Rebalancer

The Rebalancer is an automated system for maintaining a healthy distribution of data in the cluster. It's the Rebalancer's job to respond to an "unhealthy" cluster by modifying the distribution and placement of data. The Rebalancer is an online process that effects changes to the cluster with minimal interruption to user operations. It relieves the database administrator from the burden of manually manipulating data placement.

The ClustrixDB Rebalancer was designed to run automatically as a background process to rebalance data across the cluster. The following sections describe how the Rebalancer works. The default values for distribution and replicas are sufficient for most deployments and typically do not require changing.

The ClustrixDB Rebalancer has been awarded two patents for distributing and slicing data.

See also:

- Managing Data Distribution
- Managing the Rebalancer

A Healthy Cluster

In ClustrixDB, user tables are vertically partitioned in representations, which are horizontally partitioned into slices. When a new representation is created, the system tries to determine distribution and placement of the data such that:

- The representation has an appropriate number of slices.
- The representation has an appropriate distribution key, to fairly balance rows across its replicas, but still allow fast queries to specific replicas.
- Replicas are well distributed around the cluster on storage devices that are not overfull.
- Replicas are distributed across zones (if configured).
- Replicas are not placed on decommissioned nodes.
- Reads from each representation are balanced across the representation's nodes.

Over time, representations can lose these properties as their data changes or cluster membership changes. This section describes the various situations that the Rebalancer is able to remedy.

Under-protection

By default, Clustrix keeps two copies (replicas) of every slice. If an unexpected node failure makes one of the replicas unavailable, the slice will still be accessible through the remaining replica. When only one replica of a slice exists, the data on that slice is vulnerable to being lost in the event of an additional failure. The number of replicas per slice can be specified via the global variable MAX_FAILURES.

When a slice has fewer replicas than desired, the Rebalancer will create a new copy an existing replica on a different node. The most common reason for this is if a node fails or otherwise becomes unavailable. Initially, the cluster will create Recovery Queues for that node's replicas so that they can be made up-to-date when that node returns to quorum. However, if the node is unavailable for an extended period of time, the Rebalancer will begin making copies of replicas from elsewhere in the cluster and will retire the Recovery Queues.

If a node becomes permanently unavailable, the cluster has reduced storage capacity. If there is not enough remaining storage capacity to make new replicas, the Rebalancer will not be able to do so and the slices will remain under-protected. The cluster does not automatically reserve capacity for re-protecting slices.

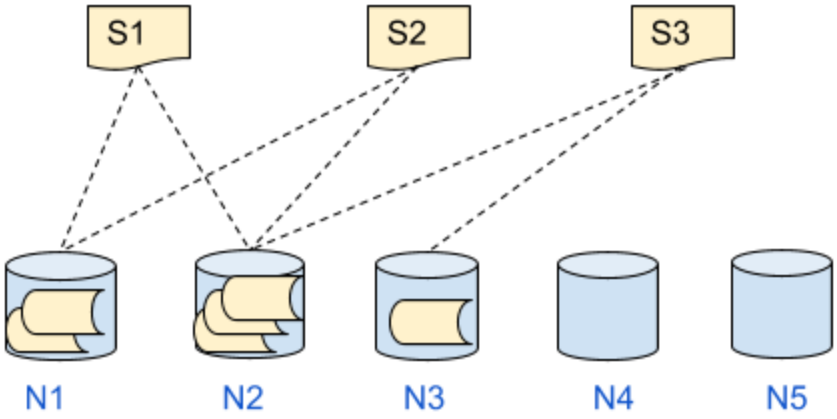
Load Imbalance

If the slices of a representation are not well-distributed across the cluster, the Rebalancer will try to move them to more optimal locations.

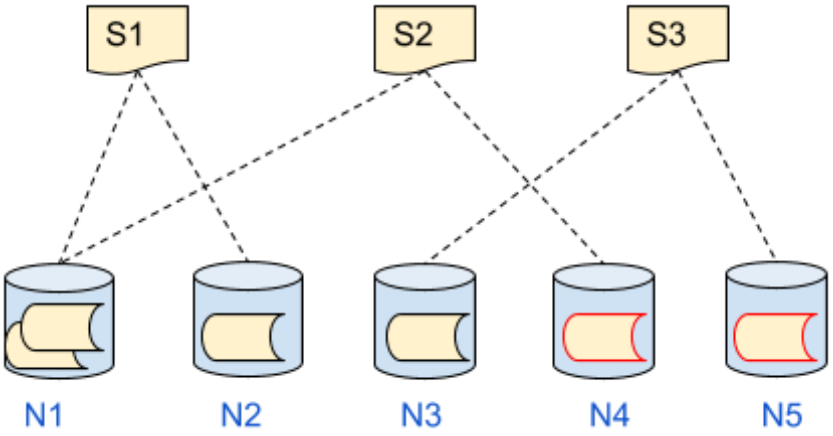
The Rebalancer evaluates the placement of each representation independently, in the following manner:

- Each slice of a representation is assumed to exert load proportional to its share of the representation's key-space. For example, if the index size of a slice constitutes 10% of the overall representation's index space, then it will also be assumed that slice will comprise 10% of the representation's load, as well. The Rebalancer considers that anticipated activity level when placing a given replica of a slice.
- The representation is well-distributed when the difference between the "most loaded" and "least loaded" nodes is minimal.

Consider the following examples of a representation with three equal-size slices, S1, S2, and S3. Each slice has two replicas distributed across a five-node cluster.



This is an example of a poor distribution of this representation. Each slice is protected against the failure of a node, but the majority of the representation is stored on node 2.



This is an example of a good distribution. The replicas outlined in red were relocated by the Rebalancer to improve cluster balance. Although node 1 has one more replica than the other nodes, there is no node that is under-loaded.

When a Node is Too Full

If a node in the cluster is holding more than its share of table data, the Rebalancer will try to move replicas from that node to a less utilized node.

Before moving any replicas, the Rebalancer computes the *load imbalance* of the cluster's storage devices. If this imbalance is below a configurable threshold, the Rebalancer will leave things alone. This is to prevent the Rebalancer from making small, unnecessary replica moves.

Balancing storage utilization is a second priority to maintaining representation distribution. In some situations, this may result in less optimal storage device utilization, in exchange for better representation distribution.

When a Slice is Too Big

Representations are partitioned into slices, each of which is assigned a portion of the representation's rows. If a slice becomes large, the Rebalancer will *sp lit* the slice into several new slices and distribute the original slice's rows among them.The larger a slice becomes, the more expensive it is to move or copy it across the system. The maximum desired slice size is configurable, but by default, the Rebalancer will split slices utilizing greater than 1 GiB. (Utilized space will be slightly larger than the size of user data because of storage overhead).

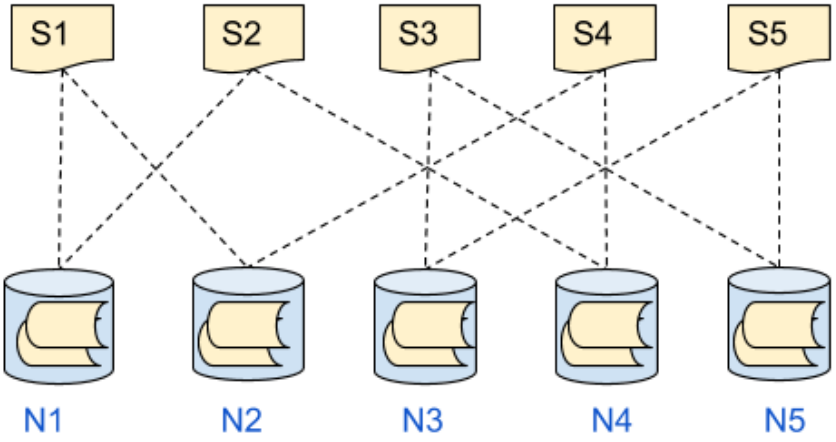
Too many slices can also be a problem: more slices means more metadata for the cluster to manage, which can make queries and group changes take longer. The Rebalancer will not reverse a split, so the usual recommendation is to err on the side of fewer slices and allow the Rebalancer to split if there is a question about how many slices a representation needs. Splits can be manually reversed with an ALTER statement to change the slicing of a representation.

Because rows are hash-distributed among slices, if a slice approaching the split threshold, it is likely that the other slices of the representation will also need to be split.

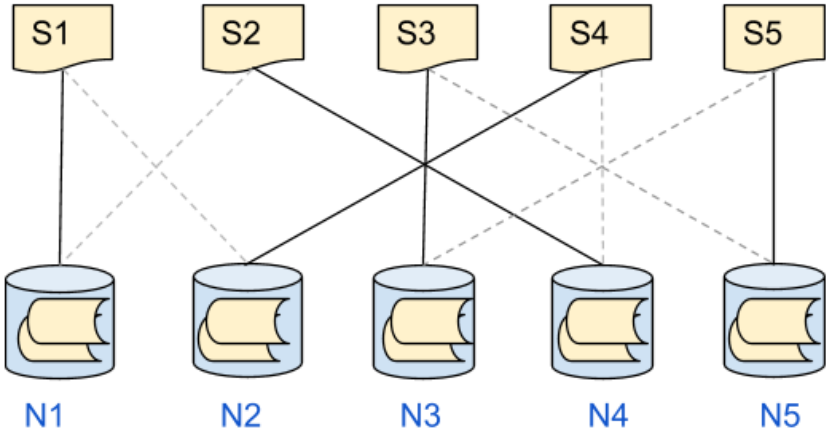
The global rebalancer_split_threshold_mb determines when a slice needs splitting. That global may be overridden for a per table or per index basis via DDL. See Slices.

Read Imbalance

ClustrixDB reads exclusively from only one replica of each slice, and that slice is designated as the *ranking replica*. This allows the Rebalancer to better manage data distribution and load for both write operations, which are applied simultaneously to all replicas, and read operations, which consistently use only the ranking replica.



From a write perspective, this five-slice representation is well distributed across a five node cluster. Each node is doing an even share of the work.



For read operations, ClustrixDB designates one replica as the ranking replica for the slice and always reads from that replica to balance load across your cluster.

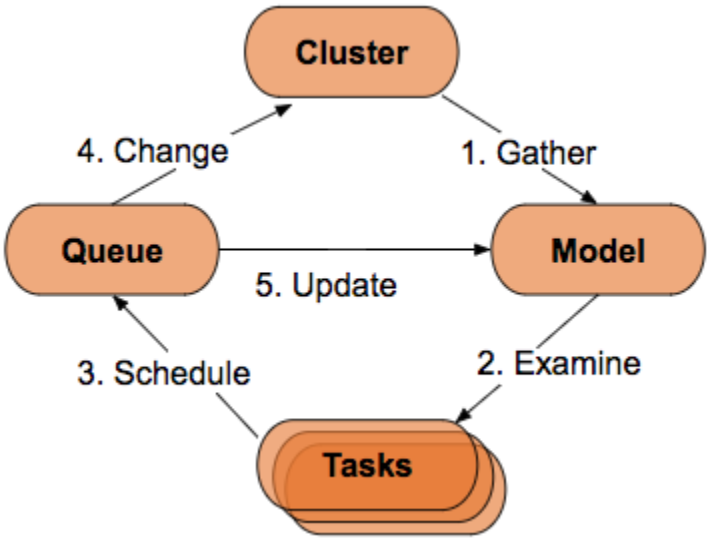
Since all replicas of a slice are identical, directing reads to a non-ranking replica would produce the same results. However, to better utilize each node's memory, ClustrixDB consistently reads from the designated ranking replica. If a ranking replica becomes unavailable, another replica takes its place.

Decommissioned Nodes

When a node is to be removed from the cluster, the administrator can designate it as *soft-failed*. This directs the Rebalancer to not place new replicas on that node, nor will it be considered when assessing storage imbalance. The Rebalancer will begin making additional copies of replicas on the softfailed node and place them on other nodes. Once there are sufficient replicas, the softfailed node can be removed from the cluster with loss of data protection.

Rebalancer Components

The Rebalancer is a system of several components:



1. Information is gathered to build a Model of the cluster's state
2. Tasks examine the Model and decide if action is necessary
3. Tasks post operations to the Queue that will schedule them
4. When an operation graduates from the Queue, it is applied to the cluster
5. When an operation completes, the Model is updated to reflect the new state

Distribution Model

Metadata about the existence and location of every replica is duplicated on each node, but the size of each replica is only known locally on the node where it lives.

The Rebalancer periodically polls all of the nodes in the cluster to create a model of the current cluster state. Between polls, representations may be created or destroyed by users and may grow or shrink in size. The Rebalancer's model of the cluster is always somewhat out of date. This means that it can sometimes make suboptimal decisions.

Tasks

A periodic task monitors each Rebalancer task and corrective action is scheduled independently, as needed. Each task shares the Rebalancer's model of the cluster's state and is aware of other Rebalancer tasks that are queued. Some parameters of these tasks, including their rate, can be adjusted by the administrator.

Summary of Rebalancer Tasks

Name	Fixes	Priority	Rate
Reprotect	Missing replicas	High	Aggressive
Zone Balance	Slice imbalance for a zone	High	Aggressive
Softfail	Slices on decommissioned hardware	High	Moderate
Reap	Extra replicas/queues	High	Moderate
Split	Large slices	Medium	Moderate
Rerank	Node/zone read imbalance	Low	Conservative
Rerank Distribution	Representation read imbalance	Low	Conservative
Rebalance	Node/zone usage imbalance	Low	Conservative
Rebalance Distribution	Representation write imbalance	Low	Conservative

Although each task is independent, they are tuned so that the decisions of one task will not conflict with the decisions of another. For example, when a slice is split, the new slices are placed with consideration to the rest of the representation's slices and the capacity of the cluster's storage. The new replicas for these slices will be ranked correctly so that the rerank process need not make adjustments later.

Rebalancer Queue

All changes effected by the Rebalancer are scheduled using a priority queue. This queue is designed to rate-limit the Rebalancer so that it does not oversubscribe the cluster with changes, including that:

- One change can be applied to a slice at a time.
- Some operations, like redistributing representations, limit the number of simultaneous executions.
- A limited number of operations can affect an individual node at one time.

Operations are assigned priorities and a queued operation can be surpassed by a higher priority one. Once an operation begins, it is not interrupted, even if there is a higher-priority operation waiting in the queue.

Rebalancer operations can fail for many reasons. If an operation fails, any change made to the cluster is reverted and the Rebalancer may subsequently choose to retry the operation. Because the state of the cluster can change at any time (while the Rebalancer is making decisions or while operations are being performed) some failures are expected even in normal operations.

The Rebalancer keeps no memory of operations after they have completed, so operations that repeatedly fail may be repeatedly retried. The Rebalancer assumes that any condition that causes an operation to fail is transient.

Recovery Queues and Versioned Metadata

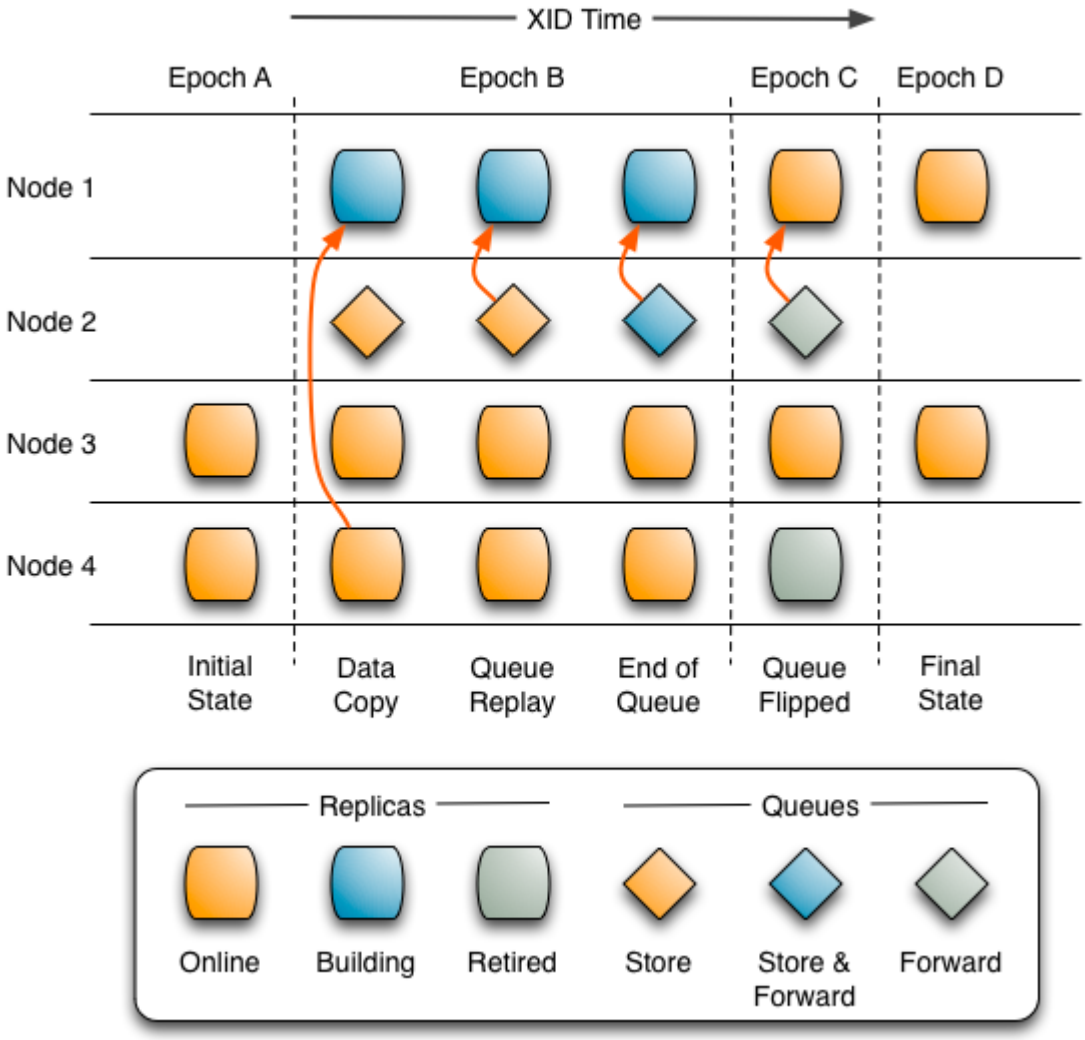
Changes to representation metadata, including replica location, are versioned through a Multi-Version Concurrency-Control (MVCC) scheme similar to that used for table data. When a representation is changed, transactions that started before the change will never observe it. Older transactions are not canceled, and newer transactions do not wait for them. Each set simply sees a different view of the metadata. This allows metadata changes to be made without coordination with currently running transactions.

Metadata changes are transactional, so if an error is encountered while effecting the change, the entire change rolls back to the state in which it began.

When the Rebalancer moves a replica between two nodes, it does so through a specific series of *DDL* changes. Between each change is an epoch during which user transactions may start. The Rebalancer is able to perform the replica move *online*, with limited disruption to new or running transactions as the operation proceeds. The key to online operations is the Recovery Queue, a log of changes missed by a replica while it was being built. (Recovery Queues are different from the Rebalancer's priority queue of operations.) The following is an example of an online replica move using a Recovery Queue. Other Rebalancer operations are more complex but proceed similarly.

Replica Move Example

Reading from left to right, this image shows the various steps needed to affect a replica move.



Initial State - Epoch A

Initially, the slice's replicas reside on node 3 and on node 4. We are moving the slice from node 4 to node 1, where no replica for this slice previously existed.

Create A New Replica - Epoch B

The first step is to create a new, empty replica on node 1. This replica is marked as *building* in the system so that queries won't access it. A Recovery Queue is simultaneously created on node 2 for this replica. That Queue looks like another replica for queries that write to the slice, but it functions differently: it logs any writes so they can be replayed later against the new replica. This allows the copy from node 4 to node 1 to proceed without blocking updates to the slice. The changes for the new replica will be stored in the Recovery Queue.

Data Copy

A serializable read from the original replica returns all of the rows from the point when the Queue was made. These rows are sent to the new, building replica. Meanwhile, any updates to the original replica will be logged to the Queue.

There may be transactions started in epoch A before the Queue was created that try to modify the slice once they are committed. These transactions won't know about the Queue and therefore won't log their updates to it. If these transactions' updates are missed by the serializable copy, their updates would be lost when the original replica is dropped and the new replica is brought online. Therefore, the system will fail any such transaction. To minimize occurrences of this situation, the serializable copy is delayed for a period (about 1 second) to allow in-process transactions to complete.

Queue Replay

When the copy is complete the new replica mirrors the original at the time the copy began. Then the contents of the Queue are read and applied to the new replica.

While queue-replay is progressing updates to the slice continue being logged to the Recovery Queue. Queue-replay is tuned to be faster than the rate at which new entries can be added to the Queue to ensure that Queue-replay finishes.

End of Queue

When the last record in the Queue is read, the Queue changes from asynchronous to synchronous operation. Updates continue to be stored in the Queue and are also applied to the new replica before the transaction is allowed to commit.

Queue Flipped - Epoch C

When the end of the Recovery Queue is reached, the original replica is retired and the new replica is made online.

The Queue is no longer updated by new transactions. Older transactions from epoch B continue to write to it, and the Queue remains in synchronous mode, forwarding updates to the new replica, now online.

The retired replica continues to receive updates but is never read by new transactions. Transactions in epoch B (from before the flip) still see it and must see the most recent updates to the slice. After the flip, the Queue no longer stores update records since queue-replay is finished.

Retired

Once all old transactions from epoch B have finished, the Recovery Queue will not receive any more writes. The Queue and retiring replica are removed and only the new replica remains.

Final State - Epoch D

Upon completion, the replica's slices now reside on node 1 and 3.