Common Misconceptions About Database Setups

What Sounds Easy - But Usually Isn't

Daniël van Eeden

PingCAP

pre-FOSDEM MySQL Belgian Days, February 2026

Agenda

- Going over multiple database setups.
- Offering some advice for some of these.
- Get the discussion started.

Databases Are Hard

- Modern systems can be complex. Simple single server databases often aren't good enough.
- Database architectures are full of trade-offs
- Many popular ideas can be oversimplified or misleading

Misconception #1

“Multi Primary setups bring more capacity”

Misconception #1

“Multi Primary setups bring more capacity”

We are mostly talking about write capacity.

Misconception #1

“Multi Primary setups bring more capacity”

We are mostly talking about write capacity.

Multi-Primary setups:

- MySQL with circular replication (remember MMM?)

Misconception #1

“Multi Primary setups bring more capacity”

We are mostly talking about write capacity.

Multi-Primary setups:

- MySQL with circular replication (remember MMM?)
- Galera

Misconception #1

“Multi Primary setups bring more capacity”

We are mostly talking about write capacity.

Multi-Primary setups:

- MySQL with circular replication (remember MMM?)
- Galera
- Group Replication (part of InnoDB Cluster)

Misconception #1

“Multi Primary setups bring more capacity”

Issues:

- Writes must still be coordinated. (depends on solution)

Misconception #1

“Multi Primary setups bring more capacity”

Issues:

- Writes must still be coordinated. (depends on solution)
- Conflict resolution adds complexity

Misconception #1

“Multi Primary setups bring more capacity”

Issues:

- Writes must still be coordinated. (depends on solution)
- Conflict resolution adds complexity
- These implementations often add restrictions or limitations.

Misconception #1

“Multi Primary setups bring more capacity”

Issues:

- Writes must still be coordinated. (depends on solution)
- Conflict resolution adds complexity
- These implementations often add restrictions or limitations.

Why this isn't true: With 3 servers every server hold a full copy and needs to process roughly the same amount of writes.

Misconception #1

“Multi Primary setups bring more capacity”

Issues:

- Writes must still be coordinated. (depends on solution)
- Conflict resolution adds complexity
- These implementations often add restrictions or limitations.

Why this isn't true: With 3 servers every server hold a full copy and needs to process roughly the same amount of writes.

These might increase the number of connections you can hold open. But if that's a concern you probably should use a connection pool, proxy or *MySQL Enterprise Thread Pool*.

Misconception #1

“Multi Primary setups bring more capacity”

Issues:

- Writes must still be coordinated. (depends on solution)
- Conflict resolution adds complexity
- These implementations often add restrictions or limitations.

Why this isn't true: With 3 servers every server hold a full copy and needs to process roughly the same amount of writes.

These might increase the number of connections you can hold open. But if that's a concern you probably should use a connection pool, proxy or *MySQL Enterprise Thread Pool*.

Note that this doesn't apply to distributed systems like MySQL Cluster (NDB) where none of the machines hold a full copy of the dataset.

Misconception #1

“Multi Primary setups bring more capacity”
Issues:

- Writes must still be coordinated. (depends on solution)
- Conflict resolution adds complexity
- These implementations often add restrictions or limitations.

Why this isn't true: With 3 servers every server hold a full copy and needs to process roughly the same amount of writes.
These might increase the number of connections you can hold open. But if that's a concern you probably should use a connection pool, proxy or *MySQL Enterprise Thread Pool*.
Note that this doesn't apply to distributed systems like MySQL Cluster (NDB) where none of the machines hold a full copy of the dataset.
These might still be solutions to solving **HA**, but they do not add write capacity.

Misconception #1

“Multi Primary setups bring more capacity”
Or might it be true at last?

Misconception #1

“Multi Primary setups bring more capacity”
Or might it be true at last?

Doing complex transaction with many reads on multiple hosts and only replicating the results (with row based replication).

Misconception #1

“Multi Primary setups bring more capacity”
Or might it be true at last?

Doing complex transaction with many reads on multiple hosts and only replicating the results (with row based replication).

And it does add read capacity.

Misconception #2

“Read/Write splitting is easy”

Want to scale out by using read replicas? Just enable some simple read/write splitting in a connector or proxy. How hard can it be?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?
- What about Prepared statements?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?
- What about Prepared statements?
- What about `SELECT GET_LOCK('abc',10)`?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?
- What about Prepared statements?
- What about `SELECT GET_LOCK('abc',10)`?
- What about `SELECT ... LOCK IN SHARE MODE`?

Misconception #2

- Simple regex on the query string?
  `^SELECT` goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?
- What about Prepared statements?
- What about `SELECT GET_LOCK('abc',10)`?
- What about `SELECT ... LOCK IN SHARE MODE`?
- What about `CALL`?

Misconception #2

- Simple regex on the query string?
  ^SELECT goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about WITH... or TABLE... statements?
- What about replication delay?
- What about START TRANSACTION? Or SET autocommit=0?
- What about SET @a := 123?
- What about CREATE TEMPORARY TABLE?
- What about SELECT VERSION()?
- What about Prepared statements?
- What about SELECT GET_LOCK('abc',10)?
- What about SELECT ... LOCK IN SHARE MODE?
- What about CALL?
- What about SELECT LAST_INSERT_ID()?

Misconception #2

- Simple regex on the query string?
  ^SELECT goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?
- What about Prepared statements?
- What about `SELECT GET_LOCK('abc',10)`?
- What about `SELECT ... LOCK IN SHARE MODE`?
- What about `CALL`?
- What about `SELECT LAST_INSERT_ID()`?
- What about `SELECT ...; INSERT ...`?

Misconception #2

- Simple regex on the query string?
  ^SELECT goes to a replica
  Everything else (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP) goes to the primary.
  What about `WITH...` or `TABLE...` statements?
- What about replication delay?
- What about `START TRANSACTION`? Or `SET autocommit=0`?
- What about `SET @a := 123`?
- What about `CREATE TEMPORARY TABLE`?
- What about `SELECT VERSION()`?
- What about Prepared statements?
- What about `SELECT GET_LOCK('abc',10)`?
- What about `SELECT ... LOCK IN SHARE MODE`?
- What about `CALL`?
- What about `SELECT LAST_INSERT_ID()`?
- What about `SELECT ...; INSERT ...`?

Misconception #2

“Read/Write splitting is easy”

Sticky replica selection in the loadbalancer can help.

Misconception #2

“Read/Write splitting is easy”
Sticky replica selection in the loadbalancer can help.

Parsing the full statement makes it easier to decide how to route it. But parsing takes time and can be error prone.

Misconception #2

“Read/Write splitting is easy”
Sticky replica selection in the loadbalancer can help.

Parsing the full statement makes it easier to decide how to route it. But parsing takes time and can be error prone. Session Tracking in MySQL can help with

- Transaction state (READ ONLY/READ WRITE/etc)
- User defined variables
- Temporary tables
- Prepared statements

Misconception #2

“Read/Write splitting is easy”
Sticky replica selection in the loadbalancer can help.

Parsing the full statement makes it easier to decide how to route it. But parsing takes time and can be error prone. Session Tracking in MySQL can help with

- Transaction state (READ ONLY/READ WRITE/etc)
- User defined variables
- Temporary tables
- Prepared statements

Best: Have a read and a write database handle in your application.

Misconception #2

“Read/Write splitting is easy”
Sticky replica selection in the loadbalancer can help.

Parsing the full statement makes it easier to decide how to route it. But parsing takes time and can be error prone. Session Tracking in MySQL can help with

- Transaction state (READ ONLY/READ WRITE/etc)
- User defined variables
- Temporary tables
- Prepared statements

Best: Have a read and a write database handle in your application.
Pro tip: Set `read_only=1` on your replica.

Misconception #2

“Read/Write splitting is easy”
Sticky replica selection in the loadbalancer can help.

Parsing the full statement makes it easier to decide how to route it. But parsing takes time and can be error prone. Session Tracking in MySQL can help with

- Transaction state (READ ONLY/READ WRITE/etc)
- User defined variables
- Temporary tables
- Prepared statements

Best: Have a read and a write database handle in your application.
Pro tip: Set `read_only=1` on your replica.
Or use a database that doesn't need read/write splitting.

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?
- Why not use give that memory to your database instead of to your cache?

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?
- Why not use give that memory to your database instead of to your cache?
- Is it a materialized result? E.g. number of views today, then your database could also do that.

Misconception #3

"Caching outside of the database is always good"

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?
- Why not use give that memory to your database instead of to your cache?
- Is it a materialized result? E.g. number of views today, then your database could also do that.
- What about a power outage, or unplanned restart? Lots of cache failures? And then a DB overload?

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?
- Why not use give that memory to your database instead of to your cache?
- Is it a materialized result? E.g. number of views today, then your database could also do that.
- What about a power outage, or unplanned restart? Lots of cache failures? And then a DB overload?
- TTL is either to long (outdated data) or to short (too many cache refreshes). It’s like an avocado...

Misconception #3

“Caching outside of the database is always good”

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?
- Why not use give that memory to your database instead of to your cache?
- Is it a materialized result? E.g. number of views today, then your database could also do that.
- What about a power outage, or unplanned restart? Lots of cache failures? And then a DB overload?
- TTL is either to long (outdated data) or to short (too many cache refreshes). It’s like an avocado...

Change data capture (binlog tailing) can be useful. Even if you need something that is calculated, like page views in the last hour.

Misconception #3

"Caching outside of the database is always good"

- Cache invalidation is one of the hardest problems
- Adds consistency and operational complexity
- What about data security?
- Why not use give that memory to your database instead of to your cache?
- Is it a materialized result? E.g. number of views today, then your database could also do that.
- What about a power outage, or unplanned restart? Lots of cache failures? And then a DB overload?
- TTL is either to long (outdated data) or to short (too many cache refreshes). It's like an avocado...

Change data capture (binlog tailing) can be useful. Even if you need something that is calculated, like page views in the last hour.

If you use caching to avoid overloading a single primary? Then consider using read replicas or a distributed database.

Misconception #4

“Distributed databases can’t guarantee consistency”

- Strong consistency is possible.
- A eventual consistent system often does not provide the consistency that applications expect from a database.

Misconception #4

“Distributed databases can’t guarantee consistency”

- Strong consistency is possible.
- A eventual consistent system often does not provide the consistency that applications expect from a database.
- Fully consistent: The cost is latency for writes.
- Eventual consistent: The cost is latency for reads or inconsistent reads.

Misconception #5

“Splitting one database into many increases availability”

- More services means more failure modes
- Network partitions become critical
- Operational overhead increases significantly

Misconception #5

“Splitting one database into many increases availability”

- More services means more failure modes
- Network partitions become critical
- Operational overhead increases significantly

One database:

- customers and orders: 99.95% availability
- total: 99.95% availability

Misconception #5

“Splitting one database into many increases availability”

- More services means more failure modes
- Network partitions become critical
- Operational overhead increases significantly

One database:

- customers and orders: 99.95% availability
- total: 99.95% availability

Two databases

- customers: 99.95% availability
- orders: 99.95% availability
- total: 99.90% (`.9995 × .9995`)

Misconception #5

“Splitting one database into many increases availability”

- More services means more failure modes
- Network partitions become critical
- Operational overhead increases significantly

One database:

- customers and orders: 99.95% availability
- total: 99.95% availability

Two databases

- customers: 99.95% availability
- orders: 99.95% availability
- total: 99.90% (`.9995 × .9995`)

Calculations are different if you do not need both databases.
Graceful degradation can help.

Misconception #6

“Handling replication delay in the application is easy”

- How do you measure replication delay?

Misconception #6

“Handling replication delay in the application is easy”

- How do you measure replication delay?
- What if there are multiple levels of replication?

Misconception #6

“Handling replication delay in the application is easy”

- How do you measure replication delay?
- What if there are multiple levels of replication?
- Does read-after-write work? pause
- Too much replication delay? Unavailable replica? Retry from the primary

Misconception #6

“Handling replication delay in the application is easy”

- How do you measure replication delay?
- What if there are multiple levels of replication?
- Does read-after-write work? pause
- Too much replication delay? Unavailable replica? Retry from the primary
  And overload your primary
- What about auto-scaling your replicas?

Misconception #6

“Handling replication delay in the application is easy”

- How do you measure replication delay?
- What if there are multiple levels of replication?
- Does read-after-write work? pause
- Too much replication delay? Unavailable replica? Retry from the primary
  And overload your primary
- What about auto-scaling your replicas?
  Now your are scaling up when you should have added an index.

Misconception #7

“Cross-region availability with zero data loss is easy”

- Synchronous replication across regions is expensive
- Latency impacts user experience
- Failover scenarios are hard to test

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.
  - Horizontal: Parititioning, but on different servers.

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.
  - Horizontal: Parititioning, but on different servers.
- Even userid's on the left server, odd ones on the right server.

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.
  - Horizontal: Parititioning, but on different servers.
- Even userid's on the left server, odd ones on the right server.
- Or hash based. `UserID mod 4`.

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.
  - Horizontal: Parititioning, but on different servers.
- Even userid's on the left server, odd ones on the right server.
- Or hash based. `UserID mod 4`.
- Or range based.

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.
  - Horizontal: Parititioning, but on different servers.
- Even userid's on the left server, odd ones on the right server.
- Or hash based. `UserID mod 4`.
- Or range based.
- Or directory based.

Sharding is easy #8

"Sharding is easy"

- Sharding
  - Functional: Split orders and customers.
  - Horizontal: Parititioning, but on different servers.
- Even userid's on the left server, odd ones on the right server.
- Or hash based. `UserID mod 4`.
- Or range based.
- Or directory based.
- Or combining multiple of these methods?

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult.

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work.

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work. - Query multiple shards, combine the results.

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work. - Query multiple shards, combine the results. - Are you sure your results are consistent?

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work. - Query multiple shards, combine the results. - Are you sure your results are consistent?
- Does your application still function when a single shard is down?

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work. - Query multiple shards, combine the results. - Are you sure your results are consistent?
- Does your application still function when a single shard is down? Orders sharded by OrderID, sharding by OrderID mod 8: A single customer might have orders in all shards.

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work. - Query multiple shards, combine the results. - Are you sure your results are consistent?
- Does your application still function when a single shard is down? Orders sharded by OrderID, sharding by OrderID mod 8: A single customer might have orders in all shards.

Avoid rolling your own crypto

Sharding is easy #8

"Sharding is easy"

So why is this hard?

- Resharding is difficult.Or expensive (lots of data to copy).Or both.
- Shard balancing is hard.
  - All servers should hold the same amount of data
  - All servers should have equal query load (read and write
- Cross shard joins can be difficult. Consistency?
- Reporting can be a lot of work. - Query multiple shards, combine the results. - Are you sure your results are consistent?
- Does your application still function when a single shard is down? Orders sharded by OrderID, sharding by OrderID mod 8: A single customer might have orders in all shards.

Avoid rolling your own cryptoor sharding. Use TiDB, MySQL Cluster, Vitess, anything. Contribute back.

Takeaways

- There are no free lunches in database design
- Every architecture involves trade-offs
- Have respect for the complexitiy in database components that is hidden from you.