ArangoDB

Whitepaper

# The new **sharding feature** in ArangoDB 2.0

In the past three months we have concentrated our development efforts for ArangoDB on the sharding feature, and as a consequence we have just released Version 2.0.0. This release is a major milestone and allows for the first time to try out things and actually use sharding. Please note that this milestone is only the first of several steps.

This document explains our design for sharding, gives a technical overview and tells you in detail, what we have achieved with this release and how our road map for the coming releases looks like. From now on we intend to release a new version every two months.

## Motivation and ultimate aim

NoSQL database management systems are known and acclaimed for their good vertical and horizontal scalability. Therefore, every self-respecting such system has to scale well in both directions. Already in earlier versions ArangoDB was running on a wide range of different machines, ranging from small Raspberry Pis to large servers with a lot of RAM, many cores, classical hard disks or solid state disks. For the horizontal scalability we had to act.

The basic idea is to combine a number of ArangoDB instances running on different physical machines into a cluster that appears to the outside world as one big and fast database. This idea offers great opportunities as well as considerable challenges.

On the plus side it is evident that one can store a much larger amount of data on multiple machines than on a single one. Furthermore, it is possible to handle a larger volume of requests and transactions if one can harness the computational and communication power of multiple machines. Finally, by adding replication and automatic failover methods into the mix, one can build a system that works reliable even when facing a limited number of failure of both servers and networks. Usually one imagines such failures to be hardware failures, but one should never underestimate the amount of failures caused by human errors.

On the negative side one encounters a lot of challenges. The software architecture is necessarily more complex than in a single instance. Keeping consistency very quickly turns out to be a problem, and it is not hard to imagine all kinds of nasty failures, starting from diverging system clocks and ranging to evil intermittent network failures. Finally there are the theoretical results like the CAP theorem which immediately implies that one will have to come up with a sensible compromise between availability and consistency, when faced with a disruption of communications.

For ArangoDB, we want to achieve the following with our variant of sharding:

1. Offer the option to scale horizontally with only the least possible changes to client code. If at all possible, a cluster of ArangoDB instances should appear to be one big and fast database.
2. Achieve the scaling effects not only for the possible data size but also for the possible read and write performance, at least under good circumstances (see below).

3. Retain the flexible feature set and multi model nature of ArangoDB, including transactional behaviour and graph facilities, as far as possible without compromising too much on performance.
4. Find a good compromise between consistency and availability in the face of network outages.
5. Achieve fault tolerance by replication and automatic failover organisation. This will of course be configurable, but it should for example be possible to set up an ArangoDB cluster that survives a single server failure or hot swap without any interruption of service.
6. Offer "zero administration" for a cluster. By this we mean that the cluster can automatically recover from a certain amount of problems and "heal itself", and that it can automatically redistribute data to balance out the load between the different database servers in the cluster. Furthermore, administrative steps like adding and removing servers and thus growing and shrinking the cluster should be as automatic as sensible. Obviously, the final decision about this will be left to the administrator.
7. Offer a powerful set of web based graphical user interfaces to administrate and monitor a running database cluster.
8. Allow for easy management of a cloud-based ArangoDB cluster.
9. Offer "distributed failover", by which we mean that in the finished implementation of sharding we will no longer have static "primary" and "secondary" DBservers, but rather have for each individual shard a "primary" and possibly more than one "secondary" instance. These instances can then be spread out over the available DBservers to achieve a better reliability and usage of resources (see below).

## General design

Note that in this section we use the present tense for features that are already implemented in the current Version 2.0 and future tense for those features that are contained in our general sharding design but will only be implemented in future versions. See the Sections "State of the Implementation" and "Roadmap" below for details.

## Processes in the cluster

In a cluster there are essentially two types of processes: “DBservers” and “coordinators”. The former actually store the data, the latter expose the database to the outside world, but have essentially no own internal state, such that the failover management can concentrate on the DBservers. The database clients talk to the coordinators exactly as they would talk to a single ArangoDB instance via the REST interface. The coordinators know about the configuration of the cluster, automatically forward the incoming requests to the right DBservers (for simple cases) and coordinate the operations on different DBservers required for a request (for more sophisticated queries and requests).

As a central highly available service to hold the cluster configuration and to synchronise reconfiguration and fail-over operations we use a set of processes called “agents” together comprising the “agency”. Currently we use an external program called etcd (see github page). It provides a hierarchical key value store with strong consistency and reliability promises. Internally, it is running a consensus protocol similar to Paxos. The etcd program actually uses a protocol named “RAFT” that aims to provide an alternative to Paxos which is easier to understand and implement. All DBservers and coordinators communicate with the agency, the clients have no direct access to the agency.

To provide fault tolerance for the actual DBserver nodes, we run pairs of servers, consisting of a primary DBserver that serves the coordinators and a secondary DBserver that replicates everything from its primary in a synchronous way. In the future we might refine this situation and offer a more flexible replication model.

Figure 1 illustrates this setup.

## Communication within and with the cluster

Note that everything in the dotted frame in Figure 1 is in the cluster and the only communication to the outside world is via the coordinators. It is possible to manipulate the setup in the agency from within the cluster, but this is left to special tools who know which places are safe to modify. The agency is drawn as a single egg since it appears as a reliable entity to the rest of the cluster.

All coordinators talk to all DBservers and vice versa, but the DBservers do not talk amongst each other, and the coordinators do not talk amongst each other.

Everybody talks to the agency, which is not only used to hold configuration data, but also to synchronise events throughout the cluster.

The information, which databases and collections exist on the cluster, is stored in the agency. A collection in the cluster consists of one or more shards, which are simply normal collections that reside on some primary DBserver and are synchronously replicated to the corresponding secondary DBserver. The knowledge about which shards comprise a cluster collection is again held in the agency.

## Queries and operations in the cluster

Therefore, the coordinators, who have to handle client requests that refer to cluster collections, have to analyse these requests, use the cluster-public knowledge in the agency, and translate the client requests into one or more requests which are sent within the cluster to the responsible primary DBserver. These DBservers do the actual work and answer back to the coordinator, who can then in turn conclude that the request is finished and answer to the client.

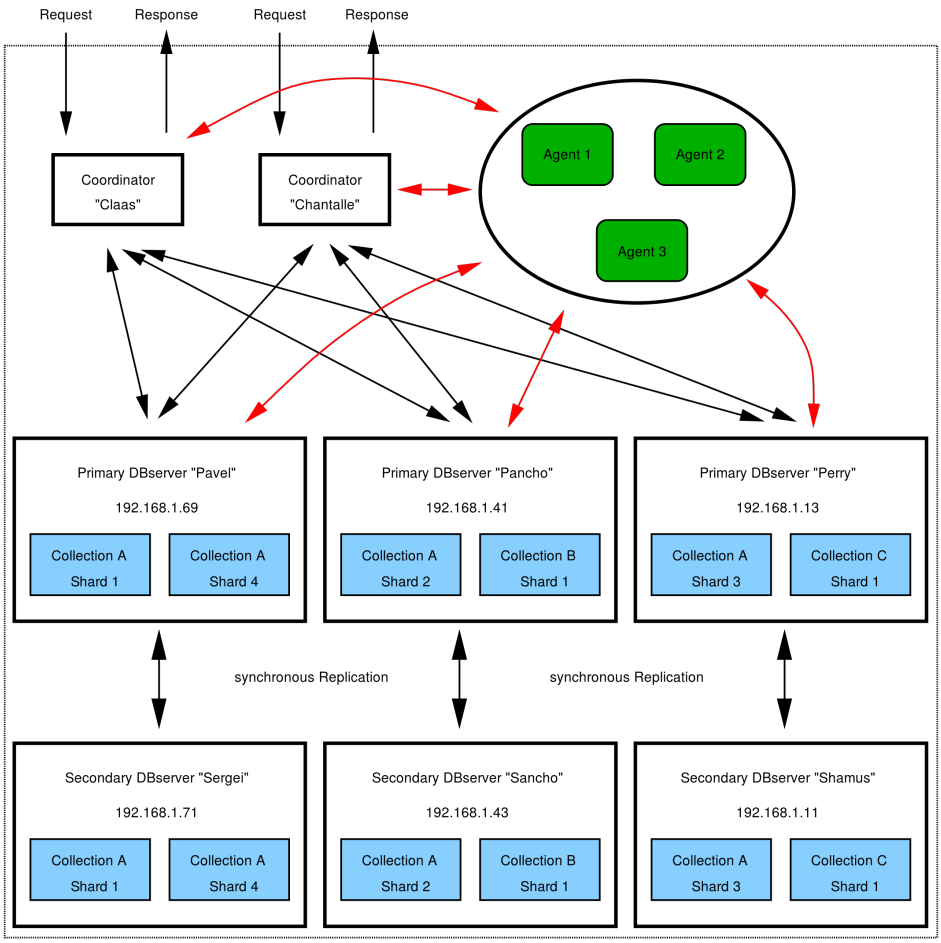


Abbildung 1: Cluster

For example creating a new document in a cluster collection involves finding out which shard will actually hold the document, then asking the DBserver who is responsible for this shard to actually create the document, and finally answering the request, when that DBserver has answered the cluster internal request.

Some queries, like returning all documents in the collection that match a certain example document, involve asking all shards, that is, all primary DBservers that are responsible for shards in the collection, and combining the returned results into a single, coherent answer.

Complex AQL queries involving multiple sharded collections and intermittent filter and sort requests, can only be coordinated by rather complex cluster-internal communication. As their name suggests, it is the task of the coordinators, to coordinate this communication and direct the responsible DBservers to do the right things in the right order.

Writing transactions are particularly challenging in this setup. Eventually we want to promise complete atomicity and isolation for them in a cluster-global way and with respect to all other operations like simple CRUD operations and simple queries. However, we do not want to introduce rather painful global locks, which would almost certainly ruin performance. Therefore we will for the near future provide a kind of “eventual atomicity” without isolation for transactions on a cluster. At this stage, the exact specifications are not finalised, but the basic idea is that for such a transaction it is guaranteed that it will either be done completely or not at all. However, the cluster reserves the right that some of the operations in the transaction are already visible to other queries running at the same time while others are not. Eventually, the whole transaction will either have been committed or have been rolled back, and from then on all other queries will see all changes or none at all. This is admittedly a sacrifice, but we deem it to be tolerable for the near future in comparison to the performance gain we can reap across the board. Rest assured, however, that operations on a single document are always completely atomic and isolated. In the final implementation we will use MVCC throughout the system and can there fore finally offer full ACID semantics for cluster-wide transactions.

This whole setup is designed such that under good circumstances, multiple requests can be executed in a parallel fashion. Obviously, if thousands of requests are received dealing with the exact same document, these can in the end not really be

parallelised. However, if for the documents with which these thousands of requests deal are distributed relatively uniformly over a big sharded collection, then the load of the actual requests will be distributed uniformly between the DBservers as well as between the different coordinators, and the whole machinery shows a good parallel performance.

## Fault tolerance and automatic failover

As mentioned above, pairs of DBservers (primary and secondary) store the actual data, that is, the individual shards. Writing operations on them are done via the primary, but are synchronously replicated on the secondary, such that the operation is only successfully completed, when both have executed it. If one of the two servers in a pair fails (or the network connection between them or to one of them), then the automatic failover procedure will detect this and decide to reconfigure the cluster to use only the healthy respectively reachable server of the two. This can mean that the primary continues to serve without its secondary, or that the secondary takes over the service and continues as a single primary.

It is rather crucial for a solid design to detect such a situation and to react on it outside the two servers involved in the pair. Therefore, we will have further diagnostic and management processes in a cluster. Note that these are not drawn in the above picture to simplify the exposition. These further processes will all communicate and synchronise using the central agency. They will all be organised in a redundant way to avoid single points of failure. The DBservers and coordinators contribute to the automatic failover protocol by sending regular heartbeats to the agency and by reacting to the directions of the failover managers.

The coordinators have to be aware of this automatic failover and retry their operations on the DBservers, should failover situations happen. Faults in coordinators are less severe since they do not hold any internal state. Therefore, the automatic failover for coordinator faults can simply be organised by using more coordinators and by letting the clients try to contact another coordinator in case of a fault of the one they had used previously. This is easy to implement on the client side and can as a positive aside be used to balance the outside communication load between the different coordinators.

## Zero administration

Similar to the diagnostic and failover management processes there will be other processes that watch over the distribution of data and possibly network traffic across the cluster. They can then initiate an automatic redistribution of shards between the different pairs of DBservers. This will for example be necessary, when the system administrator has added one or more new pairs of DBservers to react to a shortage of storage. They will of course start out empty, and it is desirable that the cluster automatically redistributes data to even out the load.

Obviously, this automatic redistribution feature needs to be controllable by the system administrator to allow for shrinking of a cluster by first emptying some DBserver pairs and then actually removing these empty ones.

## (Graphical) administration tools

All cluster features come accompanied with convenient, web-based administration tools to simplify the life of the system administrators that are responsible for a cluster. Already in Version 2.0 we offer a convenient way to plan, setup and launch a cluster, provided you have a set of machines on which ArangoDB Version 2.0 is installed, as well as powerful monitoring tools to get an overview over the state of a cluster.

In future versions we will allow for easy administration of clusters by extending these graphical tools to all aspects of the maintenance of a cluster.

The usual web front end of ArangoDB runs on the coordinators as well and gives access to the cluster database in exactly the same ways as it does this usually for a single ArangoDB instance.

## Distributed failover

In the completely finished implementation of the fault tolerance and automatic failover we will break down the rather rigid distinction between “primary” and “secondary” DBservers. In that stage of extension all DBservers will be equal. The replication setup will instead have one “primary” instance and possibly many “secondary” instances of each logical shard. These various instances of the shards are distributed over the DBservers in the cluster in a way that the failure of one (or possibly more) DBserver(s) does not lead to a loss of all instances of the logical shard. Furthermore, with this additional flexibility we will able to arrange the

distribution of primary and secondary instances amongst the DBservers, such that the load is better distributed between all machines. An example setup is illustrated in Figure 2.

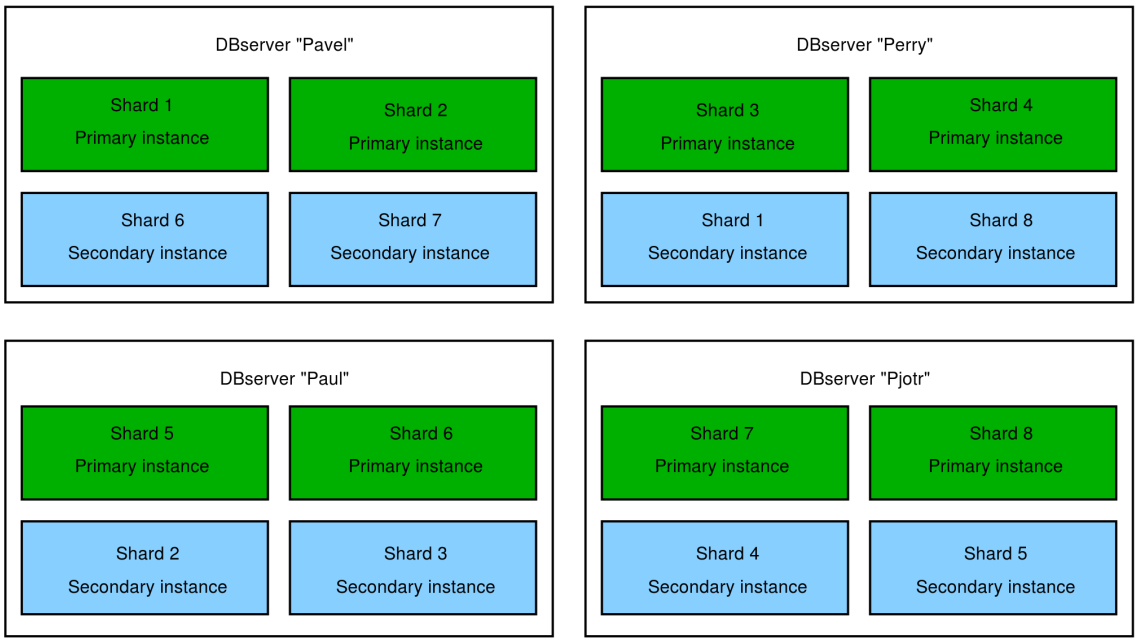


Abbildung 2: Distributed Failover

In the case that some DBservers fail or can no longer be reached, the automatic failover management will declare one of the secondary instances to be the new primary one and resume service seamlessly. The system administrator will be informed and can take action to restore the faulty DBservers. Once this is done, the shard instances on these DBservers reappear, automatically synchronise themselves to the latest stage and the full replication setup is achieved again.

In the example above one can see that if all is well, all machines will get a similar load, because every one is responsible for two primary shards. Even if one DBserver fails, the secondaries for its primaries are distributed among the other DBservers in a way, such that the new load is distributed relatively evenly between the surviving machines. If for example in this picture the server “Pavel” fails, the two servers “Perry” and “Paul” will be responsible for one more shard each, but no server has its load doubled.

In comparison to the first version of fault tolerance with the fixed primary and secondary DBservers we will gain a higher resilience against multiple failures as well as a better usage of resources, since all DBservers will be able to work actively in the usual healthy situation, because they will hold some primary and some secondary shard instances. The balancing out of memory usage and communication bandwidth will be more flexible in the final setup, too. Furthermore, performance optimisations

like returning when a quorum of instances has completed a write operation can be implemented as well.

## Scope of the final Implementation

Obviously, any database design cannot cover all possible use cases. We are fully aware of this and therefore there are some areas in the world of NoSQL databases into which we do not even plan to reach.

We will intentionally not aim for the massively parallel world of map-reduce engines and key-value stores, simply because other, more specialised solutions are covering this segment perfectly well, and because we do not want to restrict our feature set to be able to scale to these dimensions.

Furthermore, at this stage we do not intend to implement cluster-global indices, but rather use indices that are local to each shard. This is a great simplification and comes with rather small sacrifices in functionality. For example, the capabilities for unique constraints are and will be somewhat limited in sharded collections, simply we can only efficiently support unique constraints on the keys that are used for the distribution of documents to shards.

## State of the Implementation

Version 2.0 of ArangoDB contains the first usable implementation of the sharding extensions. However, not all planned features are included in this release. In particular, automatic fail-over is fully prepared in the architecture but is not yet implemented. If you use Version 2.0 in cluster mode in a production system, you have to organise failure recovery manually. This is why, at this stage with Version 2.0, we do not yet recommend to use the cluster mode in production systems. If you really need this feature now, please contact us.

In normal single instance mode, ArangoDB works as usual with essentially the same performance and func- tionality as in previous releases (see here for a list of new features in ArangoDB 2.0 and here for hints about upgrading to Version 2.0).

In cluster mode, the following things are implemented in Version 2.0 and work:

- All basic CRUD operations for single documents and edges work essentially with good performance.

- One can use sharded collections and can configure the number of shards for each such collection individually. In particular, one can have fully sharded collections as well as cluster-wide available collections with only a single shard. After creation, these differences are transparent to the client.
- Creating and dropping cluster-wide databases works.
- Creating, dropping and modifying cluster-wide collections, all work. Since these operations occur seldom, we will only improve their performance in a future release, when we will have our own implementation of the agency as well as a cluster-wide event managing system (see road map for release 2.3).
- Sharding in a collection can be configured to use hashing on arbitrary properties of the documents in the collection.
- Creating and dropping indices on sharded collections works. Please note that an index on a sharded collection is not a global index but only leads to a local index of the same type on each shard.
- All SimpleQueries work. Again, we will improve the performance in future releases, when we revisit the AQL query optimiser (see road map for release 2.2).
- AQL queries work, but there is a chance that the performance is bad for some queries. Also, if the result of a query on a sharded collection is large, this can lead to an out of memory situation on the coordinator handling the request. We will improve this situation when we revisit the AQL query optimiser (see road map for release 2.2).
- Authentication on the cluster works with the method known from single ArangoDB instances on the coordinators. A new cluster-internal authorisation scheme has been created. See here for hints on a sensible firewall and authorisation setup.
- Most standard API calls of the REST interface work on the cluster as usual, with a few exceptions, which do no longer make sense on a cluster or are harder to implement. See here for details.

This page lists in more detail the planned features of sharding that are not yet implemented and the few little features of the standard ArangoDB API that we will probably never support in cluster mode.

## Roadmap

For Version 2.1 (planned for the end of May 2014) we will concentrate on some internal changes that are needed for the more sophisticated sharding features, in particular for automatic failover. More concretely, we want to implement write-ahead-log functionality across all collections in a single ArangoDB instance. This will allow us to simplify a lot of code dealing with transactions, replication and is essential for the synchronous replication between primary and secondary DBservers.

For Version 2.2 (planned for the end of July 2014) we will improve the AQL optimiser, in particular with respect to AQL queries running in a coordinator in a cluster. This will speed up AQL queries in clusters dramatically.

For Version 2.3 (planned for the end of September 2014) we will implement fault tolerance as described above including automatic failover between primary and secondary DBservers as well as our zero administration features including self-healing clusters, automatic balancing out the data distribution within the cluster and convenient growing and shrinking tools.

Stay tuned for exciting new releases of ArangoDB later this year.