Searchlight Specs

Release 0.0.1.dev53

OpenStack Searchlight Team

May 28, 2020

CONTENTS

1 Searchlight Train Specifications **1**
1.1 Tacker Plugin for Searchlight . . . 1
1.1.1 Problem Description . . . 1
1.1.2 Proposed Change . . . 1
Alternatives . . . 2
1.1.3 Implementation . . . 2
Assignee(s) . . . 2
Work Items . . . 2
1.1.4 References . . . 2

2 Searchlight Pike Specifications **3**
2.1 Default filters . . . 3
2.1.1 Problem Description . . . 3
2.1.2 Proposed Change . . . 3
Alternatives . . . 4
2.1.3 References . . . 4
2.2 Nova service plugin . . . 4
2.2.1 Problem Description . . . 4
2.2.2 Proposed Change . . . 4
Alternatives . . . 4
2.2.3 References . . . 4

3 Searchlight Ocata Specifications **5**
3.1 Ironic plugin . . . 5
3.1.1 Problem Description . . . 5
3.1.2 Proposed Change . . . 5
Alternatives . . . 5
3.1.3 References . . . 6

4 Searchlight Newton Specifications **7**
4.1 Add nova server groups plugin . . . 7
4.1.1 Problem Description . . . 7
4.1.2 Proposed Change . . . 7
Alternatives . . . 7
4.1.3 References . . . 8
4.2 Cross-Region Search . . . 8
4.2.1 Problem Description . . . 8
4.2.2 Proposed Change . . . 9
Alternatives . . . 9
4.2.3 References . . . 9

4.3 Index Performance Enhancement . . . 10
4.3.1 Problem Description . . . 10
4.3.2 Proposed Change . . . 11
Alternatives . . . 12
4.3.3 References . . . 12
4.4 Notification Forwarding (to systems like Zaqar) . . . 12
4.4.1 Problem Description . . . 12
4.4.2 Proposed Change . . . 13
Alternatives . . . 14
4.4.3 References . . . 14
4.5 Pipeline Architecture . . . 15
4.5.1 Problem Description . . . 15
4.5.2 Proposed Change . . . 15
Alternatives . . . 16
4.5.3 References . . . 16
4.6 Support Nova microversions for Nova plugin . . . 16
4.6.1 Problem Description . . . 16
4.6.2 Proposed Change . . . 16
Alternatives . . . 17
4.6.3 References . . . 17

5 Searchlight Mitaka Specifications **18**
5.1 ElasticSearch Deletion Journaling . . . 18
5.1.1 Problem Description . . . 18
Background . . . 18
Examples . . . 19
Example #1 (Fusillade) . . . 19
Example #2 (Nostradamus) . . . 19
5.1.2 Proposed Change . . . 20
Alternatives . . . 20
Previous Design . . . 20
ElasticSearch Index Mapping Modification . . . 20
Searchlight Delete Functionality Modification . . . 21
Searchlight Query Functionality Modification . . . 21
Searchlight Create/Modify Functionality Modification . . . 21
Configuration Changes . . . 21
Deleted Field Options . . . 21
5.1.3 References . . . 22
5.2 Per resource policy control . . . 22
5.2.1 Problem Description . . . 22
5.2.2 Proposed Change . . . 22
Alternatives . . . 23
5.2.3 References . . . 23
5.3 Searching admin-only fields . . . 23
5.3.1 Problem Description . . . 23
5.3.2 Proposed Change . . . 23
Role-based filtering . . . 23
Second choice - Separate indexes . . . 25
Alternatives . . . 26
5.3.3 References . . . 26
5.4 Zero Downtime Re-indexing . . . 26
5.4.1 Problem Description . . . 27
Background . . . 27
5.4.2 Proposed Change . . . 28

Illustrated Example . . . 30
Implementation Notes . . . 38
Implementation Note #1: Multiple Indexes . . . 38
Implementation Note #2: Incompatible Changes . . . 38
Alternatives . . . 39
Alternate #1 . . . 39
Alternate #2 . . . 39
Future Enhancements . . . 40
5.4.3 References . . . 40

CHAPTER

ONE

SEARCHLIGHT TRAIN SPECIFICATIONS

1.1 Tacker Plugin for Searchlight

https://storyboard.openstack.org/#!/story/2004968

This spec is proposed to support indexing Tacker resource information into ElasticSearch.

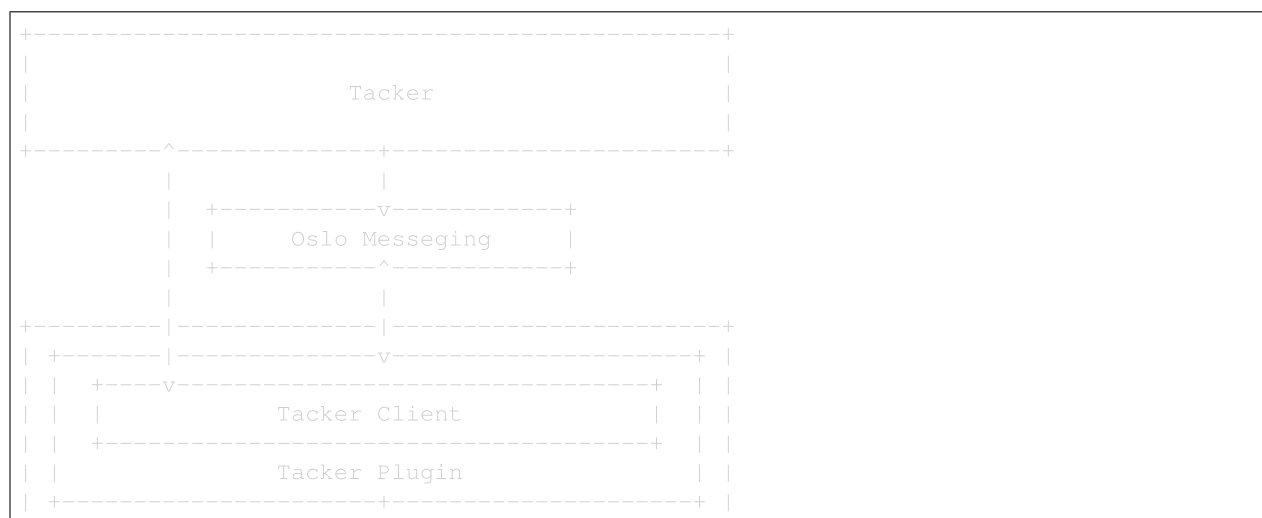
1.1.1 Problem Description

Tacker is a software that facilitates the OpenStack components to provide NFV Orchestration. While leveraging the OpenStack infrastructure to realize its elements (e.g., virtual machines as VNFs, etc.), Tacker keeps its own copy of the resource definitions in a separate database. That database can only be accessed by using the Tacker APIs. So, it would be beneficial to index Tacker resource information and events into Searchlight to provide a universal search interface for users.

1.1.2 Proposed Change

The Tacker plugin will support indexing Tacker resources via Tacker API. The plugin will use the python-tackerclient to communicate with Tacker server to query its resource information. The plugin will then index that information into ElasticSearch database. Tacker plugin also offers Searchlight listener the ability to acknowledge any change on those resources and update the corresponding data in ElasticSearch.

The following figure describes the overall architecture of the proposed plugin:



(continues on next page)

(continued from previous page)

The following Tacker resource information will be indexed:

- Network Services (NS)
- Virtual Infrastructure Managers (VIM)
- Virtual Network Functions (VNF)
- Virtual Network Function Forwarding Graphs (VNFFG)

Alternatives

None

1.1.3 Implementation

Assignee(s)

Primary assignee: Trinh Nguyen <dangtrinhnt@gmail.com>

Work Items

1. Create a Tacker plugin for Searchlight to index resource information.
2. Add unit & functional tests.
3. Add user guides.

1.1.4 References

- https://docs.openstack.org/tacker/latest/
- https://docs.openstack.org/python-tackerclient/latest/
- https://docs.openstack.org/oslo.messaging/latest/
- https://www.elastic.co

CHAPTER

TWO

SEARCHLIGHT PIKE SPECIFICATIONS

2.1 Default filters

https://blueprints.launchpad.net/searchlight/+spec/overridable-default-filters

This spec is proposed to support cases in plugins where a filter should be applied for most searches, but should be overidable by a user.

2.1.1 Problem Description

The two cases identified thus far for supporting default (but overridable) filters are glances community images, and nova deleted server instances. In both cases there are query clauses that should be applied to searches by default, but which a user should be able to explicitly disable.

2.1.2 Proposed Change

In addition to RBAC filters which are applied to all queries on a per-plugin basis, this change will allow plugins to specify additional filters that will be applied by default alongside the RBAC filters. For these defaults, however, the query builder will examine the incoming query for any instances of the fields that would be filtered on, excluding any filters that the user has explicitly specified.

This solution will have some flaws. query_string clauses are by their nature difficult to analyze; potentially we can look for instances of `key:` after splitting on `&`. For structured queries, looking for instances of the filter key in the structured query should be good enough.

In addition, it will be difficult/impossible to know whether a filter should be overridden only for specific types (e.g. if `deleted` is a default filter for the Nova servers plugin, it will be removed for any query including `deleted` as a term even if it wasnt intended to apply to Nova servers).

These limitations are somewhat unavoidable given the flexibility of the Elasticsearch DSL. The cases for removing these filters are specific enough that edge cases arent important.

Alternatives

None, but this restricts the use of Searchlight for Novas cells, and the ability to match Glances API with respect to community images.

An alternative implementation option would be a specific option for disable default filters at the query top level. This would be safer, more performant, more predictable but would require knowledge of these defaults (e.g. that a search for `_type:OS::Nova::Server AND deleted` wont return anything unless the additional override parameter is given).

2.1.3 References

2.2 Nova service plugin

https://blueprints.launchpad.net/searchlight/+spec/nova-service-plugin

This spec is proposed to support nova service plugin (OS::Nova::Service, note that OS::Nova::Service doesnt exist in heat resource type same as the hypervisor plugin, and it is an admin-only resource type), and versioned notifications are supported for services in nova [0], it would be a nice additional plugin for Searchlight.

2.2.1 Problem Description

A service[1] takes a manager and enables rpc by listening to queues based on topic. It also periodically runs tasks on the manager and reports its state to the database services table. So in cloud with large amount of compute nodes, there will be a pretty large number of services (typically one service each compute node, and four services each controller node). The list or search services (you can use command *nova service-list* to get the whole list or filter the result by host or binary) may get slow using the native nova API. And the versioned notifications for hypervisor [2] refers to the service id. In future implementation for notification in hypervisor plugin [3] we may want to fetch the service details for hypervisor create or update action by the service id.

2.2.2 Proposed Change

1. Support index services through nova API.
2. Support versioned notifications.

Alternatives

None

2.2.3 References

[0] https://github.com/openstack/nova/blob/master/nova/objects/service.py#L309-L315 [1] http://docs.openstack.org/developer/nova/services.html#the-nova-service-module [2] https://review.opendev.org/#/c/315312/11/nova/notifications/objects/compute_node.py [3] https://github.com/openstack/searchlight/blob/master/searchlight/elasticsearch/plugins/nova/notification_handler.py#L107

CHAPTER

THREE

SEARCHLIGHT OCATA SPECIFICATIONS

3.1 Ironic plugin

https://blueprints.launchpad.net/searchlight/+spec/ironic-plugin

This spec is proposed to add ironic plugin for Searchlight. Ironic is OpenStack baremetal service. Plugin should support these baremetal resources: nodes (OS::Ironic::Node), ports (OS::Ironic::Port) and chassis (OS::Ironic::Chassis).

3.1.1 Problem Description

Notifications about baremetal node state changes (power, provisioning) and create, update and delete of resources are proposed to ironic ([1],[2]). Because information about node in the database can be changed quickly during deployment specification[2] provides ways to limits flow of notifications. Using of Searchlight API with ironic plugin can reduce load on ironic API from periodical polling tasks.

3.1.2 Proposed Change

1. Searchlight listener should be changed, because ironic can use also ERROR notifications message priority. New handler for ERROR priority will be added.

2. Plugin with indexers and notification handlers for ironic nodes, ports and chassis should be implemented.

3. Custom Searchlight configurations should be used with ironic because ironic uses own hardcoded `ironic_versioned_notifications` topic ([3]).

Alternatives

None

[1] http://specs.openstack.org/openstack/ironic-specs/specs/approved/notifications.html
[2] https://review.opendev.org/#/c/347242
[3] http://docs.openstack.org/developer/ironic/dev/notifications.html

3.1.3 References

CHAPTER

FOUR

SEARCHLIGHT NEWTON SPECIFICATIONS

4.1 Add nova server groups plugin

https://blueprints.launchpad.net/searchlight/+spec/nova-server-groups-plugin

This Blueprint adds a plugin for Nova server groups (OS::Nova::Server_groups).

4.1.1 Problem Description

Currently, in nova, there are no filter support fo os-server-groups API, that means, when list server groups, all the existing server groups will be listed. As server groups are very widely used feature in commercial deployment, this will be problematic, especially for large scale Public Cloud deployments. For example, in Deutsche Telekom OTC Public Cloud, each tenant will have 10 server groups by default, when the number of tenant grows, it will be a bottleneck to list and search for particular server groups. And it will also be very user-friendly to let user search for server groups with `name`, `policy` or `members` which is not yet provided by Nova.

4.1.2 Proposed Change

Phase I:

Add a Nova server groups plugin to collect server groups data and provide the ability to search server groups using `name`, `policy`, `members`, `id` and `metadata`.

Phase II: Add new notification handler for server groups notifications once the notification for server groups in nova has been added.

Alternatives

Not add this plugin and we will lack the support for a widely used nova feature.

4.1.3 References

4.2 Cross-Region Search

SearchLight currently is targeted at being deployed alongside nova, glance, cinder etc. as part of an Openstack control plane. There has been a lot of interest in allowing SearchLight to index and search resources across Openstack regions.

4.2.1 Problem Description

A typical production Openstack deployment can provide scaling and resilience in several ways, some of which apply to all services and some specific to services that support a feature.

Availability zones provide some ability to distribute resources (VMs, networks) across multiple machines that might, for instance be in separate racks with separate power supplies. AZs are represented in resource data and are already part of the indexing that SearchLight does.

Regions can provide separation across geographical locations (different data centers, for instance). The only requirement to run multiple regions within a single Openstack deployment is that Keystone is configured to share data across the regions (with master-master database replication, for instance). All other services are isolated from one another, but Keystone is able to provide the URLs for, say, the `nova-api` deployments in each region. A keystone token typically is not scoped to a particular region; Horizon currently treats a region change as a heavy weight option because it means refreshing the dashboards and panels that should be visible in the selected region.

Multiple clouds provide total isolation such that each cloud is totally separate with no knowledge of the other. Horizon also supports this model (confusingly also referring to each cloud as a Region) though changing region requires logging into the new region.

Nova cells are a feature specific to Nova that allows horizontal scaling within a single region. Cells spread the compute API over several databases and message queues in a way thats mostly transparent to users. Cells will be considered beyond the scope of this document since they address performance rather than resilience, and thus arent directly related to this feature (though may be the basis of another).

For deployments using multiple regions, the ability to search aggregated data can provide value. A Nova deployment in a fictional Region-A is unaware of resources in a fictional Region-B, so a user must make requests to each region to get information. This makes Horizon somewhat cumbersome (changing region triggers reloading pages to change the context, although authentication status is preserved).

These are the potential deployment options for multi-region clouds. The options that follow are presented in order of effort and complexity. Relative performance is noted.

1. Deploy Searchlight in the same fashion as Keystone. API endpoints can exist in both regions. Data will be duplicated between regions (by some external process - Elasticsearch explicitly does not support or recommend splitting clusters across geographical locations); Searchlight indexing will write to its local cluster and queries will be run against a local cluster. All region-aware resources will have a region id attached to them. **Best performance**, **most difficult maintenance**, **zero client complexity**.
2. Searchlight will run in each location, but data will not be duplicated across locations (similar to how nova and glance work). To allow searching across regions, Elasticsearch is configured with a tribe node that acts as a federated search node; indexing operations are always performed locally. A search against an alias on the tribe node will act as a search against that alias in all clusters to which the tribe node is joined. **Worst performance** (as bad as the slowest node, though single region searches can be optimized), **easier maintenance**, **zero client complexity**.
3. Run Searchlight in both regions separately; either have clients make queries against both regions explicitly or have Searchlights API echo requests to other regions. Likely this would enforce segregating results by region (which might be a good outcome). **Variable performance** (can receive results as they are available), **easiest maintenance**, **complexity pushed to client**.

It should be noted that options 1 and 2 provide a similar functional experience; searches will appear to run against a single API endpoint returning data in multiple regions (sorting and paging appropriately). Option 3 treats each region separately; paging and sorting would apply to each regions results. There is a decision to be made even at that level what is actually desirable; should the UI segregate results by region or is a merged view more desirable?

4.2.2 Proposed Change

After discussion at the Newton summit in Austin (where the alternative below was presented) it was agreed that a unified view is potentially very useful, but requires significant awareness of the security implications and a lot more testing. The general feeling given the workload for Newton was that this functionality can be adequately supported from the client.

As such, we will document the method by which Searchlight can be run using tribe nodes, and the networking implications it brings. We will also add `region_name` to all mappings.

This may be expanded upon in subsequent releases.

Alternatives

To enable use of tribe nodes to support searches in multiple regions:

1. Set up a stock devstack with Searchlight
2. Deploy a second devstack configured for multi-region (I set it up on a second local VM)
3. Ensure that `searchlight.conf` included the correct `region-name` in the auth credential sections
4. Ensure that the Elasticsearch cluster names were different (`cluster.name:  region-one`)
5. Check that `searchlight-manage` indexes correctly in each region
6. Set up a tribe node (again with a different cluster name) on a different port on the first devstack VM. I used manual host discovery
7. Configure a separate searchlight-api running off the tribe node. Searches against the alias return results across both clusters

This technique will suffer from the same problem that resulted in us disabling multi-index support; if the index mappings are different errors will result. The solution (as I suspect there) is to ensure identical mappings across indices even if no data is indexed into a given index.

The work required here in addition to setting up Elasticsearch would be to make `searchlight-api` use the tribe node (potentially one in every region) rather than the cluster the listener uses (or perhaps both depending on search context). This change is relatively minor. We would also need to make all resources region aware (which is a sensible change) and make sure Searchlight itself is aware of its own region (also a sensible change).

4.2.3 References

- Openstack scaling http://docs.openstack.org/openstack-ops/content/scaling.html
- Elasticsearch tribe nodes: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-tribe.html

4.3 Index Performance Enhancement

https://blueprints.launchpad.net/searchlight/+spec/index-performance-enhancement.

This feature will improve the performance of indexing resource types within Searchlight.

4.3.1 Problem Description

If the above link is too troublesome to follow, please indulge us while we plagiarize from the blueprint.

When indexing (first time or re-indexing) we will index all resource group types sequentially. We loop through all plugins, indexing each one in turn. The result is that the time it takes to re-index is equal to the sum of the time for all plugins. This may take longer than it should. In some cases a lot longer.

The time it takes to complete the full index is:

```
    n
O(  T(p) )
    p=0
```

When n is the number of plugins and T(p) is the time it takes for plugin p to index.

We should change the algorithm to index in parallel, rather than in serial. As we are looping through each plugin to re-index, we should spin each indexing task into its own thread. This way the time it takes to index is the time it takes the longest plugin to re-index.

With this enhancement, the time it takes to complete the index is:

```
         n
O( MAX( T(p) ) )
        p=0
```

To provide context for the design, we will review the current design for re-indexing. A re-indexing starts when the admin runs the command:

`searchlight-manage index sync`

Under the cover, `searchlight-manage` is doing the following:

- Determine which resource groups need to be re-indexed.
- Determine which resource types within each resource group needs to be re-indexed.
- For each resource type that *does* need to be re-indexed, `searchlight-manage` will call the plugin associated with that resource type. The plugin will make API calls to that service and re-index the information.
- For each resource type that *does not* need to be re-indexed, `searchlight-manage` will call ElasticSearch directly and re-index from the old index into the new index.
- Once all re-indexing is complete, the ES aliases are adjusted and `searchlight-manage` returns to the user.

This implies the following:

- The admin must wait for all of the re-indexing to complete before `searchlight-manage` finishes.
- When `searchlight-manage` finishes, the admin knows the exact state of the re-index. Whether it completed successfully or if there was an error.

4.3.2 Proposed Change

As described in the blueprint, we would like to reduce the time to complete the re-index. Based on discussions with the blueprint and this spec, we will be implementing only the first enhancement in the blueprint. We will be using python threads to accomplish this task. We need to understand the design issues associated with implementing a multi-thread approach.

1. **Are the indexing plugins thread-safe?**

If there are a lot of inter-dependencies within the plugins, it may not pay off to try to multi-thread the plugins. Reviewing the code and functionality of the plugins, they appear to be separate enough that they are good candidates to be moved into their own threads. The plugins are isolated from each other and do not depend on any internal structures to handle the actual indexing.

Design Proposal: The individual plugins can be successfully threaded.

2. **At what level should we create the indexing threads?**

The obvious candidates are the resource type (e.g. OS::Nova::Server) or the resource type group (e.g. the index searchlight). The main reason that we are considering this enhancement is due to the large amount of time for a particular resource type, but not for a particular resource type group.

Internal to `searchlight-manage`, this distinction fades rather quickly. We use the resource type groups to only determine which resource types need to be re-indexed. We also have an existing enhancement within `searchlight-manage` where we re-index through the plugin API only the resource types that were explicitly demanded by the user. All other resource types are re-indexed directly within ElasticSearch. We need to keep this enhancement.

Keeping the current design intact means we will want to thread on the fine resource type level and not at the gross resource type group level. Based on the parent/child relationship that exists between some of the resource types, this is the fine level we will be considering.

Since we are already using bulk commands for Elasticsearch re-indexing, we will place all of the Elasticsearch re-indexing into a single thread. Considering that this will be I/O bound on Elasticsearchs side, There does not appear to be any advantage of doing an Elasticsearch re-indexing for each resource type in a separate thread.

Design Proposal: Whenever the indexing code currently calls the plugin API, it will create a worker in the thread pool.

Design Proposal: All of the calls to ElasticSearch to re-index an existing index, will be placed in a single worker in the thread pool.

3. **Mapping of plugins to threads**

There may be a large number of plugins used with Searchlight. If each plugin has its own thread, we may be using a lot of threads. Instead of having a single thread map to a single plugin, we will use a thread pool. This will keep the number of threads to a manageable level while still allowing for an appropriate level of asynchronous re-indexing. The size of the thread pool can be changed through a configuration option.

Design Proposal: Use a thread pool.

4. **When will we know to switch the ElasticSearch aliases?**

In the serial model of re-indexing, it is trivial to know when to switch the ElasticSearch alias to the use the new index. Its when the last index finishes! Switching over to a model of asynchronous threads running in parallel potentially complicates the alias update.

The indexing code will wait for all the threads to complete. When all threads have completed, the indexing code can continue with updating the aliases.

Design Proposal: The alias switching code will be run after all of the threads have completed.

5. **How do we clean up from a failed thread?**

The indexing code will need to have the threads communicate if a catastrophic failure occurred. After all workers have been placed into the Thread pool, the main program will wait for all of the threads to finish. If any thread fails, it will raise an exception. The exception will be caught and the normal clean-up call will commence. All threads that are still waiting to run will be cancelled.

Design Proposal: Catch exceptions thrown by a failing thread.

For those following along with the code (searchlight/cmd/manage.py::sync), here is a rough guide to the changes. We will reference the sections as mentioned in the large comment blocks:

- First pass: No changes.
- Second pass: No changes.
- Step #1: No changes.
- Step #2: No changes.
- Step #3: No changes.
- Step #4: Use threads. Track thread usage.
- Step #5: No changes.
- Step #6: No changes.

Alternatives

We can always choose to not perform any enhancements. Or we can go back to the first draft of this spec.

4.3.3 References

4.4 Notification Forwarding (to systems like Zaqar)

https://blueprints.launchpad.net/searchlight/+spec/notification-forwarding

This feature adds the ability to forward the Searchlight consumed indexing notifications to external providers such as Zaqar[1] via a simple driver interface. This would include the data enrichment that Searchlight provides on top of the simple OpenStack notifications.

4.4.1 Problem Description

There are a number of use cases by projects within OpenStack and outside of OpenStack that need to know when OpenStack resources have been changed. OpenStack provides a notification bus for receiving notifications published by services, however there are a few limitations:

- The OpenStack message bus is not typically directly exposed external consumers for security concerns.
- The notifications often only contain a subset of the data about a particular resource, that is incomplete, not rich enough
- The notification data does not look the same as the API results
- The notification bus does not support all the message subscription semantics or mechanisms needed, like per-subscriber filtering or web sockets/hooks

Searchlight handles a number of the above problems and will continue to evolve with OpenStack so that as notifications change / are enriched, it will evolve with them. It does this by indexing the data from the OpenStack Services from a variety of locations, including listening to notifications. It enriches the data provided by the base notifications in a number of ways, which allows the index to be used directly instead of the APIs [2]. The latter has the additional advantage of reducing API call pressure on the various services.

Searchlights enhanced searching capabilities [3] and performance [4] over typical API responses are compelling; consequently it has been integrated into horizon[5][6][7] and elsewhere.

Current Searchlight users will still need to use polling mechanisms to get updated information from Searchlight. Many Searchlight consumers will need an asynchronous way to stay informed of OpenStack Resource changes, such as Availability and Status, instantly. They will also need a way that ensures that the status updates they receive are coherent with the status of the resources indexed into Searchlight.

1. A Horizon painpoint today, despite the availability of Searchlight and its many benefits stemming from richer notifications and elastic search, is that Horizon still needs to periodically poll Searchlight for updates. (Where a notification is not handled by Searchlight, direct calls to OpenStack service APIs are necessary. However, more and more OpenStack projects are being integrated with Searchlight.) A notification service that instead provides updates as they occur would be attractive. Consider for instance a VM launch request. In the ideal scenario the Horizon UI updates automatically to display the various stages in a virtual machine launch as the transitions occur. Today this is achieved in Horizon behind the scenes by frequently polling Searchlight and/or other OpenStack API services.

2. Telco vendors, with their demanding service up time requirements and low tolerance for service delays, also seek resource change notifications. Typically they will have a management layer/application that seeks visibility into resource updates. For example, CRUD of a user, VM, glance image, flavor or other. For instance, the arrival of a new user may need to trigger a workflow such as welcome messages, proferring services, and more. Searchlight today detects such events and indexes them but is unable to push them to Telco management layer instantly, near real time.

3. For third party applications, often there is interest in monitoring tools that want insight into resources and their availability. These might span flavors, instances, storage, images, and more and their status. Typically these third party monitor systems will not have access to the OpenStack message bus for security reasons. Consider for example, a content provider, who would like to advertise to their customer base when a new movie is uploaded into Swift. Yet another example may be triggering a upgrade action when a security patch gets uploaded into Glance.

4.4.2 Proposed Change

We propose that Searchlight add the ability to forward notifications. Further, a forwarding infrastructure that supports a pipeline of forwarding entities would provide maximum flexibility. For example, consider paste pipeline, within the order and entities as specified in the paste.ini file. Note, in a sense Searchlight is the first element in notification pipeline, the entity that takes as input the primary notifications from the OpenStack message bus and enriches them in addition to indexing them.

A notification consumer might be Zaqar, the OpenStack messaging as a service project or even simpler message forwarding mechanisms such as WebSockets, or even a simple logger.

The notification consumer would need to be fast so as not to bog down Searchlights notification push subsystem. Further, rules of engagement need to be agreed upon. For instance, if Searchlight is unable to contact a registered notification consumer, what should it do? See bug [8]. Log the error and forget it? Should it re-try some pre-configured number of times after some pre-configured wait interval between tries? Should it hold the notifications till it can successfully send them? This latter solution may be overly demanding on Searchlight especially when there are multiple registered consumers.

Our solution strategy is to define a consumer-plugin for Searchlight for each consumer. If Zaqar is the consumer, than a Zaqar plugin in Searchlight. The intricacies of error handling on connection fail can be left to the Zaqar plugin, any filtering of notifications to be actually transmitted from Searchlight to Zaqar can further be handled there. Leaving the error handling to the plugin makes sense because some consumers may care about lost notifications while others may not. For example, a mail program displays messages as they arrive, but occasionally the VPN goes down, or there is

no wireless connectivity or other problem. The mail reader then may on reconnect just issue a mail-synch. It is in this vein that we leave notification handling to the plugin and its associated consumer and its end-user API. Essentially re-try behavior, re-synch behavior are all left to the plugin-consumer pair.

The main notification/message flow would look like the below: OS Internal Service > Searchlight > Zaqar-plugin > Zaqar > External app

For each Searchlight plugin, it may have a notification forwarder configured. After Searchlight has received a notification, performed any data enrichment, and indexed into ElasticSearch, it would send the enriched data to the configured notification forwarder via its plugin.

The notification forwarder would support filtering out forwarded fields:

- Unsearchable fields (already configurable by Searchlight)
- Admin only fields
- By regex (similar to Glance property protections)

Searchlight may ease plugin development by refactoring the above functionality into a common utility. The respective plugins may want to leave filtering as configurable parameters or hard code them. This is entirely up to the plugin-consumer pair.

This way, when a resource gets updated by notification, the updated resource will be sent out to a messaging system like Zaqar. The Zaqar message body will include the complete resource data from Searchlight.

Alternatives

An alternative is introducing a brand new service, split out from ceilometer, to listen to notifications, but that would have the following shortcomings.

Would either not have or would have to rebuild Searchlights ability to enrich data and to know about sensitive data.

Generically solving the issue of cache coherency from listening to notifications and from periodic re-synchs would still be an issue. Ideally no consumer should be obliged to deal with a flood of notifications resulting from another consumer initiating some action. Consider for instance the Horizon table view, such as the instance table which lists all instances. If Horizon was just consuming the notification data to display new instances as they become available, it could call the Nova API to supplement any currently displayed list of instances. However, the results the user sees from searching/ Searchlight (see Horizon blueprints referenced) are different. Likewise the results will vary should the search criteria be changed. By keeping Searchlight as the first hop in a notification pipeline, we ensure the user has a consistent view of all notifications, barring any re-synchs the user initiates.

4.4.3 References

[1] https://launchpad.net/zaqar

[2] https://www.youtube.com/watch?v=0jYXsK4j26s&feature=youtu.be&t=2053

[3] https://www.youtube.com/watch?v=0jYXsK4j26s&feature=youtu.be&t=167

[4] https://blueprints.launchpad.net/horizon/+spec/searchlight-search-panel

[5] https://blueprints.launchpad.net/horizon/+spec/searchlight-images-integration

[6] https://blueprints.launchpad.net/horizon/+spec/searchlight-instances-integration

[7] https://www.youtube.com/watch?v=0jYXsK4j26s&feature=youtu.be&t=1771

[8] https://bugs.launchpad.net/searchlight/+bug/1524998

4.5 Pipeline Architecture

https://blueprints.launchpad.net/searchlight/+spec/pipeline-architecture

This feature enables flexible pipeline architecture to allow Searchlight to configure multiple publishers to consume enhanced data.

4.5.1 Problem Description

Currently when a notification comes to Searchlight, it gets processed in a notification handler. The handler enriches notifications in a number of ways and indexes enriched data into Elasticsearch. This is a simple and straightforward approach because Elasticsearch is the only storage backend in Searchlight now. As we are going to introduce notification forwarder [1], this process becomes inflexible. A pipeline architecture is needed to provide extra flexibility for users to define where enriched data is published to. It also allows Searchlight to support other non-elasticsearch backends in the future.

4.5.2 Proposed Change

We propose that Searchlight change to a pipeline architecture to provide flexibility for forwarding notifications.

The current main message flow in Searchlight is looking like the below: Source(Notifications) -> Enrichment&Index to Elasticsearch.

Currently notification handlers wait for notifications, transform them and index data into Elasticsearch. Its tight coupled and not modular. A consistent workflow is needed.

With pipeline in place, the main message flow would look like: Source(Notifications) -> Data Transformer(Enrichment) -> Publishers(Elasticsearch, Zaqar).

To achieve this, some refactors to Searchlight are needed. Notification handlers focus solely on capturing supported notification events. After that, notifications are forwarded to data transformers. There are mainly two kinds of data transformation. One is to normalize a notification into an OpenStack resource payload. The payload is api compatible format without publisher additional metadata. The normalization is always done by either calling OpenStack API services or updating existing Elasticsearch data. These resource transformers are plugin-dependent. For example, a nova create instance notification could be normalized into a server info document. Besides resource data enrichment, there might be some publisher metadata to be attached, like user role field, parent child relationship, version in Elasticsearch. These transformations should be separated from resource data enrichment.

Publishers should implement a method to accept enriched data, notification information, as well as an action indicated resources CURD. For example, if a nova server has been updated, publishers in the pipeline will receive server full info, server update notification and an update action. It is entirely up to the publisher to decide how to deal with those information.

We see Elasticsearch indexing as a case of publisher. It could be the default publisher because for some plugins resource update needs to fetch old documents from Elasticsearch, partial update doesnt work without Elasticsearch, though in the future we may solve this issue. The order of publishers in the pipeline doesnt matter. Publisher can choose how to deal with errors, either request a requeue or just ignore the exception. The requeue operation is especially useful for Elasticsearch publisher, because data integrity is important for search functions of Searchlight. Requeue should not affect other configured publishers, thus a filter is needed to make sure publishers wont deliver same message twice.

Currently Searchlight gets its data in two ways, one is incremental updates via notifications, the other is full indexing to ElasticSearch via API calls. Incremental updates are notified to all the publishers configured in the pipeline. For reindexing, it is up to publishers to decide if they want reindexing or not.

There are two alternatives of pipeline design. One is the pipeline only consists of publishers. Notifications are normalized by resource transformer, then passed to configured publishers. Publishers could attach specific metadata inside themselves. Users can only control what publishers Searchlight data is heading for. Another alternative is make both transformers and publishers configurable. By combining different transformers and publishers, one can produce different pipeline on same notification.

Alternatives

4.5.3 References

[1] https://blueprints.launchpad.net/searchlight/+spec/notification-forwarding

4.6 Support Nova microversions for Nova plugin

https://blueprints.launchpad.net/searchlight/+spec/support-microversion-for-nova

This feature adds the support for Nova APIs with microversions. This would support Searchlight providing the fields that are added in new microversions of Nova APIs.

4.6.1 Problem Description

Nova have deprecated v2.0 API and starting to remove the codes from tree. Nova v2.1 API is designed with the microversion mechanism, that is, when new data fields are added to one particular resource, the backward compatibility is ensured by adding new microversions to related APIs[1].

For example, in[2], a new field `description` is added for servers, it is used for users to make a simple string to describe their servers, it would be very useful if we can also provide this field.

The changes that were made for each microversions could be found in[3].

4.6.2 Proposed Change

Currently, when we initialize nova client, it is hard coded to use `version=2`, this is bad in two ways:

1. The v2.0 nova API is deprecated and the code will be removed in Newton[4].
2. It can not support microversions if it is hard coded.

In this BP, a new configure option `compute_api_version` will be added in the configuration file. When we initialize nova client, this config option will be used as the version of the API version. The default value of this config option will be set to 2.1 in the design of this BP and can be modified in the future according to the changes in Nova API.

The supported data fields will be also updated according to the provided microversion.

[1] http://docs.openstack.org/developer/nova/api_microversions.html
[2] https://blueprints.launchpad.net/nova/+spec/user-settable-server-description
[3] https://opendev.org/openstack/nova/src/branch/master/nova/api/openstack/compute/rest_api_version_history.rst
[4] https://blueprints.launchpad.net/nova/+spec/remove-legacy-v2-api-code

Alternatives

Hard code the version to 2.1 as 2.0 will no longer be usable soon. But the newly added data fields cannot be supported.

4.6.3 References

CHAPTER

FIVE

SEARCHLIGHT MITAKA SPECIFICATIONS

5.1 ElasticSearch Deletion Journaling

https://blueprints.launchpad.net/searchlight/+spec/es-deletion-journaling

This feature enables tracking of object deletion in ElasticSearch to allow for better coherency and asynchronous operations.

5.1.1 Problem Description

As a service providing a snapshot of the OpenStack eco-system, we expect the following trait:

- The snapshot is up to date and coherent, independent of the order of the updates received from the other various OpenStack services.

Anything less and we will have failed our users with a deluge of undependable data.

Background

When created, an ElasticSearch document is stored in an index. When deleted, we barbarically engage in Damnatio Memoriae and the ElasticSearch document is permanently removed from the index. In most cases this is not an issue and the document is not missed. But computer science is fraught with corner cases. One such corner case is introduced with the new Zero Downtime Re-indexing functionality [1]. Now Searchlight is required to remember the existence of deleted documents.

With the addition of the Zero Downtime Re-indexing, Searchlight will be performing CRUD operations on documents simultaneously from both re-syncing (which uses API queries) and notifications sent from the services. All within a distributed environment. These notifications will result in an asynchronous sequence of ElasticSearch operations. Searchlight needs to make sure that the state of the eco-system is always correctly reflected in ElasticSearch, independent of the non-determinate order of notifications that are being thrown by the services. If a delete notification is received before the corresponding create notification, this sequence of events still need to result in the correct ElasticSearch state.

In light of this harsh reality, we need to make the Searchlight document CRUD commands order independent. This implies a way to track the state of deleted documents. See below for more concrete examples of the issues we are resolving with this blueprint.

As a side note, keeping track of deleted documents will allow Searchlight to easily provide a delta functionality if so desired in the future.

The previous incarnation of this spec detailed setting a deletion flag and using ElasticSearchs TTL functionality to delay deleting documents. It turns out that ElasticSearch has this functionality built in when versioning is in use (for an explanation, see the bottom of [3]).

Examples

Some more concrete example may help here. We will use Nova as the resource type, As a reminder here is how the plug-in commands map to ES operations.

- A Create document command in the Nova plug-in turns into an ES index operation with a new payload from Nova.
- An update document command in the Nova plug-in turns into an ES index operation with a new payload from Nova that already contains the modifications. Once thing to note here is that we do neither an ES update operation nor a read/modify/write using multiple ES operations. From ElasticSearchs frame of reference an update command is the same as a create command.
- A delete document command turns into an ES delete operation.

Example #1 (Fusillade)

Consider the following Nova notifications Create Obj1, Modify Obj1, Modify Obj1, Modify Obj1 and Delete Obj1. Due to the distributed and asynchronous nature of the eco-system. the order the notifications are sent by the listener may not be the same order the operations are received by ElasticSearch. In some cases, the last ElasticSearch modify operation will arrive after the ElasticSearch delete operation. ElasticSearch will see the following operations (in this order):

```
PUT /searchlight/obj1         # Create Obj1
PUT /searchlight/obj1         # Modify Obj1
PUT /searchlight/obj1         # Modify Obj1
DELETE /searchlight/obj1      # Delete Obj1
PUT /searchlight/obj1         # Modify Obj1
```

After all of the operations are executed by ElasticSearch, the net result will be an index of the Nova object Obj1. When queried by an inquisitive user, Searchlight will embarrassingly return the phantom document as if it corporeally exists. Folie a deux! Not good for anyone involved.

Example #2 (Nostradamus)

We will also need to handle the simplistic case of Nova creating a document followed by Nova deleting the document. This case could be rather common in the Zero Downtime Re-indexing work [1]. This sequence results in the Nova notifications Create Obj2 and Delete Obj2. If the ElasticSearch create operation arrives after the ElasticSearch delete operation, ElasticSearch will see the following operations (in this order)

```
DELETE /searchlight/obj2      # Delete Obj2
PUT /searchlight/obj2         # Create Obj2
```

After both operations are executed by ElasticSearch, the net result will be an index of the Nova object Obj2. This naughty behavior is incorrect and also needs to be avoided.

This example also illuminates a subtlety with out of order deletion notifications. There may be times when ElasticSearch is being asked to delete a (currently) non-existent document. This omen of a future event needs to be interpreted and thus handled correctly.

5.1.2 Proposed Change

There is an index setting named index.gc_deletes that defaults to 60 seconds. When a document is deleted *with a specified version* it is not immediately deleted from the index. Instead, its marked as ready for garbage collection (in the document sense, not memory sense). If an update is posted with a later version, the document is resurrected. If a document is posted with an earlier version, it raises a ConflictError, as would be the case with an alive document. The deleted document is *not* visible in searches.

Since this is essentially identical to the implementation described below, and appears to be fully supported in Elastic-Search 2.x, there is no point implementing it ourselves. We may decide to recommend a higher *gc_deletes* value, but the only change necessary is to pass a version as part of delete operations (see [4]).

Alternatives

An alternative is expressed in the previous design that follows.

Previous Design

Ecce proponente! With this blueprint, the basic idea is to keep the state of a deleted document around until no longer needed. At a high level, we will need to make three major modifications to Searchlight.

- We will need to modify the ElasticSearch index mappings.
- We will need to modify the delete functionality to take advantage of the new mapping fields.
- We will need to modify the query functionality to be aware of the new mapping fields.

ElasticSearch Index Mapping Modification

Two modifications are needed for the mapping defined for each index.

The first modification is to enable the TTL field. We need to define the mapping for a particular index like this:

```
{
    "mappings": {
        "resource_index": {
            "_ttl": { "enabled": true }
        }
    }
}
```

By not specifying a default TTL value, a document will not expire until the TTL is explicitly set. Exactly what we need.

The second modification is to add a new metadata field to the mapping. The metadata field would be named deleted and would always be defined. When the document is created/modified the field would be set to False. When the document is deleted the field would be set to True. There is some concern that we need more than a boolean for this field. A version or timestamp may be more appropriate. This is a detail for the design and can be fleshed out at that time if needed.

Searchlight Delete Functionality Modification

When a document is deleted, we will need to set both the TTL field and the metadata field. This is considered a modification to the original document.

If the document does not already exist, we will need to create the document and set the deleted and TTL fields. This will prevent an out-of-order create/update operation from succeeding.

Searchlight Query Functionality Modification

When a document is queried, we will need to modify the query to exclude any documents whose metadata indicates the document has been deleted. We will also need to filter out the metadata field.

Searchlight Create/Modify Functionality Modification

When a document is created, the mapping will need to add the new deleted field and enable TTL functionality. The deleted field will need to be set appropriately. If the deleted field is set to true we will not modify the document. These modifications depend on the version functionality being in place [2].

Configuration Changes

We need to define the TTL value to determine how long a deleted document endures. This default value can be overridden by a configuration value.

Setting a TTL value is not enough to delete a document. In tandem we need ElasticSearch to run its purge process. This purge process will poll all documents and delete those with expired TTL values. The default is to run the purge process every 60 seconds. This default value can be overridden by a configuration value.

Deleted Field Options

For historical completeness, here are the different options that were considered for the deleted metadata field.

(1) The metadata field would be named deleted and would be defined only when a document has been deleted. When a document is created/modified this field is not defined. To detect if a document is deleted we will search for the existence of this field. This simplifies the create/modify code, but complicates the query code.

(2) The metadata field would be named deleted and would always be defined. When the document is created/modified the field would be set to False. When the document is deleted the field would be set to True. This adds a little bit of work to the create/modify but simplifies the query command.

(3) The metadata field would be named state and would always be defined. The value of state would be the current state of the document: Created, Modified or Deleted. More work is needed in this option to distinguish between Modified and Create, since they are treated the same say in the plug-ins. This will allow for delta functionality to be added to Searchlight in the future. This work is the same as option (2).

5.1.3 References

[1] The Zero Downtime Re-indexing work is described here: https://blueprints.launchpad.net/searchlight/+spec/zero-downtime-reindexing

[2] External versions added to ElasticSearch documents is described here: https://review.opendev.org/#/c/255751/

[3] ElasticSearch *document* garbage collection is discussed here: https://www.elastic.co/blog/elasticsearch-versioning-support

[4] Bug report for handling out-of-order notifications: https://bugs.launchpad.net/searchlight/+bug/1522271

5.2 Per resource policy control

https://blueprints.launchpad.net/searchlight/+spec/per-resource-type-policy-control

5.2.1 Problem Description

Current policy control allows us to restrict who can query, list plugins or retrieve facets using the oslo policy engine [1]. Openstack is moving towards supporting more fine-grained controls, and for Searchlight a step in that direction is allowing control over individual resource types. For instance, it might be the case in a given cloud that non-administrative users should not be permitted to search a particular resource type. Longer term this allows us to move towards a model where RBAC is defined by policy control; rather than the hard-coded project-based RBAC we use for each plugin, we might replace or augment it with the typical *is_admin_or_owner* policy rule employed by projects.

5.2.2 Proposed Change

The proposed change will allow *policy.json* to include rules for individual plugins. A rule *resource:<resource type>:allow* will control overall access to a plugin. In addition, *allow* can be replaced with other actions to allow more precise control. For instance, rules might be:

```
"default": "",
"resource:OS::Glance::Image:allow": "@",
"resource:OS::Glance::Image:facets": "is_admin:True"
"resource:OS::Nova::Server:query": "!"
```

A future extension may extend this to support RBAC rule specification through policy. For instance, the following rules might translate into the existing RBAC query we have today:

```
"admin_or_owner": "is_admin:True or project_id:%(project_id)s",
"resource:OS::Nova::Server:allow": "admin_or_owner",
```

If a resource is *not* allowed via policy, it will be removed from the list of types to be searched; if this results in no allowed types, the search will return an empty result set.

Alternatives

Disabling plugins entirely in setup.cfg is one possibility that can be done with the current codebase.

Disabling indexing for non-administrators would require a few changes.

The ideal long-term solution (which is one that this proposal drives towards) is to consume the service `policy.json` files as does horizon. Ultimately the hard-coded RBAC rules might be expressed as policy rules in many cases, allowing greater configuration flexibility (for instance, restricting access to a resource to the user that created it and not the project/tenant). This will make it easier to keep searchlight deployments in sync with the rules deployed with each service.

5.2.3 References

[1] Oslo policy documentation: http://docs.openstack.org/developer/oslo.policy/api.html

5.3 Searching admin-only fields

https://blueprints.launchpad.net/searchlight/+spec/index-level-role-separation

Our aim is to allow all fields to be searchable and available in facets but only for users where that is appropriate; as such, we introduced the idea of filtering search results based on whether or not a user has the admin role.

The flaw that we discovered towards the end of Liberty is described in https://bugs.launchpad.net/searchlight/+bug/1504399, but very briefly, merely removing fields from the result is not sufficient. It is possible to fish for values for known fields by running searches against them and examining whether results come back; an attacker might use range or wildcard queries to reduce the time it takes to locate values that return results.

5.3.1 Problem Description

We wish to allow plugins to define fields (whether in code or in configuration) that cannot be seen by non-administrative users, be that in search results, visible in facets or by searching for values for a field. Administrators should be subject to none of these restrictions.

Prior to the fix in bug #1504399 Searchlight fulfilled the first two of these criteria. Unfortunately the fix (which was under a very tight time restriction) prevented even administrators from searching fields. The problem, therefore, is to ensure these conditions.

5.3.2 Proposed Change

Role-based filtering

This solution involves indexing twice and adding a field to all resources that can be used to filter a search based on a users role. For instance, taking a heavily cut-down Nova server definition:

```
{
  "_id": "aaaaabbbb-1111-4444-2222-eeee",
  "_type": "OS::Nova::Server",
  "_source": {
     "status", "ACTIVE",
     "OS-EXT-ATTR-SOMETHING": "admin only data"
  }
}
```

This is turned into two documents, identical except that: * the admin-only document has an additional field *user-role: admin* * the user document has an additional field *user-role: user* * the user document does not contain the *OS-EXT-ATTR-SOMETHING* field * the ids for each document have a role added (*111111-4444-2222-eeee:ADMIN*)

Indexing

Indexing operations are unchanged except that two operations (or one bulk operation) are needed. Admin-only fields would be stripped from the serialized source document for the non-admin copy:

```
{
  "_id": "aaaaabbbb-1111-4444-2222-eeee_ADMIN",
  "_type": "OS::Nova::Server",
  "_source": {
    "status", "ACTIVE",
    "OS-EXT-ATTR-SOMETHING": "admin only data",
    "_searchlight_user_role": "admin"
  }
},
{
  "_id": "aaaaabbbb-1111-4444-2222-eeee_USER",
  "_type": "OS::Nova::Server",
  "_source": {
    "status", "ACTIVE",
    "_searchlight-user-role": "user"
  }
}
```

This solution allows resources that dont need an admin/user separation to index a single document with both roles:

```
{
  "_id": "abcdefa-1222",
  "_type": "OS::Designate::Zone",
  "_source": {
    "_searchlight-user-role": ["admin", "user"]
  }
}
```

Searches

The server can apply a non-analyzed (term) filter on *_searchlight-user-role* based on the request context:

```
{
  "query": {... },
  "filter": {"term": {"_searchlight-user-role": "admin"}}
}
```

Filters are cached and very fast. An alternative, once we switch to using aliases (see the zero downtime spec proposal), is applying the filter on the alias:

```
{
  "index": "searchlight-<timestamp>",
  "alias": "searchlight-admin",
  "filter": {"term": {"_searchlight-user-role": "admin"}}
}
```

The search API would query against *searchlight-admin* or *searchlight-user* as appropriate. There is some precedent for this; its a common way to make data appear to be segmented based on a field (index per user - Reference) without the overhead of multiple lucene indices.

Second choice - Separate indexes

Note: This began as my frontrunner, but the added maintenance headache has pushed me towards filter-based separation.

Another solution is to maintain separate indices for admin and non-admin users. While this seems offensive from a duplication point of view, its very common in non-relational-databases to store information based on the kinds of queries you want to run. There will be an impact on indexing speed and data storage, though I believe the volume and throughput of data we store makes this impact insignificant. The major downside is the increased maintenance overhead (at a minimum, two indices would be required at least for those plugins requiring it).

Technically, introducing a pair of indices isnt terribly complicated; all write operations become two, and searches determine which index theyre using before running. As far as a user sees, there will be no impact (except that admins will once again be able to run searches against admin-only fields).

Indexing

Information in the -users index can be restricted with dynamic_mapping template (that can tell Elasticsearch not to store or index matching fields with *index:no* and *include_in_all:no*). Along with result filtering (or *_source* filtering or removing these fields from the indexed document) this achieves all three requirements.

Some plugins do not have admin-only fields, and those plugins could run against the same index. I believe, though, that it would be necessary to use a separate shared index in that case, because otherwise a query could potentially run against (say) *OS::Nova::Server* in both indices. For example, the structure below assumes *OS::Something::Else* doesnt need two indices, and all data is in the user index:

```
searchlight-admin:
   OS::Nova::Server
searchlight-user:
   OS::Nova::Server
   OS::Something::Else
```

An admin query against both types would have to run against both indices, running the risk of duplicate results for *OS::Nova::Server* resources. This might need more discussion, but safer would be to either mandate storing information twice for all types, or:

```
searchlight-admin:
   OS::Nova::Server
searchlight-user:
   OS::Nova::Server
searchlight-all:
   OS::Something::Else
```

Searches

Little would change as far as a user is concerned. The search code would have some extra conditionals in it to determine which index to use. This would be complicated if an index contains both admin- and non-admin- data.

Alternatives

There are two other alternatives Im aware of.

1. Elasticsearch Shield. Shield adds a number of features to Elasticsearch, all aimed at security and authentication. One of those features (supported only by Elasticsearch 2.0) is field level access control. This requires an inclusive list of fields to be given in configuration on a per-index basis, and also requires Shields authentication to be enabled (there are various plugins available). It disables the *_all* field for users who are subject to field level restrictions.

   Most importantly, Shield is a commercial, closed-source product that runs on the server, and so is able to do things we are not (since it has access to the parsed query).

2. Modify or reject incoming queries. We already strip certain fields from search results for non-admin users, and in theory we could restrict searches in the same way (or raise Not Authorized exceptions). While naively this seems straightforward, in reality it becomes complex quite quickly. Imagine the following queries against Nova for a protected field called *hypervisor_id*:

   ```
   {"query": {"term": {"hypervisor_id": "abcd1"}}}
   {"query": {"query_string": {"query": "hypervisor_id:abcd1"}}}
   {"query": {"multi_match": {"query": "abcd1", "fields": ["hypervisor_id"]}}}
   {"query": {"query_string": {"query": "abcd1"}}}
   ```

   Constructing filters to catch those queries isnt impossible, but becomes increasingly complex; we would essentially need to parse the query, and wed need to do so for each plugin type.

5.3.3 References

- https://bugs.launchpad.net/searchlight/+bug/1504399
- https://review.opendev.org/#/c/233225/ (patch for above)
- Shield
- https://www.elastic.co/guide/en/elasticsearch/guide/current/faking-it.html

5.4 Zero Downtime Re-indexing

https://blueprints.launchpad.net/searchlight/+spec/zero-downtime-reindexing

This feature enables seamless zero downtime re-indexing of resource data from an API users point of view.

5.4.1 Problem Description

As a user of the searchlight API, we expect the following traits:

- The index is up to date and coherent with the source data
- The index is available
- That we are not affected by updates and upgrades to the searchlight service

As a deployer, we expect the following:

- That we can roll out service upgrades and update the index with new data
- That we can bring the index back into coherency without downtime
- That we can tune the service deployment according to performance needs
- That we can have easy deployment of new / patched plugins
- That we can change data mappings and re-index the data

Background

ElasticSearch documents are stored and indexed into an index (imagine that). The index is a logical namespace which points to primary and replica shards where the document is replicated. A shard is a single Apache Lucene instance. The shards are distributed amongst nodes in a cluster. API users only interact with the index and are not exposed to the internals, which ElasticSearch manages based on configuration inputs from the administrator.

Certain actions can only be done at index creation time, such as changing shard counts, changing the way data is indexed, etc. In addition to changing the data, re-populating an index that has lost coherency with source service data is much easier to do from scratch rather than determining what differences there are in the data. Due to this the data and indexes should be designed so that it is possible to re-index at any time without disruption to API users. The re-indexing happens while the services are in use, still indexing new documents in ElasticSearch.

In Searchlight 0.1.0, we allowed for each plugin to specify the index where the data should be stored via configuration in the searchlight-api.conf file. By default, all plugins store their data in the searchlight index. This was simply chosen as a starting point, because the amount of total data indexed for resource instance data is believed to be quite small in comparison to typical log based indexing for small deployments, but this may differ dramatically based on the resource type being indexed and the size of the deployment.

To reiterate, all resource types in Searchlight 0.1.0 (either the plug-in or the searches) have the ElasticSearch index hard-coded into them. This hard-coded functionality prevents Searchlight from doing smart things internally with ElasticSearch. Exposing indexes directly to the users is generally not recommended by the user community or by ElasticSearch. Instead, they recommend using aliases. API users can use an alias in exactly the same way as an index, but it can be changed to point to different index(es) transparently to the user. This allows for seamless migration between indexes, allowing for all of the above use cases to be fulfilled.

The concept of aliases is described in depth in the ElasticSearch guides [1].

5.4.2 Proposed Change

With this blueprint, we will divorce the plug-ins and searches from knowing about physical ElasticSearch indexes. Instead we will introduce the concept of a Resource Type Group. A Resource Type Group is a collection of Resource Types that are treated as a single unit within ElasticSearch. All users of Searchlight will deal with Resource Type Groups, instead of low-level ElasticSearch details like an index or alias. A Resource Type Group will correspond to ElasticSearch aliases that are created and controlled by Searchlight.

The plug-in configuration in the searchlight-api.conf file will no longer specify the index name. Instead the plug-in will specify the Resource Type Group it chooses to be a member of. It is important for a plug-in to know which Resource Type Group it belongs. When some operations are undertaken by one member of a Resource Type Group, it will need to be done to all members in the group. There will be more details on this later.

Now that the users are removed from the internals of ElasticSearch, we can handle zero downtime re-indexing. The basic idea is to create new indexes on demand, populate them, but use ElasticSearch aliases inside of Searchlight in a way that makes the actual indexes being used transparent to both API users and Searchlight listener processes.

We will not directly expose the alias to API users. We will use resource type mapping to transparently direct API requests to the correct alias. When implementing this blueprint, we may choose to still expose an index through the plug-in API. Exposing an index may allow other open-source ElasticSearch libraries (which are index-based) to still work. Currently we are not using any of these libraries, but we may not want to exclude their usage in the future.

Searchlight will internally manage two aliases per Resource Type Group. Note: Having these two aliases is the key change enabling zero downtime indexing.

- API alias
- Listener/Sync alias

The names of the aliases will be derived from the Resource Type Group name in the configuration file. Exactly how this is handled will be left to the implementation. For example, we can append -listener and /Sync-search to the Resource Group Type name for the two aliases.

The API alias will point to a single live index and only be switched once the index is completely ready to serve data to the API users. Completely ready means that the new index is a superset of the old index. This allows for transparently switching the incoming requests to the new index without disruption to the API end user.

The listener alias will point to 1* indexes at a time. The listener simply knows that it must perform CRUD operations on the provided alias. The fact that it might be updating more than one index at a time is transparent to the listener. The benefit to this is that the listeners do not have to provide any additional management API as ElasticSearch handles this for us automatically.

The algorithm for searchlight-manage index sync will be changed to the following:

- Create a new index in ElasticSearch. Any mapping changes to the index are done now, before the index is used.
- Add the new index to the listener(s) alias. At this point, the listeners alias is pointing to multiple indexes. The new index is now live and receiving data. Any data received by the listener(s) will be sent to both indexes.
  - There is an issue with indexing an alias with multiple indexes [2]. The issue is that this case is not allowed! In this case we will catch the exception and write to both indexes individually in this step. For more details, refer to the Implementation Notes subsection below.
- Bulk dump of data from each Resource Type associated with the old index to the new index in ElasticSearch.
  - The same issue with multiple indexes mentioned above applies here also.
- **Atomically switch the aliases for the API alias to point to the new index.**
  - We will use the actions command with remove/add commands in the same actions API call. Elastic-Search treats this as an atomic operation. [2]:

```
{ "actions" : [ { "remove" : { ...} }, { "add" : " {...} } ] }
```

- Remove the old index from the listener(s) alias.
- Delete the old index from ElasticSearch. We do not want the index to hang around forever. We can figure out when the index is no longer being used and then delete it (asynchronous task, a type of internal reference count, etc). If this turns out to be too unwieldy we can revisit this action.

Notes:

- This algorithm assumes that we can handle out of order events. See below for more details.
- During the re-syncing process, the listener(s) will be adding any new documents to both indexes.
- The listeners will always keep the ElasticSearch index associated with the API alias up to date.
- The listeners will keep the old index up to date after the API alias has switched over to minimize any race conditions.

A critical aspect to all of this is that the batch indexer and all notification handlers MUST only update documents if they have the most recent data. This is being handled by a separate bug [3]. In addition, Searchlight listeners and index must start setting the TTL field in deleted documents instead of deleting them right away. This functionality is covered in the ES deletion journal blueprint [4].

We are operating on a Resource Type Group as a whole. We need to make sure that the entire Resource Type Group is re-indexed instead of just a single Resource Type within the group. For example, consider the case where a Resource Type Group consists of Glance and Nova. When Searchlight gets a command to re-index Glance, Searchlight needs to also re-index Nova. Otherwise the new index will not have the previous Nova objects in it. If Nova did not re-index, the new index will not be a superset of the old index. When the alias switches to this new index it will be incomplete.

The CLI must support manual searchlight-manage commands as well as automated switchover. For example:

- Delete the specified or current index / alias for a specific resource type group.
- Create a new index for the specified resource type group.
- Switch API and listener aliases automatically when complete (default - yes).
- Delete old index automatically when complete (default - yes).
- Provide a status command so that progress can be seen. * List all aliases and indexes by resource type with their status * Can be used from a GUI or a separate CLI concurrently to monitor progress.

This change affects:

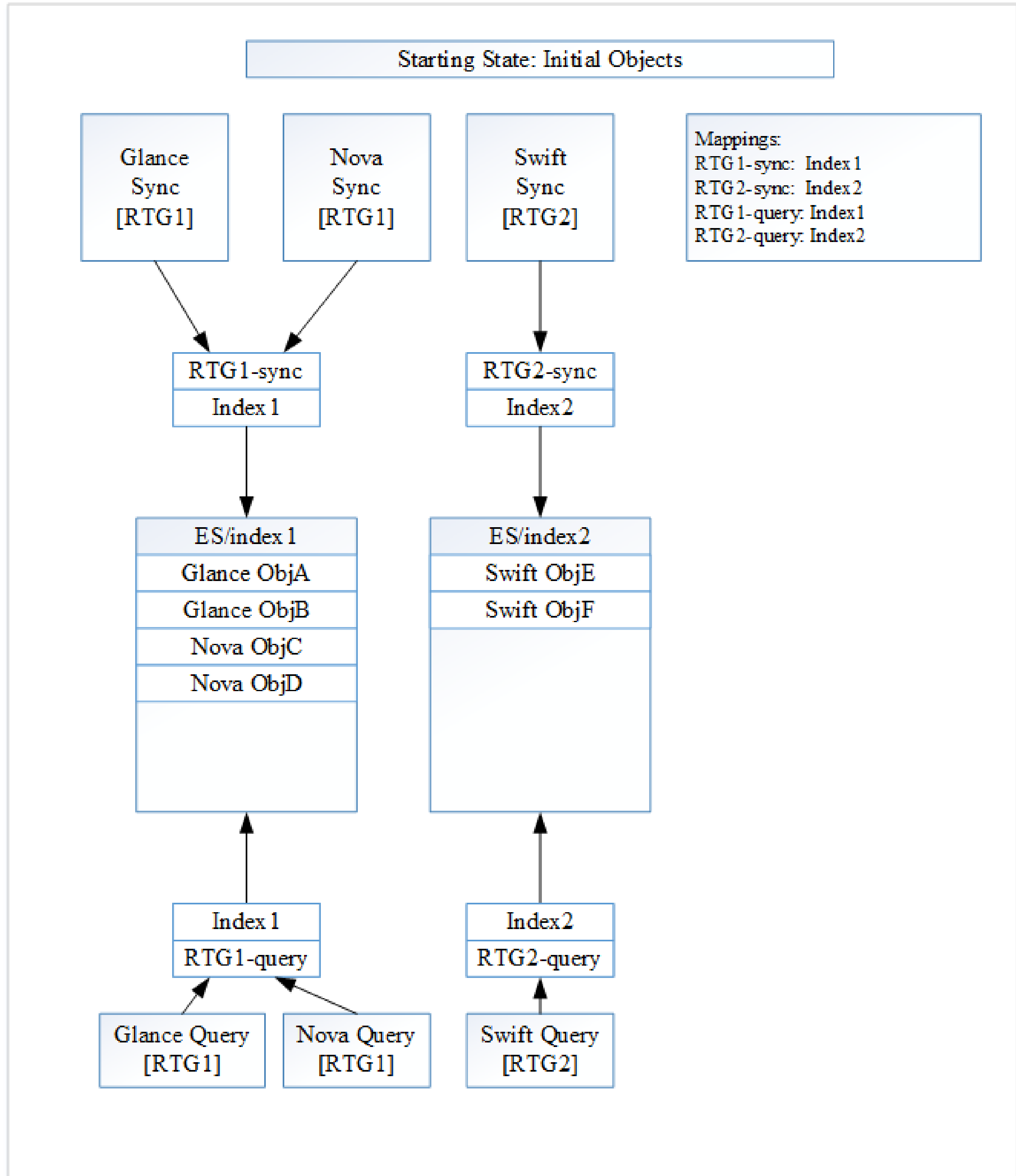
- The plugins API which lists plugins
- The API
- The Listener
- The bulk indexer
- The CLI

Illustrated Example

To further illuminate the blueprint we will turn to a series of images and save ourselves thousands of words. The images shows the state of Searchlight during sequence of operations.

For this example we have three resource types: Glance, Nova and Swift. There are two Resource Type Groups. The first group, RTG1, contains Glance and Nova. The two aliases associated with RTG1 are RTG1-sync for the plug-in listeners and RTG1-query for the plug-in searches. The second group, RTG2, contains Swift. The two aliases associated with RTG2 are RTG2-sync for the plug-in listener and RTG2-query for the plug-in search.

Figure 1: The initial State



First Searchlight will create the ElasticSearch index Index1 for use by RTG1. The ElasticSearch aliases RTG1-sync and RTG1-query are created and will both be associated with the index index1. Next Searchlight will create the ElasticSearch index Index2 for use by RTG2. The ElasticSearch aliases RTG2-sync and RTG2-query are created and will both be associated with the index Index2.

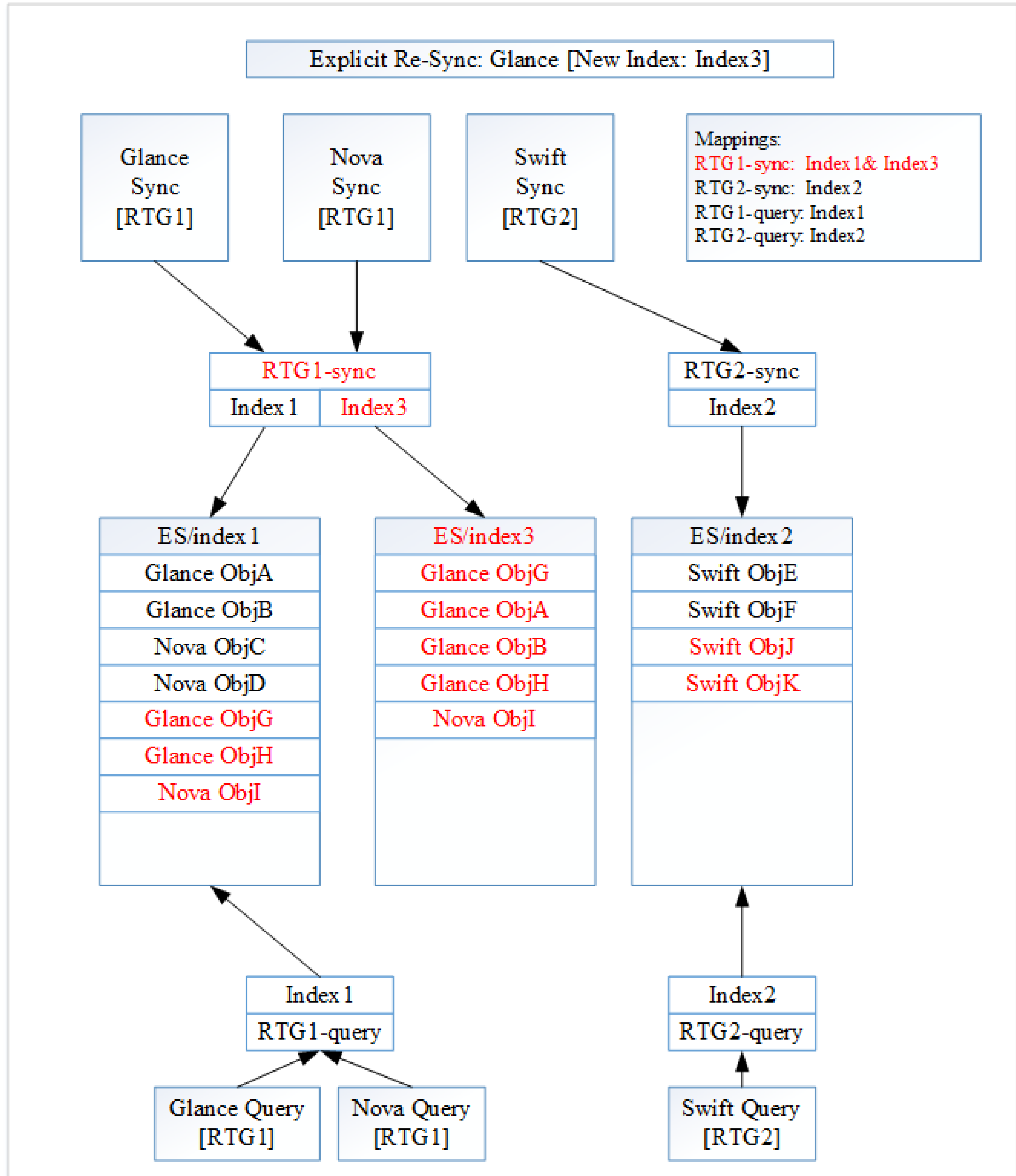
Glance has now created two documents Glance ObjA and Glance ObjB. Nova has created two documents Nova ObjC and Nova ObjD. These four new documents for the first Resource Type Group are now indexed. They will be indexed against alias RTG1-sync and end up in index Index1.

Swift has now created two new documents Swift ObjE and Swift ObjF. These two new documents for the second Resource Type Group are now indexed. They will be indexed against alias RTG2-sync and end up in index Index2.

Figure 1 shows the current state of Searchlight.

A Glance search will be made against RTG1-query. Going to Index1 it will return Glance ObjA, Glance ObjB, Nova ObjC and Nova ObjD. A Swift search will be made against RTG2-query. Going to index2 it will return Swift ObjE and Swift ObjF.

Figure 2: Explicit Glance Re-sync



All of the changes from Image 1 are highlighted in red.

Searchlight receives a re-index command for Glance. After the re-sync, Glance creates two new documents Glance ObjG and Glance ObjH. Nova creates one new document Nova ObjI. Swift creates two new documents Swift ObjJ and Swift ObjK.

Searchlight will create a new ElasticSearch index Index3. Since Glance is re-syncing, the new index is associated with RTG1. Searchlight now associates both Index1 and Index3 to the alias RTG1-sync. Since the new index Index3 is not a superset of the index Index1 yet, we do not change the RTG1 search alias RTG1-query. It remains unchanged for

now.

As the Glance re-sync occurs, the previous Glance documents Glance ObjA and Glance ObjB get indexed into Index3. The new documents for RTG1 (Glance ObjG, Glance ObjH and Nova ObjI) are indexed against the alias RTG1-sync. These documents end up in both Index1 and Index3.

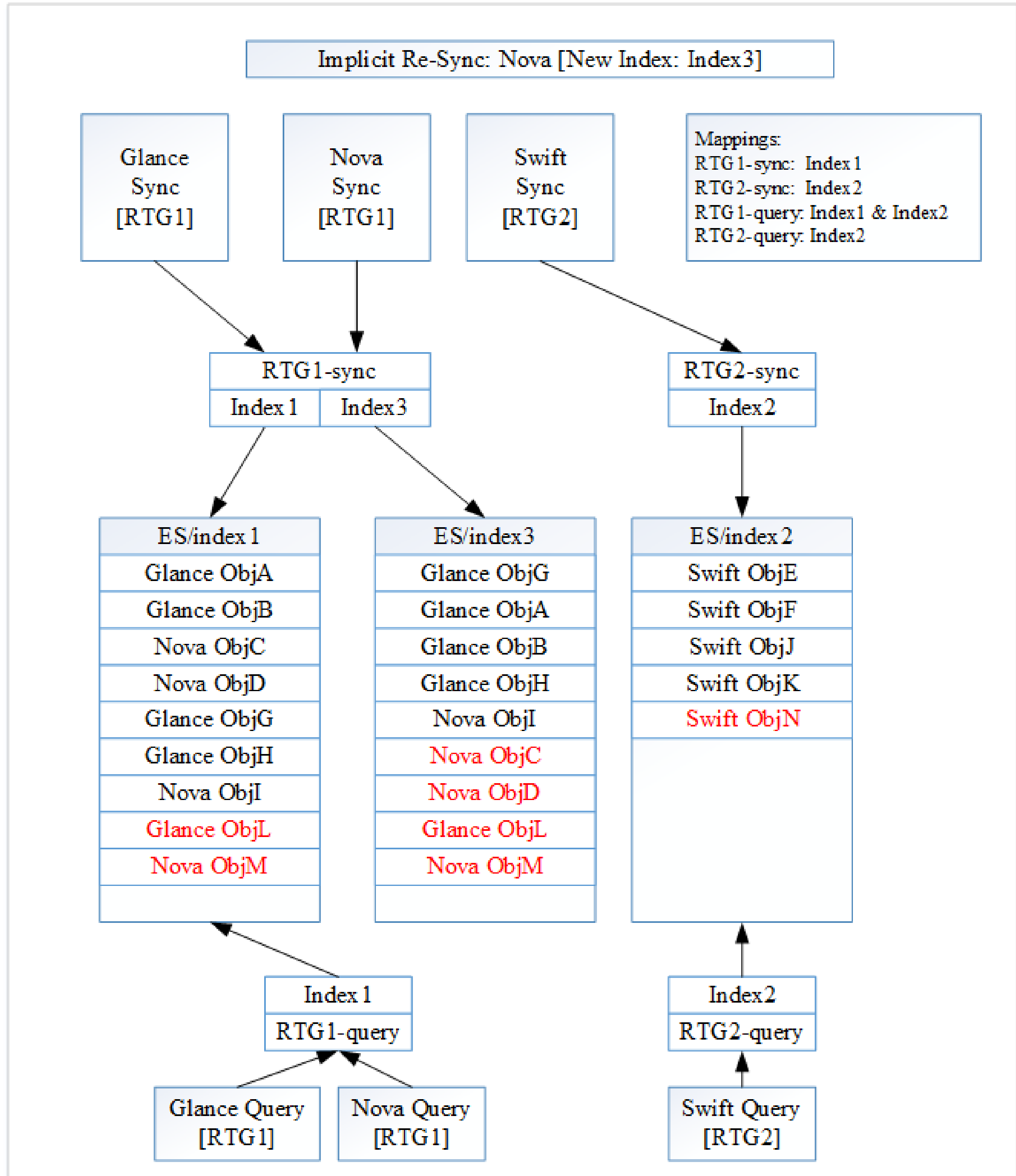
The new documents for RTG2 (Swift ObjJ and Swift ObjK) are indexed against the alias RTG2-sync. These documents end up in Index2.

Figure 2 shows the current state of Searchlight.

A Glance search will be made against RTG1-query. Going to Index1 it will return Glance ObjA, Glance ObjB, Nova ObjC, Nova ObjD, Glance ObjG, Glance ObjH and Nova ObjI. A Swift search will be made against RTG2-query. Going to index2 it will return Swift ObjE, Swift ObjF, Swift ObjJ and Swift ObjK.

This diagram shows the subtle point that all resource types within a Resource Type Group need to re-synced together. If we did not re-sync Nova and updated the RTG1 search alias RTG1-query to be associated with the new index Index3, the Searchlight state is incorrect. A Glance search will now be made against Index3 and it will return Glance ObjA, Glance ObjB, Glance ObjG, Glance ObjH and Nova ObjI. This is incorrect as it does not include the earlier Nova documents: Nova ObjC and Nova ObjD. This incomplete state is the reason that all resources in a Resource Type Group need to be re-synced before the Resource Type Group re-sync is to be considered completed.

Figure 3: Implicit Nova Re-Sync



All of the changes from Image 2 are highlighted in red.

Searchlight starts an implicit Nova re-sync, since Nova is a member of RTG1. All of the aliases are still set up correctly, so they do not need to change. After the re-sync, Glance creates one new document Glance ObjL. Nova creates one new document Nova ObjM. Swift creates one new documents Swift ObjN.

As the Nova re-sync occurs, the previous Nova documents Nova ObjC and Nova ObjD get indexed into Index3. The new documents for RTG1 (Glance ObjL and Nova ObjM) are indexed against the alias RTG1-sync. These documents end up in both Index1 and Index3.

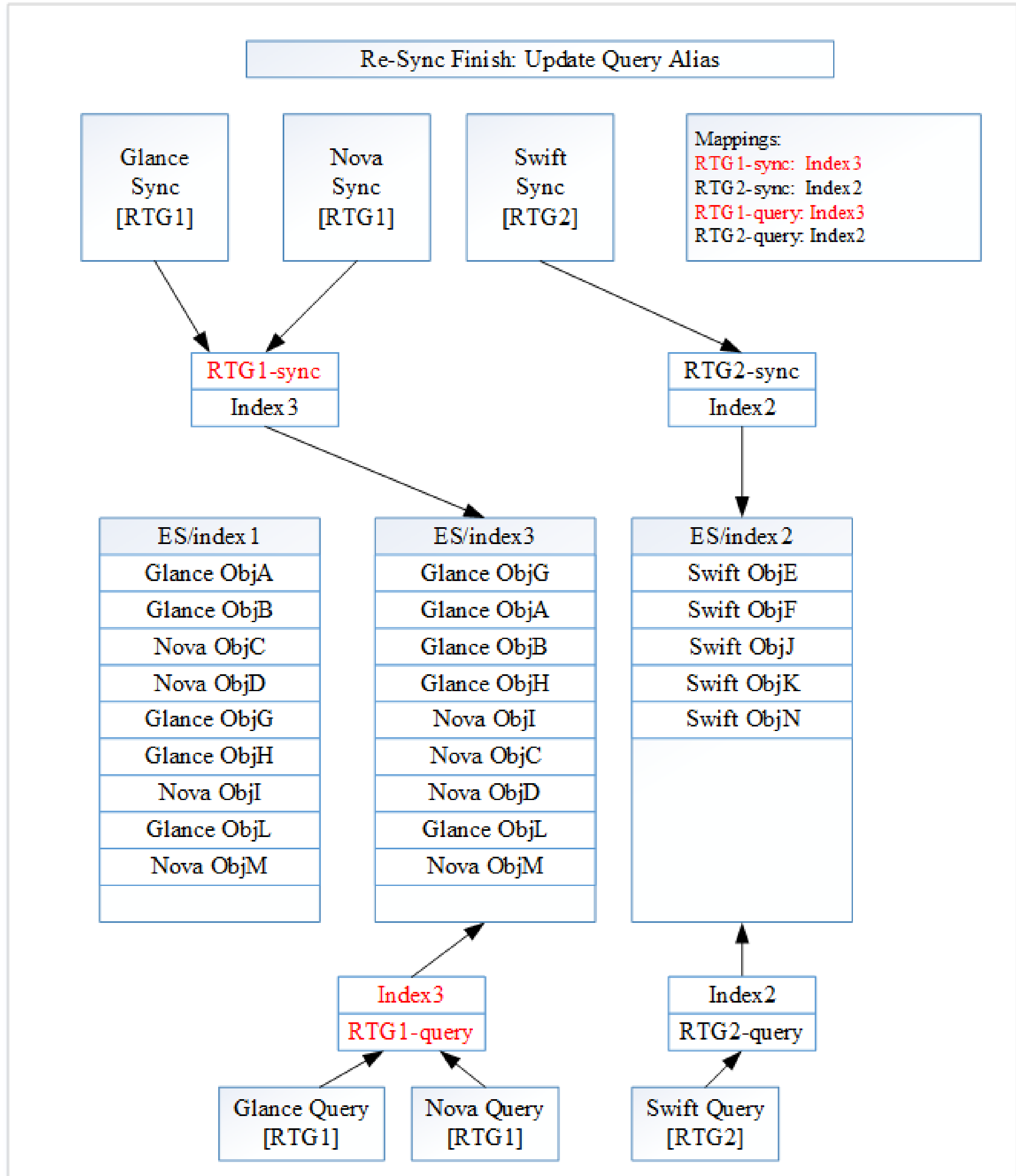
The new document for RTG2 (Swift ObjN) is indexed against the alias RTG2-sync. This document ends up in Index2.

Searchlight has not yet acknowledged the Nova re-sync as being completed. Therefore RTG1-query has not been updated yet.

Figure 3 shows the current state of Searchlight.

A Glance search will be made against RTG1-query. Going to Index1 it will return Glance ObjA, Glance ObjB, Nova ObjC, Nova ObjD, Glance ObjG, Glance ObjH, Nova ObjI, Glance ObjL and Nova ObjM. A Swift search will be made against RTG2-query. Going to index2 it will return Swift ObjE, Swift ObjF, Swift ObjJ, Swift ObjK and Swift ObjN.

Figure 4: RTG1 Re-Sync Complete



All of the changes from Image 3 are highlighted in red.

All resource types within RTG1 have finished re-syncing. Searchlight will now update the RTG1 search alias RTG1-query. The alias RTG1-query will now be associated with index Index3. After updated the RTG1 search alias, Searchlight will update the RTG1 plug-in listener alias RTG1-sync. The alias RTG1-sync will now be associated with the index Index3.

The alias updates need to happen in this order to handle the corner case of a new RTG1 document being indexed while the aliases are being modified. If we modified the RTG1 plug-in listener alias first a new document would be indexed

to index Index3 only. But a search will still go to index Index1, thus missing the newly indexed document.

Figure 4 shows the current state of Searchlight.

A Glance search will be made against RTG1-query. Going to Index3 it will return Glance ObjA, Glance ObjB, Nova ObjC, Nova ObjD, Glance ObjG, Glance ObjH, Nova ObjI, Glance ObjL and Nova ObjM. A Swift search will be made against RTG2-query. Going to index2 it will return Swift ObjE, Swift ObjF, Swift ObjJ, Swift ObjK and Swift ObjN.

The internal Searchlight state is correct, coherent and ready to continue. Sometime in the future we will be able to delete Index1 completely.

Implementation Notes

Implementation Note #1: Multiple Indexes

Upon careful review of the ES alias documentation [2], there is this warning lurking in the shadows: It is an error to index to an alias which points to more than one index. Yikes. Now the simple solution of adding additional indexes to an alias and having the re-indexing just work, will not work. ElasticSearch will through an ElasticsearchIllegalArgument exception and return a 400 (Bad Request).

The plug-ins will need to be aware of this exception and react to it. Through experimentation, ElasticSearch will return this error:

```
{"error":"ElasticsearchIllegalArgumentException[Alias [test-alias] has more than one␣
↪indices associated with it [[test-2, test-1]], can't execute a single index op]",
↪"status":400}
```

From this error message, we have the actual indexes. After extracting the names of the indexes, the plug-ins will be able to complete the task. The plug-in will now index iterating on each real index, instead of using the alias. This case applies only to the case where there are multiple indexes in an alias (i.e. the re-syncing case). When not re-syncing, the plug-in will not receive this exception.

We need to be careful when parsing the error message. This is a potential hazardous area if the error message ever changes. The catching of the exception and parsing of the message should be as flexible as possible.

Implementation Note #2: Incompatible Changes

A corner case in the rationale for triggering a re-index needs to be addressed. Sometimes an incompatible change between indexes has occurred. For example a new plug-in has been added or the documents from the service of changed in an incompatible way (different ElasticSearch mapping). In any of these cases we need to be able to handle the changes and roll them out seamlessly.

Some possible options to handle these cases would include:

- Disable re-indexing into the old index.
- Run two listeners, one understanding the old index and the other understanding the new index.

Alternatives

Alternate #1

An alternate usage scenario would look like the following:

Queries to v1/search/plugins would change so that the index listed for each type would actually be the alias (the API user wont know this).

The searchlight-manage index sync CLI will change to support the following capabilities:

- Re-index the current index without migrating the alias (no change from 0.1.0).
- Delete the specified or current index for a specific type.
- **Create a new index for specified resource types.**
  - Specified name or autogenerated name using a post-fix numbering pattern.
  - Contact and stop all listeners from processing incoming notifications for specified types.
  - Switch alias automatically when complete (default - no ?).
  - Delete old index automatically when complete (default - no?).
  - Contact and start all listeners to process incoming notification for specified types.
- Switch alias on demand to new index(es).

All of the above must account for 1 * indexes for a single alias.

All listener processes must now support a management API for them to stop notification processing for specified resource types. Without this ability, there will remain a race condition for populating a new index. For example, if it takes N seconds to populate all Nova server instances, there will be a delay in time from when the original request for data to Nova was sent and when any updates to the data happened. Therefore, notification should be disabled while a new index is being populated and then turned back on.

Alternate #2

This alternate explores a way to avoid the multiple indexes in a single alias while indexing exception as described in the Implementation Notes subsection.

The idea is that instead of having two indexes in the Sync alias and one index in the search alias, we invert the index usage in the aliases. Now we consider adding multiple indexes to the search alias while leaving a single index in the sync alias.

When we start a re-sync, we create a new index. We update the sync alias to point to this new index, replacing the old index. Since there is only a single index in the sync alias, we will not get the ElasticsearchIllegalArgument exception. We also add the new index to search alias.

At this point, the search alias contains just the new index while the search alias contains both the old and new index. When a search occurs it will find old documents as well as any new documents.

The main issue with this alternative is that the search will find a lot of duplicates while the re-sync is occurring. All of the documents in the old index will eventually be added to the new index. In order to be usable, we would need to figure out a way to filter out these duplicates. The initial investigation into filtering ideas led to solutions that were deemed to fragile and defect prone. Hence the inclusion of this idea at the bottom of the alternate proposals.

Future Enhancements

Optimizations:

- Use the ElasticSearch index sync functionality instead of having each Resource Type do a manual re-index. ElasticSearch does not have a native re-sync command, but it can be accomplished using scan and scroll with the ElasticSearch Bulk API. [5] This optimization needs to be carefully considered. It would only be performed when we are absolutely sure that the old ElasticSearch index is coherent and complete.

5.4.3 References

[1] The concept of aliases is described in depth here: https://www.elastic.co/guide/en/elasticsearch/guide/current/index-aliases.html

[2] How ES treats an alias is described here: https://www.elastic.co/guide/en/elasticsearch/reference/1.7/indices-aliases.html

[3] All searchlight index updates should ensure latest before updating any document https://bugs.launchpad.net/searchlight/+bug/1522271

[4] ES deletion journal blueprint: https://blueprints.launchpad.net/searchlight/+spec/es-deletion-journal

[5] ES scan and scroll is discussed here: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-scroll.html ES Bulk API is discussed here: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html