O'REILLY®

Fourth Edition

High Performance MySQL

Optimization, Backups, and Replication

Early Release

RAW & UNEDITED

Silvia Botros & Jeremy Tinley

1. 1. MySQL Architecture
    a. MySQL’s Logical Architecture
        i. Connection Management and Security
        ii. Optimization and Execution
    b. Concurrency Control
        i. Read/Write Locks
        ii. Lock Granularity
    c. Transactions
        i. Isolation Levels
        ii. Deadlocks
        iii. Transaction Logging
        iv. Transactions in MySQL
    d. Multiversion Concurrency Control
    e. Replication
    f. MySQL’s Storage Engines
        i. The InnoDB Engine
        ii. JSON document support

        - iii. Other Built-in MySQL Engines
        - iv. Summary

2. 2. Scaling MySQL
    - a. What Is Scaling?
    - b. Read Versus Write Bound Loads
    - c. Scaling Reads with Read Pools
        - i. Managing Configuration for Read Pools
        - ii. Health Checks for Read Pools
        - iii. Choosing a Load Balancing Algorithm
    - d. Queuing
    - e. Sharding
        - i. Choosing a Partitioning Scheme
        - ii. Multiple Partitioning Keys
        - iii. Querying Across Shards
        - iv. Vitess
        - v. ProxySQL
    - f. Summary

High Performance MySQL

FOURTH EDITION

Optimization, Backups, and Replication

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Silvia Botros and Jeremy Tinley

High Performance MySQL

by Silvia Botros and Jeremy Tinley

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Virginia Wilson and Andy Kwan

Production Editor: Deborah Baker

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

January 2022: Fourth Edition

Revision History for the Early Release

- 2020-01-04: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492080510 for release details.

The views expressed in this work are those of the authors, and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies

978-1-492-08051-0

[LSI]

Chapter 1. MySQL Architecture

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 1 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

MySQL's architectural characteristics make it useful for a wide range of purposes. Although it is not perfect, it is flexible enough to work well in both small and large environments. These range from a personal website up to large scale enterprise applications. To get the most from MySQL, you need to understand its design so that you can work with it, not against it.

This chapter provides a high-level overview of the MySQL server architecture, the major differences between the storage engines, and why those differences are important. We've tried to explain MySQL by simplifying the details and showing examples. This discussion will be useful for those new to database servers as well as readers who are experts with other database servers.

MySQL’s Logical Architecture

A good mental picture of how MySQL’s components work together will help you understand the server. Figure 1-1 shows a logical view of MySQL’s architecture.

The topmost layer contains the services that aren’t unique to MySQL. They’re services most network-based client/server tools or servers need: connection handling, authentication, security, and so forth.

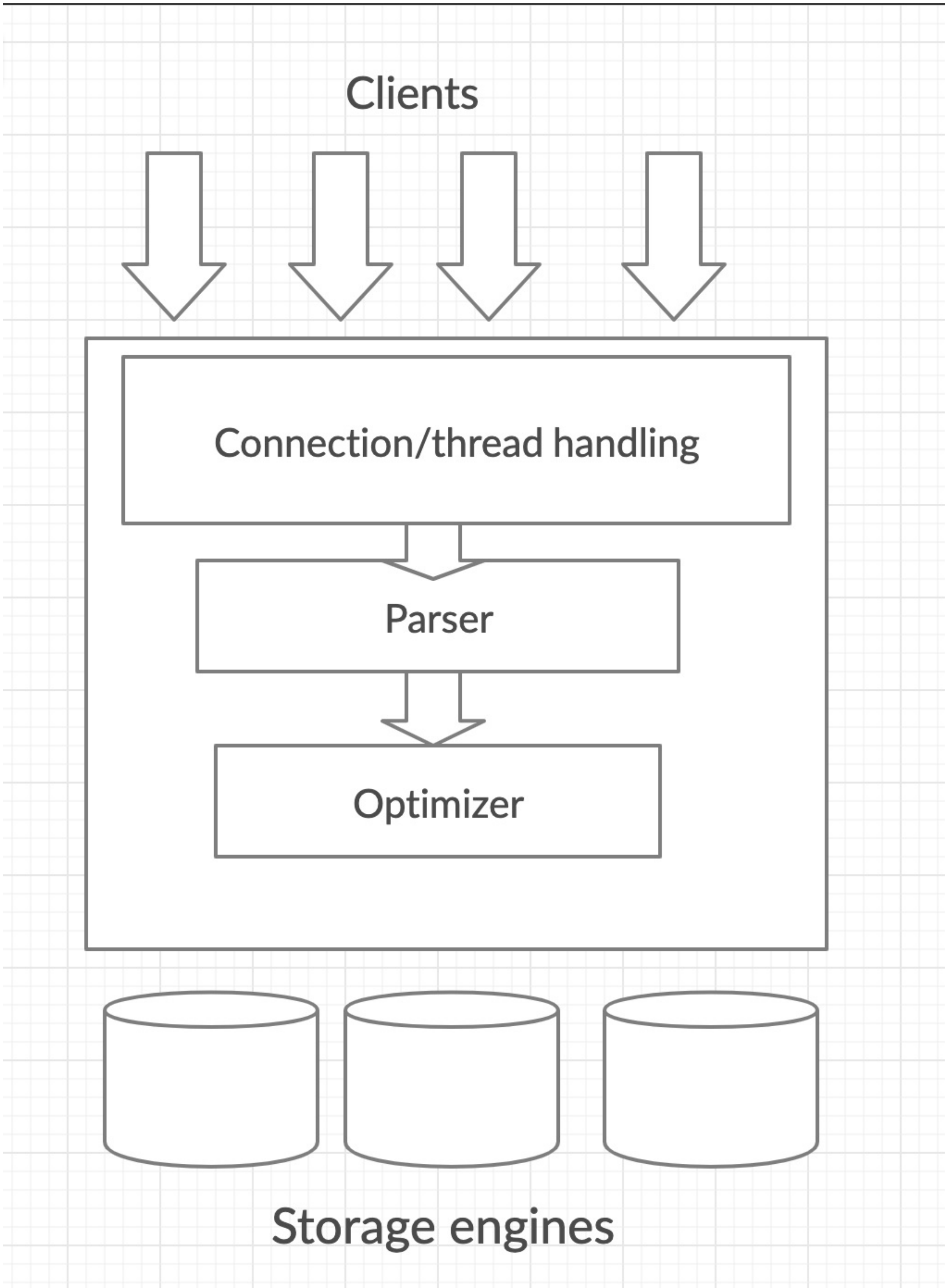

Figure 1-1. A logical view of the MySQL server architecture

The second layer is where things get interesting. Much of

MySQL’s brains are here, including the code for query parsing, analysis, optimization, caching, and all the built-in functions (e.g., dates, times, math, and encryption). Any functionality provided across storage engines lives at this level: stored procedures, triggers, and views, for example.

The third layer contains the storage engines. They are responsible for storing and retrieving all data stored “in” MySQL. Like the various file systems available for GNU/Linux, each storage engine has its own benefits and drawbacks. The server communicates with them through the *storage engine API*. This interface hides differences between storage engines and makes them largely transparent at the query layer. The API contains a couple of dozen low-level functions that perform operations such as “begin a transaction” or “fetch the row that has this primary key.” The storage engines don’t parse SQL[1] or communicate with each other; they simply respond to requests from the server.

Connection Management and Security

By default, each client connection gets its own thread within the server process. The connection’s queries execute within that single thread, which in turn resides on one core or CPU. The server caches threads, so they don’t need to be created and destroyed for each new connection.[2]

When clients (applications) connect to the MySQL server, the server needs to authenticate them. Authentication is based on username, originating host, and password. X.509 certificates can also be used across a TLS (Transport Layer Security) connection. Once a client has connected, the server verifies whether the client has privileges for each query it issues (e.g., whether the client is allowed to issue a SELECT statement that accesses the Country table in the world database).

Optimization and Execution

MySQL parses queries to create an internal structure (the parse tree), and then applies a variety of optimizations. These can include rewriting the query, determining the order in which it will read tables, choosing which indexes to use, and so on. You can pass hints to the optimizer through special keywords in the query, affecting its decision-making process. You can also ask the server to explain various aspects of optimization. This lets you know what decisions the server is making and gives you a reference point for reworking queries, schemas, and settings to make everything run as efficiently as possible.

The optimizer does not really care what storage engine a particular table uses, but the storage engine does affect how the server optimizes the query. The optimizer asks the storage engine about some of its capabilities and the cost of certain operations, and for statistics on the table data. For instance, some storage engines support index types that can be helpful to certain queries. You can read more about indexing and schema optimization in Chapter 8.

In older versions, MySQL made use of an internal query cache to see if it can serve the results from there. However, as concurrency increased, the query cache became a notorious bottleneck. As of MySQL 5.7.20 the query cache was officially deprecated as a MySQL feature and in the 8.0 release, the query cache is fully removed. Even though the query cache is no longer a core part of the MySQL server, caching frequently served result sets is a good practice. In Chapter 5 we will go over ways you can implement such a cache using different technologies.

Concurrency Control

Anytime more than one query needs to change data at the same time, the problem of concurrency control arises. For our purposes in this chapter, MySQL has to do this at two levels: the server level and the storage engine level. Concurrency control is a big topic to which a large body of theoretical literature is devoted, so we will just give you a simplified overview of how MySQL deals with concurrent readers and writers, so you have the context you need for the rest of this chapter.

To illustrate how MySQL handles concurrent work on the same set of data, we will use a traditional spreadsheet file as an example. A spreadsheet consists of rows and columns, much like a database table. Assume the file is on your laptop and only you have access to it. There are no potential conflicts, only you can make changes to the file. Now, imagine you need to collaborate with a coworker on that spreadsheet. It is now on a shared server that both of you have access to. What happens when both of you need to make changes to this file at the same time? What if we have an entire team of people actively trying to edit, add and remove cells from this spreadsheet? We can say that they should take turns making changes but that is not efficient. We need an approach for allowing concurrent access to a high volume spreadsheet.

Read/Write Locks

Reading from the spreadsheet isn't as troublesome. There's nothing wrong with multiple clients reading the same file simultaneously; because they aren't making changes, nothing is likely to go wrong. What happens if someone tries to delete cell number A25 while others are reading the spreadsheet? It depends, but a reader could come away with a corrupted or inconsistent view of the data. So, to be safe, even reading from a spreadsheet requires special care.

If you think of the spreadsheet as a database table, it's easy to see that the problem is the same in this context. In many ways, a spreadsheet is really just a simple database table. Modifying rows in a database table is very similar to removing or changing the content of cells in a spreadsheet file.

The solution to this classic problem of concurrency control is rather simple. Systems that deal with concurrent read/write access typically implement a locking system that consists of two lock types. These locks are usually known as *shared locks* and *exclusive locks*, or *read locks* and *write locks*.

Without worrying about the actual locking mechanism, we can describe the concept as follows. Read locks on a resource are shared, or mutually non-blocking: many clients can read from a resource at the same time and not interfere with each other. Write locks, on the other hand, are exclusive—i.e., they block both read locks and other write locks—because the only safe policy is to have a single client writing to the resource at a given time and to prevent all reads when a client is writing.

In the database world, locking happens all the time: MySQL has to prevent one client from reading a piece of data while another is changing it. It performs this lock management internally in a way that is transparent much of the time.

Lock Granularity

One way to improve the concurrency of a shared resource is to be more selective about what you lock. Rather than locking the entire resource, lock only the part that contains the data you need to change. Better yet, lock only the exact piece of data you plan to change. Minimizing the amount of data that you lock at any one time lets changes to a given resource occur simultaneously, as long as they don't conflict with each other.

Unfortunately, locks are not free - they consume resources. Every lock operation—getting a lock, checking to see whether a lock is free, releasing a lock, and so on—has overhead. If the system spends too much time managing locks instead of storing and retrieving data, performance can suffer.

A locking strategy is a compromise between lock overhead and data safety, and that compromise affects performance. Most commercial database servers don't give you much choice: you get what is known as row-level locking in your tables, with a variety of often complex ways to give good performance with many locks.

MySQL, on the other hand, does offer choices. Its storage engines can implement their own locking policies and lock granularities. Lock management is a very important decision in storage engine design; fixing the granularity at a certain level can improve performance for certain uses, yet make that engine less suited for other purposes. Because MySQL offers multiple storage engines, it doesn't require a single general-purpose solution. Let's have a look at the two most important lock strategies.

TABLE LOCKS

The most basic locking strategy available in MySQL, and the one with the lowest overhead, is *table locks*. A table lock is analogous to the spreadsheet locks described earlier: it locks the entire table. When a client wishes to write to a table (insert, delete, update, etc.), it acquires a write lock. This keeps all other read and write operations at bay. When nobody is writing, readers can obtain read locks, which don't conflict with other read locks.

Table locks have variations for improved performance in specific situations. For example, READ LOCAL table locks allow some types of concurrent write operations. Write locks also have a higher priority than read locks, so a request for a write lock will advance to the front of the lock queue even if readers are already in the queue (write locks can advance past read locks in the queue, but read locks cannot advance past write locks).

Although storage engines can manage their own locks, MySQL itself also uses a variety of locks that are effectively table-level for various purposes. For instance, the server uses a table-level lock for statements such as ALTER TABLE, regardless of the storage engine.[3]

ROW LOCKS

The locking style that offers the greatest concurrency (and carries the greatest overhead) is the use of *row locks*. Going back to the spreadsheet analogy, row locks would be the same as locking just the row in the spreadsheet. This strategy allows multiple people to edit different rows concurrently without blocking each other. This enables the server to take more concurrent writes, but the cost is more overhead in having to keep track of who has each row lock, how long they have been open, what kind of row lock it is and cleaning up locks when they are no longer needed.

Row locks are implemented in the storage engine, not the server. The server is mostly[4] unaware of locks implemented in the storage engines, and as you'll see later in this chapter and throughout the book, the storage engines all implement locking in their own ways.

Transactions

You can’t examine the more advanced features of a database system for very long before *transactions* enter the mix. A transaction is a group of SQL queries that are treated *atomically*, as a single unit of work. If the database engine can apply the entire group of queries to a database, it does so, but if any of them can’t be done because of a crash or other reason, none of them is applied. It’s all or nothing.

Little of this section is specific to MySQL. If you’re already familiar with ACID transactions, feel free to skip ahead to “Transactions in MySQL”.

A banking application is the classic example of why transactions are necessary. Imagine a bank’s database with two tables: checking and savings. To move $200 from Jane’s checking account to her savings account, you need to perform at least three steps:

1. Make sure her checking account balance is greater than $200.
2. Subtract $200 from her checking account balance.
3. Add $200 to her savings account balance.

The entire operation should be wrapped in a transaction so that if any one of the steps fails, any completed steps can be rolled back.

You start a transaction with the START TRANSACTION statement and then either make its changes permanent with COMMIT or discard the changes with ROLLBACK. So, the SQL for our sample transaction might look like this:

```
1 START TRANSACTION;
2 SELECT balance FROM checking WHERE customer_id =
10233276;
3 UPDATE checking SET balance = balance - 200.00 WHERE
customer_id = 10233276;
4 UPDATE savings SET balance = balance + 200.00 WHERE
customer_id = 10233276;
5 COMMIT;
```

But transactions alone aren’t the whole story. What happens if the database server crashes while performing line 4? Who knows? The customer probably just lost $200. And what if another process comes along between lines 3 and 4 and removes the entire checking account balance? The bank has given the customer a $200 credit without even knowing it.

Transactions aren’t enough unless the system passes the *ACID test*. ACID stands for Atomicity, Consistency, Isolation, and Durability. These are tightly related criteria that a well-behaved transaction processing system must meet:

Atomicity

> A transaction must function as a single indivisible unit of work so that the entire transaction is either

applied or rolled back. When transactions are atomic, there is no such thing as a partially completed transaction: it's all or nothing.

Consistency

The database should always move from one consistent state to the next. In our example, consistency ensures that a crash between lines 3 and 4 doesn't result in $200 disappearing from the checking account. Because the transaction is never committed, none of the transaction's changes are ever reflected in the database.

Isolation

The results of a transaction are usually invisible to other transactions until the transaction is complete. This ensures that if a bank account summary runs after line 3 but before line 4 in our example, it will still see the $200 in the checking account. When we discuss isolation levels, you'll understand why we said *usually* invisible.

Durability

Once committed, a transaction's changes are permanent. This means the changes must be recorded such that data won't be lost in a system crash. Durability is a slightly fuzzy concept, however, because there are actually many levels. Some durability strategies provide a stronger safety guarantee than others, and nothing is ever 100% durable (if the database itself were truly durable, then how could backups increase durability?). We discuss what durability *really* means in MySQL in Chapter 6.

ACID transactions and the guarantees provided through them in the InnoDB engine specifically are one of the strongest and most mature features in MySQL. While they come with certain throughput trade offs, when applied appropriately, they can save you from implementing a lot of complex logic in the application layer.

Isolation Levels

Isolation is more complex than it looks. The ANSI SQL standard defines four isolation levels. If you are new to the world of databases, we highly recommend you get familiar with the general standard of ANSI SQL[5] before coming back to reading about the specific MySQL implementation. The goal of this standard is to define the rules for which changes are and aren't visible inside and outside a transaction. Lower isolation levels typically allow higher concurrency and have lower overhead.

NOTE

Each storage engine implements isolation levels slightly differently, and they don't necessarily match what you might expect if you're used to another database product (thus, we won't go into exhaustive detail in this section). You should read the manuals for whichever storage engines you decide to use.

Let's take a quick look at the four isolation levels:

READ UNCOMMITTED

In the READ UNCOMMITTED isolation level, transactions can view the results of uncommitted transactions. At this level, many problems can occur unless you really, really know what you are doing and have a good reason for doing it. This level is rarely used in practice, because its performance isn't much better than the other levels, which have many advantages. Reading uncommitted data is also known as a *dirty read*.

READ COMMITTED

The default isolation level for most database systems (but not MySQL!) is READ COMMITTED. It satisfies the simple definition of isolation used earlier: a transaction will see only those changes made by transactions that were already committed when it began, and its changes won't be visible to others until it has committed. This level still allows what's known as a *non-repeatable read*. This means you can run the same statement twice and see different data.

REPEATABLE READ

REPEATABLE READ solves the problems that READ UNCOMMITTED allows. It guarantees that any rows a transaction reads will “look the same” in subsequent reads within the same transaction, but in theory it still allows another tricky problem: *phantom reads*. Simply put, a phantom read can happen when you select some range of rows, another transaction inserts a new row into the range, and then you select the same range again; you will then see the new “phantom” row. InnoDB and XtraDB solve the phantom read problem with multiversion concurrency control, which we explain later in this chapter. REPEATABLE READ is MySQL’s default transaction isolation level.

SERIALIZABLE

The highest level of isolation, SERIALIZABLE, solves the phantom read problem by forcing transactions to be ordered so that they can’t possibly conflict. In a nutshell, SERIALIZABLE places a lock on every row it reads. At this level, a lot of timeouts and lock contention can occur. We’ve rarely seen people use this isolation level, but your application’s needs might force you to accept the decreased concurrency in favor of the data stability that results.

Table 1-1 summarizes the various isolation levels and the drawbacks associated with each one.

Table 1-1. ANSI SQL isolation levels

Isolation level	Dirty reads possible	Non-repeatable reads possible	Phantom reads possible	Locking reads
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

Deadlocks

A *deadlock* is when two or more transactions are mutually holding and requesting locks on the same resources, creating a cycle of dependencies. Deadlocks occur when transactions try to lock resources in a different order. They can happen whenever multiple transactions lock the same resources. For example, consider these two transactions running against the StockPrice table:

Example 1-1. First Transaction

```
          START TRANSACTION;
          UPDATE StockPrice SET close = 45.50 WHERE
stock_id = 4 and date = '2020-05-01';
          UPDATE StockPrice SET close = 19.80 WHERE
stock_id = 3 and date = '2020-05-02';
          COMMIT;
```

Example 1-2. Second Transaction

```
          START TRANSACTION;
          UPDATE StockPrice SET high = 20.12 WHERE
stock_id = 3 and date = '2020-05-02';
          UPDATE StockPrice SET high = 47.20 WHERE
stock_id = 4 and date = '2020-05-01';
          COMMIT;
```

Each transaction will execute its first query and update a row of data, locking it in the process. Each transaction will then attempt to update its second row, only to find that it is already locked. The two transactions will wait forever for each other to complete, unless something intervenes to break the deadlock.

To combat this problem, database systems implement various forms of deadlock detection and timeouts. The more sophisticated systems, such as the InnoDB storage engine, will notice circular dependencies and return an error instantly. This can be a good thing—otherwise, deadlocks would manifest themselves as very slow queries. Others will give up after the query exceeds a lock wait timeout, which is not always good. The way InnoDB currently handles deadlocks is to roll back the transaction that has the fewest exclusive row locks (an approximate metric for which will be the easiest to roll back).

Lock behavior and order are storage engine–specific, so some storage engines might deadlock on a certain sequence of statements even though others won't. Deadlocks have a dual nature: some are unavoidable because of true data conflicts, and some are caused by how a storage engine works.

Once they occur, deadlocks cannot be broken without rolling back one of the transactions, either partially or wholly. They are a fact of life in transactional systems, and your applications should be designed to handle them. Many applications can simply retry their transactions from the beginning.

Transaction Logging

Transaction logging helps make transactions more efficient. Instead of updating the tables on disk each time a change occurs, the storage engine can change its in-memory copy of the data. This is very fast. The storage engine can then write a record of the change to the transaction log, which is on disk and therefore durable. This is also a relatively fast operation, because appending log events involves sequential I/O in one small area of the disk instead of random I/O in many places. Then, at some later time, a process can update the table on disk. Thus, most storage engines that use this technique (known as *write-ahead logging*) end up writing the changes to disk twice.

If there's a crash after the update is written to the transaction log but before the changes are made to the data itself, the storage engine can still recover the changes upon restart. The recovery method varies between storage engines.

Transactions in MySQL

Storage engines are the software that drives how data will be stored and retrieved from disk. While MySQL has traditionally offered a number of storage engines that support transactions, InnoDB is now the golden standard and the recommended engine to use. Transaction primitives described here will be based on transactions in the InnoDB engine.

AUTOCOMMIT

By default, a single INSERT, UPDATE or DELETE statement is implicitly wrapped in a transaction and committed immediately. This is known as AUTOCOMMIT mode. By disabling this mode, you can execute a series of statements within a transaction and at conclusion, COMMIT or ROLLBACK.

You can enable or disable the AUTOCOMMIT variable for the current connection by using a SET command. The values 1 and ON are equivalent, as are 0 and OFF. When you run with AUTOCOMMIT=0, you are always in a transaction, until you issue a COMMIT or ROLLBACK. MySQL then starts a new transaction immediately. Additionally, with AUTOCOMMIT enabled, you can begin a multi-statement transaction by using they keyword BEGIN or START TRANSACTION. Changing the value of AUTOCOMMIT has no effect on non-transactional tables which have no notion of committing or rolling back changes.

Certain commands, when issued during an open transaction, cause MySQL to commit the transaction before they execute. These are typically Data Definition Language (DDL) commands that make significant changes, such as ALTER TABLE, but LOCK TABLES and some other statements also have this effect. Check your version's documentation for the full list of commands that automatically commit a transaction.

MySQL lets you set the isolation level using the SET TRANSACTION ISOLATION LEVEL command, which takes effect when the next transaction starts. You can set the isolation level for the whole server in the configuration file, or just for your session:

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ
COMMITTED;
```

MySQL recognizes all four ANSI standard isolation levels, and InnoDB supports all of them.

MIXING STORAGE ENGINES IN TRANSACTIONS

MySQL doesn't manage transactions at the server level. Instead, the underlying storage engines implement transactions themselves. This means you can't reliably mix different engines in a single transaction.

If you mix transactional and non transactional tables (for instance, InnoDB and MyISAM tables) in a transaction, the transaction will work properly if all goes well.

However, if a rollback is required, the changes to the non transactional table can't be undone. This leaves the database in an inconsistent state from which it might be difficult to recover and renders the entire point of transactions moot. This is why it is really important to pick the right storage engine for each table and to avoid mixing storage engines in your application logic at all costs.

MySQL will not usually warn you or raise errors if you do transactional operations on a non transactional table. Sometimes rolling back a transaction will generate the warning "Some non transactional changed tables couldn't be rolled back," but most of the time, you'll have no indication you're working with non transactional tables.

> **WARNING**
>
> It is best practice to not mix storage engines in your application. Failed transactions can lead to inconsistent results as some parts can roll back and others cannot.

IMPLICIT AND EXPLICIT LOCKING

InnoDB uses a two-phase locking protocol. It can acquire locks at any time during a transaction, but it does not release them until a COMMIT or ROLLBACK. It releases all the locks at the same time. The locking mechanisms described earlier are all implicit. InnoDB handles locks automatically, according to your isolation level.

However, InnoDB also supports explicit locking, which the SQL standard does not mention at all:[6]

```
SELECT ... FOR SHARE
SELECT ... FOR UPDATE
```

(

SELECT … FOR SHARE is a MySQL 8.0 feature which replaces SELECT … LOCK IN SHARE MODE of previous versions.

)

MySQL also supports the LOCK TABLES and UNLOCK TABLES commands, which are implemented in the server, not in the storage engines. These have their uses, but they are not a substitute for transactions. If you need transactions, use a transactional storage engine. Because InnoDB supports row level locking, LOCK TABLES is unnecessary.

TIP

The interaction between LOCK TABLES and transactions is complex, and there are unexpected behaviors in some server versions. Therefore, we recommend that you never use LOCK TABLES unless you are in a transaction and AUTOCOMMIT is disabled, no matter what storage engine you are using.

Multiversion Concurrency Control

Most of MySQL's transactional storage engines don't use a simple row-locking mechanism. Instead, they use row-level locking in conjunction with a technique for increasing concurrency known as *multiversion concurrency control* (MVCC). MVCC is not unique to MySQL: Oracle, PostgreSQL, and some other database systems use it too, although there are significant differences because there is no standard for how MVCC should work.

You can think of MVCC as a twist on row-level locking; it avoids the need for locking at all in many cases and can have much lower overhead. Depending on how it is implemented, it can allow nonlocking reads, while locking only the necessary rows during write operations.

MVCC works by keeping a snapshot of the data as it existed at some point in time. This means transactions can see a consistent view of the data, no matter how long they run. It also means different transactions can see different data in the same tables at the same time! If you've never experienced this before, it might be confusing, but it will become easier to understand with familiarity.

Each storage engine implements MVCC differently. Some of the variations include *optimistic* and *pessimistic* concurrency control. We’ll illustrate one way MVCC works by explaining a simplified version of InnoDB’s behavior.

InnoDB implements MVCC by storing with each row two additional, hidden values that record when the row was created and when it was expired (or deleted). Rather than storing the actual times at which these events occurred, the row stores the system version number at the time each event occurred. This is a number that increments each time a transaction begins. Each transaction keeps its own record of the current system version, as of the time it began. Each query has to check each row’s version numbers against the transaction’s version. Let’s see how this applies to particular operations when the transaction isolation level is set to REPEATABLE READ:

SELECT

> InnoDB must examine each row to ensure that it meets two criteria: InnoDB must find a version of the row that is at least as old as the transaction (i.e., its version must be less than or equal to the transaction’s version). This ensures that either the row existed before the transaction began, or the transaction created or altered the row. The row’s deletion version must be undefined or greater than the transaction’s version. This ensures that the row wasn’t deleted before the transaction began. Rows that pass both tests may be returned as the query’s result.

INSERT

InnoDB records the current system version number with the new row.

DELETE

InnoDB records the current system version number as the row’s deletion ID.

UPDATE

InnoDB writes a new copy of the row, using the system version number for the new row’s version. It also writes the system version number as the old row’s deletion version.

The result of all this extra record keeping is that most read queries never acquire locks. They simply read data as fast as they can, making sure to select only rows that meet the criteria. The drawbacks are that the storage engine has to store more data with each row, do more work when examining rows, and handle some additional housekeeping operations.

MVCC works only with the REPEATABLE READ and READ COMMITTED isolation levels. READ UNCOMMITTED isn’t MVCC-compatible[7] because queries don’t read the row version that’s appropriate for their transaction version; they read the newest version, no matter what. SERIALIZABLE isn’t MVCC-compatible because reads lock every row they return.

Replication

MySQL is designed for accepting writes on one node at any given time. This has advantages in managing consistency but leads to trade offs when you need the data written in multiple servers or multiple locations. MySQL offers a native way to distribute writes that one node takes to additional nodes. This is referred to as replication. In MySQL, replication works in a “pull model” meaning replicas periodically query the source host for the latest binlog location and pull binary logs as needed. In this diagram we show a simple example of this setup usually called “a topology tree” of multiple MySQL servers in a source and replica setup.

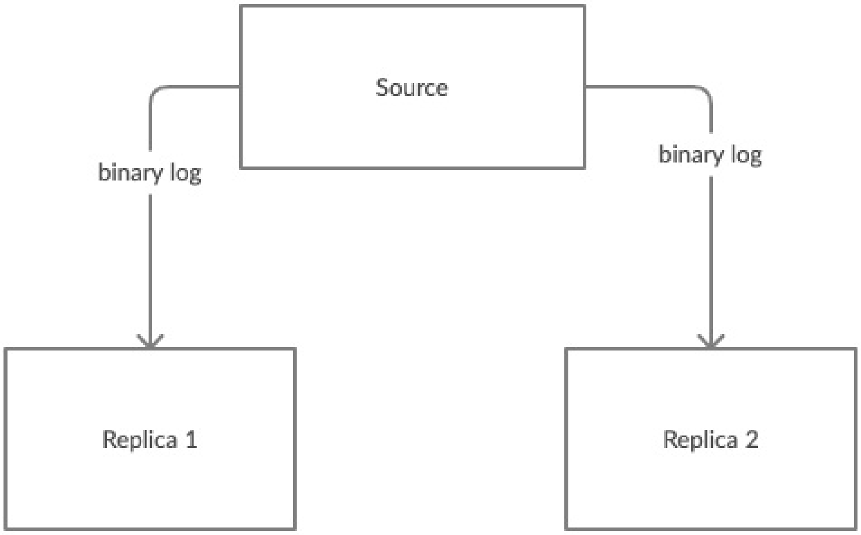

Figure 1-2. A simplified view of a MySQL server replication topology

For any data you run in production, you should use replication and have at least 3 more replicas, ideally distributed in different locations (in cloud hosted environments, known as *regions*) for disaster recovery planning.

Over the years, replication in MySQL gained more sophistication. From global transaction identifiers, multi source replication, parallel replication on replicas, and semi sync replication being some of the major updates. We will be covering replication in a lot more detail in Chapter 9.

MySQL’s Storage Engines

This section gives an overview of MySQL’s storage engines. Storage engine is the layer in MySQL that handles persistence of your data. The different engines in MySQL mean you can have different behaviours based on your application requirements. We will be talking about choosing engines in the next chapter. Since engine support and how MySQL stores information about the tables change dramatically between 5.7 and 8.0, we will refer explicitly to version names as we go on. Even this book, though, isn’t a complete source of documentation; you should read the MySQL manuals for the storage engines you decide to use.

In versions before 8.0, MySQL stored each database (also called a *schema*) as a subdirectory of its data directory in the underlying filesystem. When you created a table, MySQL stored the table definition in an *.frm* file and an *.ibd* file that stored the data. Both files are named after the table they represent. Thus, when you create a table named MyTable, MySQL stores the table definition in *MyTable.frm* and the data in *MyTable.ibd*. If you used partitions you would also see *MyTable.par*.

In version 8.0, MySQL redesigned table metadata into a data dictionary that is included with a table’s *.ibd* file. This makes information on the table structure support transactions and atomic data definition changes. Instead of relying only on information_schema for retrieving table definition and metadata during operations, we are introduced to the dictionary object cache which is an LRU based in memory cache of partition definitions, table definitions, stored program definitions, charset and collation information. This major change in how the server accesses metadata about tables reduces IO and is efficient especially if a subset of tables is what sees the most activity and therefore is in the cache most often. The *.ibd* and *.frm* files are replaced with serialized dictionary information (*.sdi*) per table.

Now that we have covered how table metadata and file structure has changed between the recent major versions, we’ll go into an overview of the most recommended and used native storage engine in MySQL, Innodb.

The InnoDB Engine

InnoDB is the default transactional storage engine for MySQL and the most important and broadly useful engine overall. It was designed for processing many short-lived transactions that usually complete rather than being rolled back. Its performance and automatic crash recovery make it popular for non-transactional storage needs, too.If you want to study storage engines, it is also well worth your time to study InnoDB in depth to learn as much as you can about it, rather than studying all storage engines equally.

> **NOTE**
>
> It is best practice to use the InnoDB storage engine as the default engine for any application

InnoDB is the default MySQL general purpose storage engine. By default, InnoDB stores its data in a series of data files that are collectively known as a *tablespace*. A tablespace is essentially a black box that InnoDB manages all by itself.

InnoDB uses MVCC to achieve high concurrency, and it implements all four SQL standard isolation levels. It defaults to the REPEATABLE READ isolation level, and it has a *next-key locking* strategy that prevents phantom reads in this isolation level: rather than locking only the rows you've touched in a query, InnoDB locks gaps in the index structure as well, preventing phantoms from being inserted.

InnoDB tables are built on a *clustered index*, which we will cover in detail in chapter 8 when we discuss schema design. InnoDB's index structures are very different from those of most other MySQL storage engines. As a result, it provides very fast primary key lookups. However, *secondary indexes* (indexes that aren't the primary key) contain the primary key columns, so if your primary key is large, other indexes will also be large. You should strive for a small primary key if you'll have many indexes on a table.

InnoDB has a variety of internal optimizations. These include predictive read-ahead for prefetching data from disk, an adaptive hash index that automatically builds hash indexes in memory for very fast lookups, and an insert buffer to speed inserts. We cover these later in this book.

InnoDB's behavior is very intricate, and we highly recommend reading the "InnoDB Locking and Transaction Model" section of the MySQL manual if you're using InnoDB. There are many subtleties you should be aware of before building an application with InnoDB, because of its MVCC architecture. Working with a storage engine that maintains consistent views of the data for all users, even when some users are changing data, can be complex.

As a transactional storage engine, InnoDB supports truly "hot" online backups through a variety of mechanisms, including Oracle's proprietary MySQL Enterprise Backup and the open source Percona XtraBackup.

Beginning with MySQL 5.6, InnoDB introduced Online DDL which at first had limited use cases that expanded in the 5.7 and 8.0 releases. In place schema changes allow for specific table changes without a full table lock and without using external tools which greatly improved the operationality of MySQL InnoDB tables. We will be covering options for online schema changes, both native and external tools in chapter 11.

JSON document support

One of the most notable additions to MySQL in 5.7 and 8.0 is extensive support of JSON data type that goes further than simple BLOB storage. Long requested and a feature in other relational database offerings for a long while, such as Postgres, this new rich datatype makes a class of business needs a first class citizen in MySQL and brings the guarantees of the InnoDB storage engine to a fast growing class of data modeling.

First introduced to InnoDB as part of the 5.7 release, the JSON type arrived with automatic validation of JSON documents, optimized storage that allowed for quick read access. A significant improvement to the tradeoffs of old style BLOB storage engineers used to resort to for JSON documents.

Along with the new datatype support, InnoDB also introduced SQL functions to support rich operations on JSON documents.

A further improvement in MySQL 8.0.7, adds the ability to define multi-valued indexes on JSON arrays. This feature can be a powerful way to even further speed read access queries to JSON types by matching the common access patterns to functions that can map the JSON document values.

Other Built-in MySQL Engines

MySQL has a variety of special-purpose storage engines. Many of them are deprecated in newer versions, for various reasons. Some of these are still available on the server, but must be enabled specially.

THE MYISAM ENGINE

MyISAM was the first engine in MySQL and was the default one until the release of MySQL 5.6. It does not support transactions or row based locks which makes for a much simpler concurrency model (only one write can ever happen at a given time) but much more limited use in modern applications. With the release of MySQL 5.6 in 2013, MyISAM was no longer the default engine in MySQL. In MySQL 8.0 it was completely removed.

THE ARCHIVE ENGINE

The Archive engine supports only INSERT and SELECT queries, and it does not support indexes until MySQL 5.1. It causes much less disk I/O than MyISAM, because it buffers data writes and compresses each row with *zlib* as it's inserted. Also, each SELECT query requires a full table scan. Archive tables are thus best for logging and data acquisition, where analysis tends to scan an entire table, or where you want fast INSERT queries.

Archive supports row-level locking and a special buffer system for high-concurrency inserts. It gives consistent reads by stopping a SELECT after it has retrieved the number of rows that existed in the table when the query began. It also makes bulk inserts invisible until they're complete. These features emulate some aspects of transactional and MVCC behaviors, but Archive is not a transactional storage engine. It is simply a storage engine that's optimized for high-speed inserting and compressed storage.

THE BLACKHOLE ENGINE

The Blackhole engine has no storage mechanism at all. It discards every INSERT instead of storing it. However, the server writes queries against Blackhole tables to its logs, so they can be replicated or simply kept in the log. That makes the Blackhole engine popular for fancy replication setups and audit logging, although we've seen enough problems caused by such setups that we don't recommend them.

THE CSV ENGINE

The CSV engine can treat comma-separated values (CSV) files as tables, but it does not support indexes on them. This engine lets you copy files into and out of the database while the server is running. If you export a CSV file from a spreadsheet and save it in the MySQL server's data directory, the server can read it immediately. Similarly, if you write data to a CSV table, an external program can read it right away. CSV tables are thus useful as a data interchange format.

THE FEDERATED ENGINE

This storage engine is sort of a proxy to other servers. It opens a client connection to another server and executes queries against a table there, retrieving and sending rows as needed. It was originally marketed as a competitor to features supported in many enterprise-grade proprietary database servers, such as Microsoft SQL Server and Oracle, but that was always a stretch, to say the least. Although it seemed to enable a lot of flexibility and neat tricks, it has proven to be a source of many problems and is disabled by default. A successor to it, FederatedX, is available in MariaDB.

THE MEMORY ENGINE

Memory tables (formerly called HEAP tables) are useful when you need fast access to data that either never changes or doesn't need to persist after a restart. Memory tables can be up to an order of magnitude faster than InnoDB tables. All of their data is stored in memory, so queries don't have to wait for disk I/O. The table structure of a Memory table persists across a server restart, but no data survives.

Here are some good uses for Memory tables:

- For "lookup" or "mapping" tables, such as a table that maps postal codes to state names
- For caching the results of periodically aggregated data
- For intermediate results when analyzing data

Memory tables support HASH indexes, which are very fast for lookup queries. Although Memory tables are very fast, they often don't work well as a general-purpose replacement for disk-based tables. They use table-level locking, which gives low write concurrency. They do not support TEXT or BLOB column types, and they support only fixed-size rows, so they really store VARCHARs as CHARs, which can waste memory. (Some of these limitations are lifted in Percona Server.)

MySQL uses the Memory engine internally while processing queries that require a temporary table to hold intermediate results. If the intermediate result becomes too large for a Memory table, or has TEXT or BLOB columns, MySQL will convert it to a MyISAM table on disk and in 8.0 will even convert it to the safer InnoDB engine. We say more about this in chapters 7 and 12.

People often confuse Memory tables with temporary tables, which are ephemeral tables created with CREATE TEMPORARY TABLE. Temporary tables can use any storage engine; they are not the same thing as tables that use the Memory storage engine. Temporary tables are visible only to a single connection and disappear entirely when the connection closes.

THE MERGE STORAGE ENGINE

The Merge engine is a variation of MyISAM. A Merge table is the combination of several identical MyISAM tables into one virtual table. This can be useful when you use MySQL in logging and data warehousing applications, but it has been deprecated in favor of partitioning (see Chapter 7).

THE NDB CLUSTER ENGINE

MySQL AB acquired the NDB database from Sony Ericsson in

2003 and built the NDB Cluster storage engine as an interface between the SQL used in MySQL and the native NDB protocol. The combination of the MySQL server, the NDB Cluster storage engine, and the distributed, shared-nothing, fault-tolerant, highly available NDB database is known as MySQL Cluster.

Now that we covered all table engines included in MySQL's releases, you should know that in chapter 2 we will cover notable third party engines and how to choose among all these options depending on your use case.

Summary

MySQL has a layered architecture, with server-wide services and query execution on top and storage engines underneath. Although there are many different plugin APIs, the storage engine API is the most important. If you understand that MySQL executes queries by handing rows back and forth across the storage engine API, you've grasped one of the core fundamentals of the server's architecture.

In the past few major releases, MySQL has settled on InnoDB as its primary development focus and has moved even its internal bookkeeping around table metadata, authentication and authorization to InnoDB after years in MyISAM. This increased investment from Oracle in the InnoDB engine has led to major improvements such as atomic DDLs, more robust Online DDLs, better resilience to crashes, and better operability for security minded deployments.

[1] One exception is InnoDB, which does parse foreign key definitions, because the MySQL server doesn’t yet implement them itself.

[2] MySQL 5.5 and newer versions support an API that can accept thread-pooling plugins though not commonly used. The common practice for thread pooling is done at access layers which we discuss in Chapter 5

[3] We will cover in chapter 13 more techniques of schema changes including ONLINE native alter statements and tools that provide online, lock free schema changes

[4] There are metadata locks which are used when dealing with table name changes or changing schemas and in 8.0 we are introduced to “application level locking functions”. But in the course of run of the mill data changes, internal locking is left to the InnoDB engine.

[5] Read a summary of ANSI SQL at by Adrian Coyler https://blog.acolyer.org/2016/02/24/a-critique-of-ansi-sql-isolation-levels/ and an explanation of consistency models by Kyle Kingsbury in http://jepsen.io/consistency

[6] These locking hints are frequently abused and should usually be avoided.

[7] There is no formal standard that defines MVCC, so different engines and databases implement it very differently, and no one can say any of them is wrong.

Chapter 2. Scaling MySQL

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Chapter 11 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Running MySQL in a personal project, or even in a young company, is very different from running it in a business with an established market and 'hockey stick growth' where traffic is growing orders of magnitude year over year, and the product complexity and accompanying data needs are accelerating.

In this chapter, we explain what scaling means and walk you through the different axes where you may need to scale. We explore why read scaling is essential and show you how to accomplish it safely, with strategies like queuing for making scaling writes more predictable. Finally, we cover sharding datasets to scale writes using tools like ProxySQL and Vitess. By the end of this chapter, you should be able to identify what seasonal pattern your system has, how to scale reads, and how to scale writes.

What Is Scaling?

Scaling is the system’s ability to support growing traffic without resorting to a linear increase of the infrastructure and headcount costs. A well scaling system needs little to no increase in infrastructure cost to support growing traffic. Systems that do not scale well reach a point of diminishing returns and can’t grow further without significant increase in cost.

Capacity is a related concept. The system’s capacity is the amount of work it can perform in a given amount of time.[1] However, capacity must be qualified. The system’s maximum throughput is not the same as its capacity. Most benchmarks measure a system’s maximum throughput, but you can’t push real systems that hard. If you do, performance will degrade, and response times will become unacceptably large and variable. We define the system’s actual capacity as the throughput it can achieve while still delivering acceptable performance.

Capacity and scalability are independent of performance. To make an analogy with cars on a highway:

- The system is the highway and all the lanes and cars in it.
- Performance is how fast the car is.
- Capacity is the number of lanes times the maximum safe speed.

- Scalability is the degree to which you can add more cars and more lanes without slowing traffic.

In this analogy, scalability depends on factors such as how well the interchanges are designed, how many cars have accidents or break down, and whether the cars drive at different speeds or change lanes a lot—but generally, scalability does not depend on how powerful the cars' engines are. This is not to say that performance doesn't matter, because it does. We're just pointing out that systems can be scalable even if they aren't high-performance.

From the 50,000-foot view, scalability is the ability to add capacity by adding resources.

Even if your MySQL architecture is scalable, your application might not be. If it's hard to increase capacity for any reason, your application isn't scalable overall. We defined capacity in terms of throughput a moment ago, but it's worth looking at capacity from the same 50,000-foot view. From this vantage point, capacity simply means the ability to handle load, and it's useful to think of load from several different angles:

Quantity of data

The sheer volume of data your application can accumulate is one of the most common scaling challenges. This is particularly an issue for many of today’s web applications, which never delete any data. Social networking sites, for example, typically never delete old messages or comments.

Number of users

Even if each user has only a small amount of data, if you have a lot of users it adds up—and the data size can grow disproportionately faster than the number of users. Many users generally means more transactions too, and the number of transactions might not be proportional to the number of users. Finally, many users (and more data) can mean increasingly complex queries, especially if queries depend on the number of relationships among users. (The number of relationships is bounded by (N * (N–1)) / 2, where N is the number of users.)

User activity

Not all user activity is equal, and user activity is not constant. If your users suddenly become more active, for example because of a new feature they like, your load can increase significantly. User activity isn’t just a matter of the number of page views, either—the same number of page views can cause more work if part of the site that requires a lot of work to generate becomes more popular. Some users are much more active than others, too: they might have many more friends, messages, or photos than the average user.

Size of related datasets

If there are relationships among users, the application might need to run queries and computations on entire groups of related users. This is more complex than just working with individual users and their data. Social networking sites often face challenges due to popular groups or users who have many friends.

Read Versus Write Bound Loads

One of the first things you should examine when thinking about scaling your database architecture is whether you are scaling a read bound load or a write bound load.

If we are to assume that at the beginning of designing your product you took the shortcut of using the one source host for all database traffic, it is clear at this point that adding more application resources may scale the client's serving requests but will ultimately be capped by the ability of your one source database host to respond to these read requests. This is a read bound load, and this is where scaling read traffic using replicas comes in. We will discuss later in this chapter how to scale your reads using read replica pools, how to healthcheck these pools, and what pitfalls to avoid when you start using that architecture.

You may also be encountering a write bound load. Perhaps signups are growing exponentially; or it is peak ecommerce season and sales are growing, along with the number of orders to track; or it is election season and you have a lot of campaign communication going out—all of these are business use cases that lead to exponentially more database writes that you now have to scale. Again, a single source database, even if you can scale it up for some time, can only go so far. When the bottleneck is the write volume, you have to start thinking about ways to split your data so you can accept writes in parallel on separate subsets. We will talk about how to shard for write scaling later in this chapter as well.

A logical question at this point is “what if I am seeing both types of growth?”. It is important to closely inspect your schema and identify whether there is a subset of tables growing faster in reads vs another subset growing in write needs. Trying to scale a database cluster for both at the same time is asking for a lot of pain and incidents. We recommend separating tables in different functional clusters to independently scale reads and writes; this is a prerequisite for making scaling read traffic with read pools far more impactful.

Now that you have determined whether you have a read or write bound load, let’s talk about how to scale for read bound loads using replica read pools.

Scaling Reads with Read Pools

Replicas in a cluster serve more than one purpose. First and foremost, they are candidates for failing over writes either in a planned or unplanned manner when the current source needs to be taken out of service for any reason. But since these replicas are also constantly running updates to match the data in the source, you can use them to serve read requests as well.

In Fig 11-1, we start by getting a visual of what this new setup with read replica pools looks like.

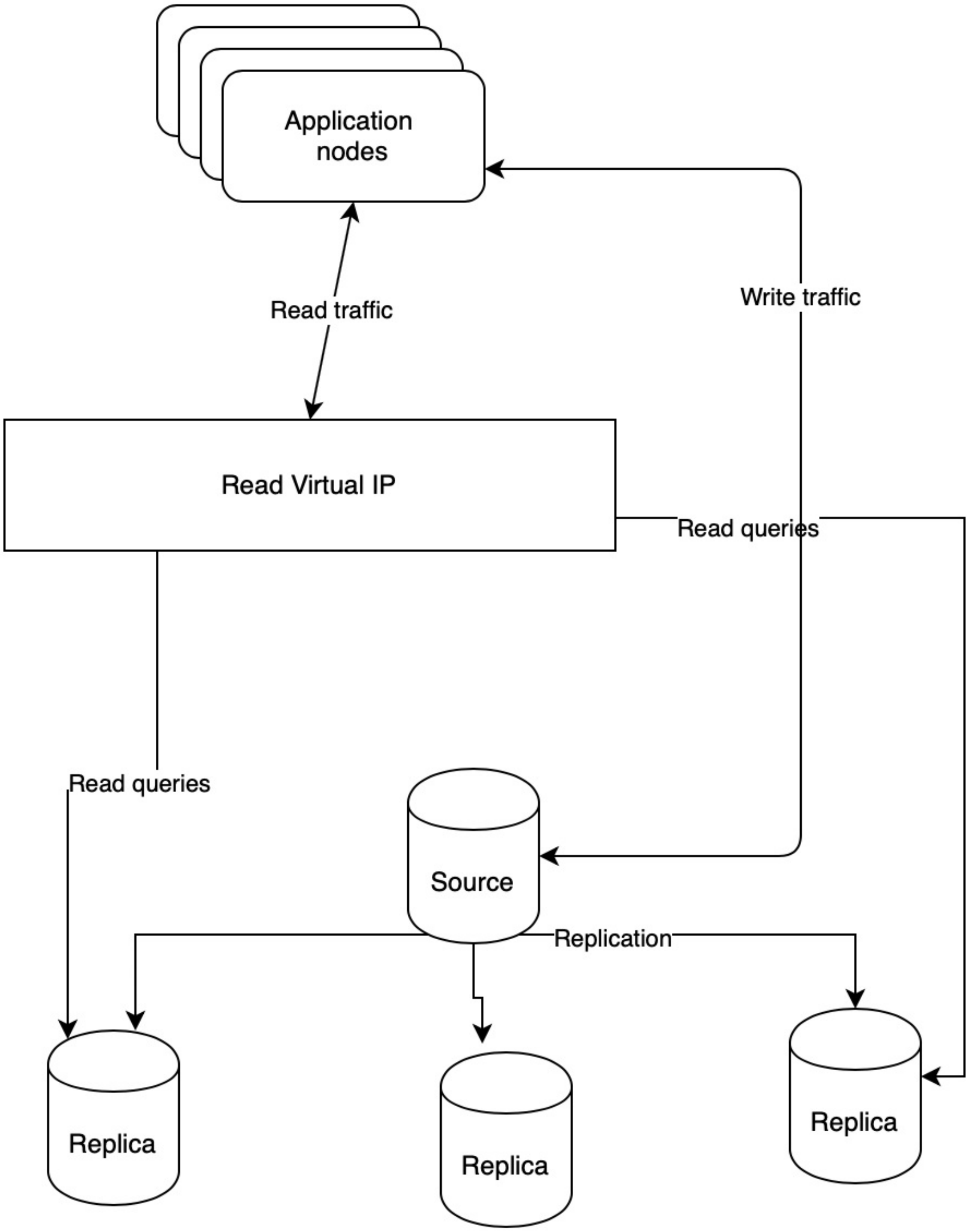

Figure 2-1. Application nodes using a virtual IP to access read replicas

For the sake of simplicity, we will pretend application nodes still fulfill write requests by directly connecting to the source database. We will later talk about how that specific part scales better. Note though that the same application nodes connect to a virtual IP which acts as a middle layer between them and the read replicas. This is a replica read pool, and this is how you spread the growing read load to more than one host. You may also note that not all replicas are in the pool. That is a common way to prevent different read workloads from affecting each other. If you have reporting processes, or your backup process tends to consume all the disk I/O resources, it is prudent to leave out one or more replica nodes to fulfill those tasks and exclude it from the read pool that serves customer facing traffic.

The flexibility of turning your read replicas into interchangeable resources grows significantly when there is a single point the application talks to for reads and you can manage these resources seamlessly without impact to your customers.

Now that the database hosts serving read requests are more than one, there are a few things to consider for smooth production sailing:

- How do you route traffic to all these read replicas?
- How do you evenly distribute the load?
- How do you run health checks and remove unhealthy or lagged replicas to avoid serving stale data?
- How do you avoid accidentally removing all of the nodes, causing more damage to the application traffic?

A very common way of managing these read pools is using a load balancer to run a virtual IP that acts as a middle man for all traffic meant to go to the read replicas. Technologies for this include HAProxy, Hardware Load Balancer if you self host, or a Network Load Balancer if you are running in a public cloud environment. In the case of using HAProxy, all application hosts will connect to that one ‘frontend’ and HAProxy takes care of directing those requests to one of the read replicas defined in the backend. Tooling that facilitates sharding, such as Vitess and ProxySQL, can also act in a load balancer fashion. We’ll cover these tools towards the end of the chapter.

Managing Configuration for Read Pools

Now that you have a ‘gate’ between the application nodes and your replicas, you need a way to easily manage the nodes included, or not included, in this read pool using your load balancer of choice. You do not want this to be a manually managed configuration. You are already on a trajectory of scaling to lots of database instances, and managing configuration files manually will lead to mistakes, slower response time to host failures and simply does not scale.

Service discovery is a good type of service to use here for automatically discovering what hosts can be in this list. This may mean deploying a service discovery solution as part of your tech stack, or relying on a managed service discovery option at your cloud provider if that is available. The important thing to be careful with here is to be very specific on the criteria that make a read replica qualify for this read pool. Ideally, you exclude the source node, and potentially one or more replicas dedicated for reporting. But maybe you need something even more complex where the replicas are further segmented to serve different application read loads? We recommend at minimum 3 nodes per pool of replicas serving a specific purpose in addition to your backups/reporting server and the source node.

Whether you run your own service discovery or use something offered in your cloud, you should be aware of the guarantees of that service. How soon can it detect the failure of a host? How fast does that data propagate? When there is a failure how will the configuration refresh on your load balancer, and does it do it as a background process or will it require severing existing connections?

With flexibility comes complexity, and you must balance the two for optimal outcomes in production when failures happen. Your job here is to always tether your decisions to what SLIs and SLOs are being pursued and not to achieve a mythical 100% uptime goal.

Now that you know how to populate the configurations and update them as hosts come and go, it's time to talk about how to run health checks for the members of a replica read pool.

Health Checks for Read Pools

At this point you will need to consider what are the acceptable criteria that deem a read replica healthy and ready to accept read traffic from the application. These criteria can be as simple as “the DB process is up and running, the port responds” and can become more complex such as “the database is up, and replication lag needs to be no more than 30 seconds, and read queries need to be running at a latency no higher than 100 ms”. Deciding how far to take these health checks should be a conversation with your application developer teams so that everyone understands and aligns on what behavior they expect when reading from the database. Here are some questions to ask the team that can help guide this decision process:

- How much data staleness is acceptable? If the data returned is a few minutes old, what does that affect?
- What is the maximum acceptable query latency for the application?
- What, if any, retry logic exists for read queries, and if it exists is it exponential backoff?
- Do we already have an SLO for the application? Does that SLO extend to query latency or only address uptime?
- How does the system behave in the absence of this data? Is that degradation acceptable? if so for how long?

In many cases, you will be fine only using a port check for proving the MySQL process liveness to deem it healthy. This means as long as the database is running it will be part of that pool and serving requests.

However, sometimes you may need something more sophisticated because the dataset involved is critical enough that you do not want to serve it when replication lags more than a few seconds or if replication is not running at all. For these scenarios you can still use read pool, but augment the health check with an HTTP check. The way this works is that your load balancer of choice will run a command (usually a script) and, based on the response code, will determine if the node is healthy or not. In HAProxy, for example, the backend would have lines of code like this:

```
option httpchk GET /check-lag
```

This line means that for every host in the read pool, the load balancer will call the path /check-lag using a GET call and inspect the response code. That path runs a script that holds the logic as to how much lag is acceptable. The script compares existing lag status with that threshold and, depending on that, the load balancer either considers the replica healthy or not.

WARNING

While they are a powerful tool, be careful with using health checks with complex logic in them (such as the lag check described above)and make sure you have a plan for “what to do if all replicas in the pool fail the health checks”. You can have a static ‘fallback’ pool that brings all the nodes back in for certain global failures (e.g., the entire cluster is lagged) to avoid accidentally breaking all read requests. For more detail on how one company has implemented this, you can look at this post by Github[2].

Choosing a Load Balancing Algorithm

There are many different algorithms to determine which server should receive the next connection. Each vendor uses different terminology, but this list should provide an idea of what's available:

Random

The load balancer directs each request to a server selected at random from the pool of available servers.

Round-robin

The load balancer sends requests to servers in a repeating sequence: A, B, C, A, B, C, etc.

Fewest connections

The next connection goes to the server with the fewest active connections.

Fastest response

The server that has been handling requests the fastest receives the next connection. This can work well when the pool contains a mix of fast and slow machines. However, it's very tricky with SQL when the query complexity varies widely. Even the same query can perform very differently under different circumstances, such as when it's served from the query cache or when the server's caches already contain the needed data.

Hashed

> The load balancer hashes the connection's source IP address, which maps it to one of the servers in the pool. Each time a connection request comes from the same IP address, the load balancer sends it to the same server. The bindings change only when the number of machines in the pool does.

Weighted

> The load balancer can combine and add weight to several of the other algorithms. For example, you might have single- and dual-CPU machines. The dual-CPU machines are roughly twice as powerful, so you can tell the load balancer to send them an average of twice as many requests.

The best algorithm for MySQL depends on your workload. The least-connections algorithm, for example, might flood new servers when you add them to the pool of available servers—before when their caches are warmed up.

You'll need to experiment to find the best performance for your workload. Be sure to consider what happens in extraordinary circumstances as well as in the day-to-day norm. It is in those extraordinary circumstances—e.g., during times of high load, when you're doing schema changes, or when an unusual number of servers go offline—that you can least afford something going terribly wrong.

We’ve described only instant-provisioning algorithms here, which don’t queue connection requests. Sometimes algorithms that use queuing can be more efficient. For example, an algorithm might maintain a given concurrency on the database server, such as allowing no more than N active transactions at the same time. If there are too many active transactions, the algorithm can put a new request in a queue and serve it from the first server that becomes “available” according to the criteria. Some connection pools support queuing algorithms.

Now that we have covered how to scale your read load and how to health check it, it’s time to discuss scaling writes. Before looking for how to scale the writes directly, you can look at places where queueing can make the write traffic growth more manageable. Let’s discuss how queueing can help scale your write performance.

Queuing

Scaling your application layer becomes a lot more complex when scaling write transaction with a datastore that favors consistency over availability by design. More application nodes writing to the one source node will lead to a database system more susceptible to lock timeouts, deadlocks, and failed writes to have to retry. All this will ultimately lead to customer facing errors or unacceptable latencies.

Before looking into sharding the data, which we discuss next, you should examine the write hotspots in your data and consider whether all the writes are truly required to persist to the database actively. Can some of them be placed into a queue and written to the database within an acceptable time frame?

Let’s say you have a database that stores large datasets of customer historical data. Customers occasionally send API (Application Programming Interface) requests to retrieve this data but you also need to support an API to delete this data. You can plausibly serve read API calls from a growing number of replicas, but what about deletes? The HTTP RFC allows for a response code 202 Accepted. You can return that, place the request in a queue (e.g., Kafka, SQS) and process these requests at the pace that doesn’t lead to overloading the database directly with delete calls.

This is obviously not the same as a 200 response code which implies the request has been instantaneously fulfilled. This is a common spot where negotiation with your product team is crucial for making the guarantees of the API plausible and achievable. The difference between the 200 and 202 response codes is the difference of all the engineering work that would work in sharding this data to support a lot more parallel writes.

One important design choice to make if you do apply queuing to a write load is to determine up front the desired time frame within which these calls are expected to be fulfilled after being placed in queue. Monitoring the growth of the time a request spends in a queue is going to be your metric for when this strategy has run its course and you really need to start splitting this data set to support more parallel write load. You can do that using sharding, which we discuss next.

Sharding

If you cannot manage write traffic growth with optimized queries and queueing writes, then sharding is your next option.

Sharding means splitting your data into different, smaller, database clusters so you can execute more writes on more source hosts at the same time. There are 2 different kinds of sharding or partitioning you can do-functional partitioning and data sharding.

Functional partitioning, or division of duties, means dedicating different nodes to different tasks. An example of this might be putting user records on one cluster and their billing on a different cluster. This approach allows each cluster to scale independently. A surge in user registrations might put a strain on the user cluster. With seperate systems your billing cluster is less loaded, allowing you to bill customers. Conversely, if your billing cycle is the first of the month, you can run that knowing you won't be impacting user registration.

Data sharding is the most common and successful approach for scaling today's very large MySQL applications. You shard the data by splitting it into smaller pieces, or shards, and storing them on different nodes.

Most applications shard only the data that needs sharding—typically, the parts of the dataset that will grow very large. Suppose you’re building a blogging service. If you expect 10 million users, you might not need to shard the user registration information because you might be able to fit all of the users (or the active subset of them) entirely in memory. If you expect 500 million users, on the other hand, you should probably shard this data. The user-generated content, such as posts and comments, will almost certainly require sharding in either case, because these records are much larger and there are many more of them.

Large applications might have several logical datasets that you can shard differently. You can store them on different sets of servers, but you don’t have to. You can also shard the same data multiple ways, depending on how you access it.

Choosing a Partitioning Scheme

The most important challenge with sharding is finding and retrieving data. How you find data depends on how you shard it. There are many ways to do this, and some are better than others.

The goal is to make your most important and frequent queries touch as few shards as possible (remember, one of the scalability principles is to avoid crosstalk between nodes). The most important part of that process is choosing a partitioning key (or keys) for your data. The partitioning key determines which rows should go onto each shard. If you know an object’s partitioning key, you can answer two questions:

1. Where should I store this data?
2. Where can I find the data I need to fetch?

We’ll show you a variety of ways to choose and use a partitioning key later. For now, let’s look at an example. Suppose we do as MySQL’s NDB Cluster does, and use a hash of each table’s primary key to partition the data across all the shards. This is a very simple approach, but it doesn’t scale well because it frequently requires you to check all the shards for the data you want. For example, if you want user 3’s blog posts, where can you find them? They are probably scattered evenly across all the shards, because they’re partitioned by the primary key, not by the user. Using a primary key hash makes it simple to know where to store the data, but it might make it harder to fetch it, depending on which data you need and whether you know the primary key.

Cross-shard queries are worse than single-shard queries, but as long as you don’t touch too many shards, they might not be too bad. The worst case is when you have no idea where the desired data is stored, and you need to scan every shard to find it.

A good partitioning key is usually the primary key of a very important entity in the database. These keys determine the unit of sharding. For example, if you partition your data by a user ID or a client ID, the unit of sharding is the user or client.

A good way to start is to diagram your data model with an entity-relationship diagram, or an equivalent tool that shows all the entities and their relationships. Try to lay out the diagram so that the related entities are close together. You can often inspect such a diagram visually and find candidates for partitioning keys that you'd otherwise miss. Don't just look at the diagram, though; consider your application's queries as well. Even if two entities are related in some way, if you seldom or never join on the relationship, you can break the relationship to implement the sharding.

Some data models are easier to shard than others, depending on the degree of connectivity in the entity-relationship graph. Figure 11-2 depicts an easily sharded data model on the left, and one that's difficult to shard on the right.

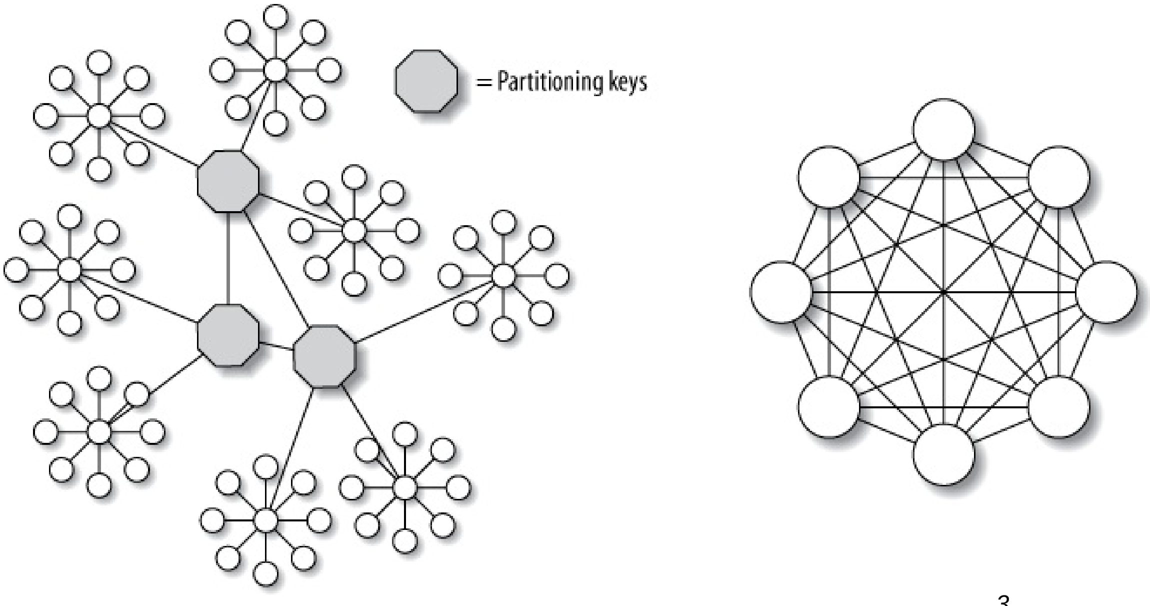

Figure 2-2. Two data models, one easy to shard and the other difficult [3]

The data model on the left is easy to shard because it has many connected subgraphs consisting mostly of nodes with just one connection, and you can “cut” the connections between the subgraphs relatively easily. The model on the right is hard to shard, because there are no such subgraphs. Most data models, luckily, look more like the left hand diagram than the right hand one.

When choosing a partitioning key, try to pick something that lets you avoid cross-shard queries as much as possible, but also makes shards small enough that you won’t have problems with disproportionately large chunks of data. You want the shards to end up uniformly small, if possible, and if not, at least small enough that they’re easy to balance by grouping different numbers of shards together. For example, if your application is US-only and you want to divide your dataset into 20 shards, you probably shouldn’t shard by state, because California has such a huge population. But you could shard by county or telephone area code, because even though these won’t be uniformly populated, there are enough of them that you can still choose 20 sets that will be roughly equally populated in total, and you can choose them with an affinity that helps avoid cross-shard queries.

Multiple Partitioning Keys

Complicated data models make data sharding more difficult. Many applications have more than one partitioning key, especially if there are two or more important “dimensions” in the data. In other words, the application might need to see an efficient, coherent view of the data from different angles. This means you might need to store at least some data twice within the system.

For example, you might need to shard your blogging application’s data by both the user ID and the post ID, because these are two common ways the application looks at the data. Think of it this way: you frequently want to see all posts for a user, and all comments for a post. Sharding by user doesn’t help you find comments for a post, and sharding by post doesn’t help you find posts for a user. If you need both types of queries to touch only a single shard, you’ll have to shard both ways.

Just because you need multiple partitioning keys doesn’t mean you’ll need to design two completely redundant data stores. Let’s look at another example: a social networking book club website, where the site’s users can comment on books. The website can display all comments for a book, as well as all books a user has read and commented on.

You might build one sharded data store for the user data and another for the book data. Comments have both a user ID and a post ID, so they cross the boundaries between shards. Instead of completely duplicating comments, you can store the comments with the user data. Then you can store just a comment’s headline and ID with the book data. This might be enough to render most views of a book’s comments without accessing both data stores, and if you need to display the complete comment text, you can retrieve it from the user data store.

Querying Across Shards

Most sharded applications have at least some queries that need to aggregate or join data from multiple shards. For example, if the book club site shows the most popular or active users, it must by definition access every shard. Making such queries work well is the most difficult part of implementing data sharding, because what the application sees as a single query needs to be split up and executed in parallel as many queries, one per shard. A good database abstraction layer can help ease the pain, but even then such queries are so much slower and more expensive than in-shard queries that aggressive caching is usually necessary as well.

If you have chosen your sharding scheme well, cross-shard queries should become the outlier not the norm. This means to strive to make the most of your queries as simple as possible and contained within one shard. For those cases where some cross shard aggregation is needed, we recommend you make that part of the application logic.

Cross-shard queries can also benefit from summary tables. You can build them by traversing all the shards and storing the results redundantly on each shard when they’re complete. If duplicating the data on each shard would be too wasteful, you can consolidate the summary tables onto another data store, so they’re stored only once.

Non-sharded data often lives in the global node, with heavy caching to shield it from the load.

Some applications use essentially random sharding when perfectly even data distribution is important, or when there is no good partitioning key. A distributed search application is a good example. In this case, cross-shard queries and aggregation are the norm, not the exception.

Querying across shards isn't the only thing that's harder with sharding. Maintaining data consistency is also difficult. Foreign keys won't work across shards, so the normal solution is to check referential integrity as needed in the application, or use foreign keys within a shard, because internal consistency within a shard might be the most important thing. It's possible to use XA transactions[4], but this is uncommon in practice because of the overhead.

You can also design cleanup processes that run intermittently. For example, if a user's book club account expires, you don't have to remove it immediately. You can write a periodic job to remove the user's comments from the per-book shard, and you can build a checker script that runs periodically and makes sure the data is consistent across the shards.

Now that we have explained the different ways you can split your data across multiple clusters and how to choose a partitioning key, let's cover 2 of the most popular open source tools that can help facilitate both sharding and partitioning.

Vitess

Vitess is a database clustering system for MySQL. It originated within YouTube then became PlanetScale, a separate product and company headed by Jiten Vaidya and Sugu Sougoumarane.

It enables a number of features such as:

- Horizontal sharding support. Including sharding the data
- Topology management
- Managing source node failover
- Schema change management
- Connection pooling
- Query rewriting

Let's explore Vitess' architecture and its components.

VITESS ARCHITECTURE OVERVIEW

This diagram from Vitess' website shows the different parts of its architecture.

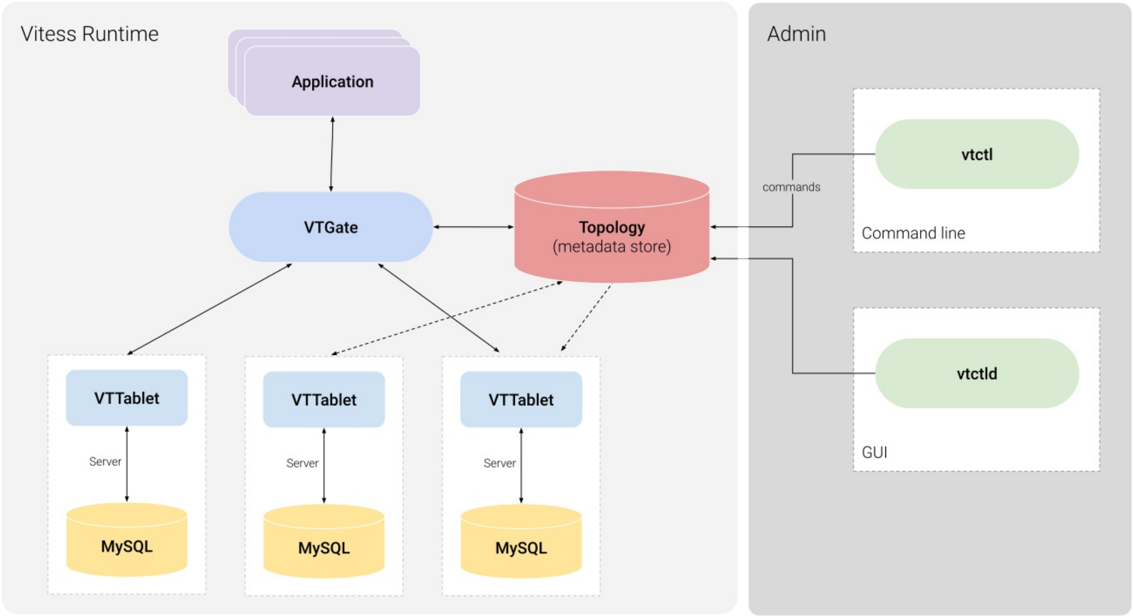

Figure 2-3. Vitess architecture diagram. Credit: vitess.io

Here are some terms you need to know:

- Vitess Pod is the general encapsulation of a set of databases and the Vitess related pieces that support sharding, managing topology, managing schema changes and application access to said databases
- VTgate is the service that controls access to the database instances for applications and operators trying to manage topology, add nodes or shard some of the data. It is akin to the load balancer in the architecture described previously.
- VTTablet is the agent running on each database instance managed by Vitess. It can receive DB management commands from operators and execute them on the operators’ behalf
- Topology Metadata store holds the inventory of database instances managed by Vitess in a given pod, and also holds accompanying information

- vtctl is the command line tool to make operational changes to a Vitess pod
- vtctld is a graphical interface for the same management operations

Vitess’s architecture starts with a consistent topology store that holds definitions for all the clusters/MySQL instances and vtgate instances. This consistent metadata store plays a crucial role in managing topology changes. When an operator is looking to make a change to the topology of a cluster managed by Vitess, it is really sending commands, through a service called vtctl, to that datastore, which then sends the component operations of that command to vtgate.

Vitess offers database operators that can deploy the vtgate layer and the metadata store in kubernetes. Having its control plane in a platform like Kubernetes increases its resilience to single points of failure.

One of Vitess’s greatest strengths is its philosophy towards how to scale MySQL, which includes the following.

A preference for using smaller instances. Split your data functionally, horizontally or both. But smaller instances make for a smaller blast radius when failures happen.

Replication and automated write failover increase resilience. Vitess does not promise ‘100% online writes’ through multi writer node tricks. Instead, it automates write failover, and during that failover manages both the topology change and application access to the database nodes to make the write downtime as short as possible.

Durability using semi-sync replication. Vitess strongly recommends semi-sync replication (as opposed to the default asynchronous) to ensure writes are always persisted by more than one node in the database layer before acknowledging them to the application. This is a crucial tradeoff in latency for the sake of guaranteed durability that pays its dividends when Vitess needs to failover the writer host in an unplanned manner.

These architectural principles can help sustain exponential growth in your business traffic with a lot more resilience in the database layer of your infrastructure. And many of these best practices listed above you should heed regardless of whether you specifically use Vitess or another solution as part of your architecture.

MIGRATING YOUR STACK TO VITESS

Vitess is an opinionated platform for running the database layer and is not a drop-in solution. Therefore, you need to thoughtfully plan how implementing such a transition would happen before you adopt it as the access layer for your database.

Specifically, be sure to consider the following migration steps as you evaluate Vitess as a possible solution:

Test and document the latency you’re introducing to the overall system. Introducing a complex stack like Vitess to an application stack will definitely add some amount of latency, especially when you consider the enforcement of semi-sync replication. Make sure this tradeoff is well documented and explicitly communicated so your downstream dependencies are making informed decisions when building service level objectives that rely on this database architecture.

Use the canary deployment model.[5] During the transition in production, you can configure vttablet as ‘externally managed’. This allows for both vttablet and direct connections to the database server as you slowly ramp up the connection change through your application node fleet.

Start sharding. Once all the application layer access is through vtgate/vttablet and not directly to MySQL, you can start using the full feature set of Vitess to split tables off in new clusters, shard data horizontally for more write throughput, or simply add replicas for more read load capacity[6].

Vitess is a powerful database access and management product that has come a long way from its early days within YouTube at Google. It has proven its ability to enable dramatic growth and a resilient database infrastructure. However, this power and flexibility comes at a cost of added complexity. Vitess is not as simple as a load balancer passing through traffic and you should weigh the needs of the business with the cost to introduce and maintain a database management tool as complex as Vitess.

ProxySQL

ProxySQL is written specifically for the MySQL protocol and released with a GPL license. René Cannaò, a DBA who has consulted for many companies and long time MySQL contributor, is the primary author. It is now a full fledged company that offers paid support and development contracts of the ProxySQL product.

Let’s dig into some details about its architecture, configuration patterns, use cases and features.

PROXYSQL ARCHITECTURE OVERVIEW

You can use ProxySQL as a layer in between any application code and MySQL instances. ProxySQL provides a session aware, MySQL-protocol-based interface for applications to interact with the databases. Instead of applications opening connections directly to the database instances, ProxySQL opens them on the application’s behalf.

This design makes the proxy seem invisible to the application nodes. Its session awareness allows for moving these connections between mysql instances without downtime. This is especially useful when you are dealing with applications no longer being invested in because you can now utilize features in ProxySQL without needing to make any changes to code you may not feel confident changing.

ProxySQL also provides powerful connection pooling.

Connections opened by applications to ProxySQL are separate from the connections ProxySQL opens to database instances it is configured to connect to. This separation allows for protecting the database instances from sudden traffic spikes in the application layer from impacting the databases.

When you have the ability to manage client side connections separately from how many connections actually are made to the database, you introduce flexibility you did not have before. You can now scale out the application node pool without having to worry that it will increase connection load to the database beyond what you want to support. This allows for diverse scenarios of application and business needs as we will explain in the common patterns when using ProxySQL.

CONFIGURING PROXYSQL

ProxySQL uses a configuration file for startup but maintains its runtime configuration both in memory and in an embedded SQLite file that you can access directly and query using an admin interface.

ProxySQL's admin interface allows you to issue commands to change the running configuration then dump that new configuration out to disk for persistence using MySQL commands. This allows you to make zero downtime changes to a running ProxySQL instance. You can also use this admin interface to make automated changes issued by your configuration management or automated failover scripts. You can see in Figure 11-4 the general idea of how your architecture would leverage both ProxySQL and service discovery to provide a robust access layer for services.

WARNING

It is important to note that while we show ‘ProxySQL’ as one object in this diagram, we strongly recommend in production environments to leverage its clustering mechanism and deploy multiple instances in a given stack. Never run a Single Point of Failure (SPoF).

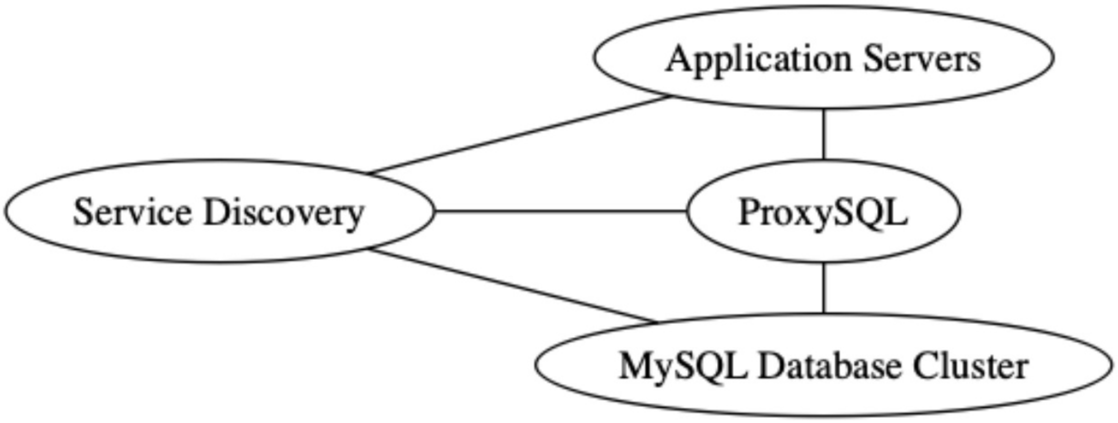

Figure 2-4. Figure showing the interaction between application nodes, ProxySQL and service discovery’s role. Credit: Bill Sickles

ProxySQL has independent and hierarchical health checking for databases it connects to. Based on the results of these health checks, ProxySQL adds/removes hosts or adjusts traffic weights. You can specify replication lag thresholds, time to connect successfully, and connection retries on failure among many other configuration options to control how much fault tolerance is acceptable within the context of your service and application needs. These configuration options allow ProxySQL to react accurately to unresponsive hosts by either temporarily removing backend databases then repeating the health check later, or by fully removing the struggling backend member until an operator is involved.

COMMON USE CASES FOR PROXYSQL

Let's discuss some common patterns where using ProxySQL can help alleviate common issues in fast growing environments.

CONNECTION SHIELD FROM APPLICATIONS

In many applications, 'open more connections to the database' is a common pattern we see when query latency starts to climb. However, in practice this can lead to outages[7] and tends to leave a lot of connections idle, consuming resources but not doing any work. When you open more connections by the application layer directly to the database, the amount of resources the database server spends on connection management also increases. This snowballs into thousands of connections overwhelming already overloaded database instances. All this activity leads to prolonged downtimes, failures cascading in multiple microservices, and extended customer facing impact.

ProxySQL's connection management architecture helps shield the database layer from unexpected application peaks by only opening to the database the number of connections that can do work. ProxySQL can reuse those connections for different client side requests. This behavior maximizes the work that a single connection to the database servers can do. This in turn reduces the amount of resources managing connections and allows for more efficient use of the database server memory resources.

OTHER NOTABLE FEATURES IN PROXYSQL

ProxySQL has a number of other features that are notable and that stand out in a general use application proxy. These include:

- Query routing based on port, or user or simply a regex match
- TLS support on both the frontend application connections and backend connections to databases
- Support for various MySQL flavors such as AWS Aurora, Galera cluster or Clickhouse
- Connection Mirroring
- Result set caching
- Query rewrites
- Audit log

You can read about the extensive feature set of ProxySQL (which goes well beyond sharding support) by visiting its documentation.[8]

ProxySQL is a powerful tool you can use for scaling out your application with proper performance protections for the database layer and with added features that support all sorts of business needs (like compliance, security rules, etc). If you are in a company that is finding itself on a high growth trajectory with a robust mix of new and less new services sharing database resources, it can be a powerful tool in continuing that growth safely. ProxySQL provides easy to deploy abstraction that can be more sophisticated than HAPrxoy but with less upfront investment in infrastructure and complexity. However it also does not offer some of the more advanced features such as automated sharding of data sets, managing schema changes and VReplication[9] which is a powerful tool for enabling ETL pipelines and compliance needs as we will discuss in chapter 14.

Summary

Scaling MySQL is a journey. You should come out of this chapter more prepared to assess your scaling needs, and understanding how to scale reads, how to scale writes and how to make your traffic growth more predictable by adding queuing to your architecture. You should also now understand sharding to scale writes and all the complex decisions that come with that decision.

Running MySQL at the scale where you have hundreds of instances complicates a lot of common tasks. Choosing how to grow with the demand is only the beginning, which is why our next chapter is about how to run MySQL at the scale of hundreds to thousands of instances.

[1] In the physical sciences, work per unit of time is called power, but in computing “power” is such an overloaded term that it’s ambiguous and we avoid it. However, a precise definition of capacity is the system’s maximum power output.

[2] https://github.blog/2016-08-17-context-aware-mysql-pools-via-haproxy/

[3] Thanks to the HiveDB project and Britt Crawford for contributing these elegant diagrams.

[4] https://dev.mysql.com/doc/refman/8.0/en/xa.html

[5] https://www.cncf.io/wp-content/uploads/2020/03/Migrating-MySQL-to-Vitess-CNCF-Webinar.pdf

[6] This deployment strategy is explained in detail by Morgan Tocker in a talk at Kubecon 2019 at https://www.youtube.com/watch?v=OCS45iy5v1M

[7] See https://en.wikipedia.org/wiki/Thundering_herd_problem

[8] https://proxysql.com/documentation/

[9] https://vitess.io/docs/reference/vreplication/

About the Authors

Silvia Botros is a senior principal engineer at Twilio. During her time at SendGrid, she helped deploy and maintain various MySQL datastores that support the mail pipeline and other products that SendGrid offers and to drive MySQL designs from inception to production.

Jeremy Tinley is a senior staff engineer at Etsy, with over 20 years of MySQL experience. Throughout his career, he has managed tens of thousands of MySQL instances with an eye towards availability, reliability, and operational efficiency.