Building a Semantic Search Engine with Vectorized Databases

Introduction:
The objective of this project is to design and implement an indexing system for a semantic search database. The system should be able to efficiently retrieve information based on vector space embeddings. The project will focus on building an indexing mechanism for a vector column only.

Semantic Search:
Semantic search is a technology that enables search engines to understand the meaning of the search queries and provide relevant results based on the context and intent of the searcher. Unlike traditional keyword-based search methods, which rely solely on matching keywords, semantic search uses natural language processing and machine learning algorithms to analyse the relationships between words, phrases, and concepts, and retrieve results that match the underlying intent of the search query. This means that even if a student enters a query using different words or phrasing than what's contained in the database, the semantic search engine can still return relevant results because it understands the context and intent of the search. For example, if a student searches for "What are the best ways to study effectively?", a semantic search engine would return results related to studying tips, time management strategies, and productivity techniques, rather than simply returning results that contain the individual keywords "study," "effectively," and "ways." Please read this page for more about semantic search and why it is used.

Here is a full scenario for semantic search, example of images search engine:
From the user's perspective:
The user wants to search for images that are similar to the one they have. They start by uploading the image and waiting for the search results. The system responds with a set of 10 images that are related to the original image. If the user doesn't find what they're looking for in this initial batch, they can click through to the next page to see another 10 images. This process continues until the user finds what they need, at which point they can exit the session.
From the system's perspective:
To enable this functionality, the system must first crawl the entire internet to gather all available images. It then passes each image through an AI model that converts it into a vector representation, which captures the essence of the image. The system stores both the image and its corresponding vector in a database for future reference. When a user submits a new image, the system passes it through the same AI model to generate a vector representation. It then performs a search using this vector to identify the top 10 vectors that are most similar to it. Finally, the system retrieves the images associated with these vectors and displays them to the user. If the user requests additional pages of results, the system repeats this process to find the top 20 similar images, and so on.

Project Scope:
The project will require the development of an indexing system that can efficiently store and retrieve data based on vector space embeddings. The following requirements must be met:

- The database must have only two columns (ID, embedding -vector of dim 70-)
- The database must have an indexing system that retrieve the top_k most similar rows to the given input
- The indexing system must be able to handle large datasets (up to 20 million).
- The indexing system must be able to efficiently retrieve the top 'k' results for a given query (k is up to 10).
- The system must response in a reasonable time

Here's a comprehensive overview of the project:
We (TAs) will provide you with vectors, as if they were generated by an AI model. Your task is to store them and create an index for the vector column. Next, we'll give you a vector, as if it's the image to search for, along with the number of desired vectors to retrieve (k). Your job is to retrieve the most similar vectors to the given one. Cosine similarity will be used to calculate the similarity between vectors. We'll assess the quality of the retrieved vectors by comparing them to the actual most similar vectors. While approximations are acceptable, accuracy is important. We'll also evaluate the time it takes to retrieve the vectors.

Deliverables:

- Design Document: A detailed design document outlining the proposed indexing system, including its architecture, data structures, and algorithms. The document should also discuss the reasoning behind the design choices and any trade-offs made during the development process.
- Implementation: An implemented indexing system that meets the requirements listed above. The implementation should be written in Python and you shouldn't use any database management system (e.g., MySQL, PostgreSQL).

Evaluation Criteria:
The project will be evaluated based on the following criteria:

1. Accuracy (Recall): The indexing system must accurately retrieve the top 'k' results for a given query.
2. Efficiency: The indexing system must efficiently retrieve the results, with a reasonable query time and memory usage.
3. Scalability: The indexing system must be able to handle large datasets and scale appropriately.
4. Documentation: The design document must be well-written and provide sufficient detail for someone to understand the indexing system.

Timeline:

- Week 7: Submit the initial version of the design document.
- Week 11: Submit final deliverables (final version of the design document and the implementation). Further instructions of how to submit the code will be announced later

This project document provides a general outline of the project requirements and expectations. The details of the project may change based on the needs and goals of the team, but the core objectives remain the same. Good luck with your project!

Building a Semantic Search Engine with Vectorized Databases
Project Details

Deliverables:

The index files for four databases, which will be used to evaluate the project. The databases are provided to you with varying sizes: 20M, 15M, 10M, and 1M. More details can be found in the Evaluation.ipynb.

System Constraints:

DB size/ Constraint	**1 M Row**	**10 M Row**	**15 M Row**	**20 M Row**
Peak RAM Usage (on retrieval phase)	20 MB	50 MB	50 MB	50 MB
Time Limit (on retrieval phase)	3 sec	6 sec	8 sec	10 sec
Score (Min accepted)	-5000	-5000	-5000	-5000
Index Size (Max accepted)	50 MB	100 MB	150 MB	200 MB

Notebook and Code Walkthrough:

First you'll fork this repo: https://github.com/farah-moh/vec_db

In the repo you'll find a vec_db.py file. This file contains the implementation for your vector index. It has multiple functions:

1. **generate_database():**
   This function generates a random database of a given size. You should run this to generate the databases for the specified sizes in the project requirements. It then calls a build_index() function to build the index on the generated data.
2. **_build_index():**
   This function is empty. You should add your index building logic here.
3. **_write_vectors_to_file():**
   This is a utility function that is called by generate_database() function to write the randomly generated data on disk. It uses a memory mapped implementation which enables you to read specific rows from the database without reading the whole file on memory.
4. **_get_num_records():**
   This is a utility function that calculates the number of records in the database.
5. **insert_records():**
   This function inserts new records in the data file. It extends the size of the current file and adds the new data. It then calls the _build_index() function again to rebuild the index. If your index supports insertions, you should call the insert function of your index instead (but handling insertions is not required).
6. **get_one_row():**
   This function reads one row from the data file given its index. It doesn't load the whole file on memory, but only the needed row.

7. **get_all_rows():**
   This function returns all the database rows. Take care as it will load the whole file on memory.
8. **retrieve():**
   This function is the main retrieval function in your index. It returns the top K vectors given a query vector. The current implementation is a brute-force KNN solution (very slow). You should replace it with your retrieval logic, **without changing the function name or parameters.**
9. **_cal_score():**
   This is a utility function that calculates the cosine similarity between two vectors. This is the similarity measure that we will be using.

There's also a evaluation.py file which has the following functions:

1. **run_queries():**
   This function will run a given number of queries against your index and return your result and the ground-truth (actual top K nearest vectors).
2. **eval():**
   This function evaluates your results against the ground-truth. Maximum score is 0.
3. In the **main** function you can try generating some data and running some queries to test your retrieval recall.

Then you can use this notebook to evaluate your code:
https://colab.research.google.com/drive/1NDjJl623MuTBXJcVvtd09zIW-zF5sjIZ#scrollTo=hV2Nc_f8Mbqh&uniqifier=1

The notebook clones your repository and loads your index files, guiding you through the evaluation process. Follow the steps in the notebook to ensure your code is evaluated correctly.

We'll agree on a common seed for the database, so everyone has the same data. On the submission day, the query seed number will be changed.

Important Note: The __init__() and retrieve() functions must retain the same name and parameters as in the repository, as it will be called during evaluation. You are free to restructure the code and add additional functions, but make sure your code runs without errors in the notebook.
Important Note: You are not allowed to use any form of caching during the retrieval process. Any variable created in the __init__() function is considered a cache, except for fixed-size values such as strings, integers, or other simple data types. You must not store large objects like lists, NumPy arrays, dictionaries, or any in-memory structures.
The retrieve() function must also avoid all caching. This means your index must be saved on disk and read directly from disk every time the retrieve() function is called. In other words, the retrieval process should not rely on any preloaded data kept in memory between queries.
Important Note: The entire project must be implemented in Python. You may use common scientific and machine learning libraries such as NumPy, Pandas, SciPy, and scikit-learn, as well as other standard Python packages. However, you are not allowed to use specialized vector search libraries such as FAISS, Annoy, or HNSWlib.