SYSTEM DESIGN / DOCUMENTATION

This was a pretty interesting challenge to work on and In this doc, I've outlined major design and architectural decisions I made to tackle the presented challenges, with strong focus on functionality, accuracy, scalability, performance and reliability.

Optimizing Performance for Fetching/Searching Records.

```
const allRecords = await this.recordModel.find().exec();

const filteredRecords = allRecords.filter((record) => {
  let match = true;

  if (q) {
    match =
      match &&
      (record.artist.includes(q) ||
        record.album.includes(q) ||
        record.category.includes(q));
  }

  if (artist) {
    match = match && record.artist.includes(artist);
  }

  if (album) {
    match = match && record.album.includes(album);
  }

  if (format) {
    match = match && record.format === format;
  }

  if (category) {
    match = match && record.category === category;
  }

  return match;
});

return filteredRecords;
```

This initial implementation to get records with optional filters came with a lot of pain points, to name a few:

1. **pulls every document into memory:** it loads the entire records collection (100k+ entries was it? smiles) into the application every time we search. As the dataset grows, this will become both incredibly slow and memory-hungry.
2. **no database-Level filtering**: because we do all string checks in JavaScript, we are not harnessing the great capabilities of the Mongodb database. Even if those fields were indexed, we are ignoring them, so every query becomes a full collection scan on the client side.
3. **case-sensitivity logic**: .includes() is a plain javascript substring check, which is case-sensitive. If user searches for "Drake", includes("drake") would return false which is inaccurate
4. **no fuzzy/autocomplete or relevance ordering**: a true "search" usually means allows prefix/suffix/fuzzy matches, ranking by relevance. By just doing includes(), we are effectively doing a dumb substring match with no fuzzy tolerance (for typos), no prefix optimization, and no way to sort by which match is most "relevant."
5. **No server-side/database-level pagination**: having the entire 100k+ documents in the database returned to the Frontend application would highly impact latency, performance

and memory usage on the frontend side. In fact, there is a high chance the client side would have some form of pagination because it can't render that much data at once, hence the Frontend is harbouring the cost of slicing the data for each page.

Solutions - First Iteration

1. (**searchWithText**) My first step was bringing data filtering to the database level and letting mongodb handle fetching data and matching to the respective filters. I used database indexing and created six BTree indexes to cover various combinations of the query params we can filter by.
    - **RecordSchema.index({ artist: 1, album: 1, format: 1 }, { unique: true });**
    - **RecordSchema.index({ album: 1, format: 1 });**
    - **RecordSchema.index({ artist: 1, format: 1 });**
    - **RecordSchema.index({ category: 1, format: 1 });**
    - **RecordSchema.index({ artist: 1, album: 1, category: 1, format: 1 });**
    - **RecordSchema.index({ format: 1 });**

   Of course we might not need all of these, in a real-world system, we want to create these indexes depending on the most frequently used filters. For this challenge, my assumption is we have an equal and high volume of search requests for all the different combinations of **artist, album, format, category.** This significantly cuts down the time to filter based on these fields. During query and filtering, format and category are exact match filters, but for artist and album filters, we use regex with prefix matching, this ensures that the artist filter = 'Chains' will still match the document with the artist 'Chainsmokers' and since we are using a regex with anchored/prefix matching (not a substring match), it will leverage our defined indexes and give better performance.

For the param **q** that allows searching across multiple fields like artist, album, category. We want a more sophisticated behaviour (I mean, I'd want that in an app). For this, I've leveraged the mongodb full-text search with the text index -> **RecordSchema.index({ artist: 'text', album: 'text', category: 'text' });** this one creates an inverted index and tokenizes terms in each field and maps each term in the index to documents that have the terms in one of it's indexed fields, so it's pretty efficient for searching. E.g let's say we have a document with **"album": "First and Last and Always",** it will be indexed under the terms: **first, last and always (**after stemming and removing stop words). So a search queries like *"first love", "first always me", "last", "always", "mr first", "last of us"* will all hit the document and return when looked up. We also sort by relevance. This is efficient but sometimes cannot catch some search queries especially prefixes (one of the biggest cons of $text index for FTS. *though, I solved this more efficiently later with MongoDB Atlas Search - continue reading*), The query -> "Firs" will not match the document with **"album": "First and Last and Always".** I added a nice fallback in the case when our $text search matches zero documents, to do a simple regex prefix matching across those fields (artist, album, category).

2. **Pagination:** Added pagination to process limited sets of data, this increases performance and also to reduce latency because the necessary data is transferred over the network. Depending on the usecase of our application, I could implement either offset-based pagination or cursor-based pagination, they both have their pros and cons. Tho, cursor-based pagination might be more efficient in a lot of cases, but i've implemented the offset-based pagination due to the ability to jump to a specific page which i think is pretty cool and have seen in a lot of search interfaces.

3. **Caching HTTP Responses:** I've implemented a cache interceptor (built on redis) to cache responses for the GET /records. Added a TimeToLive of 15 mins (this is a tricky TTL but i believe should come in handy in cases where users execute the same search within a short period of time), the TTL is best set to a low period as well if we are observing memory constraints. To ensure we don't return stale data to clients, the write methods (create/update) in our records db repository are configured to fire the **records.updated** event after a successful execution signifying the state of our records collection has changed, the event is handled and the cache is invalidated (removing all the cached responses from the store).
4.

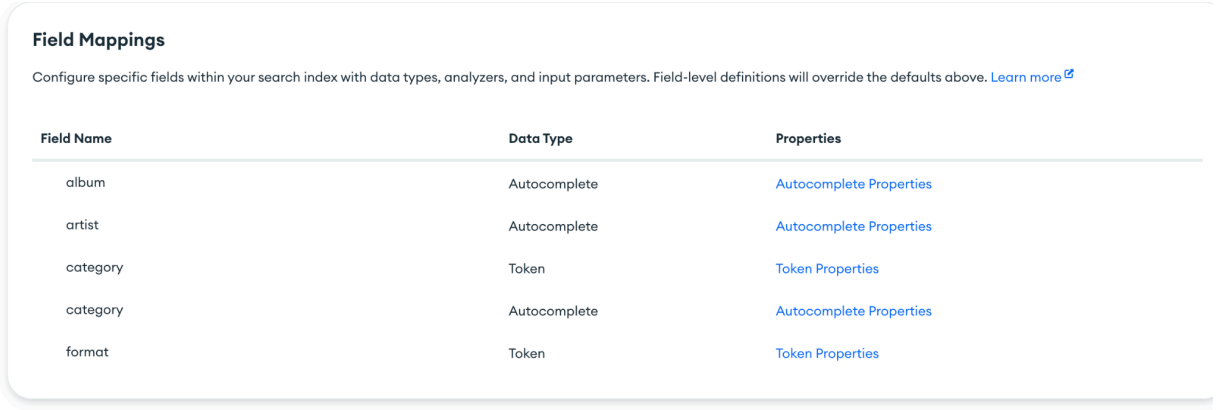
Solutions - Second Iteration

(**searchWithAtlas**) The $text index + regex solution i've implemented in the first iteration works pretty good to an extent and it's well supported in a self-managed mongodb deployment environment (docker, etc). However there are still some limitations, which can be handled very better if we switched to (or are already building on) a mongodb infrastructure running in the cloud managed by Atlas, we'd have access to the Atlas Search and we can define more complex indexes and handle more edge cases while ensuring a top-tier performance.
Atlas Search essentially offers Full Text Search as well. It also creates an inverted index and tokenizes terms in each field and maps each term in the index to documents that have the terms in one of its indexed fields. but it has more advanced tokenizers and analysers to make performant querying easier. There are like 2 major improvements i got here here

1. **Fuzzy (typo-tolerant) search -** if we go back to our example document with ***"album": "First and Last and Always"***, and search for ***q="firsst"***, the $text search will not match this document because no term matches ***firsst,*** but we can have Atlas generate similar terms at query time that can lead to a match for this document (using a maxEdits we specify)
2. **Edgengrams** tokenizers **-** a form of ngram tokenizer that generates prefixes for a token so they can be indexed. For **"album": "First and Last and Always",** a standard lucene analyzer in Atlas will analyze it into the terms: **first, last and always,** then from these, an edgengram tokenizer can generate the tokens: *f, fi, fir, firs, first, l, la, las, last, a, al, alw, alwa, always*. The document is indexed under all these tokens, so just simply searching for "Fir", or "Las" can match this document as a result. Of course we can get the ranking by relevance and sort our output by that. Auto Indexing a field for autocomplete uses edgengrams and can help enhance user search experience

So, just from these two properties, making a search for *q="Firsst and Lasd",* will still return data, a good UX for users.

This is my index definition in Atlas:



I've indexed the ***album, artist and category*** fields with an autocomplete type, with all the possible prefixes of each term (up to len 15) stored in our inverted index and mapped to respective documents. ***category (again) and format*** are indexed as token types so they can be used for exact match filter (equals operator)

For this challenge, I added the extra env variable MONGO_USE_ATLAS=true, to kind of simulate a feature flag to state whether or not the current mongo URI is on ATLAS, so we can programmatically create search indexes and run search aggregation pipelines for Atlas.

Record Creation and Update with MBID

We want to hit the create record or update record endpoint with a new **Mbid**, we must call the MusicBrainz API so we can save the tracklist for the record.
Of course, a naive approach here will be to simply call the MusicBrainz API synchronously during the handling of the create/update request and the MB API returns some data, we parse it to get the track list and then save it to the **tracklist** field of the Record.
This approaches comes with a couple issues:

1. From my research, the MusicBrainz API implements a rate-limit that allows only ***1req/sec*** from the same IP***.*** Let's Imagine It's a very high traffic period for our little broken-store api, with multiple concurrent requests trying to create/update a record with an **mbid**, and all of these consequently trying to hit the MusicBrainz API to pull tracklist data, the server will be rate-limited (probably blocked for a period of time by the MB API) and will eventually terminate the create/update requests if not gracefully handled.
2. If we make having the tracklist data a prerequisite to creating/updating a record, it creates a dependency on the MB API for our api/backend and important business operations can be interrupted by MusicBrainz API having a downtime.
3. Response data from the MB API can be surprisingly huge in certain cases, now fetching this data, parsing it and getting track list from it in a single request-response cycle would not only have significant impact on response latency, but also consume CPU time that could have been be utilized by other requests (parsing can be computationally expensive)

Solution

To tackle these all at once, I went with asynchronous processing for getting a tracklist for a record, using a **rate-limited queue**.

1. When the backend receives a request to create or update a record with a new **mbid**, it creates/updates the record, adds a *GetRecordTracklistJob* to our queue and returns a response to the client immediately.
2. The queue (running on Bull and Redis) is configured to ensure only a maximum of **1 job** can execute within a **1 second window**. This limit elegantly ensures our API never goes beyond the MusicBrainz API rate-limit constraints.
3. Processor picks up the job, fetches data from MusicBrainzz API, parses data to get tracklist, and saves tracklist in the respective record, in the background
4. To add another layer of optimization, we save the tracklist for an mbid in redis cache for 1 hour, so when a request for the tracklist of an mbid is received again within that period, we don't hit the Musicbrainz API again with another round of parsing, etc. for example, a user can set an **mbid**, delete or replace it, then update record again with the initial **mbid** for some reason in a 1 hour window. Therefore, in step (1), before we add the *GetRecordTracklistJob* to the queue we first check if the tracklist we long for is already available to us, in our cache, if it is, we simply save it alongside the record.
5. Other further steps we can take here is to track the status of a job, implement an endpoint to check status of job or send events to the client app when a job has finished to update UI, but this also depends on business and application requirements
6. We could also implement our queue processors to run on a separate thread (a worker_thread or another Node.js process), this decouples the processor from the main Node.js thread serving client requests. This is very ideal in high traffic situations.

Creating Orders for a Record

A record has the **qty** field that denotes records available in stock. I added a new model for
Order { recordId, quantity, unitPrice, totalPrice }
quantity -> quantity of the record to order
unitPrice -> price of the record when we ordered
totalPrice -> totalPrice * quantity.

1. We can't create an order for a record that doesn't exist
2. If the record exists, we can't create a record if it has insufficient stock
3. We first decrement the available stock quantity in the record by the quantity requested and then we create the order.

A major consideration here is that step (3) involves writing data across two separate collections, the Record and the Order. Some possible issues:

- We don't want a decremented stock if there was an issue creating the order and we don't want a new order when stock was never decremented. Simply put, we need this entire operation to be one single atomic transaction, pass together or fail together.
- If process A only decrements the stock price, we don't want another concurrent process B to read the decremented stock price even before A creates an Order. What if A failed to create the order? So we need to ensure isolation and also consistency in high concurrency scenarios.

We execute the two operations using a transaction in Mongodb.

Transient Transaction Error can occur in mongodb and this happens when the transaction fails to commit for some reason like network connectivity to mongodb instance or there was a conflict (like a transaction tries to commit a change for a document that another transaction already committed a conflicting change for). These are retriable transactions, so I've added the utility to retry such transactions for a maximum of 3 times.

IMPLEMENTATION AND TESTING

- I refactored the entire codebase to follow a clean, modular and hexagonal architecture (better separation of business logic, infrastructure, and presentation layers), Following NestJS good practices.
- Unit and E2E Tests for Core Functionalities
- Generalized existing docker compose file for infra setup to include Redis service

Written in entirety by
Godwin Joseph
https://github.com/josephtesla
godwinjoseph693@gmail.com