WHY TRADITIONAL SQL DATABASES FAIL TO SCALE WRITES & READS EFFECTIVELY

And How to Successfully Solve

Marc Staimer, President & CDS Dragon Slayer Consulting

DRAGON SLAYER CONSULTING EBOOK | *Sponsored by* CLUSTRIX

Executive Summary

Traditional OLTP database scaling is a major polarizing issue with both database administrators (DBAs) and application developers. Many people call a relational database to be SQL Database and say that SQL databases fundamentally cannot scale. They say it is just not possible because SQL databases were not designed to truly scale, especially writes, and definitely not to cloud scale. Others will assert that those who believe SQL databases cannot scale lack the knowledge, experience, and expertise to actually scale SQL databases. Some DBAs say that's why there are now NoSQL databases.

There is one relational database with MPP architecture (HP NonStop SQL) that provides OLTP scalability, and there are relational databases for Data Warehousing and OLAP with MPP architecture that scale near linearly such as Paraccel, Teradata, Greenplum, and Vertica. All are excessively expensive and some do not run on commodity hardware.

So, is traditional SQL database scaling for online transaction processing (OLTP) and online analytics processing (OLAP) an issue? The answer is a definitive yes. The crucial contention will be around what is meant by the term "effectively". Answering that question requires a more thoughtful discussion as to the market requirements for SQL database scaling, common workarounds to known problems, SQL database availability requirements, database application malleability, and organizational tolerance for manually labor-intensive sweat equity. In other words, the framing of the problem determines the answer.

People have taken SMP (single server, multi-processor) SQL databases and made creative attempts in scaling them. There are DBAs that can make an open source SQL database such as PostgreSQL or MySQL sing, and others who can do the same for Microsoft SQL Server™ in what can only be called a scaled implementation. They leverage one or more SQL database workarounds such as sharding, read-only slaves, master/master replication, linked servers, distributed partitioning, as well as data dependent routing. All of them tend to cost a heck of a lot more than anticipated with serious operational stipulations as well. For example: the implementation is often limited to a specific type of use case; or it lacks elasticity (automated ability to dynamically adapt to changing demands); or has a single point of failure; or there's a ceiling on how high it can scale; or there is substantial downtime; or it can't provide real-time, live-data analytics; or lacks an operational dashboard. There always seems to be some downplayed caveat.

DBAs and application developers want their SQL database to scale well beyond their current needs: simply (no expertise or contortions required); transparently (to the applications); **reliably** (no downtime); **elastically** (allocates available resources on demand, plugs and plays with their current applications/scripts); and especially **cost effectively** (pay-as-you-go approach). This white paper will examine why most SQL databases have problems in achieving those requirements with the most common workarounds, and, in addition with some of the newer workarounds such as Docker containers and Cloud based DBMS services. It will then assess how ClustrixDB, a scale-out distributed clustered SQL database, meets those requirements and overcomes current marketplace problems.

Table of Contents

Describing The SQL Scalability Problem

Why is SQL database scaling a problem? It's a problem because the market says it's a problem. Dragon Slayer Consulting has surveyed nearly four hundred small to medium IT organizations. These conversations consistently expose that database scalability in their organization is a frustrating problem as more and more applications are relying on SQL databases. That doesn't mean it's impossible to scale their SQL database. It just means they are having severe issues doing so. Although the specific SQL database architecture is the principal factor in the cause of their frustration, they rarely realize it. What they identify are the symptoms. They report it as a lack of knowledge, experience, or skill; or financial constraints (i.e. it costs way too much--more on costs shortly). Frequently, they complain about the time sink caused by extensive on-going manually labor-intensive interventions. And sometimes they see the scale problem as a database uptime issue.

Common Workarounds Pros and Cons

DBAs are by definition smart, clever professionals. They don't just sit back and throw their hands up and say "No MAS!" They will attempt to solve the problem. One of the first things a DBA will try to do to solve their scalability issues is to scale-up (versus scale-out).

Scale-Up

Scaling-up is a relatively easy thing to do. Scale-up is the process of moving to a bigger, more powerful server. It's the simple step of moving from a smaller server with 4 or 8 processors to something significantly more powerful such as 64, 96, 128, or 256 processors with more memory, IO, bandwidth, etc. The SQL database, especially an SMP architected database, is scaling by throwing hardware at it. When the current SQL database noticeably slows, and the hardware can no longer keep up, purchasing a bigger server with more processors and more memory is positioned by the SQL database and server vendors as a no-brainer solution. They claim there is rarely any need to significantly change the database as long as the server architecture is the same.

Purchase the database license key for the bigger server (if it's a commercial database or purchase the bigger support/maintenance contract for an open source database), install the database on the bigger server, then run it the same way as previously with more processing and memory to handle the heavier load.

So what's wrong with this type of scaling solution? If money is no object, and you're not already on the biggest available server, it appears as if this can be a pretty good scalability solution; but then again, when is money ever no object?

Bigger servers are not exactly inexpensive. Pricing per processor tends to rise as the number of processors scales. While counter-intuitive the price goes up because of increasing server complexity. That complexity and cost grows even higher if no single point of failure is required, which it normally is, and should be. Database licensing is also typically tied to cores or sockets (CPUs) making the database in that "bigger" server significantly more expensive. Even the supported open source SQL database support and maintenance costs increases with server size. Then there is the eventual cap problem. All servers eventually run out of steam no matter how big they are today. Also, an SMP box cannot be bigger than certain number of cores and certain amount of memory. One cannot a buy an SMP box with infinite number of cores and memory. So, even if someone wants to spend money one eventually hits a ceiling. And what happens if the SQL database is in the largest server available at that point in time? Further "scaling up" is not an option.

Which leads to the problem of risk mitigation. To forestall that eventual cap, many DBAs will attempt to purchase the largest server they can get. This reduces the number of labor-intensive migrations but also raises the server's capital expenditures (CapEx) and operating expenditures (OpEx) as well as the software licensing (or subscription), maintenance, and support costs. There is no such thing as a free lunch.

What makes the scale-up option worse is that it is never as easy to implement as the assertions. Upgrading the server is a convoluted and complicated process. It customarily requires an awful lot of manual labor-intensive tasks from the DBA, server admin, network admin, storage admin, facilities admin, etc., resulting in substantial downtime. Downtime is painful at best, and often impossible to bear because of lost revenues and productivity. The escalating total costs make this option untenable for many IT organizations. Also, one should note that SMP boxes do not scale-up linearly due to their shared memory and shared disk architecture. Furthermore, resource contention increases as you add more and more cores to the box and scalability tapers-off after a certain number of cores.

The next most common option many DBAs will attempt is SQL scale-out. Here's a look at each scale-out option.

Scale-Out

Scale-out is a horizontal scaling strategy where more servers equals more database performance and more database capacity. The ideal scale-out SQL database and infrastructure should be completely and totally transparent to the applications. There should be no single points of failure (SPOF), and in fact the database should be able to survive multiple concurrent failures (storage drives or server nodes) without losing data or causing the database to fault. Such scalability can be achieved only with loosely coupled and shared nothing architecture (also known as MPP architecture).

Since building MPP system are difficult and time consuming, users have started evolving alternate but less efficient methods such as Sharding and alternate ways of achieving scale-out.

Scale-Out: Sharding

Sharding requires multiple database instances (hence the scale-out) and is theoretically highly scalable. "Highly scalable" is more theory than reality since sharding is as much art as it is science.

There are two types of sharding. Horizontal sharding is the process of dividing the dataalong a specific application boundary among multiple database instances. So dividing user names by their alphabetical order, last names starting with A through H, I through Q, and R through Z go on different database instances is an example of sharding. Another could be allocating them by their id number to the different database instances based on a numeric range. Horizontal sharding is not a simple process nor for the faint of heart. It requires a deep understanding of the application, careful planning, detailed integration execution, as well as a thorough alignment between the partition scheme, database schema, and types of queries that are made. The application almost always has to be modified and the application layer becomes responsible for ACID (Atomicity, Consistency, Isolation, Durability) compliance requirements.

Vertical sharding is somewhat different than horizontal sharding in that it moves whole tables to other database instances. It's similar to horizontal sharding in complexity and labor intensity. Application changes are required and avoiding cross database queries either requires redundant data or a master/slave relationship.

As previously mentioned, sharding is as much art as it is science. DBAs typically feel a sense of accomplishment and relief once the sharding process is complete. It is a non-trivial task. When it works as designed there is pride especially if it doesn't break during production.

Unfortunately, there are many challenging problems with sharding. First and foremost, sharding is time consuming, error-prone, incredibly laborious, and requires complicated manual load balancing, failover, recovery, and failback. Was time consuming mentioned? After it's complete, it is not dynamic. DBAs report changes are frequently put off. It's pretty easy for the various database instances to become imbalanced. There are hotspots and capacity partition limitations. Cross database query/joins are highly inefficient. And consistent backups across the entire dataset are exceedingly difficult, increasing the risk of data loss. Restoring the entire dataset to a specific point in time is nigh impossible.

Relationships in a sharded database are lost between the database instances, destroying much of the database value. That means those relationships have to be reconstructed in the application utilizing that sharded database. The application almost always has to be modified as well, and that is rarely a trivial task. Consistency between shards can't be maintained by the RDBMS since there is no master RDBMS in charge of the shards. The application layer becomes responsible for ACID compliance requirements.

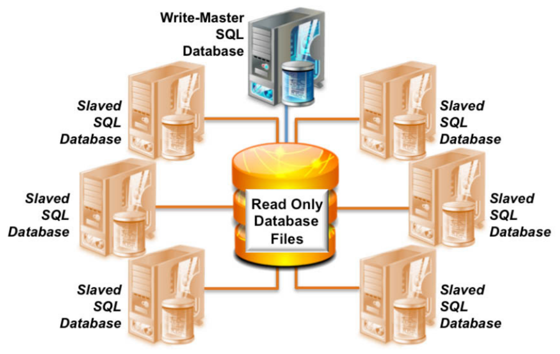
Sharding in general is also not as inexpensive as it first seems. That loss of database relationships as previously mentioned may require application modification, more instances, more servers, more, infrastructure leading to higher CapEx and OpEx once again. Sharding has lots of single-points-of-failure. Sharded SQL databases can rarely survive one outage let alone multiple. Additionally, the sweat equity is extraordinarily high on an on going basis for any kind of dynamic environment. In other words, the costs are higher than expected.

Even the latest MySQL Fabric does little to address the vast majority of sharding shortcomings. It adds failover or HA. And it partially automates a few sharding labor-intensive processes when it comes to adding data. It's a very limited first step.

Another scale-out option DBAs will attempt is read-only slaves.

Scale-Out: Read-Only Slaves

Read-only slaves are reasonably simple to implement, which is why they're one of the more common scale-out implementations for large MySQL, PostgreSQL, and Microsoft SQL Server™ installations. It starts with a master database that replicates to a series of slave databases. The number of slaves varies, but they are ultimately limited by the master database's ability to replicate in a reasonable amount of time without affecting the write performance. Those slaves are also for reads only. All reads are routed to a read-slave database based on policies while the writes are dedicated to the master database then replicated or mirrored to the read-slaves.



Read-only slaves generally have nominal set-up, maintenance, and application headaches. This technique is in fact transparent to the applications, which is one essential reason why it's so popular.

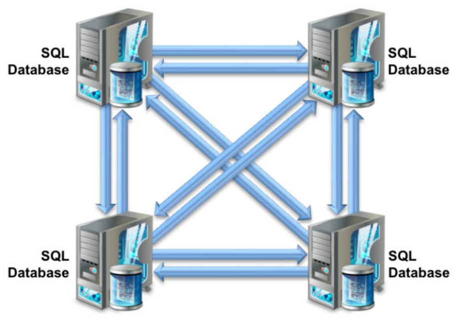
But read-only slaves cannot overcome the bottleneck of the master SQL database especially for write-intensive applications. The master is still a single database limited by its hardware architecture. Granted, that database does not have to service the reads extending the life of the master databases performance curve. However, it still has limits in both performance and capacity. Since the read-only slaves are complete copies of the database, write frequency of that master database affects performance. More frequent writes equals more frequent updates/mirrors to the read-slaves. They're also each subject to the same hardware architectural limitations as the master database. Then there's the single-pointof-failure (SPOF) issue. That SPOF can be disastrous when performing maintenance, updates, patches, and especially when there is a master database failure. It requires DBA intervention to promote one of the slaves to the master during the master's outage. Although this has gotten simpler (AWS RDS console, smart load balancing, MySQL fabric, etc.) ensuring all of the IPs are correctly re-directed can typically be a bit tricky and complicated. And of course after the outage, the process must be reversed taking another system outage.

Read-only slaves consume a lot of storage capacity raising the costs significantly. Each read-only slave requires 100% additional storage overhead. That's 100% more storage CapEx and OpEx. Proponents of read-only slaves assume database copies are cheap. They're not.

An alternative variation of database read-only slaves is master/master replication.

Scale-Out: Master/Master

Master/master is usually utilized when the SQL database is updated relatively infrequently. Whereas read-only slaves work from a single master copy of the database, master/master with replication works from multiple database copies. Each database engine manages and maintains its own copy of that database. Replication is required to update the other database engines when a change has been made.



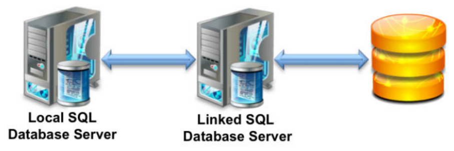
It's easy to see where this type of scale-out can and does have problems. How are the inevitable conflicts resolved? This is a complicated and difficult problem to overcome, which is why it's primarily used with databases that have infrequent updates. When updates are frequent there is a much higher probability of application errors. One clever and complex way DBAs attempt to eliminate or at least mitigate conflicts is to implement stewardship. Stewardship limits what data can be updated in a specific database engine. Even though all database engines can see all data they can only alter the portion for which they have responsibility. Conflicts can be avoided by stewardship but, once again, there are quite extensive manual, labor-intensive requirements of the DBAs (such as a deep understanding of the applications) and there are multiple single-points-of-failure. Scalability for specific datasets is still limited by the underlying server infrastructure. All of which increases CapEx and OpEx costs beyond the level of the gains in scale.

When stewardship is not practical because more than one database engine is required to update the database, merge replication can be utilized to manage conflicts. Merge replication requires a lot more resource overhead than stewardship, and it doesn't eliminate any of the issues and limitations of stewardship. Merged replication can and will noticeably reduce database performance as database change rates surge and conflicts increase.

A fourth scale-out workaround is linked servers and distributed queries.

Scale-Out: Linked Servers and Distributed Queries

Linked servers and distributed queries is the ability of a local database to query remote databases as if they were part of the local database. To the applications the local and remote databases look like a single database. Database update frequency has little to no impact on linked servers. This scale-out technique is employed when the databases can be split into functional areas requiring minimal coupling or when splitting data by type. But there are significant issues with this scale-out technique.



Referential integrity constraints are frequently a problem with linked servers making it vital those local and remote databases minimize data relationships. Each time a query spans across to the remote database there is additive latency from the round trip on the network plus the remote database processing. That additional latency lengthens response time with increased volatility. Smart, effective database designs should aim at minimizing latency, making linked servers difficult to pull off successfully.

Use of linked servers is not for the inexperienced DBA. It requires a ton of DBA expertise, meticulous database design, deep knowledge of application query patterns, and more. Linked servers are more often utilized as an enhancement to another scale-out strategy such as read-only slaves or master/master replication.

The fifth scale-out workaround is distributed portioned views.

Scale-Out: Distributed Partitioned Views

Distributed partitioned views (DPV) enable the SQL database to deliver (supposedly) transparent scale-out of partitioned data to the applications. A table's data is partitioned among tables in numerous distributed databases based on a partitioning key. Update-intensive applications are the best target for DPVs. In fact DPVs are the only scale-out workaround providing good query performance for update-intensive applications such as OLTP. This is because typical transaction updates have minimal impact on very few rows. This in turn means they are likely to execute on a single database.

DPVs work with check constraints (references) to direct query execution only on the specific SQL database containing that type of data. If the query lacks the check constraint it will require a lot more time as it is executed across all of the database partitions. DBAs attempt to avoid very many queries requiring data movement between database partitions. Those tend to slow responses down considerably (i.e., gums up the works.) DPV implementations come down to how partionable is the data. Data with good partitioning keys that evenly partitions the data and minimizes cross-partition queries   are good candidates for DPV. If there aren't very good portioning keys, etc., DPV is not a good scaling choice. Up-front analysis and planning on each application is an absolute necessity to find those effective partitioning keys. This makes DPVs painstaking to implement. Although DPVs are supposed to be application transparent, real-world experience paints a very different picture. Some application changes will likely have to be made to efficiently utilize the DPV. What can make DPV truly an exercise in frustration is attempting to determine a partitioning schema that works for many different applications. That is highly unlikely and reduces DPVs to more specific or vertical applications, curbing its usefulness.

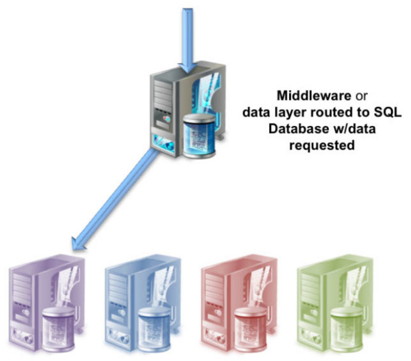
One other thing, data coupling doesn't really affect a partitioned view. But when there are numerous unpartitioned tables with a high level of coupling, data distribution becomes quite trying. As with most scale-out workarounds, DPV is not for the inexperienced.

The sixth scale-out workaround is data dependent routing.

Scale-Out: Data Dependent Routing

Data-dependent routing (DDR) partitions the data to multiple databases while placing responsibility on the application or middleware to route the queries to the correct database. By definition DDR is not transparent to the applications and requires the DBA to have a deep understanding of the applications or the implementation of middleware services to effectively utilize. The premise behind DDR is that the application should have a deeper understanding of the data and therefore will make a better query routing decision. Even if the reality invalidates that premise, DDR is still a very popular database scale-out option especially when that scale-out is distributing and processing across hundreds or even thousands of database servers.

DDR is also really quite complicated. Besides requiring in depth application knowledge and programming skills, it demands strong architectural skills. When there are dozens, hundreds, or thousands of database servers, the queries should be directed to the correct single server. This means the database partitions must be architected so that all the data required by a query or update is located on the same database server. The applications or middleware must be architected to access the data stored based on the data requested. DDR like DPV is for high transaction type applications such as OLTP. It is considered the best option by many DBAs for mega scale-out transaction volumes.



But there are serious complex issues with DDR. DDR demands sophisticated expertise in SQL database construction and partitioning as well as an equal or better level of sophistication in regards to the applications or middleware. Each and every application must be customized to utilize DDR or expensive (high CapEx and OpEx) middleware software must be licensed or developed. That customization is a major time sink. And that cost is not just for implementation, either. DDR has to be maintained, patched, modified, upgraded, documented, quality assurance tested, and retested for every change in the ecosystem. These tasks are non-trivial. They are also non-flexible, non-dynamic, and non-elastic. DDR requires a lot of planning and overprovisioning. All of which adds to the total cost, which in the end, is always higher than expected or planned.

Do any of these workarounds meet the ACID requirements of databases? ACID is what the vast majority of DBAs depend on when evaluating database and application architectures. However, based on production realities, the answer more often than not is "no". Then what about NoSQL databases?

The seventh scale-out workaround is NoSQL databases.

Scale-Out: NoSQL Databases

NoSQL databases are becoming pretty popular. They scale out nearly linearly in both performance and capacity. They're really exceptional at data ingest and have very flexible schemas that are relatively easy to implement, operate, and manage.

Yet, they are not designed to solve the same problems as SQL databases. NoSQL databases are being primarily utilized for large datasets and non-real-time data. NoSQL databases are quite good for analytics and data warehousing while being frustratingly slow for transaction-based applications such as OLTP. Analytics on unstructured data (data not easily digested into rows and columns) is another NoSQL strength. But when is comes to transactional high performance or data that changes frequently, NoSQL databases come up short. They lack real-time dashboards and reporting but are good for historical reporting and big amounts of data. Good for analyzing historical data, not good for scanning and analyzing data as it's changing right now. NoSQL databases often lack the system management tools DBAs are used to.

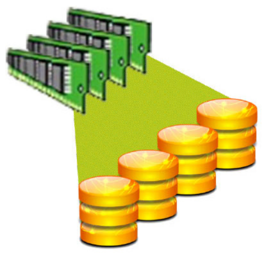
In essence, the NoSQL database technologies are generally good for big unstructured data, and not so good for transactional data or ad-hoc queries, nor are they a good substitute for SQL databases. They also don't pass the ACID test.

Newer SQL Database Scaling Workarounds: In-Memory, Docker Containers, and Cloud-Based DBMS

In-Memory

Running relational databases in-memory (IMDB) is also known as main memory database (MMDB), or memory resident database (MRDB). This SQL database workaround is a hardware solution to a software problem and doesn't truly solve SQL database scalability issues especially writes. IMDB are faster than disk or flash-optimized databases. IMDB delivers more predictable latencies and response times.

As the database scales so do the DRAM requirements. DRAM is volatile so the SQL database can be lost or corrupted with a power surge, power drop, or power loss. That volatility makes IMDB non-compliant with the durability portion of the SQL database ACID requirements. Fixing that compliance demands that the DRAM to be made non-volatile. Non-volatility comes in different forms: battery backup, super-capacitors, NAND flash backed, or complete NAND flash DDR4 DIMMs. All solve the ACID durability compliance issue at a very high cost. And in the end, they are limited in expanding a SQL database's scalability. Eventually, the SQL database becomes the bottleneck itself.



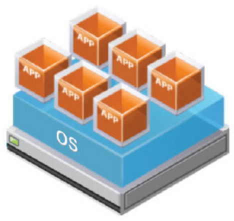
The root cause of this bottleneck is found in the underlying architecture of SQL databases. Every user operation causes the SQL database to launch multiple tasks. Take, for example, a common query. The SQL database has to parse it, optimize it, execute it, manage the transaction logs, maintain transaction isolation, and row level locks. Throwing memory at the job seems to intuitively make sense; except based on extensive conversations between DSC and several dozen DBAs, reality is somewhat different. In several cases that additional memory and process constructs such as buffers do not benefit and actually produce negative results. Keep in mind that the architecture of every SQL database is somewhat different.

In the case of the highly popular MySQL, increased buffer size because of increased memory, actually leads to a astonishing performance decrease by as much as 10x! Scaling the query cache or the ON DUPLICATE KEY UPDATE from the increased memory counter-intuitively also degrades performance in a similar manner. For query caching it is especially degraded if there are more queries in the cache. For ON DUPLICATE KEY UPDATE, it becomes more pronounced when working on larger tables particularly when it requires updating of more duplicated rows. There are similar IMDB performance degradation issues for several other popular MySQL functions. And these issues are not unique to MySQL either. There are similar problems with MemSQL and VoltDB.

At some point SQL database architectures will have to change to take definitive advantage of the additional available memory. As new lower cost non-volatile memory options become available a few years down the road, it will become more imperative. However, in today's market, IMDB has limited value for most SQL databases. Throwing expensive memory at the SQL database scaling issue can mask some scalability issues while exacerbating others. It's a temporary salve at best, not effective at worst, and not a complete or permanent SQL database scalability solution.

Docker Containers

Docker containers have become another tool for many DBAs to implement the scale-out workarounds previously described. Unlike virtual machines (VMs), a Docker container does not include the virtualization of the operating system (OS). Docker containers only virtualize the application, or in this case, the SQL database. This allows multiple instances of that SQL database to be spun up in just a couple of minutes. Since Docker containers do not require a hypervisor, each Docker container SQL database instance has as much as an order of magnitude less overhead than a VM based instance. Now because most database vendors do not charge additional licenses or support fees for multiple SQL database instances on the same OS today (they do charge for VM instances which also virtualize the OS), it reduces scale-out costs.



There are several things wrong with that picture. The first is obvious. Vendors will fix their licensing and support fee models in short order. The second is that running Docker container SQL database instances does nothing to fix the previously discussed inherent architectural issues with the scale-out workarounds. Docker does in fact provide an easier methodology in deploying multiple instances, making evaluations and repeated development deployments more useful. But it doesn't resolve the underlying scale problems for production workloads. And linking the different instances can add significant complexity to the SQL database architecture.

Cloud-Based SQL DBMS

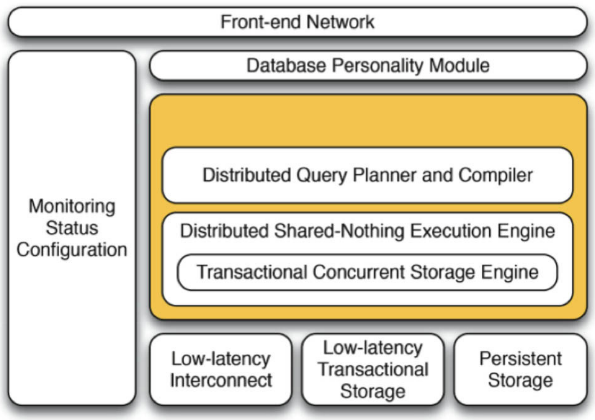
Many DBAs believe they can solve their SQL database scaling problems by going to a cloud database service. There are relational database cloud services, NoSQL database cloud services, DBMS cloud services based on MySQL, Oracle 12c, MongoDB, DB2, and more. However, just by putting the SQL database into the cloud does not make it any more scalable. It doesn't solve the problems of scaling writes. Many of these services have specific caveats on write performance. Amazon AWS has several cloud database services. None of them including Aurora™, RDS™, ElastiCache™, Redshift™, or even their NoSQL service DynamoDB™ solve the SQL database write scaling problems. They do ok for read scaling via read replicas, but nothing for scaling writes. The underlying SQL database architectures do not change just because they are now located in a cloud service provider's data center. If the SQL database doesn't scale before it goes into the cloud, it will not scale after it's there.

Circling back to the original issue. How is it possible to effectively scale the reads and especially the writes of SQL databases without compromising on performance, without requiring an architectural PhD in SQL databases and applications, without single-points-of-failure (SPOF), and most importantly, doing so cost effectively?

This is where ClustrixDB comes in.

How Clustrix Addresses the SQL Database Effective Scalability Problem

Clustrix started with a clean sheet of paper, a profound and deep understanding of the SQL scale problem, mixed in with a strong determination to make SQL databases Internet or Cloud scale. They applied cloud computing elastic scalability principles to designing a new SQL database. Doing so enabled Clustrix to build a unique clustered "shared nothing" scalable SQL database called ClustrixDB. ClustrixDB is software, which runs in white box commodity server hardware, virtual machine, or as a cloud instance. Each physical host, virtual machine, or cloud instances is a node that contains a:


Front-end Network
Database Personality Module
Monitoring Status Configuration
Distributed Query Planner and Compiler
Distributed Shared-Nothing Execution Engine
Transactional Concurrent Storage Engine
Low-latency Interconnect
Low-latency Transactional Storage
Persistent Storage

- **Query Compiler:** distribute compiled partial query fragments to the node containing the ranking replica.
- **Data Map:** all nodes know where all replicas are, and current ranking replicas.
- **Database Engine:** all nodes can perform all database operations (no leader, aggregator, leaf, or data-only, etc. nodes).
- **Data Table Slices:** All table slices (default: replicas=2) auto-redistributed by the Rebalancer.

The ClustrixDB feels like off the shelf SQL databases DBAs have used for years but without the scale out limitations.

The ClustrixDB Solution

Near Linearity

It is that "shared nothing" clustered elasticity that permits ClustrixDB performance to scale near linearly from a minimum of 3 nodes to 50 and more, in a single database image cluster. Each additional node in a cluster adds more capacity and performance. ClustrixDB produces high performance for reads, writes, mixed workloads, low to extremely high concurrency, and from simple OLTP queries to extremely complex OLAP queries. Sharding or any of the other previously discussed complicated SQL workarounds is never required, ever. It's able to do this because each node handles writes and reads. Data is automatically rebalanced across a cluster. Applications always see just a single SQL database instance, period. ClustrixDB's high level of parallelism meets or exceeds most fast ingestion requirements as well.

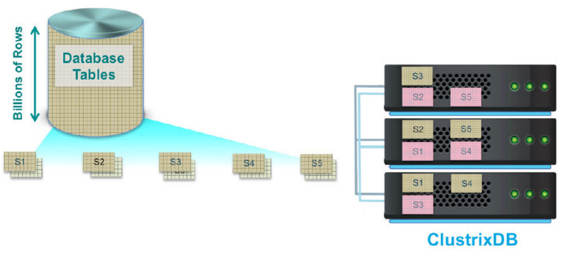
Theoretically, ClustrixDB has no known scalability limits. Pragmatically, it comes down to testing. What this means is that the database and application design need be done once and only once.

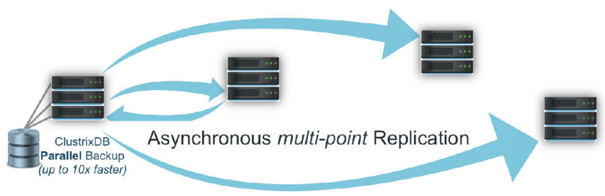
No SPOF

The ClustrixDB smarter data distribution comes with built-in fault tolerance. Should a server node fail or go off line for any reason (partial node, full node, drive, CPU, cable, transceiver, switch, etc.), data is automatically rebalanced across the remaining nodes. No disruptions or outages occur to the database or applications using that database. The result is no single point of failure (SPOF).

Extraordinary Availability

Greater data availability comes from the ClustrixDB data table slicing. It slices tables per index across all the nodes via distributed hash. Data resilience and durability is enhanced via multi-copy mirroring (MCM) for each of the slices. MCM replicates each slice a minimum of 2 times while spreading them across different nodes in the cluster. That data resilience technique is what enables the database data to survive multiple concurrent server nodes or hardware component failures or one for each replica. Additional replicas increase the number of potential concurrent failures that can occur without data loss. The ClustrixDB previously described autonomic rebalancing and self-healing eliminates any requirements for DBA intervention to deal with those failures.




ClustrixDB Parallel Backup (up to 10x faster)
Asynchronous *multi-point* Replication

The way ClustrixDB protects against site disasters is with high performance parallel backup and asynchronous multi-point replication. Those functions empower the ClustrixDB to replicate to/from any data center or cloud anywhere providing fast and simple business continuity.

Fast, Easy Growth and Tech Refresh

Adding or removing multiple ClustrixDB nodes to or from the cluster is a single action, accomplished by a few clicks. The physical or virtual nodes can be added live online with minimal (typically under a minute) of database 'pause'. Just as important, nodes can be refreshed and retired live and online as well without database downtime or disruption. Software upgrades are accomplished with a similar minimal database 'pause'.

Leveraging Standard Off-the-Shelf Low Latency Networking

Clustrix recommends at least 1Gbps Ethernet for its low latency. ClustrixDB server nodes can be interconnected using standard off-the-shelf commodity 1Gbps or 10Gbps Ethernet TCP/IP networking. On Ethernet networks below 1Gbps latency can be more variable, less deterministic, and result in somewhat lower performance.

Consistent Scale-Out Query Performance

ClustrixDB parallel architecture permits it to handle queries on a distributed basis. Queries are fielded by any peer node and routed directly to the node holding the data. Complex queries are split into fragments and processed in parallel. These fragments are auto-distributed for optimized performance. All server nodes handle writes and reads. The result is aggregated and returned to the user in a consistently fast response time.

ACID Compliant and Fault Tolerant

ClustrixDB has full relational capabilities, as any SQL database would be expected to have. ClustrixDB is completely transactionally ACID compliant. See table below:

ClustrixDB *Atomicity*

ClustrixDB uses a combination of two-phase locking and multi-version concurrency control (MVCC) to ensure ***atomicity***. Whereas this can increase latency if every node participated in every transaction, ClustrixDB avoids that problem through the use of the Paxos consistency protocol. The Paxos consistency protocol compels transaction participants to include the originating node, three logging nodes, and the nodes where data is stored. A single node may serve multiple functions in a given transaction, ensuring that simple point selects and updates have a constant overhead. OLAP transactions compute partial aggregates on the node where the data is located and therefore similarly don't suffer from high overhead.

ClustrixDB *Consistency*

ClustrixDB shared nothing architecture is *immediately consistent*, which is significantly different from NoSQL shared nothing architectures that are only "eventually" consistent. Eventual consistency can produce significant errors that can invalidate or corrupt a relational database. ClustrixDB delivers immediate ***consistency*** by ensuring relational constraints, such as foreign keys, are enforced properly by implementing those foreign keys as triggers and evaluating at commit time. As a result, clients connecting to any node in the cluster to issue queries will see the same data across all nodes.

ClustrixDB *Isolation*

ClustrixDB produces MVCC ***isolation*** at the container level. The ClustrixDB atomicity guarantees that all applicable replicas receive a particular write before it reports the transaction committed. This makes ClustrixDB isolation equivalent to non-clustered MySQL isolation. And unlike other systems like Galera Cluster, ClustrixDB snapshot isolation does use 'first-committer-wins'.

ClustrixDB *Durability*

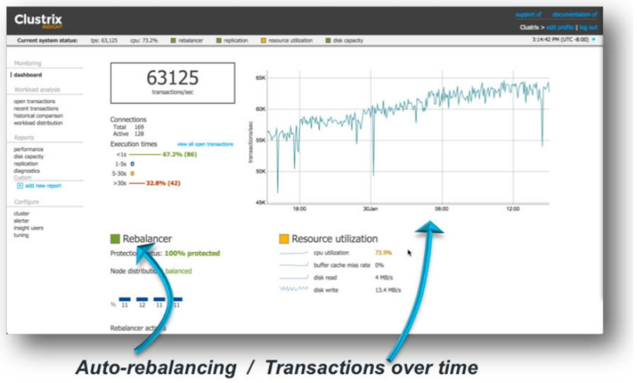
ClustrixDB provides ACID ***durability*** via normal write ahead logging (WAL) as well as replication of relational data within the cluster. Durable consistency is ensured by hash distributing relational data across nodes on a per-index basis as 'slices.' Each slice has a minimum of two replicas (more are optional) spread throughout the cluster as protection against multiple concurrent nodes or drive failures.

MySQL Compatible

ClustrixDB is a drop-in replacement for MySQL. Migrating an application currently running on MySQL to ClustrixDB is very simple, typically requiring little to no change to the application and/or its queries. ClustrixDB supports almost all MySQL data types and can also be dropped seamlessly into any MySQL replication topology (full support for MySQL replication protocol: master and slave, row based and statement based), or for simple deployments and testing. It also works as a master to multiple slaves with different binlogs or as a slave to multiple masters

Intuitive Ease of Use

ClustrixDB doesn't require extensive experience, knowledge or skills manipulating databases to achieve the previously described scalability, availability, and flexibility. It is meant to be intuitive to the vast majority of SQL DBAs and experienced application developers. Any expertise required outside of managing a single SQL database is built into its "DNA". To the DBA, the entire ClustrixDB cluster is managed as a single database no matter how large it grows. Scaling out ClustrixDB is a simple matter of adding new nodes to the cluster with a few clicks. The software automatically re-balances the data. It looks feels and acts very much like a single MySQL database and, as previously mentioned, is drop-in compatible. There are no storage configurations, drivers, or kernel versions to manage. And online schema changes can occur with no locking of tables.



Concurrent Real-Time Operational Analytics

ClustrixDB runs operational reporting with dashboard applications in real time as transactions are ongoing. This is very useful for many markets including Retail (restock high demand items right away), Gaming, AdTech, Internet-of-Things, and Web companies in general (many of which are using MySQL). Any organization that can adjust to real-time information will find this capability indispensible.

Singularly Cost-Effective

ClustrixDB total cost of ownership (TCO) including development, operations, management, etc., is extraordinarily low when compared to other scale-out SQL strategies. ClustrixDB licensing and ongoing costs are a true pay-as-you-go approach. ClustrixDB runs on low-cost commodity white box or name brand servers, networks, and storage hardware, virtual machines, or cloud instances. In other words, the supporting infrastructure costs are minimized.

What's a bit more impressive is that ClustrixDB delivers extensive TCO savings even against free-ware open source SQL databases. Five-year TCO comparisons to open source MySQL sharded environments can typically demonstrate as much as 90% savings even with the assumption that the MySQL has no license or third party support costs. That's quite extraordinary.

When it comes to scalable SQL databases, there is no more cost-effective database today than ClustrixDB.

In Summary

Most traditional SQL databases such as MySQL, PostgreSQL, and even Microsoft SQL Server, have problems scaling out effectively. There are several tried and true SQL workarounds including bigger more expensive database servers, sharding, read-only slaves, linked servers, distributed partition views, and data dependent routing. Regrettably, they all have issues and problems that require application changes, extensive ongoing manual labor-intensive DBA management and troubleshooting and result in single-points-of-failure. They can work, but it is neither easy nor sustainable as the size of these databases continues to expand. Even NoSQL databases are not the answer for OLTP. They scale well but are not a replacement for SQL when it comes to transactional applications or real-time scanning and analyzing.

ClustrixDB solves SQL scalability with none of the issues that come with the workarounds. It's linearly scalable in both capacity and performance. It's highly available, flexible, and intuitive. It is easy to operate, including the expanding and contracting database scale, and feels familiar to the DBA. It delivers real-time operational analytics. And it's highly cost effective.

For More Information

Contact Clustrix at **www.clustrix.com** or via email at **info@clustrix.com**.

Paper sponsored by Clustrix.

ABOUT THE AUTHOR: Marc Staimer, as President of the 17-year-old Dragon Slayer Consulting in Beaverton, OR, is well renown for his in depth and keen understanding of user problems, especially with storage, networking, applications, and virtualization. Marc has published thousands of technology articles and tips from the user perspective for internationally renown online trades including SearchStorage.com, SearchCloudStorage.com, SearchSolidStateStorage.com, SearchSMBStorage.com, SearchVirtualStorage.com, SearchStorageChannel.com, SearchModernInfrastructure.com, SearchVMware.com, SearchDataBackup.com, SearchDisasterRecovery.com, SearchDataCenter.com, SearchServerVirtualization.com, SearchVirtualDesktop.com, SearchNetworking.com, and Network Computing. Marc has additionally delivered hundreds of white papers, webinars, and seminars to many well known industry giants such as: Brocade, Cisco, DELL, EMC, Emulex (Avago), HDS, HP, LSI (Avago), Mellanox, NEC, NetApp, Oracle, QLogic, SanDisk; as well as smaller less well known vendors/startups including: Asigra, Clustrix, Condusiv, DH2i, Diablo, FalconStor, Gridstore, Nexenta, Neuxpower, NetEx, NoviFlow, Permabit, Qumulo, Tegile, and many more. His speaking engagements are always well attended, often standing room only because of the pragmatic, immediately useful information provided. Marc can be reached at marcstaimer@me.com or at (503)-579-3763, in Beaverton OR, 97007.