Getting Started With Distributed SQL

CONTENTS

ANDREW OLIVER
SR. DIRECTOR OF PRODUCT MARKETING, MARIADB

Developers building mission-critical applications that require data integrity, high read/write throughput, and 24/7 global availability with little or no downtime or maintenance windows require a new type of database. Older relational systems fail to meet these needs in terms of scalability, availability, resilience, or performance under load. NoSQL databases do not offer the robust functionality, standard query language, or transactional integrity required for systems of record.

This Refcard aims to acquaint you with distributed SQL database technology, how it works, the problems and types of applications it solves, and how to evaluate different offerings.

ABOUT DISTRIBUTED SQL

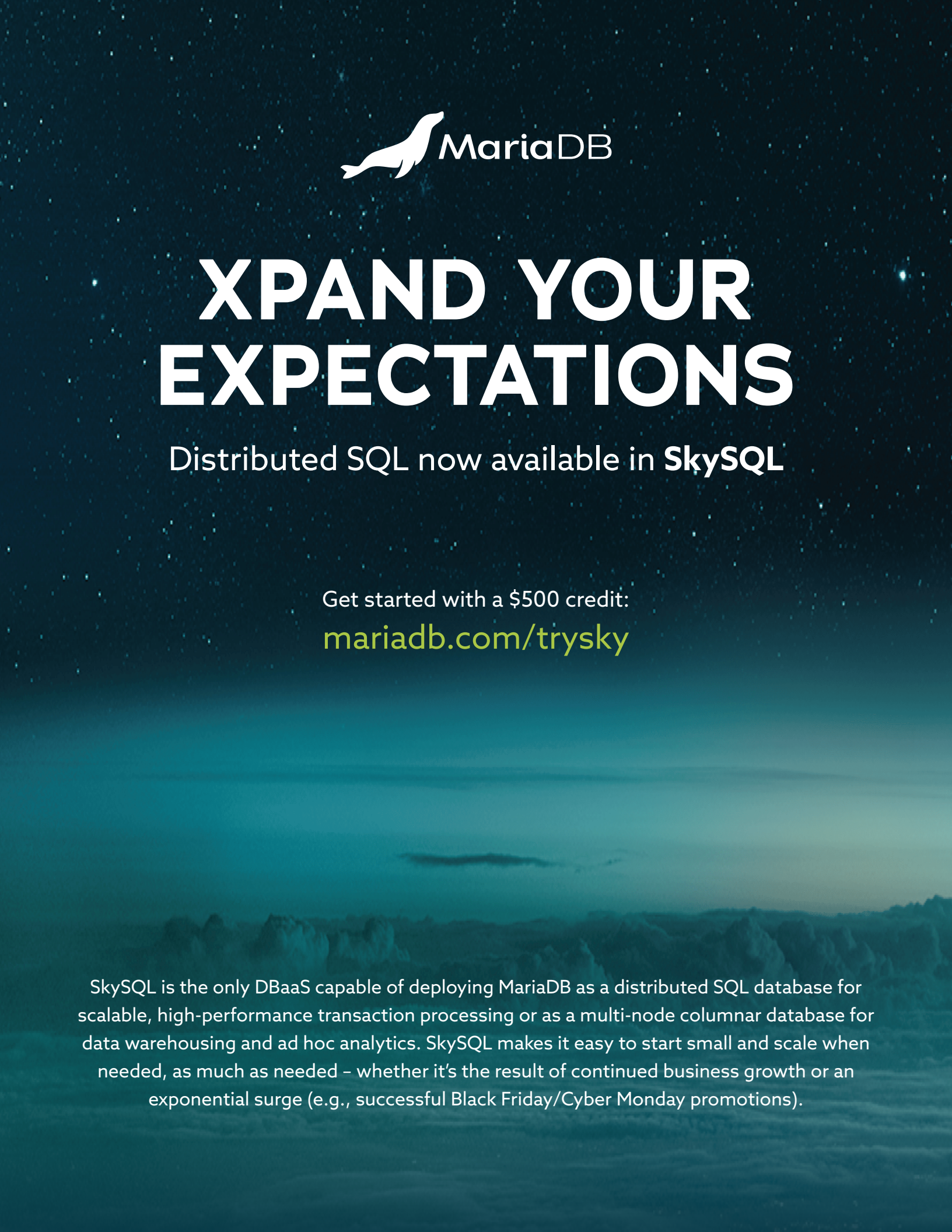
Distributed SQL databases combine the resilience and scalability of a NoSQL database with the full functionality of a relational database. They distribute data and processing across multiple servers, containers, or virtual machines (VMs). They offer the same ACID guarantees of traditional relational database management systems (RDBMSs) along with the scale and availability of a distributed database. Compared to traditional relational databases, they offer greater scale, reliability, and larger database sizes. Compared to NoSQL databases, distributed SQL offers more robust functionality and consistency. Inherent to distributed SQL databases is the use of SQL as a standard query language.

Distributed SQL databases are designed to be general-purpose operational databases and are most useful as operational stores where scale, availability, and disaster recovery requirements exceed the capabilities of a traditional relational database. For example, Samsung uses a distributed SQL database to store its customer information for their Samsung cloud service, a photo and information service similar to Apple's iCloud. ShortStack uses a distributed SQL database to handle their user data for running online contests.

Example use cases include:

E-commerce data	User interaction, transaction, product
Financial services	Trade and transaction, fraud prevention, customer and account information
General business	Supply chain, inventory, financial, customer and account information

Distributed SQL databases are distinct from some other types of nontraditional relational databases. For instance, Amazon Aurora and Google Alloy allow only a single writer with many replicas or two writers (multi-master) with no additional replicas. Aurora relies on shared storage for reliability and scalability. The term "NewSQL" was previously used to be more inclusive of other types of databases, including in-memory databases like VoltDB.


MariaDB
XPAND YOUR EXPECTATIONS
Distributed SQL now available in SkySQL
Get started with a $500 credit:
mariadb.com/trysky
SkySQL is the only DBaaS capable of deploying MariaDB as a distributed SQL database for scalable, high-performance transaction processing or as a multi-node columnar database for data warehousing and ad hoc analytics. SkySQL makes it easy to start small and scale when needed, as much as needed – whether it's the result of continued business growth or an exponential surge (e.g., successful Black Friday/Cyber Monday promotions).

While keeping all or most data in memory can lead to lower latency and is good for specialized use cases, it is not cost effective for applications at a greater level of scale. Some NewSQL databases are actually analytical stores whereas distributed SQL databases are primarily transactional.

HOW DISTRIBUTED SQL WORKS

Distributed SQL databases use a hashing algorithm to assign writes to different units called partitions (or slices in some databases). Figures 1 and 2 show how those partitions are distributed among multiple compute nodes such as VMs, containers, or physical hardware. Each partition is replicated to at least two nodes (generally more).

While this shares some similarity with partitioning or sharding in non-distributed databases, it is automatically assigned by hash rather than value ranges and automatically balanced by the database rather than operator intervention.

Figure 1

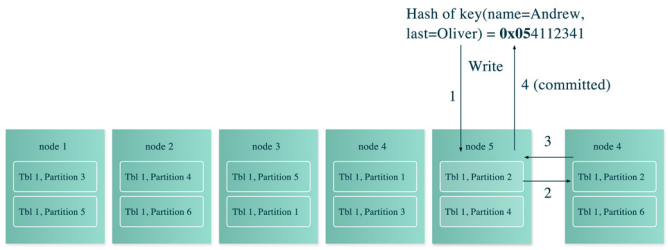
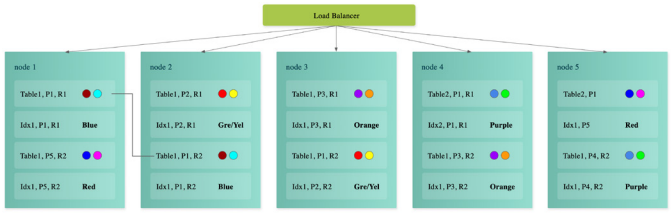


Figure 2



When a client reads from a distributed SQL database, the database computes the hash and selects one or more nodes to surface the requested data.

Likewise, queries may also be similarly distributed among multiple nodes in the database. Because data is distributed, reads can pull from multiple storage devices at the same time. In order to ensure data is consistent when written or updated, the database uses a type of distributed transaction protocol similar to two-phase commit.

Modern distributed SQL databases primarily use a consensus algorithm such as Paxos or Raft. These protocols coordinate membership in the cluster along with ensuring that data is written to the correct nodes in order to guarantee data consistency and reliability.

Distributed SQL databases work best in the cloud if replicas are distributed among cloud availability zones (or different racks in private data centers). In the event a zone or region becomes unavailable, a new leader is elected in one of the remaining zones or regions.

Data is copied from surviving replicas to existing nodes to maintain fault tolerance and data distribution (see Figures 3 and 4). If new nodes are added, the data is rebalanced among the new nodes, increasing distribution and performance.

Figure 3

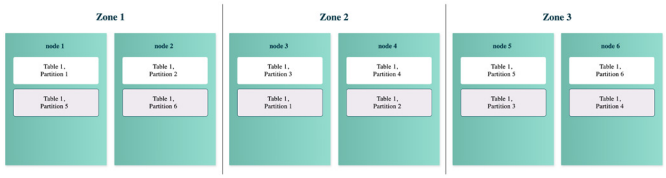
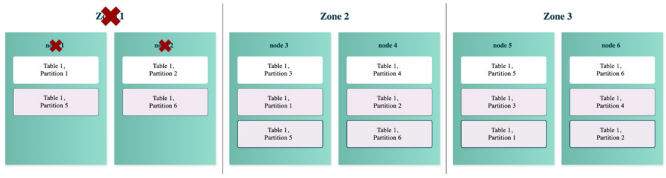
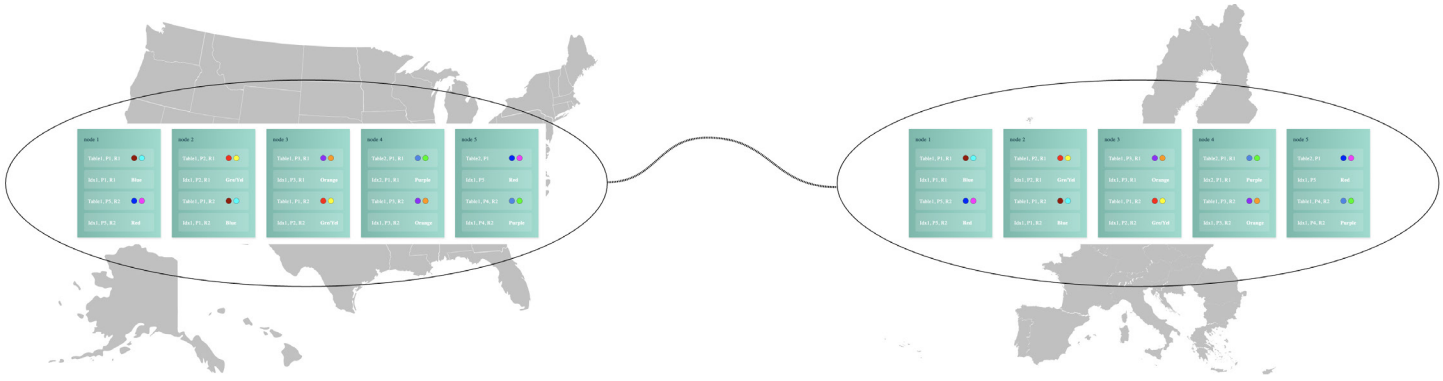


Figure 4



Some databases support distributing replicas or partitions across geographic regions. This significantly increases latency and impacts overall performance. To address this, an eventually consistent latency-tolerant replication protocol is used across data centers (see Figure 5).

Figure 5



SHARED CHARACTERISTICS OF DISTRIBUTED SQL DATABASES

While no two distributed SQL database products are exactly alike, they do have shared characteristics that distinguish them from other types of databases. First and foremost, distributed SQL databases are operational stores as opposed to analytical stores. Though some distributed SQL databases are combined with analytical stores, that functionality is outside of distributed SQL itself — similar to how some traditional relational databases supply full-text search.

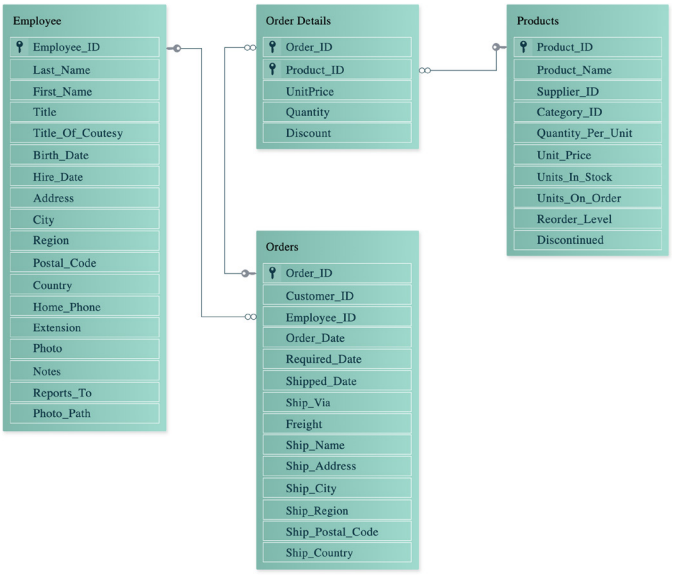
RELATIONAL MODEL

Distributed SQL databases use a relational model, in which:

- Data is represented in tables, rows, and columns
- Records are rows and fields are columns
- Each row is identified by a unique identifier called a primary key
- Data is joined between tables by shared values called foreign keys
- A unique identifier, called a primary key, identifies each row
- Shared values, called foreign keys, join data between tables

As with some traditional relational databases, the underlying storage may be substantially different than what is represented:

Figure 6



Despite the similarity and intentional compatibility, there are often differences in how data is modeled compared to traditional relational databases. The most obvious is that sequences are highly discouraged because generating a sequence across a distributed cluster creates a bottleneck that hampers scalability and performance. Instead, natural keys or randomly generated unique keys are preferred.

GENERAL ARCHITECTURE

Distributed SQL databases are based on the same general architecture. Data is stored on multiple nodes. Writes are balanced between those nodes and assigned via a hashing algorithm, while reads are likewise balanced. Data is replicated to more than one node, so a distributed SQL database can survive the loss of one or more nodes. Writes and updates are handled via a distributed transaction that is coordinated among nodes. Some combination of client-side proxies or a load balancer directs traffic between database nodes.

ACID TRANSACTIONS

Unlike other distributed database technologies (i.e., NoSQL), distributed SQL databases are designed for systems of record. They supply transactional integrity and strong consistency from the ground up with coordinated writes, locked records, and other methods such as multi-version concurrency control.

SYNCHRONOUS REPLICATION

Distributed SQL databases use synchronous replication between nodes to ensure transactional integrity with continuous availability. When a write takes place, each node acknowledges the write. Other similar types of databases, like Amazon Aurora, use asynchronous replication, which could cause inconsistent writes between nodes.

QUERY DISTRIBUTION

Compared to client-server database technologies, distributed SQL database queries are replicated to any number of database nodes. Additionally, data can be pulled from multiple nodes and aggregated into a single result set. Some distributed SQL databases even distribute processing parts of complex queries (i.e., joins, subqueries) to different nodes.

DIFFERENCES BETWEEN DISTRIBUTED SQL DATABASES

While the basic architectural approach of distributed SQL databases is easily recognized and distinct from both NoSQL and traditional relational databases, there are some key differences between them.

DELIVERY (CLOUD/DBAAS, ON-PREMISES, HYBRID)

At this time, every distributed SQL database can be installed in the cloud; however, not all of them offer a fully managed database-as-a-service (DBaaS). Some distributed SQL databases are available in DBaaS formation, as a customer install, and even hybrid installations where the DBaaS can manage local instances and replicate between a private data center and a cloud installation, and vice versa.

COMPATIBILITY

Distributed SQL databases strive to be compatible with existing traditional RDBMSs. However, similar to the previous generation of relational databases, there are differences in dialects, data types, and extended functionality like procedural languages. Leading distributed SQL databases have varied approaches to address compatibility.

MariaDB Xpand, for example, maintains compatibility with both MySQL and MariaDB databases. This compatibility includes both wire and SQL dialect, which means you can use most of your existing tools and frameworks that work with MySQL or MariaDB. CockroachDB attempts to be wire compatible with PostgreSQL but reimplements the query engine to distribute processing. This increases compatibility but reduces some opportunity for distributed query processing and tuning. For complex applications migrating to distributed SQL, an

existing traditional RDBMS front end in compatibility mode may make the most sense, particularly if you are using extended features of a traditional database. However, if you're running in production over the long term, migrating to a performance topology is likely a better option than using an existing front end.

CONSENSUS ALGORITHM

In the early 2010s, NoSQL databases were widely popular for their scalability features. However, they relaxed transactional consistency and removed key database features, including joins. While adoption of NoSQL was swift for applications where scale and concurrency were the most important factors, most mission-critical applications that required transactional integrity remained in client-server databases like Oracle, MySQL, PostgreSQL, and SQL Server.

Ongoing research into consensus algorithms (e.g., Paxos Raft) enabled the creation of better horizontally scaling databases that maintain consistency. There are academic arguments over which is better, but from a user standpoint, they serve the same basic purpose. This research and other developments made some of NoSQL's compromises unnecessary: It's no longer necessary to rely on "eventual consistency" or BASE instead of strong or ACID-level transactions.

SCALABILITY

The distributed SQL architecture enables horizontal scalability; however, implementation details have a large impact on production reality. The key to scalability is *how data is assigned to nodes* and *how data is rebalanced over time*. Additionally, load balancing plays a central role in both scalability and performance. Some databases rely on the client to "know" which node to address. Others require traditional IP load balancers or use more sophisticated database proxies that understand more about the underlying database.

FAULT RECOVERY

All distributed SQL databases are largely fault-tolerant. However, they differ in what happens during a fault. Does the client have to retry the failed transactions, or can they be recovered and replayed? How long does it take for the database to rebalance data between nodes in the event one is lost?

KUBERNETES

The major distributed SQL implementations support Kubernetes, but implementation and performance varies between them based on how IOPS are handled. While some allow bare-metal installations, self-healing and other functionality is limited or lost when running without Kubernetes.

MULTIMODAL

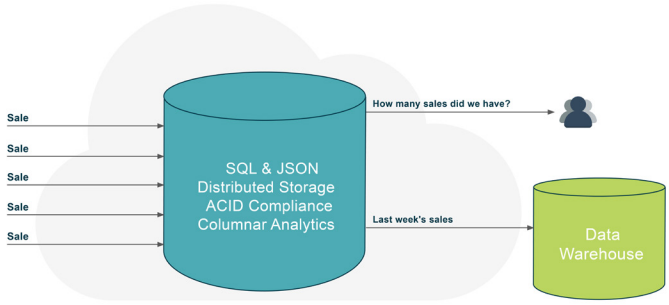
Strictly speaking, multi-modal functionality is not a distributed SQL function but is based on whether ancillary processing or data storage types are provided with the database and how consistency guarantees apply to that functionality. Examples include column storage, analytics, and document storage. If a distributed SQL database provides these additional features, it's possible to combine real-time analytics along with operational capabilities.

COLUMNAR INDEXES/MIXED WORKLOAD SUPPORT

Distributed SQL databases are operational or transactional databases by nature. However, by adding columnar indexes, distributed SQL databases can handle real-time analytical queries. Consider the case of e-commerce: The majority of queries will be light reads and writes, but eventually, someone will want to report on the sales or types of customer engagements — or even offload summaries into a data warehouse. These are long-running analytical queries that may benefit from a columnar index.

Most distributed SQL databases do not yet have this capability, but it can be expected to become more commonplace as developers look to consolidate and simplify their data architecture.

Figure 7



EVALUATING DISTRIBUTED SQL DATABASES

The most important aspect of designing a proof of concept (PoC) is to focus on data and queries that closely match your actual application. There is a temptation to test the platform's limits with unrealistic queries (e.g., 15 joins with six tables that pulls back 1M rows or a single row point query) and measure the performance between different systems. Database technologies make trade-offs and optimize for particular usage patterns. In the case of distributed SQL, the database optimizes for throughput of transactional volume.

In designing a PoC, actual production data and application traffic is optimal. Second best is a simulation that closely matches the general pattern in terms of table structure, query complexity, and proportion of reads and writes. It's important to set goals beyond a single factor, such as pure database latency, and focus on overall app performance at nominal and peak usage. This means that if at nominal use, a traditional database offers 1ms latency but 1,000ms at peak usage, and the application performs at 4s but has a performance goal of 3s, it's not meeting the objective. If a distributed SQL database performs at 15ms under nominal usage but performs at 20ms at peak usage — and the application meets its 3s goal — it has met the requirement.

In generating load, it is essential to ensure that the infrastructure can generate sufficient load to test the database system capacity at the intended performance goal. For instance, if observed latency increases significantly at 1,000 transactions per second, but overall resource utilization of disk, CPU, and network do not appear to be bottlenecked, it may be that the load generation infrastructure is maxed rather than

the system under test. It is equally essential to ensure the client network and other infrastructure between the load generator and system under test have sufficient capacity.

COST CONSIDERATIONS

Evaluating cost is more complex than simply reviewing licensing, cost per hour, or any other vendor-advertised measure. It is important to consider the entire cost of the system, including factors such as:

- Staff training
- Ongoing maintenance
- Risk of loss of service during a failure
- Downtime during upgrades
- Support and support quality
- IOPS for cloud services

LEARN MORE

Distributed SQL databases are one of the hottest new technologies in cloud computing. They offer transactional integrity without sacrificing scalability and are built for reliability in the cloud. This new technology makes it possible to bring applications that require a system of record to the cloud. The following resources provide additional information on distributed SQL databases:

- "Distributed SQL" – https://en.wikipedia.org/wiki/Distributed_SQL
- "What You Need to Know About Distributed SQL" – https://dzone.com/articles/what-you-need-to-know-about-distributed-sql
- "Distributed SQL Essentials" Refcard – https://dzone.com/refcardz/distributed-sql-essentials

WRITTEN BY ANDREW OLIVER,
SR. DIRECTOR OF PRODUCT MARKETING, MARIADB

Andrew C. Oliver is the Senior Director of Product Marketing for MariaDB. He is a prolific writer about technology — particularly open source and distributed database technologies. In the past, he served on the board of the Open Source Initiative, founded Apache POI and was an early part of JBoss, Inc. before its acquisition by Red Hat. Find him over on Twitter @acoliver.

DZone
600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.