Brought to you by:

MariaDB

Distributed SQL

for dummies®
A Wiley Brand

Get to know distributed SQL

Create distributed SQL applications

Decide if distributed SQL works for you

MariaDB
Special Edition

Andrew C. Oliver
Ted Coombs

Distributed SQL

MariaDB Special Edition

by Andrew C. Oliver and Ted Coombs

for dummies®
A Wiley Brand

Distributed SQL For Dummies®, MariaDB Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN: 978-1-394-15978-9 (pbk); ISBN: 978-1-394-15979-6 (ebk). Some blank pages in the print version may not be included in the ePDF version.

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Manager: Carrie Burchfield-Leighton

Sr. Managing Editor: Rev Mengle

Managing Editor: Camille Graves

Client Account Manager: Jeremith Coward

Special Help: Patrick Bossman, Lewis Carr, Alejandro Duarte

Table of Contents

Introduction

Distributed SQL is a relatively new technology that provides a modern way to scale large databases while maintaining ACID-level consistency, high availability, and disaster recovery. In database systems, ACID (atomicity, consistency, isolation, durability) refers to a standard set of properties that guarantees database transactions are reliably processed. Other scaling solutions for large databases such as NoSQL don't allow for the same consistency and ease of querying with industry-standard SQL.

Xpand, the distributed SQL database from MariaDB, gives you cloud-level scalability while lowering the work your data team needs to perform in managing a growing database. In this book, you discover where distributed SQL fits among other scaling methods and how it's an improvement over methods such as sharding that require skill and manhours to maintain.

Automated balancing data across nodes eliminates hotspots, or overworked nodes, common in sharding methods of scaling. The built-in rebalancer continuously balances the data across nodes by using a unique indexing scheme to assure that data is organized in a way that distributes read-and-write loads between nodes.

Maintaining slices of data distributed among the nodes provides a powerful method of fault tolerance. You can lose a node, and the system simply redistributes data among the existing nodes, relying on replicated slices on the remaining nodes.

Use distributed SQL in the cloud with SkySQL and allow your database to take advantage of the cloud-centric architecture that makes adding nodes simple and inexpensive. SkySQL allows for simple distributed SQL database management in a powerful and secure web client.

About This Book

This book is an introduction to a new and modern way of scaling large databases. This important resource for data and IT professionals, developers, and IT managers helps you understand the basic principles of using distributed SQL and acts as a guide for developers to get started quickly using many of the commonly used application development languages.

Icons Used in This Book

Like most *For Dummies* books, you'll find some icons in the margins that help you spot important information highlighted in this book. Here is what they mean:

TIP

The Tip icon points out helpful information. This content may help you save time or money.

REMEMBER

The Remember icon marks particular information that you may want to file away for later use or help you recall important details at a later date.

TECHNICAL STUFF

The Technical Stuff icon points out bits of information of a more technical nature. Depending on your level of expertise, you may want to skip these points.

WARNING

The Warning icon alerts you to information that may save you from making decisions that are harmful or helps you avoid pitfalls.

Conventions Used In This Book

A book like this is not as wide as a computer screen. When you see this character — ↩ — it means the line of code is too long and carries over to the next line.

Beyond the Book

Entire volumes could be written about the topics covered in this small book, so if you want information beyond what this book offers, check out these additional resources:

- `mariadb.com/products/enterprise/xpand`
- `mariadb.com/docs/products/mariadb-xpand`
- `mariadb.com/products/skysql`

IN THIS CHAPTER

- » **Recognizing the need for distributed SQL**
- » **Looking into client-server options**
- » **Introducing distributed SQL**
- » **Getting to know distributed SQL topologies**

Chapter 1 Getting to Know Distributed SQL

Client-server databases such as Oracle, PostgreSQL, Microsoft SQL Server, MySQL, and MariaDB were originally designed to handle smaller data sets at lower throughput and scale. Increasing scale or throughput requires the capability of a single machine. These databases now achieve high availability by replicating the entire database to more machines. NoSQL databases such as Cassandra and MongoDB focused on scale but with reduced capabilities, especially with regards to joins, transactional integrity, and SQL — the most popular standard query language. This chapter explains how distributed SQL addresses scale, integrity, and availability while maintaining the feature set of a full relational database.

Understanding the Need for Distributed SQL

This section provides an overview of some of the methods of achieving high availability (HA) and scalability. HA and scalability don't always achieve the same thing. You discover how HA is achieved by using distributed SQL databases and how to best design a highly available system that best meets your needs.

Managing scale

Scaling a database requires potentially handling multiple issues:

- **The overall size of the database:** Even with attached storage managing, querying or writing a large database requires more resources in terms of memory and compute.
- **Managing a lot of client traffic in terms of reads:** This requires more network and compute just to manage client connections and traffic in addition to the memory to handle read operations.
- **Managing writes:** Writes are more difficult to manage than reads because they must be consistent across any copies. This makes scaling writes more difficult. In addition to scale, most high-traffic databases are also required to be highly available and minimize downtime.

Distributed SQL databases manage scale not by adding more CPU, memory, or network resources to a single machine, but by slicing up the database and dividing it automatically among multiple machines. This enables

- Larger database sizes
- Faster read-throughput under load
- Higher write volume

Think of it this way. If you were moving from one home to another, you have to move all your stuff. All your friends agree to help you move. You could rent one giant moving truck and load your furniture into it, or you could rent a lot of smaller trucks. While the large truck may use less fuel than several smaller trucks, your friends step on each other loading the truck bed — meaning they contend for resources. With the smaller trucks, you could have each of your friends load one truck by themselves and drive to the new place and unload it. The smaller trucks allow more work to be done in parallel. The only catch is there is more coordination required — overhead in dividing the work and directing it to the right place.

If you have a small apartment and not a lot of stuff, the single-moving truck method is probably the best. It uses less fuel, requires less coordination and you'll probably get done quicker.

What if you have a house on the scale of the Winchester Mansion (160 rooms)? In that case, you may not even be able to rent a big enough moving truck! Renting a lot of smaller ones with more help is the only feasible method of handling that much load.

The difference between the analogy of trucks and computers is that until the advent of the cloud there wasn't an easy way to rent the small trucks let alone the larger trucks. Cloud has created a flexible business model for computer rental that enables you to easily rent a very large truck or more than one truck and coordinate them with ease.

This is also the case with databases. Smaller loads may do better with a traditional client-server database such as MariaDB Server, but larger loads with more traffic are probably better with MariaDB Xpand, a distributed SQL database. Using Xpand in the Cloud makes it possible for workloads to be rapidly and affordably scaled, including across regions and globally.

REMEMBER

Scale is the one reason to use a distributed database, but it's not the only reason. Availability is another. The one thing computing can guarantee is that things will break. Faults can happen on a single machine, network device, or even the powerlines that feed them. For modern services, especially on the scale of a large bank or SaaS company, losing service for any period of time is unthinkable. Distributed SQL databases store redundant copies of data on additional machines in additional locations (such as cloud availability zones).

Scaling with sharding

Distributed SQL isn't the only way to scale. One of the ways larger databases have been supported is by sharding MariaDB and MySQL databases. This is a method by which databases are partitioned and distributed across multiple machines. The smaller pieces allow queries and writes to use more resources. While this allows databases to scale, it introduces new problems, such as complexity, difficulty querying, and application specificity.

When a system needs a larger database size, or better read/write performance, you can split tables based on values in a column — for example, creating a database partition built on customers based on zip code ranges or on customers' last names alphabetically. This is known as *sharding.*

Some databases, such as Oracle, can automatically partition a database on these values. Other databases may require application developers to manually divide the data. While sharding is a good way to share resources, the partitions must also be replicated to ensure availability.

WARNING

Sharding isn't the optimal way to meet increasing demand. Architecting and maintaining shards requires ongoing human effort and makes an application brittle. Unless your business has sufficient engineering resources to apply outside your core area of business expertise, a different solution is needed.

WARNING

Query performance may suffer in a partitioned system because data must be rejoined in queries that cross partitions. A problem with sharding techniques is hotspots, where more traffic goes to one node than another because of the uneven nature of how the data is partitioned. For example, if you're partitioning on last name, more Smiths exist than Chadwicks.

Sharding isn't used for availability; it's used for scalability but is sometimes used along with high availability techniques to allow for larger database sizes. When you're choosing a method for high availability, decide which tradeoffs, such as risk, performance, and complexity, that you can live with. When making your decision, you should also take into consideration feasibility issues like data size and network topography.

Evaluating Existing Client-Server Options

You may already be using some of the methods we discuss in this section to achieve high availability. Each method is discussed to help you determine which is best suited for your system by weighing the drawbacks of each.

REMEMBER

Older books and documentation may use terms like *master* to refer to database nodes that do writes, *slave* for read replicas, and *multi-master* for database systems with multiple writer nodes. The industry is transitioning away from these terms now because they're offensive and inaccurate.

In this book, we use *primary* or *writer* and *replica* or *read node.*

Single writer multiple replicas

When traditional databases such as Microsoft SQL Server, MySQL, or MariaDB Server need to handle heavy read with fewer write operations, it's common to configure a cluster where one node handles the writes while reads are directed to other nodes. Often, writes are handled synchronously, meaning all read replicas must complete updates before a write transaction commits. As long as an application client isn't interested in a new or updated row, they won't have to wait. Any client interested in a row locked in the transaction may block until the transaction commits. Because this blocking occurs until all replicas acknowledge the update, it may take longer than on a single-instance database.

REMEMBER

Database sizes are limited in a single writer multiple replica setup because a copy of all data exists on every single node. A fault-tolerance downside of this method is that when a primary instance fails, writes aren't possible. A way around this is to employ a database proxy such as MariaDB MaxScale, where a read replica can be promoted to become the new primary instance and assume responsibility for writes.

TIP

Systems that fit on a single node without splitting the data are extremely economical and efficient. Replicas give you high availability and read-scale but don't work well in systems where a large number of write operations exist or the data set grows too large.

Multiple writer system

For traditional databases such as Oracle or MySQL, which need HA specifically for writes, a multi-primary system is often used. This type of system strictly provides HA but not scalability. Every write is sent to every writer node. When two primary nodes exist, the write performance is less than a single-node system because writes have to be written to both nodes and coordinated. It's fault-tolerant because if a write node goes down, the other takes over its responsibilities.

REMEMBER

You achieve an advantage over a single writer system with multiple replicas because of lower failover time. A multiple writer system configuration can be combined with other techniques although the complexity of such a system normally requires additional hardware or software.

Standby instance

Using a synchronously replicated standby instance is a variation of a multiple-writer system. Instead of having writes go to two (or more) nodes and replicate to each node, writes go to one node, and a copy is sent to a standby node. If the primary goes down, the standby becomes the new primary.

The advantage of a standby instance is that it's simpler to operate, and there's no chance of a transaction collision, which is conflicting writes occurring on two nodes, both attempting to lock similar data. Standby instances are strictly for availability and offer no performance advantage over a single-node system. In fact, there is some overhead to assure consistency between the primary and standby nodes.

Asynchronous replication

The other methods we talk about in this section assume that absolute transactional integrity is required with no chance of data loss during failures. For some data sets, performance is more important than transactional integrity. It's possible to configure replicas with lower levels of assurance than full transactional acknowledgment. In these cases, your system may be okay with simply receiving a message that the data was sent or received instead of waiting for the data to be committed.

Introducing Distributed SQL

Distributed SQL is a database that slices and distributes data and indexes across multiple nodes to achieve read and write scale and maintain availability. In this way, the data, writes, and reads are spread evenly throughout the cluster.

REMEMBER

Most distributed SQL databases are compatible with MySQL and MariaDB or PostgreSQL but operate differently. Tables are automatically partitioned into multiple slices. In MariaDB Xpand, rows are assigned to slices based on a hash of a subset of their columns (the primary key by default in Xpand). Figure 1-1 shows how those slices are then distributed to multiple nodes in the cluster. Each slice has at least one copy, called a *replica*, which exists on a separate node from the original copy.

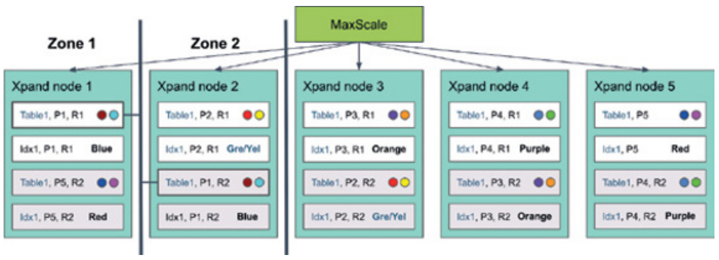


FIGURE 1-1: Every slice has a replica on one of the other nodes.

Writing to a distributed SQL database

When data is written to a distributed SQL database, the write is forwarded to each node to which that row of data is assigned. Multiple writes are handled concurrently. When the transaction commits, writes are coordinated with the replica (or replicas) of that slice. When queries are executed, the system automatically determines which nodes contain the slices of data being requested and distributes the query work across the cluster.

Scalability and availability in a distributed SQL database

Distributed SQL databases achieve scalability through distributing data and work across the cluster. They achieve availability by ensuring there's an additional copy of each slice stored somewhere in the cluster. For cloud environments, the replicas are usually divided across separate availability zones (AZs). If a node fails, the writes and reads are directed to another node with the necessary replica. During failover a new copy of the slices on the dead node is created and redistributed across the cluster. If a node recovers, the data is redistributed among the nodes.

TECHNICAL STUFF

Availability zone is a term used by cloud service providers to refer to a separate data center located in the same region. Software is often aware of AZs and ensures that redundant copies are stored in separate AZs as opposed to two replicas stored in the same AZ. Older documentation uses the term *rack aware* to refer to the same technical concept.

Distributed SQL achieves better distribution of load than other sharding/partitioning schemes because data is algorithmically assigned to slices via hash values, shown in Figure 1-2, and is distributed evenly across the cluster.

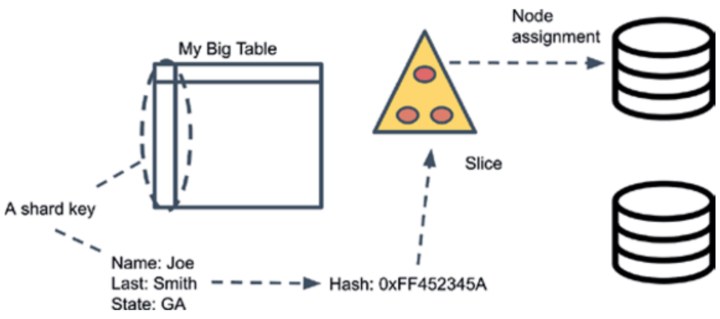


FIGURE 1-2: Data is assigned to slices using a hash value for even distribution.

WARNING

Like sharding schemes, hotspots can occur. To combat hotspots, distributed SQL systems detect them and either rerank the replicas (changing the primary) or move the slice to another node. These distributed SQL systems are superior to other single-writer and multi-writer systems in the following ways:

- Single-writer systems with multiple replicas because writes are also distributed and perform better
- Multi-writer systems because both reads and writes are distributed and across more nodes

REMEMBER

A single node system will usually perform better than a distributed SQL database for a single small transaction on a system not under load because of the overhead involved in distributing reads and writes. Execution of a single simple query may also be better on non-distributed databases because of the cost of gathering data from multiple nodes especially when using joins. Distributed databases outperform non-distributed databases with high scale data volume or throughput.

Indexing with Xpand

Unlike older databases where each node had its own siloed index, Xpand indexes enable you to find query matches across multiple nodes. This way, queries no longer need be run across every node, only those with matching data.

REMEMBER

Distributing the work of queries across multiple nodes and only those with the appropriate data increases performance and scalability.

Balancing the load with Xpand

Distributed SQL databases use different methods to balance the load across the nodes by detecting when a node is overused and responds by moving the data to even out the load. In addition to traditional load balancing reads and writes, MariaDB redistributes data when a node gets too busy by using one of the following methods:

- Rerank the replicas changing which nodes are assigned reads for a particular slice.
- Move the slice to a different node.
- Reslice the data so it's distributed across additional nodes.

The data distribution spreads writes, reads, and queries across the cluster of nodes for greater performance and availability. Xpand stores data redundantly on multiple nodes and can ensure data is replicated across multiple AZs. Xpand also automatically re-protects data if a node or zone fails.

REMEMBER

Traditional replication techniques don't scale reads, only writes.

Shared nothing architecture

Distributed SQL systems are usually shared-nothing architectures that scale linearly. For a large database, or for a system of scale, performance under load will be higher in a distributed SQL system. From an availability standpoint, distributed SQL databases can be configured to ensure there are as many (or more) replicas as there are AZs. This performance is generally superior to what's achieved with standbys or multi-writer systems. Check out Figure 1-3 to see how rebalancing is automatic.

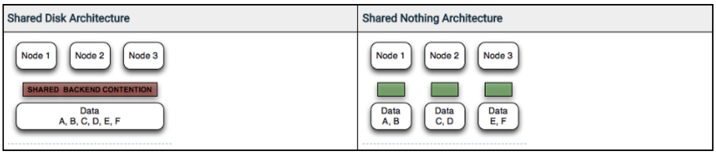


FIGURE 1-3: When there is a node fault, rebalancing is automatic.

Understanding Distributed SQL Topographies

When deciding how to deploy a distributed SQL database, consider the following:

- How much fault-tolerance do you need?
- How many AZs does your region offer?
- What kind of disaster recovery should your database support?

Most distributed SQL databases support defining how many replicas of each slice should be created. Fault tolerance requires at least two. Most users configure replicas to two; some use three for increased fault tolerance. Replicas can be configured to ensure redundant nodes are distributed across AZs. There must be the same number of nodes in each zone or an error occurs, as shown in Figure 1-4.

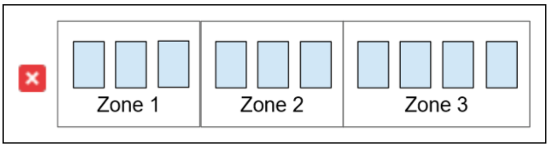


FIGURE 1-4: A fault based on an unequal number of nodes.

TIP

In MariaDB Xpand, you can add a node to a zone with the following code:

```
ALTER CLUSTER nodeid [, nodeid] ... ZONE zone
```

Disaster recovery

For disaster recovery, some distributed SQL databases support synchronous replication between regions. Synchronous replication delivers recovery point objective — zero data loss. It is true there is severe write performance penalties. So many systems use highly efficient parallel replication combined with regular backups. In the event of a disaster, the application can failover to a different geographic region.

MariaDB Xpand supports parallel asynchronous replication as well as parallel backup. These capabilities use multiple nodes in the cluster to replicate data. This capability is essential for high throughput systems that need frequent backups or those that replicate to a disaster recovery region.

Alternatives to distributed SQL

Oracle RAC and Oracle Clusterware combine multiple techniques, including partitioning in an automated system. Copies of data are moved across the cluster and ownership is transferred based on the workload. Compared with distributed SQL, it's a more complicated system that achieves some of the same results but at a higher price point.

WHAT ABOUT AMAZON RDS, AMAZON AURORA, AND GOOGLE ALLOY?

Amazon RDS is a system for deploying Oracle, MySQL, MariaDB, and PostgreSQL on Amazon Web Services. It can configure a single writer with standby instances as well as multiple replicas. It uses Amazon EBS (a shared/distributed storage architecture) in order to achieve higher performance.

Amazon Aurora deploys instances of MySQL or PostgreSQL with multi-writer and/or replica systems. Amazon makes changes to MySQL and PostgreSQL to move more database functionality to Amazon's proprietary EBS storage system to achieve higher performance.

Google Alloy deploys instances of PostgreSQL. It's a similar architecture to Amazon Aurora but with capabilities specific to Google Cloud.

IN THIS CHAPTER

- » **Understanding when distributed SQL is used**
- » **Look into technical use cases**
- » **Examining specific industry use cases**

Chapter 2 Distributed SQL in the Real World

Distributed SQL is used for systems of record at high scale with high availability. Many use cases in many industries have these requirements. In this chapter, you look at a few of those use cases across multiple industries.

Technical Use Cases

Distributed SQL has many uses but some of the technical ones provide high availability in critical systems such as those used to log in, authorize, and authenticate. Applications have historically used protocols like Lightweight Directory Access Protocol (LDAP) to query user databases such as Active Directory for authentication and authorization. These work for many internal business needs but tend not to scale to requirements such as those in Software-as-a-Service (SaaS) applications, Internet of Things applications, or consumer facing services.

Distributed SQL to the rescue. High availability and scalability enable rapid user registration, authentication, and authorization, all critical to the end-user to get things done.

Ecommerce Use Cases

Ecommerce handles an incredible amount of data. Some tables tend not to change often but must be able to handle a great many read operations. Additionally, transactionally intense operations, such as purchasing, require high availability and scalability.

Ecommerce is a perfect fit for the strengths of distributed SQL. These strengths include the capability to handle high data load, number of users, and transactional integrity.

REMEMBER

Lookup tables such as product catalog data, particularly if they're small data sets, can be copied in their entirety to all the nodes.

Some catalogs use semi-structured data types. When this data no longer fits neatly into tables, you can use JSON fields. JSON gives you similar flexibility in storing unstructured data offered by NoSQL databases, but with the strengths of using a relational SQL database.

In today's ecommerce, up-to-date pricing information is critical and sourced from many systems, even crawling other ecommerce sites leading to a high number of transactions. These potentially large data sets are also the subject of real-time analytical queries. Unlike other distributed SQL databases, MariaDB Xpand provides a columnar index that increases the speed of range and analytical queries.

Finance Use Cases

Financial Services companies are early adopters of distributed SQL technology to handle their high-throughput read and write transactions, large data sets, and zero downtime requirements. Typical financial data sets are ledgers that include financial trading systems such as those used in buying and selling stock to cryptocurrency. These systems often include the anti-fraud, Know Your Customer (KYC) systems where an additional feed is sent in real-time to counter fraud. In these cases, transactional integrity and ability to serve as a system of record is critical. Systems such as these require efficient range queries and real-time analytical queries.

IoT and Manufacturing Use Cases

There are a multitude of uses for distributed SQL in the Internet of Things (IoT) use cases found in manufacturing, energy exploration, power generation and distribution, and transportation and logistics systems. Data from sensors and asset tracking require a high number of write operations and fewer read operations. These data sets are commonly accessed by range queries.

REMEMBER

JSON is the field type of choice for semi-structured data, common in sensor data and shop floor control systems.

Device information tracking is common in electric companies and smart grids. These were the original IoT. These complex systems track everything from the meter to the transformer, often wirelessly, and reconfigure the network based on system conditions in real-time. Telemetry data from these devices is write-heavy and control systems rely on range queries to determine both current and past network status for system analysis and troubleshooting out-of-range conditions.

Another example of transaction-heavy applications are call data record systems of telecommunication networks. These records are used in customer billing and communications network analysis. Due to the billions of cellular customers, these transactions are high volume and write-intensive. Furthermore, they must be processed in near-real-time. Telecommunications networks must be highly available and be able to handle an incredible number of write operations. Read operations aren't as intensive.

Like telecommunication networks, smart networks also generate mountains of data for network reconfiguration and data security. 5G networks in particular act like smart grids and process data in real time. One of the issues with networks distributed across a wide area is latency. Your application may need to process data in real time as well as at the edge of the network closer to your clients.

Gaming Use Cases

Gaming is a serious and large business worldwide. Games require low latency and can involve massive amounts of data coming in at high throughput. Distributed SQL allows games to handle a lot of concurrent reads and updates and ensure that the system is available globally at all hours of the day and night.

REMEMBER

Distributed SQL database systems can handle multiple zone failures without a loss of availability.

Games require storing security information, handling login and authorization. Beyond that they have a lot of reference data (cards, player types, roles, game pieces) that may need to be replicated to every node and joined against player data. Ultimately, games revolve around players, their actions, and objectives. Each of these results in data being read, stored, or updated, meaning high read and write throughput.

IN THIS CHAPTER

- » **Deciding whether to cloud host**
- » **Using Docker**
- » **Deploying with SkySQL**
- » **Designing the ideal schema**

Chapter 3 Getting Started with Distributed SQL

Selecting the correct deployment model and designing proper schemas optimize your use of a distributed SQL database and your investment of time and resources. This chapter looks at reasons to deploy on-premises or in the cloud. It also describes how to craft schemas, including the key field necessary to ensure proper data distribution and other issues particular to distributed databases.

Deciding on the Right Solution

Distributed SQL is a good solution for applications and services that require absolute availability with read and write scale (check out Chapter 1 for more info on why and Chapter 2 for where it matters). MariaDB Xpand also includes columnar indexes that make operational analytics and ad hoc queries work efficiently, making distributed SQL a good choice for even more use cases. However, after a team decides to use distributed SQL, it must also choose a deployment model and location.

Many distributed SQL databases are available on-premises or in the cloud. When deciding on a cloud solution, a team must also decide if it wants to host the solution itself or purchase a database-as-a-service solution. This decision is based on whether

- Requirements necessitate an on-premises solution
- Your organization has recently procured suitable infrastructure
- Requirements can't be met using a vendor-hosted solution
- Overall cost considerations
- Your team has sufficient expertise to care for and maintain a distributed database

Hosting on-premises

Hosting on-premises allows a team more flexibility in terms of how a system is set up, which resources it's assigned, if it can meet specific regulatory requirements. Occasionally, on-premises systems are used due to more stringent latency or data sovereignty requirements or because, in an industry like manufacturing, the system is as reliable as the shopfloor systems. However, on-premises systems must ensure that adequate backup and upgrade procedures are in place. Additionally, it's important to make sure that there are sufficient zones or protections for redundancy. For instance, you can use separate racks or ideally a separate building with sufficient isolated resources.

WARNING

Choosing to host on-premises comes with the added burdens of installation, configuration, management, and continued monitoring of the infrastructure and DBMS. You may also need to purchase additional hardware and software, undertake extensive capacity planning, and plan, build, and secure the physical data center.

Cloud hosting

Self-hosting a distributed database in the cloud allows for more flexibility but also allows the database to grow and shrink as needed. Instead of procuring more infrastructure than needed, the organization can use pay as you go. However, it still requires some of the same expertise as well as backup and upgrade procedures. But, it's easier to ensure deployment in separate availability zones, and many of the redundancies the system would need are built into the cloud infrastructure.

A database-as-a-service deployment is much simpler than on-premises or self-hosted and will generally follow best practices for that particular database. In the case of MariaDB Xpand, databases can be deployed to AWS and Google Cloud using MariaDB's database-as-a-service system called SkySQL. SkySQL allows you to manage many aspects of your database system, including system upgrades.

SkySQL has multiple security certifications including SOC 2 Type II, which can alleviate compliance concerns. With SkySQL, organizations can allow MariaDB to take care of backup and upgrade procedures and even use MariaDB's staff as adjunct database administrators for query tuning and other concerns.

WARNING

While database-as-a-service deployment is much easier, the drawback is losing some degree of control and flexibility because many offerings provide one-size-fits-all or limited options for configurations, scaling, and types of availability.

Using the Docker Instance of Xpand

The MariaDB Xpand docker image is a convenient way to get started and familiar with Xpand. The docker image is a single node Xpand and isn't appropriate for performance or production use, but it's ideal as a development and learning sandbox. It's also excellent for compatibility testing of existing applications. When you're ready to try a full cluster, you could deploy using SkySQL or initiate a trial.

To discover more, visit `mariadb.com/downloads/community/xpand`.

Activate Xpand by Deploying with SkySQL

You can use MariaDB SkySQL to deploy Xpand in just a few steps:

1. **Create a SkySQL account.**

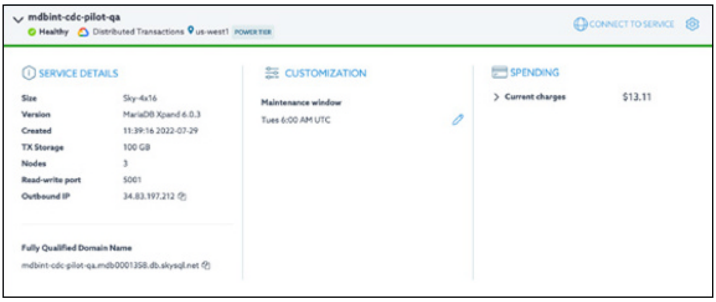
   Visit the SkySQL official website at `https://cloud.mariadb.com` and register for a MariaDB ID and account.

2. **Start services.**

   After you register an account, click Access SkySQL.

3. **Click Launch New Service, choose the Xpand Distributed SQL topology, and then you can select options for cloud provider, instance size, and storage size.**

   For this case, Figure 3-1 shows the 3-node option, but there are 1-, 3-, or 6-node options to choose from in the panel.



mdbint-cdc-pilot-qa
Healthy · Distributed Transactions · us-west1 · POWER TIER

CONNECT TO SERVICE

SERVICE DETAILS

Size	Sky-4x16
Version	MariaDB Xpand 6.0.3
Created	11:39:16 2022-07-29
TX Storage	100 GB
Nodes	3
Read-write port	5001
Outbound IP	34.83.197.212

Fully Qualified Domain Name
mdbint-cdc-pilot-qa.mdb0001358.db.skysql.net

CUSTOMIZATION

Maintenance window
Tues 6:00 AM UTC

SPENDING

Current charges $13.11

FIGURE 3-1: View your service details in SkySQL.

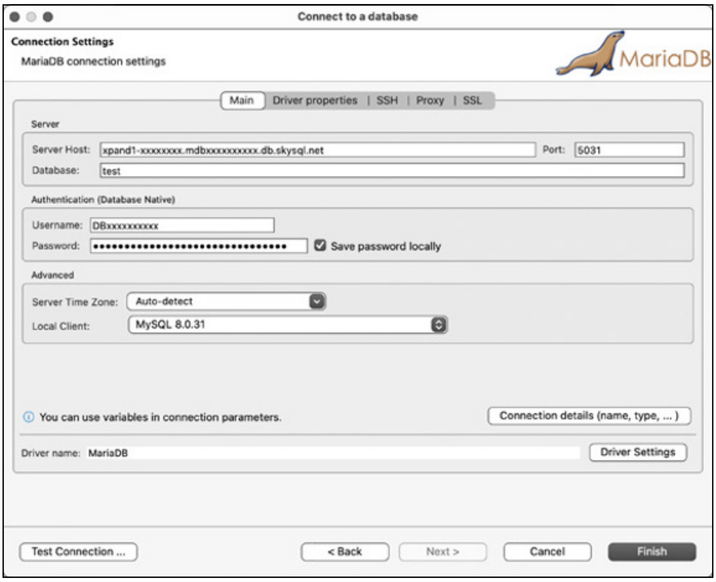
4. **Click Security Access and add your own IP so you can access it from your own device.**

After you set up your account to deploy Xpand, you have a couple of connection options:

- **Connect via Command Line Interface (CLI).** Click Connect to Service, download the certificate, and follow the instructions on screen. You're connected to MariaDB Xpand server.
- **Connect via DBeaver.** Similar to other options, you fill in the information for host, port, user, password, and path to the certificate authority. In DBeaver, you do that through the graphical user interface (GUI).

REMEMBER

Most options are in the Main tab in the Connection Setting screen, as shown in Figure 3-2. The path to the CA certificate is in the SSL tab.


Connect to a database
Connection Settings
MariaDB connection settings
MariaDB
Main | Driver properties | SSH | Proxy | SSL
Test Connection ...
< Back
Next >
Cancel
Finish
Connection details (name, type, ...)
Driver Settings

Server

Server Host: xpand1-xxxxxxxx.mdbxxxxxxxxxx.db.skysql.net Port: 5031
Database: test

Authentication (Database Native)

Username: DBxxxxxxxxxx
Password: ••••••••••••••••••••••••••••••••• [✓] Save password locally

Advanced

Server Time Zone: Auto-detect
Local Client: MySQL 8.0.31

You can use variables in connection parameters.

Driver name: MariaDB

FIGURE 3-2: The database connection detail dialog box.

Crafting the Perfect Schema

Schemas define the fields and data formats that comprise tables within your database. Schemas are composed primarily of

- The definitions of tables
- The primary keys that define a unique row in the table
- The foreign key that relates two tables
- The indexes that ensure queries run efficiently

In a distributed database, you also have an additional distribution key that helps determine how tables and indexes are distributed throughout the cluster.

This section overviews how to construct keys and slices with a distributed SQL database, including with mixed data such as JSON.

Keys and slices

Traditional relational databases often use autoincrement fields or database sequences to create a surrogate primary key. These aren't efficient on even high-traffic, client-server databases but are a bigger problem on a high-scale distributed database. The performance issue is the database must synchronize threads in order to assure monotonically incremental values. In order to work around this, some developers have used BINARY(16) fields and generated universally unique identifiers (UUIDs). These avoid the issue of thread synchronization with an extremely low risk of duplicates, but because UUIDs fill the keyspace randomly at a distance, they index poorly.

MariaDB Xpand offers AUTO_UNIQUE and AUTO_INCREMENT. AUTO_UNIQUE outperforms AUTO_INCREMENT. AUTO_INCREMENT guarantees rows get keys incremented by one in exactly the order they're inserted. AUTO_UNIQUE generates unique values but doesn't guarantee sequential order. Unlike generated random UUIIDs, the keys generated by AUTO_UNIQUE are close together and index well.

AUTO_UNIQUE can be used to create a unique key when a table's primary key isn't available. Each value generated by the database is guaranteed to be unique. AUTO_UNIQUE looks a lot like a sequence but doesn't guarantee the value will be sequential in transactional order.

The following create statement generates a table with an AUTO_UNIQUE field:

```
CREATE TABLE hq_sales.invoices (
    invoice_id BIGINT(0) UNSIGNED AUTO_UNIQUE NOT ↩
   NULL,
    branch_id INT NOT NULL,
    customer_id INT,
    invoice_date DATETIME(6),
    invoice_total DECIMAL(13, 2),
    payment_method ENUM('NONE', 'CASH', 'WIRE_ ↩
   TRANSFER', 'CREDIT_CARD',    'GIFT_CARD'),
    PRIMARY KEY(invoice_id)
);
```

In addition to the primary key, distributed SQL databases use a distribution key. This field or set of fields is hashed in order to determine which slice of the database the row will be assigned to. By default, the key will be the first column and the primary key.

However, for some data it makes sense to pick a different set of fields in order to avoid hotspots based on usage patterns or the data itself. When picking a key, ensure the field or set of fields has a high number of distinct values (NDV). This means if a table has STATE_CODE (referring to states in the United States) and is used as a distribution key, the table can't be divided into 100 slices (because there are only 50 states and a handful of districts and territories), so pick a key that has enough NDV.

The following code is a create statement, generating a table with the primary key used to distribute the table and multiple fields used to distribute the index "user_id_posted_on_idx":

```
CREATE TABLE user_posts (
     post_id int AUTO_INCREMENT,
     user_id int,
     posted_on timestamp,
     data blob,
     PRIMARY KEY (`post_id`) /*$ DISTRIBUTE=1 */,
     KEY `user_id_posted_on_idx` (`user_id`, ↩
   `posted_on`) /*$ DISTRIBUTE=2 */
     );
```

This key helps the database pick which nodes get which record or index entry.

Tables big and small

One of the advantages of distributed SQL databases over NoSQL databases is their ability to support joins. However, the cost of joins in any distributed database is higher than in a client-server database. Depending on the distributed SQL solution, these can be more or less efficient, but nothing is as efficient as joining on a single server.

TIP

Excessive schema normalization isn't generally a good practice on a client-server database, but it's even more important to be judicious on a distributed database. Generally, distributed SQL databases are for larger tables with a lot of data, shown in Figure 3-3, not many small tables.

If the system must track each address associated with a person in a one-to-many relationship, it makes sense to use the format on the right of Figure 3-3. However, if this is just the person's current shipping address, the form on the left will perform better.

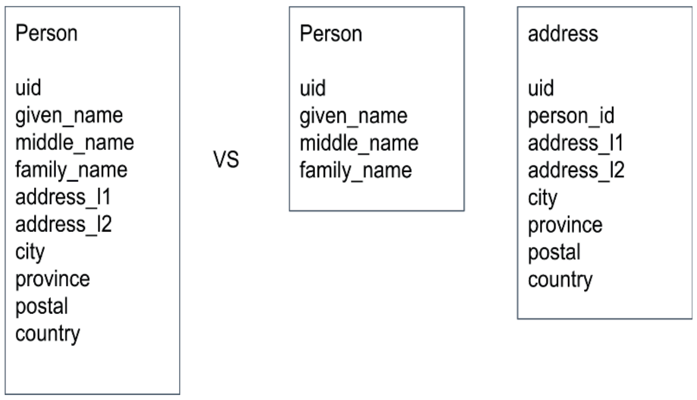


FIGURE 3-3: Consider using larger tables over many small tables.

It can help to understand how queries work. For Xpand, simple queries (point selects) and inserts scale nearly linearly. Queries are broken down into fragments and each fragment may require at most one network hop (unless the data happens to be local). However, joins require more work. Data must move to the node that needs it next and then aggregate with data on another node in which it's joined.

Because data is divided among multiple nodes, Xpand is able to use parallel processing. When more processor cores are available, overall query processing is sped up. To be efficient, data is usually forwarded to only one node.

REPLICAS=ALLNODES

Some tables are frequently accessed as reference data. It's more efficient to distribute these tables to every node in the cluster. Consider doing this for tables that are relatively small (<10 megabytes [MB]), rarely updated, read from frequently, and are frequently joined with other tables. To do this, specify REPLICAS=ALLNODES like the following code:

```
CREATE TABLE tbl_name (col_names) [REPLICAS = ↩
  ALLNODES]
ALTER  TABLE tbl_name             [REPLICAS = ↩
  ALLNODES]
```

Using this feature allows joins against reference data to avoid a hop and to happen locally on a single node making for more efficient queries. The tradeoff is that writes to this data are slow (due to the fact that all nodes have to be updated).

Range queries

Queries that use operators like < and > or keywords like *between* are *range queries.* These types of queries can be more efficient if a columnar index is used instead of a row index. Columnar indexes essentially turn the table sideways because data in a column is often repetitive, so it compresses better than a row index. A lot of compressed values can be traversed more quickly, so this works well for range queries.

Columnar indexes can be used for a single column or for a set of columns and used in combination with standard row indexes. In addition to making range queries more efficient, they can be used for ad hoc queries where a perfect index doesn't exist; this use outperforms an imperfect index or full table scan. When using columnar indexes, load data into the table before creating the columnar index with the following code:

```
CREATE COLUMNAR INDEX idx_invoices_date_total
   ON hq_sales.invoices (invoice_date, ↩
  invoice_total);
CREATE TABLE hq_sales.invoices (
   invoice_id BIGINT UNSIGNED NOT NULL,
   branch_id INT NOT NULL,
   customer_id INT,
   invoice_date DATETIME(6),
   invoice_total DECIMAL(13, 2),
   payment_method ENUM('NONE', 'CASH', 'WIRE_ ↩
  TRANSFER', 'CREDIT_CARD', 'GIFT_CARD'),
   PRIMARY KEY (invoice_id),
   COLUMNAR INDEX idx_invoices_date_total ↩
  (invoice_date, invoice_total)
);
```

JSON

Most modern relational database support JSON. The ANSI SQL 2016 standard even incorporates it. Distributed SQL databases aren't different, but how they support it and to what extent varies. In Xpand, JSON is stored in its native format.

WARNING

There is a drawback to using JSON in Xpand because the Query Results Cache (QRC) must be disabled (set global qrc_enabled = false;). This eliminates a performance optimization from the database.

Some examples that include JSON data in Xpand are as follows:

- To create a JSON table, use the following code:

  ```
  create table files (id int primary key auto_ ↩
    increment, doc json);
  ```

- To insert data using JSON, use the following code:

  ```
  insert into files (doc) values ('{"foo": ↩
    {"bar": 1}, "baz": [1,2,3,4]}');

  insert into files (doc) values ('{"foo": ↩
    {"bar": 2}, "baz": [3,4,5]}');
  ```

- To add a JSON column, use the following code:

  ```
  alter table files add column foobar json;
  ```

IN THIS CHAPTER

» **Developing applications with Java**

» **Growing your JavaScript ecosystem**

» **Writing distributed SQL applications with Python**

Chapter 4 Creating Distributed SQL Applications

Xpand fully supports most popular languages, frameworks, and technologies. In fact, you can generally use the same tools and technologies that you use to connect to MariaDB and MySQL. This chapter overviews three of the more popular frameworks in which you can create distributed SQL applications.

But before you jump into the sections in this chapter, you may want to check out Chapter 3 (if you haven't already). The information in this chapter is based on the SkySQL setup in Chapter 3.

This chapter outlines some of the important parts of connecting to, querying, inserting into, and deleting from an Xpand database. The examples are provided in Java, JavaScript, and Python. You can grab the complete examples from GitHub by cloning from the command line with

```
git clone https://github.com/mariadb-developers/ ↩
  xpand-dummies
```

Or you can use your favorite git user interface (UI).

Regardless of the language you chose, ensure you have: installed the mariadb client utilities, connected to your SkySQL Xpand instance, and created the tables the examples depend on. These are the steps necessary to do that:

1. **Install MariaDB client utilities.**

   On Linux or Windows, follow the instructions from the documentation at `mariadb.com/docs/connect/clients/mariadb-client`.

   If you're on a Mac, follow the instructions at `mariadb.com/resources/blog/installing-mariadb-10-1-16-on-mac-os-x-with-homebrew` to install via Homebrew. (***Note:*** This will also install MariaDB Server, but you don't have to run the server.)

2. **Connect to Xpand.**

   Connect to Xpand by using the following command:

   ```
   mariadb --host YOURHOST --port YOURPORT --user ↩
     YOURUSER --default-character-set=utf8 ↩
     -A -p --ssl-ca ~/Downloads/skysql_chain.pem
   ```

3. **Create the ORDERS database.**

   From the mariadb prompt, type "CREATE DATABASE orders;" and "use orders;"

4. **Create the ORDERS table.**

   Create the ORDERS table with the following SQL statement:

   ```
   CREATE TABLE orders (
        order_id BIGINT(0) UNSIGNED AUTO_UNIQUE ↩
     NOT NULL,
        customer_id INT NOT NULL,
        order_date DATETIME(6),
        order_created TIMESTAMP,
        entered_by TINYTEXT
   );
   ```

5. **Create the ORDER_ITEMS table.**

Create the ORDER_ITEMS table with the following SQL statement:

```
CREATE TABLE order_item (
     item_id BIGINT(0) UNSIGNED AUTO_UNIQUE ↩
   NOT NULL,
     order_id BIGINT(0) UNSIGNED NOT NULL,
     line_num INT NOT NULL,
     product_id INT NOT NULL,
     Description TINYTEXT
);
```

Java

Java is one of the most common languages for developing applications with distributed SQL databases, especially MariaDB Xpand. The code you see in this section demonstrate the processes of connecting to your created database, inserting, querying, and deleting rows using the Java Database Connectivity (JDBC) application programming interface (API).

Before you code and compile the example, ensure you have the following:

- **Java software development kit (SDK):** On MacOS, you can install via Homebrew with the following command:

```
brew install java
```

TIP

Make sure you follow the instructions at the end of the process for adding the `Java development kit` (JDK) to your path.

On Red Hat Enterprise Linux and CENTOS-based Linux distributions, use

```
sudo yum install java-11-openjdk-devel
```

- **Maven:** Maven is a popular build tool. On MacOS, you can install it with Homebrew by using this command:

  ```
  brew install maven
  ```

  On RHEL/CENTOS, use the following command:

  ```
  sudo yum install maven
  ```

  TIP

  Most Java developers use an Integrated Development Environment (IDE) such as IntelliJ IDE or Eclipse. You can find them here:

  - www.jetbrains.com/idea/download
  - www.eclipse.org/downloads

- **A directory structure:** Create a directory to hold your overall project and a subdirectory structure matching src/main/java/com/example.

MAVEN POM FILE

To build a project, you need a Maven build file (pom.xml), an Application.java file, and the appropriate directory structure. You can create the Maven build file called pom.xml in the project directory by using your favorite editor.

The most important part for building an application that connects to Xpand is the JDBC driver dependency:

```
<dependencies>
    <dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client ↩
  </artifactId>
        <version>3.0.8</version>
    </dependency>
</dependencies>
```

You can find a complete example of creating the Maven build file here: `github.com/mariadb-developers/xpand-dummies/blob/main/java/pom.xml`.

Creating the main application

Create the main Java application by using your favorite editor at src/main/java/com/example/Application.java. You can find the complete listing at the following:

```
https://github.com/mariadb-developers/xpand-
dummies/blob/main/java/src/main/java/com/example/
Application.java
```

To connect to the database, construct a properties object that supplies your database username and password and directs the database driver to find your downloaded SSL certificate chain (downloaded from SkySQL). Follow this code:

```
Properties connConfig = new Properties();
connConfig.setProperty("user", "YOURUSER");
connConfig.setProperty("password", "PASSWORD$");
connConfig.setProperty("useSsl", "true");
connConfig.setProperty("sslMode", "verify-full");
connConfig.setProperty("serverSslCert", ↩
  "/path/to/your/skysql_chain.pem");
```

After you've entered that code, supply your server, port, and database name along with the properties object to the Java JDBC driver manager. You do that by following this code:

```
connection = DriverManager.getConnection(
"jdbc:mariadb://YOURSERVER:YOURPORT/orders",
connConfig);
```

Data can be inserted by constructing an SQL insert statement. The statement inserts a row into the created orders table with two parameterized values and inserts the current date and time into the *order_date* and *order_created* fields. The code to insert this statement looks like this:

```
String SQL_INSERT_ORDERS = """
INSERT INTO ORDERS (customer_id, order_date, ↩
  order_created, entered_by)
VALUES (?, CURDATE(), CURTIME(), ?) """;
```

A prepared statement can be constructed inside of a Java try/catch block. The parameterized values are supplied in order by calling the appropriate "set" functions on the statement object. Finally, *executeUpdate()* tells the database to execute the statement.

```
PreparedStatement stmtOrders = connection. ↩
  prepareStatement(SQL_INSERT_ORDERS, Statement. ↩
  RETURN_GENERATED_KEYS);
stmtOrders.setInt(1, 1);
stmtOrders.setString(2, "andy");
stmtOrders.executeUpdate();
```

Deletes and updates are done in a similar way but with a different SQL statement. For example, the following statement deletes an order line item:

```
DELETE FROM order_item WHERE order_id = ? AND ↩
  line_num = ?
```

Selects are done similarly. A query joins the *ORDERS* and *ORDER_ITEM* table based on the order id. It returns all rows from the *ORDER_ITEM* table in the database. The query looks like this:

```
String SQL_QUERY_ORDERS_ITEMS = """
SELECT o.order_id, o.customer_id, o.order_date, ↩
  o.order_created, o.entered_by,
i.item_id, i.line_num, i.product_id, i.description
FROM orders o INNER JOIN order_item i ON o.order_ ↩
  id = i.order_id""";
```

A statement is constructed, and a *ResultSet* object is created from the statement by executing the statement and supplying the query. The *Resultset* is used to iterate through the returned rows. The code used is as follows:

```
try (Statement stmtQuery = connection. ↩
  createStatement()) {
   try (ResultSet rs = stmtQuery.executeQuery(SQL_ ↩
  QUERY_ORDERS_ITEMS)) {
      while (rs.next()) {
            orderid = rs.getLong("ORDER_ID");
            customerid = rs.getInt("CUSTOMER_ID");
            orderdate = rs.getDate("ORDER_DATE");
            ordercreated = ↩
    rs.getDate("ORDER_CREATED");
            enteredby = rs.getString("ENTERED_BY");
            itemid = rs.getLong("ITEM_ID");
            linenum = rs.getInt("LINE_NUM");
           productid = rs.getInt("PRODUCT_ID");
           description = ↩
    rs.getString("DESCRIPTION"));
        }
     }
  }
```

JDBC is a low-level method of supplying SQL statements to the database for querying and manipulating data. Most developers use object-relational mapping tools such as Hibernate, JOOQ, and MyBatis along with frameworks such as Spring Data and the Jakarta Persistence API (JPA). The good news is that if you're already familiar with these tools then you're already familiar with working with a distributed SQL database like MariaDB Xpand.

Running the application

Build and run the application using Maven with the following code:

```
  mvn compile exec:java -Dexec.mainClass="com. ↩
    example.Application"
```

If everything is installed correctly, you should see some output on your screen. You can also query the database from the command line to see the results. Each successful run results in one row added to the *orders* table and two rows added to the *order_item* table. However, one row is also deleted from the *order_item* table leaving only one row in each table.

JavaScript

JavaScript and TypeScript are the fastest growing ecosystems for new cloud applications, especially MariaDB Xpand. Before you begin using JavaScript, ensure you have the following installed:

- **Node.js:** On MacOS, you can install via Homebrew with "*brew install node*." On RHEL/CENTOS Linux distributions, "*sudo yum install nodejs*" should work.
- **MariaDB connector:** A command like "*npm install mariadb*" should work if node is installed correctly.
- **A directory structure:** Create a directory to hold your overall project.

Creating the main application

You can create the main JavaScript application by using your favorite editor, and call it "application.js." If you need a complete listing of the code, you can find it here: `raw.githubusercontent.com/mariadb-developers/xpand-dummies/main/javascript/application.js`.

For any Xpand application, you need to ensure the mariadb client library is loaded with the following:

```
const mariadb = require("mariadb");
```

You need the filesystem library as well in order to read the SSL certificate. You can get that by running the following code:

```
const fs = require("fs");
```

Read the pem file, including the path where you downloaded it, as follows:

```
  const serverCert = fs.readFileSync("skysql_ ↩
chain.pem", "utf8");
```

To connect, ask the mariadb library to create a connection passing your hostname, port, the read in SSL certificate, username, password, and the database name (*orders*). To do this task, use the following code:

```
let connection = await mariadb.createConnection({
   host: "YOURHOST",
   port: YOURPORT,
   ssl: {ca: serverCert},
            user: "YOURUSER",
            password: "PASSWORD$",
            database: "orders"
```

After this, you need to specify the character set immediately after connecting by using the following code:

```
await connection.execute("SET NAMES utf8");
```

To insert a row, declare the SQL statement to be passed in with parameters. The *?* will be specified later but *curdate()* and *curtime()* set the *order_date* and *order_created* fields to the current date and time:

```
const sqlInsertOrders = "insert into orders ↩
  (customer_id, order_date, order_created, ↩
  entered_by) values (?, curdate(), curtime(), ?)";
```

Pass the SQL statement to the database by using the connection object and specifying the two parameters (in order). This sets *customer_id* and *entered_by* fields to 1 and *'andy'* respectively. By default, statements are executed asynchronously, and a promise object is returned, *await* tells JavaScript to wait until the value is returned and put it in "res." This is done so you can get the orderId:

```
let res = await connection.query(sqlInsertOrders, ↩
  ['1','andy']);
orderId=res.insertId;
```

Queries are done the same way. The following query joins data from the *orders* and *order_items* table together and returns all rows in the database:

```
const sqlQueryOrdersItems = "select o.order_id, ↩
  o.customer_id, o.order_date, o.order_created, ↩
  o.entered_by, i.item_id, i.line_num, i.product_ ↩
  id, i.description from orders o inner join ↩
  order_item i on o.order_id = i.order_id";
```

In your function, send this to the database by using the query function of the connection object. The result is returned in the "rows" object that can be iterated through the following code:

```
let rows = await connection. ↩
  query(sqlQueryOrdersItems);
   console.log("returned: "+rows.length);
   rows.forEach(row => {
   console.log(row);
});
```

Deletes and updates are handled the same way. This SQL statement accepts an order id and line number as parameters and deletes the row with those values from the *order_item* table:

```
const sqlDeleteItem = "delete from order_item ↩
  where order_id = ? and line_num = ?";
```

To execute, pass the SQL and parameters to the query function of the connection object just like the insert and select. If you don't need to wait on a response, you can exclude the await keyword. Perform with the following code:

```
await connection.query(sqlDeleteItem, [orderId, 2]);
```

Using the connection object and connector library directly is a fairly low-level way of interacting with Xpand. Most developers use something like Sequelize, TypeORM, or TinyORM (while coding in TypeScript). If you're using one of those frameworks, it's just a matter of configuring it to connect to your Xpand database.

Running the application

Build and run the application with the following code:

```
node application.js
```

REMEMBER

Provided that everything was installed correctly, you should see some output on your screen. You can also query the database from the command line to see the results. Each successful run results in one row added to the *orders* table and two rows added to the *order_item* table. However, one row is also deleted from the *order_item* table, leaving only one row in each table.

Python

Partly because of Python's strength in analytics and Xpand's columnar index capability, there are an increasing number of distributed SQL applications written in Python. Before you begin ensure you have the following installed:

- **Python:** On MacOS, you can install via Homebrew with "*brew install python*." On RHEL/CENTOS Linux distributions, "*sudo yum install python3*" should work.
- **MariaDB connector:** A command like "*pip install mariadb*" should work if python is installed correctly.
- **A directory structure:** Create a directory to hold your overall project. On Mac and Linux, you can do this at the command line from your home directory with a command like "*mkdir xpand-python;cd xpand-python.*"

Creating the main application

Create the main Python application by using your favorite editor and call it "app.py." You can find the complete listing of code here: `raw.githubusercontent.com/mariadb-developers/xpand-dummies/main/python/app.py`.

To use Xpand, you need to ensure the *mariadb* connector library is imported. Use the following code:

```
import mariadb
```

To connect, ask the *mariadb* library to create a connection passing your hostname, port, the full path to your downloaded SSL certificate, username, password and the database name (orders). Ask with the following code:

```
connection = mariadb.connect(
    host="YOURHOST",
    port=YOURPORT,
    ssl_ca="skysql_chain.pem",
    user="YOURUSER",
    password="PASSWORD$",
    database="orders",
 )
```

Construct a cursor object from the connection specifying that you want results to be returned as *named_tuples*. You must also specify the character set (utf8) immediately after by using this code:

```
cursor = connection.cursor(named_tuple=True)
cursor.execute("SET NAMES utf8")
```

TECHNICAL STUFF

Consult `dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor.html` for other options, such as raw or dictionaries, for the field named_tuples.

To insert a row, declare an SQL statement to be passed in with parameters. This example allows the statement to pass in the customer_id and entered_by fields while setting order_date and order_created to the current date and time:

```
sqlInsertOrders = "insert into orders ↩
   (customer_id,
order_date, order_created, entered_by)
values (?, curdate(), curtime(), ?)"
```

Execute the SQL statement by passing it to the cursor with the parameters 1 and "andy," which will be set in the *customer_id* and *entered_by* fields respectively. Use the following code:

```
cursor.execute(
sqlInsertOrders, (1,"andy"))
```

Be sure to commit your transaction. This step tells the database to finish writing the change and make it available to other queries. Commit your transaction through the following code:

```
connection.commit()
```

Queries are done the same way. The following query joins data from the orders and *order_items* table together and returns all rows in the database:

```
sqlQueryOrdersItems = "select o.order_id, ↩
   o.customer_id,
o.order_date, o.order_created, o.entered_by, ↩
   i.item_id,
i.line_num, i.product_id, i.description
from orders o inner join order_item i on o.order_ ↩
  id = i.order_id"
```

Send this to the database using the execute function below of the cursor object.

```
cursor.execute(
    sqlQueryOrdersItems
)
```

You can then iterate through the returned values by using a for/range function as follows:

```
for x in range(0,cursor.rowcount):
    row = cursor.fetchone()
    print(row)
```

Deletes and updates are handled the same way. The following SQL statement accepts an order id and line number as parameters and deletes the row with those values from the *order_item* table:

```
sqlDeleteItem = "delete from order_item where ↩
  order_id = ? and line_num = ?"
```

Send this to the database by using the execute function of the cursor object along with the parameters:

```
print("deleting second item");
cursor.execute(
    sqlDeleteItem,
    (orderid,2))

connection.commit()
```

Executing the SQL statements directly against the database is a fairly low-level way of working with Xpand. Most developers use frameworks like Django or SQLAlchemy. Luckily, if you're using one of those, using Xpand is merely a matter of configuring them to connect to your database.

Running the application

Build and run the application with the following command:

```
python app.py
```

If everything is installed correctly, you should see some output on your screen. You can also query the database from the command line to see the results. Each successful run results in one row added to the *orders* table and two rows added to the *order_item* table. However, one row is also deleted from the *order_item* table, leaving only one row in each table.

BUT IT WAS JUST MariaDB!

Are you already a relational database developer? Minus the way the primary key for *orders* and *order_item* was generated, the code is essentially the same thing you would write for MariaDB or MySQL and really similar to anything you would write for Oracle, SQL Server, or PostgreSQL. Compatibility with existing databases is a key strength of Xpand.

What's next? You don't want to make unpooled connections with no error handling, so you can find more resources at the following sites:

- `mariadb.com/products/skysql/docs/quickstart`: A quick-start using SkySQL with Python, PHP, JavaScript, or Java
- `mariadb.com/products/skysql/docs/reference/sample-code`: Additional language-specific tutorials in different languages using different frameworks such as R2DBC
- `mariadb.com/docs/products/mariadb-xpand`: A link to the Xpand documentation
- `mariadb.com/developers`: The MariaDB Developer Hub

IN THIS CHAPTER

» Considering if distributed SQL is the solution

» Needing distributed SQL

Chapter 5 Deciding if Distributed SQL Is Right for You

Distributed SQL databases may not be the right solution for every application. This chapter helps you consider when a distributed SQL database *is* the correct application technology and it's *not.*

Determining When Distributed SQL Isn't the Answer

Distributed SQL is a powerful solution but not every awesome technology meets the specific needs of your business. Here are a few reasons *not* to consider distributed SQL:

- **When you have many small tables:** Distributed SQL databases are designed for scale. Complex small data doesn't perform as well using a distributed database. With many tables and high throughput, denormalizing the data in a distributed SQL database may be a solution.
- **When you have low query and transaction rates:** Distributed databases tend to be more costly than single database server solutions. When your transaction or query

rate is less than 10,000 queries per second (QPS) or 5,000 transactions per second (TPS), a distributed SQL database may not be the answer.

- **When you have primarily JSON data that can't be represented as tables:** When you need to store complex JSON documents, subdocuments, and fields with no reasonable way to map them to tables, a distributed SQL database may not perform as well as a dedicated NoSQL document database.

  REMEMBER

  Having JSON data doesn't mean using distributed SQL is wrong. Distributed SQL databases can handle JSON fields and also represent tables as JSON.
- **When the application is purely analytical:** Distributed SQL databases weren't created to handle heavy analysis. They're designed for transactional data. While MariaDB Xpand has some analytical capabilities with columnar indexes, it shouldn't be used for complex analytical queries that extract large quantities of data for a relatively small number of users.

  TIP

  For heavy analysis, consider using a dedicated analytical database such as MariaDB ColumnStore.
- **When data consistency isn't an issue:** In a scenario where data consistency (expecting all versions of the data to be the same) isn't important, database solutions exist with relaxed consistency. This is particularly true for read-only databases where data rarely, if ever, changes. Distributed SQL databases may reduce architecture complexity but have strict consistency requirements.

REMEMBER

The reasons in this list don't make up a definitive checklist, but they can help narrow your choice.

Exploring Why You May Need Distributed SQL

Distributed SQL databases offer a number of advantages over traditional SQL databases or NoSQL solutions. Choosing distributed SQL makes sense for the following reasons:

- **You have a large database.** When you start reaching the maximum capacity of databases such as MySQL or start to hit the 50 terabyte (TB) mark with an Oracle database, things start to get a little hairy. There are scalability solutions (check out Chapter 2), but each of these performance solutions begins to involve extra labor and may create bottlenecks. Because distributed SQL databases spread the data across nodes, they can use more resources and handle larger database sizes without intervention by the application developer.

TIP

When your database tables have hundreds of millions of rows, consider using a distributed SQL database.

- **You have a high number of client connections.** In applications that carry many simultaneous client connections, a server failure means a high client reconnection requirement. Your operations team needs to regulate cold-start performance against the number of connections your database can open at once. By spreading those connections across more nodes, distributed SQL databases can handle more connections opened and sustained at once.

  TIP

  Use multiple database proxies to balance the load and reduce the chance of bottlenecks.

- **You need high availability with multi-zone failure tolerance.** While it's possible to handle faults using read replicas and electing a new primary with specialized software, such as MariaDB MaxScale, performance always takes a hit. Because distributed SQL databases don't have primary nodes and every node can handle traffic, client interruption is avoided. Data traffic can simply be redirected to other instances. The recovery process, copying data to additional nodes, is automatic.
- **Your application is read-heavy.** While a primary database that uses multiple read replicas can handle a lot of reads, it doesn't scale linearly. Distributed SQL databases handle more reads by making sure they're evenly distributed across the cluster.
- **You have a lot of writes.** Client-server databases can scale reads with read replicas, but most don't handle a heavy number of writes per second. Distributed SQL databases spread writes evenly to slices across the cluster. Because only replicas of a slice need to be locked, writes have little impact on data not involved in the transaction.
- **You have varying or seasonal demand.** A key benefit of distributed SQL is that you can add and remove nodes as

needed. This “elastic” scale is useful for scenarios where a site receives lots of user traffic followed by much lower demand.

- **An existing sharding solution has grown too complex.** Many people have scaled existing relational databases by manually splitting the data into shards. It is difficult to maintain the balance and distribution of data while assuring integrity. Distributed SQL databases balance and rebalance data automatically.

  REMEMBER

  Distributed SQL databases were designed for high transaction applications.

- **This is a system of record.** Systems of record are those that contain authoritative data requiring strict control over the reliability of their data. Distributed SQL databases were designed to be atomicity, consistency, isolation, durability (ACID) compliant from the ground up. For this reason, they’re excellent choices for systems of record.

  WARNING

  NoSQL databases can provide fault tolerance and scale. However, they don’t operate the same way that traditional systems of record operate. Transactions generally don’t perform as well compared to relational databases.

- **You’ve outgrown your existing relational database.** If you’re using an SQL database but need one that’s more powerful, reliable, distributes the load more evenly, and scales reads and writes, a distributed SQL database is a natural fit.

  TIP

  When migrating from a MySQL or MariaDB database, MariaDB Xpand is highly compatible. If migrating from databases such as Oracle or PostgreSQL, they have features that make migration to a distributed SQL database fairly painless.

- **You’re migrating to the public cloud.** The MariaDB Xpand distributed SQL database is designed to use availability zones (AZs) that give your application resiliency and edge computing advantages. While you can host MySQL, MariaDB, PostgreSQL, and even Oracle or SQL Server instances in a public cloud, they aren’t architected for AZs. You can get them to work, but it’s not a clean fit. Distributed SQL databases are designed to be self-healing even if an AZ goes down.

  AZs can be geographically located near your users, which improves application efficiency.

IN THIS CHAPTER

» Loading data in parallel

» Disabling the rebalancer

» Using MaxScale for distribution

» Extending your global reach

» Simplifying with SkySQL

Chapter 6 Ten Tips for Success with Distributed SQL Applications

After you've gotten started with distributed SQL, you can do several things to improve performance and ensure that your applications are optimized. In this chapter, here are ten tips to ensure that success.

Load Data in Parallel

With MariaDB Xpand, you can load your data in parallel by using Xpand data import tools and pre-slice tables. This process saves you valuable time. This section discusses how you go about pre-slicing tables.

TIP

By default, Xpand uses a slice size of 8 gigabytes (GB) (controlled by the split_threshold_kb global variable) and sets tables to have the same slices as the number of nodes (hash_dist_min_slices=0). If the estimated size of the data you're importing is more than 8GB times the number of nodes, you may want to pre-slice the tables. This step speeds up data loads by allowing for additional parallelism, so the rebalancer doesn't try to reslice the data as it's being loaded.

To set slice count at table creation time, simply append SLICES=N to the end of your CREATE statement. You can also reslice the table with ALTER TABLE table_name SLICES=N. Note that in both cases the slicing for the base representation and all indexes are also set to N.

Disable the Rebalancer

The rebalancer, by default, is constantly making sure that your data is spread evenly across nodes. This process creates some load on your servers.

TIP

To improve performance, disable the rebalancer during peak hours or data loads, so it runs during off-peak hours. Chances are good that your data won't change that drastically between rebalancing periods. Save the rebalancing until times when your system is generally not under load.

Apply Concurrency When Load Testing

Concurrency, often defined as the number of simultaneous users, is an important factor in load testing. When testing, increase your concurrency setting in reasonable increments, like 16, 32, 64, and 128 threads, to help you find the performance plateau more easily than, for example, changing from 100 to 1,000 threads. It may also help in determining whether one of your other test components may be hitting a bottleneck when you're making smaller increases in concurrency.

Use MaxScale for Load Balancing

MariaDB MaxScale is a database proxy extending the high availability, scalability, and security of MariaDB Xpand. At the same time, it simplifies application development by decoupling it from the underlying database infrastructure. Here are some of the MaxScale features:

- Load balancing so connections are evenly distributed across an Xpand cluster
- Automatic transaction replay if a node becomes unavailable
- Automatically detects node additions/removals and adds them to the load balancing pool
- Improved select performance by caching and result reuse via the MaxScale Cache filter

These features and others such as security and traffic control make using MaxScale a bit like having a super power.

Use AUTO_UNIQUE to Generate Unique Keys

TIP

For applications that require a unique, auto-generated values and a high degree of parallelism and concurrency, AUTO_UNIQUE provides better performance than AUTO_INCREMENT. The reason for better performance in AUTO_UNIQUE is that AUTO_INCREMENT requires the server to guarantee that each value is monotonically increasing and consecutive (in order and increasing in the amount specified or 1 by default). AUTO_UNIQUE guarantees unique values and allows Xpand to not serialize on a sequence value, unlocking additional concurrency and scale.

Use Columnar Indexes to Increase Aggregation and Analytical Query Efficiency

Columnar indexes in MariaDB Xpand can be used for optimizing aggregation queries. Xpand allows you to maintain more of your data in your operational database and run real-time analytical queries on it. By creating a columnar index on your row-based data, you can run queries that efficiently scan your transactional data, saving on unwieldy and time-consuming extract, transform, load (ETL) processes to a separate system.

You can create a columnar index with MariaDB Xpand by using the CREATE COLUMNAR INDEX statement:

```
CREATE COLUMNAR INDEX idx_invoices_date_total
   ON hq_sales.invoices (invoice_date, invoice_ ↩
  total);
```

TIP

MariaDB recommends loading data into the table prior to adding a columnar index for best performance.

Use Parallel Replication for Multiple Regions

While clusters are deployed in a single region, sometimes it makes sense to copy to multiple regions for either performance or disaster recovery purposes. With MariaDB Xpand's parallel replication, you can implement truly scale-out replication, both for the primary and the replica cluster. Xpand's parallel replication maintains transactional order preserving consistency while ensuring replication keeps pace with write throughput.

Use SkySQL for Simplicity

SkySQL is a simple-to-use database-as-a-service that lets you deploy and manage Xpand distributed SQL, columnstore analytical databases, and MariaDB Enterprise Server databases. SkySQL combines automation with human expertise to support and manage mission-critical deployments.

TIP

Use MariaDB SkySQL to gain fast and easy control over your database, get built-in high availability, and clear visibility into performance and managing your database.

Add Nodes for More Performance

One of the standard ways to scale an application is to add more nodes. With MariaDB Xpand, you can simply add nodes to get more database capacity as well. Xpand automatically manages distributing the data across the new nodes, all while staying online.

WARNING

Adding nodes can be a more expensive option when adding physical hardware. You may want to consider using cloud infrastructure to make adding nodes (or even subtracting nodes when they aren't needed) more cost effective and simple.

Use MariaDB Support

The MariaDB support and services team can help you with your projects and pain points. The team provides skills and expertise in deploying, maintaining, and scaling distributed SQL systems and can help train your team and provide ongoing support as your projects grow. To find out more information, visit `mariadb.com/services/technical-support-services`.

MariaDB

XPAND INTO THE SKY

Distributed SQL now available in **SkySQL**®

Get started with a $500 credit:
mariadb.com/tryxpand

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.

Scale large databases with distributed SQL

Distributed SQL gives you cloud-level scalability while lowering the workload for your data teams. You discover where distributed SQL fits among other scaling methods and how it's an improvement over other methods that require skill and manhours to maintain. Take advantage of the cloud-centric architecture that makes adding nodes simple and inexpensive.

Inside...

- Seeing SQL in the real world
- Understanding where to start
- Creating distributed SQL applications
- Figuring out if distributed SQL is for you
- Getting tips for success

MariaDB

Andrew C. Oliver is a Senior Director of Product Marketing for MariaDB and has a long history in open source, database, and cloud computing. **Ted Coombs** is a multi-disciplinary scientist with a long IT career who has authored many books over the last three decades.

ISBN: 978-1-394-15978-9
Not For Resale

9 781394 159789

for dummies®
A Wiley Brand