

Choosing a Database Proxy for MySQL and MariaDB

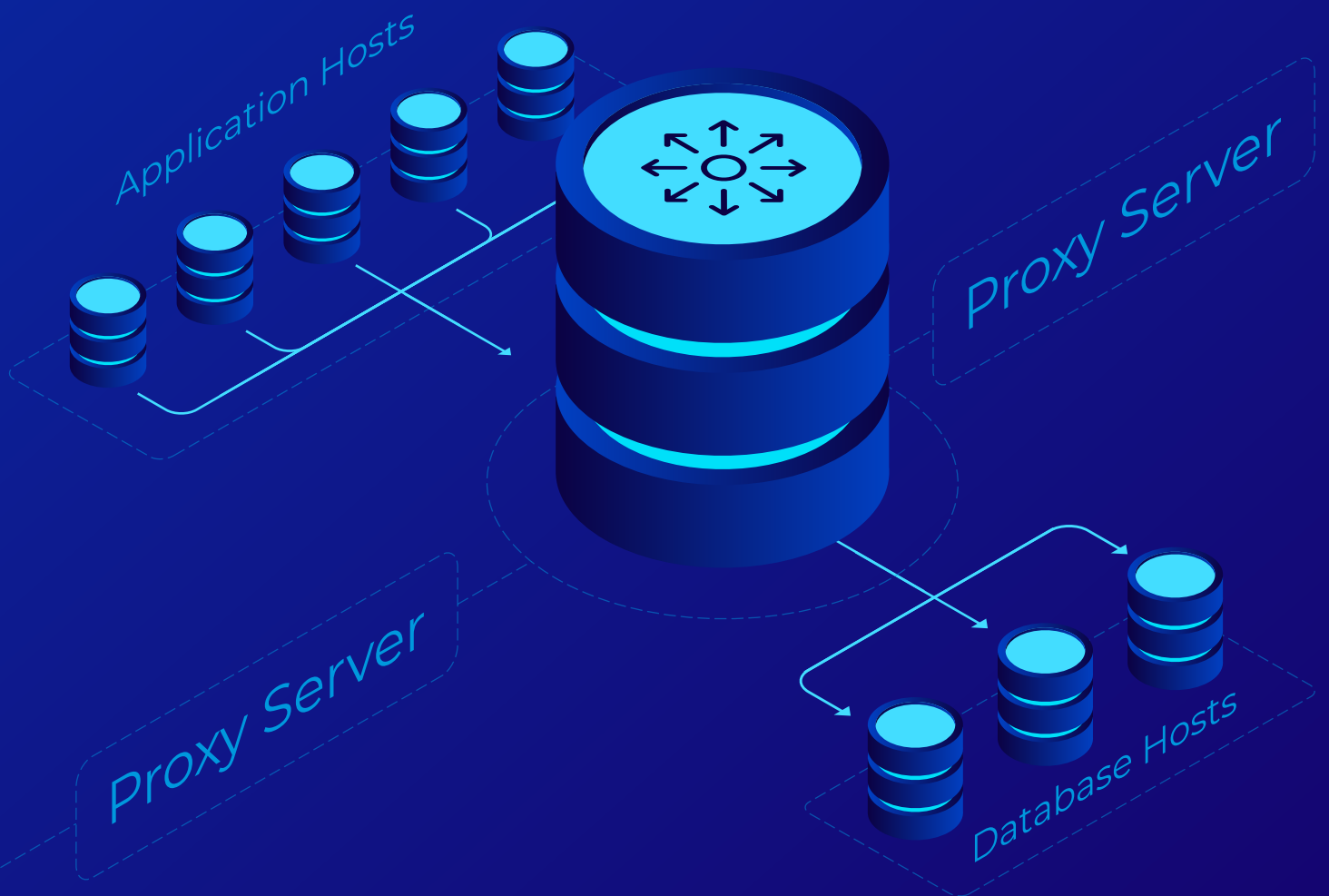


Table of Contents

1. Introduction	3
1.1. What Is a Database Proxy?	3
1.2. Proxy Types	4
1.2.1. Layer 4 Proxies	4
1.2.2. SQL-Aware Proxies	5
2. Features of Advanced, SQL-Aware Proxies	7
2.1. Query Routing	7
2.2. Query Rewriting	8
2.3. Query Caching	9
2.4. Traffic Control	9
2.4.1. Kill a Query	9
2.4.2. Slow Down Traffic	9
3. Designing Highly Available Proxy Architecture	11
3.1. Dedicated Proxy Instances	11
3.1.1. A Proxy with a Virtual IP Assigned	11
3.1.2. Elastic Load Balancing	12
3.1.3. Application-Side Proxy Discovery	13
3.2. Collocating Proxies on Application Hosts	14
3.3. Silo Approach	16
4. Managing Large Proxy Deployments	19
5. Setting up Highly Available Proxy Layers with ClusterControl	21
5.1. ProxySQL Deployment	21
5.2. HAProxy Deployment	23
6. Which Proxy Should I Pick?	25
6.1. SQL-Aware Proxy or Not?	25
6.2. Which SQL-Aware Proxy Should Be Chosen?	25
About ClusterControl	27
About Severalnines	27
Related Resources	28

Introduction

These days, high availability is an imperative. Data is distributed across multiple instances, or even multiple datacenters. Clusters can be scaled out on more nodes. Failures can cause cluster reconfigurations. This begs the question - how does the application know which database node to access? How does an application detect that the database topology has changed? How do we shield the applications from the complexity of the underlying databases?

At some point, the concept of man-in-the-middle became popular and database environments started integrating proxies. This whitepaper will discuss what proxies are, what is their use and how to build a highly available and highly controllable database environment using modern proxies.

1.1. What Is a Database Proxy?

A proxy is a software which handles connectivity between two sides. Within a context of databases, a proxy sits in the middle, between application and database. The application connects to a proxy, which forwards connections to the database. Let's stop here for a second and try to analyze this statement and see what might be the gains of using a proxy? For starters, one, huge gain is that application connects to the proxy only. In the database world, it is not easy to determine where traffic should be directed to. There are writeable or intermediate masters, and read replicas. The replication topology constantly evolves. It is not a good idea to hardcode connectivity patterns. On the other hand, writing code to track topology changes is something that needs to be carefully planned, designed and tested. This is where the proxy comes in. With a use of proxy, applications can connect to it (or to a pool of proxies) and the application may expect that the traffic will be routed to a functioning database.

Since traffic is relayed by the proxy, the latter can be also a great source of information about the traffic itself. It can provide statistics on the traffic, e.g. number of queries executed per second, their execution time, statistical data like 95 percentile, maximum, minimum, average, all based on the collected metrics.

Advanced proxies can also alter the traffic - as everything passes through them, such proxies can provide to an admin a high degree of control over queries - queries can be cached, rewritten, rerouted, stalled or killed. This allows the DBA to shape the traffic and react to the issues immediately, even without requiring an application developer to modify the application and redeploying it.

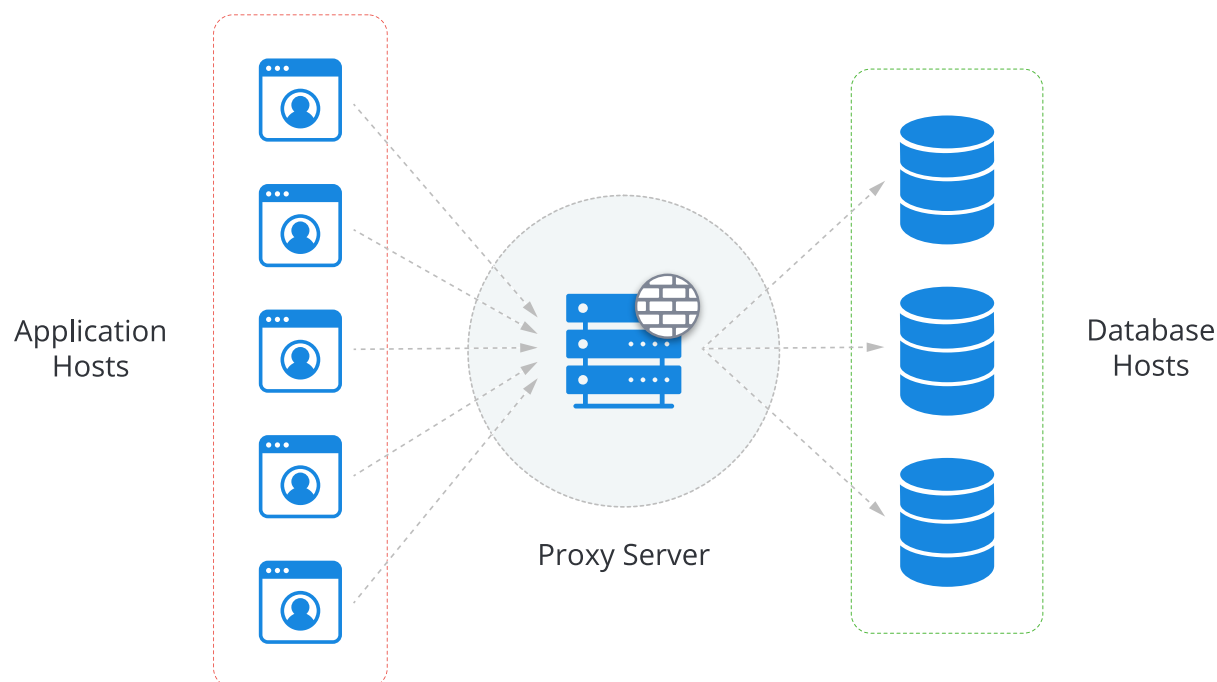
Finally, proxies can help to scale the environment not only through sending traffic to multiple slaves but also they can help to build sharded setups using traffic routing logic created within the proxy. As you can see, an advanced database proxy is not just a packet routing device but it can be utilized in multiple ways, improving the options of the operations team to manage the database tier.

1.2. Proxy Types

Before we dig into details of how proxies can be utilized, in this chapter we will discuss the two main types of proxies. We will look at examples for each type, and cover the main differences between them.

1.2.1. Layer 4 Proxies

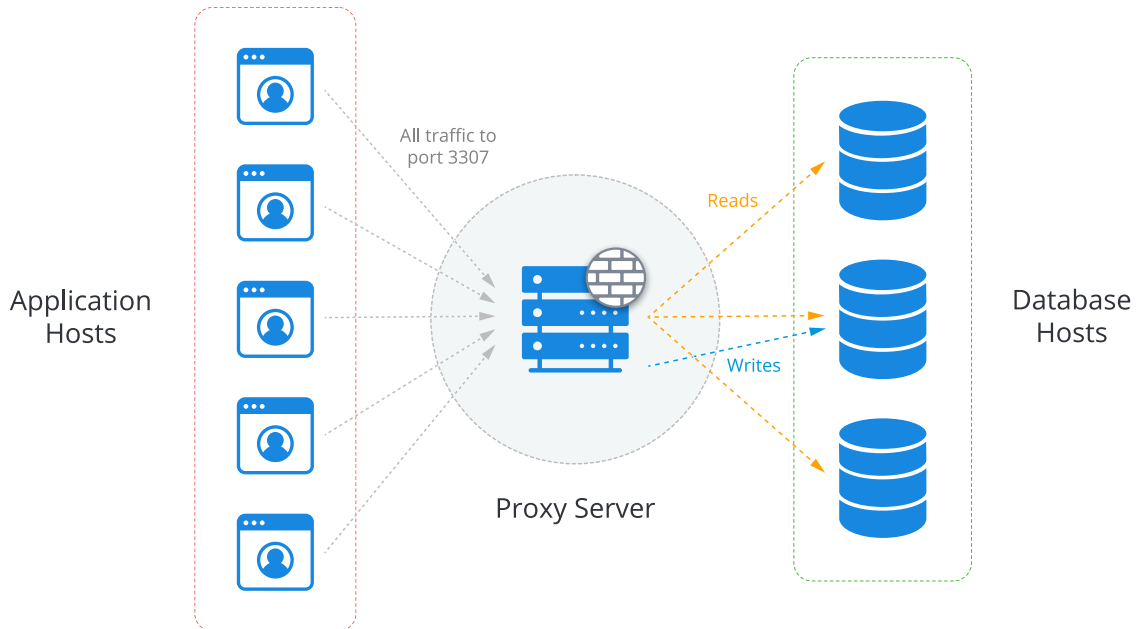
The first and the oldest type are proxies which operate at layer 4 of ISO/OSI network model, the transport layer. Those proxies work on a package level. They receive TCP sessions and they route the traffic to pre-defined backend services.



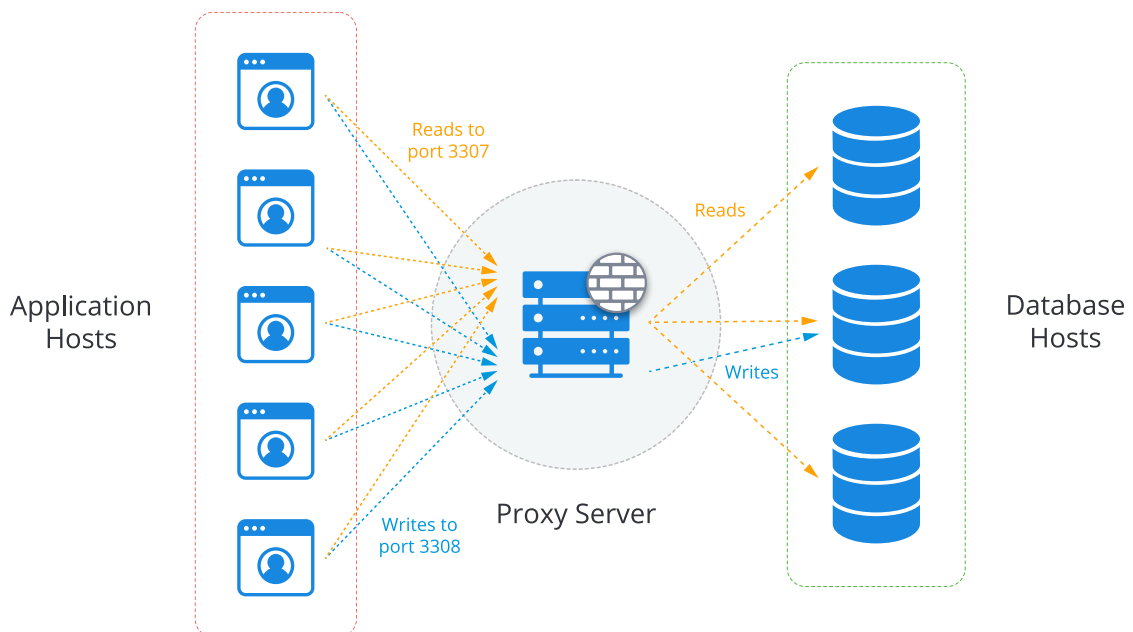
Such proxy server does not care about what it routes, as long as it can send it to the backend and it's in line with load balancing policy. Typically you can pick options like round robin, least amount of connections or sticky connections coming from one frontend host to the same background host. The best known example of such proxy is HAProxy, Nginx is another one.

The main problem with this type of proxies is that they operate only at the network layer. They do not understand the MySQL protocol, nor they understand the states that the MySQL or MariaDB backends are in. For replication setups, it would be a master or a slave. For Galera cluster this will become much more complex (Primary, non-Primary, donor or desynced, joining, joiner etc.). External scripts had to be developed to make it possible for such proxies to understand the state of MySQL backends.

An example of such a script is Percona's clustercheck and all its modifications. The lack of understanding of the MySQL protocol results in more complex connectivity to the proxy. As we mentioned earlier, an application would ideally connect to a proxy and send all its traffic there and the proxy will direct writes to a single host and scale reads across all of MySQL backends.



Unfortunately, if a proxy cannot read the MySQL protocol, it cannot distinguish SELECTs from other queries. This is a serious issue - in a replication environment, you typically have just one host to send your writes to - the master. Galera can work in a multi-writer setup, but sometimes there are cases that mandate applications to direct all of the writes to one node in order to reduce conflicts between writes. As a result, the setup is less transparent to the application as it is the application which has to sort SELECTs and other queries and send them to correct ports on the proxy (assuming one has defined separate write and read-only ports).

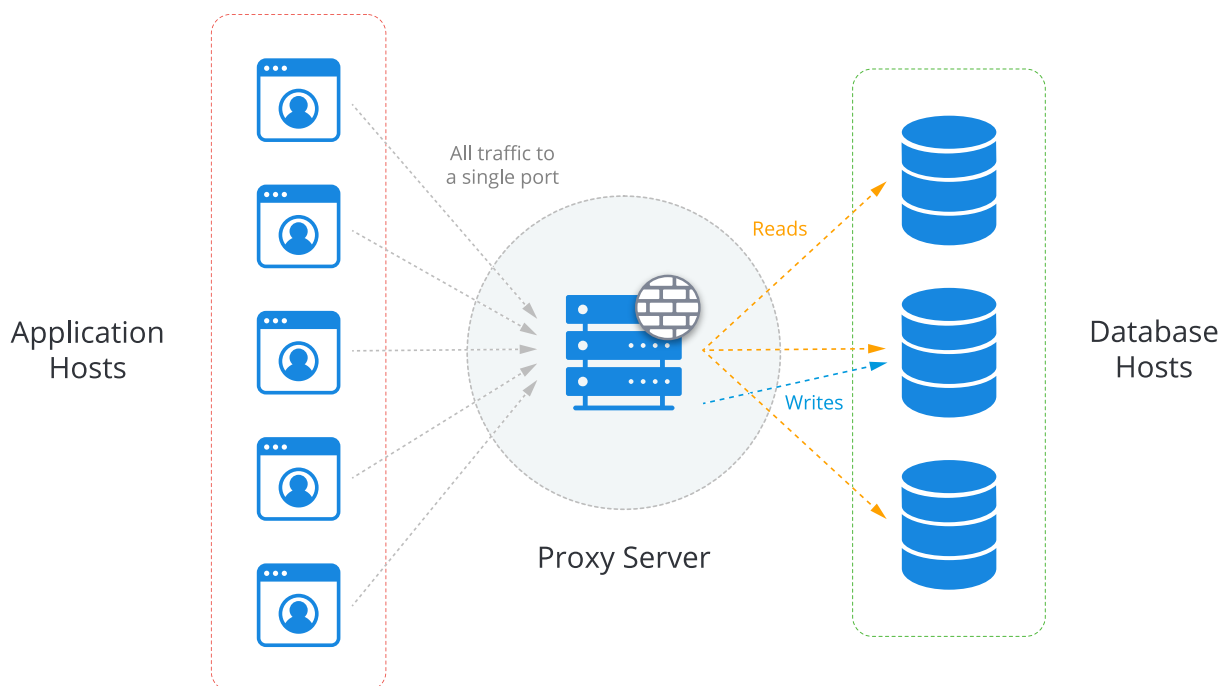


1.2.2. SQL-Aware Proxies

The other type of proxy is the SQL-aware proxy. That software can understand MySQL protocol and, typically, offer a range of features related to that fact. First of all, such proxies should be able to understand the MySQL state. They are designed to differentiate between master and slaves. Some of them can also track and understand

Galera states. All of this results in faster and more reliable responses to whatever happens to the MySQL topology.

Probably the most welcomed feature is the fact that, given their understanding of MySQL protocol, proxies can perform a read/write split. This makes it possible to implement a transparent proxy layer and ensure that application will not have to track anything related to the database tier. It will just connect to a predefined host and port, and that's all it needs to know.



Of course, the fact that the proxy handles all the database traffic makes it possible to use the proxy for other things like traffic shaping, query routing, query blocking etc. It's important to note that not all of the proxies share the same feature set. Some, like MySQL Router, can do the query routing but this is mostly it. Others, like ProxySQL or MaxScale, can be used to perform advanced tasks and can make it possible for a user to significantly alter the way how the traffic is sent to the databases.

SQL-aware proxies, typically, do not use external scripts to monitor or track the state of the databases. They rely on built-in tests which are intended to take care of that. One exception is ProxySQL and Galera cluster monitoring. Up to version 2.0 ProxySQL relied on external script, which was intended to track the state of Galera nodes. The internal support was introduced in ProxySQL v2.0, which, at the time of writing, is in beta state.

Features of Advanced, SQL-Aware Proxies

We discussed the difference between SQL-aware and non-SQL aware proxies. We also touched a bit upon the features that make the difference between both types. In this section we will take a deeper look into them and discuss why advanced proxies became so popular and what are the main reasons and use cases for them.

2.1. Query Routing

Query routing is one of the most common features people are looking for in modern, SQL-aware proxies.

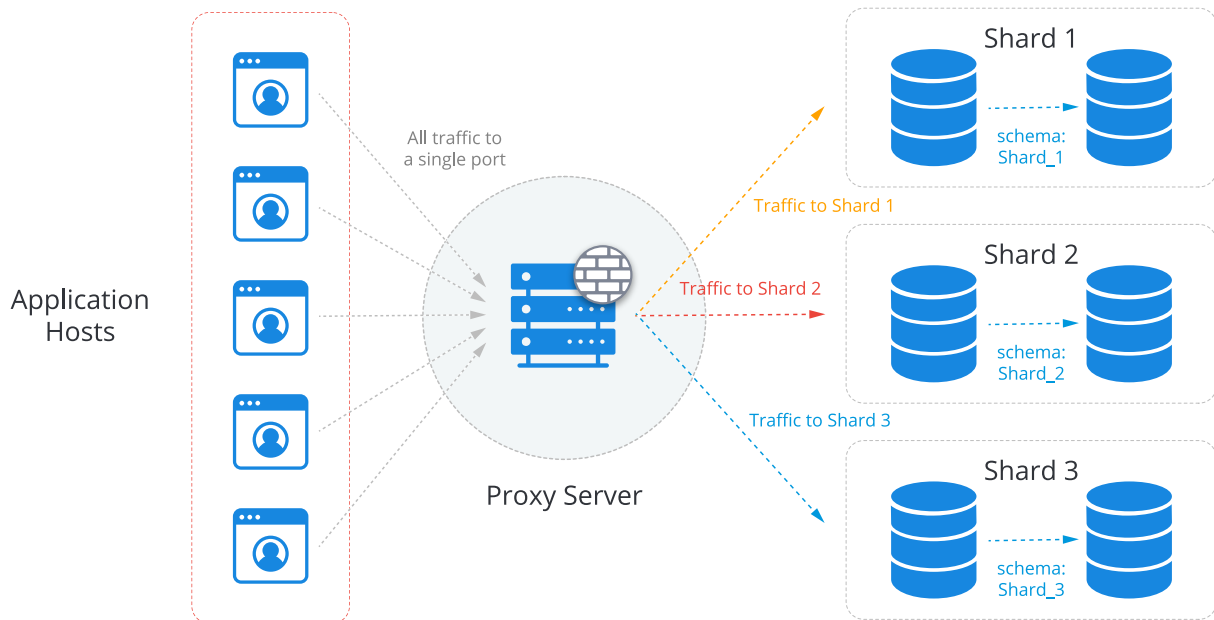
The ability to define how queries should be routed is very important. The main use case is the read/write split which allows the database environment to be almost transparent to the application. It is not the only use case, though and some of the advanced proxies like ProxySQL or MaxScale can utilize regular expressions to match any query and route it appropriately.

Depending on the proxy, you can direct queries in a more or less precise way. ProxySQL, for example, uses hostgroups as targets for query routing so it is not just master or replicas you pick between. One can route some of the queries to a particular host. This can be very useful in numerous cases. For example, let's imagine that, on top of your application traffic, you want to execute some reporting queries.

With ProxySQL, if you prepared a separate host which is supposed to handle those queries, you can create rules with regular expressions that would match your reporting queries and direct them to that particular replica. It doesn't have to be a regular expression. Maybe the reporting module uses some particular users or schemas. If so, traffic can also be routed based on a query being executed by a particular user, or a query being directed towards a particular schema.

Query routing can also be used to build an abstraction layer on top of a sharded environment.

Let's imagine a setup with several shards created using different schemas. You can use either ProxySQL or MaxScale to detect to which schema queries are being sent and redirect them to a correct shard. You can also leverage the read/write split feature to ensure that queries are routed correctly to masters and replicas of given shard. Such option makes it really easy to implement simple sharding scenarios.



2.2. Query Rewriting

If a proxy can understand the MySQL protocol and can read a query to route it correctly, can it modify the query itself? Yes, it can and this is one of the most amazing features of the modern proxies - an ability to perform a query rewrite on the fly. Of course, you have to work within constraints of the application. If your application assumes a given number of columns in the result set, it might be impossible to change your query into something which returns a different set of columns. Luckily, this limitation will not prevent the operations team from leveraging this feature to ensure database servers will not be impacted by badly written queries.

For starters, as long as the result set is the same, you can rewrite the query into any format you like. Wanna move that `IN()` subquery and rewrite into `JOIN`? No problem. Would you like to remove some of the index hints which are outdated? Here you are. What about adding new hints to force MySQL optimizer to use a better index for a given query? It's just a few minutes away. The biggest problem this feature solves is the amount of time it typically takes to modify queries.

Most likely people running databases, be it operations staff, sysadmins, SRE's or DBA's, do not have access to the application code. This is quite an expected pattern to ensure security and reliability of the application. This also means that you have to file a ticket with the development team, or otherwise alert developers that a code change is required to rewrite some poor SQL statements. This has to be picked up by a dev team, it has to be implemented, tested and deployed. You simply cannot expect it to happen "now", while it's "now" that the database is struggling under inefficient or badly executed SQL.

With a modern SQL-aware proxy, all this is not required as one can just modify the proxy configuration to implement query rewriting. If you need to add an index hint, you can do it within minutes (a change itself will take seconds but you have to allow some time for testing it on your staging environment before applying it to production).

2.3. Query Caching

Queries have to be cached, there is no other way to scale an environment to “web-scale”. In the past, a dedicated caching layer was the most common choice. It typically consisted of some sort of key-value datastore like Redis, memcached or couchbase. This adds complexity - such datastores are not without their own problems. For instance, you have to write code to maintain the cache and invalidate values that have been updated in the database. They can become quite complex to scale as well - different key-value stores implement some sort of replication for redundancy. It takes time and resources to maintain the caching layer.

These days, given that proxies are in place to receive the whole traffic to the database tier, and given that modern proxies have an ability to cache result sets, the need for dedicated caching layer is significantly reduced. Why keep a dedicated Redis setup if you can cache your results in ProxySQL?

To make it even faster, you can collocate your ProxySQL instances with your application hosts and access them through Unix sockets to reduce latency even further. We will discuss deployment patterns later in this whitepaper, but such deployment described above is quite common and it would significantly reduce the latency related to accessing cached data.

2.4. Traffic Control

Caching or rewriting sometimes is not enough to reduce the impact coming from badly engineered queries. Sometimes you cannot rewrite the query because the query is in optimal form, and there’s way too many queries like that which are overloading the database. This can happen for numerous reasons. For example, a bug within an application causes a query to be triggered with incorrect WHERE clause and, as a result, instead of using an index, that query runs a table scan. Modern proxies can help even in this case.

2.4.1. Kill a Query

For starters, in such cases it is quite common to just go ahead and prevent such incorrect query from running. There are numerous ways of doing it, including rewriting such query into a “SELECT 1”. Some of the proxies have dedicated options to kill such query on a proxy level. If a query matches particular conditions, it will not be sent to the database at all, reducing the impact from such queries virtually to 0.

2.4.2. Slow Down Traffic

There are cases where killing of a query is not an option because it is a valid query, even if it is badly optimized. In that case you can try to reduce how often a query can be executed.

In ProxySQL you may notice a “delayed” column when you create query rules (as seen from the ClusterControl screenshot below).

This is basically a way of reducing the rate of execution for a given query. You can set it, let’s say, to 10000 ms allowing it to run not often than every 10 seconds.

Database Clusters Cluster-6

Activity Deploy Import Global Settings Logout

Limit: 100 Order By: Rule Id Sort: asc

Add new Rule

Rule ID: 400

Match Pattern: NULL

Replace Pattern: NULL

Proxy Address: NULL Proxy Port: NULL

Digest: NULL

Match Digest: NULL

Comments: NULL

Username: NULL

Client Address: NULL

Destination Hostgroup: Choose one...

Schema Name: NULL

Rule Settings

Active:

Timeout (ms): NULL

Retries: NULL

flagIN: 0

flagOUT: NULL

Reconnect: Default Disabled Enabled

Log: Default Disabled Enabled

Sticky Conn: NULL

Multiplex: NULL

Apply

Cache TTL (ms)

Delay (ms)

Mirror Hostgroup

Mirror flag out

Negate Match Pattern:

Number of milliseconds to delay the execution of the query. This is essentially a throttling mechanism and QoS, allowing to give priority to some queries instead of others. This value is added to the mysql-default_query_delay global variable that applies to all queries. Future version of ProxySQL will provide a more advanced throttling mechanism.

Designing Highly Available Proxy Architecture

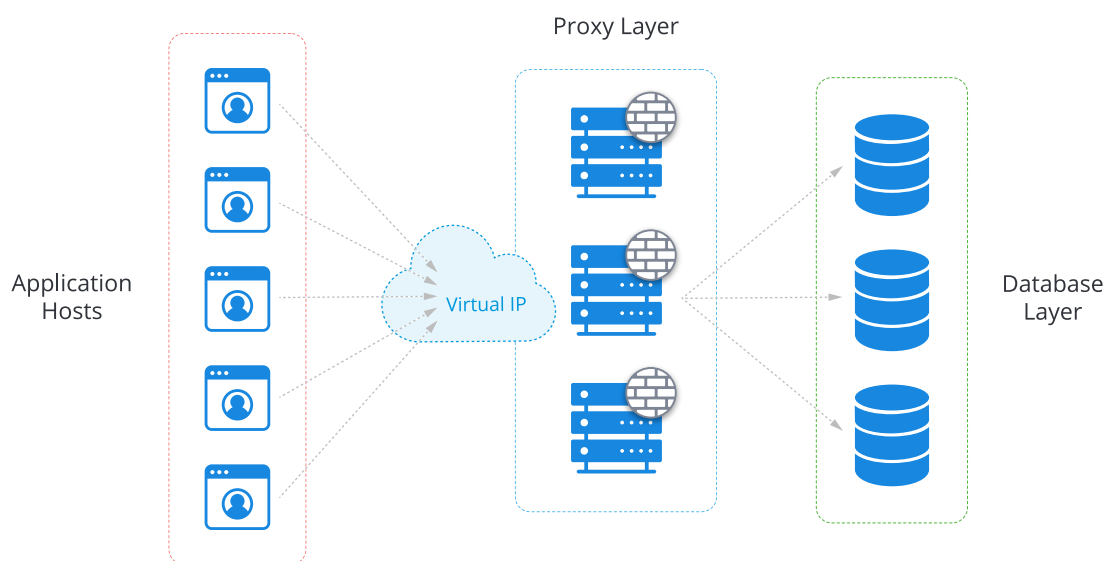
Based on what we covered so far, it shouldn't be a surprise that the place of the proxy in modern database environment is rather central. Proxies accept all of the database traffic and passes it to the backend hosts. Proxies analyze the traffic and, according to the configuration, perform actions on it - route queries, shape the traffic, make the whole complexity of database tier transparent to the application.

As can be expected, this creates a huge pressure to make proxy layer highly available. If a proxy layer is not reachable, it is as if the whole database tier is not available to the application. How can we design the proxy layer to make it highly available? In this chapter, we will go over some of the deployment patterns, discussing their pros and cons.

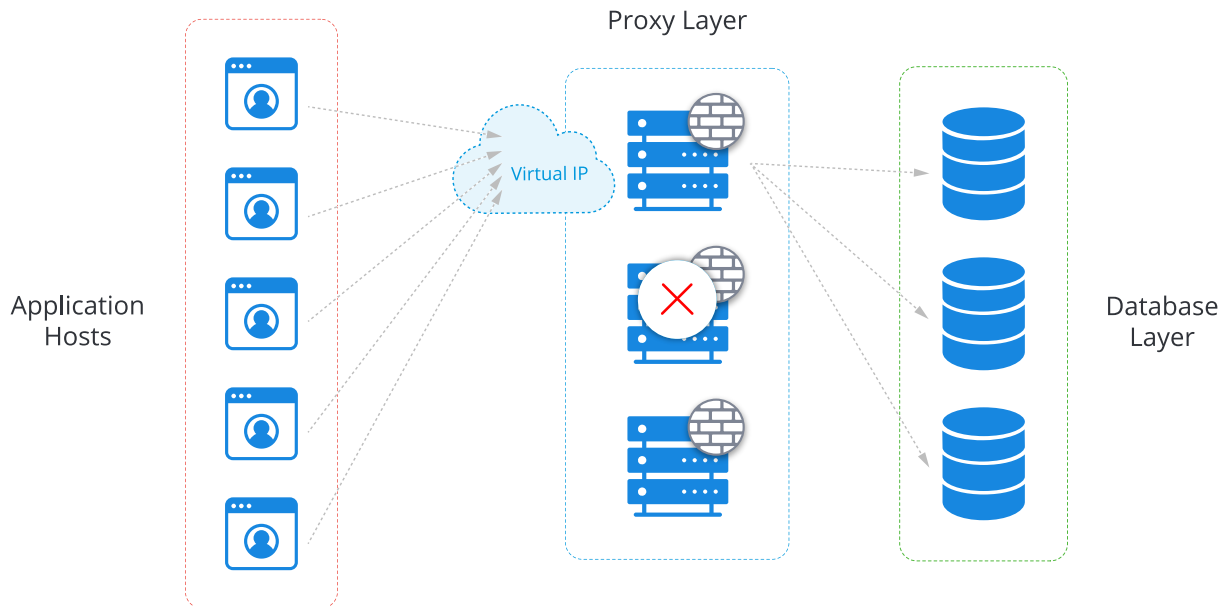
3.1. Dedicated Proxy Instances

One way to deploy a proxy layer would be to use several dedicated instances to setup proxies. Depending on the setup, you can route traffic to one of them, leaving others in standby mode, ready to take over the traffic should the active proxy fails. It is also possible to implement some sort of a round-robin type of routing where all proxies are getting connections. Let's see some examples of both types of the setup.

3.1.1. A Proxy with a Virtual IP Assigned



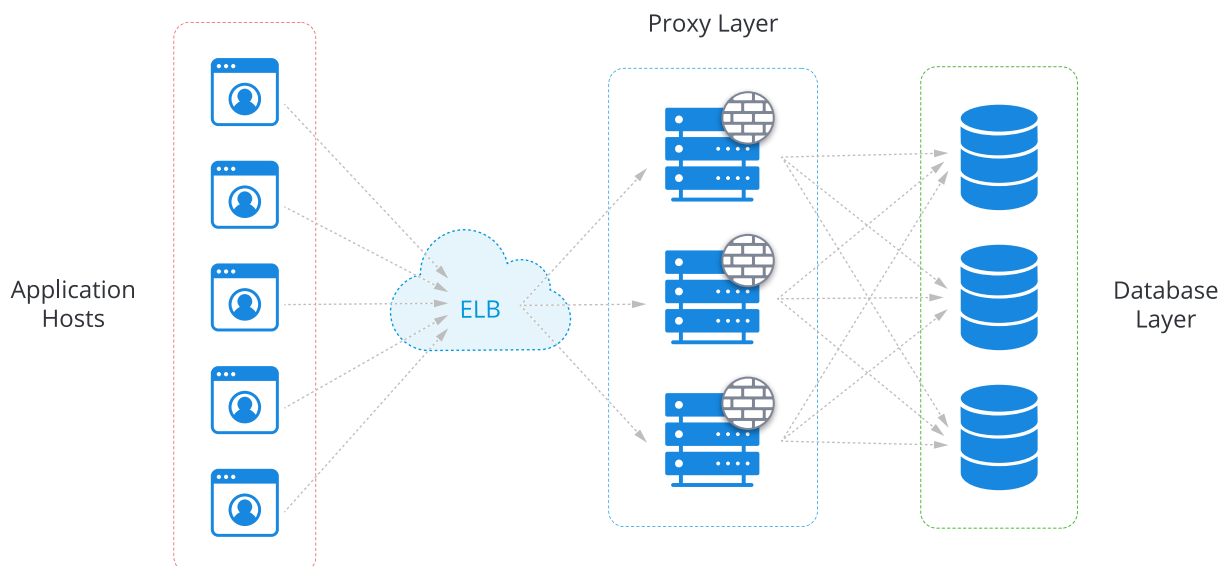
In this setup you will see a single active proxy with a Virtual IP (VIP) assigned to it. The remaining proxy nodes are acting as standby instances. Should the active node fail, the VIP would be reassigned to one of the remaining proxies.



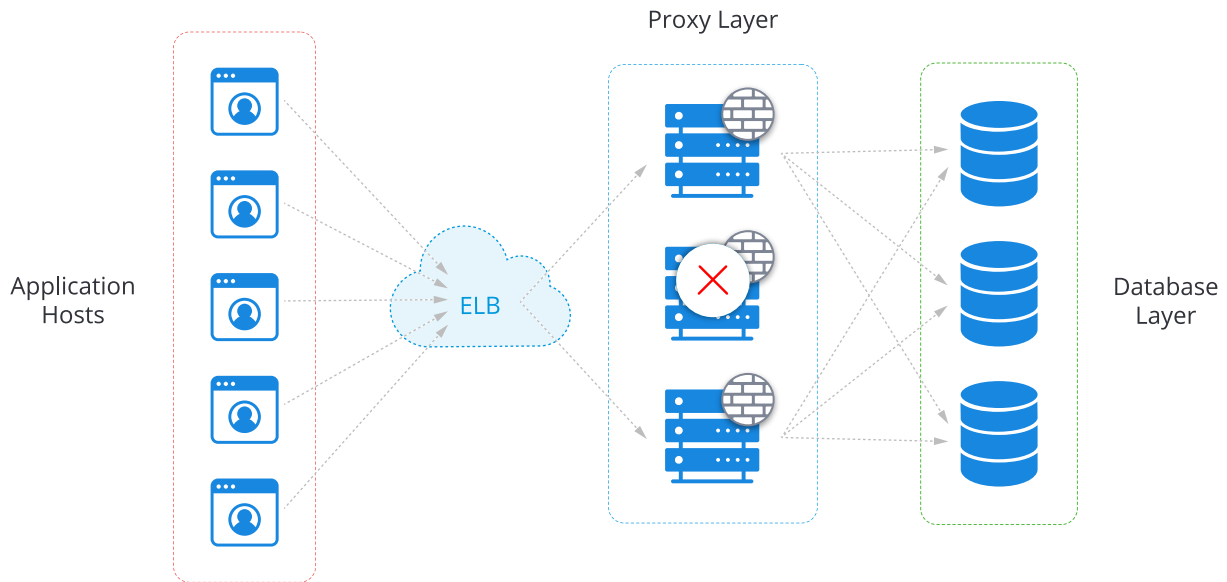
How exactly the virtual IP is managed would be up to the user. One of the methods could be to leverage Keepalived to track the health of proxy nodes and manage VIP as needed. Instead of Keepalived, one can use Pacemaker/Corosync - it also has an option to manage a floating IP and track the state of the underlying nodes. If we are talking about a setup deployed in a cloud environment, more options are becoming available.

3.1.2. Elastic Load Balancing

What's better than a proxy layer? Two layers - in cloud setups we can easily leverage building blocks like ELB, which are readily available.



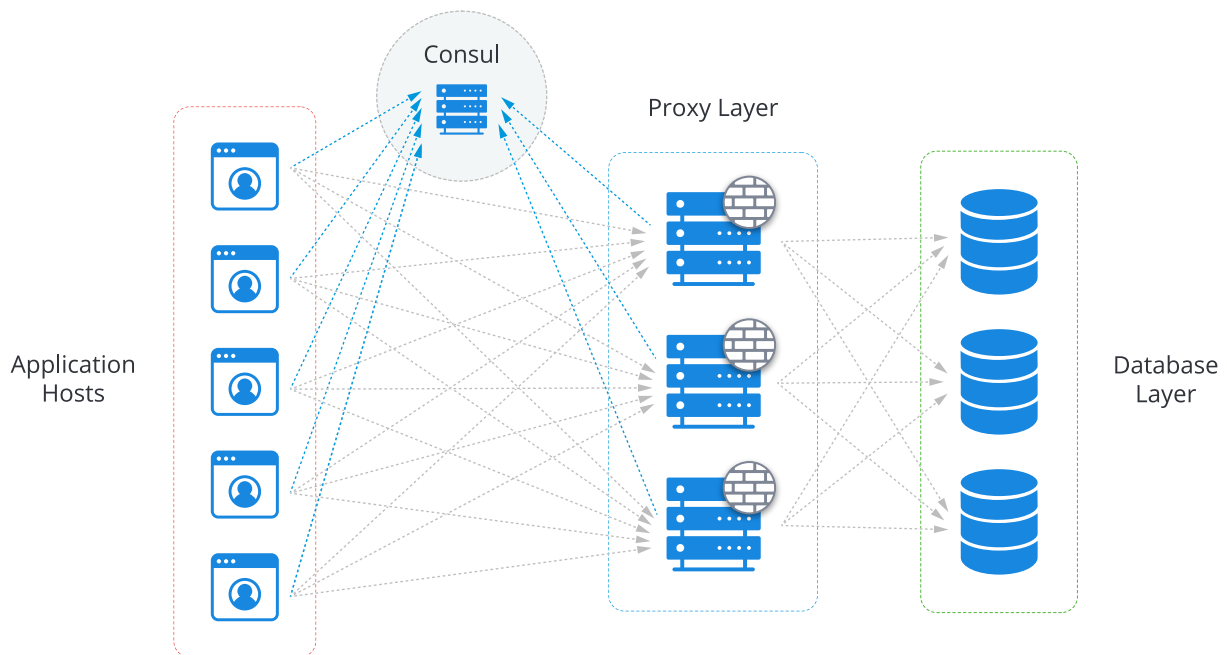
ELB is already highly available so there's no need to be worried about its availability. It can also automatically scale, so there's no need to manage it. A black box, which happens to do what we want to achieve - when configured with proxies as backends, it will attempt to detect if the backend nodes are up or not, and if yes, it will send the traffic to them.



Should the issue be detected, the healthchecks will fail and the problematic backend is removed from the pool. Traffic will not be sent to it until the healthchecks get back to normal.

3.1.3. Application-Side Proxy Discovery

Another way of building high availability of the proxy layer would be to let the application know what proxies are available and how to connect to them.

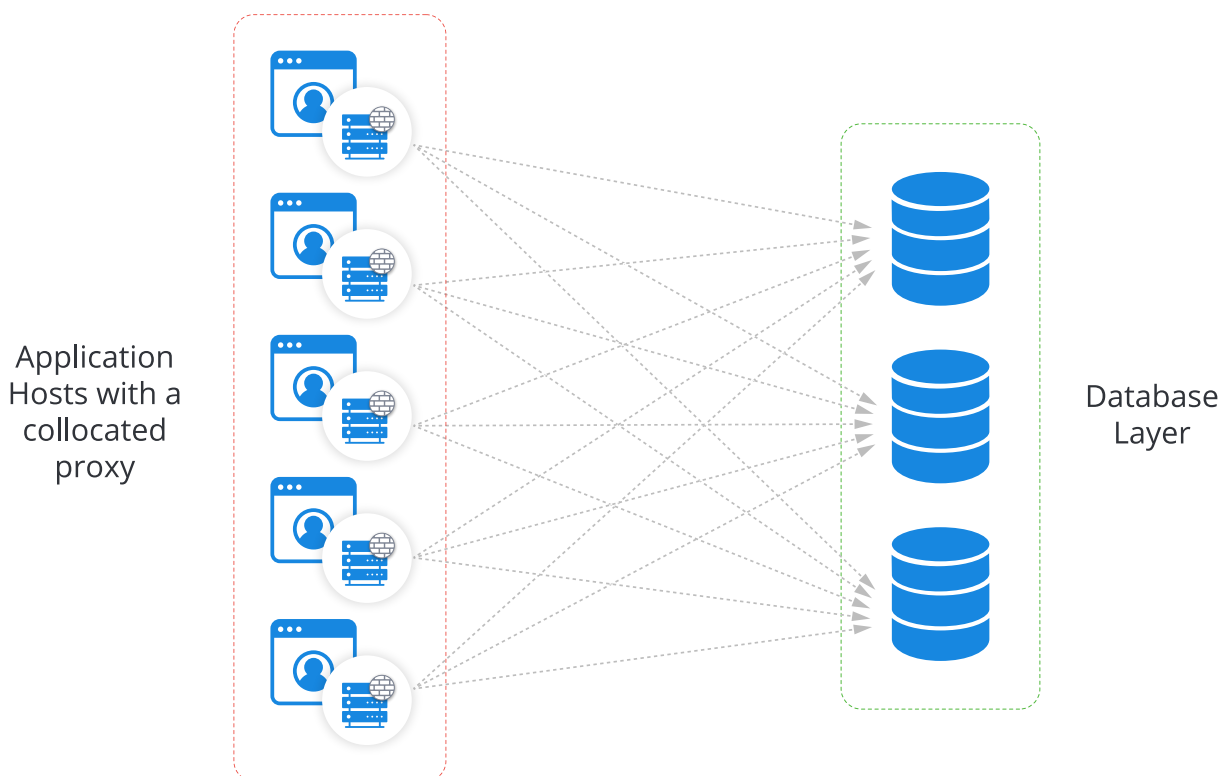


This can be done in many ways. Using a service discovery approach, one would have the proxies register in Consul and then let application nodes connect to Consul and get a list of proxies to connect to. Application hosts would then try to connect to the proxies from the list, hoping that it will be possible to get through. You can have some sort of health check which would validate if entries in Consul are indeed reachable or not.

Another option might be to hardcode the proxy layer in the application, and prepare code which would attempt to connect to proxies using a round-robin (or smarter) algorithm. This is much less flexible option as hardcoding anything in the application means you have to change application to add new proxies or change IP/hostname of an existing proxy. But in smaller environments, this can be a viable option.

3.2. Collocating Proxies on Application Hosts

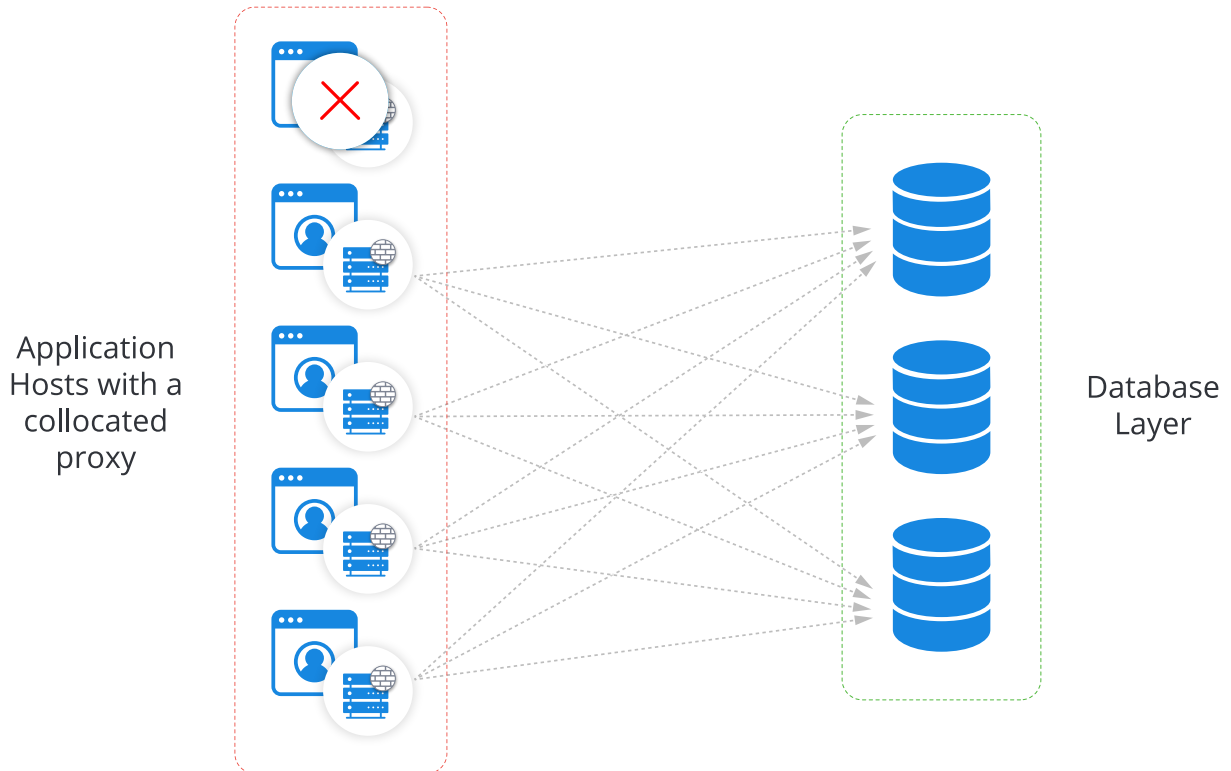
Another way of deploying proxies might be to collocate them with application hosts. This has its own set of pros and cons. Let's consider following setup:



We have five application hosts, each application has a collocated proxy. First of all, we have to keep in mind that the proxy is typically CPU-intensive process. Especially SQL-aware proxies which not only route packets, but they also have to examine packages, read the data sent in them, and process this data to, for example, rewrite a query.

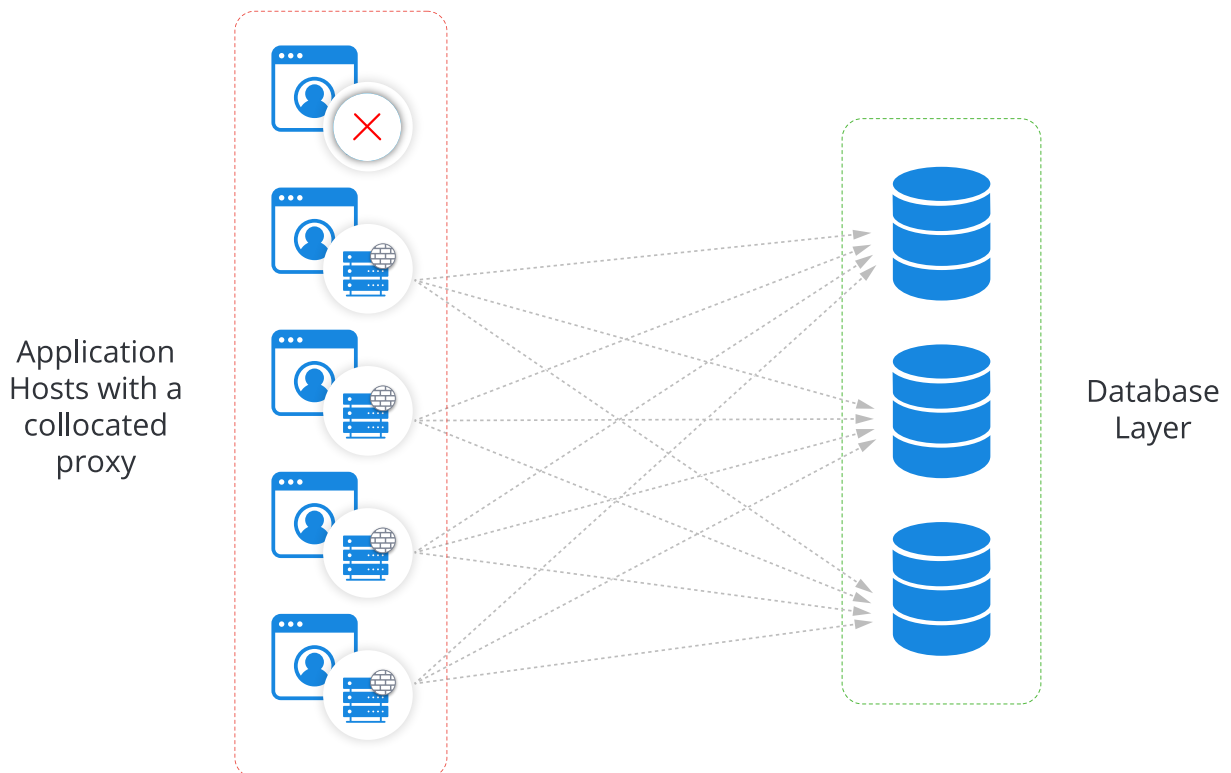
This adds significant CPU load and such collocated services may require a larger instance size than just a web host. On the other hand, typically you will see more proxies in such deployments thus the required size of an instance is reduced compared to two or three dedicated proxy instances.

Another concern may be for example that the whole instance fails. This is perfectly fine because both application and proxy server will not be available.

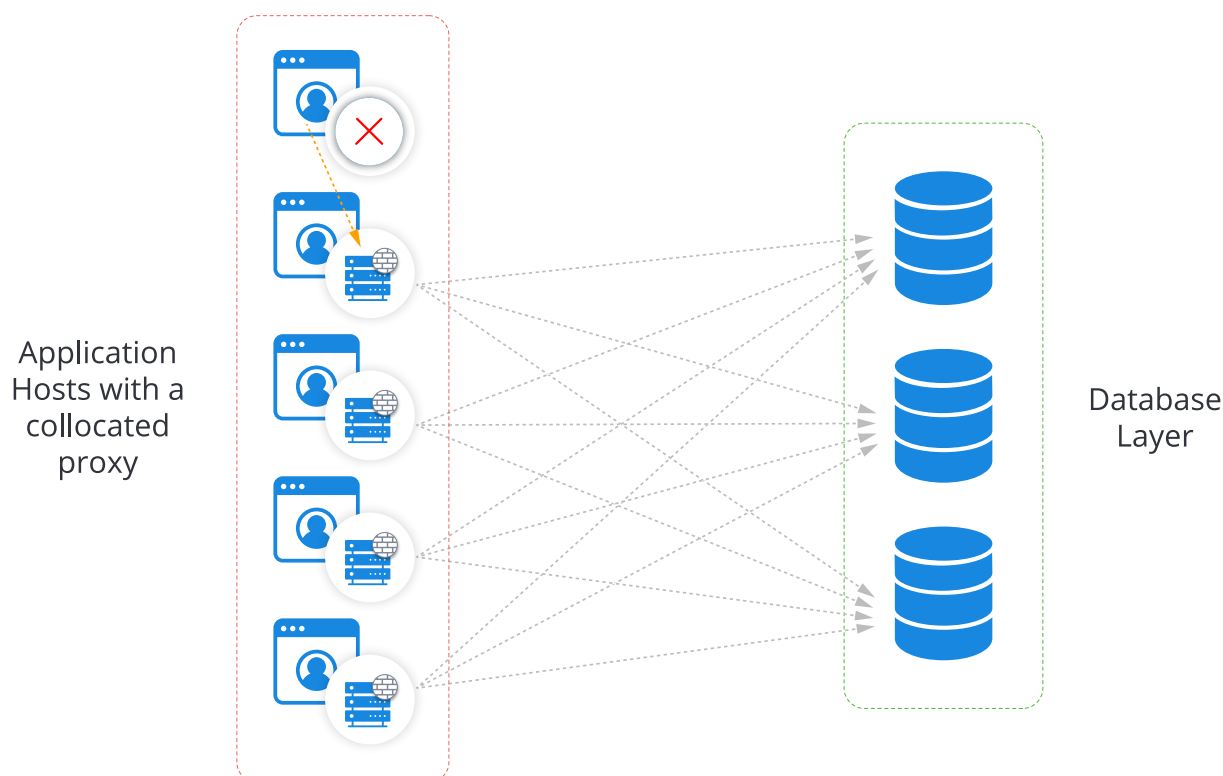


Application host failures have to be handled one level above what we discuss here, using, for example, hardware load balancer or simple HAProxy setup. From a database tier it does not matter how it is covered. What's important is that it should already be handled separately from the database.

You may ask - what would happen if only the proxy failed? Segfault, OOM killer, something else. In that case, errors will show up as that particular application host won't be able to connect to the local proxy instance.



There are ways to handle such situations. First of all some of the proxies, like ProxySQL, are designed to reduce the impact in this particular case. ProxySQL uses an angel process to monitor the health of the ProxySQL daemon. If a crash is detected, a new process is started. This can reduce downtime even to a fraction of second, making it



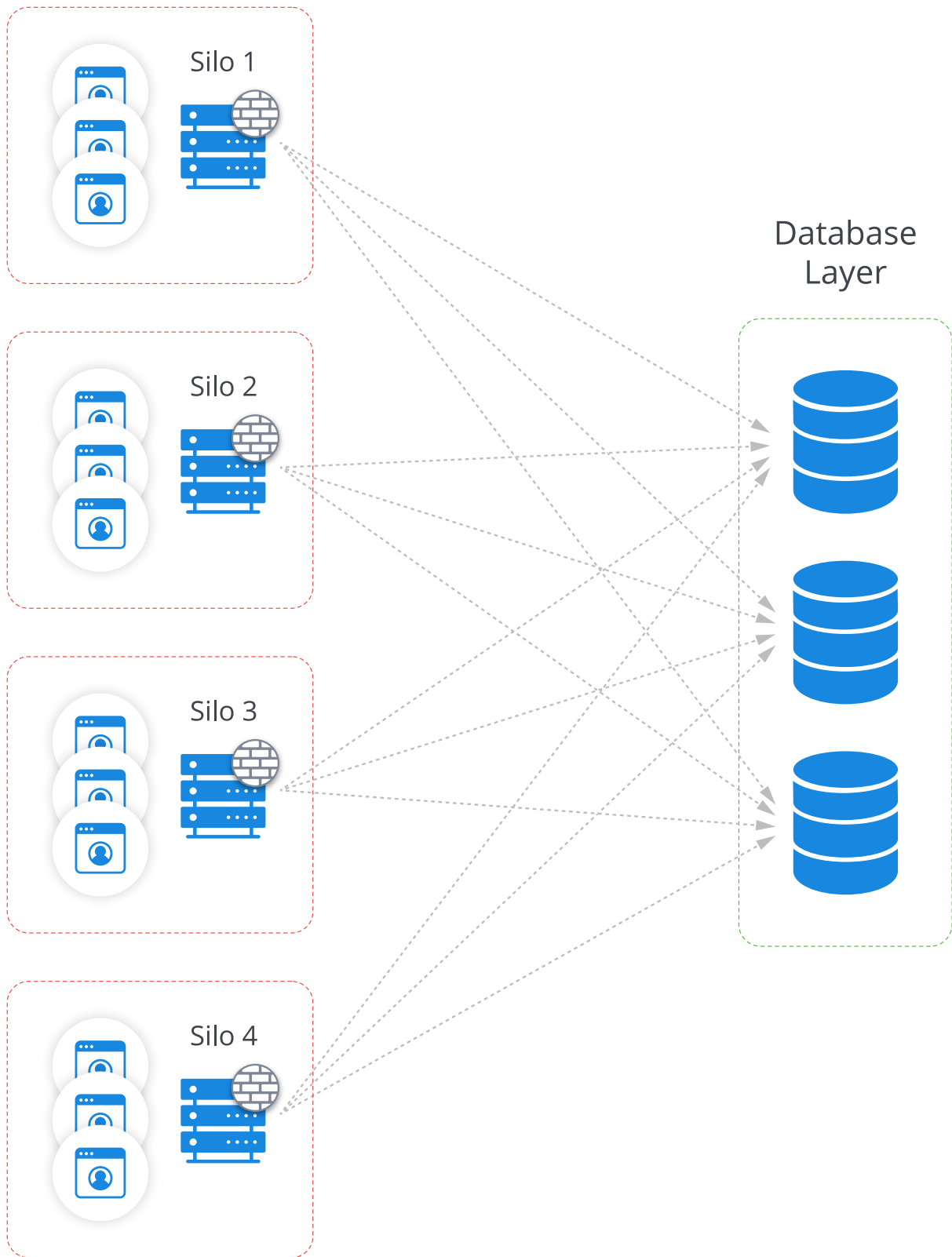
quite easy to handle from the application. The application can just retry its attempt to connect to the proxy. If the crash was not a one-off case but it is happening over and over again as your environment keeps hitting some bugs, this will not help. For such cases you can implement a "backup" proxy.

Should the local proxy not be available, an application can have (or get from external service like Consul) a list of other proxies to connect to over TCP.

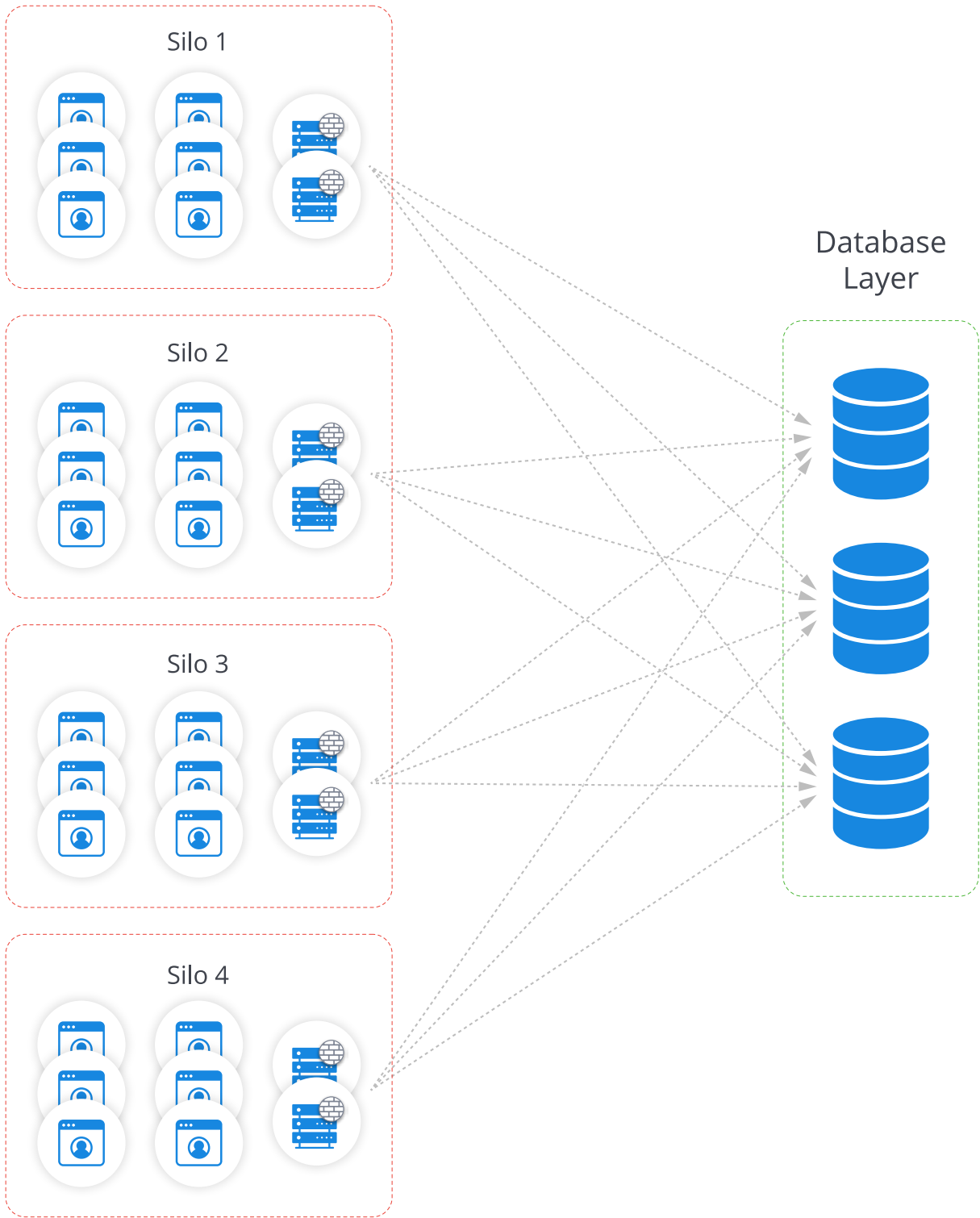
3.3. Silo Approach

If the number of application hosts grows to a high number, it may be not feasible to deploy and manage tens of proxy instances. In that case, a silo approach might be considered. Instead of deploying a proxy per web host, we could deploy a proxy per number of web hosts.

Before, we have shown 5 application hosts covered by 5 proxy instances. In the diagram below you can see 12 application hosts and 4 proxy instances. Of course, if a proxy goes down, three application hosts won't be able to connect to the database tier. Sometimes this is perfectly fine per SLA - it all depends on how the SLA is defined and how big percentage of users should be affected by an incident before it is treated as a critical one. Of course, you can also implement redundancy within a silo.



Instead of one proxy, use two. To keep the application host to proxy ratio, double the application host number per silo (see previous example).



Managing Large Proxy Deployments

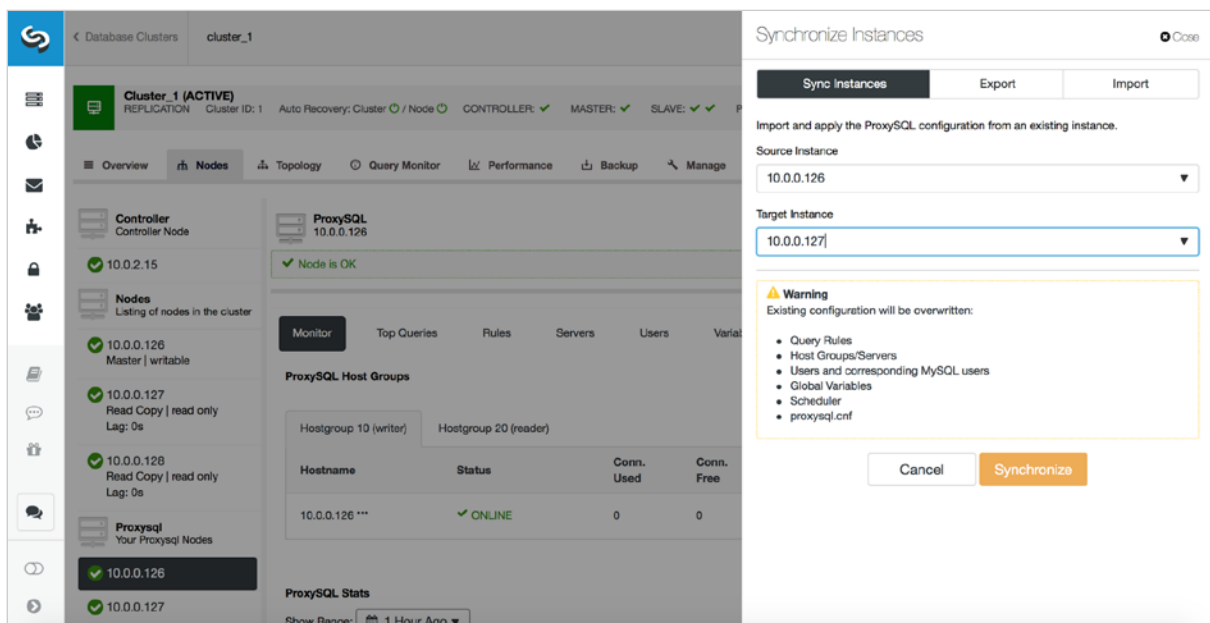
As we have previously seen, proxy deployments can quickly become large. This poses a challenge as to how to manage them. Generally speaking, all proxies should have the same configuration, otherwise an incorrect behavior might be triggered and such issues are typically tricky to debug. On the other hand, configuration changes in proxies are rather common - the margin of error can be quite significant. It also means that managing configuration changes on all proxies manually would be quite a tedious and time-consuming task.

In this chapter we will discuss methods which can be used to keep the configuration in sync.

One popular method is to rely on infrastructure orchestration tools like Ansible, Chef, Puppet or Salt Stack to keep the configuration of the proxy layer in sync. All configuration changes should be executed through the tool - this speeds things up, reduces any manual work and helps to keep configurations in sync. This requires a deep understanding of how the proxy works, as the behaviour has to be coded by the user.

Another method consists of relying on platforms like ClusterControl, which have specific logic built-in on how to deploy and manage proxies.

In the case of ProxySQL ClusterControl provides a way to sync configuration between instances.



The screenshot displays the ClusterControl web interface for a ProxySQL cluster. The main view shows the cluster 'cluster_1' with a list of nodes and their status. A 'Synchronize Instances' dialog box is open, allowing configuration synchronization between instances. The dialog includes fields for 'Source Instance' (10.0.0.126) and 'Target Instance' (10.0.0.127). A warning message states: 'Warning: Existing configuration will be overwritten: Query Rules, Host Groups/Servers, Users and corresponding MySQL users, Global Variables, Scheduler, proxysql.cnf'. Buttons for 'Cancel' and 'Synchronize' are visible at the bottom of the dialog.

This lets users make a change on one proxy and then lets ClusterControl perform a sync to other proxies in the setup.

Finally, ProxySQL has a clustering feature. When configured to build a cluster, ProxySQL will automatically sync any changes that happen on one of the clustered proxies with the rest of the cluster.

Setting up Highly Available Proxy Layers with ClusterControl

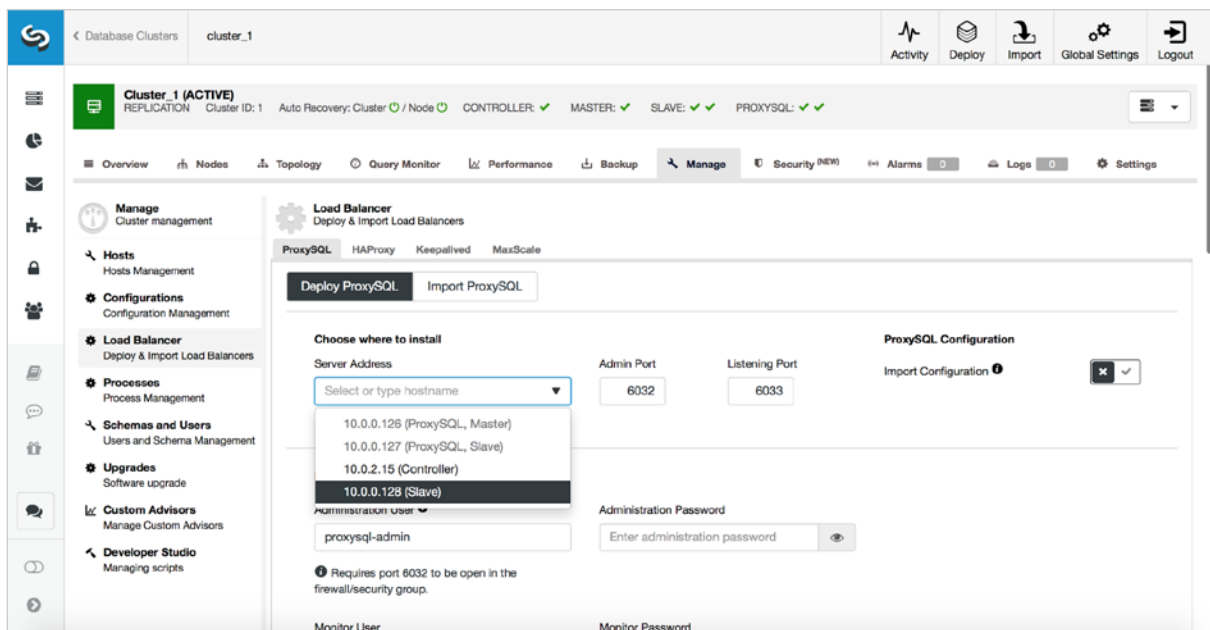
The deployment of a highly available proxy layer is very time-consuming when done manually. You can use infrastructure orchestration tools to automate the process although it still takes time to prepare playbooks and code the logic, or modify existing ones to match your environment.

With ClusterControl you can easily, in just a couple of clicks, deploy such a setup. ClusterControl uses a floating VIP approach for high availability.

It is also possible to deploy proxies on any node, including application hosts, to build either colocated proxy infrastructures or go for a silo approach.

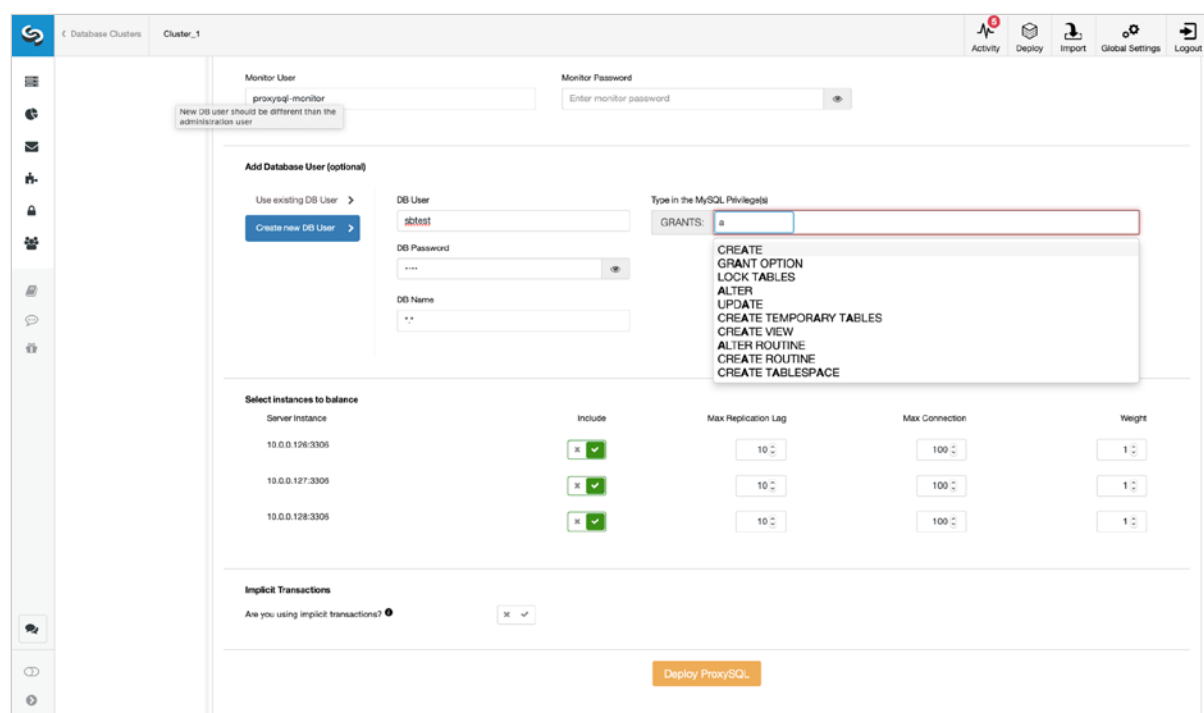
Below we are going to show how to easily deploy ProxySQL and HAProxy in a highly available manner using ClusterControl.

5.1. ProxySQL Deployment



First of all, you have to pick where ProxySQL should be deployed.

It can be either one of the hosts in the cluster or you can type any IP or hostname you like. If you already have ProxySQL deployed, you can import a configuration from it. If not, you need to fill the rest of details.



Fill in the password for the administrative user for the CLI and the password for the monitoring user, which ProxySQL will use to reach the databases and do additional monitoring.

ProxySQL has to know the access details for the MySQL users which will connect through ProxySQL. This is required as the authentication is done against ProxySQL instead of the backend.

After that, ProxySQL opens a backend connection. If you have users already created in MySQL, you can import them to ProxySQL. If not, you can easily create new users on both the ProxySQL and MySQL backend.

The authentication process goes as follows:

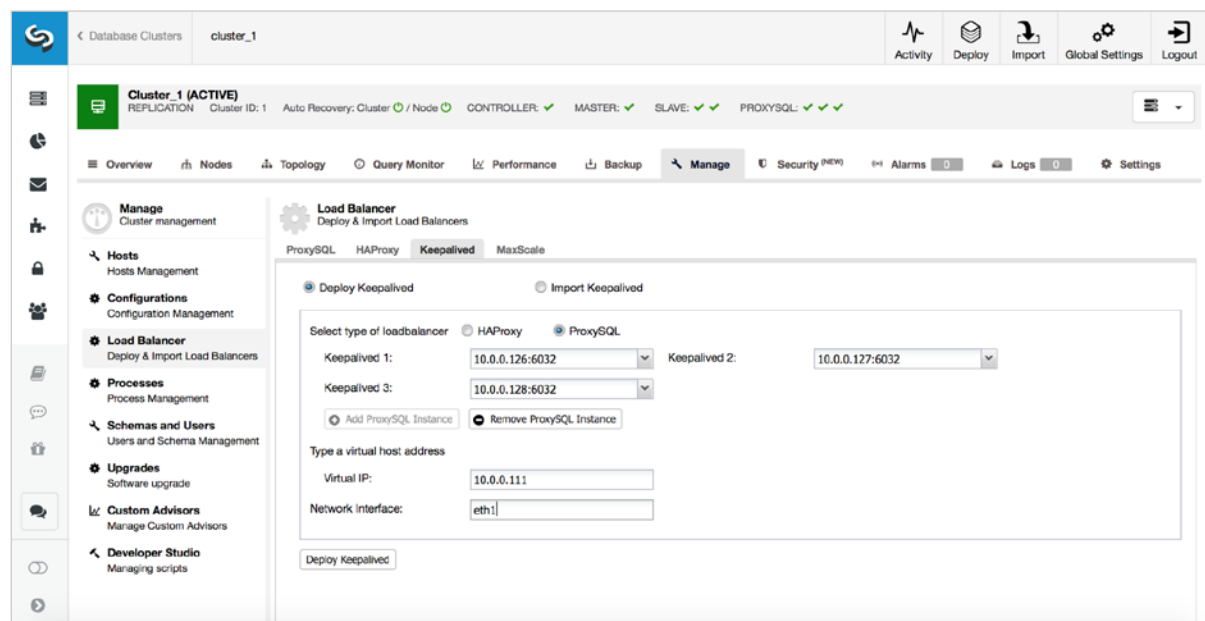
First, the application connects to ProxySQL. It authenticates with the user and password. ProxySQL verifies it has such a user defined and accepts the connection. The application sends the first query. ProxySQL processes it and, based on query rules, opens a connection to one of the nodes defined in the hostgroup to which the given query should be routed. It uses the same user and password that the application used to connect to the proxy. MySQL then verifies the user, password, host and privileges and, if all is good, a connection is opened to the backend.

If there are already open connections, ProxySQL will attempt to reuse them. As long as it is possible (for example no session variables were set on that connection), one of the existing connections will be used to connect to MySQL.

After you decided on the user to be imported or created, you may want to change some of the settings like the maximum replication lag before a slave is removed from the pool of active servers, the maximum number of connections to a given backend

or weight. You should also tell whether you use implicit transactions created by "SET autocommit=0" or not. This affects how ProxySQL will be configured.

If you use implicit transactions, to maintain transactional behavior only one node will be accessible in ProxySQL - there will be no read scaling, just high availability. If you do not use implicit transactions, ProxySQL will be configured for read/write split and read scaling.



Once the ProxySQL instances are deployed, it's time to deploy Keepalived. You need to pick ProxySQL as the load balancer type, pick up to three ProxySQL instances, define what VIP should look like and on which network interface it should be created.

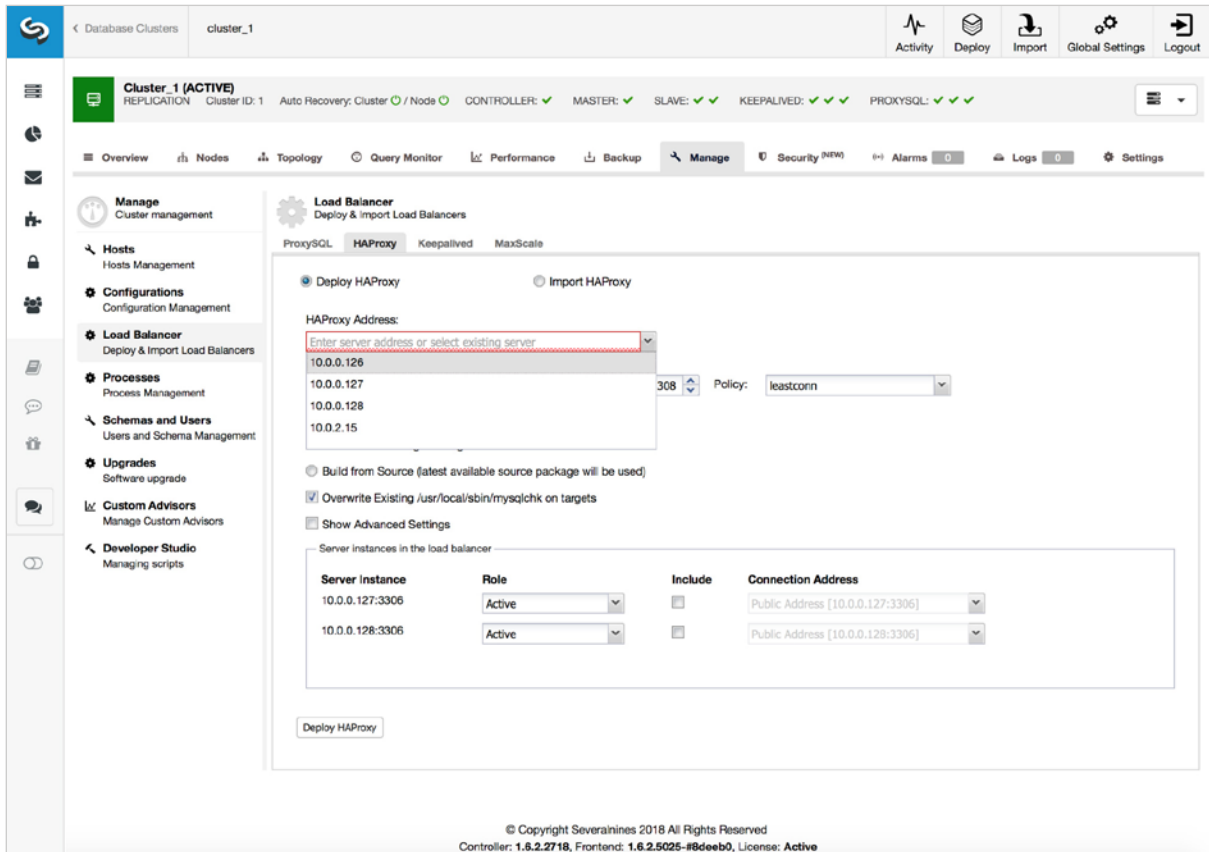
This is all you need to do.

As soon as Keepalived services are ready, the application can use the IP to connect to one of the ProxySQL instances. Should that ProxySQL go down, Keepalived will move the VIP to another ProxySQL instance.

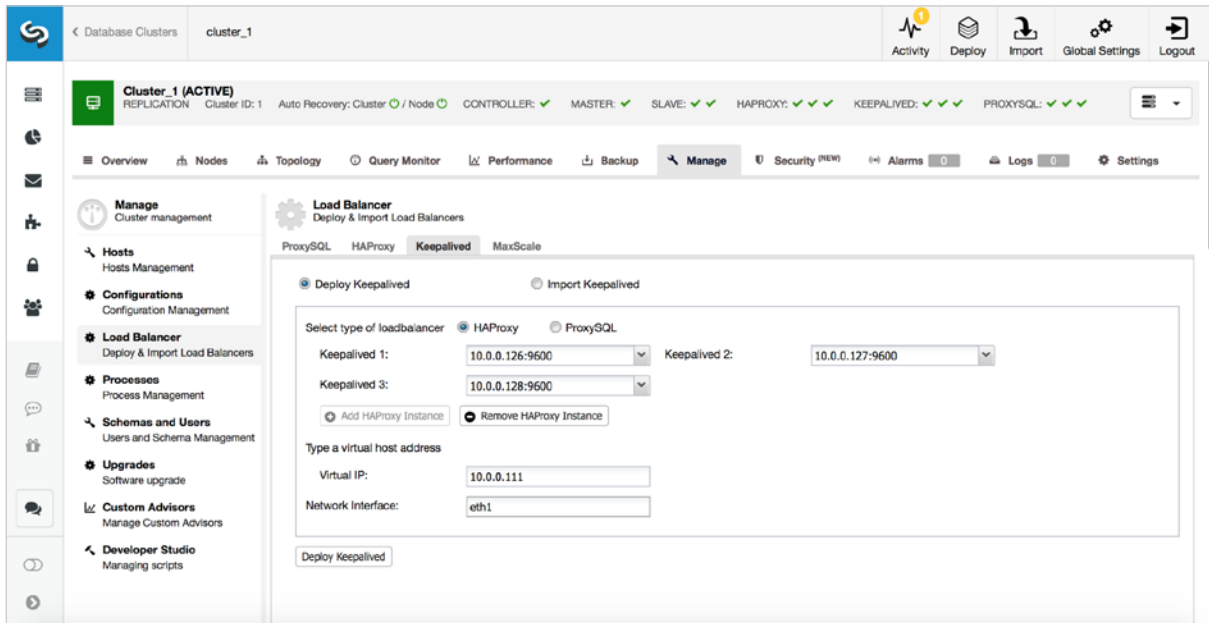
5.2. HAProxy Deployment

ClusterControl also supports a highly available HAProxy deployment. At first, you have to deploy HAProxy.

As with ProxySQL, you need to setup a configuration. You have to pick a host to deploy to either from the list of hosts available in the cluster or you can type your own IP or hostname. You should decide if you want to deploy HAProxy in a read/write split mode, using two backends, or if it should just be round-robin routing. You can define which members of the cluster should be included and what their role should be.



Once you have HAProxy deployed, you should deploy Keepalived.



The process is exactly the same as for ProxySQL, the only difference being that you pick HAProxy as the load balancer type.

Which Proxy Should I Pick?

We discussed some of the different choices you have when deciding which proxy you would like to deploy. In this chapter we will spend some time discussing what the differences are and what might be a preferable choice in a given situation.

6.1. SQL-Aware Proxy or Not?

The main choice is whether to use SQL-aware proxy or just stick to a simple option like, for example, HAProxy. Modern, SQL-aware proxies come with a variety of useful features but it's the simplicity where HAProxy shines.

Proxies like that are very robust and simple to operate. They require less resources to run. Sure, you cannot do fancy things with them but, through external scripts, they can still follow MySQL topology changes the way traffic is routed when a topology has changed. ClusterControl actually deploys the appropriate external scripts with HAProxy to perform healthchecks on different types of MySQL topology backends.

This may be "good enough", especially if your application can perform read/write split on its own and is able to send reads to one backend and writes to another. Also, you should consider what your team is familiar with.

First of all, shiny features of modern proxies might not always be clear on how to use them. Also, HAProxy or Nginx are quite popular outside of the database world. If you use them already and have operational experience with maintaining that software, tuning it, debugging issues etc, it might not be a bad idea to actually use those tools also for databases. Sometimes all you want is a simple load balancer, no bells and whistles.

On the other hand, modern proxies are powerful and, if used correctly, can significantly help with handling the database traffic. If you are in a position where some of those features can help you with your databases, go for it. We would even say that the default choice should be one of the modern proxies.

Even if you are not going to use any of the features they provide straight away, it might still be a good idea to use them. You never know when you might benefit from the flexibility provided by them. This is especially true as the majority of features are intended to help with typical production issues: block a query, rewrite it, redirect a given query to a particular host, etc.. It is quite likely that you will end up running into problems where access to such a feature would help you significantly.

6.2. Which SQL-Aware Proxy Should Be Chosen?

Unless we are talking about some very specific cases, the choice is mostly between ProxySQL and MaxScale.

Feature-wise, those proxies are very similar and for most of the cases any of them will work just fine. There are a couple of differences between them and we will go over them here.

One important difference is the licensing model. ProxySQL is released on GPLv3. MaxScale, on the other hand, is released on a Business Source License. ProxySQL is free to use, MaxScale can be used freely in a non-production environment. In production, only up to two nodes can be used. Using it on more nodes would require a commercial license from MariaDB Corporation.

One can argue that ProxySQL is more flexible in terms of how it can be used. The user has more control over the query routing, sending traffic even to a particular host. The user can develop more ways to shard a database, other than using schemas. For example, one can use regular expressions to match queries and route them to a correct shard. Significant advantage of ProxySQL is the support for clustering - in large, complex environments, the ability to synchronize the configuration across proxy nodes can significantly reduce overhead of managing a proxy layer.

ProxySQL, as of now, works with all MySQL flavors - Oracle's MySQL, MariaDB or Percona Server. This may change in the future, but for now everything is supported. On the other hand, MaxScale favors MariaDB. There are some features, an example can be failover handling added to MaxScale 2.2, which simply do not work if the backends use Oracle MySQL or Percona Server (since these use a different implementation of Global Transaction ID as compared to MariaDB).

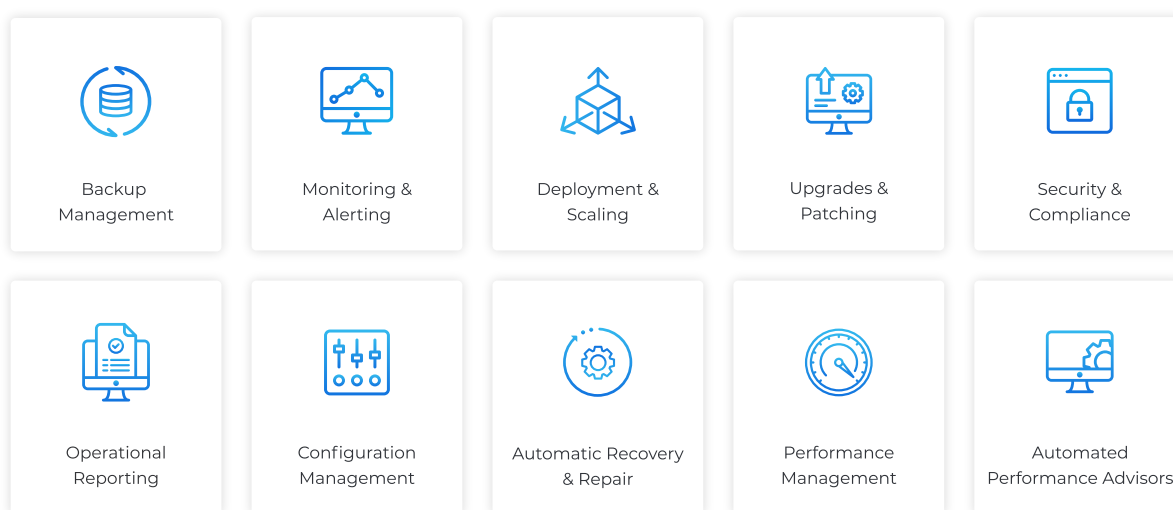
On the other hand, MaxScale is simpler to use. It might be that some features are limited or not as configurable as with ProxySQL, but it is easier to use.

ProxySQL can sometimes be overwhelming with all the configuration options and settings it comes with, while with MaxScale, it is easier to use a feature. Also, MaxScale has some nice options which do not have their equivalent in ProxySQL. For example, a binary log server. MaxScale can be used as a binary log server, helping to reduce the load on a master. As mentioned above, MaxScale can also be used to manage replication topology and perform failovers (as long as the backend databases are MariaDB).

All in all, the choice is up to the user of course. If you use MariaDB, it may make sense to go with MaxScale, as long as you are ok to pay for a commercial license. Otherwise, ProxySQL might be a good option to choose.

About ClusterControl

ClusterControl is the all-inclusive open source database management system for users with mixed environments that removes the need for multiple management tools. ClusterControl provides advanced deployment, management, monitoring, and scaling functionality to get your MySQL, MongoDB, and PostgreSQL databases up-and-running using proven methodologies that you can depend on to work. At the core of ClusterControl is its automation functionality that lets you automate many of the database tasks you have to perform regularly like deploying new databases, adding and scaling new nodes, running backups and upgrades, and more.



About Severalnines

Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

Severalnines' products are used by developers and administrators of all skills levels to provide the full 'deploy, manage, monitor, scale' database cycle, thus freeing them from the complexity and learning curves that are typically associated with highly available database clusters. Severalnines is often called the "anti-startup" as it is entirely self-funded by its founders. The company has enabled over 12,000 deployments to date via its popular product ClusterControl. Currently counting BT, Orange, Cisco, CNRS, Technicolor, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with offices in Singapore, Japan and the United States. To see who is using Severalnines today visit:

<https://www.severalnines.com/company>

Related Resources



[Database Load Balancing for MySQL and MariaDB with ProxySQL - Tutorial](#)

ProxySQL is a lightweight yet complex protocol-aware proxy that sits between the MySQL clients and servers. It is a gate, which basically separates clients from databases, and is therefore an entry point used to access all the database servers.



[MySQL Load Balancing with HAProxy - Tutorial](#)

We have recently updated our tutorial on MySQL Load Balancing with HAProxy. Read about deployment and configuration, monitoring, ongoing maintenance, health check methods, read-write splitting, redundancy with VIP and Keepalived and more.



[How to deploy and manage HAProxy, MaxScale or ProxySQL with ClusterControl](#)

In this webinar we talk about support for proxies for MySQL HA setups in ClusterControl: how they differ and what their pros and cons are. And we show you how you can easily deploy and manage HAProxy, MaxScale and ProxySQL from ClusterControl during a live demo.



[ProxySQL: All the Severalnines Resources](#)

This blog aggregates all the great resources we have created to help you understand, deploy, and maximize the potential of the new ProxySQL load balancing technology.

This whitepaper discusses what database proxies are, what their use is and how to build a highly available and highly controllable MySQL and MariaDB database environment using modern proxies.

