

Getting Started With Distributed SQL

DOMENIC RAVITA

FIELD CTO, VP OF PRODUCT MARKETING

CONTENTS

- Introduction
- The Need for Distributed SQL
- Sharding Middleware
- Distributed SQL Architecture
- Comparing Architectures of Distributed SQL Databases
- Additional Characteristics of Distributed SQL Databases
- Query Execution Architecture
- Distributed DDL
- Distributed DML
- Distributed Joins
- Conclusion

INTRODUCTION

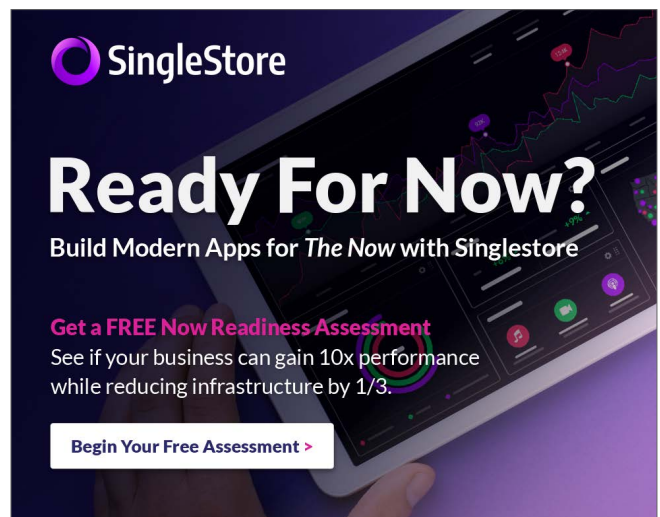
Modern applications are built as cloud-native, distributed systems. Applications supporting transactional processing are built as stateless, microservice-based distributed systems to support the scale, speed, resilience to failure, and elasticity. For applications supporting analytical processing, the driving design need is to support the growing dataset sizes while simultaneously supporting higher concurrency. To scale the data tier for both types of applications, it's not uncommon to find sharding middleware managing single-node database instances. But this approach has its limitations and management overhead challenges, which have led to distributed databases gaining in popularity across the workload spectrum.

In recent years, NoSQL distributed databases have become common, as they are built from the ground up to be distributed. Yet they force difficult design choices such as choosing availability over consistency, data integrity, and ease of query to meet their applications' need for scale. Using this approach often means giving up on relational SQL and performing complex logic in the application such as joins. These tend to be error-prone and inefficient compared to traditional RDBMS systems, which provide much better *data independence*, meaning that query logic in applications is less tightly coupled to the physical structure of the data. These trade-offs characterize NoSQL distributed databases, which were developed to address the scale problems of existing traditional single-node database systems and to take advantage of horizontal scaling. Distributed SQL databases,

by contrast, offer the benefits of scale-out, while also providing consistency and an ANSI-compliant SQL interface. This Refcard serves as a reference of the key characteristics of distributed SQL databases and provides information on the benefits of these databases, as well as insights into query design and execution.

THE NEED FOR DISTRIBUTED SQL

The speed at which businesses are building new applications in the cloud and are moving legacy applications to the cloud is increasing. The drivers are faster development cycles, the ability to scale on-demand, and the ability to pay-as-you-go, avoiding large capital outlays — and the need for extensive justification — up front.



SingleStore

Ready For Now?

Build Modern Apps for *The Now* with Singlestore

Get a FREE Now Readiness Assessment
See if your business can gain 10x performance while reducing infrastructure by 1/3.

[Begin Your Free Assessment >](#)



Are You Ready For Now?

Build Modern Apps for *The Now* with SingleStore.

Get a **FREE Now Readiness Assessment**

See if your business can gain 10x performance while reducing infrastructure by 1/3.

Begin Your Free Assessment >

SingleStore builds *The Database of Now™*, so you can build the future.

SingleStore.com

Simultaneously, large monolithic applications are being refactored into, or replaced by, distributed microservice-based applications, which offer the benefits of development independence and shortened development cycles. Distributed SQL databases offer an advantage for these new, modern, cloud-native applications. They provide better performance at lower cost than a solution built using a traditional database like PostgreSQL or MySQL, plus sharding middleware to achieve scale. Also, distributed SQL databases don't force users to give up on ACID-compliant transactions and joins in the database to achieve availability and scalability. While that trade-off may be acceptable for some use cases and applications, it certainly is not for a broad set of applications like those in financial services.

SHARDING MIDDLEWARE

Sharding middleware supports the distribution of a single-server database, such as MySQL, across multiple independent servers. The user chooses a key — the shard key — that's employed to decide which records go to each server. Each server acts as a separate database, but the sharding middleware logically combines them, so they appear from the outside as a single, distributed database.

The advantage of using sharding middleware, versus going to a NoSQL solution, is that SQL support is maintained. However, the downsides of using this middleware work directly against the hoped-for benefits. Downsides are likely to include:

- **Shard key choice.** Choosing the shard key, or partition key in some systems, correctly is vital. If this is done incorrectly, or if needs change, the entire database must be re-sharded, which takes time, is likely to involve downtime, and may come at some risk to the data stored in the database.
- **Slower performance.** The middleware is an extra layer, so both updates and reads will be slowed to some degree. Joins, in particular, become quite complex. As the join logic must be written in the application, it's often less efficient and more error-prone than what distributed databases provide.
- **Operational complexity.** The sharding middleware introduces more moving parts, which increases the cost of management. It also impacts routine operations such as security patches, backups, and recovery.
- **Limited scalability.** The problems above impose practical limits on the degree of scalability that's achievable with sharding middleware.
- **Cost.** The sharding middleware may have a license fee of its own, and additional servers, operations personnel, and operations steps are required. Whatever degree of redundancy needed, it is likely to be difficult to achieve

without additional spending and staffing. All of this increases cost. Moreover, developers often resort to writing distributed query processing logic into application software, diminishing their productivity.

DISTRIBUTED SQL ARCHITECTURE

Distributed SQL database software has automatic but configurable sharding or partitioning included in the database software itself. This reduces or eliminates the problems that sharding middleware brings with it. This software is often referred to as NewSQL database software. It includes databases such as Google Spanner, Google BigQuery, VoltDB, and SingleStore (MongoDB tries to achieve most of the same goals using a NoSQL architecture).

Some database software is targeted more at transactional workloads, and some at analytical workloads — but the ideal for most of these products is to combine both kinds of workloads into one. Distributed SQL databases accomplish this by the use of an architecture that's made up of three layers: (i) SQL API, (ii) distributed query execution, and (iii) distributed storage.

SQL API

The SQL API allows you to interact with your tables and data stored inside your database, as if you are running queries against a single-server relational database. You may leverage the SQL API to insert, update/delete, perform join operations, or select data from tables for your web application.

DISTRIBUTED QUERY EXECUTION

An efficient query optimizer should be able to deliver an efficient query plan with minimal resource consumption and fast response time. In order to avoid bottlenecks on a single node, the query execution is distributed across nodes in the cluster.

DISTRIBUTED STORAGE

In a highly scalable distributed system, data is sharded automatically and uniformly stored across nodes in a cluster. Sharding optimizes query performance for both distributed aggregate queries and filtered queries with equality predicates. A distributed storage system allows scaling by adding more servers, increasing capacity and performance linearly. Distributed storage architecture allows you to scale out the cluster size horizontally based on demand. Distributed databases employ different architectures to achieve optimized execution and user experience for their target workloads. Next, we'll classify these architectures and list their characteristics.

COMPARING ARCHITECTURES OF DISTRIBUTED SQL DATABASES

SYSTEM ARCHITECTURE	CHARACTERISTICS	EXAMPLES
Shared Everything	Not distributed SQL but included for comparison. These are single-node database instances with local CPU, local memory, and local disk storage.	MySQL, PostgreSQL
Shared Memory	Compute nodes access a common memory address space via high-speed network	High-end computing in scientific simulations. Not common in business practice.
Shared Storage	Distributed SQL database where compute nodes are independent of durable storage. Compute nodes have local memory and buffer pool for ephemeral data. Pays the penalty of not having data locality. Updates require messaging between compute nodes to notify them of the changed state.	Oracle Exadata, Snowflake, Google BigQuery
Shared Nothing	Distributed SQL database where each node has its own local CPUs, memory and local storage. This design offers the best performance and efficiency due to data locality, moving the least amount of data across the network	SingleStore, VoltDB

ADDITIONAL CHARACTERISTICS OF DISTRIBUTED SQL DATABASES

SYSTEM CHARACTERISTICS	DESCRIPTION
Types of Nodes	Uniform or role-based. In distributed SQL systems with uniform nodes, every node is identical. The downside of this approach is that each node must communicate with many nodes in the cluster to obtain sufficient metadata for cluster operation. In systems with role-based nodes, nodes perform in one role out of two or more. One benefit of this approach is the isolation of metadata management to only nodes in that role.
Distributed Storage	To leverage storage across independent storage devices, a database is partitioned, or sharded, across multiple nodes. The DBMS executes query fragments on each partition and then combines the results to produce a single answer. Applications and users may have no knowledge of where data is physically located or how tables are partitioned or replicated. A partitioning scheme should be chosen that maximizes single-node transactions to avoid the need to coordinate the behavior of concurrent transactions running on other nodes.
Distributed Queries	Generally, there are two approaches for where query execution is performed: (1) move the query computation to the data or (2) move the data to the query computation. More modern distributed SQL databases leverage both approaches, choosing the method which provides the greatest efficiency for the query type.
Replication	Data is automatically copied to multiple nodes to increase availability, protecting against node failures. The number of copies depends on the distributed SQL implementation. There are two approaches: (1) master-replica and (2) multi-master.

Some of the additional characteristics that make a distributed SQL database special include:

- **ANSI SQL.** The distributed SQL environment makes it extremely easy to query your data, whether in rowstore or columnstore tables, with a well-understood set of performance tradeoffs involving Data Definition Language (DDL) and Data Manipulation Language (DML) query design.
- **Distributed DDL.** Every distributed table DDL has exactly one shard key, which can contain any number of columns.
- **Distributed Joins.** Efficiently execute any join query, taking advantage of opportunities to improve efficiency based on sharding and replicated (reference) tables. Since reference tables are fully replicated on every machine in the cluster, leaves can join against their local copies of reference tables, with optimal performance.
- **Distributed Query Optimizer.** Leverages shard keys to determine how a query should be executed. For example, queries that fully match the shard key can be routed directly to a single partition on a single leaf server. For queries that need to shuffle data across nodes, data movement is minimized.

QUERY EXECUTION ARCHITECTURE

Modern query execution engines are capable of handling queries against fast-moving operational data with high performance and low latency. The reference information below covers the Data Definition Language (DDL) and Data Manipulation Language (DML) affecting query design.

Most common distributed query execution serves in one of two roles: master/aggregator nodes or leaf nodes. Aggregators can be thought of as load balancers or network proxies, through which SQL clients interact with the cluster. The only data aggregators store is metadata about the machines in the cluster and the partitioning of the data. The leaves function as storage and compute nodes.

As a user, you interact with an aggregator as if it were “the” database, running queries and updating data as normal via SQL commands. Under the hood, the aggregator queries the leaves, aggregates intermediate results (hence the name), and sends final results back to the client. All of the communication between aggregators and leaves for query execution is also implemented as SQL statements.

Data is ideally sharded across the leaves into partitions. The number of partitions is generally configurable on a cluster level with a set variable or available as an optional parameter to the DDL statement. In the context of query execution, a partition is the granular unit of query parallelism. In some systems, every parallel query is run with

a level of parallelism equal to the number of partitions. In others, an additional degree of parallelism is provided which is intra-partition parallelism.

Note: The syntax of the SQL commands shown in this reference card are examples and may vary slightly from one distributed SQL database to another.

Different databases provide different sharding techniques. For example (this varies from one database to another): To partition the tables, some databases allow you to simply add a PARTITION TABLE statement to the database schema.

Here is an example statement that we can add to our schema to partition both tables by the State num column:

```
PARTITION TABLE towns ON COLUMN state_num;
```

DISTRIBUTED DDL

Traditionally, a schema designer must consider how to lay out columns, types, and indexes in a table. Many of these considerations still apply to a distributed system, with a few new concepts.

Every distributed table has exactly one shard key, or shard index. This functions like a normal table index and can contain any number of columns. This key also determines which partition a given row belongs to.

When you run an INSERT query, the aggregator computes the hash value of the values in the column or columns that make up the shard key, does a modulo operation to get a partition number, and directs the INSERT query to the appropriate partition on a leaf machine.

The only guarantee that you have about the physical location of data in the system is that any two rows with the same shard key value are guaranteed to be on the same partition.

Modern distributed query optimizers leverage shard keys to determine how a query should be executed. For example, queries that fully match the shard key can be routed directly to a single partition on a single leaf server. Group-by queries where the set of keys are guaranteed to not overlap between partitions can be executed in parallel on the leaves, with the results streamed back without any additional processing on the aggregator.

SHARD KEYS

Most common usage of shard keys can be classified into the following types:

PRIMARY KEY AS THE SHARD KEY

For example: If you create a table with a primary key and no explicit

shard key, the primary key will be used as the shard key by default. This helps avoid data skew.

Note: Designating a shard key is similar to the following syntax, but the actual syntax for any particular distributed SQL database may vary.

```
CREATE TABLE clicks (
  click_id BIGINT AUTO_INCREMENT PRIMARY KEY,
  user_id INT,
  page_id INT,
  ts DATETIME);
```

NON-UNIQUE SHARD KEY

The general syntax for a non-unique shard key is `SHARD KEY (col1, col2, ...)`.

For example:

```
CREATE TABLE clicks (
  click_id BIGINT AUTO_INCREMENT,
  user_id INT,
  page_id INT,
  ts TIMESTAMP,
  SHARD KEY (user_id),
  PRIMARY KEY (click_id, user_id)
);
```

In this example, any two clicks by the same user will be guaranteed to be on the same partition. You can take advantage of this property in query execution for efficient `COUNT(DISTINCT user_id)` queries, which knows that any two equal (non-distinct) `user_id` values will never be on different partitions.

DISTRIBUTED DML

How a table is partitioned affects the performance of some kinds of `SELECT` queries. In this section, we will look at common query patterns and how they are executed through the distributed system. You can use the `EXPLAIN` command, or graphical `EXPLAIN` in SingleStore Studio, to examine a query's aggregator-level and leaf-level query plans.

Note: An example to determine how query patterns are executed through the distributed system is similar to the following syntax, but the actual syntax for any particular distributed SQL database may vary.

Let's assume the following schema:

```
CREATE TABLE a (
  a1 int,
  a2 int,
  a3 int,
```

CODE CONTINUES IN NEXT COLUMN

```
SHARD KEY (a1, a2),
KEY (a3)
);
```

```
CREATE TABLE b (
  b1 int,
  b2 int,
  b3 int,
  SHARD KEY (b1, b2)
);
```

```
CREATE REFERENCE TABLE r (
  r1 int,
  r2 int,
  PRIMARY KEY (r1),
  KEY (r2)
);
```

INDEX MATCHING

Matching the Shard Key. If you specify an equality on every column in the shard key, then the aggregator will direct the query to exactly one partition. Most queries do not fall into this pattern; instead, the aggregator must send queries to every partition in the cluster for intermediate results, then stitch them together.

These queries are sent to one partition:

```
SELECT * FROM a WHERE a1 = 4 AND a2 = 10;
SELECT a3, count(*) FROM a WHERE a1 = 4 AND a2 = 10
GROUP BY a3;
```

These queries are sent to all partitions:

```
SELECT * FROM a WHERE a1 = 4;
SELECT * FROM a WHERE a1 = 4 OR a2 = 10;
```

Secondary Index Matching. If your query uses a secondary (non-shard) index, then the aggregator must send the query to every partition in the cluster. Locally, each partition's table will use its part of the secondary index to speed up the query. While the overall performance of the query is dictated by the seek and scan time of these indexes, the fact that the query must be sent everywhere in the cluster can increase the variance (and therefore overall latency) of the query.

This query matches the secondary index on the column `a3`:

```
SELECT * FROM a WHERE a3 = 5;
```

No Index Matching. Queries that do not match any index perform a full table scan on all partitions. From the perspective of the aggregator, these queries are the same as queries that match a secondary index.

AGGREGATOR MERGING

Most queries that don't involve aggregates, group-bys, or order-bys don't require any further processing on the aggregator. These queries are forwarded verbatim to one or many partitions, and the partition's results are streamed back to the client. More complex queries do require additional processing on the aggregator to merge the results from the leaves.

Order By. `ORDER BY` queries that don't involve aggregates or group-bys can sort rows on the leaves and then merge the sorted intermediate results on the aggregator. For example, a query like `SELECT * FROM a WHERE a3 = 5 ORDER BY a1` will follow this pattern. These queries leverage distributed (leaf) processing to do the majority of filtering and sorting, which makes them scalable with the amount of data in the system.

Aggregates. Queries with aggregates compute aggregate values on the leaves and then use aggregate merging on the aggregator to compute a final result. Each non-associative aggregate is converted into an expression that is associative. For example, `AVG(expr)` is converted to `SUM(expr) / COUNT(expr)` automatically by the aggregator. This is also known as local-global aggregation.

Distinct Aggregates. Distinct aggregates like `COUNT(DISTINCT ...)` are not as efficient as simple aggregates like `COUNT(*)`. Distinct values must be resolved across partition boundaries (you could have `a3=10` on two different partitions in `SELECT COUNT(DISTINCT a3) FROM a`), so each partition must send every distinct value it has back to the aggregator. Queries with distinct aggregates ship one row per distinct value per partition back to the aggregator, and can therefore be expensive if there are a lot of distinct values.

There is an exception to this rule: if you run a `DISTINCT` aggregate over the shard key, distinct values can be resolved on the leaves, and the aggregator can merge aggregate values as it would with simple aggregates. An example of such a query would be `SELECT COUNT(DISTINCT a1, a2) FROM a`.

Group By. `GROUP BY` queries are spread very efficiently across the leaves. The aggregator sends the `GROUP BY` construct to the leaves so that the leaves process data down to the size of the final, grouped result set. The aggregator then merges together these grouped results (combining aggregates along the way) and sends the final result back to the client. The cost of a distributed `GROUP BY` query is usually proportional to the number of rows in the final result set, since the traffic through the system is roughly the number of partitions multiplied by the size of the grouped result set.

Having. `HAVING` clauses are processed entirely on the aggregator, since they perform filtering after the `GROUP BY` operation is complete.

`HAVING` can be pushed down if the `GROUP BY` is fully partitioned.

DISTRIBUTED JOINS

Reference Joins. Distributed SQL architecture is fundamentally designed to efficiently execute any join query with a single sharded table and as many reference tables as you'd like. Since reference tables are fully replicated on every machine in the cluster, leaves can join the shards they own against their local copies of reference tables.

Note: An example to execute any join query with a single sharded table and as many reference tables is similar to the following syntax, but the actual syntax for any particular distributed SQL database may vary.

These queries leverage reference joins:

```
SELECT * FROM a INNER JOIN r ON a.a1 = r.r1;
SELECT * FROM r LEFT JOIN a ON a.a1 = r.r1;
SELECT * FROM a INNER JOIN
  (SELECT DISTINCT r1 FROM r) x
  ON a.a1 = x.c;
```

Aligning Shard Keys for Performance. Aligning the shard keys of large tables enables more efficient joining, known as *collocated join*. It is possible to perform arbitrary distributed joins across any tables and along any column. However, if you join two tables with identical shard key signatures along that shard key, the joins will be performed local to the partitions, reducing network overhead.

```
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT,
  user_name VARCHAR(1000),
  account_id BIGINT,
  PRIMARY KEY (id)
);

CREATE TABLE clicks (
  click_id BIGINT AUTO_INCREMENT,
  account_id BIGINT,
  user_id BIGINT,
  page_id INT,
  ts TIMESTAMP,
  SHARD KEY (user_id),
  PRIMARY KEY (click_id, user_id)
);
```

In this example, `id` is the shard key of the `users` table, and the shard key on the `clicks` table has the same signature (a single `BIGINT`). These queries join locally without network overhead:

```
SELECT * FROM users INNER JOIN clicks ON users.id =
  clicks.user_id WHERE clicks.page_id = 10;

SELECT avg(c1.t - c2.t) FROM clicks c1 INNER JOIN
  clicks c2 ON c1.user_id = c2.user_id WHERE c1.page_id >
  c2.page_id;
```

Whereas this query will stream rows between leaves:

```
SELECT u.account_id, count(distinct user_id), count(1)
FROM users u INNER JOIN clicks c ON u.account_id =
c.account_id GROUP BY u.account_id;
```

If you identify your data layout and join patterns in advance, this technique can be an extremely effective way to run performant joins between distributed tables.

CONCLUSION

Distributed SQL databases have numerous advantages over single-process SQL databases that are scaled through sharding middleware and over NoSQL databases, which lack the expressive power and computational efficiency of relational SQL:

Horizontal Scaling with Ease. The ability to distribute and process data evenly across all the machines in the cluster supports horizontal scale-out operation with considerably less operational complexity than the approach with sharding middleware and with greater ease than NoSQL databases in many cases. Also, the elastic scale-out architecture, with distributed massively parallel data processing, helps to easily manage large volumes of data.

Elasticity. The flexibility to grow or shrink the cluster size to match with the workload fluctuations can be easily achieved in modern distributed SQL databases.

Fault-Tolerance. Providing fault-tolerance through the crash recovery mechanisms is another major benefit offered by distributed SQL architecture, as the modern web applications are expected to be available all the time and site outages will have a direct impact on the business.

Partitioning/Sharding. Offers the flexibility to horizontally divide the data into multiple fragments based on table's columns (i.e., partitioning attributes). This approach enables distributed systems to handle query execution against large tables in an efficient manner, by distributing the execution of queries to multiple partitions across multiple nodes and then combining their results together into a single result.

Data Resilience and High Availability. Distributed systems can process data across clusters of machines to maximize resilience and high availability, with options to leverage in-memory and disk infrastructure. Durability can be achieved with the flexibility in writing transactions directly to disk, or writing to memory, with full data persistence and archiving. Many of the database solutions ensure that each data shard remains highly available in case of any unexpected failures. In case of any failures at instance level, the native self-healing mechanism in modern distributed systems can automatically kick off the shard replicas within a fraction of a second and keep the database application running without any major outages.

WRITTEN BY DOMENIC RAVITA,
FIELD CTO, VP OF PRODUCT MARKETING



Domenic brings 23 years of experience across consulting, software development, architecture and solution engineering leadership. He brings product knowledge and a senior technical perspective to field teams and customers. He has experience in distributed, in-memory data grids, streaming analytics, integration, and data science.



DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
 600 Park Offices Drive
 Suite 150
 Research Triangle Park, NC 27709
 888.678.0399 919.678.0300

Copyright © 2020 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.