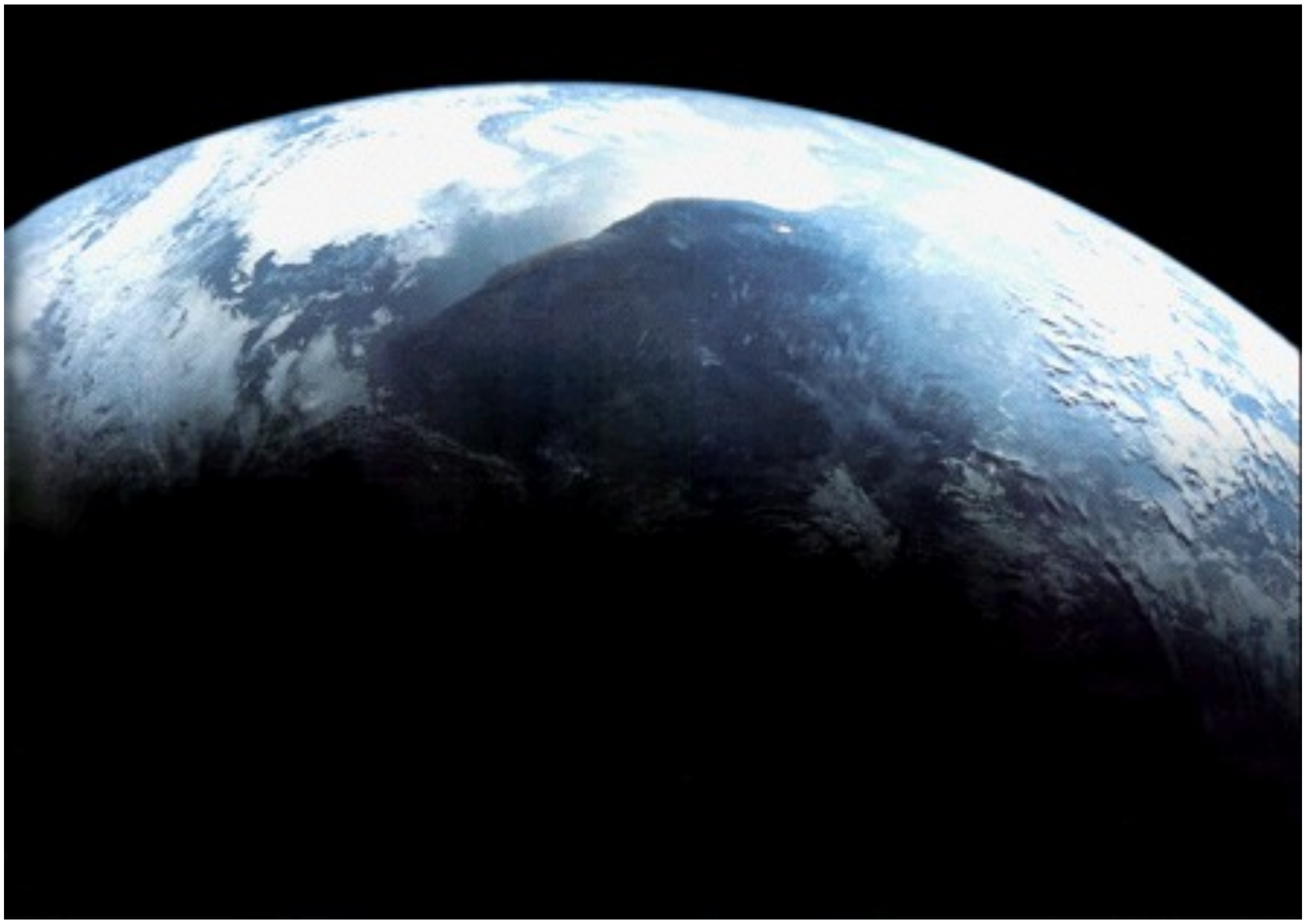
W H I T E   P A P E R

The Pros and Cons of Database Scaling Options

By Craig S. Mullins



Mullins Consulting, Inc.
http://www.mullinsconsulting.com

Table of Contents

Modern IT organizations rely on the ability to quickly react to changing business trends and technology forces. The days of long development cycles requiring many months or even years are in the rear-view window of history. And as systems change, usage changes, and that requires effective ways to adjust for fluctuations in demand.

Database Management Systems are at the heart of most applications and, as such, the ability to scale database systems elastically is important. This white paper will examine the various methods used to accomplish database availability and elasticity, as well as introduce a new architectural option for database scaling.

Industry Trends Impacting Scalability Needs

Today's business systems are undergoing a revolutionary transformation to accommodate digital transformation. There are four over-arching trends that are driving digital transformation today summarized by the acronym SMAC: social, mobile, analytics and cloud.

Mobile computing has transformed the way most people interact with applications, while social media has transformed the way people interact with companies and each other. Just about everybody has a smartphone, a tablet, or both. And the devices are being used to keep people engaged throughout the day, no matter where they are located. This change means that customers are engaging and interacting with organizations more frequently, from more diverse locations than ever before, and at any time around-the-clock. End users are constantly checking their balances, searching for deals, monitoring their health, and more from mobile devices. And their expectation is that they can access their information at their convenience.

> *Organizations are increasingly looking to move away from high-performance static hardware to dynamically changing, virtual, commodity infrastructure.*

Cloud computing, which is the practice of using a network of remote servers hosted on the internet to store, manage, and process data and applications, rather than a local host, enables more types and sizes of organizations than ever before to be able to deploy and make use of computing resources—without having to own those resources. Applications and databases hosted in the cloud need to be resilient in order to adapt to changing workloads. In addition, the cloud's intense distribution of compute and storage resources means that responding to scale is no longer about just throwing sheer power at the problem, but also about marrying the right resources in the right location to an individual problem.

Finally, the Big Data phenomenon has boosted the amount of data being created and stored. The amount and types of data that can be accessed and analyzed continue to grow by leaps and bounds. And when analytics is performed on data from mobile, social, and cloud computing, it becomes more accessible and useful by anyone, anywhere, at any time.

In addition, as cloud and container-based environments become ubiquitous, organizations are increasingly looking to move away from high-performance static hardware to dynamically changing, virtual, commodity infrastructure that can be purchased or rented in a "pay-as-you-go" manner or that can be easily reallocated when not in use.

The impact of these forces on our computing infrastructure is that systems need to expand and contract easily to meet an ever-changing demand. Organizations need to be adaptable and to be able to scale their systems as workload changes.

Defining Scalability and Elasticity

But why exactly do we care about scalability and elasticity? At a high level, both help to improve availability and performance when demand is changing, especially when changes are unpredictable. If the data is not available, applications cannot run. If applications cannot run or run slowly, the company loses business. Therefore, it is important to be able to ensure that databases are kept online and operational.

So how do scalability and elasticity help to improve availability and performance?

> ***Scalability*** refers to the capability of a system to handle a growing amount of work, or its potential to perform more total work in the same elapsed time when processing power is expanded to accommodate growth. A system is said to be scalable if it can increase its workload and throughput when additional resources are added.
>
> A related aspect of scalability is availability and the ability of the DBMS to undergo administration (e.g. schema changes) and servicing (e.g. upgrades and maintenance) without impacting applications and end user accessibility. A scalable system can be changed to adapt to changing workloads without impacting its accessibility, thereby assuring continuing availability even as modifications are made.
>
> A scalable system can react to evolving needs with adjustable resources to serve a changing workload without requiring downtime.
>
> ***Elasticity*** is the degree to which a system can adapt to workload changes by provisioning and deprovisioning resources in an on-demand manner, such that at each point in time the available resources match the current demand as closely as possible.
>
> The goal of elasticity is to match the amount of resources allocated to a service with the amount of resources it actually requires. This means that both over-provisioning and under-provisioning can be avoided. Over-provisioning occurs when more resources are allocated than required, and it should be avoided in a cloud model because the service provider must pay for all allocated resources, which can increase the cost to cloud

customers. With under-provisioning fewer resources are allocated than required, and this is to be avoided because it typically results in performance problems; severe cases can look like downtime to the end user, resulting in customers abandoning the application, which has a financial impact.

From a database perspective, elasticity infers a flexible data model and clustering capabilities. The greater the number of changes that can be tolerated, and the ease with which clustering can be managed, the more elastic the DBMS.

Types of Database Scalability

First, let's look at the ways that databases can be scaled and examine the benefits and drawbacks of each method. There are two broad categories for scaling database systems: vertical scaling and horizontal scaling.

Vertical scaling, also known as scaling up, is the process of adding resources, such as memory or more powerful CPUs to an existing server. Removing memory or changing to a less powerful CPU is known as scaling down.

Adding or replacing resources to a system typically results in performance gains, but realizing such gains often requires reconfiguration and downtime. Furthermore, there are limitations to the amount of additional resources that can be applied to a single system, as well as to the software that uses the system.

Vertical scaling has been a standard method of scaling for traditional RDBMSs that are architected on a single-server type model. Nevertheless, every piece of hardware has limitations that, when met, cause further vertical scaling to be impossible. For example, if your system only supports 256 GB of memory, when you need more memory you must migrate to a bigger box, which is a costly and risky procedure requiring database and application downtime.

Horizontal scaling, sometimes referred to as scaling out, is the process of adding more hardware to a system. This typically means adding nodes (new servers) to an existing system. Doing the opposite, that is removing hardware, is known as scaling in.

With the cost of hardware declining, it makes more sense to adopt horizontal scaling using low-cost "commodity" systems for tasks that previously required larger computers, such as mainframes. Of course, horizontal scaling can be limited by the capability of software to exploit networked computer resources and other technology constraints. And keep in mind that traditional database servers cannot run on more than a few machines. In such cases, scaling is limited, in that you are scaling to several machines, not to 100x or more.

Horizontal and vertical scaling can be combined, with resources added to existing servers to scale vertically and additional servers added to scale horizontally when required. It is wise to consider the tradeoffs between horizontal and vertical scaling as you consider each approach.

Horizontal scaling results in more computers networked together and that will cause increased management complexity. It can also result in latency between nodes and complicate programming efforts if not properly managed by either the database system or the application. That said, depending on your database system's hardware requirements, you can often buy several commodity boxes for the price of a single, expensive, and often custom-built server that vertical scaling requires.

On the other hand, depending on your requirements, vertical scaling actually can be less costly if you've already invested in the hardware; it typically costs less to reconfigure existing hardware than to procure and configure new hardware. Of course, vertical scaling can lead to over-provisioning which can be quite costly. At any rate, virtualization perhaps can help to alleviate the costs of scaling.

Traditional Database Scalability Methods

Let's turn our attention to traditional methods for achieving scalability in database systems. Databases scalability is often implemented by clustering. With clustering, multiple servers are used to serve database requests.

There are two predominant architectures for implementing database clustering: shared-disk and shared-nothing. At a high level, these names do a reasonable job of describing the nature of the architecture, but let's take a more in-depth look into each. Both are forms of horizontal scaling. Later in this paper, we'll explore a new architecture that combines many of the benefits of both of these architectures.

Shared-Disk

With a ***shared-disk*** environment all of the connected systems share the same disk devices. Refer to *Figure 1*. Each processor still has its own private memory, but all the processors can directly address all the disks. This means that there is no need to break apart data into separate partitions because **all** of the data is shared in shared-disk implementations.

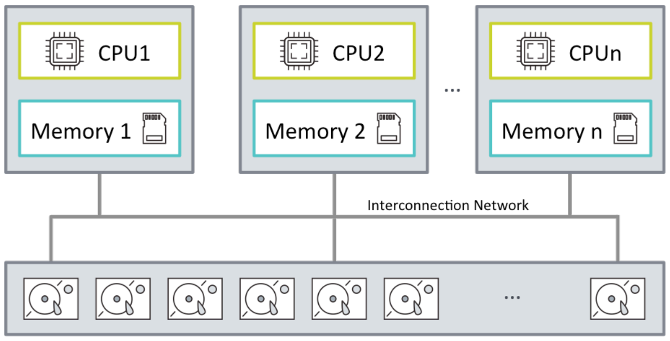


Figure 1. The shared-disk architecture.

But it is important to understand that only the disks are shared. Main memory is not shared; each processor has exclusive access to its memory. Because any processor can cache the same data from disk, a cache coherency mechanism is necessary to ensure consistency when multiple nodes modify the data. A distributed lock management capability is also required to manage the consistency of the data as it is being requested and modified by multiple network nodes.

A shared-disk implementation offers several benefits including potentially lower cost, extensibility, availability, load balancing, and relatively simple migration from a centralized system. However, shared-disk benefits from potentially costly Storage Area Networks (SANs), which can drive up the cost. Typically, shared-disk clustering tends not to scale as well as shared-nothing for smaller machines. But with some optimization techniques, shared-disk is well-suited to larger enterprise processing, as such is done in the mainframe environment.

The specialized technology and software of the Parallel Sysplex capability of IBM's mainframe family makes shared-disk clustering viable for DB2 (and IMS) databases. In particular, the coupling facility and DB2's robust optimization technology helps to enable efficient shared-disk clustering. Mainframes are already very large processors capable of processing enormous volumes of work. Great benefits can be obtained with only a few clustered mainframes – whereas many workstation processors would need to be clustered to achieve similar benefits.

Shared-disk is usually viable for applications and services requiring modest shared access to data, as well as applications or workloads that are very difficult to partition.

Examples of database systems that implement a shared-disk approach for clustering include DB2 for z/OS (with Data Sharing), DB2 for LUW (with PureScale), and Oracle RAC.

Shared-Nothing

In a ***shared-nothing*** environment, each system has its own private memory and one or more disks. Refer to *Figure 2*. The clustered processors communicate by passing messages through a network that interconnects the computers. In addition, requests from clients are automatically routed to the system that owns the resource. Only one of the clustered systems

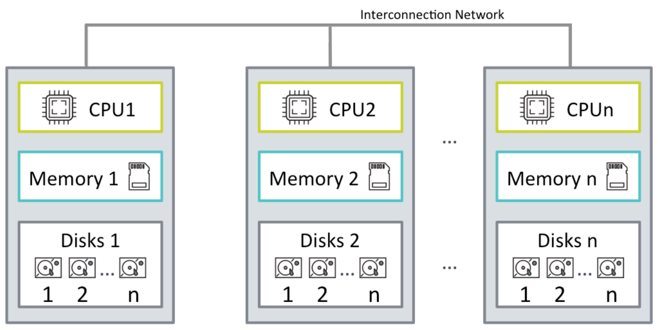


Figure 2. The shared-nothing architecture.

can “own” and access a particular resource at a time. Of course, in the event of a failure, resource ownership may be dynamically transferred to another system in the cluster.

The main advantage of a shared-nothing architecture is improved scalability.

The main advantage of a shared-nothing architecture is improved scalability. In theory, a shared-nothing multiprocessor can scale up to thousands of processors because they do not interfere with one another – nothing is shared. However, in practice, shared-nothing scaling of database systems is implemented on far fewer nodes. The scalability of shared-nothing clustering makes it ideal for read-intensive analytical processing typical of data warehouses.

A disadvantage of shared-nothing is that a partitioning scheme must be designed to apportion the data across the nodes of the database. Data is usually partitioned horizontally by row. This requires identifying a column (or set of columns) to be used to split a table into multiple tables, each with a different subset of the rows that were in the initial table. The scheme may be simple, as when the data matches a segment of the business. For example, partition 1 stores data about the Western region and partition 2 stored the Eastern region data. Or the scheme may be more complicated (such as based on a hash key), especially when there is no easy way to separate the data in a business-relevant manner. Automatic partitioning is known as sharding (and it is discussed in an Database Shardin of this paper).

Remember, the data is not shared, so it must reside on (at least) a single node and the DBMS must know how to partition and access the data based on the partitioning scheme. Vertical partitioning, or splitting a table into subsets of columns is also a possibility.

The shared-nothing approach is based on chopping up the data into smaller subsets because larger single-image databases that are not partitioned can be more difficult and costly to administer and query. Additionally, creating and maintaining a very large database in one place can require high-end, costly computers, whereas partitioning can be accommodated using multiple, cheaper distributed commodity servers.

But partitioning almost always involves trade-offs. One partitioning scheme may work well for certain applications, but another scheme works better for others. There is no universal way to partition data that optimizes all application usage. As usage patterns change and evolve and data volume grows, you may need to re-address the partitioning scheme to better accommodate the data in your database. Repartitioning is a non-trivial

A disadvantage of shared-nothing is that a partitioning scheme must be designed to apportion the data across the nodes of the database.

exercise that is not conducive for 24/7 processing because it requires DBA and programmer effort as well as database downtime causing application outages.

Another challenge arises whenever data must be accessed or modified across multiple partitions. Shared-nothing works well when access and modification is performed only to a single partition, but whenever data from more than one partition is required, complexity arises and ACID compliance can break down.

Depending upon the capabilities of the DBMS being used, the partitioning scheme may allocate data redundantly to more than one node (for failover and availability requirements).

Examples of database systems that implement a shared-nothing approach for clustering include Teradata, MySQL, and many NoSQL and NewSQL offerings. Shared-nothing clustering can be particularly effective for NoSQL databases. This is so for several reasons. For example, many NoSQL database systems do not support ACID, instead relying on eventual consistency, which is easier to implement, but can result in applications and users reading outdated data. Additionally, NoSQL systems typically work on commodity hardware with no built-in high availability features, like a SAN, thereby enabling quorum replication to be used for replicating the data to all pertinent nodes.

Database Sharding

Sharding is often used with a shared-nothing approach to automate partitioning and management. The word "shard" means a small part of a whole. So database sharding is a technique for partitioning databases that separates large amounts of data into smaller, more easily managed parts called shards.

Instead of scaling up, sharding breaks apart data to allow scaling out. Sharding is similar to horizontal partitioning in that it splits tables by row, but the data is partitioned across multiple instances of the schema. The primary benefit of sharding is that the processing workload against a large partitioned table can be spread across multiple servers.

There are drawbacks to sharding. For example, after sharding, instead of having a single database to manage, there are now multiple databases each with its own server, CPU, and memory requirements. Additionally, sharding can negatively impact fault-tolerance. When one shard goes down, the data on that shard is not accessible. That is why sharding is also often accompanied by replication to have a duplicate data set ready for usage in case of failure (Replication).

A preordained method for determining how to shard is required; that is, in which shard each specific row should be placed based on the data and an algorithm that apportions the data to a specific shard, or partition.

Once the data is sharded, each shard lives in a totally separate logical schema instance. This can be across physical database servers, multiple data centers, or even across multiple continents. There is no ongoing need to retain shared access (from between shards) to the other unpartitioned tables in other shards.

Examples of database systems that implement automatic sharding include Apache HBase, Couchbase, and Informix.

Replication

In many cases, database systems that support shared-nothing with sharding also support redundant replicas of data to bolster fault tolerance. Replication involves setting up a separate copy of the data on a different node.

Of course, if this is all that you do then the data will quickly become stale as the original data is processed. To remove this problem, the database system provides a replication engine. When data is changed on the master copy of the data the replication engine ensures that the changes are replicated to other copies. The exact method for replication will differ from system to system, and may require reading the database logs and retrying modifications until they success across all replicas.

Replication across multiple servers can be easy to setup but on-going administration and management will be required. Of course, replication requires additional storage (for each replica), as well as additional I/O and CPU usage to support the data replication process.

A Note on NoSQL Database Systems

NoSQL database systems are growing in popularity for modern workloads such as those required for mobile and cloud applications. A NoSQL database can offer the high scalability frequently required for such workloads. But NoSQL is not a panacea.

As mentioned earlier, NoSQL systems often forgo ACID compliance in favor of an alternative approach known as BASE. ACID is an acronym for Atomicity, Consistency, Isolation and Durability. ***Atomicity*** means that a transaction must exhibit "all or nothing" behavior. Either all of the instructions within the transaction happen, or none of them happen. Atomicity preserves the "completeness" of the business process. ***Consistency*** refers to the state of the data both before and after the transaction is executed. A transaction maintains the consistency of the state of the data. In other words, after a transaction is run, all data in the database is "correct." ***Isolation*** means that transactions can run at the same time. Any transactions running in parallel have the illusion that there is no concurrency. In other words, it appears that the system is running only a single transaction at a time. No other concurrent transaction has visibility to the uncommitted database modifications made by any other transactions. To achieve isolation, a locking mechanism is required. ***Durability*** refers to the impact of an outage or failure on a running transaction. A durable transaction will not impact the state of data if the transaction ends abnormally. The data will survive any failures.

NoSQL systems can scale up by relaxing ACID requirements. When atomicity is eliminated you can reduce lock waits, by dropping consistency you can scale up writes across nodes, or by dropping durability you can remove disk latency.

Most NoSQL database systems follow the BASE model, where BASE stands for Basically Available, Soft State with Eventual Consistency. ***Basically Available*** means that there is no guarantee of any specific piece of data being available, but the system will respond to any request. In a ***Soft State*** system changes are constantly happening. But the data you retrieve at a given point in time may eventually get over-written by more recent data. ***Eventual Consistency*** means that there will be times when the database is in an inconsistent state. When multiple copies of the data reside on separate servers, an update may not be immediately made to all copies simultaneously. So the data is inconsistent for a time, but the database replication mechanism will eventually update all of the copies of the data to be consistent.

Some applications can tolerate inconsistent data, but many cannot. For this reason, a few NoSQL database systems provide ACID compliance as an option, and others are working to support ACID capabilities. Of course, this places the same restrictions on the NoSQL database systems that are on other ACID-compliant systems, such as relational database systems.

NoSQL systems are also different from relational database systems in that they do not always offer SQL for data access and modification. But SQL is a standard across the industry and most users wish to continue to use SQL for data access and modification. And so, once again, many NoSQL database systems are adding SQL support; albeit, not complete ANSI SQL support in many cases.

Elastic SQL: An Alternative, Services-Based Approach

So far we have examined the status quo, wherein various architectural alternatives of achieving improved elasticity and scalability using existing solutions are offered. But each of these methods has some significant drawbacks: shared-disk requires expensive hardware and software to achieve, shared-nothing requires partitioning the data that may not match the requirements of all applications, and NoSQL eliminates ACID and often, SQL. Clearly an alternative approach is needed – one that does not try to work around the shortcomings of the existing solutions.

Many organizations desire a database platform that maintains industry-standard SQL and ACID for data integrity, but with an architecture that offers the flexibility and agility to elastically scale up and down as needed. Fortunately, an elastic SQL database system delivers a compelling, services-based alternative to traditional approaches that were spun out of a client-server mindset. There are a number of different approaches to elastic SQL – from a variation on synchronous replication that relies on global timestamps to a two-tiered approach that uses multi-version concurrency control. Since the former relies heavily on minimal latency to operate, we'll focus primarily on the latter.

There are several aspects of an elastic SQL approach that enable it to deliver scalability while eliminating many of the drawbacks of the traditional approaches.

An Example of Elastic SQL Architecture

Consider for example, a two-tier architecture that splits the transactional tier and the storage tier. This is not dissimilar to what you might find in Hadoop, where compute processing is independent from storage. The transactional tier provides an in-memory, on-demand cache distributed across multiple servers and potentially even geographically dispersed data centers. The storage tier uses a set of peer-to-peer coordination messages to manage commit processing and access data when it is not available in the transactional cache.

This two-tier architecture delivers ACID compliance but separates data durability (the "D" in ACID) from the transactional processing (which handles the "ACI" in ACID). Such a scalable architecture is neither as sensitive to disk throughput (like shared-disk), nor does it require explicit partitioning and sharding (like shared-nothing).

Because transaction processes are deployed separately from storage processes, they each can be scaled independently of the other. When throughput increases, additional transaction processes can be started very quickly – even in new geographies. Upon adding another transaction process, the system authenticates it for processing and balances the workload. This allows for seamlessly scaling to use the additional transaction processing capability with no outage. Natural data affinity builds in the caches based on usage patterns, speeding performance. If workload decreases, transaction processes can be stopped, and the system will rebalance the workload across the remaining, operational transaction peers.

The biggest benefit of the Elastic SQL approach is that it delivers transactional consistency and durability with industry-standard SQL and elastic scale-out simplicity.

Similarly, more than one storage process can be started. Doing so results in multiple, independent, durable copies of the database. When a new storage process is started, the system automatically synchronizes it with the active database. In some deployments of elastic SQL, disk I/O can be managed to better support intensive modification requirements. By determining how much of the data set is controlled by each durability peer (i.e. a subset of the database rather than the entirety) and how many data copies are managed, you can improve disk I/O in a manner invisible to the application.

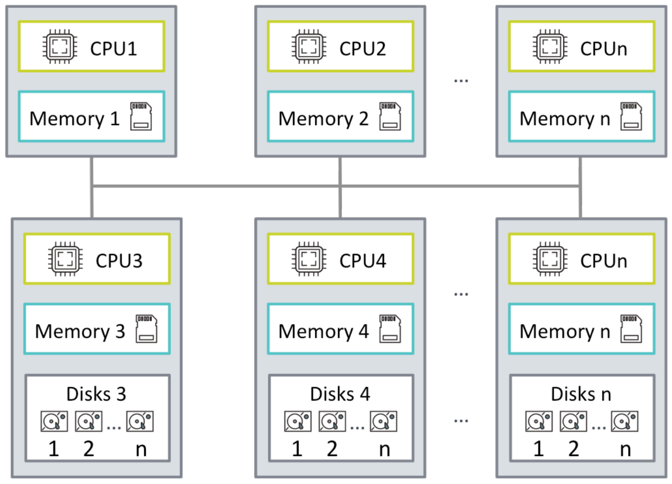


Figure 3. The Elastic SQL architecture.

Refer to *Figure 3* for an overview of the elastic SQL architecture.

Since data can be cached in one or more locations, data needs to be self-replicating to ensure consistency across all of the independent transaction and storage tiers. If a peer needs data that is not in its own cache, it requests it from the nearest peer that has it. Disk I/O is only required when no other peer has the needed data or during an update, insert, or delete. Each peer knows the other locations of each piece of data stored in its own cache and can therefore coordinate consistency.

In such an architecture, database performance, latency, redundancy, and availability can be adjusted by bringing peers online and offline as needed, with no downtime. No single peer is wholly responsible for a given task or piece of data, so any task can fail or be purposefully shut down without impacting service availability.

Coordination of read/write access to the system is enabled using MVCC, or Multi-Version Concurrency Control. MVCC works by treating all data as being versioned. This means that all modification operations create new versions of the data. Each transaction process caches multiple versions of any given object with new versions pending until the associated transaction commits. Because data is never changed in-place, updates can be handled optimistically; a rollback simply drops the pending modification from the cache. By handling lock management at the row (or group of rows) level, the database can reduce conflict and interference. As long as the lock manager is not fixed, it can shift location as access patterns change.

The biggest benefit of the elastic SQL approach is that it delivers transactional consistency and durability with industry-standard SQL and elastic scale-out simplicity. Although there are many benefits to an elastic SQL architecture, there are, of course, drawbacks. The strength of Elastic SQL systems today is in OLTP workloads. That said, it is possible to use Elastic SQL for Hybrid

Transactional Analytic Processing (HTAP); that is, to simultaneously perform both OLTP and real-time analytic queries without interfering with each other by dedicating separate transaction processes for each. Since there are different architectural approaches to achieving Elastic SQL, additional limitations are vendor-dependent.

NuoDB is an example of an elastic SQL database system. Others include Google Cloud Spanner and CockroachDB, which both rely on global timestamps for consistency coordination. When evaluating Elastic SQL database systems, be sure to compare and contrast the SQL capabilities (not all provide the same coverage of SQL) and evaluate successful customer use cases that match your needs.

Table 1 offers a summary of the capabilities of shared-disk versus shared-nothing clustering versus elastic SQL, and outlines the basic pros and cons of the three architectural approaches discussed in this white paper.

Shared-Disk	**Shared-Nothing**	**Elastic SQL**
Adaptability to changing workloads; load balancing	Can exploit simpler, cheaper hardware	Uses simpler, cheaper hardware to adapt to changing workloads. Automated load balancing
High availability; failover	High scalability; more downtime	Continuous availability; resilience to failure; native support for active-active deployment; rolling upgrades; high scalability
Simple ability to scale-in	Data partitioning incompatible with scale-in	Simple ability to scale-in
Performs well in a heavy read environment	Works well in a high-volume, read-write environment	Works well in both heavy read and mixed read-write environments
Data need not be partitioned	Data must be partitioned across the cluster	Data need not be partitioned, but can be grouped for improved I/O throughput

Table 1. Shared-Disk Versus Shared-Nothing versus Elastic SQL

Summary

Scalability and elasticity are important qualities for database systems in the age of digital transformation and the cloud. Both shared-nothing and shared-disk architectures have been around for some time and support scalable systems with varying levels of elasticity. Be sure to understand the pros and cons of each of these methods before implementing them at your shop.

Additionally, it is always wise to learn about new capabilities and offerings, such as the Elastic SQL approach to scalability that eliminates many of the drawbacks of the earlier approaches.

About the Author

Craig S. Mullins is a respected database expert and researcher who is president & principal consultant of Mullins Consulting, Inc. He has worked with multiple database systems and technologies and his experience spans multiple industries.

Craig is the publisher/editor of The Database Site and he also writes for many computer/IT publications. His technical articles can be found in many IT/database journals and web sites including Database Trends and Applications, TechTarget, and others.

Craig is a frequent speaker at IT conferences, having spoken about database issues to thousands of professionals at conferences including Data Summit, World of Watson, IDUG, SHARE, DAMA Symposium and Oracle World. He has spoken at events on 4 continents (North America, Europe, Asia, and Australia).

Mullins Consulting, Inc.
http://www.mullinsconsulting.com

(281) 494-6153

© 2017, Mullins Consulting, Inc.