Search

Problem statement

Users need to perform full-text search over instances, items, holdings. The following requirements should be fulfilled:

- **Assumption:** For over 20+ millions of records search request should be executed in less than 500ms
- Multi-tenant search (e.g. for consortia) should allow to share metadata view permission from one tenant to another
- Search should be efficient in terms of multi-tenancy, namely the amount of data in each tenant shouldn’t affect request execution time for other tenants
- Result counts should be precise for any query and amount of data
- Auto-complete should be based on titles. The response time should be less than 100ms
- Facets (e.g. how many Print-books are found for the search request) should be precise
- **Assumption:** Rich full-text functionality which will be beneficial for user, should be provided, namely:
  - Stemming for words (e.g. find record with term “books” for query “book”)
  - Stop-words (e.g. and. or for English language) should not affect relevancy
  - Relevancy scoring should be based on TF-IDF frequencies, in order to provide the most relevant records at the top
  - Didyoumean for input string spelling correction should be implemented and showed as tip if there is more significantly more relevant query
  - All language depended features should support certain predefined list of languages
- There should be support for CQL queries from input string

Purpose of this page

The purpose of this page is to capture good ideas that have been discussed regarding introduction of a search engine, so that they would not be lost. In addition to that, since these are highly relevant to the possible blueprint item they could provide the beginning of that conversation and therefore this page will be linked to that list.

Proposed solution

Overall architecture

Main points of the data flow:

- The architecture guarantees fault-tolerance, as even if search microservice is down or cannot process the requests at the concrete moment of time the message will be consumed later when search microservice will be up
- The architecture doesn’t affect performance of create/update/delete operations in Inventory directly, as sending of Kafka messages costs virtually nothing
The at least one strategy is enough in such case, as the process of indexing is idempotent. When the data is modified in the inventory service, id of the modified resource and tenantId are sent in the Kafka message to the Kafka topic for this type of resources.

Note:

In terms of overall architecture, it is reasonable to have golden source (e.g., central reliable source of data) for each type of data. For now, both inventory and SRS are sources of metadata information. Their roles should be clearly separated, as inventory has additional information (items, holdings), but SRS has actual marc files. Possibly, inventory could be the golden source of metadata information and SRS is storage for corresponding marc files.

Demo 2020-10-20

https://youtu.be/DW_P93DjZ94 Benevolence team presents the Elastic Search FOLIO modules developed for Shanghai Library from 3:00 - 18:43.

Some documents about this development are available on the Benevolence Benevolence Projects page in the "Elastic Search" row.

Search microservice

In order to implement all the requirements, the new module should be implemented in Folio. Primarily, its responsibility should be to index instances, holdings, and items, which are stored in the inventory microservice.

The data store for it should be Elasticsearch, as it satisfies all the mentioned requirements and is purely linear scalable and therefore multi-tenancy friendly (see corresponding section below).

CQRS principle should be implemented: inventory should still be used for any write queries and get all by date range queries and should still be the golden source of this information. Contrarily, all full-text requests from UI should be processed only by this module for these type of data. Other data from various components could be also indexed and searched by this microservice, but every particular case should be considered separately, in terms of data size (e.g., more than 1 mln) and search type (e.g., only for full-text search). No transactional calls should be performed against search microservice as Elasticsearch is not ACID and have eventual consistency.

CQRS in such cases doesn’t mean that it will be the same API on okapi side for both reads and writes. In order to keep it simple, UI will send query requests to search microservice and data modification requests to inventory microservice.

For providing of updates from other microservices to search microservice, Apache Kafka should be used (see corresponding section below). On any change of data (e.g., instances, holdings, items are created/updated/deleted) in inventory storage, it should send Kafka message containing id of the modified resource and tenantId to the Kafka topic for this type of resources (e.g., inventory.instance.ids topic). Search microservice subscribes to the topic and when the message is received, it makes a GET request inventory for the data. If there would be requirements for specific representation of data in Search or indexing instances along with its holding information in one index, it can be implemented by means of database views while preparing response and embedding this information in the instances in Elasticsearch index.

The existing schema (json) from inventory can be used as the golden source of metadata for search microservice in order to build scalable mappings for indexes. However, in such case, it should be expanded because there should be additional attributes which are relevant only for Elasticsearch.

High Level Elastic Search Client should be used for doing the requests: https://www.elastic.co/guide/en/elasticsearch/client/java-rest/master/java-rest-high.html.

Inventory domain events

Inventory should send notifications, when there is any change of domain entities (instance/holding/item). For the documentation of the architectural pattern please see: https://microservices.io/patterns/data/domain-event.html.

The pattern means that every time when an instance/item is created/updated/removed a message is posted to Kafka topic:

- inventory.instance - for instances;
- inventory.item - for items;
- inventory.holdings-record - for holdings records.

The event payload has the following structure:

```json
{
   "old": {...}, // the instance/item before update or delete
   "new": {...}, // the instance/item after update or create
   "type": "UPDATE|DELETE|CREATE|DELETE_ALL", // type of the event
   "tenant": "diku" // tenant name
}
```

X-Okapi-Url and X-Okapi-Tenant: headers are set from the request to the Kafka message.

Kafka partition key for all the events is instance id (for items it is retrieved from associated holding record).

Domain events for items

The new and old records also includes instanceId property, on the same level with other item properties, which defined in the schema:
Domain events for delete all APIs

There are delete all APIs for items instances and holding records. For such APIs we're issuing a special domain event:

- **Partition key**: 00000000-0000-0000-0000-000000000000
- **Event payload**:

  ```json
  {
    "type": "DELETE_ALL",
    "tenant": "<the tenant name>"
  }
  ```

Searching over linked entities fields

In order to search over linked entities fields (e.g. search for item barcode and find related instance) this fields should be embedded into the instances. If related entity is updated the instance should be reindexed. "Partially updating" the instance with the only new data (item/holding) is impossible due to impossibility to maintain consistency of the data (considering multiple indexer module instances) in such case. For every change inside the inventory-storage (e.g. item is changed), the flow should be the following:

1. Message with id and type should be sent to Kafka-topic.
2. Indexer module should receive the message and find all related indexed entities ids (e.g. instance ids)
3. These ids (instance ids) along with types should be sent as messages to the Kafka-topic

Notes:

- Relations could be described in metadata json files
- Indexer module could be a part of search module or a separate module.
- There could be different topics for different kind of entities

Elasticsearch configuration

For Elasticsearch deployment there are the follwing options:

1. AWS (or other cloud/on-premises) instances (e.g. AWS m5.xlarge with 500 Gb EBS type GP-3 for storage) with deployed Elasticsearch 7.10 docker container
2. AWS Elasticsearch service

For all tenants data should be stored in tenantId_instance index. Every index should have 4 shards. The replication factor 2 should be setup for each shard. To make the system high-available for writing replication factor should be 3, so if some node goes down, data is still available for searching and updating. But it would need more resouces. Therefore having replication factor of 2 is a reasonable tradeoff.

This approach provides horizontal scalability for search. Data is spread over cluster nodes, search queries are performed on different nodes. If the number of tenants becomes too big, we just add nodes to cluster and data will be rebalanced over all nodes.

Apache Kafka configuration

Existing Apache Kafka cluster, which is used for mod PubSub can be used. For every resource type there should be separate Kafka topic. In case of multiple search microservices are deployed, the same consumer group should be set for all of them. Id of the record should be used as partition key of the message.

Multi-tenancy

The same Elasticsearch cluster should be used for all tenants. But separate index should be created for each tenant. All the requiquests must contain standard OKAPI_TENANT header. This header is used to use appropriate tenant index for each query.

If mod-search is not enabled for the tenant, then mod-search should skip the Kafka messages for it (otherwise search would wait forever if tenant was not created forever and failed to process messages for other tenants).

NB: When the consortia cross-tenant search requirements are clear, the proposed solution could be enhanced with out-of-the-box solution for ES aliases handling cross-tenant searching (for consortia) in case when permission of viewing tenant metadata is granted from one tenant to another. In order to search over several tenants their ES aliases should be used for request as string concatenation joined by comma. Elasticsearch in such case will use appropriate routing parameter (and therefore shards) and filters. If there is extremely huge tenant, separate index could be created for it and alias could be switched to it without restart, downtime or any code manipulations. Aliases should be automatically created for all tenants and should have filter for "tenant-id" and "routing" parameter set to tenant-id in it. For every entity _routing field should have value of tenant_id.

Reindex in case of data structure changes
In some cases indexes should be recreated and data should be reindexed by sending all ids of existing records from inventory to the Kafka topic. It shouldn't be ordinary operation and in best case scenario shouldn't happen in production.

**No need to reindex data:**

- New column is added and there were no record in database with these information
- Some column is deleted (if the aggregated fields are not implemented)

**Reindex should be performed:**

- The existing data, which was excluded from indexing should be indexed
- Some column is renamed or its data-type is changed

**Calling modules from asynchronous job**

In order to make calls to other modules (e.g. find campus by library_id) there should be a system user for the module with the corresponding permissions. This user should be created on module deployment (it shouldn't be created if it exists). The same strategy was used in mod-data-import.

**Security**

There should be authentication to make elasticsearch requests. The same approach, which is used for database access should be used for elasticsearch. The values should be a env variables, which are used in application.yml

**Java library (maven dependency)**

Java library should be created in order to provide ability to send Kafka messages for indexing in case of data modification. Declarative approach could be leveraged: methods, which are supposed to modify entities could have corresponding java annotations. Common message format (e.g. id, type fields) should be presented in this library.