Efficient Hybrid Search for Music Discovery

CS6400-A Project Proposal (Group 21)

Aditya Raghavan
araghavan68@gatech.edu

Vanshika Shah
vshah316@gatech.edu

Eric Shao
eric.shao@gatech.edu

Yangsheng Zhou
yzhou895@gatech.edu

1 INTRODUCTION

The music industry has seen a rapid growth over the years and there is a lot of scope in building a efficient and accurate search system that can take care of both semantic aspects of the audio and structured constraints of metadata. Traditional approaches do exist for the performing hybrid search. These methods perform a vector similarity search followed by a structurred filtering. Such methods depend on simple pre-filtering or post-filtering pipeilnes which often suffer from scalability bottlenecks and wasted computation [3] . For this project we looked into some improvements that can be made to search systems to improve performance and speed [2, 6].

In this project we will evaluate a hybrid music search system that fuses vector similarity over Spotify's 19-dimensional audio feature vectors with precise metadata constraints in a single query process. We are currently looking into two methodologies to achieve the same: (1) Intersect-then-rank, where vector and metadata are computed independently then intersected before ranking, and (2) Predicate Pushdown, where metadata filter are done in the approximate nearest neighbor (ANN) search to reduce candidates during search.

In this proposal, we have included the stages within the methodologies and our implementation plan and resources used. We have also included the schema for our dataset along with evaluation metrics we plan to employ. Finally we, have broken down the timeline of how we will complete the tasks mentioned throughout the proposal.

2 PROPOSED APPROACH

Our project aims to build a hybrid music search system that combines vector similarity search over Spotify's 19-dimensional audio feature vectors (e.g., danceability, energy, valence, tempo) with precise metadata constraints. Instead of relying on pre-filtering or post-filtering, our core strategy is to fuse vector embedding search with structured metadata filtering in a single, efficient query. This approach pushes down metadata constraints directly into the search process, avoiding the bottlenecks associated with naive pre- or post-filtering methods.

To achieve this, we are currently exploring two primary hybrid search models: a *intersect-then-rank* strategy and *predicate pushdown* approach, where metadata constraints are applied during candidate generation to reduce expensive approximate nearest neighbor (ANN) computations.

2.1 Hybrid Execution Strategy

As we are currently in the exploratory phase of the project, we have not finalized a single search strategy. We have identified two potential approaches and plan to choose one in the coming weeks.

2.1.1 Hybrid Execution Strategy 1: Intersect-then-Rank [2]. This strategy treats vector similarity search and metadata filtering as two independent processes, combining their results in a final step to identify matching candidates.

- **Candidate Retrieval:** The query audio is converted into a vector. We use a standard ANN index (e.g., one based on clustering or graphs) to rapidly identify an initial set of acoustically promising candidate tracks.
- **Constraint Application:** Concurrently, we use efficient in-memory metadata structures to identify the complete set of tracks that satisfy the user's filters.
- **Fused Candidate Selection:** The key to this approach is performing a fast set intersection between the candidates from the acoustic search and those from the metadata filtering. This step produces a much smaller, highly relevant set of tracks that satisfy both criteria.
- **Final Re-ranking:** We compute exact distances for this refined set of candidates by retrieving their full vectors, ultimately producing a precisely ranked list for the user.

2.1.2 Hybrid Execution Strategy 2: Predicate Pushdown [5]. To give a comparison to the above mentioned method, this method integrates constraints during candidate retrieval phase, to ensure that approximate nearest neighbor (ANN) computations are only performed for items that satisfy the filters. The steps involved in this method are:

- **Candidate Generation with filtering:** Instead of retrieving all acoustically similar candidates and filtering them afterwards, the ANN index (e.g., IVF or HNSW) is traversed selectively. During this traversal, metadata constraints (e.g., price range, release year, explicit flag) are checked before performing expensive vector distance computations.
- **Cluster/ Graph pruning:** For IVF-based indexes, only inverted lists corresponding to items that satisfy metadata constraints are probed. For HNSW graphs, edges leading to nodes that violate metadata constraints are skipped.
- **Efficient Metadata Structures :** Metadata conditions are pre-indexed into auxiliary bitmaps or hashmaps. These structures allow constant-time checks to decide whether to expand a candidate partition or node during ANN search.

- **Final Re-ranking:** From the reduced candidate pool, exact vector distances are computed for only the valid tracks, followed by ranking to produce the final result list.

2.2 Why it will Outperforms Baselines

Both the methods mentioned above are expected to outperform the baseline methods due to the following reasons:

- **Pre-filtering:** When filters are broad (e.g., duration), the baseline must perform a slow, brute-force search on a massive subset. The hybrid systems leverage the speed of the ANN index to accelerate this search.
- **Post-filtering:** When filters are narrow (e.g., a niche artist and specific year), the baseline wastes computation retrieving thousands of acoustically similar tracks, only to discard most of them. Our system filters early, ensuring that resources are spent only on candidates that already match the required attributes or we skip over the parts that don't match the filters
- **Intersect-then-Rank:** Intersect-then-rank performs ANN search and metadata filtering independently, followed by intersection. Predicate pushdown aims to reduce redundant storage and avoid large intermediate sets to improve performance.

3 IMPLEMENTATION PLAN

3.1 Technology & Core Libraries

Please note the below core technical aspects of our project:

- **Language:** Python
- **ANN Indexing:** `faiss-python (IVF,IVF-PQ` or `hnswlib` for their high-performance and versatile ANN implementations.
- **Data Management:** `pandas` for loading, cleaning, and managing the dataset's metadata.
- **Utilities:** `numpy` for efficient vector operations and numerical computing. bitmaps for boolean filtering

3.2 Audio Embedding Strategy and Technology

While are first goal of the project is to build an efficient search pipeline for queries, we also want to explore the possibility of audio matching and search. Below are the technologies we will employ for the same

- **Librosa:** Our foundational toolkit for audio pre-processing, used to load, resample, and standardize audio files before analysis.
- **OpenL3:** Provides pre-trained models that generate high-dimensional vectors, making it a strong candidate for capturing general-purpose acoustic similarity.
- **PANNs:** A state-of-the-art option that produces rich, descriptive embeddings from models trained on the massive AudioSet dataset, ideal for detailed music analysis.
- **Torchaudio:** As a core PyTorch library, it offers advanced models such as Wav2Vec2, providing flexibility for custom model fine-tuning and experimentation.
- **Spotify Annoy:** A Spotify library that uses a forest of random projection trees to provide a fast, memory-efficient, and simple solution for the search for similarity on static datasets [1].

We are still in the exploratory phase of our project and will finalize the strategy for this section.

3.3 Implementation Scope

Our implementation will focus on building the novel components from scratch while leveraging established libraries for standard tasks.

What We'll Reuse.

(1) **Core ANN Algorithms:** We will use a library like `faiss` and `hnswlib` for the underlying ANN index structure.
(2) **Vector Representations:** We'll start with pre-trained models or pre-computed features for track embeddings.

What We'll Build from Scratch.

(1) **Data Processing Pipeline:** A robust pipeline to ingest, clean, and prepare the chosen dataset for indexing.
(2) **Metadata Indexing Module:** A custom module to build and query efficient in-memory indices for the metadata. For Predicate pushdown method, we would use masks for categorical and numeric filters to increase lookup speed.
(3) **Hybrid Search Orchestrator:** The core query executor that manages the multi-stage search logic. This is the central piece of our original work. This executor fuses the embedding search and metadata pruning.
(4) **Baseline Implementations:** Complete workflows for both pre-filtering and post-filtering to ensure a fair and rigorous performance comparison.

3.4 Baselines for Comparison

Our baselines will be carefully implemented to provide a strong benchmark.

- **Pre-filter Baseline:** This method will first isolate the subset of tracks matching the metadata filters and then perform a brute-force search on their corresponding vectors.
- **Post-filter Baseline:** This method will use the same core ANN index as our proposed system. It will retrieve an oversized set of candidates from the index and then apply the metadata filters to this result set.

4 DATASET AND QUERY WORKLOAD

4.1 Embedding Dataset

For Embedding dataset, we are planning to use the Spotify dataset from Kaggle [4], which contains 170,653 vectors across 19 dimensions. For measuring similarity, we will use either Euclidean distance or cosine similarity.

4.2 Metadata Schema Table

The Table 1 is our metadata schema. The following is the decription for each column:

- **id**: The Spotify ID for the track.
- **valence**: A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track.
- **year**: The year when a track was released.

Table 1: Metadata schema table.

Column	Range	Unique values	Data type
id	N/A	170653	string
valence	[0,1]	1733	float
year	[1921,2020]	100	integer
acousticness	[0,1]	4689	float
artists	N/A	34088	list of string
danceability	[0,1]	1240	float
duration(ms)	[5108,5400000]	51755	integer
energy	[0,1]	2332	float
explicit	[0,1]	2	integer
instrumentalness	[0,1]	5401	float
key	[0,11]	12	integer
liveness	[0,1]	1740	float
loudness	[-60, 3.85]	25410	float
mode	[0,1]	2	integer
name	N/A	133638	string
popularity	[0,100]	100	integer
release date	N/A	11244	string
speechiness	[0,0.97]	1626	float
tempo	[0,244]	84694	float

- **acousticness**: A confidence measure from 0.0 to 1.0 of whether the track is acoustic.
- **artists**: The artists' names who performed the track. If there is more than one artist, they are separated by a ";".
- **danceability**: Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity.
- **duration(ms)**: The track length in milliseconds.
- **energy**: Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy.
- **explicit**: Whether or not the track has explicit lyrics (true = yes it does; false = no it does not OR unknown).
- **instrumentalness**: Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal".
- **key**: The key the track is in. Integers map to pitches using standard Pitch Class notation.
- **liveness**: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live.
- **loudness**: The overall loudness of a track in decibels (dB).
- **mode**: Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived.
- **name**: Name of the track.
- **popularity**: The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are.
- **release date**: The date when a track was released.
- **speechiness**: Speechiness detects the presence of spoken words in a track.
- **tempo**: The overall estimated tempo of a track in beats per minute (BPM).

4.3 Query Workload

Below are example queries designed to compare the performance of the Intersect-then-Rank and Predicate Pushdown strategies under different conditions.

High-selectivity queries (narrow filters): These queries have precise metadata constraints that match only a small number of tracks. They test the efficiency of filtering early, a key advantage of the Predicate Pushdown method.

(1) **Point query with multiple exact matches**
   - **Audio feature constraint:** Search for tracks acoustically similar to a specific query song.
   - **Metadata constraint:** Filter for songs by a specific niche artist and from a specific release year
   - **Query example:** Find songs similar to [Track-X] by artist "Niche Artist" released in year 2022.

Low-selectivity queries (broad filters): These queries have less restrictive metadata filters that match a large portion of the dataset. They test how efficiently the system can perform the ANN search without being bogged down by a slow, large-scale filter, which challenges simpler baseline methods.

(1) **Range query with a broad range**
   - **Audio feature constraint:** Search for tracks similar to a specific query song.
   - **Metadata constraint:** Filter for songs with a very broad tempo range.

- **Query example:** Find songs similar to [Track-Y] with a tempo between 80 and 150 bpm.

(2) **Multi-column query with broad filters**
- **Audio feature constraint:** Search for tracks similar to a specific query song.
- **Metadata constraints:** Filter for songs from a popularity higher than some threshold and released after a specific year.
- **Query example:** Find songs similar to [Track-Z] with popularity higher than 50, released after 2010.

Combined selectivity queries These queries blend specific and general filters to stress-test the system's performance. The outcome will depend on which filter the execution strategy prioritizes.

(1) **Mix of narrow and broad filters**
- **Audio feature constraint:** Search for tracks similar to a specific query song.
- **Metadata constraints:** Filter for a specific record label (narrow filter) and a broad range of song duration (broad filter).
- **Query example:** Find songs similar to [Track-A] by artist "Niche Artist" with a duration between 3 and 5 minutes.

(2) **Multi-column range queries**
- **Audio feature constraint:** Search for tracks similar to a specific query song.
- **Metadata constraints:** Filter for a specific danceability range and a specific energy range.
- **Query example:** Find songs similar to [Track-B] with danceability between 0.6 and 0.8 and energy between 0.7 and 0.9.

5 EVALUATION PLAN

Our evaluation plan is designed to rigorously test the effectiveness and efficiency of our proposed hybrid search approaches (Intersect-then-Rank and Predicate Pushdown) against the baseline methods (pre-filtering and post-filtering). We will conduct experiments using the Spotify Kaggle dataset (170k tracks, 19-dimensional vectors with rich metadata) and the synthetic query workload described in Section **??**. The evaluation will focus on three key dimensions: *effectiveness*, *efficiency*, and *scalability*.

5.1 Effectiveness Metrics

To verify that our hybrid search approaches return high-quality results:

- **Recall@K:** Measures the fraction of ground-truth nearest neighbors (obtained by brute-force search) that appear in the top-K results.
- **Precision@K:** Measures the proportion of retrieved tracks that satisfy both similarity and metadata constraints.
- **NDCG@K:** Normalized Discounted Cumulative Gain will be used to assess ranking quality, ensuring that higher-ranked results are more relevant.

5.2 Efficiency Metrics

To test runtime performance under different filtering conditions:

- **End-to-End Latency:** Average and 95th percentile (P95) query latency for different query types (point, range, multi-column).
- **Index Probing Cost:** Number of vector distance computations performed per query, comparing wasted computations across approaches.
- **Memory Overhead:** Index size and auxiliary metadata structure storage costs.

5.3 Scalability and Robustness

We will measure how the system behaves as the dataset and query workload scale:

- **Dataset Size Scaling:** Run experiments on subsets (50k, 100k, full 170k) and stress-test with synthetic 200k+ tracks to evaluate performance trends.
- **Filter Selectivity:** Vary selectivity (from broad filters like `year > 1980` to highly restrictive filters like `artist = specific name`) to observe performance under different query distributions.
- **Parameter Ablations:** Test sensitivity of FAISS parameters (`nprobe`) and HNSW parameters (`efSearch`, graph degree) on both hybrid strategies.

5.4 Baselines and Comparisons

We will compare both hybrid strategies against two baselines:

- **Pre-filter Baseline:** Filter by metadata first, then brute-force similarity search.
- **Post-filter Baseline:** ANN retrieval followed by metadata filtering.

Results will be reported using:

- Plots of latency vs. filter selectivity (broad → narrow).
- Recall@K and NDCG@K vs. latency trade-off curves.
- Resource usage summaries (computation cost, memory).

5.5 Success Criteria

Our hybrid system will be considered successful if:

- It achieves similar or better Recall@K and NDCG@K compared to baselines.
- It reduces query latency, particularly under narrow filters where post-filtering is wasteful.
- It scales efficiently to larger datasets and complex multi-attribute filters.

6 TIMELINE AND DIVISION OF WORK

We have broken down our tasks into weekly goals and noted them in table 2. Since we currently have two proposed methodologies, we have noted down the tasks depending on both of the methods.

The division of work is mentioned in table 3. Our team consists of four team members and we have divided the tasks according to mutual understanding on a broader level. Each of us will also help each other to carry out the respective tasks for the deliverables of this project.

Table 2: Oct 1- Dec 1 Project timeline with weekly tasks for Intersect-then-Rank and Predicate Pushdown.

Week	Dates	Hybrid Method 1 (Intersect-then-Rank)	Hybrid Method 2 (Predicate Pushdown)	Deliverables
Week 1	Oct 1–Oct 6	Dataset setup (Spotify/Kaggle), audio + metadata prep, build baseline FAISS ANN index	Same setup	Repo + dataset ready; **Proposal Submitted**
Week 2	Oct 7–Oct 13	Implement **pre-filter** & **post-filter** baselines; design **query workload** (mix of broad & narrow metadata filters, 200 queries)	Same: baselines + query workload	Baseline runs + workload spec
Week 3	Oct 14–Oct 20	Build metadata indexing module, implement Intersect-then-Rank pipeline	Prototype Predicate Pushdown (integrate metadata constraints during ANN search)	Hybrid draft implementation
Week 4	Oct 21–Oct 27	Optimize set intersection, scale to 100k dataset, test on workload	Implement ANN pruning (IVF/HNSW) + metadata-aware candidate filtering	First hybrid evaluation
Week 5	Oct 28–Nov 3	Evaluate vs baselines (Recall@K, latency, P95, filter selectivity curves), report writeup	Tune pushdown parameters, evaluate vs baselines on workload; report write-up	**Milestone Report (Nov 3)**
Week 6	Nov 4–Nov 10	Ablation: vary ANN parameters (nprobe, efSearch), measure workload performance	Same ablations on pushdown parameters	Plots + analysis
Week 7	Nov 11–Nov 17	Collect tail-latency results, stress-test workload at larger scale (e.g., 200k)	Same	Final eval dataset runs
Week 8	Nov 18–Nov 24	Write methodology & results, finalize system design diagram	Same	Draft report sections
Week 9	Nov 25–Dec 1	Finalize full report, polish plots, prepare demo/CLI/notebook	Same	**Final Report (Dec 1)**

Table 3: Division of Work (4 Members)

Member	Primary Responsibilities	Secondary Responsibilities
Aditya Raghavan — ANN Systems	FAISS setup (HNSW/IVF-Flat); ANN parameter sweeps; candidate retrieval pipeline	Report writing; support Planner module
Vanshika Shah — Predicate Planner	Metadata indexing (inverted lists, buckets, bitmaps); selectivity estimator; query planner logic	Report writing; support ANN optimization
Eric Shao — Baselines & Evaluation	Implement pre-filter and post-filter baselines; build constrained ground truth; query workload design; evaluation metrics	Report writing; support Data prep/cleaning
Yangsheng Zhou — Data & Reporting	Dataset cleaning and preparation; metadata tables; figures; experiment logging; visualization	Report writing; support Baseline evaluation

7 AI USE STATEMENT

Please note we used AI tools to conduct research on our implementation plan and to gain deeper understanding of some of the steps involved. We also used AI to assist with formatting parts of this report in proper Overleaf LaTeX.

REFERENCES

[1] [n.d.]. spotify/annoy: Approximate Nearest Neighbors in C++/Python. https://github.com/spotify/annoy.
[2] Vespa Engineering Blog. 2022. Query Time Constrained Approximate Nearest Neighbor Search. https://blog.vespa.ai/constrained-approximate-nearest-neighbor-search/.
[3] ApX Machine Learning. 2025. Advanced Filtering Strategies: Pre vs. Post Filtering. https://apxml.com/courses/advanced-vector-search-llms/chapter-2-optimizing-vector-search-performance/advanced-filtering-strategies/. Accessed: 2025-09-25.
[4] Vatsal Mavani. 2020. Spotify Dataset (1921–2020). https://www.kaggle.com/datasets/vatsalmavani/spotify-dataset/data. Accessed: 2025-09-25.
[5] Jiayang Shi, Yuzheng Cai, and Weiguo Zheng. 2025. Filtered Approximate Nearest Neighbor Search: A Unified Benchmark and Systematic Experimental Study. *arXiv preprint arXiv:2509.07789* (2025). https://arxiv.org/abs/2509.07789
[6] Zhixin Zhao, Xiaoyang Wang, Dongjie He, Jinhui Yuan, and Bin Cui. 2023. A Comprehensive Study of Hybrid Vector Search: When and How to Combine Vector with Attribute Filters. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991. https://doi.org/10.14778/3598581.3598594