



# Shared-Disk vs. Shared-Nothing

## Comparing Architectures for Clustered Databases

by  
Mike Hogan, CEO  
ScaleDB Inc.

## Overview

The following chart provides a framework for evaluating which database clustering architecture best fits your needs. It is, however, specific to the technical and business requirements of your application. If, for example, your data is partitioned such that there is no data shipping—very rare in enterprise environments—then shared-nothing can deliver superior performance. On the other hand, if you want to implement distributed 2-phase commit, then shared-disk will deliver superior performance. The remainder of this white paper delves into each issue in detail.

Issue	Shared Disk (SD)	Shared Nothing (SN)
<b>Set-up &amp; Maintenance</b>		
Initial Set-up Effort	🟢 No extra effort	🔴 Partitioning & routing tables
Ongoing Maintenance	🟢 No extra effort	🟡 Ongoing re-partitioning
<b>Performance &amp; Overhead</b>		
Inter-Nodal Messaging	🔴 Modest messaging overhead	🟢 No inter-nodal messaging
Function/Data-Shipping	🟢 None	🔴 Can be very high, tuning issue
Benchmark Performance	🟡 Messaging overhead makes it harder to tune for benchmarks	🟢 Ideal for snapshot/benchmark tuning
Evolving Performance	🟢 Adapts to evolving requirements via load balancing	🟡 Re-partitioning may be required to handle evolving usage
Temporal Performance	🟢 Load balancing handles temporal shifts in usage	🔴 Fixed partitioning cannot adapt to temporal changes
2-Phase Commit	🟢 2-phase commit is handled quickly by a single node, but may not be needed in shared-disk	🔴 2-phase commit distributed across multiple nodes is very slow
Data Access Speed	🔴 SAN/NAS data access latency	🟢 Local disk, bus speed access
<b>Features &amp; Capabilities</b>		
Load Balancing	🟢 Dynamic load balancing	🔴 Fixed load balancing based upon the partitioning scheme
High-Availability	🟢 Master-master failover	🔴 More planned and unplanned downtime
Data Consistency	🟢 Single copy of the data	🟡 If slaves are used for read access, data consistency problems
Scalability	🟡 Messaging overhead limits total number of nodes	🟡 Depends on partitioning, data shipping can kill scalability
Scale-in to fully utilize your server	🟢 Shared-disk architecture supports Scale-in	🔴 Data partitioning is incompatible with Scale-in
Parallel Processing Across Nodes	🟢 Processes can be parallelized without additional effort	🔴 Partitioning trade-offs, parallel vs. unified view, problematic
Cloud Computing: Plug-and-Cluster for DaaS	🟢 Supports dynamic nodal scalability	🔴 Requires middleware, results in non-optimal partitioning
Total Cost of Ownership (TCO)	🟢 Software can be more expensive, but set-up and maintenance costs are lower	🔴 Higher ongoing/maintenance costs for partitioning, tuning, slave replication, etc.

## Other Complementary White Paper Resources

[A Primer on Database Cluster Architectures](#): A quick overview of database architectures.

[Cloud Databases](#): Summarizes the database requirements for cloud databases and compares the suitability of different database architectures to cloud computing.

## Set-up & Maintenance

Database clustering architectures are in many ways analogous to automotive transmissions. Shared-disk is like the automatic transmission, because it automates much of the complexity of cluster set-up and maintenance, thus lowering total cost of ownership (TCO). Shared-nothing is analogous to the manual transmission because it provides more granular control in exchange for increased manual effort on the part of the owner.

### Initial Set-up

Shared-Nothing:

The following are the steps for initial set-up of a shared-nothing database:

1. Design a partitioning<sup>1</sup> scheme to minimize or eliminate inter-nodal messaging (see “Inter-nodal Messaging” section below)
2. Split your data among the servers in the cluster
3. Assemble the routing tables (middleware or application code) to route database requests to the appropriate server
4. Repeat the entire process with each re-partition (see “Maintenance” section below)

Shared-Disk:

Shared-disk does not rely on data partitioning, every node of the cluster has full access to read and write against the entirety of the data. Like the automatic transmission, you simply get in and go.

### Ongoing Maintenance

Shared-Nothing:

Over time, your application may mature, user patterns may evolve, data volume grows, and your database may also fall prey to data skew<sup>2</sup>. Shared-nothing databases respond to these things by relying more and more on function- or data-shipping. While your database partitioning scheme might have been perfect for a snapshot in time, it can quickly become sub-optimal, which results in degradation of database performance. The only way to improve performance is to re-partition your database. Keep in mind that partitioning is a practice in trade-offs, partitioning one way may be good for certain things, but another way is good for other things. There is no universal cure-all for partitioning. Needless to say, partitioning is also an expensive and time-consuming process.

Shared-Disk:

Because shared-disk systems do not rely on data partitioning, you do not have this problem or expense.

---

<sup>1</sup> Wikipedia, Partition (database) [http://en.wikipedia.org/wiki/Partition\\_\(database\)](http://en.wikipedia.org/wiki/Partition_(database))

<sup>2</sup> Handling data skew in parallel joins in shared-nothing systems, Yu Xu et al., <http://portal.acm.org/citation.cfm?id=1376720>

## Performance & Overhead

If your application dynamics—users, data size, application requirements, etc.—are static and the data can be partitioned in a manner that minimizes or eliminates function- or data-shipping then shared-nothing can be tuned to deliver superior performance. The problem is that most real-world enterprise applications do not demonstrate these static characteristics. If you have an application that is evolving or experiences temporal variation, shared-disk systems will generally deliver superior real-world performance.

### **Inter-Nodal Messaging**

Messaging refers to the nodal and cluster-wide status information that is shared with other nodes on the cluster. Messages include information about locking, buffering, node heartbeat and general status. There is a trade-off between messaging (used by shared-disk systems) and function- and data-shipping (used by shared-nothing systems). Messaging scales linearly with the number of nodes in a cluster, while function and data-shipping typically increase exponentially with the number of nodes in a cluster.

#### Shared-Nothing:

No inter-nodal messaging, no additional overhead.

#### Shared-Disk:

Shared-disk relies on inter-nodal messaging. These messages are small fixed-size messages. The volume of inter-nodal messaging in a shared-disk system increases linearly according to the number of nodes in the cluster.

### **Function/Data-Shipping**

When a shared-nothing database receives a request that spans two or more nodes on the cluster, e.g. with a join function, it relies on shipping either the function or the data between nodes to satisfy the request. This is generically referred to as data-shipping<sup>3</sup>. Ideally, you want to partition the database, such that there is no data-shipping. This ideal may be achievable in business intelligence (BI) applications, but not typically in online transaction processing (OLTP) applications, which involve more ad hoc usage patterns. Unlike inter-nodal messaging, data-shipping sends variable sized collections of data between nodes—typically much larger than inter-nodal messages. The more nodes you have in your cluster, the more frequently your queries will span multiple collections or nodes. As a result, data-shipping frequency increases exponentially with an increase in the number of nodes in the cluster. For these reasons, data shipping generally creates far more overhead than inter-nodal messaging.

#### Shared-Nothing:

Shared-nothing relies on function- or data-shipping to satisfy database requests (e.g. joins) that span multiple nodes. Depending on the partitioning scheme, data volume and the number of nodes, this traffic can quickly limit performance and nodal scalability.

#### Shared-Disk:

No function- or data-shipping, so there is no additional overhead.

---

<sup>3</sup> Scalable Performance through Cooperative Data Shipping, Sujata Banerjee, Panos K. Chrysanthis, and A. Deshpande, August 2001.

## Benchmark Performance

Shared-nothing DBMS will win all benchmarks. Benchmarks represent a static scenario for evaluating performance. As such, they are a perfect fit for a highly-tuned static database. You can tune the database for that specific benchmark scenario, to eliminate all data-shipping, so you get great performance. However, Michael Stonebraker, leading proponent for the shared-nothing architecture, admits in his paper *The case for shared-nothing*<sup>4</sup>, that “To ordinary mortals tuning is a black art.”

Shared-disk databases are very flexible for dealing with changing usage patterns. However, they are not hand-tuned for specific and static benchmarks. Shared-disk systems also suffer from the small but perceptible effect of inter-nodal messaging and data access times from shared storage.

So, shared-nothing wins all benchmarks, but benchmarks do not reflect reality. As 19th century German military strategist, Field Marshall Helmuth von Moltke said: “No battle plan survives contact with the enemy.” Your users will do things that you have not anticipated. Maybe you’ve heard a database architect lament: “My database design is perfect, it’s the users and application developers that are the problem.” Shared-nothing, with its fixed partitioning scheme, simply cannot dynamically adapt like shared-disk. Furthermore, in the real world, shared-nothing databases suffer from data-shipping, which is hand-tuned out, to win static benchmarks.

Comparing shared-nothing and shared-disk in benchmarks is analogous to comparing a dragster and a Porsche. The dragster, like the hand-tuned shared-nothing database, will beat the Porsche in a straight quarter mile race. However, the Porsche, like a shared-disk database, will easily beat the dragster on regular roads. If your selected benchmark is a quarter mile straightaway that tests all out speed, like Sysbench, a shared-nothing database will win. However, shared-disk will perform better in real world environments.

### Shared-Nothing:

Because they can be hand-tuned for static benchmarks, shared-nothing databases will win benchmark competitions, but benchmarks are not reflective of real-world scenarios.

### Shared-Disk:

Because of the small, but omnipresent inter-nodal messaging employed by shared-disk, it will lose benchmarks. However, in real-world situations, the shared-disk system can dynamically adjust to varying usage patterns enabling them to deliver superior performance over time.

## Evolving Performance

Usage patterns can change or evolve over time, creating havoc for databases that are partitioned and tuned on the basis of a fixed usage pattern. For example, users might begin discovering or utilizing a certain functionality of the application. This can result in a gradual shift in usage patterns. Another example is when you launch a new feature that enjoys rapid adoption, or if your application enjoys viral growth. All of these changes in usage patterns can cause shared-nothing databases to suffer from data skew, data-shipping and sub-optimal partitioning, all of which negatively impact performance. Whether these changes are gradual or rapid, the seminal question is “How well does your database handle evolving usage patterns?”

---

<sup>4</sup> The case for shared nothing, Michael Stonebraker, 1986. <http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf>

If the usage patterns of your database are static, or if you have a priori knowledge about usage patterns (e.g. business intelligence applications), you can address these issues very effectively within the framework of a shared-nothing database. However, if you have an evolving usage pattern or ad hoc evolution of usage patterns (e.g. online transaction processing applications), the shared-disk model will deliver superior real-world performance because of its ability to handle such variations through cluster-level dynamic load-balancing (see “Load Balancing” section below).

### Shared-Nothing:

Shared-nothing clusters use a static partitioning scheme that is optimized for a certain usage pattern. As a result, they cannot dynamically adjust to trends in usage patterns. The performance of a shared-nothing database is optimized for a snapshot in time; thereafter it is on a steadily declining performance trajectory.

To accommodate this, the database administrator (DBA) invests time monitoring the database performance in order to know when it is appropriate to re-partition the database to address these trends. This manual process is very expensive in terms of time and costs to monitor and then re-partition the database, not to mention the fact that re-partitioning involves scheduled downtime for the database.

### Shared-Disk:

Since shared-disk databases enable all nodes to feed from a single trough of data—avoiding the need to partition data—it does not suffer from steadily declining performance like its shared-nothing brethren. As usage patterns change, the load is seamlessly addressed by the collection of nodes in the cluster. In the examples above, more of the processing power of the nodes may be consumed by the newly popular capabilities, all without repartitioning. The only decision the DBA must make is whether or not to add another node.

## **Temporal Performance**

Databases must accommodate temporary oscillations in usage patterns, also known as temporal shifts. Let me provide an example to better describe the issue:

*Diagram 6: A Partitioned Human Resources Application*



In the scenario above, you have vertically partitioned your Human Resources database. During the mornings, user verification might be at peak utilization. Then around the 15<sup>th</sup> and 30<sup>th</sup> of the month, the payroll server is peaking. These shifts in usage are temporal shifts, they are not static. Yet the partitioning of shared-nothing databases is static, it doesn't adjust to accommodate these temporal changes.

The shared-disk architecture, on the other hand, allows all nodes of a cluster to access all of the data. So if the database above were running on a shared-disk database, you might have three of the four servers handling user verification in the morning. Then around the 15<sup>th</sup> and 30<sup>th</sup> of the month, you might have two of the four nodes handling payroll. This capability is referred to as cluster-level dynamic load-balancing (see “Load Balancing” section below).

### Shared-Nothing:

Shared-nothing databases rely on a static partitioning scheme, yet hotspots or bottlenecks shift over time. In effect, given the current situation, partitioning is often in a sub-optimal state, resulting in sub-optimal performance.

### Shared-Disk:

Because the shared-disk architecture enables any node on the cluster to satisfy any database request, it dynamically adjusts to accommodate temporal shifts in usage.

## **2-Phase Commit**

Two-phase commit (2PC) is a process where two or more changes are queued so that they are committed or written at the same time. For example, if an individual is wiring money from an account, you would want the outgoing wire to be orchestrated with debiting the account. 2PC orchestrates both transactions so that either (a) both happen; or (b) neither happens. The worst case, in the example above, is that the wire goes out and the account is not debited.

In order to make 2PC work, the commits must be orchestrated to happen at the same time. In a shared-disk system, this is no problem, because any node can access both tables and execute the 2PC itself. However, in a shared-nothing architecture, each node must lock its data for the 2PC and then it must wait for all of the other nodes involved to lock their data. This coordination ensures that the commits are executed in tandem and also that no other transactions access or modify the underlying data in the meantime. As a result, the locks are left in place, waiting for the slowest node. This can have a very serious impact on performance.

Let us use an example to illustrate the problem. If you have a 6-node shared-nothing cluster and your 2PC must run across three of those nodes, nodes number 1 and 2 lock data and then wait for node 3. However, in a shared-disk environment, a single node can orchestrate all of the locks and then commit all three changes by itself, thus tying up only a single node. If we use the analogy of the 6-lane highway, shared-nothing would require closing three lanes, one at a time, until things line up properly. Shared-disk, on the other hand, would mean closing just one lane. You can imagine the impact of each of these scenarios on the flow of traffic or data.

### Shared-Nothing:

Distributed 2PC locks the applicable data across all of the nodes involved, so all other requests for that data are forced to wait. Then it waits for the slowest node to lock its data. Then they all commit in tandem. In the DBA lexicon, this is a very expensive process, meaning it can dramatically reduce your database performance.

### Shared-Disk:

Any single node can lock and coordinate the changes itself, so there is no waiting for the slowest node. This results in much faster performance versus shared-nothing.

## **Data Access Speeds**

Shared-disk systems incur a small amount of latency in accessing data from a SAN (storage area network) or NAS (network attached storage), while shared-nothing databases access data from a local disk at faster bus speeds. Given high-speed interconnects such as Fiber Channel and gigabit ethernet, the latency difference between a local disk and a shared disk can be negligible. In fact, by leveraging the superior caching of many shared storage solutions, you may find performance

improving relative to a local disk. If you are using a shared-disk system for scale-in, data is transferred at bus speeds, meaning that in that case this point is moot.

#### Shared-Nothing:

Local data access is delivered at very fast bus speeds.

#### Shared-Disk:

(a) For scale-out clusters, the shared-disk interconnect speeds are generally slower than bus speeds, but the performance differential is typically minimal, and may be superior due to sophisticated caching.

(b) For scale-in clusters, where you have multiple copies of MySQL running on a single server, the point is moot, since they transfer data at bus speeds.

## Features & Capabilities

This section addresses some of the features or capabilities that are typically associated with enterprise database requirements. The underlying architecture determines whether these capabilities are native to the database or whether they are bolted on top of the database. If a certain capability is integral to the database architecture, it is provided invisibly, without requiring any additional work by the application developer or the DBA.

If the capability must be addressed at the application or middleware levels, or if it requires third-party software that sits between your application and your database, you are introducing an additional layer of complexity, work, licensing, and a separate maintenance and upgrade path. Furthermore, some of these bolt-on solutions are partial solutions that introduce trade-offs of their own. As a result, it is clearly preferable to use a database architecture that provides your required capabilities natively.

### **Load Balancing**

Wikipedia describes load balancing as “...a technique to spread work between two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, throughput, or response time.”<sup>5</sup> One of the primary purposes of clustering is load balancing. However, the important difference is how the work is shared among the nodes of the cluster.

In a shared-nothing database, the load is spread amongst servers or nodes on the basis of which server owns that data. Data ownership is fixed via static data partitioning. In other words, each node must be able to handle the peak load for its data by itself; there is no dynamic load balancing. There are partitioning schemes designed to spread the load more effectively, namely horizontal partitioning. Of course, there is a trade-off with horizontal partitioning if you want to run processes across multiple nodes. In any case, the dynamic aspect of the shared-disk provides inherent and dynamic load-balancing across the entire cluster.

When using a shared-nothing system, hotspots—where one or more nodes is overwhelmed with traffic—require one of these two options:

1. Re-partition to spread that load, again using a fixed model for splitting the data between the nodes. This may involve adding more servers (scale-out); or
2. Buy larger servers (scale-up) to handle the load.

---

<sup>5</sup> Wikipedia, Load Balancing (computing), [http://en.wikipedia.org/wiki/Load\\_balancing\\_\(computing\)](http://en.wikipedia.org/wiki/Load_balancing_(computing))

Using a shared-nothing database, your server costs increase, because you either need larger and more expensive servers, or you need more of them.

**Example:** Your database has a load factor of 2, and each server can handle a peak load of 1. Here is how you would configure each respective database:

- (a) Shared-nothing: 4 master servers with each operating at .5 and peaking at 1. Then you would assemble 4 slaves to handle fail-over. This gives you a total of 8 servers.
- (b) Shared-disk: 3 servers, each running at approximately .66, so that if one fails, the other two operate at 1 each until you restart the third server.

*Note:* slaves cannot assist in balancing the load for updates, but they can balance the load for reads. However, the replication of data to slaves typically involves a delay that can result in data consistency issues (see Data Consistency section).

#### Shared-Nothing:

Because it lacks dynamic load balancing, shared-nothing requires that each server accommodate the peak load for the data it owns. This causes shared-nothing implementations to invest more in servers using either a scale-up or a scale-out approach.

#### Shared-Disk:

Shared-disk enables any node to access the entire data set, so any node can service any database request. This results in more fluid load-balancing than with shared-nothing. This fluidity is the driving factor behind shared-disk's ability to smoothly accommodate temporal and evolutionary changes in usage patterns. It also enables you to run each server at a higher CPU utilization, because peak loads are spread across all of the servers in the cluster.

### **High-Availability**

If operational continuity is a key criterion for you, the DBMS architecture is a significant consideration. To better understand the issue we will address the two types of downtime: planned and unplanned.

#### Planned Downtime:

Shared-nothing databases must be shut down each time a new partitioning scheme is implemented. The process is as follows, with the red sections representing processes implemented while the database is shut down:

1. Develop and evaluate the new data partitioning scheme
2. Create the changes to the routing tables
3. Shut the database and application down
4. **Commit the changes**
5. **Move the physical data to the appropriate servers**
6. **Restart the application and database**
7. [Probably] Re-index the database to accommodate the changes
8. Test and repeat as necessary

Shared-disk databases don't use partitioning, so this process goes away. As a result, your planned downtime will decrease significantly. The questions you need to answer are: (a) how problematic is planned downtime and; (b) how frequently do you expect to re-partition your data?

### Unplanned Downtime:

Software conflicts, hardware failure, lost connections and other things can cause unplanned downtime at the same rate under both database architectures. Technically, since fewer servers are required with shared-disk, their unplanned downtime is slightly lower, but the difference is negligible. The primary question is “How well does each architecture handle system failure?”

The shared-nothing architecture relies on a collection of single points of failure—master nodes—which are typically supported by one or more slaves per master. These slaves are waiting for the master node to fail, at which point they must be promoted to master status. This promotion is typically a manual process that can take several minutes to hours, once the alert has been received. The newly promoted master must also apply any uncommitted transactions in its log. The application must also know how to reroute the database calls to this new master.

Shared-disk clusters respond to server failure by routing database requests to the next available node. There is no need to promote servers or alert the application or data layers to re-route database requests. The failure of a server, in a shared-disk database, doesn’t result in a fire drill. Instead the load is spread over the remaining servers in the cluster, while the DBA revives the troubled node at his leisure.

Our chief architect talks about an implementation at a major Japanese bank—based on his shared-disk architecture—that did not have *any* downtime in excess of a single minute during a four-year period. Based solely on the re-partitioning issue, not a single shared-nothing database, in a mission critical implementation, can claim this level of availability. High-availability is one of the strongest selling points for shared-disk DBMS.

### Shared-Nothing:

- (a) Requires planned downtime to implement re-partitioning
- (b) Failure of a master node results in downtime until a slave node is promoted

### Shared-Disk:

- (a) No partitioning, so planned downtime is reduced dramatically
- (b) Unplanned downtime is also greatly reduced. With a server failure, shared-disk dynamically routes around the failed server without shut-down.

## **Data Consistency**

Data consistency, very simply means that the information you receive is valid and accurate across your entire cluster. There are two ways databases can respond with inconsistent data: (1) a node reads the “dirty” data either in the middle of a change by another node, or as a result of a failure by a node; (2) as a result of time delays for data replication, a copy of the data is out of date, resulting in one node giving one answer and another giving a different answer at the same time. These two issues might sound arcane and trivial, but that depends upon your business requirements. If your application handles sharing web-based pictures of little Ryan batting, this probably isn’t a big concern. However, if your application serves information that is actionable and mission-critical, this could be a serious issue. Let me give you two examples:

### ***Example 1 (reading dirty data in the middle of a transaction):***

Your company is developing the wiring bundles for a new private jet. These bundles of wires connect the flight panel to the various control systems on the plane. Your bundle design is generated in the middle of a change by the wire bundle designer, where one end of the wire had been changed, but the other end of the wire has not yet been changed. Unfortunately, nobody

realizes the error. So the bundle is manufactured and placed in 1,000 private jets. One of these jets has a problem with their right engine in flight, it starts spitting out flames. No problem, it can fly on a single engine. The pilot flips the switch to kill the right engine and douse it with fire retardant. The only problem is the wire bundle sends that message to the left engine instead, because the design was based on dirty data extracted in mid-transaction. Now both engines are dead, and the plane crashes; clearly a sub-optimal outcome.

***Example 2 (time delay with data replication):***

A database uses a collection of slaves. These slaves maintain a copy of data that is replicated from the master node. However, this replication process can take as long as 10 minutes for changes to replicate completely through the slaves. Kendall has \$20M in her account in the Cayman Islands. She wires that \$20M to a Swiss account. A minute later, Kendall goes to the teller at a bank in the Cayman Islands to cash a check for \$15M. The cashier, reading an old copy of the data from a slave, sees that Kendall's account has \$20M, so he gives her \$15M in cash. Kendall just took the bank for \$15M.

Assuming the database is transactional, both shared-nothing and shared-disk architectures enforce data consistency...until you introduce copies of the data. Shared-nothing introduces copies of the data in the form of slaves. If data consistency is a priority, you can either: (a) configure slaves for no read access, which means they do not provide any load balancing for read access; or (b) use synchronous replication, which is similar to 2PC, in that the transaction is not committed until all copies are updated. There is a significant performance penalty to synchronous replication. Practically speaking, most slaves are configured for read access and they use asynchronous replication, which introduces potentially serious data consistency issues.

Shared-disk, on the other hand, alerts all nodes to transactions in process, so the other nodes simply wait until the update lock on that block is removed before reading. Some shared-disk systems can also be configured as read-preferred, so the updates are delayed until reads are completed. Furthermore, shared-disk uses a single networked copy of the data. Because there are no replicas of the data, there is no way to introduce outdated replicas or data consistency problems.

Many people don't realize the impact of data inconsistency until it is too late. Others may not have a need for strong consistency with the current version of their application. The problem is that application requirements might change, making data consistency mission-critical. Consider your situation today and in the future, and consider how inconsistent data can impact your business. If data consistency is mission-critical, either use a DBMS with a shared-disk architecture, or, if you use a shared-nothing architecture, do not configure your slaves for read-access.

**Shared-Nothing:**

You have three options: (1) live with inconsistent data; (2) configure your slave servers not to provide read access and reduce read performance; (3) configure your transactions to include synchronous replication inside the transaction, thus reducing overall performance.

**Shared-Disk:**

Data consistency is inherent because there is only one copy of the data.

**Scalability**

In a shared-nothing database, the level of function- and data-shipping is a significant determinant of the cluster's scalability. The primary factor that determines this level of inter-nodal interaction is how you partition the data. There are two ways to partition data, vertically and horizontally. You

can think of this as splitting by columns (vertically) and by rows (horizontally)<sup>6</sup>. One partitioning model that has gained popularity is sharding<sup>7</sup>. Sharding is a horizontal partitioning model that utilizes identical schemas on each node. A popular sharding model in the web 2.0 world is to shard based upon users (e.g. users 1-9999 on server one, 10,000-19,999 on server two, and so on). Then when the web server wants to assemble a page, it can simply call the nodes to provide their respective pieces. Sharding is ideal where the pieces can be assembled and presented for the user to process (read). Sharding supports hyperscale implementations, such as Google search.

Where sharding falters is when you want the system, not the user, to extract information across the various data stores for that individual. For example, if you store a user profile separate from his comments and you want to find information about all individuals who commented on a specific topic. This would involve joining the comments with the sharded user profiles. In short, no matter how cleanly you partition your data, someone will want to extract data that crosses the partitioning scheme you have imposed upon them. If your system is using transactions that operate across shards, then you have big challenges.

Shared-disk also has its scaling challenges, but these challenges are driven largely by the inter-nodal messaging. Shared-disk databases rely upon a message or token-passing model, where the messages alert the nodes to the status of the other nodes. The more nodes you add to the system, the longer it takes to pass this token, which results in longer wait-states. The impact of this on scalability is the single biggest knock against shared-disk databases. Shared-disk databases might be excellent in clusters of five or ten nodes, but might have issues at twenty nodes. The reality is that given the power of commodity machines today, a ten server cluster can handle almost any job. Companies, like ScaleDB, are working on this in hopes of achieving even higher scalability. One approach to addressing scalability limitations is by using Scale-in (see "Scale-in" section below).

#### Shared-Nothing:

Depending upon your partitioning scheme, function- and data-shipping volume can seriously limit your scalability. For websites in search of hyperscale, sharding is scalable and very popular. Sharding is ideal for read-only or read-centric solutions that don't involve multi-node transactions.

#### Shared-Disk:

Because of the timing for token-passing in a shared-disk database, there are a maximum practical number of servers per cluster. Scale-in is one way of addressing this.

### **Scale-in: An Approach to Scalability in Modern Multi-Core Servers**

Modern entry-level servers typically contain two or more processors. Today, the standard is a 4-core CPU, and this will jump in the next twelve months to 8 cores. As a result, entry-level servers in 2009 will have at least 16 cores. Of course, high-end servers today offer upwards of 32 cores. Not all DBMS can fully utilize this level of computing power. According to Jay Pipes, a leading MySQL evangelist, "MySQL's ability to scale up does not compare to Oracle's... From one to four processors, the inefficiency of MySQL in handling additional processors isn't as obvious."<sup>8</sup>

By leveraging a shared-disk architecture, this problem can be addressed by running several instances of MySQL on a single machine (hence the term Scale-in), with a shared-disk storage

---

<sup>6</sup> Wikipedia, Partitioning (database) [http://en.wikipedia.org/wiki/Partition\\_\(database\)](http://en.wikipedia.org/wiki/Partition_(database))

<sup>7</sup> Rahul Roy (July 28, 2008). *Shard - A Database Design*. <http://technoroy.blogspot.com/2008/07/shard-database-design.html>

<sup>8</sup> Jay Pipes: <http://www.jpipes.com/index.php?/archives/175-The-Ambiguously-Vague-Duo-Scale-Out-and-Scale-Up.html>

engine orchestrating these instances by coordinating disk access. For example, you can expect a 32-core machine to run 32 instances of MySQL running on top of a shared-disk storage engine. In effect, you are running a 32-server cluster inside a single server. You still get high-availability for MySQL-related software failure, and a second server can be added to provide failover for other problems.

#### Shared-Nothing:

Scale-in can be achieved with a shared-nothing architecture, but it involves both the database and the storage engine. It also requires that the application address each instance of MySQL independently. The disk would also have to be partitioned into separate stores for each instance. It quickly becomes an unmanageable problem.

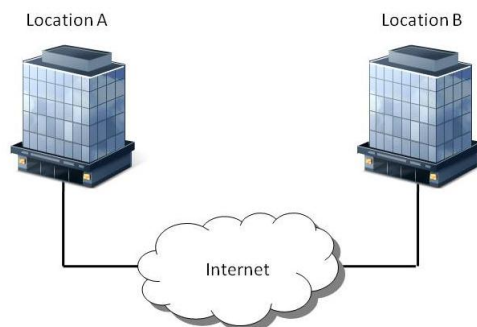
#### Shared-Disk:

Shared-disk databases are ideally suited to scale-in, a model that fully utilizes the multiple cores in modern servers. Database requests can be distributed to any instance, with a shared-disk engine serving the complete data set to any database instance.

### **Geo-Distribution**

When implementing a database in a single location, disk I/O is often a major performance bottleneck. Developers and DBAs respond by minimizing disk I/O in order to improve performance. When implementing a geographically distributed database—one spread across two or more locations—the major bottleneck becomes the high-latency connection between locations. To address this constraint, you must limit or eliminate any transactions that cross the geographic boundary. In other words, it is fine to have traffic between the remote locations as long as no local transactions must wait for a response from a remote data center. In the diagram below, any time a process at Location A requires information from Location B (or vice versa), you have a serious problem.

*Diagram 7: Geographically Distributed Data Centers*



The latency between geographically distributed sites has the same impact on both shared-nothing and shared-disk systems, the question is how well does your database handle this bottleneck.

Shared-nothing is designed to minimize all inter-nodal traffic, which includes messaging, function-shipping and data-shipping. However, as companies' databases, applications and user habits evolve, they want to traverse the artificial bounds imposed on the data by any partitioning scheme. In a geographically distributed implementation, the cost of data shipping is huge and should be avoided. A lower-cost approach is function-shipping, but it too has a big price, if the response is data-shipping back to the requestor. That said, the underpinnings of shared-nothing are the isolation of data into independent silos. This is well suited to geographically distributed databases.

Shared-disk, literally wants to share all data across all nodes on the cluster. Nodal interaction is managed through messaging between the nodes, so that updates are queued and reads are cognizant of any updates to their target data. Implementing such inter-nodal messaging across geographically distributed nodes results in a significant performance hit. Then ensuring that the data is consistent between various locations in real-time would bring such a system to its knees.

Whether you are using shared-nothing or shared-disk, geo-distributed data centers must be designed to operate as independently as possible. This doesn't mean that shared-disk doesn't work in a geographically distributed environment, it merely means that instead of one large shared-disk design, you must implement two or more separate clusters, one per location, that operate as independently as possible.

Example: A major auto insurance company might have two data centers, one located on the east coast and one located on the west coast. Regardless of which DBMS architecture you use, you want each data center to operate independently for any particular customer process (e.g. registration, billing, claim, etc.). So in this scenario, there are two distinct physical clusters, one in each data center. People living east of the Mississippi river might be routed to the data center on the east coast, while those individuals west of the Mississippi would be routed to the west coast data center. This limits data sharing between the two data centers to processes that do not impact mission-critical performance, such as backups and global reporting.

In short, geographical distribution works for either architecture, as long as each location hosts an independent cluster that does not share data or messaging in any significant way across locations. Because the two clusters operate independently, your application will require some level of routing for database requests, just as you would do in a shared-nothing environment.

#### Shared-Nothing:

Because of latency issues between geographically distributed databases, you should partition data based upon geo-location to minimize or eliminate function- and data-shipping between disparate locations.

#### Shared-Disk:

Shared-disk has the same latency issues between geographically remote locations as shared-nothing. Address this situation by building two separate shared-disk systems, in effect partitioning them, like you would with a shared-nothing database.

### **Cloud Computing: Database-as-a-Service (DaaS)**

One of the key compelling benefits of cloud computing, and Database as a Service (DaaS), is the ability to dynamically add nodes to a cluster when you need them, and release nodes when you don't need them. This dynamic expansion and contraction of nodes, in response to changes in usage, requires a compatible database architecture. Unfortunately, this type of dynamic scalability is anathema to databases that require static and manual partitioning.

If you implement a shared-nothing database, partitioning design can be done manually or it can be automated. Manual partitioning is untenable in a dynamic environment, because it is time-consuming and it requires restarting the cluster for each new partitioning scheme to take effect. If you are bringing down your application each time you want to manually add/remove a server in your cluster, this is anything but dynamic. Middleware can automate the partitioning, but these middleware solutions result in sub-optimal partitioning, which can result in very heavy function- and data-shipping. This, in turn, can cripple performance of the cluster. In fact, any gains you might

achieve by adding a node can be overwhelmed by the performance hit from data sharing between nodes.

A shared-disk architecture, on the other hand, is ideally suited for dynamic nodal scalability, or elasticity. Since the data is separated from the nodes where it is processed, you can add or remove nodes without impacting the underlying data. At ScaleDB we describe this as Plug-and-Cluster™, the ability to add or remove nodes on the cluster dynamically.

For more detail on cloud databases, see the white paper “Cloud Computing & Databases”<sup>9</sup>

#### Shared-Nothing:

Partitioning poses a real problem for cloud databases, where nodal elasticity is one of the compelling benefits. There are two options for partitioning and both of them are bad: (1) hand-tuned partitioning results in minimal function- and data-shipping but it is incompatible with dynamic nodal elasticity; (2) automated partitioning enables nodal elasticity, but this partitioning is sub-optimal, meaning it increases function- and data-shipping, which degrades performance. To make matters worse, the poor performance gets worse as you add more nodes.

#### Shared-Disk:

Shared-disk is compatible with nodal elasticity. Since all nodes feed from the same shared data store, you can simply increase/decrease the number of nodes with no impact on the underlying data.

### **Total Cost of Ownership (TCO)**

When comparing TCO between the shared-disk and shared-nothing architectures, you need to take into account the following:

- Software license costs
- Hardware costs
- Development & Maintenance of:
  - Application
  - Middleware
  - Database

#### **Software License Costs**

Shared-nothing is much easier for the DBMS *vendor* to build. The vendor simply builds a standalone database, adds data shipping and possibly some replication for slaves, and ships the product. Shared-disk systems are designed from the ground up to accommodate clustering. Building a shared-disk DBMS is much harder for the DBMS vendor, who has to orchestrate nodal interaction through inter-nodal messaging. This involves orchestrating data access, locking, failure, recovery, etc. Because of this added layer of complexity, shared-disk DBMS are typically substantially more expensive than shared-nothing. For example, Oracle charges 75% more for Oracle® RAC versus Oracle 11g.

#### **Hardware Costs**

Shared-disk provides load balancing across all nodes (or servers), so you scale your architecture at the cluster level. Shared-nothing isolates the load to each individual server. As a result, shared-

---

<sup>9</sup> Mike Hogan, “Database Virtualization and the Cloud”  
[http://www.scaledb.com/pdfs/Cloud\\_Databases\\_WhitePaper2.pdf](http://www.scaledb.com/pdfs/Cloud_Databases_WhitePaper2.pdf).

nothing dictates that you scale each node to handle the peak load for that data. In other words, shared-nothing forces you to scale your cluster at the *server* level, while shared-disk enables you to scale your server at the *cluster* level. This means that shared-nothing will require more servers in the cluster and those servers may need to be more powerful than with a shared-disk database. This means shared-nothing clusters will increase your server budget. However, while shared-disk reduces your server costs, it adds the cost of shared storage which can mean an additional server, NAS or SAN, depending upon your needs.

### **Development & Maintenance Costs**

*Application Level Costs:* The shared-nothing architecture requires that the applications route the database requests to the correct database server. This requires a database routing table. This routing table may be integrated into the application, or it may be a middleware solution. In either case, it must be created upon set-up and it must be updated over time to reflect re-partitioning.

Shared-disk, on the other hand, does not use routing tables, since every node can handle any database call. In a shared-disk database, the database requests are simply routed to the next available node in the cluster. Because it doesn't use routing tables, the initial set-up and ongoing maintenance costs at the application level, with a shared-disk database, are lower.

*Middleware Level Costs:* Shared-nothing databases have no knowledge of the presence of other nodes. As a result, you will need to handle things like master-slave replication (as applicable), failover, and other management functions that enable you to orchestrate the independent servers in the cluster. This may be handled at the application, middleware or even the database layers.

Shared-disk, which is designed from the ground up for clustering, incorporates these functions in the database. Because it doesn't require replication and it handles failover inherently, shared-disk has lower middleware costs for set-up and maintenance.

*Database Level Costs:* Shared-nothing requires manpower-intensive performance monitoring, data partitioning and re-partitioning. This partitioning must then be incorporated into the database routing table with each re-partition. Since shared-disk eliminates the entire process of data partitioning and re-partitioning, these savings can be substantial.

Another issue impacting maintenance of the database is addressing hotspots. Data traffic jams, also known as hotspots, can be caused by a variety of factors. The 80/20 rule applies, where 80 percent of the traffic is generated by accessing 20% of the data. For this reason, a shared-nothing database that applies an even distribution of the data, based solely on volume, is typically sub-optimal and will result in hotspots.

With shared everything, any number of nodes can process the data, so this eliminates the challenge of detecting and then re-partitioning to remove hotspots. This can substantially reduce the database level costs.

### **TCO Comparison Summary**

When comparing the maintenance costs of the two architectures, it is clear that once again the analogy of the manual and automatic transmissions applies. Just as it is much easier for the driver to use an automatic transmission, shared-disk architectures are much easier to set-up and maintain. By eliminating data partitioning, re-partitioning, request routing and middleware tools, shared-disk can dramatically reduce manpower costs. Shared-nothing, like the manual transmission, requires additional manual work that can substantially increase your total cost of

ownership. Traditionally, the reduced manual costs from shared-disk architectures were offset by the increased licensing costs of the DBMS, just as automatic transmissions are more expensive than manual transmissions. However, with the introduction of open source DBMS and ScaleDB's plug-in engine, the shared-disk premium is almost eliminated, and shared-disk offers a superior TCO.

#### Shared-Nothing:

With open source shared-nothing databases available, you have no license costs. However, the manpower required to create and maintain data partitioning, tuning, monitoring, re-partitioning, slave replication, and more, will increase the total cost of the system considerably.

#### Shared-Disk:

Shared-disk databases have much lower labor costs for set-up and maintenance. The software, being more technically advanced than shared-nothing, typically sells for a significant premium. ScaleDB, by selling a shared-disk storage engine underneath the free open source MySQL software, virtually eliminates the traditional shared-disk premium. Combine this with the savings on manual set-up and maintenance processes and the cost decision is a no-brainer.

## Conclusion

The comparison between shared-disk and shared-nothing is analogous to comparing automotive transmissions. Under certain conditions and, in the hands of an expert, the manual transmission provides a modest performance improvement. But under the vast majority of real world conditions, the automatic transmission provides a better overall experience. Similarly, shared-nothing can be tuned to provide superior performance, assuming you can minimize the function- and data-shipping. Unfortunately, this is rarely a valid assumption. Shared-disk, much like an automatic transmission, is easier to set-up and it adjusts over time to accommodate changing usage patterns.

The chart at the front of this white paper provides a comparison between the two database architectures. It is up to you to weigh the importance of each of the various issues, relative to your application. The choice between shared-nothing and shared-disk is a matter of personal preference, application requirements (today and going forward), costs, system demands (e.g. high-availability), and the level of set-up and manual database tuning that makes you comfortable.

***About the Author:*** The author is Mike Hogan, CEO of ScaleDB Inc. a company that provides a storage engine for MySQL utilizing the shared-disk architecture. Mike has never owned a car with an automatic transmission. While Mike put the words on paper, the real genius behind this paper is Vern Watts. Vern, a Distinguished Engineer at IBM, was the chief architect of IBM IMS, which has employed the shared-disk architecture since 1978. The IMS database currently serves as the workhorse for most of the largest companies in the world, processing 50 Billion Gigabytes of data. The insights in this paper are based on Vern's forty years of real-world database deployment experience. Vern is now the chief architect at ScaleDB Inc.