NuoDB Architecture

This white paper provides an introduction to the NuoDB architecture. It surveys the internals of a database, the management model, and the key differentiators of the technology. This paper is designed to provide the reader a fundamental understanding of, and motivations for, the NuoDB architecture.
Table of Contents

Introduction and Background 3

The NuoDB Architecture 4
  Two Tiers 4
  Peer-to-Peer Caching, Coordinating, and Scale-Out 5
  Atoms: Internal Object Structure 6
  Multi-Version Concurrency Control 7
  Data Durability 9
  A Tunable Commit Protocol 9

The NuoDB Architecture: Examples 10
  Cache Population 10
  Data Update 11
  Commit Protocol 12

Management and Operational Model 13
  Database Backup and Provisioning 14

Benefits of the Architecture 15
  Single, Logical Database 15
  Multi-Data Center Support 15
  Flexible Schemas 16
  Operational and Analytic Mixed Workloads 16
  Multi-Tenancy and Resource Efficiency 17
  Live Upgrade and On-Demand Migration 18
  Reactive High Availability 18
  Table Partitioning Using Storage Groups 19
  Scale-Out Performance 20

Conclusion 21

Learn More 22
Introduction and Background

Traditionally, relational databases were designed for scale-up architectures. Supporting more clients or higher throughput required an upgrade to a larger server. Until recently, this meant that implementing a scale-out architecture either required a NoSQL database, therefore abandoning traditional relational database benefits, or relying on sharding and explicit replication. There were no solutions able to scale out and still provide complete ACID (Atomicity, Consistency, Isolation, and Durability) -compliant semantics. This tension is what inspired the NewSQL movement\(^1\) and ultimately led to today’s modern distributed structured query language (SQL) databases.

In the distributed SQL database model and in modern distributed data centers, on-demand scale-out databases that maintain ACID semantics are an architectural requirement\(^2\). Key features associated with cloud-native are also critical, such as ease of provisioning and management, security, agility in the face of unpredictable workloads or failures, and support for widely distributed applications. Widely distributed applications, in turn, require distributed services that are highly available and able to provide low latency. These are the design goals that define the NuoDB architecture.

NuoDB is a distributed SQL database designed with distributed application deployment challenges in mind. It’s a true SQL service that provides all the properties of ACID-compliant transactions and standard relational SQL language support. It’s also designed from the beginning as a distributed system that scales the way cloud services must scale, providing continuous availability and resiliency with no single points of failure. Unlike traditional shared-disk or shared-nothing architectures, NuoDB’s patented distributed database presents a new kind of peer-to-peer, on-demand independence that provides continuous availability, low-latency, and an easily managed deployment architecture.

Unlike some cloud services or distributed SQL databases, however, NuoDB was not designed with a specific operating system, network backplane, or virtualization model in mind. It is a general piece of software that draws on the resources available. This makes the development and operational models simpler, but also means that the underlying architecture must stay ahead of potential failures or resource limitations.

This paper covers the NuoDB architecture, how it was designed to solve these classes of problems, and what new solutions it brings to bear on old challenges. It also highlights the key concepts and architectural differences that set NuoDB apart from traditional relational databases and even other distributed SQL databases, the motivation for those design decisions, and the resulting deployment and management model. It concludes with a discussion about why the architecture positions NuoDB to be a true, general-purpose distributed SQL database uniquely designed to service both on-premises SQL-based applications as well as hybrid, private, and public cloud-based application environments deployed on physical hosts, virtual machines, or containers.

---

1. Aslett, M., "How well will the database incumbents respond to NoSQL and NewSQL?", 451 Analyst Report, April 2011.
2. Shute, J et al, "F1: A Distributed SQL Database that Scales", VLDB’13, August 2013
The NuoDB Architecture

This section focuses on the core architecture supporting a single, active database. It covers the communications, consistency, and durability models. The sections that follow build on this architecture to describe client access and automation.

Two Tiers

NuoDB is a distributed architecture split into two layers: a transactional tier and a storage tier. It also has an administration component. This section focuses on the transactional and storage management tiers that support database activity and the motivation for this design.

Splitting the transactional and storage management tiers is key to making a relational system scale. Traditionally, an SQL database is designed to synchronize an on-disk representation of data with an in-memory structure (often based on a B-tree data-structure). This tight coupling of processing and storage management results in a process that is hard to scale out. Separating these roles allows for an architecture that can scale out without being as sensitive to disk throughput (as seen in shared-disk architectures) or requiring explicit sharding (as seen in shared-nothing architectures).

In NuoDB, durability is separated from transactional processing. These tiers are scaled separately and handle failure independently. Because of this, transactional throughput can be increased with no impact on where or how data is being stored. Similarly, data can be stored in multiple locations with no effect on the application model. Not only is this key to making a database scale, it enables NuoDB to scale on-demand and implement powerful automation models.

![Figure 1: The architecture is made up of two independent tiers.](image-url)
The NuoDB transactional layer is responsible for maintaining the Atomicity, Consistency, and Isolation portions of the ACID database transaction properties. It has no visibility into how data is made durable. It is a purely in-memory tier, maintaining a high-performance shared data cache that optimizes application data access. The transactional tier is an always-active, always-consistent, on-demand cache. The storage management tier ensures Durability. It is responsible for making data durable on commit and providing access to data when there’s an in-memory cache miss in the transactional tier. It does this through a set of peer-to-peer coordination messages.

Peer-to-Peer Caching, Coordination, and Scale Out

The two tiers discussed above consist of processes running across an arbitrary number of hosts or containers. NuoDB defines these tiers by running a single executable in one of two modes: as a Transaction Engine (TE) or as a Storage Manager (SM). All processes are peers, with no single coordinator or point of failure and with no special configuration required at any of the hosts or containers. Because there is only one executable, all peers know how to coordinate even when playing separate roles. By default, all peers are mutually authenticated using the TLS security protocol and communicate over industry standard AES-256 encrypted channels.

TEs accept SQL client connections, parsing and running SQL queries against cached data. All processes (TEs and SMs) communicate with each other over a simple peer-to-peer coordination protocol. When a TE takes a miss on its local cache, it can get the data it needs from any of its peers (either from another TE that has the data in-cache or from an SM that has access to the durable store). By regularly running a simple cost function, the TE knows which peers are most responsive and therefore how to populate its cache the fastest.

This simple, flexible model makes bootstrapping, on demand scale out, and live migration very easy. Starting and then scaling a database is simply a matter of choosing how many processes to run, where, and in which roles. The minimum ACID NuoDB database consists of two processes, one TE and one SM, running on the same host.

Starting with this minimal database, running a second TE on a second host doubles transactional throughput and provides transactional redundancy in the event of failure. When the new TE starts up, it mutually authenticates with the running processes, populates a few root objects in its cache, and then is available to take on transactional load. The whole process takes less than 100ms on typical systems. The two TEs have the same capabilities and are both active participants in the database.

Similarly, maintaining multiple, independent, durable copies of a database is done by starting more than one SM. A new SM can be started at any point, and automatically synchronizes with the running database before taking on an active role. Once synchronized, the new SM provides data redundancy by maintaining an additional active, consistent archive of the complete database.
In this manner, NuoDB supports on-demand scale-out and migration. The example above started with a TE and SM running on the same host; to establish redundancy, you can separate these processes by starting a new TE on a new host. Once both TEs are running, you can shut down the original TE process. This demonstrates NuoDB’s support for live migration with no loss of availability.

These simple steps demonstrate how NuoDB supports on-demand scale out efficiently. The lightweight, process-based, peer-to-peer, and on-demand caching models enable this flexibility and control. The other essential piece that makes this model scale is how the data is cached and shared within the database processes.

**Atoms: Internal Object Structure**

The front end of the transactional tier accepts SQL requests. Beneath that layer, all data is stored in and managed through objects called Atoms. Atoms are self-coordinating objects that represent specific types of information, such as data, indexes, or schemas. All data associated with a database, including the internal metadata, is represented through an Atom.

The rules of Atomicity, Consistency, and Isolation are applied to Atom interaction with no specific knowledge that the Atom contains SQL structure. The front end of a TE is responsible for mapping SQL content to the associated Atoms, and likewise part of the optimizer’s responsibility is to understand this mapping and which Atoms are most immediately available.

**Figure 2: Running across four hosts provides a fully redundant deployment that can survive any host failing.**

**Figure 3: A Transaction Engine has a client-facing layer that accepts SQL requests and internnally drives transactions and communicates with its peers in a language-neutral form.**
Atoms are chunks of the database that can range in size, unlike pages or other traditional on-disk structures that are fixed size. Atoms are also self-replicating, ensuring that an Atom's state is consistent across TEs and SMs. The size of an Atom helps to maximize efficiency of communication, minimize the number of objects tracked in-cache, and simplify the complexity of tracking changes.

In addition to database content, there are Catalog Atoms, which are used to resolve other Atoms across the running processes. This provides a distributed and self-bootstrapping lookup service and ensures that it is efficient and always consistent.

When a TE first starts, it needs to populate exactly one object in its cache: the root Catalog Atom, which is named the Master Catalog. From this Atom all other elements of the database can be discovered. This is how a TE starts participating quickly, and from this structure a TE knows whether a needed object is available in the cache of another TE or whether it has to be requested from a Storage Manager's durable state.

This bootstrapping is part of why NuoDB uses an on-demand cache. Only required objects are pulled into a cache, so startup is fast, but so too is object resolution. Once an object is no longer needed, it can be dropped from the cache, and the catalog is updated accordingly. A TE can request an object it needs from another TE cache any time. If a TE doesn't have a given Atom in its cache, it doesn't participate in cache update protocols for that Atom.

Structuring data as Atoms also ensures consistency of the database as a whole. Because metadata and catalog data are both stored in the same Atom structure as database data, all changes are happening in the context of an atomic transaction, and are treated equally. There is no risk of updating one class of data while failing to update the other.

**Multi-Version Concurrency Control**

Central to providing ACID semantics is having a clear consistency model. Part of the challenge in scaling a transactional system is providing strong consistency while mediating conflict. Traditional approaches, including deadlock detection or explicit lock management, become very expensive in terms of time when scaled beyond a few hosts, and without a synchronized clock, order isn't meaningful. To address all of these issues, NuoDB uses Multiversion Concurrency Control, (MVCC) to handle conflict and provide a clear model for consistency.

MVCC works by treating all data as versioned, and all updates or deletes as operations that are simply creating a new version of the data. TEs are caches and those caches hold multiple versions of any given object: the canonical version and any number of pending or historical versions that may need to be available to current transactions. A version is pending until the associated transaction commits successfully.

A side effect of being able to hold separate versions in-cache is that nothing is ever changed in place. Updates are communicated optimistically, because a rollback is as simple as dropping a pending update from the cache. In NuoDB, these messages are also sent asynchronously, allowing a transaction to proceed with the assumption that an update will succeed. Asynchrony within a transaction can mask network round-trip
time, a particularly important optimization for environments with unpredictable or high-latency networks. If a transaction gets to the point of committing and doesn’t know whether an update has been allowed (see below), then of course it must be blocked.

MVCC also defines a clear visibility model for NuoDB. While modes such as ReadCommitted are supported, by default NuoDB runs with a Snapshot Isolation model. This provides a consistent view of the database from the moment that a transaction started. Multiple transactions may see overlapping views based on when they were started and what pending versions were known. In a distributed system with no single clock to coordinate events, using snapshot isolation guarantees a clear isolation model and visibility that matches what can actually be observed in reality. A nice benefit of this approach is that it also minimizes the number of messages required to coordinate the database.

![Diagram](image)

**Figure 4:** By default, NuoDB runs with snapshot-isolation visibility, meaning that an open transaction observes the state of the world as it existed when the transaction started.

In this mode, one transaction can read a value at the same time another transaction is updating that value and there is no conflict. What still needs to be mediated is the case of two transactions both trying to update the same value. On update or delete, NuoDB chooses a TE where the Atom resides to act as tiebreaker. This TE is called the Chairman, and for each Atom there is a known TE playing this role. Only TEs that have a given object in their cache can act as Chairman for that object, so in the case where an object is only cached in one TE, all mediation is local. When a TE shuts down, fails, or drops an object from its cache there’s a known way to pick the next Chairman with no communications overhead.

Note that versioning is done on records within Atoms, not on Atoms themselves. Atoms may contain many records and therefore may be too coarse-grained to effectively minimize conflict. The goal of picking a Chairman is to spread out the role of mediation but keep it cache-coherent.
Data Durability

Abstracting all data into Atoms is done in part to simplify the durability model. All access and caching in the architecture is on the Atom-level, and all Atoms have some unique identifiers, so Atoms are stored as key-value pairs. By design, durability can be done with any filesystem that supports a create, read, update, and delete (CRUD) interface and can hold a full archive of the database.

![Diagram](image)

*Figure 5: The SM handles caching and storage of Atoms to disk, including journal and archive management.*

In addition to tracking the canonical database state in its archive, SMs also maintain a journal of all updates. Because NuoDB uses MVCC, the journal is simply an append-only set of diffs, which in practice are quite small. Writing to and replaying from the journal is efficient.

![Diagram](image)

*Figure 6: The TE sends transactional changes to all SMs, which records updates to the journal and the archive while committing the change to disk.*

A Tunable Commit Protocol

To acknowledge successful commit to an SQL client, NuoDB must ensure that all properties of an ACID transaction have been met. In NuoDB, users can make a tradeoff between durability and performance with fast but transient storage in-memory to slower but more durable on-disk persistent storage. This is referred to as the commit protocol. The default is “safe” commit, which requires that all durable transactions are committed to all SMs.
All transactional changes are sent to all peers that need to know about the change. As discussed above, that means the changes are sent to any TE with the associated Atoms in-cache and all SMs. The fastest method for committing the changes is by sending reliable messages asynchronously to all interested peers.

As long as all of the SMs don’t fail simultaneously at this moment, this ensures the data will be durable. Regardless, data will always be correct and consistent. For some applications, this kind of k-safety (a measure of fault-tolerance) at the transaction tier and eventual durability at the storage tier is sufficient. Many applications, however, want to know that data has been made durable on at least one SM before acknowledging commit to the client. This is tunable by running with the Remote:N setting.

In this context, N is the number of SMs that must acknowledge data has been made durable before commit can be considered successful. For instance, Remote:2 requires acknowledgement from at least two SMs.

**The NuoDB Architecture: Examples**

This section provides a few concrete examples of the architectural scenarios discussed in the previous section. These examples use the smallest, fully redundant database configuration (two Transaction Engines and two Storage Managers). From there, it is simple to extrapolate how the interaction works on larger deployments.

**Cache Population**

The caching tier in NuoDB is an on-demand cache where each TE maintains a set of Atoms based on access patterns. There are two ways that this data may be populated in a given TE’s cache. In both cases, assume an SQL client connected to TE1.

First, as part of an SQL transaction, data may be created (for example, by performing an INSERT into a table). In the scope of the active transaction, a pending record now exists in-cache, and messages are sent to the SMs immediately. Once the transaction successfully commits, the change is now visible in the TE’s cache for any other transactions to use. This new value is also now part of the durable state of the database.

*Figure 7: In this cache population method, an initial SQL transaction triggers data creation. The data is captured in an Atom, which the TE stores in cache and sends to the SM for commit. Once committed, the new value is now part of the database and available for use in other transactions.*
The second way a cache populates is when there's a cache-miss on a required Atom. For instance, assume an Atom has been created but isn't currently in-cache at any TE. The TE uses its Catalog Atoms to discover this and then goes to any SM to fetch the Atom. This has the effect of both populating it in the TE's cache and updating the Catalog to reflect this change. Note that because this is the only TE with this Atom in its cache, the TE also becomes the Chairman for its data.

Now suppose that a client on TE2 starts a transaction that needs the same Atom. The Catalog shows that the required Atom is available from TE1 in addition to any SM. TE2 may choose where to get the Atom, but in practice it is likely to fetch it from TE1 (its next nearest neighbor), because this is much faster than retrieving it from the durable store.

![Figure 8: When the client experiences a cache miss, the TE seeks to quickly retrieve the data from another peer’s cache. Commonly this comes from another TE that has cached the data rather than from an SM.](image)

The Atom is now in-cache at both TE1 and TE2, and the Catalog reflects this fact. In this way caches are built up based on where data is created or accessed. Either TE is free to drop an Atom at any time, as long as no active transaction requires it. If an Atom is no longer in-cache at any TE, it is always available from an SM. Note that because each Data Atom is likely to contain several rows of data, populating an Atom typically has the side effect of pre-fetching data that is also needed at the TE.

**Data Update**

In the example above, an Atom was first cached at TE1 and then replicated to TE2, and now assume some data it contains is modified (for example, by performing an UPDATE on a row that is contained within a Data Atom). This requires (at least) two messages: permission from the Chairman, and pending updates to any peer with a copy of this Atom. The first message is to verify that the update may proceed. In this example, if the update is happening on TE1, where the Chairman is located, then the check is handled locally with no communication. The second message is then sent asynchronously to both the SM and TE2 to notify them that an update is occurring.
If transactions had been running on both TEs trying to update the same data (that is, the same row in the table), a conflict occurs. In this case, the Chairman acts as a tiebreaker: whichever transaction got its update message to the Chairman first “wins.”

Commit Protocol

In the previous example, the pending update messages were sent asynchronously (over reliable channels) with no requirement that any SM acknowledge the change before reporting commit back to the SQL client. This is the default behavior of NuoDB, ensuring that all changes are always consistent, and that any change is made durable as long as at least one SM is active. In this case, the update is replicated to three hosts, all of which have to fail in order to lose the update. The tunable Commit Protocol is what provides flexibility on commit.

If the same update is run, now using the Remote:1 option, the TE waits to hear from at least one SM before acknowledging commit back to the SQL client. Running with Remote:2 requires acknowledgement from both SMs.

Figure 9: When an update is made to an Atom, the change is verified by the Chairman (TE1) and then propagated to all TE and SM peers with a copy of that Atom.

Figure 10: NuoDB has a tunable commit protocol. With the Remote:1 commit option, the TE is required to wait for confirmation of commit from at least one SM before acknowledging the commit back to the SQL client.
Management and Operations Model

Along with the two database layers is a management tier. As with other databases, the management tier is a collection of peer processes, and by default peer communications are also secured using TLS authentication and encryption. These processes are called Admin Servers, and one runs on every host where a database is active in a physical deployment model. In a containerized deployment model there is no requirement to run a separate Admin process on each node in the containerized network. Starting an Admin management server (or process) on a host or in a container is a provisioning step; it makes the Admin process available to run database processes that are visible to the management environment. This collection of provisioned hosts (or containers) is called a Management Domain.

A Domain is a management boundary. It defines the pool of resources available to run databases and the set of users with permission to manage those resources. In traditional terms, a database administrator (DBA) focuses on a given database and a system administrator works at the management domain level.

Each Admin server process is responsible for the host it runs on in a physical deployment (that is, the local host). An Admin process can start and stop any TE and SM process, monitor those processes, query, and configure the running processes and perform other various administrative tasks. An Admin process also has a global view of all Admin processes in the Domain, and therefore all processes, databases, and events that are useful from a monitoring point of view.

All connection Admin servers have the same view of the Domain and the same management capabilities. So, like a database, there is no single point of failure at the management level as long as multiple Admin processes are deployed. In a physical deployment, provisioning a new host for a Domain is done by starting a new Admin process that is peered to one of the existing Admin processes on another host.

![Diagram](image.png)

*Figure 11: An admin client sends a single management message to an Admin process to start a process on some host. Once the TE is started, management messages flow back to all Admin processes.*
When an SQL client wants to communicate with a TE it starts by connecting to an Admin process. The Admin process tells the client which TE to use and the client disconnects before connecting directly to the TE. This connection brokering is one of the key roles of an Admin process, and thereby, the Admin process is also a load-balancing point. Load-balancing policies are selectable at runtime and can be implemented to support optimizations around key factors such as resource utilization or locality.

Just as an SQL programmer addresses a NuoDB database as a single, logical entity even though it is distributed across many processes, a system administrator addresses a Domain as a single, logical point of management.

This is done through any of the Admin processes. They provide a single place to connect to a Domain, manage and monitor databases, and ensure that the system is running as expected.

Database Backup and Provisioning

Continuing with this simple administrative model, the database can be easily backed up in two different modes: online or offline.

In online mode, you can execute a simple administrative command to make a copy of an SM’s journal and archive directories with minimal performance impact. The copies of the database journal and archive directories will be fully consistent versions of the database.

In offline mode, you can run a redundant SM as part of the database, or on demand when you want to perform a backup. To perform a full backup, first issue a clean shutdown of that SM so that the underlying archive can be copied. When the copy is done, restart the SM, which automatically synchronizes with the running database and then continues active participation. Since there are other SMs able to service the database, the database itself never goes down.

![Figure 12: After issuing a stable shutdown command to any SM, the archive is usable as a backup or as a way to start an independent database provisioned with the archive content. A copy of the archive is set aside to be used as a starting point for a new database or to restore that specific database version.](image-url)
This model for backup works in part because an SM can be started against any arbitrary archive, so any copy of an archive allows you to restore the database to the version represented by that archive. This same model makes database provisioning simple. Start a single TE and SM and populate the database with the content needed for all other databases. The resulting archive can be used as the starting-point for any other databases simply by copying it to a new location for use by a new database.

Benefits of the Architecture

The unique nature of NuoDB’s architecture addresses a number of typically challenging and mutually exclusive problems. This section covers a few of these problems and highlights specific aspects of the architecture critical to addressing them.

Single, Logical Database

NuoDB is a distributed database, composed of an arbitrary process deployment across an arbitrary set of hosts. Programming models such as Java Database Connectivity (JDBC), however, are designed to access a single database. One explicit challenge introduced by using shards or an active-passive scale-out model is that burden is put on the application to understand and build against that deployment.

As has been suggested throughout this paper, one of the key benefits offered by NuoDB is the view of a single, logical database. The deployment model can change to support scaling, provisioning, and availability needs without any impact on the application; an SQL client addresses what looks like a single database. Likewise, managing any database is also simplified by this logical view. This is a key building block for many of the other benefits offered by the architecture.

Multi-Data Center Support

NuoDB provides the model of a single, logical database that is always active and consistent, both within a single data center and across multiple data centers. Common reasons for running a distributed database are to achieve higher availability and fault-tolerance or to support applications and users that are distributed themselves.

Figure 13: Distributed databases can run with full access to the database in all regions, but clients talk with local TEs, and commit can be set as synchronous only locally to minimize latency.
In-memory Atoms in NuoDB TEs are partially replicated so only the objects needed are held in transactional caches; if an object isn’t in-cache then a TE won’t participate in any coordination messages. This means that if data has reasonable locality relative to a physical region, most caching and coordination will also be local to that region. When some object is needed in another region, however, it’s always available and always consistent.

Update messages are sent asynchronously from TEs to SMs. It is up to the commit protocol to determine whether a response is needed to acknowledge commit to an SQL client. As long as the system continues to run, however, all SMs will make a given change durable, and if an SM fails for any reason, it automatically re-synchronizes on restart. Configuring a durable database that scales across distributed regions, therefore, is supported by running with a commit protocol that only requires acknowledgement from a local SM.

**Flexible Schemas**

NuoDB is a relational database, which means that developers give structure to their data by defining schemas. This is a useful way to think about data, but often developers want to evolve schemas, either during development or after a database has been deployed. For instance, new fields need to be added to a table or existing fields need to be removed, renamed, or retyped.

Making these kinds of changes in relational databases is often expensive in terms of time. Sometimes these changes require downtime, because all data in a table must be traversed to apply changes or to check that constraints are still being met correctly. This has led developers to adopt a schema-free model in which data is stored without structure, so the burden is on the application to either enforce some known structure or interpret stored data at runtime and resolve incompatibilities then.

Within NuoDB, all data is stored in Atoms that are SQL-agnostic. Applying the rules of a schema is done at the SQL layer as Atoms are read or written, using the applicable schema Atom(s). Because of this, operations such as adding, renaming, or removing a column or dropping a table are done in constant time.

**Operational and Analytic Mixed Workloads (HTAP)**

NuoDB is a transactional system well suited to deliver high transactional throughput that Online Transaction Processing (OLTP) databases of record demand. At the same time, NuoDB is also uniquely suited for today’s mixed workload environments. A perfect example of this is the area of hybrid transaction/analytical processing (HTAP) — the ability to perform both online transaction processing and real-time operational intelligence and decision making processing simultaneously within the same database.

The operational model of these systems is typical of scale-out web applications, which need a database that can support many concurrent clients doing regular, small, localized updates. While techniques such as sharding or replication are hard to apply to OLTP applications, they can be used for operational workloads that have strong locality. The problem is that these approaches make it hard to do real-time analysis of the data.
Supporting HTAP requires the database to handle transactions at in-memory speeds. Often the solution is to export data from the operational database(s) into a separate in-memory service that is used only for performing analysis on the data. NuoDB provides a scale-out architecture, supporting transactions at in-memory speeds. In this way NuoDB supports scale-out operational data deployments where real-time operational analytics need to be run on the same data set.

Because NuoDB has a flexible load-balancing policy, it’s also possible to dedicate specific TE’s to specific application tasks and roles. For instance, a single database can be scaled out across smaller systems for typical operational access patterns. One or more larger systems (with more memory and processing power) can be dedicated to running analytic transactions. The application is still viewing a single, logical database that is always consistent across all the hosts but with appropriate resources dedicated to specific tasks.

**Multi-Tenancy and Resource Efficiency**

NuoDB has a formal management tier that supports the scale-out use cases discussed earlier. In a cloud environment, however, managing many smaller databases may be more important than scaling out a single, large database: for instance, hosting sites for Software as a Service (SaaS). Often these are supported by running one database or a small number of large databases, which provide separation through schemas or views. These or other mechanisms may require the application to understand how isolation should be enforced.

The management tier in NuoDB that supports scale out also supports running multiple databases on a single host or across shared hosts. Because a database is simply a collection of processes, supporting a multi-tenant deployment can be done by running separate processes for separate databases on the same host. Unlike traditional approaches, these databases have process-level isolation, use different credentials to establish separate, secure channels, and store their durable archives in physically separate locations. The same management routines that support on-demand scale out make it easy to scale and re-allocate individual tenant databases as needed to manage resources more efficiently.

![Diagram](image)

*Figure 14: Three databases are hosted across five hosts. Database 1 (DB1) has Host 1 dedicated to it; DB2 is a smaller deployment with no redundancy, and DB3 shares resources.*
Part of the advantage to running a multi-tenant deployment as separate databases is better efficiency. For instance, many applications were not written to support deployment against shared databases, so this multi-tenant model enables consolidation from separate database deployments down to one or a small number of hosts. It’s also common that some databases are active while others are idle. In this case, it’s better to focus the system resources on the databases that need them.

Because database processes can be started so quickly in NuoDB, when a given database is inactive, you can shut down its processes completely and re-start them on-demand as needed. This capability is called Database Hibernation. Hibernation supports deployments in which very large numbers of databases need to be available, but only a subset of these databases are active at any moment in time. This functionality has been shown to support tens of thousands of databases under real-world load on small quantities of inexpensive hardware.

**Live Upgrade and On-Demand Migration**

NuoDB has a scale-out model that supports heterogeneous combinations of hardware and operating systems. In such a distributed environment it’s important to maintain systems and be able to upgrade regularly. Cloud environments are typically virtualized, which means that it’s also important to allow migration between containers and servers. These requirements typically conflict with the need for high uptime.

The simple process model and on-demand cache work together to make it easy to bring new processes online. As indicated earlier, this makes it simple to support upgrades with no downtime. New TEs are started on new hosts to meet capacity requirements and then existing TEs can be shut down to perform upgrades. In this way, NuoDB supports a rolling upgrade of software or hardware with no downtime or loss of availability.

Using the same model, if a new TE is left running and the existing TE is shut down permanently, the database migrates resources with no loss of availability. Because NuoDB provides monitoring data, just as a database could be hibernated when not active, it could also be migrated when different resources are needed. In NuoDB this is called Database Bursting, and it naturally complements the Database Hibernation described above. The previously cited density example used low-power systems to be as efficient as possible, but when a single database needed more capacity, it was migrated to a more capable server until activity slowed down.

**Reactive High Availability**

NuoDB provides traditional, proactive approaches to High Availability by running with additional TEs, SMs, and Admins. This model extends beyond a single data center, and supports upgrade and migration. In any deployment model, however, there’s a trade-off between the required availability and resources allocated to take over on failure. For example, some sites may want to sacrifice a small amount of availability on failure for applications that aren’t as critical or are less likely to fail. In these cases, the cost of pre-provisioning resources may outweigh the cost of potential lost availability.
Because NuoDB is dynamic and able to react to resource availability changes, it is also able to bring new resources online on-demand to take over for any that have failed. When one host fails, a new host can be started to join the running domain, and the TEs that were running on the original host can quickly be restarted on the new host. If SMs were running on the failed host, and their archives are still reachable (for instance, on a remote volume or in some network service), then those SMs can also be re-started. As long as all databases were run in a fully redundant deployment (on multiple hosts), there is no downtime for the database as a whole, only decreased capacity while the new host comes online.

Similarly, a host can be run as part of a domain for the sole purpose of being available to pick up work when another host fails. In this way, the window for reduced availability is cut to the time it takes to observe failure and restart the processes (often measured on the order of seconds or less). In a multi-tenant deployment, the cost of running this host is amortized over the total number of databases, making it much more cost-effective.

### Table Partitioning Using Storage Groups

In the default implementation, each SM represents a complete, independent copy of the database. This makes it very simple to manage redundant replicas, supporting continuously available deployments, non-intrusive backup, and low-latency access across multiple data centers. It also simplifies the operations model.

There are also several valid use cases that require each SM to contain only a subset of the total database. Some of these use cases have to do with performance, such as applications with high insert rates, or with large databases that cannot easily be contained by a single storage device or service. Others are focused on explicit choices about where to store data, for example minimizing coordination between distributed sites or defining provenance and policy requirements. In a distributed system, there are also advantages to segmenting data sets to support more graceful failure modes.

![Diagram](image.png)

*Figure 15: Storage groups allows users to control the physical location of where the data is located and how many copies of that data are stored for redundancy, continuous availability, and separate processing purposes.*
Because all SQL access to a database is through the TEs, the durable state of the database can be partitioned without any change to the programming model. In other words, data partitioning at the durability level doesn't affect the client view of a single, logical database. NuoDB administrators have the option to let a database store subsets of the total persistent database on a specific SM. This is accomplished using NuoDB Storage Groups. A storage group is a logical storage unit that is serviced by one or more SMs in the database. Tables can easily be partitioned by value or range, and those partitions are then stored in a specific Storage Group.

**Scale-out Performance**

A scale-out database needs to provide increasing throughput as hosts are added. Because NuoDB operates as an in-memory database, and because that tier of the system uses an on-demand caching scheme, any application can see this throughput improvement without the use of an additional third-party data cache products (for instance Memcached) or any changes to the application logic itself. These requirements can be shown in the context of two specific benchmarks.

The first benchmark is the Yahoo Cloud Serving Benchmark (YCSB) 10, designed to simulate modern web-facing workloads on back-end databases. It can be tuned along factors such as dataset size, read/write bias, volume of queries, and number of servers. The second benchmark is DBT-211, an open source implementation of the TPC-C benchmark. It simulates warehouse management, but generally models applications with heavily localized sets of data that are accessed both locally and globally by different types of simulated users. Together, these represent (respectively) modern and legacy real-world workloads that benefit from scale-out architectures.

In both cases, the benchmarks are run with no modifications to the code except for minor changes to support the NuoDB SQL dialect and, in the case of DBT-2, the NuoDB stored procedure language. While the benchmark tests themselves are unchanged, what is interesting is the deployment model used to run the tests.

These tests are run to show scale-out behavior. To ensure that the load driver itself doesn't become a bottleneck, it must also be scaled out to drive increasing load to the database as more TEs are added. While YCSB was designed with scale out in mind, TPC-C was not. It was, however, designed to simulate both local access and distributed access in a real-world manner. Both of these benchmarks can be used to show horizontal scale by scaling out TEs paired with the clients that drive testing load.

This is consistent with typical web application deployment models, and is one of the reasons that NuoDB’s architecture works so naturally with modern applications. In a scale-out web deployment there is often a single web container on each host paired with a local caching process or layered on top of some scale-out cache13. This helps minimize latency and centralize database management. It also means that extra coordination is needed to either keep the distributed cache consistent or shard the application logic, or to do both. As with the previous footnote, caches are typically transient and key-value oriented, which requires the application to work specifically with these constraints.
Figure 16: Depicted on the left is a common web-scale deployment pattern using an open source third-party distributed memory cache product, Memcached. On the right, the same pattern using Transaction Engines as local cache and scale-out components.

NuoDB supports a similar deployment strategy where a single TE can be run on each web host. This affinity model provides the same low-latency and in-memory performance provided by Memcached. This approach ensures that the cache is always consistent even when there is contention between multiple web containers. It puts no requirements on the application to be aware that there is any caching layer or to support any explicit sharding or hashing model. In the case of DBT-2, this means that each client (which represents a Terminal) is co-located with a TE. Because of the standard TPC-C behavior, 95% of the client’s work is local to a Warehouse, and therefore most transactions are dealing with locally cached data and with a local Chairman for that data. This is a side effect of how the test models real-world workloads, not a modification to any standard behavior.

In the case of YCSB, benchmarks can be run with several different access patterns. For instance, with uniform access of all data across all hosts, as more clients are added NuoDB shows scale-out behavior even with a small number of TEs on separate hosts. To model a more realistic workload, where a subset of the database is more active followed by a trail-off pattern, TEs are co-located with the YCSB client drivers and paired using simple affinity. This deployment shows near-linear scale and low-latency as hosts are added during the test.

Conclusion

As software development organizations are moving to a cloud-deployment model, a new database architecture is essential. This paper covered the key architectural elements of NuoDB – the distributed SQL database – and showed how that architecture uniquely combines the transactional consistency and durability that databases of record demand, with the scale-out simplicity, elasticity, and continuous availability that cloud applications require. It also showed how that architecture and its simple, peer-to-peer model solves many modern deployment challenges.

Learn More

+ Visit our website: www.nuodb.com
+ Email us: sales@nuodb.com
+ Follow us on Twitter: @NuoDB
+ Watch NuoDB in Action: www.nuodb.com/full-demo
+ Try out NuoDB: www.nuodb.com/download

TRY OUT NuoDB

By now you’ve heard plenty about what we think we can do. Now it’s time to find out for yourself. nuodb.com/download

NuoDB is a distributed SQL database company that helps enterprise organizations solve the highly complex database challenges they face when trying to move enterprise-grade workloads to the cloud.

For technical organizations adopting a distributed, cloud-first strategy through hybrid-cloud, container-native, microservices and other modern architectures, NuoDB is the only solution offering flexibility without sacrificing performance, SQL dependency, and availability. NuoDB ensures that mission critical, SQL dependent applications are capable of maximizing performance by providing a distributed, highly available, container-native solution.

NuoDB is platform agnostic, capable of running in public clouds, private clouds, and on-premise. Through a robust, stand-alone platform or through integrations with Red Hat and Temenos, NuoDB help digitally advanced enterprise organizations achieve a data management architecture that extends the competitive advantages gained by cloud adoption.

PHONE: +1 (857) 999-0066 EMAIL: sales@nuodb.com WEB: nuodb.com TWEET: @NuoDB