SIP-8 Autoscaling policy engine V2

Status

Current state: Implemented

Discussion thread: https://the-asf.slack.com/archives/CEKUCUNE9/p1585240648004600, #solr-scaling Slack channel

JIRA:
- SOLR-14275 - Policy calculations are very slow for large clusters and large operations
- SOLR-14409 - Existing violations allow bypassing policy rules when adding new replicas
- SOLR-14613 - Provide a clean API for pluggable replica assignment implementations

Released: 9.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast). Confluence supports inline comments that can also be used.

Note: to simplify the discussion, in this document the current policy engine implementation is called V1, and the new implementation is called V2.

Motivation

Autoscaling policy engine is responsible for calculating the placement of collection replicas across cluster nodes. The best locations for new (or moved) replicas are calculated based on a defined set of rules in the autoscaling policy configuration, and the current state of the cluster.

There are numerous problems with the current implementation of the policy engine (V1):

- The implementation is large, very complex and undocumented. It's very hard to add new functionality or fix bugs due to non-obvious interactions between the parts of the framework (see below).
- The original author is no longer in a position to efficiently maintain it long-term, and there are no other dedicated developers that understand the framework sufficiently and are committed to the ongoing support. At this point it's an undocumented black-box that cannot be maintained long-term.
- It uses an iterative process for finding the best replica locations, testing all rules for each node and for each new replica separately (even if a bulk change is requested). This makes the process very costly in terms of CPU and time (roughly $O(\text{rules} \times \text{nodes}^K \times \text{replicas}^M)$).
- This results in a paradox - the autoscaling framework is not scalable itself due to this complexity. Creating a collection with 1000 replicas on a cluster with 1000 nodes and using a simple EQUAL rule set takes hours due to the computation time spent on the policy calculations. These calculations occur in the Overseer context, which further destabilizes other cluster operations.
- Data objects manipulated by the framework during calculations are often mutable, and their modifications are often hidden as side-effects of larger operations. They are also being copied multiple times, sometimes using deep copy and sometimes shallow copy, which is further affected by side-effect modifications. All of these aspects make the modifications and their scope nearly impossible to track.
- Some computed values are cached extensively, which to a degree helps to alleviate the inherent complexity of the approach - but it also makes the data modifications and copying even more difficult to trace and prove that the cached values are up-to-date, or that they aren't evicted and recomputed needlessly. Still, the number of cache lookups is so high that in the profiling traces a significant time is dedicated just to Map.get operations.
- The rules DSL is very generic and expressive and allows for building very complex rule sets. Consequently this affects the complexity of the implementation.

The situation with the V1 implementation is unlikely to change in a significant way - there are no volunteers qualified and capable of a major effort to refactor and document the internals of this implementation, and it's equally unlikely there will be developers capable of maintaining the current implementation going forward.

For these reasons we propose that the current V1 implementation should be deprecated and a new, simpler, well-structured and well-documented V2 should be created to eventually replace the V1.

Other considerations

Containerized Solr

Chris M. Hostetter suggested that in the light of Solr being more and more often used in containerized environments (Docker, Kubernetes), which already provide their own support for up- / down-scaling, the built-in V2 framework in Solr should offer only a bare minimum to support the most common cases (eg. equal placement of replicas across nodes, by # of cores & freedisk). The scope to the Solr autoscaling would be to adequately support basic needs of standalone (non-containerized) Solr clusters. All other more sophisticated scenarios should be left out, but Solr should provide API hooks to make it easier for external frameworks to react and optimize the layout and resolve resource constraints (eg. too many / too few nodes for the # of replicas).

Clean-cut pluggable APIs
Concerns were raised that the current autoscaling implementation is too intrusive, regardless of its strengths and deficiencies. Ilan Ginzburg, Noble Paul and Andrzej Bialecki are investigating what a minimal set of APIs could look like. Some others proposed a spike to investigate how much effort would be to remove the autoscaling completely, clean up the existing APIs and add it again as a plugin (using the Plugins framework).

Requirements for the V2 policy engine

The following sections describe common user stories and autoscaling use cases that should be supported by the V2 engine.

Based on the use cases we're going to compile a prioritized list of requirements and then select a subset of the requirements for the initial MVP implementation.

User stories

*Based on these user stories we're going to compile the most common use cases.*

From Shalin:

1. Run a single replica of a shard in a node i.e. don't put more than one replica of the same shard in the same node
2. Distribute replicas for each shard equally across availability zones e.g. if there are three zones available and 6 replicas for a shard, put two replicas in each zone
3. Distribute shards equally in each zone i.e. each zone should have at least one replica of each shard available
4. Pin certain replica types to specific nodes (identified by a system property)
5. Pin all replicas of a given collection to specific nodes (identified by system property).
6. Pin a minimum (or exact) number of replicas for a shard/collection to a given node set.
7. Overall distribute replicas equally across all nodes so that resource utilization is optimum

Some notes on the above:

1. Use-case #1 is for fault tolerance. Putting more than one replica on the same node does not help with redundancy.
2. Use-cases #2 and #3 are for for fault tolerance and cost minimization:
   a. You want to survive one out of three zones failing so you need to distribute shards equally among at least two zones.
   b. You want to have (approximately) equal capacity for each shard in these zones so a zone outage doesn't eliminate a majority of your capacity.
   c. You want to have at least one replica of each shard in a given zone so that you can minimize cross-AZ traffic for searching (which is chargeable in AWS)
   d. Taking all the above scenarios on mind, either all shards of a collection must be hosted in the same two zones or all shards are hosted equally in all three zones to provide both fault tolerance as well as to minimize inter-az cost.
3. Use-case #4 is useful for workload partitioning for writes vs reads e.g. you might want to pin TLOG replicas to a certain node type optimized for indexing and PULL replicas on nodes optimized for searching.
4. Use-case #5 is for workload partitioning between analytics, search and .system collections so you can have collections specific to those workloads on nodes optimized for those use-cases.
5. Use-case #6 is useful to implement autoscaling node groups such that a specific number of nodes are always available and the rest come and go without causing data loss or moving data each time we scale down. It is also useful for workload partitioning between analytics and search use-case e.g. we might want to dedicate a few replicas for streaming expressions and spark jobs on nodes optimized for those and keep other replicas for search only.
6. Use-case #7 is for balanced utilization of all nodes. This is tricky with disk usage or heavily/lightly loaded collections.

Multi-tenant use-cases (think a collection per tenant) are trickier because now you want to take care of blast radius as well in case a node or zone goes down.

From Ilan:

A minimalistic autoscaling would offer the following properties, expressed in very vague terms:

- Prevent multiple replicas of a shard from being placed on same node,
- Try to spread replicas on all nodes (random placement ok)
- Try to spread replicas for different shards of same collection on all nodes (so that concurrent execution of queries on multiple replicas does use available compute power)
- Try to spread leaders on all nodes

Then, a periodic task/trigger moving replicas and leaders around would correct the imbalance that may result from the above operations, that therefore can be imperfect, hence the use of “try” in the descriptions (would also support adding a new empty node for example).

On top of the above, being able to spread not only on nodes but on groups of nodes (for example groups representing AZ’s) would be helpful. Nice to have also is auto add replica when nodes go down.

*Please add your user stories in the following sections...*
Prioritized requirements

Minimally Viable Product

This section describes the minimum set of functionality that the V2 engine should initially implement in order to be a practical replacement for the majority of typical Solr users. This MVP should implement the top requirements identified above.

Public Interfaces

The current V1 DSL for defining autoscaling policy rules is very expressive, which unfortunately also affects the complexity of the implementation.

That said it already exists, users and developers are already familiar with it, so we should evaluate whether it's possible to preserve a subset of this DSL in the V2 implementation, or should a new DSL be implemented from scratch.

Proposed Changes

Describe the new thing you want to do in appropriate detail. This may be fairly extensive and have large subsections of its own. Or it may be a few sentences. Use judgement based on the scope of the change.

Compatibility, Deprecation, and Migration Plan

The V1 policy engine should be kept for at least 8.x and initial 9.x releases, until the V2 implementation matures and becomes a practical replacement for most users.

Until then the default policy engine to use should be the V1, unless otherwise specified by

- cluster configuration?
- autoscaling config?
- collection configuration?

Additionally, due to the performance issues with the V1 policy engine the new one should be the default for clusters larger than N nodes (where N > 100 ?). It should still be possible to opt-out and default to the current engine.

Phase 1 of the migration: we can implement a cluster & collection property that defines what assignment strategy it should use (with collection-level property overriding the cluster-level property or default if missing). This property would select one of the existing AssignStrategy implementations or a user-provided custom one. This effectively allows users to switch policy engines on a per-collection basis.

- What impact (if any) will there be on existing us
- If we are changing behavior how will we phase out the older behavior?
- If we need special migration tools, describe them here.
- When will we remove the existing behavior?

Security considerations

Describe how this proposal affects security. Will this SIP expose Solr to new attack vectors? Will it be necessary to add new permissions to the security framework due to this SIP?

Test Plan

Describe in few sentences how the SIP will be tested. We are mostly interested in system tests (since unit-tests are specific to implementation details). How will we know that the implementation works as expected? How will we know nothing broke?

Rejected Alternatives
If there are alternative ways of accomplishing the same thing, what were they? The purpose of this section is to motivate why the design is the way it is and not some other way.

Potential alternatives:

- extend the scope of one of the other existing assignment strategies and make it compatible with the DSL of the V1 engine.
  - LegacyAssignStrategy
    - a primitive $O(\text{nodes} \times \text{replicas})$ strategy. Basic principle is that nodes are sorted by the number of existing cores, and replicas are assigned in a round-robin fashion in this order. For the number of replicas exceeding the number of nodes this strategy degenerates to equal assignment to nodes without any preference for less loaded nodes or nodes without replicas from the same shard.
  - RulesBasedAssignStrategy
    - described in detail here: https://lucene.apache.org/solr/guide/8_5/rule-based-replica-placement.html. This is deprecated and was scheduled to be removed, but it implements a significant subset of the V1 policy engine at a comparably minimal execution cost. Many V1 policy rules can be expressed in this legacy rule language (see https://lucene.apache.org/solr/guide/8_5/migrate-to-policy-rule.html). The DSL is different but we could provide a compat shim. Perhaps we should evaluate it as a starting point for V2? This strategy hasn't been tested for performance in very large clusters and very large collections, so it may very well turn out that it has the same inherent performance issues as V1.