, most NoSQL databases either do not support secondary indexes or have extremely poor performing secondary indexes (as a direct result of accessing all shards). Loss of this critical feature also slows down development since complex workarounds must be designed, implemented, and tested.

Many NoSQL databases are starting to realize the advantages of strong consistency and are allowing their eventually consistent systems to be tuned to quorum-based strong consistency settings. However, it is well proven that this form of tunable consistency is not truly strong for many situations, including dirty reads after failed writes and unpredictable reads after the last writer wins. Developers using this approach are forced to spend even more time testing their applications to guarantee predictable behaviors.



Inconsistent Customer Experiences

Even with all the tuning efforts from developers to build strongly consistent OLTP/HTAP applications on eventually consistent NoSQL databases, error conditions are unavoidable. During these error conditions, inconsistency is exposed to end customers.

For example, a few items were deleted from a retailer's product catalog since they were no-longer-available. However, those deletes were not honored when the data was presented to the customer, since the node from which that data was served did not have the deletes applied yet. Another example would be ignoring some of the time-series metric data for calculating aggregates in alerting for time-series monitoring and Internet of Things (IoT) use cases. Waking up team members in the early morning hours based on incorrect data should be avoided. Similarly, if a user's privacy preferences are not immediately honored, there is a possibility her actions will appear to other users in the same account.

Solving NoSQL Challenges With Distributed SQL

A distributed SQL database is a single logical relational database deployed on a cluster of servers. The database automatically replicates and distributes data across multiple servers. These databases are strongly consistent and support consistency across availability and geographic zones in the cloud.

At a minimum, a distributed SQL database has the following characteristics:

- A SQL API for accessing and manipulating data and objects
- Automatic distribution of data across nodes in a cluster
- Automatic replication of data in a strongly consistent manner
- Support for distributed query execution so clients do not need to know about the underlying distribution of data
- Support for distributed ACID transactions

Why Distributed SQL?

Business innovation is putting pressure on traditional systems of record. This is forcing companies to deliver high-value applications and services more quickly while lowering IT costs and reducing risk through compliance.

But these applications—in the form of microservices, born-in-the-cloud applications, and edge and IoT workloads—require a new class of database that is:

Resilient to failures and continuously available: Critical services remain available during node, zone, region, and data center failures as well as system maintenance with fast failover

Horizontally scalable: Operations teams can effortlessly scale out even under heavy load without downtime by simply adding nodes to a cluster, and scale back in when the load reduces

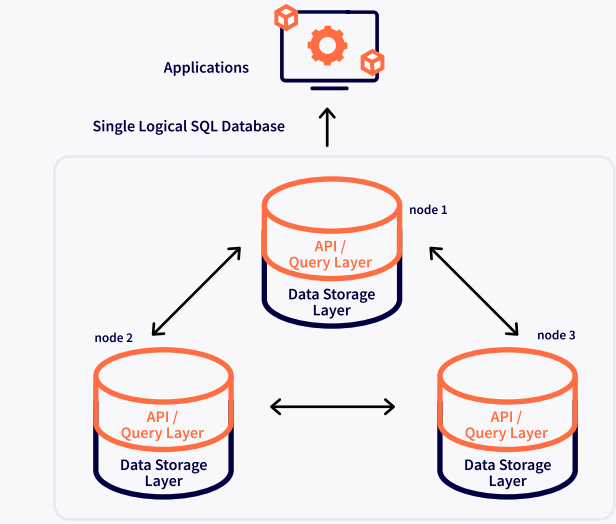
Geographically distributed: Operators can make use of synchronous and asynchronous data replication and geo-partitioning to deploy databases in geo-distributed configurations

SQL and RDBMS feature compatible: Developers no longer need to choose between the horizontal scalability of cloud native systems and the ACID guarantees and strong consistency of traditional RDBMSs

Hybrid and multi-cloud ready: Organizations can deploy and run data infrastructure anywhere—and avoid being locked-in to any specific cloud provider

Distributed SQL Database Architecture

A distributed SQL database provides the best of a traditional RDBMS with cloud native database capabilities. It has a two-layer architecture as part of a single logical SQL database:



SQL Query Layer

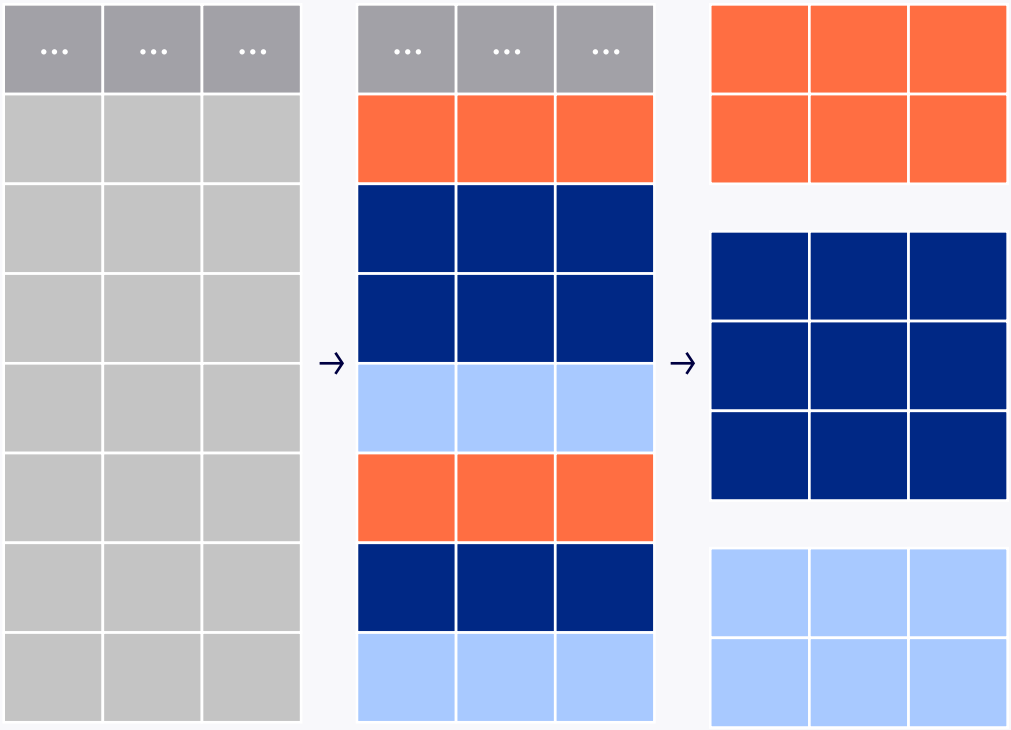
A distributed SQL database has a SQL API for applications to model relational data and also perform queries involving those relations. Queries are automatically distributed across multiple nodes of the database cluster.

Distributed Data Storage Layer

Data, including indexes, in a distributed SQL database are automatically distributed—or sharded—across multiple nodes of the cluster so that no single node becomes a bottleneck to high performance and availability.

Supporting a powerful SQL API layer requires the underlying storage layer to be built on strongly consistent replication across all nodes of the cluster. This means writes to the database are synchronously committed at multiple nodes in order to guarantee availability during failures.

And finally, the database storage layer supports distributed ACID transactions where transaction coordination is required across multiple rows located on multiple nodes.



YugabyteDB: Best-In-Class Distributed SQL For Cloud Native Databases

YugabyteDB is a cloud native distributed SQL database for transactional applications. The database is 100% open source and built to solve availability and resiliency challenges when running application workloads on Kubernetes. It is designed with 3 foundational principles in mind: operational simplicity, developer productivity, and customer delight.

This database uses replicas for high availability and supports synchronization through the use of the Raft protocol. Additional features include partitions (called tablets) for scalability, and in case of a cross-tablet transaction, the two-phase commit protocol is also implemented.

YugabyteDB automatically partitions SQL tables into tablets without user intervention. It also automatically distributes tablet replicas to the configured failure domains ensuring, as much as possible, no data loss. This behavior can be influenced by the user configuring the failure domains, replication factor, and database affinity to failure domains.

Deployment Flexibility

YugabyteDB runs in public, private, and hybrid cloud environments, on VMs, containers or bare metal. Organizations can deploy the database in any Kubernetes environment. It is also available as a multi-cloud, fully managed database-as-a-service (DBaaS) for a frictionless experience. YugabyteDB offers the widest range of replication and geo-distribution options among distributed SQL databases.

High Performance

YugabyteDB can handle high throughput, low latency transactions on Kubernetes. It is proven in production to scale beyond 1 million transactions per second and thousands of concurrent connections.

Operational Simplicity

Organizations can use the self-managed or fully managed DBaaS offerings of YugabyteDB to simplify operations at the edge and in the cloud. The database also integrates with other data sources or sinks, allowing data engineers to build pipelines for machine learning, analytics, long term storage, and disaster recovery.

Multi-API Query Layer

Eliminate database sprawl with a pluggable query layer that supports both Postgres and Cassandra APIs.

Inherently Geo-Distributed

Distribute Data across zones, regions and clouds with ACID consistency. Delivers most complete global replication.

PostgreSQL Compatibility

YugabyteDB is not just wire compatible with PostgreSQL, it is code compatible. The database also offers a comprehensive set of advanced RDBMS features including triggers, functions, stored procedures, and strong secondary indexes. This allows developers to be immediately productive with the familiar interface and the rich ecosystem of PostgreSQL compatible frameworks, applications, drivers, and tools.

Security

YugabyteDB is built from the ground up with data security in mind, enabling organizations to maintain a robust security posture even with a more distributed footprint. YugabyteDB offers features such as data encryption at rest and in flight, multi-tenancy support at the database layer with per-tenant encryption, and regional locality of data to ensure compliance as well as manage geographic access controls.

Feature Comparison

Finally, here's how YugabyteDB compares with other operational databases we discussed earlier in this paper

Feature	Traditional SQL	Traditional NewSQL	Traditional NoSQL	YugabyteDB
Schema	Relational	Relational	Multi-Model	Multi-Model
Language	SQL	SQL	Custom	SQL & CQL
Scalability	Vertical	Horizontal	Horizontal	Horizontal
Consistency	Strong	Strong	Eventual (Tunable)	Strong
Replication	Configurable	Configurable	Automatic	Automatic
Availability	Low	Low	High	High
Transactions	ACID	ACID	BASE	ACID
Caching	Independent	Independent	Independent	Built-in
Cloud Infra	Cloud Hosted	Cloud Hosted	Cloud Hosted	Cloud Native
Target Apps	OLTP	OLTP	HTAP, OLAP	OLTP, HTAP

Conclusion

Enterprises need a multi-model, enterprise-grade OLTP/HTAP data layer that gives them the same cloud native agility they are deriving from microservices and containers. However, current generation solutions aren't serving this need. SQL and NewSQL databases deliver developer productivity through strong consistency, but they add significant operational complexity given the lack of cloud native features and multi-model capabilities. On the other hand, using cloud hosted, eventually consistent NoSQL databases for OLTP/HTAP use cases leads to not just loss of developer productivity, but also increased operational complexity.

YugabyteDB frees application developers and technical operations teams of the painful and long-standing compromises listed above. Built on top of a scale out, cloud native and strongly consistent core, this database allows developers to leverage popular NoSQL languages in the context of customer-facing OLTP/HTAP application.

yugabyteDB

Get in Touch

www.yugabyte.com | contact@yugabyte.com