A Primer on Database Clustering Architectures

Last Updated: December 10, 2014

Overview

Users often implement their SQL 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-data. This paper provides a high-level comparison of these primary 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-data 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.

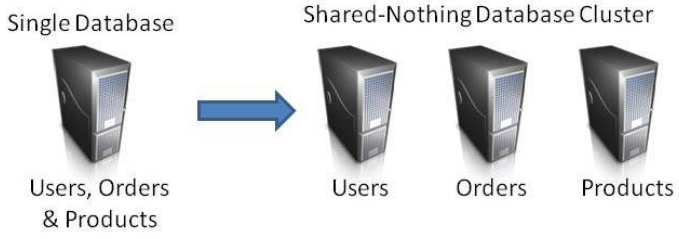
An Introduction to Clustering

Wikipedia describes cluster computing as "A computer cluster consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system." 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 to all database architectures.

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. 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 or sharding. Data can be partitioned[1] vertically or horizontally—with sharding being a type of horizontal partitioning—but this level of detail 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 database 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. In a shared-nothing database, the distinct database instances know nothing of each other. As a result, manipulating data across distinct partitions requires that such manipulation move from the database—the traditional data

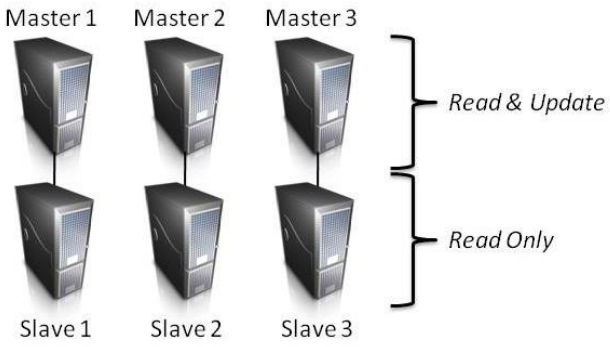
[1] Wikipedia, Partitioning (database): http://en.wikipedia.org/wiki/Partition_(database)

manipulation tool—to the application. By moving cross-shard data processing to the application tier, it increases application complexity; increasing development, QA and maintenance costs.

In addition to the impact on your application, you need to consider the performance impact. If your application is easily partitioned such that the vast majority of data manipulation can be handled in a single partition—thus avoiding moving that data to the application tier—the performance will be quite good. If your application requires a high-level of cross-partition data manipulation, you'll find that you are moving a lot of data to the application tier for processing, instead of processing it where it resides. This can have a fairly negative impact on performance.

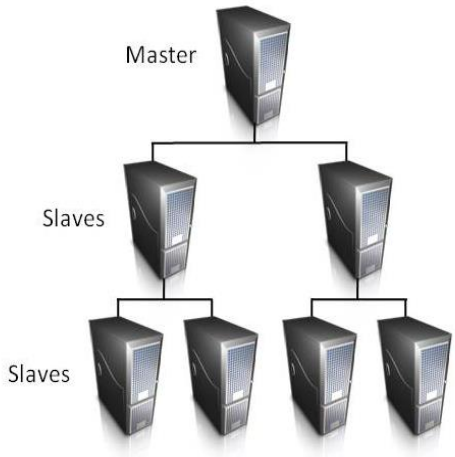
Fail-Over and the Master-Slave Configuration:
If a server fails, all applications that require access to the failed server also fail. For this reason, each node needs a replicated copy that can take over in case the primary node fails. This backup copy of the database is called a slave. In the master-slave 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-data DBMS from using a similar tree of read-only slaves, but slaves are generally associated with the shared-nothing DBMS.

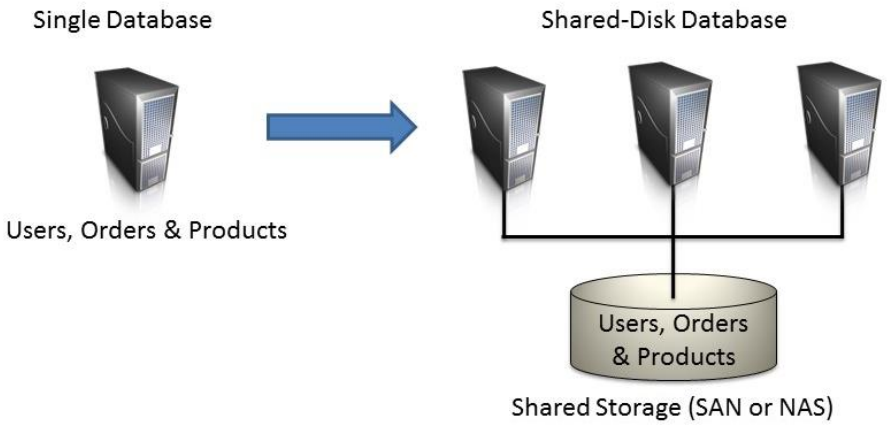
Examples of Shared-Nothing DBMS: Oracle 11g, IBM DB/2 (non-mainframe), Sybase ASE, MySQL (InnoBD, MyISAM, etc…all storage engines except ScaleDB), Microsoft SQL Server, etc.

Shared-Data

Shared-data, also known as shared-everything, works on the principle that there are multiple database nodes that all have read/write access across the entire database. In effect, these database nodes are sharing the data. Traditional shared-data DBMS leverage a single pool of disks, leading to the original term "shared-disk". However, for reasons of performance and cost, the new shared-data model has replaced the traditional shared-disk approach. The shared-data model delivers all of the same benefits of shared-disk, while addressing the negatives of cost and performance. The shared data architecture replaces the high-end storage with a pool of commodity servers, acting as both a low-cost storage tier and a high-performance distributed processing tier. The shared-data model, as described below, functions in a manner very similar to MapReduce, by pushing data processing to smart data nodes that all process the data in parallel.

Traditional (Old School) Shared-Disk Architecture:
The shared-nothing DBMS architecture originally rose to prominence based upon performance and cost advantages over the traditional shared-disk architecture. Sharing a disk, or a pool of disks, required high-end storage, to ensure high-availability. It made no sense to implement a highly-available database with storage that was not highly-available. This meant purchasing a Storage Area Network (SAN), or Network Attached Storage (NAS) that are extremely expensive. To make matters worse, the data had to be moved from the storage to the database nodes for processing. For example, if you wanted a simple row count on a large table, you would have to move that entire table to a database node, just to count the rows. The combination of high costs and poor performance is always a category killer, and this paved the way for shared-nothing dominance.

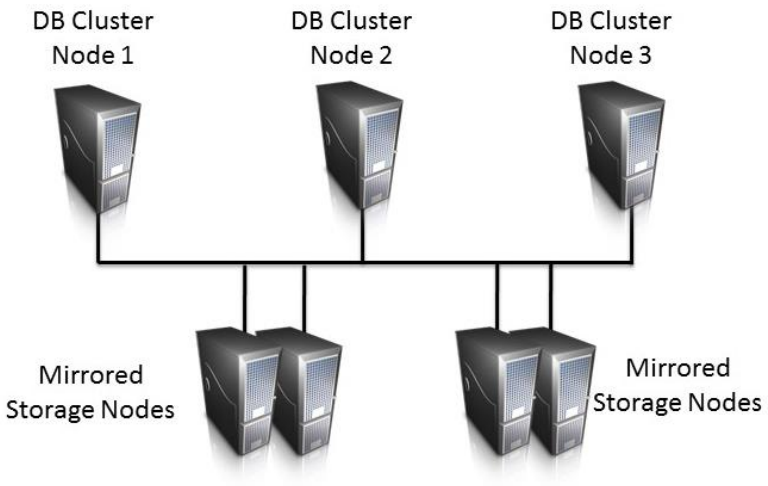
Diagram 4: Traditional Shared-Disk Cluster



Next Generation Shared-Data Architecture:
Shared-data is an architecture that maintains the benefits of shared-disk—namely high-availability, cluster-wide consistency, and shard-free partitioning—while eliminated the traditional cost and performance challenges. The shared-data architecture replaces high-cost storage with a collection of low-cost commodity servers. Software turns these servers into a highly-available storage tier

through processes like data mirroring, fail-over, etc. The added benefit of using servers is that these storage nodes can now process their local data in parallel, much like Hadoop/MapReduce.

Diagram 5: Shared-Data Database with Mirrored Storage



Moving data to a central processing node is terribly inefficient. It consumes both bandwidth and time (efficiency) to move the data. Then the problem is that the single node becomes the processing bottleneck. The shared-data approach instead moves the processing to the storage nodes. For example, consider that you have a customer table with one million customers, spread over four storage nodes, and you want to find out how many are based in California. The shared-data approach pushes that function to the four storage nodes that each send only the results to the database node. That node simply aggregates the data and passes it to the application. This process exploits the parallel processing of four storage nodes, each addressing one quarter of the work and it only ships the results, so the network is used more efficiently as well. Since the processing is, in this scenario, divided over four servers, it is roughly four-times faster than the traditional shared-disk approach.

Shared-Data Benefits:

1. High-Availability: Since all database nodes have read/write access over the entire database, there is no specialization of database nodes. In other words, if one database node fails, another database node can rollback any uncommitted transactions from the failed node and the cluster simply keeps running.
2. Flexible Load Balancing: Since any database node can process any data, changes in workloads can be evenly spread across the cluster. An example: an application serving finance might process payroll one day, benefits the next and taxes the next day. Instead of having database nodes specialize on tables/functions as with a shared-nothing database, the shared-data database nodes seamlessly transition between functions.
3. Shard-Free Scale-Out: Scaling a shared-data database is as simple as adding nodes to the cluster, and these nodes can be added or removed elastically. This is far easier and more dynamic than scaling out a shared-nothing database via sharding.
4. High-Performance: By moving the processing to the storage nodes, also known as function shipping, shared-data databases can perform exceptionally well. Shared-nothing can also

perform exceptionally well, but only if the workload is optimally distributed, otherwise you get data skew and hotspots impacting performance.

Examples of Shared-Disk DBMS: IBM IMS and DB/2 (mainframe-only).
Examples of Shared-Data DBMS: Oracle RAC, ScaleDB, Amazon Aurora (although Aurora does not handle distributed locking).

Which DBMS Clustering Architecture is the Best?

Unfortunately, this simple question does not have a simple answer. Each architecture has its pros and cons. If your application is easily partitioned and you value optimal performance over ease of use, then shared-nothing is your best bet. Keep in mind that a shared-nothing cluster will force cross-partition logic into the application tier and you will need to handle ongoing database tuning and repartitioning to maintain optimal performance.

If you want excellent performance and you value ease of use, or if your application is not easily partitioned such that all inter-database dependencies are eliminated, then shared-disk is a better option. Shared-data models also provide superior high-availability and elastic scaling, unlike shared-nothing, so if those are concerns, then shared-data is a better option for you.

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.