Whitepaper

Continuous Availability in NuoDB

Core Tech

Introduction

This paper introduces the key concepts and challenges in maintaining availability. There are technical and business decisions that affect the lifecycle of a system, both of which significantly impact overall costs. We'll explain core aspects of the NuoDB architecture and how they contribute to an overall solution to these challenges. Finally, we'll discuss future directions.

You should read this paper if you are looking for a better understanding of the challenges in modern system design, if you are weighing the features and costs of different applications, or if you are trying to get a deeper understanding of how and where NuoDB's architecture is significantly different than other data management systems.

Understanding Continuous Availability

In traditional client-server architectures, there is a single physical place to access a service. That's usually a server running some known software with attached disks that is available over a network. This view of the world has led to qualifying availability in "nines", or collectively how long the combination of hardware and software is likely to run before some piece fails and takes down the service as a whole. Individual component failure is inevitable, so the only question is how much downtime you can expect or tolerate in a given year.

Inevitability of failure has led to common techniques designed to increase uptime and recover from complete failures. Servers are often replicated and run in multiple locations either to handle independent tasks or provide a hot-standby model. The former means a failure takes down a subset of the service's functionality while the latter exposes fail-over delays. Additionally, application data is often replicated, resulting in latencies, windows of loss on server failure or both. Each of these approaches still assumes that a system should be measured against an expected amount of overall downtime. Typical High Availability models require at last "five nines" (99.999% uptime annually) and some form of data replication.

Today's modern applications, however, demand a different level of service – availability isn't a "nice to have", it's a "need to have." Especially in the context of the database, services must be designed with an always-on mentality. This idea of addressing availability from a new perspective is a critical driver for the adoption of distributed architectures to replace client-server models. When a system is composed of independent pieces running in physically separate locations, it's possible to sustain failures but keep the service as a whole running. This is what Continuous Availability is all about: not a measurement against expected downtime but a statement about overall operational availability.

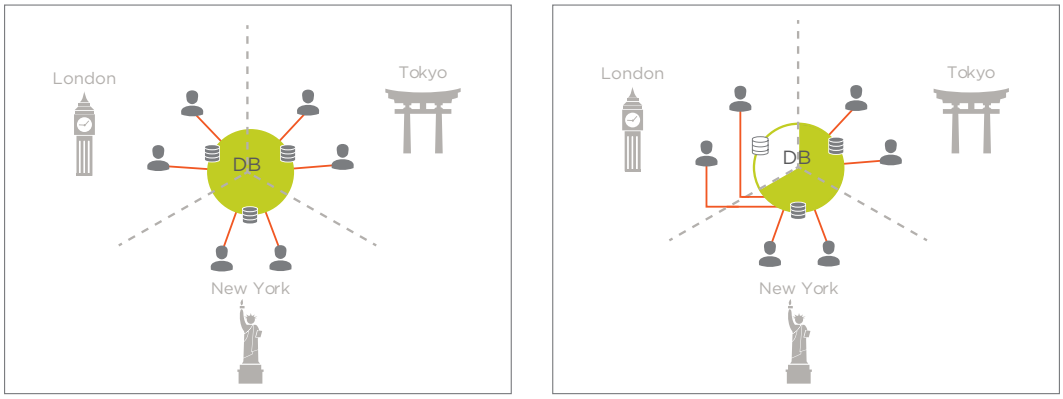

Figure 1. A globally distributed database is running across three data centers. As the London data center fails due to a power outage, incoming database requests need to be rerouted. Ideally, these requests would be redirected to the most responsive database components (in this case, the nearest data center in New York), and users would naturally experience a degradation in database performance until London comes back online.

Adversaries of Availability

There are technical and business reasons why availability is challenging. From a technical point of view, the reality is that all hardware and software will fail

eventually. Disk drives, for instance, come with clear Mean Time Between Failure numbers, a measure of how long you can expect to use the disk before it will fail. Modern data centers are designed with redundant power sources, cooling systems and network interfaces but even public cloud operators like Amazon and Google have experienced complete site failures.

From a business point of view, availability is about risk versus reward. It's possible to build complex systems out of hardened components, but doing so is both very expensive and time-consuming. Even then, traditional architectures were not designed to handle complete failures gracefully and can leave data in unknown states. Likewise, you can build fail-over capabilities to minimize the length of outages, but that requires spending twice as much for resources that are rarely used. In client-server models, therefore, people tend to assume that some downtime is inevitable and try to find the most cost-effective way to address their specific concerns.

In addition to anticipated but unpredictable failures, there are the normal disruptions that any system can expect. Software needs to be updated periodically and hardware needs to be replaced or upgraded. Network maintenance may be unavoidable and disruptive. Physical data center layout may need to be changed or new sources of power introduced.

Ultimately, individual failures are inevitable. Typically these failures in data management systems not only bring down services but result in lost or unstable data. Downtime may be acceptable but increasingly data loss is not.

Resilience & Operational Intelligence

One approach to higher availability and clean failure-handling is to shift from designing around passive redundancy to an active model. If individual components are always going to fail, rather than "failing over" to redundant components, you can instead run multiple active components and use that redundancy to ensure that the system is resilient even as components fail. This reacts to failure faster, is more cost-effective and yields higher capacity. Great examples of this design model are the redundant, active systems used in airplanes or the resiliency model employed by telcos.

Increasingly, systems are collecting and using operational data to gain insight into runtime optimization. That same operational intelligence can be applied to resilient design. A good example are the services built by Netflix to stay ahead of public cloud failures. Watching for patterns and leading indicators of possible failures or resource scarcity lets systems react before the failure or starvation occurs, leading to better overall availability. Traditional architectures, however, are not well-suited to this model.

These are the kinds of design options that modern, distributed systems can bring to bear. In order to support a resilient, resource-effective model, however, the system needs to be architected around more than just redundancy. To support continuous availability, a system needs to be designed with these operational requirements in mind. The next section introduces the core elements of the NuoDB architecture and explains how it was designed to address exactly these requirements.

NuoDB: Architected for Availability

NuoDB was architected from the start with these issues in mind. The goal is a system which can run uninterrupted for years, supporting infrastructure upgrade and application evolution and without the need to pre-provision large amounts of capacity for fail-over. Designing a system in this fashion is complicated, and the details are often subtle. What follows highlights several of the key architectural elements that collectively represent a design for continuous availability.

Independent Peers

NuoDB implements a peer-to-peer model. An active database consists of many peer processes that are independent and able to perform the same tasks. Because of this equality, any peer can fail or be shut down at any point without losing the database as a whole. There is no master, coordinator or other single point for failure. This simple model is the foundation for availability and resiliency in the NuoDB architecture.

The one difference between peer processes is the role they are asked to play. A process may be started as a transient, in-memory member of the database. Called a Transaction Engine (TE), these processes form on-demand caches based on the transactions they run but otherwise have no affinity to specific tasks or subsets of data. The same executable may also be started to focus on storage. These Storage Managers (SMs) maintain durability and provide access to data that isn't already cached in-memory, but do not accept client connections or handle SQL. From a data availability perspective, there is no difference between these peers except that some have more of the data than others. As long as one of each peer type is running, the database is available.

The on-demand, independent nature of in-memory peers means that losing one of these peer processes is roughly equivalent to invalidating a cache. If at least one TE is running, then the database is available. Typical deployments run multiple TEs for higher availability and throughput. Starting a new Transaction Engine is measured in milliseconds, and because a TE may be started on any host without access to data on-disk, reacting to failure is simple and fast.

Independent Durability

As with the Transaction Engines, Storage Managers are independent processes. In the simplest model[1] each SM maintains its own durable copy of the entire database. If there are at least two SMs running then there are at least two independent, consistent copies of the database maintained on-disk.

Running with multiple points of durability obviously provides redundancy guarantees. Losing a disk does not affect the database as a whole. If a server goes offline temporarily, when it comes back online, the SM automatically re-synchronizes with the active database. Given the peer-to-peer nature of the system, this synchronization effort is supported by all peers that can share updated state. This is a sharp contrast to traditional systems that would simply

1 As of NuoDB Larks Release 2.2, all SMs archive the entire database. Future releases will introduce a new capability to partition storage, so that each SM may keep some or all of the database. See the section on Future Directions for more detail.

replay from one storage point to another. The synchronization effort results in a known, consistent state that the restored SM has to reach before it can return to active participation.

This same process can be applied to create new points of durability. When an SM is started with state on disk, it will run the same process, synchronizing with the active database and effectively creating a new, independent point of durability. Operationally, it may be faster to create new points of durability by taking a snapshot of an existing copy and using that as a starting-point for a new SM. Either way, the storage tier provides the same properties as the (in-memory) transaction tier: redundancy, availability when any given peer fails, and the ability to bring new peers online on demand.

A simple configuration, called the commit protocol, is used to decide which points of durability are synchronous or asynchronous within the scope of an application's transactions. This lets users define which points of durability must have written changes in the case of failures, and therefore how concerns of latency versus safety are traded. For instance, one application may need to know that every update is durable in four places before acknowledging commit to a client while another is more concerned about lower latencies than catastrophic failure. This simple model supports a user's specific availability requirements and provides clarity in the case of complete system failure.

Independent Orchestration

Coordinating database peer discovery and liveness is an orchestration layer. Composed of a separate peer-to-peer network of Agents running on each host, this layer supports provisioning, monitoring, and management. As with a database, this peer network is a collection of independent processes, each of which is told to play one of two roles. All Agents are responsible for their local hosts, but a subset will be told to act as Connection Brokers and maintain the global state of the deployment.

The Brokers use this global view of the system to route management messages, collect statistics and introduce new database processes to running peers. They also support simple load-balancing policies. As long as at least one Broker is active, databases can be discovered and managed. There are no special requirements on the host or need for existing data on-disk to start a new Broker, so they can be started on-demand in reaction to failures. Because Brokers are independent peer processes, each with the same knowledge and capable of performing the same tasks, typical deployments run with several Brokers for redundancy.

The set of hosts working together through this peer-to-peer network is called a Domain. The focus of the Domain is on resource management and systems administration, not SQL. This separation makes for a lightweight tier that is easy to expand or migrate, uses minimal resources and is highly resilient. New hosts can be provisioned on demand separately from changing running database topologies. The Domain abstraction is important to database availability for a number of reasons.

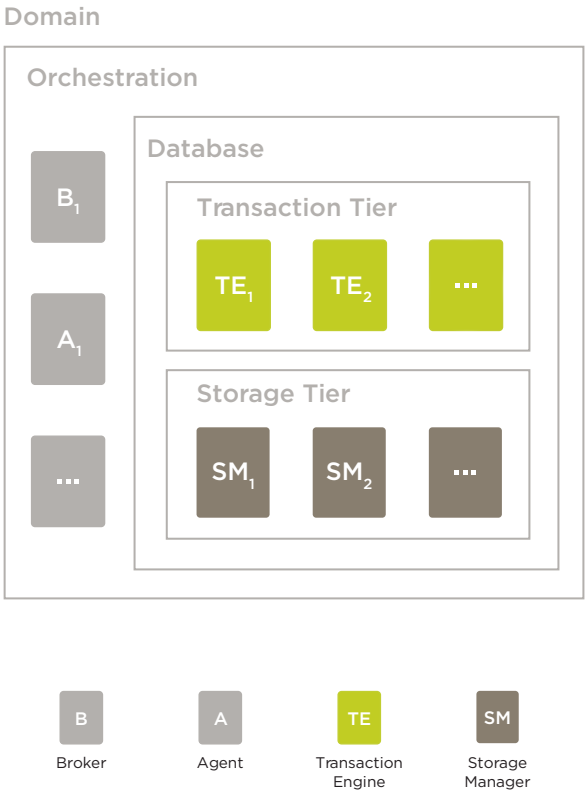

Figure 2. A database designed for continuous availability depends on a peer-to-peer architecture that treats durability, transaction handling, and orchestration as related but distinct processes. Here you can see that the database consists of the storage tiers, managed by a layer of orchestration. A domain encapsulates all the resources available to the database.

Single, Logical System

One key way that the Domain abstraction supports higher availability is by providing the view of a single, logical system. When a SQL client connects to a database, it uses a single connection string as you would with any single-server relational database. The connection string is a standard format that specifies the server and database name. In this case, however, the server is a Broker which will use its load-balancing policy to tell the client which TE it should connect to.

If the client loses its connection, it runs the same process and re-connects to the database. The fact that it may have been connected to a different TE is transparent because all TEs are equivalent. So when a TE's host fails, or is simply turned off, the client continues to operate. Standard connection pools already handle attempts at reconnection, so applications run uninterrupted when hosts fail. Likewise, connection pools will periodically drop connections and re-establish them, so if new TEs are added to a database they will be used without the client having any knowledge of the change.

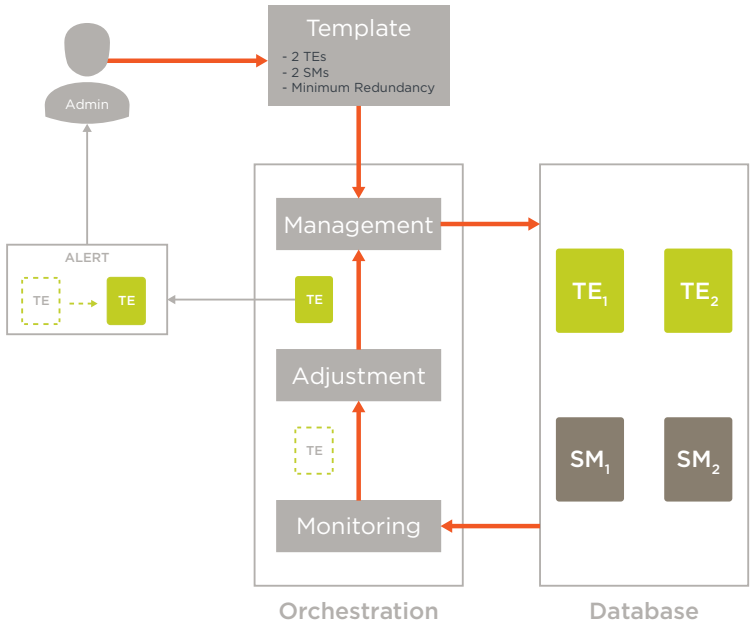

Figure 3. An administrator selects a database template outlining a minimally redundant system of two Transaction Engines (TEs) and two Storage Managers (SMs). NuoDB's orchestration capabilities start a database meeting those requirements (two TEs and two SMs). NuoDB continues to monitor the created database to ensure it meets the service level agreement (SLA). If a TE becomes unavailable, NuoDB automatically adjusts to enforce the SLA by automatically starting up a new TE. The administrator is alerted to the activity but does not need to take action.

By using DNS rules, load-balancers, or other standard tools, a single name can be given to the collection of Brokers. Doing this, an application continues normally regardless of the hosts that fail or are added to the Domain. Likewise, the tools provided to monitor and manage a Domain see a logical view by connecting to any single Broker, so simple indirection provides for a continuously available point of management.

Online Upgrade & Migration

This view of a logical database provides a high degree of availability where clients continue to operate even as resources fail unexpectedly. This model means that expected failures are also transparent to clients. One common example of this is upgrade.

Because all Domain and Database components are redundant, and because losing any single entity is transparent to an operating client, an administrator can run a rolling upgrade with no downtime. In this case, "upgrade" could mean changes to the hardware, operating system, local software, or even the NuoDB installation. It could also mean migrating to new networks or new hardware. By shutting down one service or host at a time, running the update, and then restoring that service or host, you can run a complete upgrade with no loss of availability. Because new TEs and SMs can be added at any time to a running database, an administrator can choose to pre-provision additional capacity before rolling an upgrade so that there's also no loss of capacity.

NuoDB is designed to run in a mixed-version deployment and support version-forward compatibility. For instance, if a Domain is set up using NuoDB Swifts

Release 2.1, a rolling upgrade can take that deployment to NuoDB Larks Release 2.2 as described in the previous paragraph. The upgraded peers will wait to start using new features of the protocol until all peers have been upgraded. During the process, if there is any reason the upgrade cannot be completed, the new peers can be rolled back to the current version. Only when all peers are running the same version is the durable state updated, at which point the database is now running the new version of NuoDB.

Infrastructure and Representation Agnostic

The ability to react to failure or run live upgrades is sometimes made more difficult by requiring a homogeneous environment. NuoDB Domains support running on mixed operating systems and hardware. In addition to lowering costs, this often makes it much easier to react to failure by bringing new resources into the Domain to compensate.

Internally, NuoDB is also independent in its representation of data. While the front-end looks like a standard SQL database, the peer communication is built around objects that understand what role they play and where they are replicated within the system. This results in a simpler communications model and means that durability is focused on storing named objects, not the traditional blocks or pages that are tightly coupled with SQL structure. From an availability point of view, this has two significant benefits.

The first benefit is that SMs are storing to a key-value store. That can be a local file system, Amazon's S3, HDFS or any number of other services that themselves provide higher availability and easier failover models than high-end disk arrays. The second benefit is that a SQL schema is just a mapping from object representation to application structure, not something rigid in the on-disk layout. When an application needs to change its data structure, it can do it in constant-time without having to take down the database or incur disk churn.

Service Level Agreements

The abstraction provided by the Domain has another important advantage for availability. By formalizing the provisioning model, collecting global statistics and exposing a single point for managing database peer processes, the Domain is a building block for automating database management. NuoDB exposes Templates, which are a formal way of defining Service Level Agreements. A user may run with predefined Templates or write their own. When a database is instantiated against a Template, the Domain takes care of starting appropriate processes, monitoring the state of the system, reacting to failure, and alerting the user if the system is an unfulfillable state.

This model guarantees minimum availability while letting the Domain decide how best to use its resources. In some cases, that may mean proactive changes to get ahead of likely failures. In other cases, such as multi-tenant deployments, it can mean completely shutting down databases that are not currently active. Called Hibernation, peer processes can always be re-started on-demand as long as they fit within the specified SLA. In this model, the database is still available even though it's not using any resources.

Global Operations

A critical element for true availability is sustaining complete failure to networks or data centers. Everything that has been described to this point works across physically distributed sites. A Domain or Database provides all of the same active, logical properties running in multiple locations. A complete data center can fail without any loss of data or global service availability. Obviously there is loss of capacity, but the architecture is designed to react to that efficiently. When the site is restored, the database will expand to pick up operations and the storage points will re-synchronize automatically.

Because of the heterogeneous nature of NuoDB operations, this global model extends to hybrid deployment. In one model, this may mean running across different public or private clouds to survive failure of any one service provider. In another model, it means running in one cloud but always keeping data stored in another for disaster cases. In either case, the service and its data are available in multiple locations and highly resilient to catastrophic failure.

NuoDB: Future Directions

Just as resilient systems are designed to evolve over time as needs change, the architecture described in this paper represents the building blocks for supporting increasingly distributed deployments. Without changing how the system as a whole works, new capabilities are being introduced to the existing model. This section describes three key areas of active work, novel in the data management space, that push on the limits of scale by simplifying management and improving on failure models.

The first piece, alluded to earlier, is support for storage-level partitioning. Moving from a model where each SM has a complete copy of the database to one where each SM may have a subset of the objects is a natural extension of the architecture. The peer protocols were originally designed around knowledge of where to find any given object, which is part of what gives the processes independence. Using this independence, the database maintains the view of a single, logical system, but allows operators to decide where subsets of data are stored or replicated. This in turn makes it easier to reason about the state of the system in the face of complicated failures and plan ahead for specific kinds of failure (like WAN outages) by placing copies of data in known locations.

The second piece has to do with what happens in global systems when networks fail. When a system is running in multiple locations, and the networks between some or all of those locations fail, the logical nature of the system is partitioned. The traditional choices at this point are either to shut down a subset of the system to guarantee global consistency or sacrifice consistency entirely to maintain availability in all locations. A third option is to use features like storage partitioning and commit protocol so that operations can proceed in a reasonably uninterrupted fashion. This works extremely well for most distributed applications that are designed around largely localized activity. To make this work in the general case, an extension to the architecture introduces an abstraction for provisional transactions that can be reconciled at the transaction level when networks are repaired. With this extension, a NuoDB database continues to provide global availability in the face of substantial failures.

The third piece makes it easier to work with these first two pieces through SLAs that formalize acceptable failure models. In most database systems, service levels are focused on transaction rates or average/worst-case latencies. Availability guarantees, however, are often as or more important. In that spirit, Template capabilities are evolving to capture operational requirements that survive specific types of failures. Rather than specifying where data must be stored, how the commit protocol is defined or what policy is used on network failures, an SLA captures the operational need to meet certain availability and redundancy. In the case that some failure leaves the Domain underprovisioned and vulnerable to failures, users are immediately notified and pointed to the problem.

Conclusion

This paper started by making the case for what continuous availability means and why it is such a challenging goal to achieve. What followed was a discussion of building-blocks that make up the NuoDB architecture. One on top of another, they illustrate a comprehensive design for making a database available. From independent peers and points of durability to a single logical view that can sustain failures, from online upgrade and evolution to service levels that automate availability within or across geographies, the result is a system designed for continuous availability. No matter the failures, intended or unexpected, clients should be able to access their data with no visible interruption.

In practice, few deployed services will experience one hundred percent availability over their lifetime. Unexpected failures, natural disasters and human error tend to get in the way. With NuoDB, however, the goal was to design an architecture that could supply zero global down-time. The result is a system built in layers that make it simple for clients to access a database under normal operations and continue to operate in the same fashion even as servers fail or go through normal evolution.

For More Information

To see a demonstration of how NuoDB maintains continuous availability, visit **www.nuodb.com/continuous-availability**

To see a video explaining NuoDB's distributed database architecture, visit **www.nuodb.com/nuodb-architecture-video**

To try NuoDB out for yourself, visit **www.nuodb.com/download**

About NuoDB

NuoDB was launched in 2010 by industry-renowned database architect Jim Starkey and accomplished software CEO Barry Morris to deliver a scale-out SQL database management system designed for the modern datacenter.

Used by more than 18,000 developers worldwide, NuoDB's customers include automotive after-market giant AutoZone, Europe's second largest ISV, Dassault Systèmes, Palo Alto Networks, Kodiak Networks, and many other innovative organizations. NuoDB is cited positively for the second consecutive year on the industry's leading operational database management systems analysis.

NuoDB is headquartered in Cambridge, MA, USA. For further information visit: **www.nuodb.com**.

215 First Street
Cambridge, MA 02142
+1 (617) 500-0001
www.nuodb.com

CA WP 03151