Scaling Aggregates: Challenges & Solutions

Premise

For the following sections, I will assume the following:

- **I will be working on a React/NodeJS web application.**
- **This web application serves a data stream from its Backend Express API to a React, component-based Frontend.**
- **The purpose of components within the Frontend is to display similar sorts of analytical results, upon request via user interaction.**

1. Setting up for Success

To begin on the best footing, we would apply the following scheme to our Backend API server:

- **Explicitly set NodeJS to 'Production' mode.**

  > Improve performance by disabling exception stack traces and other logging tasks falling back to the debugging mode when necessary.

- **Enable GZIP compression.**

  > Improve performance by shrinking the HTTP payload, which will be expanded by the browser on receipt via Express middleware(s).

- **Cache reusable configurations & results.**

  > Improve performance by caching frequently-used 'hooks' (such as database connection objects) once and re-use them during every request per user. The Redis Express middleware can make this a more robust and intuitive process.

- **Default to Asynchronous methods in processing and responding to requests.**

  > Improve performance by setting up all requests and methods to run independently of each other—handling eventual callbacks in a robust and logical fashion.

- **Run the application as a distributed cluster to balance server load.**

  > Application clusters can be easily configured and automatically scaled via PAASs such as Heroku.

- **Intelligently handle exceptions to maintain integrity in the user experience.**

  > This includes automatic restarts in response to fatal errors.

- **Use a query-based API scheme (such as GraphQL) to reduce payload size based on the request type intuitively.**

  > Improve performance by responding with only absolutely necessary data per request.

- **Use SSR (Server-side rendering)**

  > Improve performance by sending a fully rendered page to the client where applicable. This also simplifies and improves SEO for dynamic content.

2. Dealing with Excessive Load times for Large Datasets/Streams

Since our datasets are going to be very large, we would need a more streamlined approach to getting the data, prior to reading and analyzing it.

> - A possible solution to this would be to stream the dataset via a framework such as Oboe.js in small chunks.
>
> We would then immediately begin computation on each chunk, while

> simultaneously waiting for the next one, and proceed accordingly (in a pagination-like fashion) until the full dataset has been:
>
> - 1. Streamed completely
> - 2. Read completely
> - 3. Analyzed completed
> - 4. Analysis results cached

3. Dealing with Seemingly Non-finite Data Streams

Assuming that our dataset is too large to reasonably load before computation, or that we have no way of knowing what the worst-case scenario is:

> - The Parser class would be refactored to pass each song data entry off for analysis as it is read into the application.
> - The Analyser class would be refactored to accumulate results for any computation in a more granular fashion. It would also return results from its internal state records for repeat requests.

4. Dealing with Computational Strain on User Interfaces & User Experience

In order to reduce the impact (perceptual or otherwise) of computationally expensive tasks:

> - The Analyser class would be refactored to pass the core analysis logic to web worker threads. These threads would run any type of analysis in the background—preventing lag or strain on the Frontend—and pass back the results.

- A major benefit here is that most contemporary CPUs are now multi-core variants (even in mobile devices) so those background workers would provide a performance boost, due to being in a thread/context with more available resources.

5. Dealing with Inconsistencies within datasets/streams

Since we are dealing with user-submitted data, additional normalization steps and configurations are needed. These normalization steps include:

- Expanding regular expression cases for normalizing artists names/groupings in lyrical section metadata headers.
- Expanding regular expression cases for dealing with incorrect or unnecessary data in lyrical section metadata headers.