



# A Primer on Database Clustering Architectures

by  
Mike Hogan, CEO  
ScaleDB Inc.

## Overview

Users often implement databases in a clustered configuration in order to accommodate business requirements such as scalability, performance, high-availability and elasticity. When implementing a cluster, the architecture of the underlying DBMS is an important factor to consider. There are two primary database architectures: shared-nothing and shared-disk. From these primary models there are two derivative architectures: NoSQL and shared-cache. This paper provides a high-level comparison of these four architectures. For a more detailed comparison on the pros and cons of the database architectures, you will want to read our more detailed technical white papers.

## Other Complementary White Paper Resources

[Shared-Disk vs. Shared-Nothing](#): An in-depth comparison of the primary database architectures.

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

## An Introduction to Clustering

Wikipedia describes [cluster computing](#) as *a group of linked computers, working together closely so that in many respects they form a single computer*. In the database vernacular, clustering means that the application sees a single database, while under the covers there are two or more computers processing the data. In addition to providing scalability, database clusters can deliver additional benefits such as load balancing and high-availability, but these things are not inherent in all database clusters.

## Shared-Nothing

Shared-nothing works on the principle that each node in a cluster has sole ownership of the data on that node. Each node literally shares no data with the other nodes of the cluster, hence the term shared-nothing. As you will see, this principle is not always maintained, especially with NoSQL implementations. When you move from a single server to multiple servers, in a shared-nothing cluster, you must split the data across the servers. This process of splitting the data across servers, as indicated in the diagram below, is called partitioning. Data can be partitioned<sup>1</sup> vertically or horizontally, but this is outside the scope of this paper.

Diagram 1: Shared-Nothing (partitioning the data across nodes)



Requests for data are then processed through a routing table that routes each request to the server/node that owns that data. For example, the Server 1 above may have information about users, while Server 2 might have information about orders. If your application makes a request that involves both servers, for example requesting a list of users and order information for orders placed in the prior month, you need to involve both servers. The database would solve this request by reading the list of orders placed the prior month and then sending that list from Server 2 to Server 1 to add the information about the users. This process of sending data from one server to

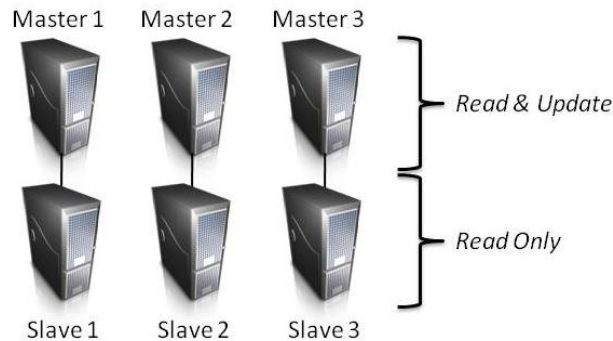
<sup>1</sup> Wikipedia, Partitioning (database): [http://en.wikipedia.org/wiki/Partition\\_\(database\)](http://en.wikipedia.org/wiki/Partition_(database))

another is called data shipping. At its core, a shared-nothing database is a standalone database with the added facility for data shipping.

**Fail-Over and the Master-Slave Configuration:**

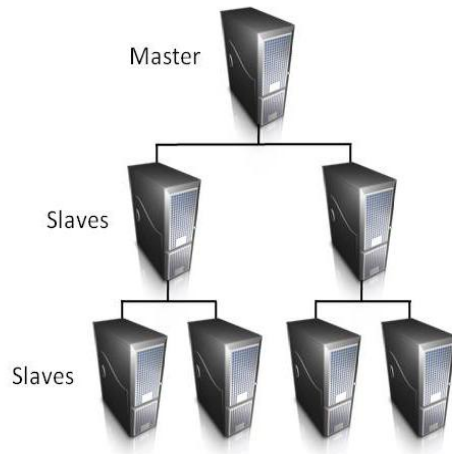
If a server fails, all applications that require access to the failed server also fail. For this reason, each node needs a replicated copy that can take over in case the primary node fails. This node is called a slave. In the master-slave configuration, the master can perform both reads and updates, while the slave can only provide read access to the slave's local copy of the data.

*Diagram 2: Master-Slave Configuration*



For applications with a high ratio of reads versus updates, the slaves can be configured in a tree, where the data from the master is replicated down the tree, and the slaves then off-load the read-access.

*Diagram 3: Tree of Slaves*



There is nothing to prevent a shared-disk from using a similar tree of read-only slaves, but slaves are generally associated with shared-nothing.

**Examples of Shared-Nothing DBMS:** Oracle 11g, IBM DB/2 (non-mainframe), Sybase ASE, MySQL (InnoDB, MyISAM, etc...all storage engines except ScaleDB), Microsoft SQL Server, etc.

## NoSQL

NoSQL is derived from the principles embodied in shared-nothing, namely the splitting of the data. However, the NoSQL database—or more precisely a data store or key-value store—puts a higher priority on scale-out, elasticity and high-availability. These can be accomplished in a shared-nothing database, however they involve a lot of work to set-up and maintain, and you pay a considerable performance penalty.

NoSQL makes a very calculated decision to relax consistency—the “C” in [ACID](#)—in order to achieve these objectives. NoSQL provides eventual consistency. This means that users querying the system at a single point in time may get different answers, but eventually those different answers will be automatically resolved to a single answer.

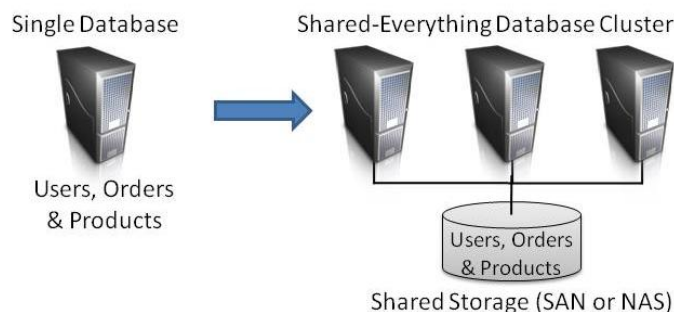
While NoSQL doesn't technically share a single piece of data, it does duplicate that data. When using a master-slave configuration in a shared-nothing SQL database, the slave gets a copy of the data as well. The big difference is that, unlike shared-nothing SQL databases, NoSQL databases allow the various servers to simultaneously change the *copies* as well as the original. Then it attempts to resolve the various copies. This is done by selecting which answer is the winner and which are the losers. While the losers are saved, just in case there is a problem, the winner is eventually selected and replicated throughout the database.

Because NoSQL is not [ACID compliant](#)—it can deliver different results to different users at the same time—it is not widely used in enterprise applications. NoSQL has been largely relegated to Web 2.0 applications where scale and availability are more important than consistent data.

## Shared-Disk

Shared-disk, also known as shared-everything, works on the principle that you have one array of disks, typically a Storage Area Network (SAN), or Network Attached Storage (NAS), that holds all of the data in that database. Each server or node in the cluster acts on that single collection of data in real-time.

Diagram 4: Shared Disk Cluster (all nodes have access to all of the data)



In a shared-disk architecture, any node can satisfy any request, because they each have access to all of the data. So, instead of going to a specific node for specific data, shared-disk can simply route the request to the next available node. Because each node can address any database request, the load is inherently balanced across the nodes in the cluster.

### Fail-Over and the Master-Master Configuration:

While shared-nothing has only one node that can update any piece of data in the database, shared-disk enables any node to update the database. For this reason, shared-disk is called a master-master architecture. As such, it fully utilizes each server in the cluster. This architecture also provides inherent failover, since each node acts as a back-up to every other node.

However, nothing comes for free. In order to enable this flexibility, the nodes communicate with each other to coordinate their activity, specifically: locking, buffering and status. This inter-nodal messaging creates some degree of overhead, but you must weigh this overhead against the impact of data/function shipping found in shared-nothing clusters.

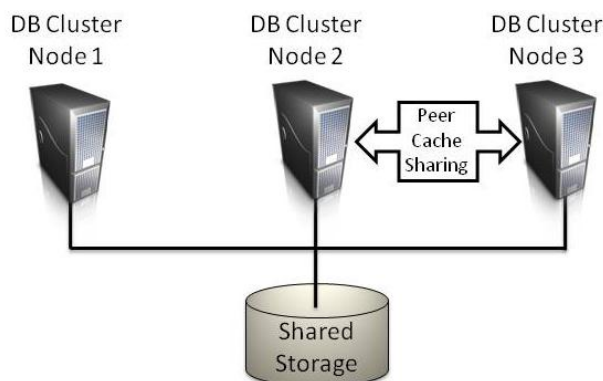
Examples of Shared-Disk DBMS: IBM IMS and DB/2 (mainframe-only).

## Shared-Cache

Shared-cache is a derivative of the shared-disk architecture. While the shared-disk architecture delivers a number of functional benefits, the overhead of sharing data via the disk—the slowest component in the system—introduces a significant performance penalty. Shared-cache uses a high-speed cache to share information between nodes. This is much faster than sharing via the disk.

There are two methods for sharing cache. Oracle RAC® uses a peer-based cache sharing approach. Diagram 5 shows a peer-based cache sharing approach, as used by Oracle RAC. When Node 3 wants data currently in the cache of Node 2, it requests the data directly from the cache, instead of going through the disk.

*Diagram 5: Shared-Cache Using Peer-Based Cache Sharing*



The other approach to sharing cache is to use servers, or virtual machines, that act as a common repository for the cached data. If you are familiar with Memcached, and how it caches data between the application and the database, the tier-based cache sharing architecture applies the same principle, between the database nodes and the storage.

One significant different between peer-based caching and tier-based caching is how they handle storage. The peer-based approach used by Oracle relies upon high-end storage (e.g. SAN or NAS) with a cluster file system (e.g. OCFS2) so that all of the nodes can address the same storage volume.

Tier-based cache sharing, like that used by ScaleDB, involves storage that is directly attached to a single cache server. The cache server then coordinates the sharing of that data in a manner that is

optimized for database functions. Because the storage is not shared directly, you have the flexibility to use lower-cost local storage and cloud storage, in addition to SAN or NAS. This also eliminates the cost and complexity of an added cluster file system. Diagram 6 shows a ScaleDB cluster with mirrored cache servers (Cache Accelerator Server or CAS) leveraging local storage.

*Diagram 6: Shared-Cache Using Tier-Based Cache Sharing with Mirrored Local Storage*

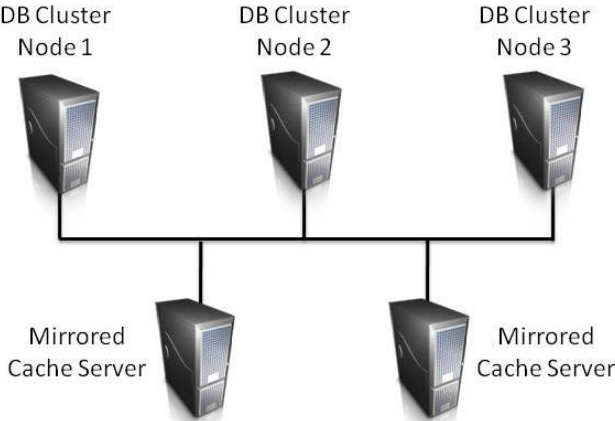
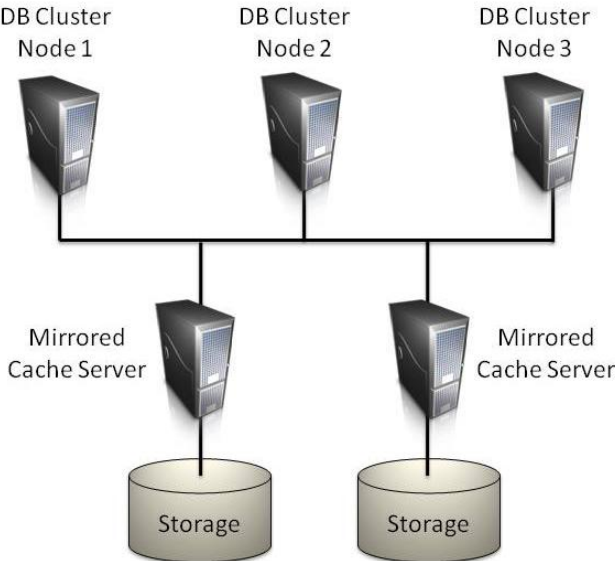


Diagram 7 shows a ScaleDB cluster with mirrored cache servers using remote storage, which can be cloud storage, SAN or NAS.

*Diagram 7: Shared-Cache Using Tier-Based Cache Sharing with Mirrored SAN, NAS or Cloud Storage*



Note: The introduction of high-performance flash memory (AKA Solid State Disk or SSD) is also changing the performance profile of the shared-disk database, but cache remains faster than flash.

### Which Architecture is Best?

Unfortunately, such a simple question does not have a simple answer. Each architecture has its pros and cons. Below is a summary of the criteria that will drive you to each database architecture:

**Shared-Nothing SQL:** If you want SQL and data consistency, and you are willing to invest significant time in set-up, maintenance and ongoing tuning, then a SQL-based shared-nothing database can, when properly tuned, deliver very good performance. SQL-based shared-nothing databases do not deliver elasticity, since partitioning is very rigid. Beware that use of slaves to satisfy queries can result in a consistency lag of up to 800 seconds.

**NoSQL:** If scaling, elasticity and high-availability are priorities, and you are willing to do without tight data consistency and the SQL language, then NoSQL is a good option. NoSQL should not be used in mission-critical environments where transactional integrity is required, such as when processing financial transactions.

**Shared-Disk:** Shared-disk and shared-cache databases provide the richest functionality. They provide full SQL support, high-availability, strict data consistency and elasticity. However, because shared-disk databases rely upon the disk for data sharing, their performance doesn't rival the performance of shared-nothing or NoSQL databases. If you are willing to trade performance for functionality, then shared-disk is a good option, but why would you want shared-disk when you can get the same functionality and superior performance from a shared-cache database?

**Shared-Cache:** Shared-cache is essentially a faster shared-disk. It delivers low cost set-up and maintenance. Tuning is a snap, since there is no need to partition your data. It also provides high-availability, elasticity and strict data consistency.

**Performance:** In judging which database is the best fit for your application, performance is always an important criteria. Unfortunately, it is impossible to say for sure which architecture will perform the best. It depends upon your application, your data and how easily it can be partitioned and more. If the data is well partitioned and all database requests can be satisfied without data shipping between nodes, then shared-nothing can perform very well. If the data is richly linked and you use queries that would span partitions, then shared-cache can perform better. As you can see, database performance really depends upon what you are doing with it.

In many ways, comparing the two primary database architectures is like comparing automobile transmissions. Shared-cache is analogous to the automatic transmission, because it automates much of the complexity of 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. And just as drivers tend to be passionate about their preferred transmission, DBAs tend to be passionate about their preferred database architecture.