Conceptual Design:

Full-Text Search

Prepared on: Oct 31st, 2022

Prepared by: Jamal Haija

Problem

Querying for DB objects based on property values (i.e. `{field: 'value'}`) is limited to exact matches. Using Regex (`{"item": {"$regex": /partial/i }}`) allows for partial matches but is not scalable and slow with larger data sets as it requires going through entire object collection and querying specific properties.

Scope

We need text-searching capability on specific data fields that is accessible via end-point parameters such as

`GET /usernames?q=delta`

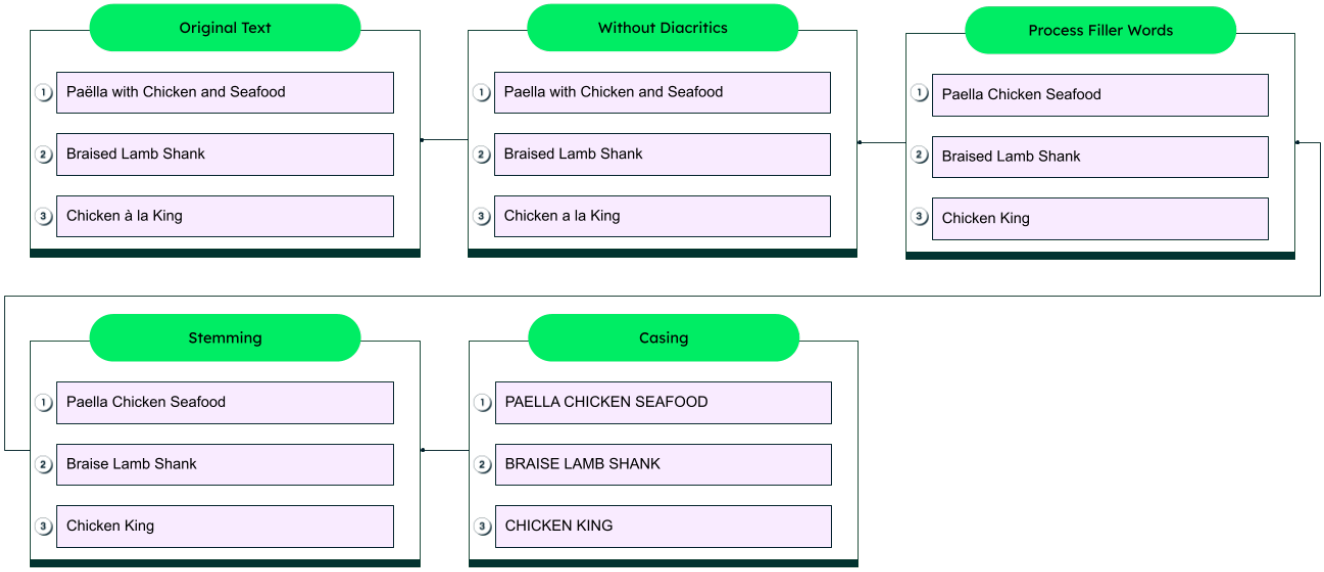
Where `delta` is a slug representing a full or partial string search query.

Solution

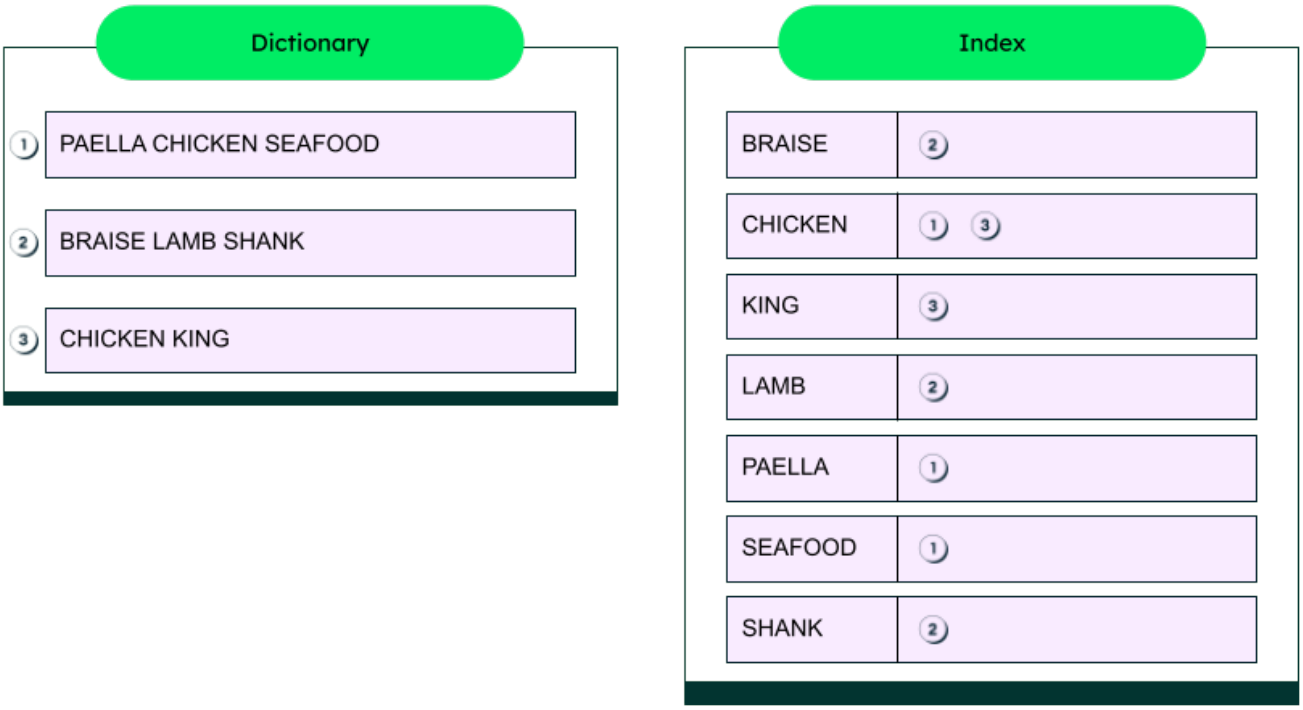
Implement a full-text search capability using data indexing techniques.

The indexing process includes:

- Identifying index fields
- Filtering diacritics and filter words
- Stemming (word forms)
- Casing (making terms case-insensitive)



An index is then created by building a dictionary/hash table with document references:



Once an index is generated, a search engine needs to be used in order to query the index based on provided terms.

Considerations

Several features need to be considered in regards to the search engine capabilities:

Fuzzy search	Typos or near-words	Not needed (?)
Synonym search	Should synonyms appear as part of the result or not (requires thesaurus index)	Not needed for usernames.
Scoring	Query scores: example “interdependent” could be returned when “interlock” is searched as they share “inter”	Not sure for usernames. Needs discussion.
Rich Queries	Intelligent context-sensitive searching. Example: searching for a date could return a date that is close to the search date, but not exact.	Not needed for usernames. We only need direct string search.
Auto-complete	Provide suggestions before querying is complete. For example: “delt” could suggest “deltaone” for the user.	Not sure. Needs discussion. Could reveal usernames when we don’t intend to.

Architectural Complexity & Costs

Adding search-engine capability means adding more software to the stack which requires maintenance and could increase costs (depending on set-up).

Original proposal was to use Elastic Search or AWS Opensearch (based on open source version of Elastic Search 7.0) as solutions, however this requires a system design in which automated document indexing occurs at the database level and the resulting index is synced with the search engine/platform.

The biggest risk with this solution is latency.

Example:

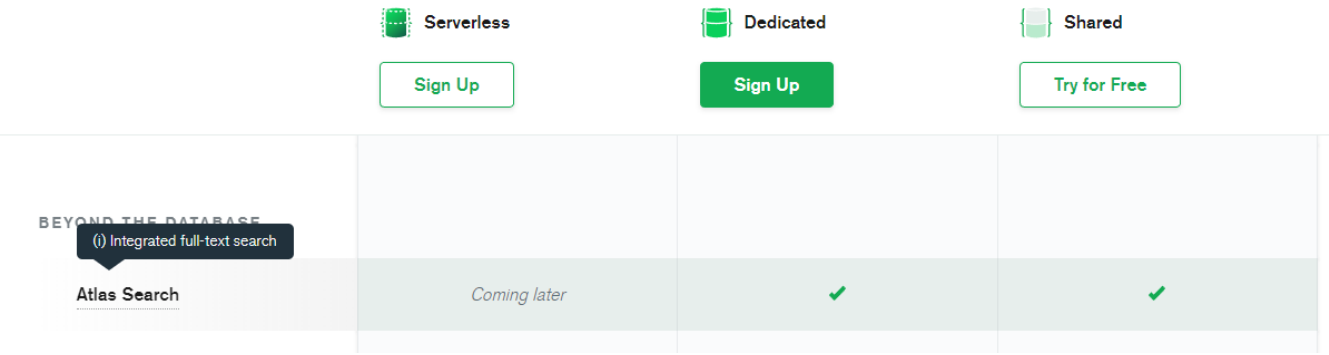
- Armin fully completes on-boarding
- Kaitlyn fully completes on-boarding at the same time
- Kaitlyn tries to search for Armin’s username
    - Armin’s username is not found because the index has not been synced with the search engine yet
    - After a certain delay, the username is searchable

This creates the impression that Armin was not registered in the system and may cause frustration to users.

Increasing syncing frequencies may result in higher costs.

Alternative

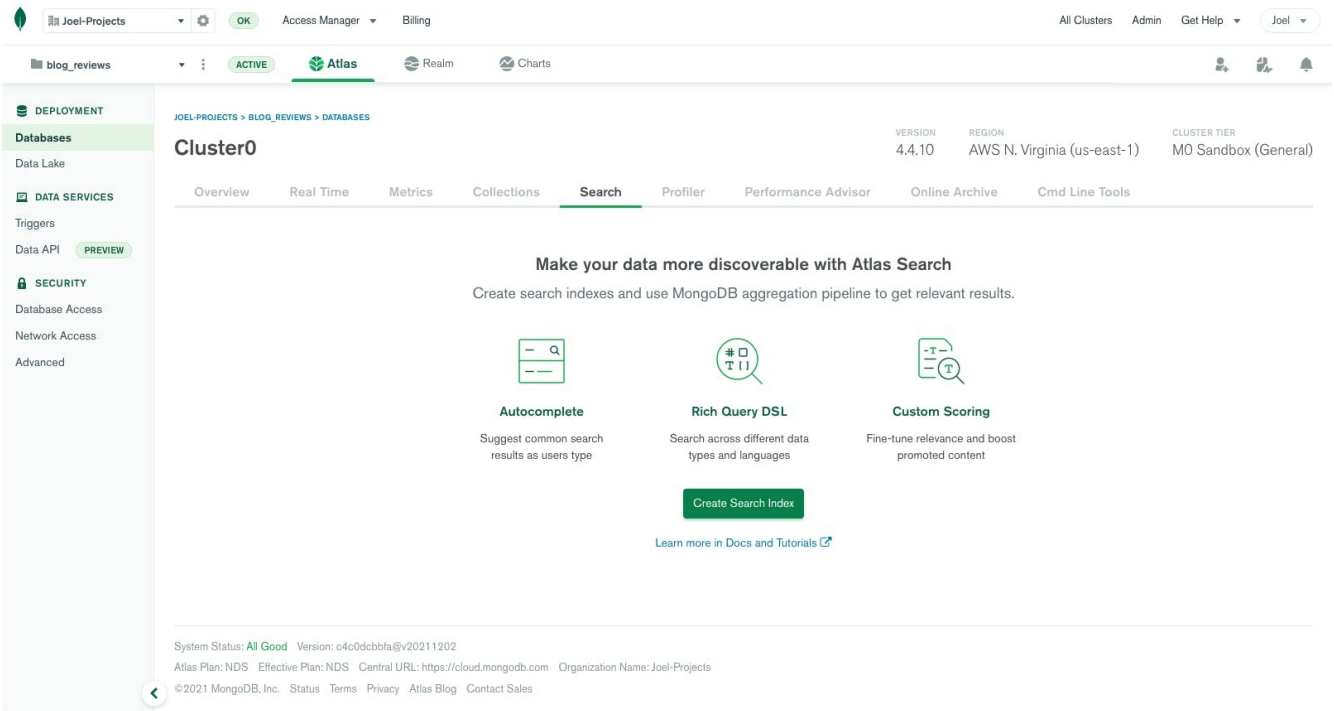
Atlas offers a cloud-based integrated full-text search.



This includes:

- Fully optimized indexing
- Search-engine (Apache Lucene)

This means the database, index and search engine live on the same cloud platform with set-up that is fairly easy:



Last Hurdle

The final challenge is ensuring zero-latency between the data set and the index. Zero-latency is not a viable option for us at this stage due to the potential costs incurred. However, low-latency may be a good-enough solution until we are in the millions of users stage.

Factors that affect latency:

- Frequency of indexing (primary factor; most costly factor as it requires the most cloud computing resources; throttled based on which tier we are on)
- Number of search-engine features (the more features, the more the index needs to be optimized; so we need to decide which search features are must-haves and how we can minimize our needs)
- Inclusion of indexing fields (reducing fields will yield better results; in our case, only a single field: usernames; do we need to consider future use?)

Additional Resources

https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/reducing-search-indexing-latency-to-one-second

https://www.marklogic.com/blog/low-latency-zero-latency-indexing/

Next Steps

- Determine what capabilities we need for querying/searching
- Determine what search features we need from the search engine
- Determine acceptable indexing frequency based on use-case scenarios and decide on latency target
- Draft alternatives
    - Track index freshness
    - If over threshold, do an exact-match query as back-up
    - Consider if this creates additional complexity
- Determine priority and importance of username search capability (remember searching usernames is the primary way people will find each other and connect; think: Instagram)