

# RFC: HCA Data Store Reindexing - DEVELOP

*DEVELOP - This is the current “DEVELOP” version of this specification, please use comments or “suggesting” mode only.*

## RFC: HCA Data Store Reindexing DEVELOP

### Use Cases

[Verify](#)

[Repair](#)

[Reindex](#)

[Stand-up new replicas](#)

[Handle diverse metadata schemata](#)

### Requirements

[Concurrent submissions](#)

[Concurrent deletions](#)

[Concurrent searches](#)

[Concurrent notifications](#)

[Concurrent subscriptions](#)

[Missed notifications](#)

### Design Decisions

[Introduce index generations](#)

[Use Elasticsearch aliases](#)

[Disallowing more than one maintenance operation at a time](#)

[Handling concurrent submissions and notifications](#)

[Disabling concurrent deletions during repair and index](#)

[Using AWS Step Functions for batch processing](#)

### Admin API signatures

[create\\_index\\_maintenance\(...\)](#)

[get\\_index\\_maintenance\(operation\\_id: str\)](#)

[delete\\_index\\_maintenance\(operation\\_id: str\)](#)

### Architecture

[Reindexing operation](#)

[Verify and repair operations](#)

[Failure Handling](#)

[Cancelling operations](#)

[Indexing specific bundles](#)

[Logging](#)

[Internal functions](#)

[create\\_index\\_maintenance\(...\) → operation\\_id](#)

[IndexMaintenanceVisitation.initialize\(...\)](#)

[prepare\\_index\\_maintenance\(replica\) → generation\\_id](#)

[index\\_bundle\(mode, replica, bundle\\_id\[, generation\\_id\]\) → generation\\_id](#)

[complete\\_index\\_maintenance\(replica\)](#)

[get\\_index\\_maintenance\(operation\\_id\)](#)

[delete\\_index\\_maintenance\(operation\\_id\)](#)

[index\\_maintenance\\_cancelled\(\)](#)

[Appendix](#)

[Alternative ways to handle concurrent notifications](#)

[Alternative 1: Allow concurrent notifications](#)

[Alternative 2: Defer submissions](#)

[Implementation plan](#)

[Future extensions](#)

## Use Cases

### Verify

A system outage or human error is suspected to have caused the Elasticsearch index to become inconsistent with the primary storage in a replica, e.g. the bucket in S3. The potential inconsistencies are

- Stale bundles (document in index is out of date)
- Missing bundles (document is missing from index)
- Ghost bundles (document in index refers to non-existing bundle)

To estimate the degree of inconsistency, an index verification operation is initiated administratively. That operation reports inconsistencies only and does not modify the Elasticsearch index.

## Repair

A system outage or human error has caused the index to become inconsistent with primary storage in a replica. To fix the inconsistencies, a repair operation is initiated administratively in order to fix either a single stale/missing bundle, a set of specific stale/missing bundles or all bundles, and optionally all ghost bundles. The repair modifies the Elasticsearch index in place.

## Reindex

The Elasticsearch index is missing, empty, has been corrupted to a degree that it can't be relied on or a new software release of the data store fundamentally<sup>1</sup> changed the way documents are stored in the index. A reindex operation is initiated administratively. The reindexing does not modify the index in place. Instead it works on an offline copy that's atomically brought online at the end.

## Stand-up new replicas

When bringing up an entirely new replica from scratch, the Elasticsearch index has to be populated with the bundles in storage for that replica. This is very similar to the reindexing use case.

## Non-use case: Handle diverse metadata schemata

Support for multiple metadata schemata (or schema versions) does **not** require reindexing. Blue-box supports those by maintaining separate document indexes in Elasticsearch. That touches on reindexing only inasmuch as that reindexing has to populate those indexes. A single reindexing operation will populate the schema-specific indices simultaneously.

## Requirements

Each use case involves a distinct type of operation: verify, repair and reindex. We'll refer to these operations as *index maintenance operations*.

Below are the non-trivial requirements satisfied by those operations. The word "should" indicates an optional requirement while "must" indicates a mandatory one.

## Concurrent submissions

Submission of new bundles **must** be allowed while index maintenance is in progress.

---

<sup>1</sup> A fundamental change is one that affects the shape of the documents significantly enough to cause the Elasticsearch mapping (aka schema) to reject those documents. Note that this is different to a change of the metadata schemata which should not require reindexing.

## Concurrent deletions

Administrative deletion of bundles **should** be allowed while index maintenance is in progress.

## Concurrent searches

Searches **must not** be affected by an ongoing verification operation. An ongoing repair operation may affect searches insofar as that search results will increasingly reflect the repairs as the operation progresses. Once a reindexing operation finishes, searches hit the new index. Concurrent searches **should** include concurrent submissions and deletions.

## Concurrent notifications

Existing subscriptions **should** be notified for concurrent additions. Notifications **must not** happen as pre-existing bundles are repaired or re-indexed, unless the corresponding index document is missing (See Missed notifications). Notifications for concurrently submitted bundles **should not** happen more than once.

## Concurrent subscriptions

Subscription registrations **must** not be affected by ongoing index maintenance. Concurrently registered subscriptions **should** be notified about matching bundles submitted concurrently.

## Missed notifications

If the reindexing or repair process determines that a pre-existing bundle in the replica's storage bucket had not been added to the index and therefore a notification had not likely been sent, that missed notification **should** be sent as part of the reindexing or repair process.

# Design Decisions

## Introduce index generations

For documents—as well as for subscriptions—there currently is one Elasticsearch index per tuple (`replica`, `stage`). The upcoming support for handling multiple metadata schemas requires the inclusion of the metadata schema version in that tuple such that, at least for document indices, the tuple becomes (`schema`, `replica`, `stage`). Finally, to support concurrent searches, this specification adds the index generation as an ISO timestamp to the tuple for document indices (`generation`, `schema`, `replica`, `stage`). For any given combination of `stage` and `replica`, the following statements apply:

- More than one generation can exist but only one generation is ever actively used by searches.
- Typically (during normal operation), only one generation exists.

- During reindexing, two generations exist: the *old* generation that's actively being used by searches and the *new* generation currently being populated by the reindexing process.

## Use Elasticsearch aliases

Elasticsearch (ES) aliases are used to refer to the document indices in the active generation that is used by search requests. An ES alias is a symbolic reference to one or more indices. Aliases can be updated atomically allowing for seamless transition from one generation to the next. Activating a generation involves atomically updating the alias to refer to the indices in that generation thereby implicitly deactivating the previously active generation. The name of an alias is derived from the tuple (*replica*, *stage*). Each alias initially refers to indices matching ( $t_1, *, \text{replica}, \text{stage}$ ) where  $t_1$  is an ISO timestamp. After the first reindexing completes, that alias will refer to indices matching ( $t_2, *, \text{replica}, \text{stage}$ ) where  $t_1 < t_2$ .

In addition to the aliases that direct searches to the active generation, the system uses a special index to track the generation that new submissions are being indexed into. The so called generations index contains a single document with two fields:

- `generation_id` – the identifier of the generation used for indexing new bundle submissions
- `operation_id` – the identifier of the next available operation (see [see below](#))

Normally, the `generation_id` field references the same generation as the alias. The same is true for ongoing verification and repair operations. However, during a reindexing operation, the generations index refers to the new generation while the alias still refers to the old one.

## Disallowing more than one maintenance operation at a time

... is a cheap way to avoid race conditions. `create_index_maintenance()` raises an exception if this constraint would be violated.<sup>2</sup> The constraint is enforced by storing the next available operation ID in the generations index in Elasticsearch and relying on the uniqueness constraint enforced on execution names by AWS' `StartExecution` API which is used by the visitation (batch) code.

## Handling concurrent submissions and notifications

During verification and repair only one generation is active. That generation handles searches, repairs and concurrent submissions. Furthermore, notifications are driven by that generation as well, even during repair or verification. Consequently, notification endpoints that run Elasticsearch queries against the system are being served results consistent with the notification. In other words, if an endpoint gets notified about the addition of a bundle and in turn

---

<sup>2</sup> Note that running multiple concurrent REPAIR or VERIFY operations with disjunct date ranges (for bundle modification date) would be harmless, albeit inefficient. If that turns out to be a use case, it could be implemented more efficiently by allowing multiple data ranges per REPAIR or VERIFY operation.

uses the search API to query for that bundle, the bundle will be returned by that search. This applies to searches by bundle identifier as well as searches that match more than one bundle.

After a system software update to the indexer code it can't generally be assumed that the old index generation would accept the document shape emitted by the indexing function. This is because the function might have been updated to emit a document shape that's incompatible with the Elasticsearch mapping in the *old* generation. That's why the reindex operation populates a separate set of offline indexes i.e., the *new* generation. Submissions that happen concurrently to an ongoing reindex operation also end up in the new generation. If concurrent notifications were strictly required, even in that use case, those notifications would then have to be driven by the new generation. The problem with this approach is that notification endpoints that run ES queries against the system would keep searching the old generation yielding search results that are inconsistent with the notification: the bundle that is the subject of a notification would be missing from the results. This may be acceptable for endpoints that perform analysis tasks but we don't think it is for endpoints in user interfaces and portals. To accommodate this scenario, the system defers concurrent notifications until a reindexing is complete and the new generation is fully populated. See the appendix for alternatives to this design choice and their respective trade-offs.

Since documents are strictly derived, indexing is inherently idempotent. It would be harmless, for example, if a concurrent submission during REPAIR is picked up by a maintenance worker at the same time as it is handled by the regular indexer. One will overwrite the document written by the other, but with exactly the same content. However, to prevent both from considering the document for sending notifications, Elasticsearch's optimistic concurrency control (aka [versioning](#)) should be used. In the above case, both the worker and the regular indexer assume that they are adding a document. With versioning, one of them would fail to add the document and in turn skip considering notifications.

## Disabling concurrent deletions during repair and index

Concurrent deletions are subject to a race between indexer and reindexer. It's possible that the reindexer adds a bundle that the indexer just deleted.

## Using AWS Step Functions for batch processing

The *visitation batch system* visits each bundle once with a high throughput mechanism that can be throttled and monitored. This can be accomplished with a sentinel-worker threading pattern implemented with AWS Step Functions, a lightweight, serverless environment providing graceful error handling and monitoring.

The number of workers is set initially but can be dynamically adjusted as needed, ideally controlled by system load. It may also be desirable to throttle individual workers.

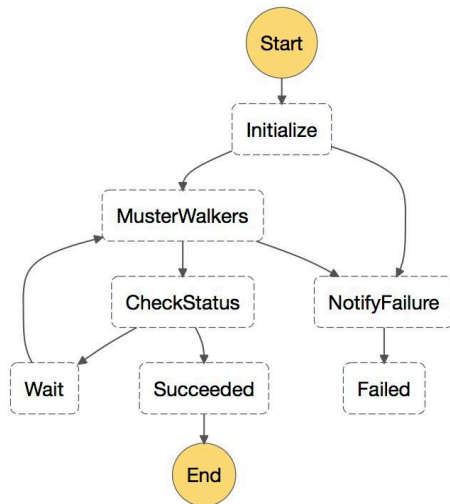
Although this mechanism is initially intended to back index maintenance operations, it can serve as a general batch processing platform for other DSS operations.

The Sentinel calls workers, and monitor for completion and failure.

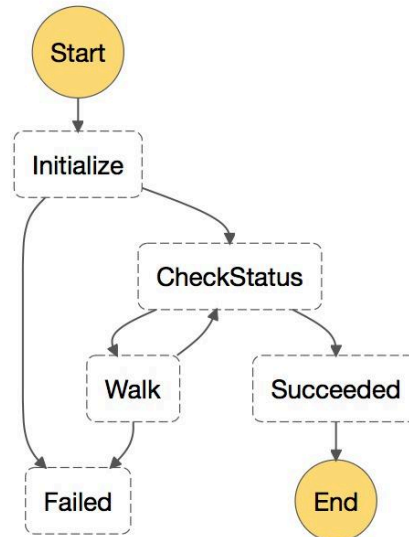
The worker visits each bundle once, calling the re-indexing function directly (as opposed to via a lambda), based on a two letter prefix.

Both are implemented as AWS Step Functions.

Sentinel Step Function diagram:



Walker Step Function diagram:



## Admin API signatures

These are AWS lambdas that can be invoked via an admin script.

### `create_index_maintenance(...)`

Initiate an index maintenance procedure and return a handle to it.

- Parameters
  - `mode`: enum, required, one of `VERIFY` | `REPAIR` | `REINDEX`  
Select maintenance mode. See section *Use Cases* above for description.
  - `replica`: str required  
The name of the replica to check on.
  - `bundles_added_after`: dateTime, optional, only allowed for `VERIFY` and `REPAIR`, no default. Restricts the maintenance operation to bundles submitted on or after the specified date. If absent, there is no lower bound on the bundle
  - `bundles_added_before`: dateTime, optional, only allowed for `VERIFY` and `REPAIR`. No default. Restricts the maintenance operation to bundles submitted before—but not on—the specified date.
- Return value

- `operation_id: str`  
An opaque handle to the asynchronous operation
- **Raises**
  - `ConflictException(operation_id: str)`  
A maintenance operation is currently ongoing.

`get_index_maintenance(operation_id: str)`

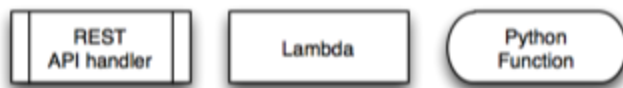
Return progress information about an ongoing index maintenance operation.

`delete_index_maintenance(operation_id: str)`

Cancel an ongoing index maintenance operation. VERIFY and REPAIR operations can be cancelled without detriment. Cancelling a REINDEX operation may cause bundles that were submitted during the REINDEX to be [missing](#) from search results. To fix this, another REPAIR or REINDEX operation must be started and left running to completion.



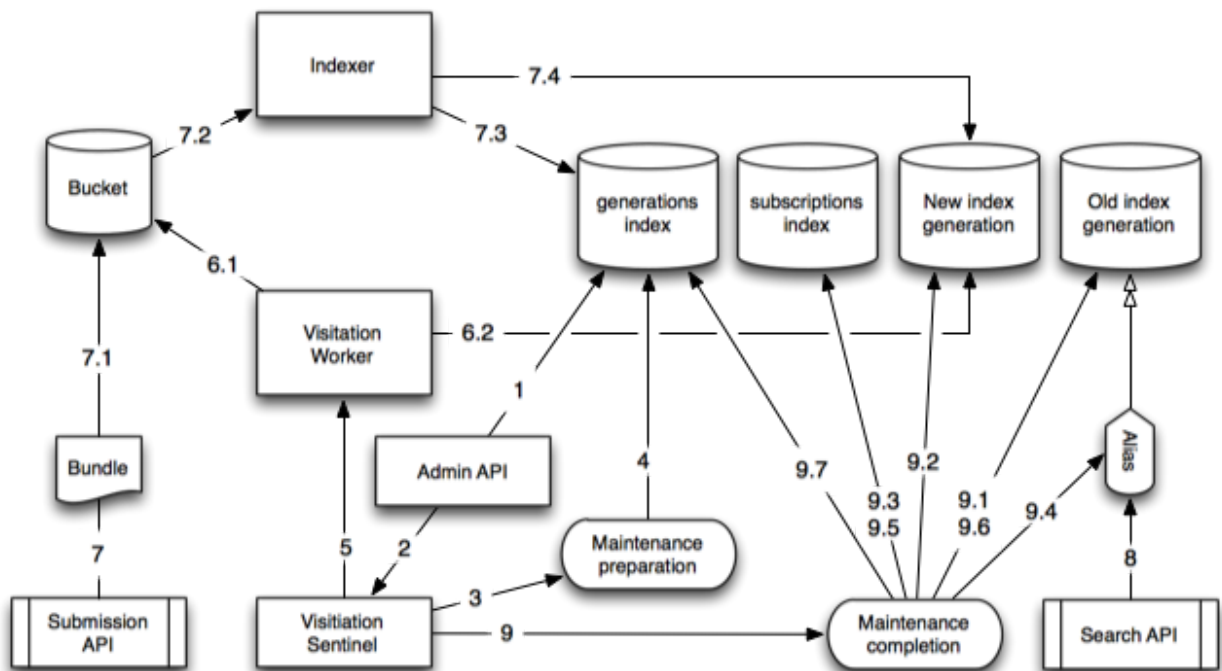
## Architecture



A REST API handler is a lambda that is invoked indirectly by way of a AWS API Gateway which handles REST requests. Regular lambdas are invoked directly through AWS Lambda service. The symbol for Python functions is used to call out specific functionality. These function will always run in the context of the lambda that invokes it.

## Reindexing operation

The diagram below illustrates the sequence of events during a REINDEX maintenance operation.



1. An admin initiates a maintenance operation by invoking the `create_index_maintenance(...)` lambda. The handler reads the current `operation_id` from the generations index.
2. The handler attempts to start an execution of the visitation sentinel state machine using the `operation_id` to name the execution. If that fails due to the uniqueness constraint on execution names, the handler raises an exception, indicating a conflict.
3. Otherwise, the sentinel lambda invokes the maintenance preparation function which ...
4. ... generates a new generation identifier and updates the generations index with it. This directs the indexer to add concurrently submitted bundles to the new generation.

5. The visitation sentinel launches a configurable # of worker state machines.
6. Each worker ...
  - 6.1. ... performs a distinct prefix query for a range of bundles in the bucket and invokes the `index_bundle` function for each resulting bundle
  - 6.2. When *reindexing*, the `index_bundle` function performs largely the same steps as during normal indexing (7.4) with the following differences:
    - Notifications are not sent out but deferred until the end of reindexing. This is done to avoid sending notifications for bundles in the new generation while the old generation is still being used by searches.
    - The generations index is not examined. Instead the target generation is passed in from the sentinel, through the worker.
7. Meanwhile,
  - 7.1. a new bundle is submitted concurrently
  - 7.2. The replica's storage bucket event triggers the index lambda, invoking the index function
  - 7.3. The index function examines the *generations* index and determines the appropriate document index to add the document to.
  - 7.4. It creates the index if necessary, composes the document representing the bundle and places the document into the index.
8. Concurrent searches are performed against the old generation.
9. Once the visitation sentinel determines that all workers are done processing their partitions of bundles from the replica's storage bucket, it invokes the maintenance completion function. The new index generation is now consistent with the bundles in storage. The maintenance completion function ...
  - 9.1. ... subtracts the set of bundle IDs in the old generation ...
  - 9.2. ... from that in the new generation:  $X = \text{new} - \text{old}$ . Set X contains ...
    - bundles that are present in the bucket but missing in the old index
    - bundles added concurrently during reindexing

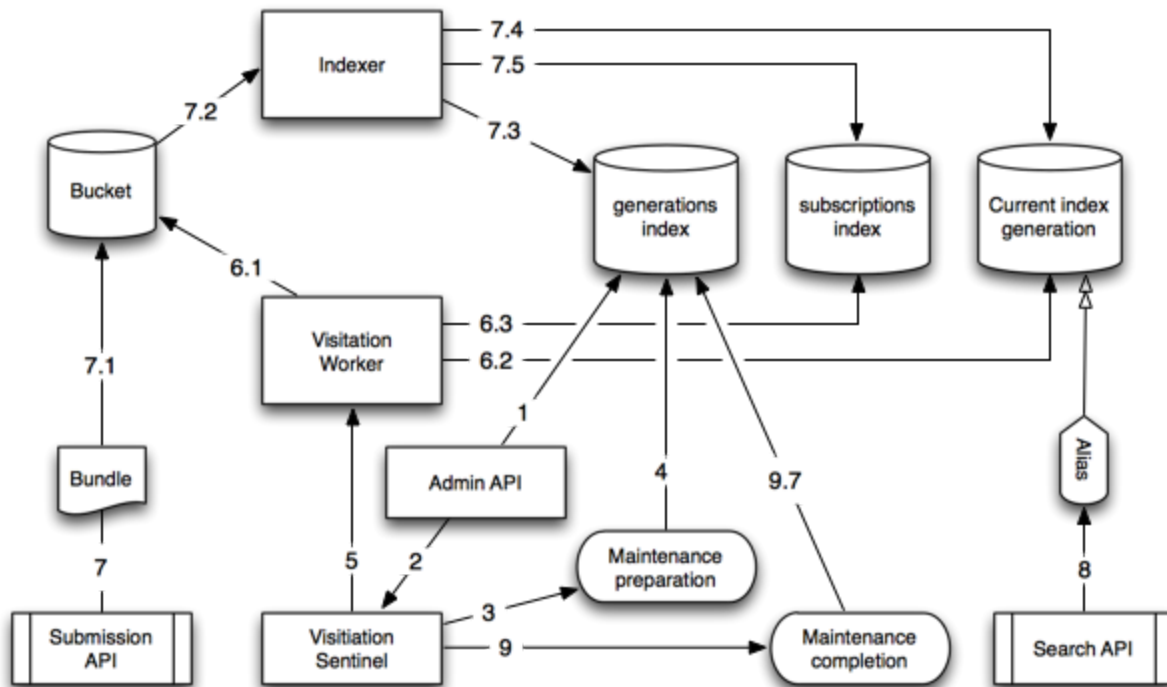
In either case, every bundle in that set should be considered for notifications because the system either *missed* considering the bundle for notifications or *deferred* doing so for concurrent submissions during reindexing.
  - 9.3. The maintenance completion function prepares the new generation of indices for subscription notifications by populating them with percolate queries from the subscriptions index<sup>3</sup>.
  - 9.4. It switches the alias to the new generation, ...
  - 9.5. ... notifies matching subscriptions about the bundles in set X, ...
  - 9.6. ... and drops the old generation of indices and finally
  - 9.7. ... increments the `operation_id` in the generations index, effectively opening the gate for subsequent maintenance operations.

---

<sup>3</sup> Doing this at the end and against a fully populated index avoid known issues with early registration of percolate queries on an empty index that uses dynamic mappings.

## Verify and repair operations

REPAIR and VERIFY mode follow a similar sequence as REINDEX.



The differences to REINDEX mode are as follows:

- Step 2 does not generate a new generation identifier but reads the current generation's identifier from the generations index instead.
- When verifying or repairing a bundle, the indexer
  - reads the current version of the corresponding document from the index and compares it with a newly composed one (6.1). If they are different or if the document is missing from the index, it will log the differences.
  - Additionally, REPAIR mode
    - overwrites the current document with the newly composed one (6.2)
    - and notifies any matching subscriptions (6.3).
- Step 7 (concurrent submissions) does not defer notifications but sends them out immediately (7.5).
- The maintenance completion function skips all 9.x steps except step 9.7 which increments the `operation_id`.

## Failure Handling

A failure of the index function should not be treated as catastrophic. In the initial implementation, a worker should retry the index function a fixed number of times for any given bundle. Later revisions could distinguish between fatal and retryable exceptions as well as let users specify a threshold for the number of bundles that are allowed to fail being reindexed before the repair or reindexing operation is automatically [cancelled](#).

## Cancelling operations

A maintenance operation can be cancelled administratively by invoking the DELETE /index/maintenance API. The corresponding DSS lambda invokes the AWS StopExecution API action to forcibly terminate any current executions of sentinel and worker state machines. It then waits long enough to be sure that all running step functions have returned and calls a function in the indexer code to remove left-overs like the new generation that was being populated by REINDEX mode.

Cancelling a REPAIR operation may, of course, leave a number bundles inconsistent or missing. but it will never increase that number.

There also is a caveat in cancelling a REINDEX operation: since it indexes concurrent submissions into the new generation and cancelling a REINDEX operation involves simply dropping that generation, concurrent submissions will be lost until a REINDEX is restarted. The lost submissions will eventually be indexed and, once the REINDEX completes, considered for notifications like any other missed notification. This could be avoided by adding concurrent submissions that occur during a REINDEX operation into both the old and the new generation but the main use case for reindexing is that the old generation cannot be populated by newly deployed code.

## Interactions with multi-schema support

Multi-schema support maintains multiple document indices per replica and stage. We refer That set of indices Things to watch out for

- ES cannot write to an alias that references multiple indices.

## Logging

During REPAIR and VERIFY, the bundle ID of every inconsistent bundle is logged at level WARN. The corresponding old and new documents will be logged at DEBUG. Workers should periodically log their progress, along with their unique identifier and the overall operation ID, such that log filtering and aggregation can be used to determine the overall progress manually.

For implementing `get_index_maintenance(operation_id)`, log filtering is not an option. The workers should persist their progress in a special ES index. Inconsistent bundles should also be persisted to that index, including the verbatim copies of the differing documents.

## Internal functions

`create_index_maintenance(...) → operation_id`

New. Implements steps 1 and 2. See [create\\_index\\_maintenance\(...\)](#).

`IndexMaintenanceVisitation.initialize(...)`

New. Implements steps 3, 5 and 9.

`prepare_index_maintenance(replica) → generation_id`

New. Implements step 4.

`index_bundle(mode, replica, bundle_id[, generation_id]) → generation_id`

Extracted from `process_new_indexable_object(...)`. Implements steps 6.{2,3} and 7.{3,4,5}.

- Eliminate `bucket_name` argument by inferring it from `replica` and `Config`.
- The `mode` argument is an enum of `INDEX`, `VERIFY`, `REPAIR` or `REINDEX`.
- The `generation_id` argument is disallowed for `INDEX` and required for the other modes.
- Invoked by the batch worker for `VERIFY`, `REPAIR` and `REINDEX` operations

`complete_index_maintenance(replica)`

New. Implements steps 9.\*.

`get_index_maintenance(operation_id)`

New. See [get\\_index\\_maintenance\(operation\\_id: str\)](#)

`delete_index_maintenance(operation_id)`

New. See [delete\\_index\\_maintenance\(operation\\_id: str\)](#)

## `index_maintenance_cancelled()`

New. Invoked by the sentinel after the workers are shut down. Notifies the index maintenance code that a maintenance operation has been cancelled. Makes sure that the cancelled operation leaves nothing behind.

## Appendix

### Alternative ways to handle concurrent notifications

This section is informational only.

#### Alternative 1: Allow concurrent notifications

- Pro: timely notifications
- Con: it would require pre-populating the mapping but that also has advantages

Currently an Elasticsearch index is created lazily when the first document is to be added to that index. Similarly, the *mapping*, i.e. the Elasticsearch schema governing the index is inferred from documents added to the index over time. The inferred mapping does not allow for field type changes, field deletions or renamed fields. That's why we need one index per metadata schema set version (different spec). To immediately notify about concurrent submissions that are only indexed into the new generation—because the old generation does not accept them, a possibility in the System Update use case—the new generation will need to support subscription notifications from the beginning. Notifications make use of percolate queries which in turn require the presence of an accurate mapping. Consequently, the indices in a new generation will need to be pre-populated with a mapping as well as the set of percolate queries for all currently registered subscriptions. This also means that all indices in the new generation would have to be prepopulated.

Deriving the Elasticsearch mapping from the bundle metadata schema is straight-forward because index documents are a simple combination of the metadata entities in a bundle.






Pre-populating mappings also has advantages:

- Mapping inferral can make the wrong type choice depending on the order in which documents are submitted. Prepopulating a mapping derived from the metadata schemas avoids this.
- Percolate queries can be registered right away and are type checked against a complete mapping. The inferred mapping may be immature and allow for queries that a complete mapping would reject.
- It's generally desirable to have a second line of defense against metadata errors. A pre-populated mapping that's derived from the metadata schemas could serve as that.
- OTOH: Templates can be used to steer the mapping inferral in the right direction

## Alternative 2: Defer submissions

Concurrent submissions could be held in a queue until reindexing finishes. From an external user's point of view this behaves similarly to deferred notifications. This queue would have to be in place in all use cases. In contrast, the system as currently specified here only defers notifications if the old generation can not accept concurrent submissions.

## Implementation plan

1.  [#610](#): This document
2. REPAIR and VERIFY
  -  Multi-schema support (pre-requisite, in progress, Trent, Michael)
    -  [#594](#): Document-related changes
    -  [#598](#): Subscription-related changes
  -  [#614](#): Batch visitation system (already in progress, Brian)
  - [#670](#): Remaining integration work (Hannes)
    - `create_index_maintenance()`
    - `prepare_index_maintenance()`
    - `index_bundle()`
    - `complete_index_maintenance()`
    - unit tests
3. REINDEX without deferring notification for concurrent submissions
  - Add "generations" index and have indexer read it
  - Switch new generation live by changing alias at the end of REINDEX
4. Send missed and deferred notifications
  - Add conditional to defer notification
  - At end of REINDEX, send deferred and missed notifications
5. `get_index_maintenance()`
6. `delete_index_maintenance()`

## Future extensions

- Handle concurrent deletes once the Admin Deletion feature is spec'ed. It may be unacceptable to delay admin deletes for several hours.
- Handle concurrent notifications, possibly at the expense of sending a minimal amount of duplicate notifications. Concurrent notifications could be sent out with a flag indicating their preliminary status.
- Gather consistency stats in special ES index
- Handle ghost bundles
  - Ghosts are documents for which there is no corresponding bundle in storage.
  - A REINDEX operation inherently fixes ghosts: when the old generation is dropped, potential ghosts will disappear along with it. If the user wants a report on ghost

bundles, step 10 can subtract the set of bundle identifiers of the old generation from that of the new generation and report the differences.

- For REPAIR, ghost removal should be optional. I see two possible avenues: mark and sweep, which requires updating a timestamp in every document, or iterating over the index and checking for the existence of the corresponding bundle in storage. The former wouldn't work for VERIFY since it shouldn't modify the index in any way. The latter would work for VERIFY but it would be harder to implement, and would have to be done in a distributed way. A rough outline would be to have every maintenance worker query the ES index for document IDs starting with the bundle ID prefix assigned to that worker and perform the set difference between bundle IDs in storage and document IDs in ES. Paging further complicates this, for example when a ghost ID falls between two pages.