NuoDB® Emergent Architecture

A 21st Century Transactional Relational Database Founded On Partial, On-Demand Replication
Executive Summary

Many years ago IBM created a technical publication called a “Redbook.” Good idea. It’s been copied with varying degrees of success since its invention. NuoDB aims to outgreen the Redbook with our own technical guidance on Cloud Data Management Systems, the new 21st century database category, with our new series of NuoDB Greenbooks.

NuoDB is a complete re-think of relational databases with innovative support for the cloud’s dynamic, asynchronous nature.

NuoDB is a Cloud Data Management System (CDMS) that is:

- 100% SQL
- 100% ACID
- 100% Elastically scalable
- 0% administrative hassle.

All in a shared-nothing, distributed architecture that scales elastically; leverages existing SQL expertise; provides blistering performance and is amazingly simple to use.

Before NuoDB, relational databases were built on synchronous, centralized client/server architectures that are incompatible with the notion of elastic scalability in the cloud. As a result, scaling a SQL database typically requires tedious and expensive database design and administration involving sharding, caching, clustering, replication and performance tuning. No more. NuoDB is an easier, more elegant operational database that eliminates all that heavy lifting. Your mobile apps are simple to download and use, the backend DBMS should be the same.

We’ve written the rules for 21st century relational databases and created a database that’s all about you and your application.

This first Greenbook will provide a deep technical dive into the core technical concepts of the NuoDB architecture.
Overview

NuoDB recreates the relational database on a new foundation: partial, on-demand replication.

Application developers recognize the benefits of the traditional relational database: transactions that provide a consistent view of data, related changes that succeed or fail as a unit, and semantics that protect applications from details of physical data storage. However, relational databases have not met the challenges of data growth, availability, and audience. More people need more access to more data all of the time. Traditional relational databases focus on moving data on and off disks through the narrow window of their cache using a single central point of control. That architecture worked in the early 1980’s when the ancestors of today’s relational systems roamed the earth. 21st century data and computing environments – huge disks, huge data, memory in gigabytes, and fast reliable networks – demand a new approach.

The NoSQL world insists that the new approach is to throw out the benefits of a SQL database along with the drawbacks of its ancestry. In the absence of transactions, protecting data that is volatile, valuable, and shared requires guarantees that exceed the skill of most application programmers. Beyond that, the time and talents of application programmers are much better used in developing attractive, functional user interfaces than reproducing capabilities that have been built into databases for thirty years. NuoDB provides the usual database guarantees of atomic transactions, data consistency, isolation of one transaction from its contemporaries, and durability of committed data through standard SQL. But its underpinnings are very different.

Beneath the traditional SQL interface, NuoDB represents the database as a variable number of in-memory objects on a variable number of cooperating computer systems, archived in multiple locations. NuoDB separates transaction management from disk storage, running Transaction Engines on some machines and Storage Managers on others. Transaction Engines manage data through partial, on-demand replication. Data resides where it is needed and used. When a client of a particular Transaction Engine requires data that is not available locally, the Transaction Engine asks its peers to provide a copy. Data that is no longer necessary on a particular Transaction Engine can be dropped as long as at least one Storage Manager has a durable copy. When a new Transaction Engine starts to serve a database, it first learns the topology of the machines currently running that database, and where to look for data it needs to respond to the requests of its clients. NuoDB is elastically scalable.

This paper sets out to explain why NuoDB is necessary and how it provides partial, on-demand replication and ACID transactions. The technique is called "emergent" because it depends on the ability of large numbers of distributed objects to co-ordinate their behavior without central control, like a flock of birds or a school of fish swirling, diving, splitting and merging, guided by a cooperative intelligence.
Document Overview

Partial, on-demand replication is one complex topic. Preserving the integrity of transactions in a shared-nothing, elastically scalable collection of computers is another. The concepts include a new programming paradigm for distributed applications, why a new paradigm was necessary, how NuoDB works from the point of view of an application and its internals. It touches on topics of data warehouses, virtualization, and both public and private clouds as well as traditional database concerns like managing conflicts between updates, detecting deadlocks, and enforcing constraints. This paper attempts to explain the environmental changes that make NuoDB necessary and how competing technologies fail to address the real-world application needs in the case of data that is volatile, valuable, and shared. Next, it describes the major characteristics of NuoDB. The next section of the paper describes the external behavior of NuoDB, touching on topics like avoiding single points of failure, availability, system management, and multi-tenancy. Lastly, it describes the way NuoDB implements partial on-demand replication.

If you want to know how NuoDB fits in the universe of data management tools, the first section should interest you. If you would like an overview of NuoDB, you’ll find it in the second section. The third section addresses the question of how NuoDB fits into your world. The fourth section is for the reader who wants to understand the technology advances in NuoDB. This paper touches lightly on a number of topics that deserve more detailed explanations and includes links to a variety of papers that delve more deeply into topics like MVCC, network partition and the CAP theorem, and how a query travels through NuoDB to return with data for the application.

Why was NuoDB Necessary?

A decade ago, the database market was stable. Oracle, IBM DB2, Microsoft SQL Server, and Sybase dominated the market, with some other small players in the commercial arena, and some bleeding-edge developers building web sites on MySQL. Databases were relational, largely transactional, and ran on heavy iron. As recently as 2007, Hellerstein, Hamilton, and Stonebraker wrote “Since market forces only support a few competitors at the high end, only a handful of successful DBMS implementations exist.” The market forces were about to change drastically.

The New Market

One change was the emergence of datacenters demonstrated by the success companies like Google and Amazon had in “scale out” rather than “scale up” using thousands of commodity processors. The application of “scale out” to databases was not immediately apparent because those systems are “shared nothing”, meaning that each machine has its own memory and disk, not accessible by other machines. Hellerstein, Hamilton, and Stonebraker found this environment unfriendly to most databases. The most common technique employed by DBMSs to support these clusters [shared-nothing parallel systems] is to run their standard process model on each machine, or node, in the cluster... The main difference is that each system in the cluster stores only a portion of the data... The tables are spread over multiple systems in the cluster using horizontal data partitioning to allow each processor to execute independently of the others.

Good partitioning of the data is required, however, for good performance. This places a significant burden on the Database Administrator (DBA) to lay out tables intelligently, and on the query optimizer to do a good job partitioning the workload.

Explicit cross-processor coordination must take place to handle transaction completion, provide load balancing, and support certain maintenance tasks. For example, the processors must exchange explicit control messages for issues like distributed deadlock detection and

2 http://mobile.blogs.wsj.com/cio/2012/07/09/cloud-is-bright-spot-in-global-it-spending
two-phase commit... This requires additional logic, and can be a performance bottleneck if not done carefully³.

The academic authorities declared shared-nothing scale out unfriendly to databases. Processors, disk, memory, and networks all grew enormously, and all but disks got much faster. These changes were most significant in low-end commodity machines, making the scale out solution ever more compelling. Most significantly, the phenomenon called “Web 2.0” exploded, moving even casual computer users from e-mail and browsing static websites to a personalized, interactive, web experience. Then smart phones and tablets became gateways to the web and downtime disappeared. Internet access was no longer a sometimes thing, it was everywhere, all the time. These changes caused exponential growth in the amount of data that needed storage and access. Social media companies grew from concept to earth-shaking overnight.

The new database market requires fast response, constant availability, and the ability to scale to match the growth of app builders’ dreams. Variable sized clusters of shared nothing machines are the obvious way to meet that demand, but leading authorities say it can’t be done. So what has to be sacrificed to get performance, availability, and scaling?

Adapt Existing Relational Systems
Organizations adapted traditional relational databases to perform in a shared-nothing scale out environment in three ways: sharding, partitioning, and master/slave replication.

In a sharded system, applications run against several databases each of which contains part of the whole picture. The application is responsible for maintaining consistency between databases. In a partitioned system, applications run against what appears to be a unified database, but is in fact several separate databases. The advantage of partitioning over sharding is that the application does not need to know where to find a particular type of data. No application changes are needed when the load changes and the data must be repartitioned. The advantage of sharding over partitioning is that it requires no direct support from the database management system, with the possible exception of support for a two-phase commit. Both suffer performance problems if a query touches data that crosses partitions or shards. Neither addresses availability or scaling dynamically.

Master/slave replication restricts update queries to the master copy of the database or the master copy of a shard, and allows readers to use a copy of the database which may not be completely up to date. Master/slave replication provides higher availability and can be combined with sharding to scale out updates. However, it does not guarantee that slaved versions of data are consistent with the master at any instant.

Drop SQL
Giving up on relational systems and ACID transactions altogether was another approach. Much of the enormous proliferation of data was not very valuable and could be somewhat inconsistent. Nobody cared if the first three hits of a search for Chanel Sunglasses weren’t the same every time. Since available databases didn’t and, according to theory, couldn’t handle the volumes of data and access latency requirements of the new world, non-database, non-SQL solutions proliferated. Many of their proponents champion the freedom of totally unstructured data. And for many applications, they are right. But for data that is volatile, valuable, and shared, eventual consistency is not good enough.

More recently, some NoSQL advocates have questioned the advantages of schema-less and transaction-less data management. One result of that questioning was the Unql⁴ project, a query language that attempted to simplify the transition of SQL skills to the NoSQL world and apparently ran into the challenge of creating a standard interface for a host of non-standard data

³ M. Hellerstein, M. Stonebraker and J. Hamilton op. cit. p. 168
⁴ http://www.dataversity.net/unql-a-standardized-query-language-for-nosql-databases
stores. Others, like Tarantool/Box suggest that NoSQL benefits from the addition of transactions.

Google is moving toward both SQL and transactions. At the NoSQL Matters conference in May of 2012, Olaf Bachmann from Google presented a paper called NoNoSQL about the evolution of interfaces that lead to grafting SQL on MapReduce data stores to provide a language that all Googlers could use. In October 2012, Google described its new database management system: Spanner a “scalable, multi-version, globally-distributed, and synchronously-replicated database... supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes.”

Data that is valuable, volatile, and shared requires consistency most easily obtained from a transactional database. Programming in a widely known high-level data access language may reduce flexibility, but it also reduces the skill-level developers need to produce reliable applications.

Create New Relational Technologies
The new database market inspired new relational technologies in a wide variety of formats: database appliances, MySQL back-ends, database as a service, data routers, etc. All the new technologies that use clusters of shared-nothing computers partition their data, making specific rows reside on specific systems. Some rely on replicated partitions scattered around the cluster. But all suffer performance problems when an application requires and generates data that crosses partitions.

Look at Problems and Opportunities
What are the problems to solve in putting a transactional, relational database on a cluster of shared nothing machines? What aspects of traditional database technology need rethinking?

One certainty is that disk speed has not increased as much as processor speed, memory size and speed, and network speed and reliability. So, think of the database as in-memory objects, not disk pages.

Use the disk for ultimate durability and to hold archival data, but expect the working portion of the database to be in memory, shared across a high speed network.

A database manager built on clusters of computers can use some computers for disk access and others for responding to client requests.

Another well-understood fact is that application load varies. The ideal system scales dynamically, allowing new machines to be introduced to a running database and become effective immediately. On the other hand, a database application should not require even one dedicated machine if its load does not justify the cost in hardware, power, and heat of even a single machine.

At the same time, the system must provide the traditional guarantees: committed changes are durable, regardless of failures, even multiple failures. Concurrent transactions cannot overwrite each other’s changes, even if they run on separate machines. Transactions must have a consistent view of their data and succeed or fail as a unit.

A database for the new market must run on hard iron, virtual machines, private clouds, public clouds, and across clouds. It must be designed for continuous operation.

Those are the problems NuoDB’s technical founder Jim Starkey, an industry-renowned database architect and innovator, wanted to address – using a new paradigm for programming on clusters of shared nothing machines. The architecture opens a variety of possible future directions including “time-travel”, the ability to create a copy of the database that recreates its state at an earlier
time; “cloud bursting”, the ability to move a database across cloud systems managed by separate groups; and “coteries” a mechanism that addresses the CAP Theorem by allowing the DBA to specify which systems survive a network partition to provide consistency and partition resistance with continuous availability.

**NuoDB from the Outside**

**From Various Points of View**

From an application’s point of view, NuoDB looks remarkably like any other SQL-based relational database. It has a native ODBC/JDBC interface, drivers for JRuby ActiveRecord, PHP/PDO, and Hibernate, and supports a large subset of ISO SQL. An application connects to a database, runs transactions, accesses and updates data. The physical topology of the database is irrelevant to the application. From the perspective of an application developer, the one difference between NuoDB and other relational databases is that NuoDB supports several different commit protocols, allowing the application to request the degrees of durability it requires.

From the database administrator’s point of view, NuoDB is also quite familiar, except that it lacks any kind of placement control. It has no partitions, no table spaces, and no file locations. NuoDB automatically places data where it is needed without requiring the DBA to explain usage patterns.

From the system administrator’s point of view, NuoDB is a set of processes, working together to provide the services of a transactional database. Every machine – virtual or real – running NuoDB has an Agent process, which starts processes on that machine and reports statistics on those processes. Every NuoDB database must have at least one Transaction Engine (TE) to handle client requests and one Storage Manager (SM) to persist data. Certain Agents are designated as Brokers. Initially connections from client applications go to a Broker. The Broker identifies a Transaction Engine for the connection and introduces them to each other. Thereafter, the application communicates directly with that Transaction Engine. The Transaction Engine communicates with other Transaction Engines and Storage Managers to access and persist data.

The Agents and Brokers also collect statistics from the Transaction Engines and Storage Managers. A control application or script gathers the statistics to allow the system administrator to understand resource usage.

**Database and Domain**

NuoDB recognizes two different administrative functions: the Database Administrator and the System Administrator. The Database Administrator has full access to the content of the database, creates and manages users, defines and assigns roles, and grants privileges to roles and users. The System Administrator manages hardware: the machines that run NuoDB processes. The System Administrator starts the processes that serve the database. The Database Administrator controls the content being served.
A database is served by several machines, and one machine can serve several databases. As the database demand scales up and down, the System Administrator adds machines to the database or removes them from it. The group of machines that a System Administrator controls is called a “Domain”. A single NuoDB Domain can include several databases.

Consider an example. The datacenter of a company that manufactures fireworks (Acme Fireworks Display) has separate databases for catalog related information, customer and supplier information, and safety checks. Each of those databases normally runs on a set of eight machines that give each database three or four Transaction Engines for concurrent access and two Storage Managers for durability. For most of the year, those processes run easily in the company’s datacenter: three databases in a single local domain.

However, customer inquiries rise seasonally in May and November, so the company adds space on a public cloud to its domain to handle the temporary demand for the catalog information. At the same time, internal machines that had supported the catalog database join the customer and supplier database.

**Transaction Engines**

Transaction Engines are the front line of NuoDB. They parse, compile, optimize, and execute SQL commands on behalf of client applications. Adding more Transaction Engines to a database allows it to service more clients, and provide higher throughput.

When a new Transaction Engine joins a database, it has a map of where to find all the information in the database. When an application needs data that is not currently available locally, the Transaction Engine consults its map, finds a Transaction Engine that has the information, and asks that Transaction Engine to send the information to it. If no other Transaction Engine has the information, the Transaction Engine asks a Storage Manager for it. The Storage Manager gets the information either from its memory or from disk. Information flows to the Transaction Engines that need it: partial, on-demand replication. No one Transaction Engine has every piece of information, but every Transaction Engine can get any piece of information when needed to respond to a client request.

Adding Transaction Engines allows NuoDB to support more clients and higher throughput.

**Storage Managers**

Storage Managers are the back office of NuoDB. They do not process SQL. They do not understand schemas, tables, rows, or indexes. What they do understand is how to ensure that changes to the database are
durable when a transaction commits and to retrieve information that has been dropped from the memory of the Transaction Engines serving the database. The information that Storage Managers handle is externally pairs of small keys and large values. NuoDB Storage Managers include a native implementation of a key/value store, but they can also use more sophisticated, higher performance key/value stores like the Hadoop Distributed File System (HDFS).

In the NuoDB Starlings release, each Storage Manager maintains a complete copy of the database, so each additional Storage Manager makes the database more durable. However, nothing in the architecture would prohibit partitioning a database among Storage Managers, but a partitioned database would require at least two Storage Managers per partition to provide durability in the case of a disk failure. Partitioning a NuoDB Database among Storage Managers would not degrade the performance of applications that require data from multiple partitions, as is common in partitioned databases, because Transaction Engines continue to have access to the whole database.

Adding Storage Managers to a NuoDB database makes it more resistant to failure. Faster disks, disk arrays, and sophisticated key/value stores all improve performance of a NuoDB database with high I/O loads.

Adding more Brokers to a NuoDB database reduces the risk of isolating a database due to broker failures.

Network Communication
Every machine serving a database communicates with every other machine using proprietary protocols layered on TCP/IP. Brokers and Agents identify themselves with the Domain password. Transaction Engines and Storage Managers identify themselves with database password. Clients identify themselves to the database with their individual passwords. NuoDB does not store passwords, but instead computes password verifiers that cannot be reverse engineered to create the password. All network communication is encrypted using the stream cipher ARC4. The encryption seed is computed initially from the password and a session specific tag, avoiding the worst weakness of that cipher.

High speed networks and switches reduce latency in a NuoDB database.

Machine Configurations and Multi-Tenancy
From day one, the design goal for NuoDB has been to provide a transactional database that runs on multiple machines and provides elastic scalability. While it is possible to run a Storage Manager and Transaction Engine for the same database on a single machine or on separate virtual machines running on the same physical computer, that configuration performs badly and provides less durability than a configuration in which each Transaction Engine and each Storage Manager for a particular database runs on separate hardware.

However, as the Acme Fireworks example shows, NuoDB supports multi-tenancy – meaning that one set of computers can manage multiple databases simultaneously. Multi-tenancy provides durability and performance for lower-load database applications.

When the load on one database increases, adding more machines lets it scale up to handle the new load.

Agents and Brokers
Every machine that runs a Storage Manager or a Transaction Engine also runs either an Agent or a Broker. Agents start Transaction Engines and Storage Managers and collect statistics from them. Brokers are agents with the additional responsibility of assigning client connections to a Transaction Engine. Once the Broker has introduced the client to a Transaction Engine, the client and Transaction Engine communicate directly without further reference to the Broker. Agents and Brokers are part of the NuoDB Domain infrastructure, and enable the system administrator to add machines to a database that needs more power. Brokers can assign a client to any Transaction Engine serving the database.

10 http://en.wikipedia.org/wiki/RC4 ARC4 means Alleged RC4, an unauthorized but legal implementation of the RC4 algorithm
Commit Protocols
One difference between NuoDB and other relational databases is that in NuoDB the DBA can set a minimum level of durability for a database and application developers can require a higher level of durability for specific applications. The mechanism for determining the durability of a transaction is governed by the commit protocol used. Because the commit sequence is critical to performance, the messages are not batched, but sent immediately. The default commit protocol is “Remote 1”, and guarantees that at least one Storage Manager has copies of all of a transaction’s changes before the transaction commits. These are the steps in a Remote 1 commit:

1. The transaction issues a commit request to the Transaction Engine it connected to – the originating Transaction Engine.
2. The Transaction Manager Atom on the originating Transaction Engine sends a pre-commit message to the Transaction Manager Atoms on Storage Managers.
3. The Storage Managers receive the pre-commit message and each one
   - Flushes its replication message journal, using direct I/O, and
   - Sends a “CommitACK” message to the Transaction Manager Atom on the originating Transaction Engine.
4. The Transaction Manager Atom on the originating Transaction Engine receives at least one “CommitACK” message,
   - Sends a “committed” message to all other Transaction Manager Atoms, and
   - Lets its Transaction Engine know that it can respond to the transaction saying its commit succeeded.

This sequence guarantees that all changes made by the committed transaction are on durable storage in at least one place. If all the Storage Managers serving the database fail simultaneously, applying the journaled replication message to the archived atoms restores all changes made by transactions that had committed before the catastrophe. Applications or databases with higher durability requirements can require a commit protocol of “Remote 2”, which requires receiving “commit received” messages from two Storage Managers, or “Remote N” where N is any number up to the number of Storage Managers serving the database. All follow the same steps, except Step 4 requires more “commit received” messages.

Applications and databases that must consider the possibility of a network partition can use the commit protocol “Majority” which requires “commit received” messages from a majority of the active Storage Managers, or, in the case of a tie, from half of the Storage Managers plus the oldest Storage Manager serving the database. This feature is not guaranteed for NuoDB Starlings. In the absence of this feature, requiring a majority of Storage Managers to participate in the commit prevent the two halves of a partitioned set of machines serving a database from diverging.

Databases with low durability and high performance requirements can be created with a commit protocol of “Local”, which allows the Transaction Manager Atom on the Transaction Engine running the committing transaction to declare the transaction committed after Step 1. The failure of all Storage Managers for a database using the Local commit protocol may result in the loss of changes made by some transactions that were declared committed, but those transactions disappear completely. The Atomicity is maintained even with a Local commit.

NuoDB from the Inside
The challenge for NuoDB’s partial on demand replication is to distribute information freely and maintain transactional consistency across dozens of machines without saturating the network. That challenge inspired a new programming paradigm for distributed objects: peer-to-peer object replication. In NuoDB parlance, the
objects being replicated are called “atoms” to reduce the confusion between the objects in the programming language and the distributed objects they implement. NuoDB holds a patent on peer-to-peer atom replication as a technique for implementing distributed applications. Like objects in Java or C++, atoms have types, equivalent to classes, and instances. Unlike objects in an object-oriented language, atom instances in NuoDB have copies. For example, NuoDB has an atom type called Table. Each table in the database is represented by an instance of the Table Atom. Every Transaction Engine using a particular table has a copy of that table’s Table Atom. The individual copies – peers – replicate changes among themselves.

Of course, atoms are not actually maintaining TCP/IP connections among themselves; they communicate through the messaging system that is part of every Transaction Engine and Storage Manager. For the purpose of describing the internal behavior of NuoDB, think of the messaging system as the telephone network. When you call a friend, a vast number of operations happen in the telephone network that are complex and fascinating, but you don’t need to understand any of it to call your friend. You punch in your friend’s phone number and talk. NuoDB has a messaging subsystem, but the effect is that one atom sends a message to copies of itself – and its peers.

Atoms never send messages to atoms that are not its peers. The Transaction Engine on a specific machine working on behalf of a specific transaction can make changes to a group of related atoms. For example, inserting a single row into a table changes a Records Atom and a Data Atom at a minimum. It may also change Index Atoms, Blobs Atoms, and Data Atoms associated with Blob or Clob data. Each atom changed sends replication messages to its peers, so that row appears everywhere.

Atoms send a variety of messages to copies of themselves, but the most common are replication messages, indicating that a Transaction Engine has made a change to the atom that should be applied by all copies of that atom. Replication messages are short (18 bytes on average) and asynchronous. The messaging system of each Transaction Engine buffers replication messages and sends batches of messages when the buffered messages reach a certain size or when a special message requires immediate action. Under load, an eight-node NuoDB database processes 80,000 messages per second.

When an atom receives a replication message from a copy of itself, it applies the changes. At any specific instant, different copies of an atom may differ because they have sent replication messages that have not yet been received and applied or messages have been sent to them that they have not received. When all messages are received and applied, all copies of each atom are consistent. It is extremely important to note, that “atom consistency” in context of NuoDB differs from the “eventual consistency” of NoSQL systems because each NuoDB transaction sees a consistent view of the database.

**NuoDB, MVCC, and Transaction Consistency**

NuoDB uses Multi-Version Concurrency Control (MVCC) to manage the consistency of both read and write transactions. MVCC has been implemented in a number of database systems, using different rules and techniques. Among them, the implementation of Firebird/InterBase and PostgreSQL are closest to NuoDB. Oracle and InnoDB use MVCC to provide consistent reads and locks on rows and ranges to avoid write/write conflicts. NuoDB has no two-phase locks – no locks that are held for the duration of a transaction; all write/write conflicts are resolved through comparing versions of rows.

The simplest explanation of MVCC is that update and delete operations create new versions of rows rather than replacing existing rows. Each transaction knows which row versions were created by transactions that had committed when it started. It reads the newest version of the row that was committed when it started.
and ignores newer versions. When a transaction attempts to update or delete a row, the Transaction Engine must determine that the newest version of the row is committed and visible to the transaction making the change. If so, the change is allowed. If not, the change fails. NuoDB adds more complexity in that changes may be made on separate Transaction Engines simultaneously. That situation is monitored by a Records Atom Chairman, which resolves inter-engine disputes.

NuoDB’s implementation of MVCC provides consistent transactions even though actions on different Transaction Engines are asynchronous. There are two conditions that make transaction consistency possible. First, all messages from Atoms on one Transaction Engine are received by other Transaction Engines and Storage Managers in the order in which they were sent. Second, each transaction runs on one Transaction Engine. The second condition could be relaxed in later versions of NuoDB, at the cost of complicating transaction bookkeeping considerably.

The result of the two conditions is that all atoms that a transaction changes have sent replication messages to all copies of themselves before the Transaction Manager Atom sends the commit message to other machines serving the database. As a result, if a transaction considers another transaction to be committed, all changes made by that transaction have been received. Some complications arise when some of the atoms needed by a transaction run by Transaction Engine A are only available from Transaction Engine B which may see a different state of a transaction run by Transaction Engine C. The solution to that problem is called “relaying”. Relaying provides consistency even when atoms are flying around the database.

**NuoDB Atom Types**

**Catalogs** are an important subset of the NuoDB Atoms. They track the locations of atoms all across the set of Transaction Engines and Storage Managers serving the database. Other atoms form a rough hierarchy with the Database at the top managing Schemas which manage Tables which manage Records and Data. They understand the mapping from the name of a part of the database the identifier of the atom that contains it. The catalog, on the other hand, knows where to find a particular atom by identifier – locally or from the most responsive remote location13. In version 1, NuoDB has two levels of catalogs, one Master Catalog, and one catalog for the atoms that make up each table.

The **Master Catalog** Atom is the foundation atom, the one from which all others can be found and which tracks the state of all the machines running Storage Managers and Transaction Engines for a database. Every Transaction Engine and Storage Manager has a copy of the Master Catalog Atom. Because the Master Catalog is everywhere, its scope of work is limited to tracking atoms that are created and dropped relatively infrequently. The Master Catalog tracks every copy of the Transaction Manager Atom, each Schema Atom, Table Atom, Table Catalog Atom, and Sequence Atom. Whenever a Transaction Engine creates a new table or sequence, its Master Catalog notifies all other Master Catalogs of that change. Every Master Catalog is notified each time a Transaction Engine requests a copy of any of those atoms.

The **Database Atom** is present on every Transaction Engine and Storage Manager serving the database. It holds information about the database as a whole: authentication information, user privileges, etc. It manages Schema Atoms.

The **Transaction Manager Atom** is also present on every Transaction Engine and Storage Manager. It tracks the state of all transactions active in the database. Whenever a Transaction Engine starts or ends a transaction, its Transaction Manager Atom sends replication messages to all its peers.

---

12 NuoDB supports several transaction isolation modes. This paper assumes that transactions run in the default isolation mode. Other modes, “write committed” and “read committed”, sacrifice consistency for throughput.

13 A NuoDB Transaction Engine determines the responsiveness of other Transaction Engines and Storage Managers by sending a ping to each of them once a second and timing the return. A slow return may indicate a slow network connection or an overloaded machine. In either case, when a Transaction Engine needs an atom that is not available locally, it asks the most responsive Transaction Engine that has a copy of the necessary atom. If no Transaction Engine has the atom, it asks the most responsive Storage Manager.
The **Schema Atom** creates and Table Atoms, Table Catalog Atoms, and Sequence Atoms for tables and sequences created in a particular Schema. It also tracks roles and their privileges. The Table, Table Catalog, and Sequence Atoms are separate from the Schema, so it is possible for a Transaction Engine to have Table Atoms present without their parent Schema Atom. The Schema Atom is needed only when creating or dropping tables and sequences. Each Schema in the database has a distinct Schema Atom, and each Schema Atom may exist on many Transaction Engines and Storage Managers.

The **Sequence Atom** creates values for automatically generated keys and other uses of SQL Sequences.

The **Table Atom** creates Records Atoms and Index Atoms for a table. It tracks the format or physical definition of the table – the name, type, and order of columns in the table. When the format of the table changes as the result of an Alter Table statement, the Table Atom creates a new format, but also remembers the old format so it can direct the transformation of rows from an older format to a newer format. The Table Atoms does not contain the data for the table, only the information necessary to find and interpret the data.

The **Table Catalog Atom**, like the Master Catalog Atom, exists to track the locations of other atoms and all their peers. Unlike the Master Catalog Atom, the Table Catalog Atom tracks volatile atoms, but only those associated with a specific table. The Table Catalog Atom tracks the locations of all the Records Atoms, Blobs Atoms, Data Atoms, and Index Atoms associated with a database table.

The **Records Atom** tracks all versions of a set of rows in a table. Each Records Atom tracks two-thousand rows and their versions. For each version of each row, the Records Atom stores the identifier of the transaction that created the version, the format of the table definition that was current when the version was created, a number that indicates the order of versions, and an indirect pointer to the actual data. All data is kept in Data Atoms. The Records Atom is the primary locus of multi-version record management. Index atoms have some knowledge of MVCC to facilitate the removal of unnecessary entries.

The **Blobs Atom** is a simpler version of the Records Atom because Blobs are not versioned. It provides an indirect pointer to Blob and Clob data. The actual data is stored in Data Atoms.

The **Data Atom** holds data. Blob data and row data do not share Data Atoms. The size of most other atoms is defined by the content they must contain, in the case of Master Catalog Atoms, Table Catalog Atoms, and Table Atoms. Data Atoms have a fixed target size of 50K bytes. When a Data Atom approaches that size, the Transaction Engine creates a new Data Atom to hold new versions of rows. A Data Atom that contains blob data holds all of one or more blobs. Blobs are not split across Data Atoms.

The **Index Atom** contains some or all of a particular index. Indexes in NuoDB terminate with a key value and a row number. Indexed access consists of walking an index looking for a particular key value or range, noting the row numbers that match, then accessing the rows by number. The row number decomposes into the number of a particular Records Atom for the table and a row within that atom. Like the Data Atom, Index Atoms have a target size and split when they exceed that size.
NuoDB represents a table and its contents, excluding blobs, in a hierarchy of three atom types. The **Table Atom** contains the basic metadata for the table and the mapping from row numbers to Records Atoms. **Records Atoms** manage row versions, tracking the identifier of the transaction that created them and the format of the table in which they were created plus the identity of the Data atom that contains the row data for that version and how to find the data in the atom. **Data Atoms** hold arrays of encoded bytes that are the contents of the table.

**Atom Chairman**

Although searches and inserts can operate autonomously, many database operations require some coordination. NuoDB pushes that coordination to the lowest possible level to reduce its impact on system performance. One copy of each atom is designated as the Chairman for that atom. The Chairman resolves disputes and distributes unique identifiers for all the other copies of that atom.

For example, Transaction Manager Atoms assign identifiers to transactions they start. Having two Transaction Managers on different Transaction Engines assign the same identifier to different transactions would cause chaos, so the Chairman of the Transaction Manager Atoms assigns blocks of transaction identifiers to Transaction Manager Atoms on other machines. The Chairman of the Transaction Manager atoms receives a replication message whenever a Transaction Manager elsewhere starts a transaction. It monitors the number of remaining identifiers for each copy of itself, and sends blocks of new identifiers if one begins to run short. The first instance of each atom is the Chairman, of course. Part of the information every atom carries is the identity of its Chairman, and a list of successors should that copy of the atom be dropped to reduce the memory load of its controlling Transaction Engine. Chairmanship is also passed along if a Transaction Engine crashes or becomes unavailable.

For many atoms, the only function of the Chairman is assigning identifiers for the atoms that its peers may create, guaranteeing that each new atom has a unique identifier. The Chairman of a particular Schema Atom hands out identifiers for Table and Sequence Atoms created in the schema. The Chairman of a Sequence Atom sends out blocks of unique values to be used as sequence values.

The Chairman of a particular Records Atom has responsibility for coordinating updates and deletions of rows in its range. Each Records Atom tracks the state of two thousand rows in a table. When a transaction on one Transaction Engine wants to update or delete a row, it makes the change locally, adding new data to a Data Atom in the case of an update, and adding information about a new record version in its Records Atom. Before either of those atoms sends replication messages to its peers, the Records Atom sends a priority (aka “forced”) message to its Chairman saying “I want to change row X, the most recent version I saw was created by transaction Y which I consider committed. I am transaction Z.” If the Chairman sees that its most recent version of the row was also created by transaction Y, it responds that the change is allowed and notes that its own most recent version is now the one created by Z. If the Chairman sees that the most recent version was created by a different transaction, it tells the would-be updater that it must wait for the transaction that created the newer version to finish. The transaction that tried the update checks in with its local Transaction Manager Atom, asking it about the state of the transaction it is waiting for. What happens next depends on the isolation mode. This is what happens in the default, snapshot isolation mode. If the earlier transaction commits, the update fails. If the earlier transaction rolls back, the update succeeds.

For example, one Records Atom controls rows 2001 to 3000 in the HorsetailFireworks table in the VideoCatalog database of Acme Fireworks. Copies of that Records Atom exist on Transaction Engines on several virtual...

---

14 Detailed table metadata is kept in the System Schema and is managed by the SQL layer. Triggers and constraints are handled by the SQL layer. The Table Atom tracks the names and order of columns and their data types.
hosts on Amazon and in the company’s own datacenter. The Chairman of that Records Atom is on DataCenter3. Generally the video catalog is very stable, which makes it a good candidate to distribute on a wide network, but two events can change it.

1. Once a week, the Viewer application adds the number of views it has served out for each video.
2. When a particular fireworks becomes unobtainable, the Maintainer application removes rows for videos of that fireworks.

The two applications connect to different Transaction Engines. Two events occur simultaneously.

1. The Viewer application has its weekly updates and wants to add twenty views to the row in HorseTailFireWorks that describes a video of a purple rotating HorseTail shot in Berlin January first of 2008 - row 2020. Its job is to update row 2020.
2. The Maintainer notices a change from the suppliers of purple rotating HorseTail fireworks, saying that the Chinese government ruled them too dangerous to manufacture, so they are no longer available. The Maintainer’s job is to delete all those rows describing videos of purple rotating HorseTail fireworks, including row 2020. Both applications want to change the same row. Neither application is connected to the Transaction Engine that holds the Chairman of the Records Atom that describes row 2020.

The Maintainer’s Records Atom asks the Chairman of the Record State Atom for permission to delete the row, including the information that the most recent version of row 2020 it sees was created by transaction 311111 and that it is transaction 355555.

The Chairman of the Records Atom responds, “Fine 355555, go ahead and delete the row,” and makes a note that the newest version of row 2020 was created by 355555.

Simultaneously, the Viewer’s Record State Atom sends the Chairman a message saying “I want to update row 2020, the most recent version I see is 311111 and I am transaction 377777.”

The Chairman checks, notices that it has a newer version, and responds “No, 377777. You must wait for transaction 355555.”

When the Maintainer finishes deleting all the purple rotating HorseTail fireworks videos, it commits transaction 355555. Its Transaction Manager Atom replicates that change to other Transaction Manager Atoms.

When the Viewer’s Transaction Manager Atom sees the commit of transaction 355555, it tells transaction 377777, “Sorry, your update failed.” Since the Viewer doesn’t care very much about updating any particular statistic, it shrugs its virtual shoulders and continues.

Unique constraints are enforced by the chairman of an Index Atom in much the same manner.

---

15 NuoDB, like other MVCC databases, deletes a row by creating a new version without data that contains a flag saying this record was deleted. The “deleted stub” and some older versions of the row remain in the database until the end of the last transaction that could see the any previous version of the row. Then row garbage collection will remove them all.
of a purple, rotating Horsetail shot in Kuala Lumpur on January 1, 1999. That row has a record number of 4040. The Viewer’s transaction 377777 is waiting for the Maintainer’s transaction 355555 because of the conflict on record 2020. The Viewer has already updated the statistics in row 4040, but that information has not yet been propagated to the Maintainer’s Transaction Engine.

The Maintainer tries to delete the row. Again, the Chairman of the Records Atom - a different atom this time, because each Records Atom tracks only two thousand rows – checks whether there is a version of the row newer than the one the Maintainer’s transaction sees. Yes, there is.

The Transaction Manager Atom responsible for the Viewer application knows that its transaction 377777 is waiting for 355555. The Transaction Manager Atom responsible for the Maintainer knows that its transaction 355555 is waiting for 377777. All progress stops, until the two transaction Manager Atoms exchange information and the deadlock is found.

**Transaction 377777 is waiting for 355555**

**Distributed Deadlock Detection**

Deadlock occurs when two transactions wait for each other. Neither can commit, so both wait forever; as do all other transactions that try to change rows they have changed.

In the Acme Fireworks example, the database includes another record in HorseTailFireworks describing a video
“My transaction X is waiting for your transaction Y.” If that Transaction Manager Atom knows that its transaction Y is waiting for transaction Z on some other Transaction Engine, it sends a message to the Transaction Managers of X and Z, saying “My transaction Y is waiting for transaction Z”. Each of the Transaction Managers checks for a “cycle in the dependency graph”, or in English, a closed loop in the list of transactions waiting for each other. Even if Z is waiting for A, and A is waiting for B, and so on through the whole alphabet, if there is a loop, the Transaction Managers always find it.

The Transaction Manager Atom managing the transaction with the highest transaction identifier tells its Transaction Engine to undo the last SQL statement by that transaction and send it a deadlock error. The application can choose to continue, commit, or roll back, but the particular statement it tried cannot succeed unless the application rolls back the transaction and retries the whole operation. Well-behaved applications roll back their transactions after a deadlock error.

In the two transaction case of Acme Fireworks, the two Transaction Manager Atoms quickly realize that transaction 355555 is waiting for 377777 and transaction 377777 is waiting for 355555. The Viewer’s transaction 377777 gets a deadlock error on its attempt to update the statistics for row 2020, and rolls back its transaction. The Maintainer continues deleting purple rotating Horsetails. With any luck, its transaction will have completed before the Viewer is done with its error handling and tries again to update the statistics rows the Maintainer is deleting.

**Atoms and the Storage Manager**

As described earlier, the Storage Managers manage the archival copy of the database. What they store are key/value pairs. The key is the identifier of the atom. The value is the serialized content of that atom. Storage Managers keep in memory at least the stub of every atom present on any Transaction Engine. When an atom is no longer in any Transaction Engine’s memory and has been written to disk in its final state, the Storage Manager can drop it from its memory and recover it later by identifier. All replication messages go to their peers on the Storage Managers. When a Storage Manager writes an atom to disk, it marks that copy of the atom as clean. When a replication message arrives for the atom, the Storage Manager marks it as dirty and puts it on a queue to be written. A busy atom may get many replication messages while it is in the write queue, but eventually it will be written, marked clean, then dirtied again and put back in the write queue. The Storage Manager uses buffered writes for atoms.

---

16 in this context, “serialized” means that the content of the atom is processed to eliminate any address references and converted to a machine-independent format.
To improve durability in the case of a total system failure, the Storage Manager also maintains a journal of the incoming replication messages which is forced to disk with direct I/O at every commit. Journal files are truncated periodically to control their size.

With one exception, the Storage Manager does not change atoms. That exception occurs when a Transaction Engine fails with active transactions. One Storage Manager becomes the recovery agent and makes whatever changes are necessary to undo actions taken by the failed transactions. This duty falls to a Storage Manager because it has copies of all atoms.

**Atom Summary**

All persistent information in a NuoDB database is kept in Atoms. Every atom has a type, of which it is a specific instance, and most have copies scattered around the set of machines serving the database. When the content of an atom is modified, the atom sends replication messages to its peers. The peers apply those changes to themselves. Replication messages from any particular Transaction Engine are sent in the order they occur.

**Emergent Architecture**

NuoDB is an ACID transactional database that scales elastically across an arbitrary number of computers to handle varying levels of load. It uses demand distribution and partial replication to move data needed by a transaction to that transaction. Under a traditional SQL interface, it operates on a system of distributed objects called ‘atoms’ that replicate their change to their peers across the network. NuoDB has no central control. The individual atoms behave cooperatively, using replaceable Chairmen where coordination is required. The architecture is emergent because it consists of a large number of autonomous objects that together form a powerful, consistent, and elegant relational database management system.

**Summary**

NuoDB is different from any other relational database in many ways but there are some specific differentiators that are the most beneficial to developers:

- How NuoDB scales: NuoDB scales elastically to improve transactions per second performance and handle both massive concurrency and data volume. Out and in; by simply adding or removing nodes. But, there’s more to NuoDB scaling than that.

NuoDB is equally able to scale a single huge operational database or many separate databases – e.g., a series of databases hosted in Amazon EC2 or other cloud infrastructure. All with ease and all on a moment’s notice.

- NuoDB’s design values simplicity: Any software solution hypes “ease of use” but NuoDB delivers usability in a meaningful way. You can stage your data in ways you never could with a relational database before. Plus, there are many things you’ve had to do that aren’t necessary any longer. Some examples:
  - Install NuoDB with just six clicks and you’re up and running a relational database
  - Built-in redundancy eliminates the need for implementing complex, fragile replication schemes

- NuoDB’s emergent architecture eliminates the need for technical acrobatics to achieve basic enterprise scalability: sharding, clustering, performance tuning, replication or other kinds of 20th century database tricks

- Dynamic schemas: don’t like the first schema you created? Oh well. Just change it. None of the rest of your data will be affected. At all. Manage and evolve schemas as you never could before: simple to build and change, powerful, easy

- Multi-tenancy: Unlike traditional 20th century databases NuoDB supports the concept of multi-tenancy out of the box. Multi-tenancy means that
a single host machine within a NuoDB Domain is capable of hosting multiple, distinct databases. This approach solves many problems. First, application developers can provision a complete database for an app on shared resources, without having to provision a new DBMS instance. Second, each application administrator has full control over the resources, security and performance of their app. NuoDB allows them to benefit from one of the cloud’s most appealing aspects – sharing resources to cut expense.

NuoDB didn’t give up the core capabilities (like SQL and ACID guarantees) that you need to grow your web app... rather, NuoDB eliminates the hassle and overhead of old generation DBMS systems.

To meet the needs of independent developers, small firms and global enterprises focused on delivering traditional database apps or Software-as-a-Service products, NuoDB is available in two distributions – the Developer and Pro Editions.

The NuoDB Developer Edition lets users develop NuoDB apps at full throttle without restrictions on data size or number of hosts. Developers can push their applications up to (and beyond) real-world scenarios at no charge. Design, develop and test applications on the Developer Edition to get a true understanding of the resources needed to achieve necessary transactional throughput and data size for pre-production applications. When your app is ready for primetime, commercial deployment, move to the Pro Edition.

Use NuoDB Pro Edition for free until your apps need to scale. Then go faster, scale at a moment’s notice, or handle big data by upgrading your Pro Edition. You will have additional hosts and you can add as much storage as you need. NuoDB allows you to scale out in production like no other database solution can. Its unique, patented architecture gives you the built-in elasticity, multi-tenancy and geo-redundancy capabilities to take your application global in a hurry. The Pro Edition can handle millions of transactions per second, a global village of concurrent users and Big Data... all at a moment’s notice; all on commodity hardware.

You can download and install NuoDB with about 6-clicks. It’s that easy to launch the only 21st century, elastically scalable database around. Welcome to the New Age.

Download now at www.nuodb.com/download