Search by Similarity

Finding the best passage given a query

Introduction

Consider the problem of fetching the most relevant passage from a corpus of text given a query.

This problem is generally solved by representing both passages and queries as a vector of double values (Embeddings). Similar text have closer embeddings.

This process consists of two stages.

1. Embedding Creation
2. Embedding Search

We explore parallelization of these two stages.

Overview

An Embedding can be created by different techniques

- **Tf-Idf**
- Embedding from a neural network

Embedding Search, ie to find the closer embedding

- Optimal Search using Manhattan Distance
- **Cosine Similarity**
- Sub-optimal Search using Approximate techniques (eg: faiss)

Dataset used. A subset of MS-Marco dataset, Real-World Bing Search Queries.

20k queries, 200k passages

Embedding Creation using TF-IDF

What is TF-IDF?

- A statistical measure used to evaluate the importance of a word in a document relative to a corpus (set of documents).
- TF-IDF is widely used in text mining and information retrieval.
- Helps in weighting words in documents for tasks like text classification, clustering, and search engines.

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term t appears in document d}}{\text{Total number of terms in document d}}$$

$$\mathrm{IDF}(t) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing term t}}\right)$$

$$\text{TF-IDF}(t,d) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t)$$

Algorithm

Step 1: Calculate IDF for all the words present in the Documents.

Step 2: Calculate TF vector for each document.

Step 3: Combine them.

Parallelization

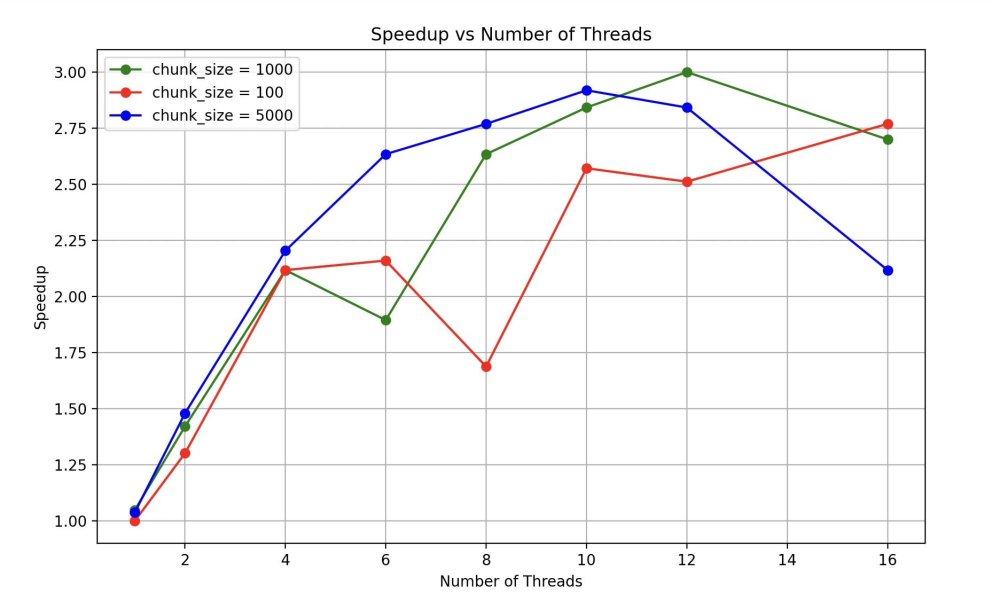
Approach 1: Brute Force parallelization

Creating a new spark for every passage in our dataset (~200k). When we tried this, as expected it was not efficient and majority of the sparks never converted.

Approach 2: Chunking and MapReduce

Splitting our input into smaller chunk and ran the algorithm on smaller chunks and combined the output appropriately. i.e.: adding the maps for idf and just concatenation for TF.

Speed up vs Number of Cores

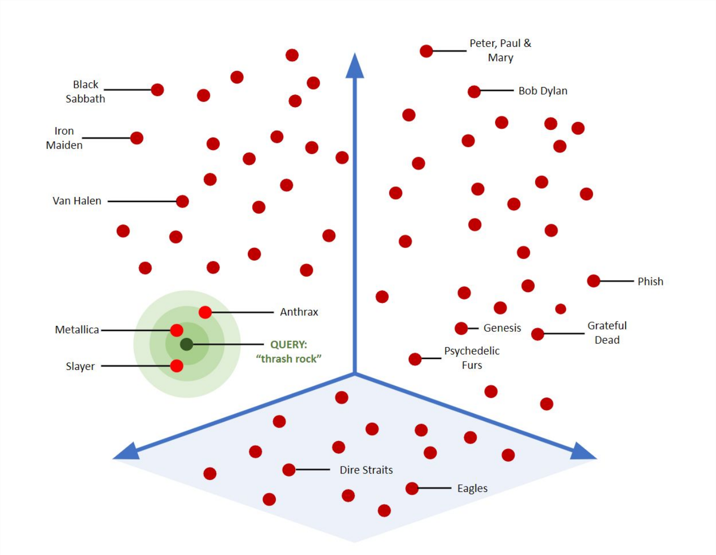


Embedding Search

Closest Embedding Search

Given a query embedding vector and a large list of text embeddings. Output the text embedding closest to the query.

Embedding Distance: Cosine Similarity




```haskell
computeSimilarities :: Embedding -> [IdEmbedding] -> [(IdEmbedding, Double)]
computeSimilarities queryEmbedding passages =
    let compute idEmb = (idEmb, cosineSimilarity queryEmbedding (snd idEmb))
    in map compute passages

-- Find the best match in a list of passages for a given query
findBestPassage :: Embedding -> [IdEmbedding] -> IdEmbedding
findBestPassage queryEmbedding passages =
    let similarities = computeSimilarities queryEmbedding passages
     in fst $ maximumBy (comparing snd) similarities
```

Parallelization Strategies

Basic Strategy vs Chunk based Strategy

Basic Strategy: Spark Creation for passage similarity calculation

```
computeSimilarities queryEmbedding passages =
    let compute idEmb = (idEmb, cosineSimilarity queryEmbedding (snd idEmb))
    in parMap rdeepseq compute passages
```

Chunk Based Strategy: Spark Creation for chunk processing

```
findBestPassage :: Embedding -> [IdEmbedding] -> IdEmbedding
findBestPassage queryEmbedding passages =
    let chunks = chunkList chunkSize passages
        -- local maximum
        bestInChunks = parMap rdeepseq (findBestInChunk queryEmbedding) chunks
    -- Global maximum
    in findBestInChunk queryEmbedding bestInChunks
```

Sparks Data Comparison

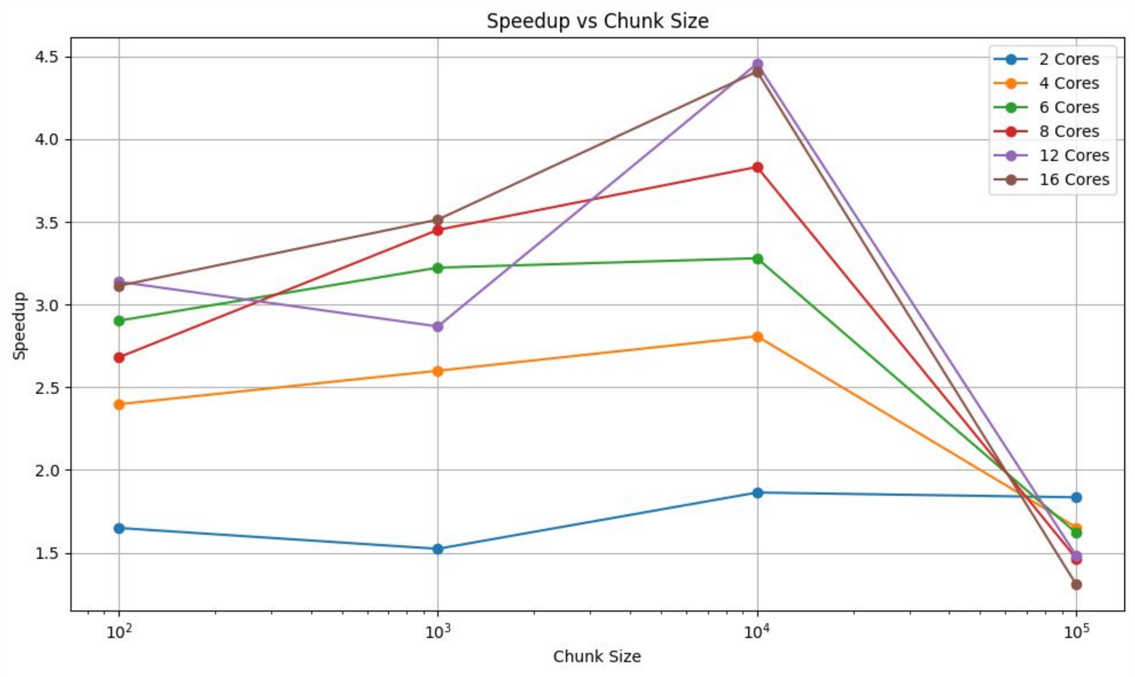
Basic Strategy

Cores	Total Sparks	Converted	Overflowed	Dud	GC'd	Fizzled
2 Cores	1,995,100	66,091	1,748,785	0	0	0
4 Cores	1,995,100	76,177	1,763,275	0	0	0

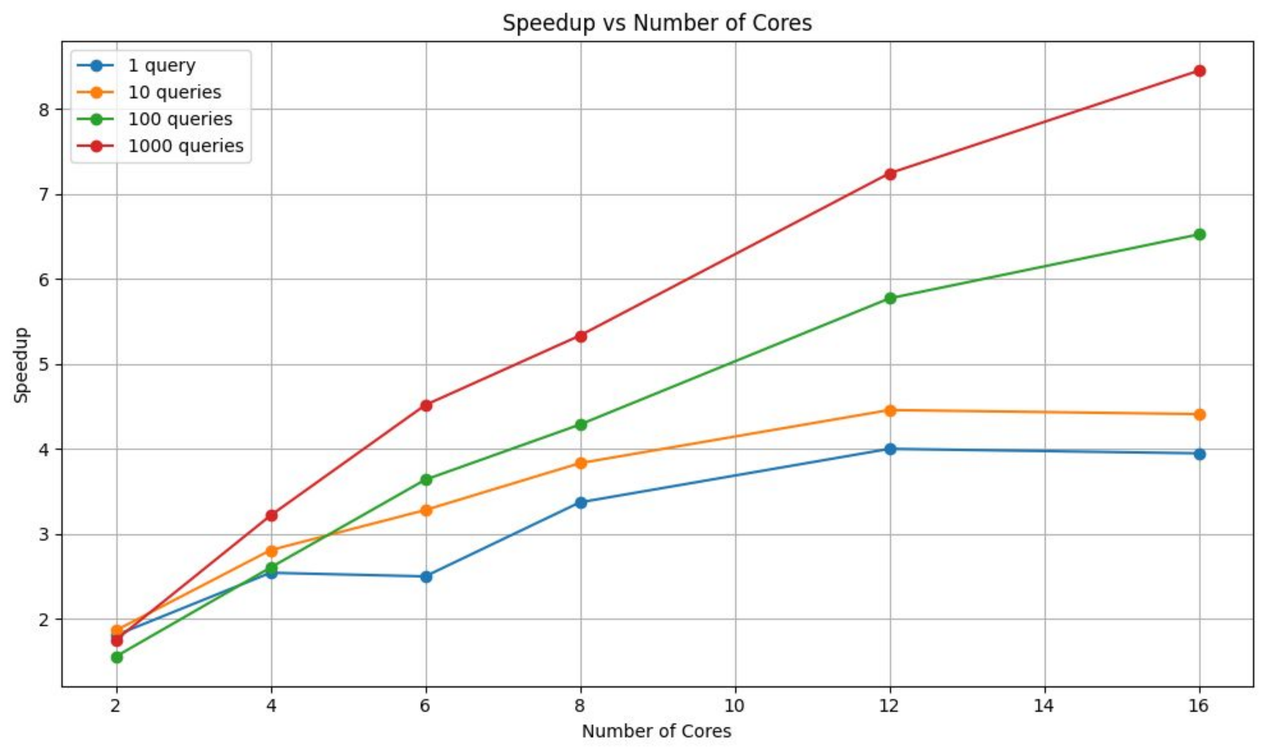
Chunk based Strategy

Cores	Total Sparks	Converted	Overflowed	Dud	GC'd	Fizzled
2 Cores	19,960	19,960	0	0	0	0
4 Cores	19,960	19,960	0	0	0	0

Chunk Size Comparison



Different Search Load, ie number of queries



Number of Passages