Database Indexes Report

Implemented Indexes

B+ Tree Index on `title`

- Query: `Song.find().sort({ title: 1 })`
- Used in: Song listing and search functionality
- Benefit: Speeds up title-based sorting and searches
- Justification: Title-based operations are frequent in the application

B+ Tree Index on `releaseYear`

- Query: `Song.find({ releaseYear: { $gte: startYear, $lte: endYear } })`
- Used in: Report generation for year filtering
- Benefit: Optimizes year range queries
- Justification: Year-based filtering is common in reports

B+ Tree Index on `artist`

- Query: `Song.find({ artist: artistId }).populate('artist')`
- Used in: Artist-based song filtering and relationship lookups
- Benefit: Speeds up artist lookups and joins
- Justification: Artist filtering is frequent in the application

B+ Tree Index on `genre`

- Query: `Song.find({ genre: genreId }).populate('genre')`
- Used in: Genre-based filtering
- Benefit: Optimizes genre-based queries
- Justification: Genre filtering is common in reports

Compound B+ Tree Index on `{ title: 1, artist: 1 }`

- Query: `Song.find({ title: title, artist: artistId })`
- Used in: Song search with artist filter
- Benefit: Supports efficient filtering on both fields
- Justification: Common query pattern in song search

Compound B+ Tree Index on `{ releaseYear: -1 }`

- Query: `Song.find().sort({ releaseYear: -1 })`
- Used in: Chronological song listing
- Benefit: Optimizes sorting by release year
- Justification: Common sorting pattern in the application

Compound B+ Tree Index on `{ genre: 1 }`

- Query: `Song.find({ genre: genreId })`

- Used in: Genre-based filtering
- Benefit: Optimizes genre-based queries
- Justification: Frequent filtering operation

Query Analysis

Report Generation Queries

Year Range Filter

```
// backend/routes/songs.js - report endpoint
if (startYear || endYear) {
  matchStage.releaseYear = {};
  if (startYear) matchStage.releaseYear.$gte = parseInt(startYear);
  if (endYear) matchStage.releaseYear.$lte = parseInt(endYear);
}
```

Benefits from: B+ Tree Index on `releaseYear`

Performance Impact: O(log n) instead of O(n) for range queries

Genre Filter

```
// backend/routes/songs.js - report endpoint
if (genreId) matchStage.genre = new
mongoose.Types.ObjectId(genreId);
```

Benefits from: B+ Tree Index on `genre`

Performance Impact: O(log n) instead of O(n) for equality matches

Artist Filter

```
// backend/routes/songs.js - report endpoint
if (artistId) matchStage.artist = new
mongoose.Types.ObjectId(artistId);
```

Benefits from: B+ Tree Index on `artist`

Performance Impact: O(log n) instead of O(n) for equality matches

Search and Listing Queries

Title Search

```
// backend/routes/songs.js - search endpoint
const songs = await Song.find({ title: { $regex: searchTerm } });
```

Benefits from: B+ Tree Index on `title`

Performance Impact: O(log n) instead of O(n) for text searches

Chronological Listing

```
// backend/routes/songs.js - GET endpoint
const songs = await Song.find().sort({ releaseYear: -1 });
```

Benefits from: Compound B+ Tree Index on `{ releaseYear: -1 }`

Performance Impact: O(log n) instead of O(n) for sorted queries

Implementation Details

All indexes are implemented using MongoDB's default B+ Tree structure, which provides:

- Efficient range queries

- Support for equality matches

- Ordered traversal

- Balanced tree structure for consistent performance

The choice of B+ Tree over Hash indexes is justified because:

1. Our queries often require range operations (e.g., year ranges)

2. We need ordered traversal for sorting

3. The data is frequently updated, and B+ Trees handle updates efficiently