Token/vNode Allocation algorithm

CASSANDRA-7032

The problem

In its simplest form the token allocation problem is a question of finding a distribution of tokens for a cluster such that we have optimal splitting of the load[1] between nodes. This simple question can be solved pretty easily using a single token per node and dividing the space into equally-sized tokens.

Once you take into account scalability, however, being able to adapt to increasing and decreasing loads by adding and removing nodes, this solution becomes severely lacking. As a simple example, in a single-token RF1 optimally distributed cluster the best thing one can achieve by adding a new node is to reduce the load of a single existing node by half, moving the rest to the new node. This creates a serious imbalance and we would want to do much better.

The classic solution that has been used in Cassandra to avoid this is the use of randomly-allocated tokens. Randomness starts imbalanced, but it is not affected by whether we created the cluster in one go or by adding node by node, and the imbalance is normally better than what happens after adding or removing a couple of nodes in a perfectly balanced cluster. To improve things further one can use multiple tokens per node (vNodes) which by virtue of averaging the allocations for more than one randomly chosen token achieves a much smoother spread of the token share per node.

However, a problem with randomness is that it is guaranteed to produce excesses as the number of nodes grows. A typical simulations of token load spread after random allocation would show something similar to:

node count	vn	min node load pct compared to optimum	max node load pct compared to optimum
20	1	-69%	122%
20	8	-39%	45%
80	8	-78%	162%

[1] This document assumes load is the token space share served by each node. It takes into account replication, but to make the problem tractable it assumes good partition hashing and even load per partition served. It cannot do anything about hot partitions and any hashing anomalies that do not spread the partitions evenly over the hash space.

80	64	-19%	25%
80	256	-9%	9%
160	256	-9%	10%
320	256	-14%	8%
640	256	-11%	11%
1280	256	-12%	12%
2560	256	-13%	13%
5120	256	-13%	13%

Cassandra-7032 was an attempt to reduce these imbalances, or at the very least to achieve a state where the imbalance **does not grow** as the number of nodes increases.

The problem can be further split into two separate parts:

- Adding nodes to an existing cluster in a way that improves the distribution of load.
- Creating new clusters with close to optimal distribution of load, which does not deteriorate as new nodes are added.

The real complexity of the problem becomes apparent only when replication and cluster topology are factored in.

Decommission was not taken into account directly at this time (see further work).

The 7032 solution

We focussed on adding nodes as a more pressing problem, and that turned out to be a good improvement on creating clusters as well.

Below we use vn as the number of vnodes being allocated with the new node.

The RF=1 case (also used when RF matches rack count)

An algorithm to allocate tokens for a new node in the case of no replication is not hard to imagine. Taking into account the fact that we can only take away load and aiming to decrease overuse as much as possible, we can take the most overused nodes and try to spread their load between them and the newcomer evenly.

We can at most affect as many nodes as the number of vnodes for the new node (vn). Sometimes the initial disbalance is heavy enough that spreading the load between the top

vn nodes plus newcomer would require increasing the ownership of some of them, so we remove the smaller ones until we have n nodes whose individual ownership is larger than the sum of ownership of the n nodes divided by n+1.

We then select tokens from the largest ranges belonging to the nodes in the list so that each node is left with equal share, which also ensures that the newcomer gets the same share.

The method is also explained in this comment.

In terms of reducing overuse as much as possible in a single step, this is the optimal solution. The previously most overused and the newcomer are all now at the same ownership and are the most heavily overused nodes in the cluster. If we could take away more, either the newcomer or some other overused node would have a higher share.

Since no replication is not a mode we prefer to use, this method is not sufficient and the algorithm was not what we initially settled on. It was committed later in Cassandra-12777 specifically to handle applications of the RF=1 case, and later turned out to have wider application in the common pattern of defining as many racks in a datacenter as the replication factor.

In terms of building clusters that can gracefully scale, however, this is not the optimal solution as it tends to create clusters with perfect distribution, which then can't be properly scaled by one. To address this problem, DB-1552 improved the algorithm to create vnode ownership of varying size using a simple heuristic:

- Give the tokens being allocated a range of sizes.
- Tighten the range as the number of nodes in the ring increases.

This change significantly reduced the variability of ownership.

Looking at the vn=1 case without replication theoretically, the best overuse with graceful scaling we can hope for[2] is to reach a state where each token owns t/i of the token space, where i ranges from n to $2n$-1 and for some target maximum ratio t over the average. If this is the case, when we split the token with currently highest ownership the next highest is already at the target ownership for n+1. The sum of these targets when n increases has a limit of t.ln 2, and thus the smallest t we can hope to attain is 1/ln 2. This results in an overuse of 1/ln 2 - 1 ≈ 44.3%. As listed in the ticket above, the latest algorithm achieves 48.3%, which results in largest node ownership within 3% of optimal.

Replication

When we have replication, it is no longer possible to control the resulting shares for the affected nodes this finely. While in the RF=1 case placing a token directly divides the

[2] See this file for a test of this idea.

responsibility between the existing owner and the newcomer, this is not the case for other replication factors.

Imagine an RF=3 cluster with the simple replication strategy, and existing tokens A, B, C, D, E and F, with different nodes owning tokens for the different positions. The node owning D takes replicas for all partitions in A-D (i.e. has *replicated ownership* over the A-D range; we will use *owns* from here on), the owner of E owns B-E and F's owns C-F. The replicated ownership is a single continuous span between owned tokens and *rf*-many tokens before them.

Placing a new token for a new node at some position X between C and D has the following effects:

- The node owning D loses the range A-B
- The node owning E loses the range B-C
- The node owning F loses the range C-X
- The newcomer gains the sum of these three, A-X

Most of the effect of the addition is not affected by the precise positioning of X: the new node always gets A-C, and the owners of D and E always lose one of the ranges. The pinpoint subtractions we could do in the non-replicated case are no longer available, and every change has collateral effects that are hard to control.

To complicate things further, the same node can own multiple tokens within the replication span, which are skipped when choosing where to replicate data. This translates into having to skip tokens from nodes already met when counting *rf* tokens backwards, as well as a *barrier* for the replication when a token from the same node is seen. In the example above, if the sequence is A, B, C_1, D_1, D_2, C_2, E (where X_1 etc. belong to the node X) the replicated ownership for the token D_1 is A-D_1, but D_2 only serves D_1-D_2 and C_2 serves C_1-C_2, while E takes all of B-E.

Inserting X between D_1 and D_2 does not change the ownership of D_2 or C_2 at all and only takes B-X from the range served by E.

Outline of the algorithm

The above appeared too complicated to track to be able to directly pick a suitable position for the new token as we could do in the non-replicated case.

Instead, we went for an approach which would pick some candidate token positions and examine the effect of choosing each of them. This would let us rank the candidates, pick the best and repeat until we have finished the allocation for the new node.

The ranking method has to improve the metric we care about, the highest overuse factor in the cluster, but it should do so without introducing other unwanted skew, e.g. higher node

underuse. A measure that makes sense for this is the standard deviation across the loads of each node.

After a lot of experimentation we found that the longer term results are better if we also take into account the standard deviation across the load associated with individual tokens, which effectively prevents them from becoming too big or too small.

The candidates we use are the simplest possible: the midpoints between existing tokens. As discussed above, the exact choice for the position of the token does not matter that much, as most of the effect is accomplished by just choosing the range to split. This was supported by experiments and simulation of versions of the algorithm which allowed a wider choice of candidates (e.g. making available 5 tokens per range and letting the algorithm choose any of them), which did not show a tangible improvement.

The final algorithm looks like this:

- Get the midpoints between all existing tokens in the ring and put them in a candidates list.
- Rank the candidates according to the decrease in *variance* (=$stddev^2$) *across node load + variance across token load* picking them would entail.
- Choose the top, rerank and repeat.

We use some extra preparation and heuristics to make the process fast.

See also the blog post on the algorithm.

Complex topologies

Topologies can also define datacentres (separate replication domains) and racks (groups of nodes where only one replica of the content should reside). These should be taken into account.

Datacentres are easy to deal with, allocation and ownership is not really affected by the existence of other datacentres. From the point of view of nodes in the datacentre, we have a local replication factor and finding a place for a partition will skip remote nodes until it reaches enough of the local ones. Thus token allocation only needs to know about the local nodes and topology.

The situation is similar in the case where exactly RF racks are defined in the local datacentre. Because in this case every piece of data has to be replicated exactly once in each rack, we can treat racks as separate allocation spaces with RF 1. Since DB-1253 the algorithm will identify this case and apply the RF 1 algorithm, ignoring all nodes in other datacentres and racks.

To take racks into account in the case where we have more racks than the replication factor defined in the datacentre, there is only a small modification that needs to be done to

the algorithm: instead of treating belonging to the same node as the barrier and skip criterion when counting *rf* tokens backwards, we use belonging to the same rack. We can ignore the node boundaries completely as they do not play a part in the effective replicated ownership.

If only one rack is defined in the datacentre, this has the same effect as all nodes being on a separate rack and is treated the same way. To reflect the fact that we can use racks or nodes as the equivalence boundary, the code uses the term *replication group* to specify the multitude of vnodes which cannot replicate the same data.

If more than one, but fewer than *rf* racks are defined, the algorithm cannot be used as the replication structure becomes too complicated.

Heterogeneous clusters

While commonly used in this way, it is not necessary for the vnode count to be uniform among nodes in the cluster. Particularly, with random allocation we would naturally get the ownership of a node to normally be proportional to the number of vnodes we have given it. This can be used to make better use of heterogeneous clusters where some nodes are more powerful than others.

This also works with the allocation algorithm: it takes the given vnode count as an indication of the desired size of the node, and takes it into account by choosing ownership targets that are proportional to the vnode count.

Substructure

The algorithm is not restricted to treating nodes as the only unit of allocation whose balance should be optimized. If we want, for example, to spread the load that a single disk takes evenly between all disks in all nodes in the datacentre, we can do that by allocating a number of tokens for each disk. If we do that, a node naturally becomes a sort of a rack for its contained disks (but if racks are defined, we can ignore this boundary as the replication group will contain all disks in the rack).

To reflect the fact that the entities we want to optimize do not have to be nodes, the code uses the term *unit* instead.

Results achieved in simulation

Below are the maximum spreads (rounded up) of the node load in a 10 to 1000-node cluster generated using this algorithm depending on replication factor and the number of tokens per node:

rf	*vn*	min node load pct compared to optimum	max node load pct compared to optimum
1 (6.7.1 or later)	1	-33%	48%
	2	-20%	24%
	4	-11%	12%
	8	-7%	6%
	16	-9%	3%
	32	-6%	2%
	64	-6%	1%
	128	-4%	<1%
	256	-3%	<1%
1 (before 6.7.1)	1	-50%	100%
	2	-22%	32%
	4	-13%	17%
	8	-9%	9%
	16	-13%	6%
	32	-9%	3%
	64	-6%	2%
	128	-5%	1%
	256	-5%	<1%
2	1	-42%	52%
	2	-32%	31%
	4	-19%	17%
	8	-16%	9%
	16	-12%	5%
	32	-9%	3%

	64	-7%	2%
	128	-5%	1%
	256	-3%	1%
3	1	-30%	37%
	2	-21%	24%
	4	-17%	14%
	8	-12%	7%
	16	-8%	4%
	32	-6%	2%
	64	-4%	1%
	128	-2%	1%
4	1	-28%	29%
	2	-21%	21%
	4	-14%	12%
	8	-9%	7%
	16	-7%	4%
	32	-5%	2%
	64	-2%	1%
	128	-1%	1%
5	1	-27%	26%
	2	-19%	19%
	4	-12%	12%
	8	-9%	6%
	16	-6%	4%
	32	-4%	2%
	64	-1%	1%

	128	-3%	7%

The cluster will stay below these values regardless of the number of nodes and after every addition of a new node.

Note: the numbers in this table refer to cases where racks are not defined, or the number of racks is greater than the replication factor. In the case where racks are defined and their number is equal to the replication factor, the values for replication factor 1 apply (post DB-1253; the distribution can be much worse before that).

For practical matters the maximum node load is the important figure as it specifies the amount of headroom that has to be available to handle the variation of token share. Useable values for that indicator (<10%, better than 256 vnodes randomly allocated) are achieved with 8 vnodes.

How to use

The algorithm is switched on using the `allocate_tokens_for_local_replication_factor` option in `cassandra.yaml`. It takes as argument the replication factor in the local datacentre where the node is started. The algorithm then chooses tokens which distribute load evenly in the DC for that replication factor and the number of tokens specified in `num_tokens`.

To add a new node to an existing DC, find the replication factor in the DC and specify it as argument to `allocate_tokens_for_local_replication_factor`, and start-up should take care of allocating the tokens. Note that the load will be proportional to the number of allocated tokens, so if you have existing nodes in the datacentre with a given number of vnodes, the number of tokens allocated to the new node should be the same, or proportional to the usage share you want to give it compared to the existing nodes (i.e. if it is new hardware and needs to take 2x as much load, use twice the number of vnodes of existing nodes).

To use the algorithm when building a new DC, one can use the following additional steps:

- The number of tokens per node does not need to be as high as for random allocation, and is independent of the number of tokens in other datacentres. The generally recommended number is 8, which should achieve <10% overutilization (see table above).
- Set `allocate_tokens_for_local_replication_factor` to the replication factor you want to use in the new DC.
- Add one node from a rack at a time.
- The expected load distribution can be seen before any data is loaded using `nodetool status`.

Further work

Removal: Some way of limited transfer of ownership after decommission is needed to be able to take better care of this case. Currently the method of restoring node balance after removal of an arbitrary node involves:

- Removing more than the necessary number of nodes
- Re-adding some back using the allocation algorithm

This involves too much streaming and a period of overuse of the cluster / DC. We can probably do better by, e.g. having a method allowing a node to reassign its tokens while it is online.

If choosing the nodes to decommission is an option, and they were added using the allocation algorithm, one should decommission nodes in the reverse of the order in which they were added (i.e. remove the last added node first). This has the effect of reverting to the state before that node was added and should thus preserve the algorithm's ownership distribution.