Technical Whitepaper

Core Tech

The Architecture & Motivation For NuoDB

NUODB®

The Architecture & Motivation For NuoDB

This whitepaper provides an introduction to the NuoDB architecture. It covers the internals of the database, the management model and the key differentiators of the technology. It does not detail all of the protocols or implementation specifics. This paper is designed to provide familiarity with both what the NuoDB architecture looks like and what the motivation was for that design.

Table of Contents

I. Introduction & Background 1

II. The NuoDB Architecture 2
- i. Three Tiers 2
- ii. Peer-to-Peer Caching, Coordination & Scale-Out 3
- iii. Atoms: Internal Object Structure 4
- iv. Multi-Version Concurrency Control 6
- v. Data Durability 7
- vi. A Tunable Commit Protocol 8

III. The NuoDB Architecture: Examples 9
- i. Cache Population 9
- ii. Data Update 10
- iii. Commit Protocol 11

IV. Management and Operations Model 12
- i. Logical Administration and Templates 13
- ii. Database Backup and Provisioning 13

V. Benefits of the Architecture 14
- i. Single, Logical Database 14
- ii. Geographical Distribution 14
- iii. Flexible Schemas 15
- iv. Operational and Analytic Mixed Workloads 16
- v. Multi-Tenancy and Resource Efficiency 16
- vi. Live Upgrade and On-Demand Migration 17
- vii. Reactive HA 18
- viii. Scale-Out Performance 18

VI. Future Directions 21
- i. Partitioned Storage 21
- ii. Distributed Queries 21
- iii. Automation 22
- iv. Commit Protocol and Consistency 22

VII. Conclusion 23

VIII. About NuoDB 24

Introduction & 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 non-SQL programming model or relying on sharding and explicit replication. There were no solutions that provided complete ACID semantics. This tension is what inspired the NewSQL movement[1].

In the NewSQL model and in modern distributed datacenters, on-demand scale-out databases that maintain ACID semantics are an architectural requirement[2]. Also critical are key features associated with being cloud-scale like 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 can provide low latency. These are the design goals that defined the NuoDB architecture.

NuoDB is a distributed database designed with cloud-scale challenges in mind. It's a true SQL service: all the properties of ACID transactions, standard SQL language support and relational logic.

NuoDB is a distributed database designed with global application deployment challenges in mind. It's a true SQL service: all the properties of ACID transactions, standard SQL language support and relational logic. It's also designed from the start as a distributed system that scales the way a cloud service has to scale providing high availability and resiliency with no single points of failure. Different from traditional shared-disk or shared-nothing architectures, NuoDB presents a new kind of peer-to-peer, on-demand independence[3] that yields high availability, low-latency and a deployment model that is easy to manage.

Unlike some cloud services, 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 will exploit the resources it's given. This makes the development and operational models simpler, but also means that the underlying architecture must do more to 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

1 Aslett, M., "How 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

3 See the section on Peer-to-Peer Caching, Coordination & Scale-Out below for more on this design and how peers maintain autonomy.

it brings to bear on old challenges. It also highlights the key concepts and architectural differences that set NuoDB apart from traditional relational databases, the motivation for those design decisions and the resulting deployment and management model. It concludes with a discussion about future directions for the product, highlighting why the architecture positions NuoDB to be a true, general-purpose SQL database for future applications.

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.

Three Tiers

NuoDB is a distributed architecture split into three layers: an administrative tier, a transactional tier and a storage tier. This section focuses on the transactional and storage tiers that support database activity, and the motivation for this design.

Splitting the transactional and storage tiers is key to making a relational system scale. Traditionally[4], 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 application and its data are coupled, and therefore hard to scale out. Separating these roles allows for an architecture that can scale out without being as sensitive to disk throughput[5] (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.

4 http://en.wikipedia.org/wiki/IBM_System_R

5 See the section on Data Durability below for more on why this is true.

Figure 1: The architecture is made up of three independent tiers.

The transactional layer is responsible for maintaining Atomicity, Consistency and Isolation in running transactions. It has no visibility into how data is being made durable. It is a purely in-memory tier, so it's efficient as it has no connection to durability or consistency. The transactional tier is an always active, always consistent on-demand cache.

The storage layer ensures Durability. It's responsible for making data durable on commit, and providing access to data when there's a miss in the transactional cache. It does this through a set of peer-to-peer coordination messages.

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

The two tiers discussed above consist of processes running across an arbitrary number of hosts. NuoDB defines these tiers by running a single executable in one of two modes: as a Transaction Engine (TE) or 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. 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 SRP[6] and communicate over encrypted channels.

Transaction Engines accept SQL client connections, parsing

6 Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, September 2000.

and running SQL queries against cached data. SMs and TEs 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 another TE that has the data in-cache or 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 an independent point 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 will automatically synchronize[7] with the running database before taking on an active role. Once started the new SM will maintain an active, consistent archive of the complete database.

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

7 See the section below on Database Backup and Provisioning for how to optimize this process.

In this manner NuoDB supports on-demand scale-out[8] and migration. For instance, the above example started with a TE and SM running on the same host. Separating these processes (e.g., for redundancy) is done by starting a new TE on a new host, and then shutting down the original TE process. This demonstrates NuoDB's support for live migration with no loss of availability.

This simple set of steps demonstrates how NuoDB supports on-demand scale-out efficiently. The lightweight, process-based peer-to-peer and on-demand caching models are what enable this. The other key to making 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 supports SQL. 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 meta-data 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. One strong side-effect of this design is that replacing the SQL front-end with a different dialect or programming model would have no impact on the ability of the system to provide ACID semantics.

Figure 3: A Transaction Engine has a client-facing layer supporting SQL, but internally drives transactions and communicated with its peers in a language-neutral form.

8 As of this paper's publication, NuoDB has been tested scaling up to over 100 hosts. See the section below on Scale-out Performance for specific data about scaling a single database.

Atoms are arbitrary chunks of the database, but unlike pages or other traditional on-disk structures Atoms are active. In a way, Atoms represent the network peers, self-coordinating between cached instances and marshaling or un-marshaling at the storage layer. The size of an Atom is chosen to help 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's efficient and always consistent to resolve any given Atom in the system.

When a TE first starts, it needs to populate exactly one object in its cache: the root Catalog Atom 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 will be updated accordingly. At any time a TE can find cached copies of the data it needs, but as long as it doesn't have an Atom in its cache it doesn't need to participate in update protocols.

Structuring data as Atoms also ensures consistency of the database as a whole. Because meta-data and catalog data are both stored in the same Atom structure as database data, all changes are happening in the context of a transaction, and are treated equally. There is no chance to update 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 like deadlock detection or explicit lock management become very expensive when scaled beyond a few hosts and without a synchronized clock, order isn't meaningful. To address

all of these issues, NuoDB uses MVCC[9, 10], 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. Transaction Engines 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 visible to the current transactions. A version is pending until the associated transaction commits successfully.

"This investment demonstrates our strong interest and belief in NuoDB's strategy and technologies for next generation cloud based services. NuoDB delivers a lot of the features required to address the market needs in terms of usages in the new world of experiences."

Dominique Florack
Senior Executive Vice President, Products-R&D
Dassault Systèmes

A side effect of being able to hold separate versions in-cache is that nothing is ever changed in-place, so updates can be communicated optimistically, because a rollback is done by simply dropping a pending update from the cache. In NuoDB these messages are also sent asynchronously, allowing a transaction to proceed assuming 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 block.

MVCC also defines a clear visibility model for NuoDB. While modes like Read-Committed are supported, by default NuoDB runs with a Snapshot Isolation[11] 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.

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 Transaction Engine 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

9 Reed, D.P., "Naming and Synchronization in a Decentralized Computer System", Doctoral Dissertation, September 1978

10 Bernstein, P.A. and Goodman, N., "Concurrency Control in Distributed Database Systems", ACM Computing Surveys Volume 13 Issue 2, June 1981

11 http://en.wikipedia.org/wiki/Snapshot_isolation

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.

Figure 4: Bernstein, P.A. and Goodman, N., "Concurrency Control in Distributed Database Systems", ACM Computing Surveys Volume 13 Issue 2, June 1981

Note that versioning is done on records within Atoms, not on Atoms themselves. Atoms could contain many records and would 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 identifier, so Atoms are stored as key-value pairs. By design, durability can be done with any service that supports a CRUD[12] interface and can hold a full archive of the database[13].

12 http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

13 See the section below on Partitioned Storage for how this model will change in future versions.

Each Storage Manager addresses its own full archive of the database. Within a single database, however, different SMs can be addressing different storage services. For instance, on Amazon Web Services a database could use two SMs, one writing to an EBS volume and a second storing in S3. This kind of mixed-mode storage provides higher performance from the EBS service but higher durability from S3. As was discussed above, TEs and SMs are running a cost function to track the responsiveness of their peers. Because of this, when a TE needs some Atom from durable store they know which SMs are more responsive than others, and that could in part be a function of how fast the backing store is for that SM.

In addition to tracking the canonical database state in its archive, Storage Managers can optionally maintain a journal of all updates. It's a best practice to always be running with journaling enabled. 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.

Figure 5: A single database can use multiple Storage Managers, each with a different backing store. If each SM uses a journal then it hides the latency of the backing store.

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. To ensure durability, the TE running the transaction must coordinate with the storage tier. This is referred to as the commit protocol.

In NuoDB the commit protocol enables a user to tradeoff performance and durability concerns. When any change is happening in the context of a transaction it gets sent to all peers that need to know about the change. As discussed above, that means any TE with the associated Atoms in-cache and all SMs. So a base line for acknowledging commit is that reliable messages have been sent to all interested peers.

As long as all of the Storage Managers don't fail simultaneously at this moment, then this ensures the data will made durable. Regardless, data will always be correct and consistent. For some applications this kind of k-safety[14] 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 Storage Manager before acknowledging commit to the client. This is tunable by running with a Remote:N setting.

In this context, N is the number of Storage Managers that must acknowledge data has been made durable before commit can be considered successful. For instance, Remote:2 requires acknowledgement from at least 2 Storage Managers[15]. Because updates are sent asynchronously, it's possible that the response arrives before the transaction is ready to commit, so there is no round-trip cost.

If a Storage Manager is running with journaling enabled then it can acknowledge an update as soon as the change has been written to the journal. Typically this can be done much faster than actually writing the change to the archive. This means that running with a journal not only helps with recovery on failure (and therefore improves durability guarantees) but it improves performance of transactions.

The NuoDB Architecture: Examples

This section provides a few concrete examples of the architectural issues discussed in the previous section. These examples use the smallest, fully redundant database (2 Transaction Engines and 2 Storage Managers) for illustration. From there it should be simple to extrapolate to how interaction works on larger deployments.

---

14 Stonebraker M. and Abadi, D. J. and Batkin, A. and Chen, X. and Cherniack, M. and Ferreira, M. and Lau, E. and Lin, A. and Madden, S. and O'Neil, E. and O'Neil, P. and Rasin, A. and Tran, N. and Zdonik, S., "C-Store: A Column-Oriented DBMS", VLDB '05, pages 553–564, 2005

15 See the section below on Geographic Distribution for other options to consider when configuring commit.

Cache Population

The caching tier in NuoDB is an on-demand cache where each Transaction Engine maintains a set of Atoms based on access patterns. There are two ways that this data could 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 could be created (e.g., 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 Storage Managers 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.

The second way a cache is populated is when there's a cache-miss on some required Atom. For instance, assume some Atom has been created but isn't currently in-cache at any Transaction Engine. The TE uses its Catalog Atoms to discover this and then goes to any Storage Manager 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 is likely to fetch it from TE1 because this will be much faster than retrieving it from the durable store.

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 Transaction Engine 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 a Storage Manager. 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 that will also be needed at the TE.

Figure 6: The on-demand cache puts the Atom in-cache at TE1 when created, and replicates it to TE2 only when needed.

Data Update

Assume the above example where an Atom was first cached at TE1 and then replicated to TE2, and now some data it contains is modified (e.g., 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 former is to verify the update may proceed and in this example if the update is happening on TE1, the Chairman, then the check is handled locally with no communications. The latter is then sent asynchronously to both the SM and TE2 to notify them that an update is occurring.

Had transactions been running on both Transaction Engines trying to update the same data (i.e., the same row in the table, not the same Atom) a conflict will occur. In this case the Chairman acts as a tiebreaker: whichever transaction got its update message to the Chairman first will "win".

Figure 7: When an update is made to some Atom in a TE the change is sent to all peers tracking that Atom.

Commit Protocol

In the previous example the pending update messages were sent asynchronously (over reliable channels) with no requirement that any Storage Manager 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 Storage Manager is active. In this case, the update is replicated to three hosts, all of which have to fail to lose the update. The tunable Commit Protocol is what provides flexibility on commit.

Figure 8: With Remote:1 the TE waits for one response. In this picture the slower service (S3) also uses a journal, so it can actually respond faster than a disk-based archive with no journal.

If the same update is run, but now the Remote:1 option is used, the Transaction Engine will wait to hear from at least one Storage Manager before acknowledging commit back to the SQL client. Running with Remote:2 will require acknowledgement from both SMs. Recall that each SM can use a Journal, so the acknowledgement is sent as soon as the diff is captured there.

Management and Operations Model

Above the two database layers is a management tier. As with databases, the management tier is a collection of peer processes. These processes are called Agents, and one runs on every host where a database could be active. Starting a management agent on a host is a provisioning step: it makes the host available to run database processes and visible to the management environment. This collection of provisioned hosts 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 DBA focuses on a given database and a systems administrator works at the management domain level.

Each Agent is responsible for the host it runs on (i.e. the local host). An Agent can start and stop Transaction Engine and Storage Manager processes, monitor those processes and the local host's resources, query and configure the running processes and perform other host-local tasks. Any Agent may be run with the additional role of Broker. A Broker is a Agent that has global view of all Agents in the Domain, and therefore all processes, databases and events that are useful from a monitoring point of view.

All connection brokers 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 Brokers are deployed. Provisioning a new host for a Domain is done by starting a new Agent peered to one of the existing Brokers.

When an SQL client wants to communicate with a Transaction Engine it starts by connecting to a Broker. The Broker tells the client a TE to use and the client disconnects before connecting directly to the Transaction Engine. This connection brokering is one of the key roles of a Broker, and means that a Broker is also a load-balancing point. Load-balancing policies are pluggable and can be implemented to support optimizations around key factors like resource utilization or locality.

Figure 9: An admin client sends a single management message to a Broker to start a process on some host. Once the TE is started management messages flow back to all Brokers.

Logical Administration and Templates

Just as an SQL programmer addresses a NuoDB database as a single, logical entity even though it's distributed across many processes, a systems administrator addresses a Domain as a single, logical point of management. This is done through any of the Brokers. They provide a single place to connect to a Domain, manage and monitor databases and ensure that the system is running as expected.

Using this single point of management or administration, databases can be started from Templates. A Template is a recipe for how to solve specific problems like maintaining full redundancy, scaling out on-demand, running in multiple locations or simply testing on a local system. Templates are written using a simple declarative language that captures deployment and service-level requirements[16]. Databases started from a Template will report whether or not they are currently meeting their expected requirements.

Database Backup and Provisioning

Taken together, this simple administrative model and the Storage Manager architecture support a non-intrusive approach to full and incremental database backup. You can run a redundant SM as part of the database, or on-demand when backup should be performed. 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

16 See the section below on Automation for more details on how Templates are used to automate the process of managing active databases.

with the running database and then continues active participation. The database itself is never taken down. Note that if you deploy against a file system or storage service that supports atomic snapshots (e.g., ZFS) you can capture a full, consistent backup instantly without shutting down any SM.

Figure 10: After issuing a stable shutdown command to any SM the archive is usable as a point-in-time backup or as a way of starting an independent database provisioned with the archive content.

This model for backup works in part because an SM can be started against any arbitrary archive, so any copy of an archive lets you roll back to the point in time 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 makes it well suited to address a number of typically challenging and mutually exclusive problems. This section covers a few of these problems, and highlights which aspects of the architecture are key in addressing them.

Single, Logical Database

NuoDB is a distributed database, composed of an arbitrary process deployment across an arbitrary set of hosts. Programming models like 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 effect on the application; an SQL client addresses what looks like a single database. Likewise, management of 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.

Geographical Distribution

NuoDB provides the model of a single, logical database that is always active and consistent both within a single data center and across data centers. This supports running a database across physically separate geographic regions (for instance, running a database across servers in Sydney, Singapore, Virginia, and so on). Common reasons for running a geo-distributed database are to achieve higher availability and fault-tolerance or to support applications and users that are geographically distributed themselves: for instance, a financial institution with offices at the major finance hubs around the world.

In memory Atoms in NuoDB TEs are partially replicated so only the objects needed are held in transactional caches, and 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.

Figure 11: Geo-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.

Update messages are sent asynchronously from TEs to SMs. It is up to the commit protocol to decide if 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 does crash 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.

Often in relational databases making these kinds of changes is expensive, sometimes requiring 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 where data is stored without structure so the burden is put 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 like adding, renaming or removing a column or dropping a table are done in constant time.

Operational and Analytic Mixed Workloads

NuoDB is a transactional system, meaning that it's well suited to support both Online Transaction Processing (OLTP) tasks and operational workloads like account management or session tracking. The operational model is typical of scale-out web applications, which need a database that can support many concurrent clients doing regular, small, localized updates. While techniques like sharding or replication are hard to apply to OLTP 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 Online Analytical Processing (OLAP) requires the database to handle longer-running transactions that span large regions of the data. Often the solution is to dump data from the operational database(s) into a separate service that is used only for doing analysis. NuoDB provides a scale-out architecture, supporting transactions that need to read large quantities of the

database. Because MVCC is used to enforce consistency, those long-running transactions can execute without causing conflict with the many short-running updates happening across the database. In this way NuoDB supports scale-out operational data deployments where online analytics operations need to be run on the same data set.

Because NuoDB has a flexible load-balancing policy, it's also possible to dedicate specific Transaction Engines to specific roles. For instance, a single database can be scaled out across smaller systems for typical operational access patterns. One or a few 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 that provide blogs or software as-a-service. Often these are supported by running one 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.

Figure 12: 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 will be 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 its processes can be shut down completely and re-started on-demand as needed. This capability is called Database Hibernation. Hibernation supports deployments where 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[17].

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 mentioned earlier, this makes it simple to support upgrades with no downtime. New Transaction Engines are started on new hosts to meet capacity requirements and then existing TEs can be shut down to perform upgrade. In this way a rolling upgrade of

17 http://www.nuodb.com/techblog/2013/04/09/extreme-density-on-hps-project-moonshot/

software or hardware is supported 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 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 HA

NuoDB provides traditional, proactive approaches to High Availability by running with additional Transaction Engines, Storage Managers and Connection Brokers. 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 the domain, and the Transaction Engines that were running on the original host can quickly be re-started on the new host. If Storage Managers 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 is brought online.

Similarly, a host can be run as part of a domain for the sole purpose of being available to pickup work when another host fails. In this way the window for reduced availability is cut to the time it takes to observe failure and re-start 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.

Scale-Out Performance

A cloud-scale database should be able to handle both traditional and modern transactional workloads. The latter is probably obvious. Supporting traditional applications that weren't designed with sharding, passive replicas or scale-out architectures, however, provides a migration path and simplifies the number of databases that a company needs to maintain. In either case, 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 data cache (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)[18], designed to simulate modern web-facing workloads on back-end databases. It can be tuned along factors like dataset size, read/write bias, volume of queries and number of servers. The second benchmark is DBT-2[19], an open source implementation of the TPC-C[20] 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.

"NuoDB enables us to add capacity seamlessly when we need it, and just as seamlessly scale back when demand shifts. That's extremely critical to managing costs."

Scott Lemon
CTO
DropShip Commerce

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. The latest published results are available online[21]. 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

18 Cooper, B. F. and Silberstein A. and Tam E. and Ramakrishnan R. and Sears R., "Benchmarking Cloud Serving Systems with YCSB", ACM SoCC '10, June 2010

19 http://sourceforge.net/apps/mediawiki/osdldbt/index.php?title=Main_Page#dbt2

20 Transaction Processing Performance Council, "TPC Benchmark™ C", February 2010

21 http://dev.nuodb.com/tags/benchmark

Transaction Engines 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 global access in a real-world manner. Both of these benchmarks can be used to show horizontal scale by scaling out Transaction Engines 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. Conventionally 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 cache[22]. This helps minimize latency and centralize database management. It also means that extra coordination is needed to either keep the distributed cache consistent, shard the application logic or 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 13: Depicted on the left is a common web-scale deployment pattern. 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 Transaction Engine can be run on each web host. This affinity model provides the same low-latency and in-memory performance of something like 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

22 http://dev.mysql.com/doc/refman/5.7/en/ha-memcached.html

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 Transaction Engines on separate hosts. To model a more realistic workload, where a subset of the database is more active and then there is a trail-off pattern, Transaction Engines 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. Again, results are available online, and are updated as new tests are run.

Beyond these popular benchmarks other tests are being run, and a new benchmark is available. Named "Storefront" its goal is to combine behaviors from both DBT-2 and YCSB to model a modern shopping site running in multiple data centers. It demonstrates aspects of scale, low-latency and the importance of simultaneously maintaining high availability.

Future Directions

This paper captures the current architecture of NuoDB, and makes the argument that by starting with a flexible, composable model the same implementation can be used to address very different problems. In that same spirit, the architecture continues to evolve to support new use-cases. Without covering all of the ways that NuoDB is evolving, here are a few key architectural elements that are currently in development for future versions.

Partitioned Storage

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

There are, however, several valid use cases that require each Storage Manager to contain only a subset of the total database. Some have to do with performance, like applications with

high insert rates, or with large databases that cannot easily be contained by a single storage device or service (note that for this latter case, storage level distribution can be handled today with NuoDB by the underlying KV-store used by an SM). Others are focused on explicit choices about where to store data, for example minimizing coordination between geo-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 (see below).

Because all SQL access to a database is through Transaction Engines the durable state of the database can be partitioned without any change to the programming model. In other words, automatic data partitioning at the durability level doesn't affect the client view of a single, logical database. This evolution will give NuoDB users the choice to continue with the existing operations model or let a database store subsets of the total durable state at different Storage Managers.

Distributed Queries

Today, each open SQL transaction is executed on a single Transaction Engine. This leads to caches forming around access patterns and makes conflict management extremely efficient when a given object is only in one cache. It also makes failure handling simpler, and minimizes the number of messages required for a given transaction.

As was previously discussed, part of the internal Atom structure is the directory for where other Atoms are available in-cache. This means that one use for distributing transactions across TEs is to move work to caches instead of moving cached objects, which is particularly effective for queries that can be parallelized. A more powerful use is applying Map-Reduce behavior to analytic queries.
By supporting a transaction model where execution is shared across TEs analytic queries can be run in parallel on the hosts that already have as many as possible of the required objects in-cache. This model will provide support for operational analysis that takes advantage of the total memory-space of all hosts.

Automation

This paper has already covered how NuoDB makes database provisioning, monitoring and management simple. The Brokers are a single place to monitor running databases, and the administration tier supports on-demand scale-out, scale-back or migration. This can all be scripted for automatic behavior, so that systems administrators can maintain control of how and where

database processes are run.

NuoDB knows about the running processes and available resources on each host, and databases can be started based on some template that defines operational requirements and configuration rules. Taken together, the software can automatically administer itself meeting those requirements. For instance automatically starting a TE on a new host when some existing host fails, or auto-scaling a database when it needs more capacity.

An automation model provides high availability with low administrative overhead. It is also the basis for supporting multi-tenant deployments where processes will be moved dynamically to make the best use possible of available resources. This direction simplifies administration. It also will lower total costs by maximizing resource utilization across arbitrary deployments.

Commit Protocol and Consistency

The flexible commit protocol, discussed earlier, gives users choices around durability of data and latency on commit. By defining where data must be durable before commit is acknowledged a deployment can be optimized for its expected workloads. This is especially important for databases that span multiple geographic regions. Requiring commit acknowledgement in only the local region guarantees that data is always durable with only a local synchronous message gating the response.

Running with this commit mode has another advantage: it provides guarantees in the case of network partition about where the most recent version of every object must be available. Paired with the rules about chairmanship, and building on the previous discussion of storage partitioning, NuoDB can support a model where strict disjoint partition can be sustained and the database can continue to run (supporting a known subset of SQL). This will give users the option to continue running the full database, even on network partition, in a mode that is guaranteed to be consistent and reconcilable when the network re-connects.

Conclusion

This paper has covered the key architectural elements of NuoDB, and shown how that architecture is flexible enough to address a wide variety of use-cases. It has also shown how that architecture and its simple, peer-to-peer model are capable of evolving to solve new challenges. It is this flexibility that will allow NuoDB to continue evolving to solve the next generation of cloud-scale database problems.

For more details of the protocols there is the Green Book[23], a deeper-dive into the internals of the system. There is also an active engineering blog[24] that covers details of many specific parts of the architecture, from MVCC and caching to data durability to the user-experience.

23 http://go.nuodb.com/the-green-book-request.html

24 http://dev.nuodb.com/techblog

About NuoDB

Everyday businesses face challenges with application deployments, maintaining business continuity and providing outstanding application performance. NuoDB leads the industry with a cloud database management system to solve these problems.

NuoDB provides scale-out performance, continuous availability and active/active operation. It's a single, logical database easily deployed in multiple locations simultaneously. This unique capability is unavailable in any other SQL product.

Launched in 2010 by industry-renowned database architect Jim Starkey and accomplished software CEO Barry Morris, the company is based in Cambridge, MA.

Used by thousands of developers worldwide, NuoDB's customers include automotive after-market giant AutoZone, Dassault Systèmes, DropShip Commerce, and other innovative organizations. NuoDB was one of only nineteen vendors cited on the Gartner Magic Quadrant for Operational DBMS.

Try NuoDB yourself. For a free download: www.nuodb.com/download

30

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

Version 1 – 10/1/2013

© 2013 NuoDB, Inc., all rights reserved.

The following are trademarks of NuoDB, Inc.: NuoDB, The Elastically Scalable Database, NewSQL Without Compromise, Nuo, and NuoConnect.