NUODB®

WHITE PAPER

What Is A Distributed Database? And Why Do You Need One?

Website:
www.nuodb.com

Email:
sales@nuodb.com

Phone:
+1 (617) 500-0001

World Headquarters

150 Cambridgepark Drive
Cambridge, MA 02140
United States

INTRODUCTION

The basic premise of this document is simple: to explain why *distributed transactional databases are the Holy Grail of database management systems (DBMS).*

The promise of these systems is to provide on-demand capacity, continuous availability and geographically distributed operations. However, most of them require substantial trade-offs in terms of overall effort, cost, time to deployment and ongoing administration. Despite those trade-offs, these offerings have dominated the industry for decades, forcing compromises from start to finish – from initial application development through ongoing maintenance and administration.

For example many IT technologists have given up on the highly desirable characteristic of transactional consistency in favor of distributed operation. That is a trade-off that may be attractive if you can't find a way to scale-out transactions, but it is a drastic choice that moves a lot of complexity and cost up the application stack.

This Holy Grail discussion is specifically about distributed transactional databases. If such a thing can be built without compromise, no one would want any other distributed data store.

Is there a way to build one of these things?

First, let's lay out the three categories of designs that have been used historically

THE THREE TRADITIONAL ARCHITECTURES AND NUODB

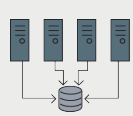
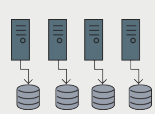
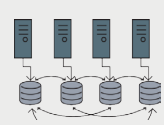
	Shared-Disk	Shared-Nothing/Sharded	Synchronous Replication	Durable Distributed Cache
Key Idea	Sharing a file system	Independent databases for disjoint subsets of data	Committing data transactionally to multiple locations before returning	Replicating data in memory on-demand
Topology				
Example	Oracle RAC, DB2 Pure Scale	Mongo DB, Volt DB, and most other NoSQL and NewSQL solutions	Google F1	

Figure 1: The Three Traditional Architectures and NuoDB

Shared-Disk Databases

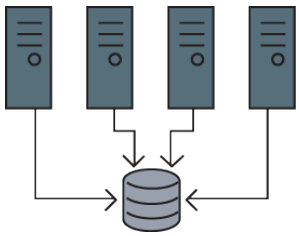


Figure 2: Shared-Disk Database Architecture

The idea of shared-disk designs is fairly self-explanatory. You have several machines in a cluster all loading and storing pages of data from the same shared-disk subsystem. A good example of a shared-disk system is **Oracle Real Application Clusters (RAC)**. The database system tracks updated pages and writes them to disk, invalidates "stale" pages, and implements some kind of a network lock manager that coordinates concurrency conflicts on a distributed basis. In this scenario, you need to be very aware of the nature of your workload.

> "Traditionally if you want a transactional database to go faster you need a bigger machine. Period. Solving that problem is the Holy Grail of the database world."
>
> Barry Morris,
> CEO AND CO-FOUNDER,
> **NUODB**

Workloads that require minimal coordination can scale fairly well on shared-disk systems. In the extreme case, consider an all-read workload. This is likely to be I/O-bound before it is limited by inter-machine coordination delays. The I/Obound issue can itself become a problem, because you don't increase I/O when you add more machines. If your database performance is limited by disk I/O and not by computational bandwidth, then there is no point adding more machines.

But coordination is often required between machines, and in these cases shareddisk systems are frequently limited by lock-based coordination protocols. The core DBMS design is typically built around the idea of efficiently managing tightly coupled memory and disk pages, and the introduction of synchronous distributed locking to guarantee page ownership introduces both latency problems and thrashing problems that can easily bring these systems to their knees. Careful allocation of transactions to machines to maximize affinity can ameliorate these issues to some extent, but this increases the fragility of the systems and substantially adds to long-term DBA expenses.

Shared-disk systems are at their best for workloads that are highly read dominated or in which the data can effectively be partitioned by machine.

Shared-Nothing Databases

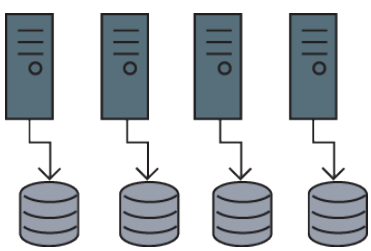


Figure 3: Shared-Nothing Database Architecture

These are both arguments in favor of shared-nothing designs. Shared-nothing is really just a cute way of saying "partitioning". In other words, you essentially run multiple databases, carefully arranging that part of the dataset residing in each partition. To take a trivial example of a database containing

all US citizens, you might put everyone with last names starting with A-L in one partition and everyone else in the other partition. Or you could put all male citizens in one and all female citizens in the other. For each incoming transaction, you detect which data it needs to access and then send it to the relevant partition.

You might ask why, if you are essentially just running multiple databases with the data split between them, is it a DBMS issue at all? Surely the application can just do that itself using any old database system? The answer is that of course you can do that, and it is called “Sharding”. Big Internet applications such as Facebook do this, and in some cases they run many thousands of shards. Shared-nothing distributed databases try to do this transparently for you, in particular striving to help you with the really hard problems of how to optimally partition the data, how to perform transactions efficiently when they touch data in multiple partitions, how to deliver transactional semantics across partitions, how to rebalance the partitions as the database grows, and so on.

Once again, partitioned stores can be very effective for certain kinds of workloads but you have to be very careful to arrange your queries and data layout to make sure it’s optimized. For some cases there is no good way to partition the data. In all cases there is a limit to how many partitions you can profitably create, which means there’s a limit to the degree that the system can scale out. If you are running a workload that has any appreciable proportion of cross-partition transactions, then you will most likely need specialized lowlatency interconnects between your servers. And the partitioning model is not very dynamic, so it does not address the desire for on-demand capacity management.

One last comment on partitioned stores: They are not all disk-based. In-memory systems typically also use partitioning in order to exploit multiple database servers in parallel.

Synchronous Commit (Replication) Databases

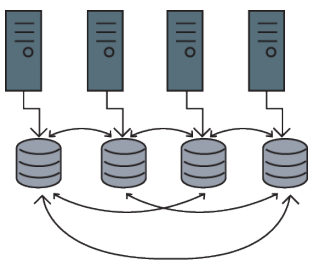


Figure 4: Synchronous Replication Database Architecture

There is a third traditional way of building distributed databases. You could call it the obvious way, the brute force way or, more technically, synchronous commit (Google calls it “Synchronous Replication”).

In a transactional DBMS, an application makes changes to the data before committing or rolling back the transaction.

If the transaction is successfully committed, then the state of the database has been (atomically) updated with the full set of committed changes. If the

commit fails, then the canonical state of the database is unchanged. And if the application performs a rollback, then the uncommitted changes are discarded. The synchronous commit approach as been adopted in various forms over the years, not least in the form of **two-phase commit (2PC)** protocols.

The idea of synchronous commit is simply that when an application makes a request for the DBMS to commit a transaction, the DBMS performs the commit in multiple locations before returning success. The best modern example of synchronous commit is Google F1. F1 is the database system that Google uses to support Google AdWords, and which will support many more Google applications in due course. Self-evidently it works for Google, and for a very demanding application. Google specifically notes that they could not address the AdWords challenge using NoSQL technologies or MySQL approaches.

Although Google has made it work, the disadvantage of synchronous commit should be self-evident. In the words of the **Google F1 White Paper**: "synchronous replication implies higher commit latency, but we mitigate that latency by using a hierarchical schema model with structured data types and through smart application design." Also, F1 relies on state-of-the-art Google networking to reduce inter-node latency, and the existence of atomic clocks on every participating machine in order to support a common notion of true time. The obvious challenge with synchronous commit is latency (in other words delays).

There are some applications for which throughput is more important than transaction latency but for most modern applications the reverse is true. A less obvious challenge relating to synchronous commit is managing error and failure conditions. Designing distributed synchronous commit protocols is relatively straightforward if you can assume 100% reliability. It is recovery detection and recovery procedures that are the hard part of the problem, and the designer is generally left with a trade-off between maintaining transactional guarantees for a large number of recovery scenarios, avoiding significant performance degradation, and limiting implementation complexity.

In consequence of the latency issues and failure management challenges the synchronous commit approach has had limited success historically. To summarize, on one hand it's clear that a truly distributed DBMS with ACID transactional guarantees could address the key pain points in modern database management, notably on-demand capacity, continuous availability and geodistributed operation. On the other hand, the three traditional designs offer much less than the promised advantages, and involve costs, complexities and/or functional limitations that curb their usefulness and general applicability.

NEW DDC ARCHITECTURE OFFERS COMPREHENSIVE SOLUTION

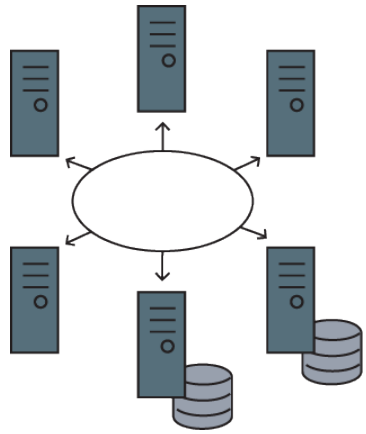


Figure 5: Durable Distributed Cache

By stepping back and rethinking database design from the ground up, **Jim Starkey**, NuoDB's technical founder, has come up with an innovative solution that makes very different trade-offs. It's an entirely new design approach that called Durable Distributed Cache (DDC). The net effect is a system that scales-out/ in dynamically on commodity machines and VMs, has no single point of failure, and delivers full ACID transactional semantics. Let's take a look at Starkey's thought process and the DDC architecture.

Memory-Centric vs. Storage-Centric

The first insight that Starkey had was that all general-purpose relational databases to date have been architected around a storage-centric assumption, and that this is a fundamental problem when it comes to scaling out. In effect, database systems have been fancy file systems that arrange for concurrent read/ write access to disk-based files such that users do not trample on each other. The DDC architecture inverts that idea, imagining the database as a set of in-memory container objects that can overflow to disk if necessary and can be retained in backing stores for durability purposes. Memory-centric vs. storage-centric may sound like splitting hairs, but it turns out that it's really significant. The reasons are best described by example.

Suppose you have a single, logical DDC database running on 50 servers (which is absolutely feasible to do with an ACID transactional DDC-based database). And suppose that at some moment Server 23 needs Object #17. In this case Server 23 might determine that Object #17 is instantiated in memory on seven other servers. It simply requests the object from the most responsive server – note that as the object was in memory the operation involved no disk IO – it was a remote memory fetch, which is an order of magnitude faster than going to disk.

You might ask about the case in which Object #17 does not exist in memory elsewhere. In the DDC architecture this is handled by certain servers "faking" they have all the objects in memory. In practice of course they are maintaining backing stores on disk, SSD or whatever they choose (in the NuoDB implementation they can use arbitrary Key/Value stores such as Amazon S3 or Hadoop HDFS). As it relates to supplying objects, these "backing store servers" behave exactly like the other servers except that they can't guarantee the same response times.

So all servers in the DDC architecture can request objects and supply objects. They are peers in that sense (and in all other senses). Some servers have a subset of the objects at any given time, and can therefore only supply a subset of the database to other servers. Other servers have all the objects and can supply any of them, but will be slower to supply objects that are not resident in memory.

Let's call the servers with a subset of the objects Transaction Engines (TEs), and the other servers Storage Managers (SMs). TEs are pure in memory servers that do not need use disks. They are autonomous and can unilaterally load and eject objects from memory according to their needs. Unlike TEs, SMs can't just drop objects on the floor when they are finished with them; instead they must ensure that they are safely placed in durable storage.

For readers familiar with caching architectures you might have already recognized that these TEs are in effect a distributed DRAM cache, and the SMs are specialized TEs that ensure durability. Hence the name Durable Distributed Cache.

Resilience to Failure

It turns out that any object state that's present on a TE is either already committed to the disk (i.e. safe on one or more SMs), or part of an uncommitted transaction that will simply fail at application level if the object goes away. This means that the database has the interesting property of being resilient to the loss of TEs. You can shut a TE down or just unplug it and the system does not lose data. It will lose throughput capacity of course, and any partial transactions on the TE will be reported to the application as failed transactions. But transactional applications are designed to handle transaction failure. If you reissue the transaction at the application level, it will be assigned to a different TE and will proceed to completion. So, what this means is the DDC architecture is resilient to the loss of TEs.

What about SMs? Recall that you can have as many SMs as you like. They are effectively just TEs that secretly stash away the objects in some durable store. And unless you configure it not to, each SM might as well store all the objects. Disks are cheap, which means that you have as many redundant copies of the whole database as you want. In consequence the DDC architecture is not only resilient to the loss of TEs, but also to the loss of SMs.

In fact, as long as you have at least one TE and one SM running, you still have a running database. Resilience to failure is one of the longstanding but unfulfilled promises of distributed transactional databases. The DDC architecture addresses this directly.

Elastic Scale-Out and Scale-In

What happens if I add a server to a DDC database? Think of the TE layer as a cache. If the new TE is given work to do, it will start asking for objects and doing the assigned work. It will also start serving objects to other TEs that need them. In fact the new TE is a true peer of the other TEs. Furthermore, if you were to shut down all of the other TEs, the database would still be running, and the new TE would be the only server doing transactional work.

> **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,
> CHIEF TECHNICAL OFFICER,
> **DROPSHIP COMMERCE**

Storage Manager's, being specialized Transaction Engine's, can also be added and shut down dynamically. If you add an "empty" (or stale) SM to a running database, it will cycle through the list of objects and load them into its durable store, fetching them from the most responsive place as is usual. Once it has all the objects it will raise its hand and take part as a peer to the other SMs. And, just as with the new TE described above, you can delete all other SMs once you have added the new SM. The system will keep running without missing a beat or losing any data.

So the bottom line is that the DDC architecture enables capacity on demand. You can elastically scale out the numbers of TEs and SMs and scale them back in again according to workload requirements. Capacity on demand is the second promise of distributed databases that is finally a reality with the DDC architecture.

Geo-Distribution

The astute reader will no doubt be wondering about the hardest part of this distributed database problem, namely that we are talking about distributed transactional databases. Transactions, specifically ACID transactions, are an enormously simplifying abstraction that allows application programmers to build their applications with very clean, high-level and well-defined data guarantees. If I store my data in an ACID transactional database, I know that it will isolate my program from other programs, maintain data consistency, avoid partial failure of state changes and guarantee that stored data will still be there at a later date, irrespective of external factors. Application programs are vastly simpler when they can trust an ACID compliant database to look after their data, whatever the weather.

The DDC architecture adopts a model of append-only updates. Traditionally an update to a record in a database overwrites that record, and a deletion of a record removes the record. That may sound obvious, but there is another way, invented by Jim Starkey about 25 years ago. The idea is to create and maintain versions of everything. In this model you never do a destructive update or destructive delete. You only ever create new versions of records, and in the case of a delete, the new version is a record version that notes the record is no longer extant. This model is called **MVCC** (multi-version concurrency control), and it has a number of well-known benefits even in

scale-up databases. MVCC has even greater benefits in distributed database architectures, including DDC.

There isn't space here to cover MVCC in further detail but it is worth noting that one of the things it does is allow a DBMS to manage read/write concurrency without the use of traditional locks. For example, readers don't block writers and writers don't block readers. It also has some exotic features, including the ability to maintain a full history of the entire database. But as it relates to DDC and the distributed transactional database challenge, there is something very neat about MVCC. DDC leverages a distributed variety of MVCC in concert with DDC's distributed object semantics that allows almost all the inter-server communications to be asynchronous.

The implications of DDC being asynchronous are far-reaching. In general it allows much higher utilization of system resources (cores, networks, disks, etc.) than synchronous models can. But specifically it allows the system to be fairly insensitive to network latencies, and to the location of the servers relative to each other. Or to put it a different way, it means you can start up your next TE or SM in a remote datacenter and connect it to the running database. Or you can start up half of the database servers in your datacenter and the other half on a public cloud.

Modern applications are distributed. Users of a particular website are usually spread across the globe. Mobile applications are geo-distributed by nature. Internet of Things (IoT) applications are connecting gazillions of consumer devices that could be anywhere at any time. None of these applications are well served by a single big database server in a single location, or even a cluster of smaller database servers in a single location. What they need is a single database running on a group of database servers in multiple datacenters (or cloud regions). That can give them higher performance, datacenter failover and the potential to manage issues of data privacy and sovereignty.

The third historical promise of distributed transactional database systems is geo-distribution. Along with the other major promises (resilience to failure and elastic scalability), geo-distribution has heretofore been an unattainable dream. The DDC architecture with its memory-centric distributed object model and its asynchronous inter-server protocols finally delivers on this capability.

DDC: THE HOLY GRAIL OF DISTRIBUTED DATABASES

Hopefully this has provided you with a quick overview of distributed database designs, with some high level commentary on the advantages and disadvantages of the various approaches. There has been great historical success with shared-disk, shared-nothing and synchronous commit models. The advanced technology companies are delivering some of the most scalable

systems in the world using these distributed database technologies. But to date distributed databases have never really delivered anything close their full promise. They've also been inaccessible to people and organizations that lack the development and financial resources of Google or Facebook.

With the advent of Durable Distributed Cache architectures, it is now possible for any organization to build global systems with transactional semantics, ondemand capacity and the ability to run for 10 years without missing a beat. The big promises of distributed transactional databases are elastic scalability, resilience to failure and geo-distribution. It's very exciting that due to Jim Starkey's Durable Distributed Cache those capabilities are finally being delivered to the industry.

NuoDB's elastic SQL database for cloud applications helps customers get applications to market faster and reduce their total cost of ownership. Software vendors and ecommerce companies rely on NuoDB to obtain the combination of scale-out simplicity, elasticity, and continuous availability that cloud applications require, with the transactional consistency and durability that databases of record demand.

As a result, customers can capitalize on modern technologies such as cloud computing and containerization to ensure their applications are ready for today's evolving expectations, as well as any future requirements.

NuoDB is headquartered in Cambridge, MA, USA, with offices in Dublin and Belfast. For more information, visit **nuodb.com**.