INF FAKULTÄT FÜR INFORMATIK

Otto-von-Guericke-University Magdeburg

Faculty of Computer Science

Department of Databases and Software Engineering

Optimization of the search experience in search engines with Vector databases and Transfer learning

Master Thesis

Author:

Ashish Soni

Examiner and Supervisor:

Prof. Dr. rer. nat. habil. Gunter Saake

Supervisor:

Dr.-Ing. David Broneske

Magdeburg, 30.03.2023

Ashish Soni
Optimization of the search experience in search engines with Vector databases and Transfer learning
Master Thesis, Otto-von-Guericke-University Magdeburg, 2023.

Abstract

In today's world, with an ever-increasing focus on digitization and exponential growth of information, it is easy to become overwhelmed by information overload, thus search engines have become an increasingly crucial technology to deliver the desired information in a quick and effective way. There are many question-and-answer platforms available today that have been developed for different purposes in different domains. Some of the widely known are - Quora, a platform for asking any question a user needs an answer to, Reddit for news, having discussions on topics and Stack Overflow for programmers to name a few. These platforms help users get answers to their questions by essentially enabling them to ask the whole community of users worldwide. To ensure that users are able to access relevant information and have an enriching user experience, these platforms rely upon identifying semantically similar questions from vast amounts of data. The task of detecting similar questions is challenging because of the complexity of natural language. A unique intent can be expressed in various ways using distinct vocabulary, different phrases and different structures of sentences. The system must be able to provide relevant information in different languages to serve the needs of a diverse set of users around the globe. Thus, the need for a semantic search application that can identify similar questions and handle different languages to optimize the search experience for users. Traditional search engines used an inverted index to represent information and present results to the user using techniques like simple key-word based search, which does not consider semantic relationships and brought a lot of complexity and engineering challenges when adding multilingual search capabilities to a search engine. To address these challenges, Modern search engines have evolved significantly, accompanied by the rise of Machine Learning (ML) and Deep Learning (DL) techniques that have led to significant breakthroughs in Natural Language Processing (NLP) to help computers understand language. Since computers only understand numerical values, the data is transformed from text into a numerical representation, called embeddings. This representation captures the underlying semantic meaning of the transformed information, to add the ability to perform a semantic search. This has also led to the development of vector databases that are built for the purpose of handling the unique structure of these embeddings, to store them. In our approach, we use embeddings and a vector database to develop a semantic search engine that enables fast and accurate search results in multiple languages. The objective of integrating these two technologies is to evaluate the importance of embeddings and a vector database in optimizing the search process. We also aim to gain a deeper understanding of the advantages and limitations of both technologies, to use their strengths and gain optimal search performance. Therefore, in this thesis, we develop a semantic search application to find similar questions based on a user query, using a vector database and a multilingual model evaluated on two languages English and German using the zero-shot evaluation (BEIR) benchmark and inference speed. Additionally, we compare the capabilities of PostgreSQL, a relational Database and Pinecone, a vector database.

Acknowledgements

To begin with, I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake and Dr.-Ing. David Broneske for giving me the opportunity to write this thesis at the Department of Databases and Software Engineering.

I would like to express my immense gratitude to my supervisor Dr.-Ing. David Broneske for his guidance and patience during the whole period of this thesis work. He has been really supportive during the whole period, offering his expertise and insightful feedback in shaping this thesis.

I would like to extend my thanks to Prof. Dr. rer. nat. habil. Gunter Saake for his time, valuable feedback and assessment of my work.

Finally, I wish to acknowledge my parents, sister and friends for their unwavering support and encouragement. They have been a source of motivation throughout my academic journey. I am grateful to have you all in my life.

Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Signature

Place, Date

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

ANN Approximate Nearest Neighbors

ANNS Approximate Nearest Neighbors Search

BEIR Benchmarking-IR

BERT Bidirectional Encoder Representations from Transformers

CRUD Create, Read, Update and Delete

CTR Click Through Rate

DL Deep Learning

ETL Extract, Transform and Load

GRU Gated Recurrent Units

GPT Generative Pre-trained Transformer

HNSW Hierarchical Navigable Small World Graphs

IR Information Retrieval

IDF Inverse Document Frequency

IVF Inverted File Index

KNN K-Nearest Neighbor

LSTM Long Short-Term Memory

ML Machine Learning

MRR Mean Reciprocal Rank

MLM Masked Language Modeling

MSE Mean Squared Error

MB Megabytes

NLP Natural Language Processing

NN Neural Network

NSW Navigable Small World

OOD out-of-distribution

PASE PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension

PQ Product Quantization

RNN Recurrent Neural Network

SBERT Sentence-Bidirectional Encoder Representations from Transformers

TF Term Frequency

1 Introduction

This chapter aims to provide an overview of the motivation behind our work and an initial discussion about the concept of semantic search and its impact on improving the search experience in search engines. Additionally, we explore the recent developments in this field, and we outline the goal of this thesis by defining our research questions. We conclude this chapter by providing an overview of the thesis structure. The chapter follows the structure mentioned below:

- In Section 1.1, we provide the motivation of the work done for this thesis and shortly discuss semantic search while also exploring the recent developments in the field.
- In Section 1.2, we present our major contributions to research.
- In Section 1.3, we outline the structure of the thesis and a short overview of each following section.

1.1 Motivation

In the 21st century, social media platforms have become a great success, which can be attributed to their vast active user base across the globe. In the age of digitization, the internet and social media have contributed to the emergence of a number of Question Answering platforms in different domains for example, some of them are stack overflow [Ove] for programmers, reddit [Red] for discussions on different topics, Stack exchange [Exc] a network of such websites covering topics in different fields, where each website covers a specific topic and there is Quora where users share their knowledge and opinions on various topics. Question and Answering sites like Yahoo and Google Answers existed for a number of years but they failed to maintain the content value [AS19] of their topics and answers due to a lot of irrelevant information on them, leading to a decline in their user base. On the other hand, Quora was launched in 2009, currently estimated to have 300 million [Quo] unique visitors every month, with access to 400,000 [Inc] topics that allow users to obtain information from other users across the globe. Since the knowledge repository of these platforms keep growing, there is a need for them to offer relevant information to their users in a quick and efficient way while also handling different languages. Therefore, to enable users to find relevant information quickly and easily, the capability to find semantically similar questions from their knowledge repository is crucial. Hence, developing a semantic search engine that can handle multiple languages is important.

Search has always been a key technology over the years in the digital world. During the 1990s, when Google was first released, it quickly captured the attention of people around the world and became the dominant search engine. As we stand now in the 21st century, in the ever-evolving digital landscape, the amount of data being produced in the world in

2021 was around 79 zettabytes [Gui21] and by 2025, this amount is expected to double as it keeps growing at blazing fast speeds [Gui21], and the amount of information being generated globally by people around the world has also grown exponentially, this has brought a huge challenge of finding the desired and relevant information quickly and easily; therefore placing even more importance on search as a technology in helping people to satisfy their information needs in an easy and effective manner. The rising trend of incorporating Artificial Intelligence (AI) capabilities in different applications also known as the Software 2.0 stack [Kar] to provide a better experience to the end-user, has transformed how the world builds software applications. As a consequence, the approach to building search engines has also changed drastically. The search engines of the 21st century, no longer just rely on traditional inverted indexes to represent the information that has to be made available for searching, also known as lexical search engines that looked for exact matches in the index for query words entered by the user. The rise of ML and DL technologies, accompanied by advancement in the field of NLP, has led to providing a helpful representation of text data and has led to the development of vector databases to store and represent data in the form of vectors for semantic search applications. Thus, transforming the way of designing search applications and providing an optimized search experience in comparison to the lexical search engines of the old.

The increased interest in neural network approaches as explained by [LBH15] [GBC16], due to better compute power, increase in the availability of big data, better Neural Network (NN) models and techniques for estimating parameters. Additionally, the introduction of *word2vec* in 2013, [MCCD13b] [MCCD13a] led to rapid recognition and adoption of word embeddings in NLP. It was a simple model and estimation process for word embeddings(also known as distributed term representations). This led to a significant shift in pursuing NN architectures for Information Retrieval(also called Neural IR) [ZRB+16]. The arrival of the Transformer Architecture [VSP+17] enabled the creation of powerful models that can understand the syntax and semantics of language and overcome the lexical gap of keywords in traditional information retrieval engines, which resulted in significant improvements in the embedding-based techniques used in NLP today, enabling many applications including semantic search. Semantic search represents both the data that needs to be made available for search and the queries entered by end users as vectors, by using word embeddings created by these models based on neural networks. These vectors can capture relationships and similarities between words or documents. Then it retrieves the relevant results and ranks them based on the distance or similarity function that describes the relationship. This enables semantic search to identify information that is semantically similar to the user query even if the query does not contain the exact same words. The main advantage of semantic search over a traditional inverted index is that it can capture the meaning of the text more accurately and does not rely on individual words as the units of meaning. Therefore, we would use this approach to tackle the problem of finding similar questions given a query from the user, calculating similarity by encoding questions and the query from the user as embeddings in a vector space. Recent developments in semantic search and its impact on the improvement of the search experience have led to a lot of interest in the field but there exists a research gap in the comparison of different storage backends and their performance for different data management tasks related to embeddings/vectors used in semantic search applications. Additionally, while there are a lot of studies on using pre-trained neural networks for semantic search and information retrieval, there is a need to further investigate the performance of these pre-trained models in handling multiple languages for semantic search applications and optimizing

the search experience for users. This thesis aims to address these gaps by comparing the performance of different storage backends and evaluating the performance of pre-trained multilingual models for semantic search applications.

1.2 Main Contributions

To achieve the goal of retrieving similar questions to a query in a multilingual setting in a quick and efficient way, the research led to the following contributions being made during this thesis work, which are listed below:

1. We compare the capabilities of vector databases that are currently available and select one that is best suited for developing a semantic search application.
2. We compare and evaluate the selected vector database with a SQL database in terms of different attributes like time taken for data insertion, updation and extraction, as well as data insertion in batches.
3. We compare multilingual models and select one, that is best suited for a semantic search application by performing a Zero-shot Evaluation of the models for two languages English and German using two datasets, provided by the BEIR benchmark [TRR+21]. Additionally, we compare the inference speed of the models.
4. We develop a prototype for a multilingual semantic search application using the Quora Dataset using the selected database and the selected multilingual model.

1.3 Thesis Structure

The thesis has the following structure:

- Chapter 2 provides an overview of the fundamental concepts of different topics and the background information for methods used in this work.
- In Chapter 3, we provide the information about our contributions
- Chapter 4 provides a detailed view of the conducted experiments and the hardware and software technologies used to produce the results.
- Chapter 5 discusses the results of our evaluation.
- Chapter 6 provides the summary of the thesis and a discussion around possible future work.
- The last section contains the bibliography and appendices used throughout our thesis work.

2 Background

In this chapter, we discuss the essential concepts of the topics that are relevant to our work. We provide an overview of traditional search systems and the old approaches to building search systems that handled more than one language and the related challenges. Then there is also a discussion on NLP for Semantic Search and the existing approaches along with its advantages and disadvantages as well as the role transfer learning. We also discuss the distance(similarity) functions to measure the similarity. Additionally, we discuss the evaluation metrics for evaluating information retrieval systems. Lastly, we discuss Vector Databases. The chapter has the following structure:

- In Section 2.1, we will start with a discussion on search fundamentals. We provide a detailed overview of the general search architecture and discuss the key search concepts and certain challenges for traditional search engines. We present the characteristics of the research problem and review relevant studies related to our thesis work.
- In Section 2.2, we discuss NLP concepts related to semantic search, including dense vectors, transfer learning, and multilingual sentence transformers.
- In Section 2.3, we introduce semantic search, its challenges and advantages. We also discuss evaluation metrics and similarity metrics and Approximate Nearest Neighbors Search (ANNS) indexes.
- In Section 2.4, we provide a comparison of vector databases with vector libraries and relational databases as well as other vector databases and discuss their characteristics and features,
- In Section 2.5, provides the summary of the chapter.

2.1 Fundamental Concepts and Background

Semantic Search is one of the most rapidly growing application domains in natural language processing. Semantic Search aims to improve the accuracy and efficiency of search results by capturing the meaning of the user's search query and finding results that are semantically similar. It has been also termed search with meaning[WBB]. It has become increasingly popular for developing search applications in various industries such as e-commerce, healthcare, education, online Q/A platforms, as well as web-based search engines where retrieving relevant information quickly is essential for serving the information needs of users, making it a factor in enhancing user experience and satisfaction. Although to understand how it is applied, it is crucial to first understand the domain, related challenges, previous approaches, and their limitations. In this section, we provide an

overview of search, the evolution from lexical to semantic search and the aforementioned topics.

2.1.1 Search Fundamentals

Search, also known as Information Retrieval is the process of taking a query written by a user, mainly in the form of free or natural text (i.e. a natural language developed by humans to communicate like English, German, Spanish and many more) and return results that a Information Retrieval (IR) system has ranked or arranged according to a score with some notion of relevance. The score field is the key feature that differentiates between a search engine and all other data stores. The history of search is a rich one. It was Vannevar Bush in 1945, who imagined a future where humans could access all kinds of useful information, using a computer in a similar way that a human brain organizes long-term memory[KSS+03]. After two decades of research, led by Gerard Salton[Aar] led to the birth of IR as a discipline, by building the first IR system in the 1960s. Later the emergence of the world wide web in 1990s introduced the possibility of accessing information about the whole world in theory. It wasn't until Google brought two key innovations: MapReduce, and PageRank, the first one to improve the scalability of the indexing process by using distributed computing across servers[CCA+10] and the second one to use link analysis to find the importance of a web page [Rog19]. Search combines different areas of computer science - in particular NLP and ML. In this section, we cover the basics of search and the terminology that is used most often in the search ecosystem.

- *Document*: A document is a primary unit of retrieval and is equivalent to a row or a record stored in a database. Every document represents a single item that is available for search [Teo19].
- *Field*: represents the core unit of search, it is similar to a column in a database. A search query typically contains keywords and the search engine returns the documents that match those keywords in the fields. Fields are also vital for faceting(i.e., filtering using a field value) and aggregation (e.g. sums and averages) [Teo19].
- *Inverted Index*: is the data structure that represents documents and fields for the purpose of retrieval in an efficient manner based on search queries entered by a user. It is responsible for associating every word to the document where that word occurs [Teo19].
- *Indexing*: is the process of adding raw data into an index to make it ready for search. It is a subpart of an Extract, Transform and Load (ETL) pipeline [Teo19].
- *Token*: represents the key unit of indexing and searching. A token typically represents a single word. number or alphanumeric identifier [Teo19].
- *Search Query*: represents the words, a user typed into the search box of a search application [Teo19].
- *Analyzer*: It is a process that converts the raw text of a field or search query to a basic form that involves steps such as converting all data to lowercase, and transforming text into individual tokens by splitting into spaces or other separators. For

efficient search, it is recommended to use the same analyzer for both transforming the raw text and the search query [Teo19].

- *Shard*: When an index becomes too large, it cannot be stored on a single server. We can mitigate the issue by splitting the large index into smaller indexes which are known as shards. The concept of Sharding is also called horizontal scaling. Then, the search application has to look for results for a search query in all of the shards in parallel and return the results after merging them. It adds complexity because the application has to handle possible failures of individual shards and deduplication of documents across shards can be a complicated process [Teo19].

Practical Applications of Search

The search box is one of the only places in most applications where the user can enter what they want in their own words, hoping that the returned results will be useful and satisfy their quest for the desired information. Let's take a look at some standard use cases:

- *Web Search*: involves ranking and retrieving information from a huge number of websites over the internet to make it accessible and available for users. Some of the most famous web search engines in the world include Google, Bing and others. Search on the web means includes requirements like scalability to millions or billions of users, also for an infinite number of web pages. And returning the desired information to a user query by selecting from a huge number of relevant results. It focuses on the number of clicks as a metric for success [Bro].

- *E-commerce Search*: is the driving force behind converting shopping intent to actual purchases. In comparison to web search, e-commerce is handled by a lot of companies, two of which are Amazon and Alibaba. Also, the total number of products is less in comparison to the number of web pages available over the internet. It has certain specialized requirements like scaling up to user traffic of millions during peak times and during holidays, while showing them relevant results at the top, otherwise they might not purchase any item. The success rate depends on whether a user, buys a product or not [SSKZ18].

- *Enterprise Search*: is part of an organization, to enable employees and other stakeholders to access proprietary content that has been stored internally within an organisation. It allows employees to search for a variety of content like emails, tools like Microsoft Office, and wikis shared across document repositories easily and quickly so that they can work in a productive manner [Whi15].

- *Local Search*: is focused on content where location is a primary attribute, commonly addresses, businesses and points of interest. It varies in granularity, where a user might be searching for a specific street address, city, region and country. It is a key feature in applications like Google Maps or Apple Maps [AB07].

- *Legal Search*: also known as eDiscovery is used to find all of the relevant content in an organization, related to a particular topic in the context of a lawsuit. While search is focused on returning some relevant content as quickly as possible. eDiscovery has a key requirement of finding all the relevant content, with the hope that results

will decide the lawsuit in favour of the user looking for the important information [ML6].

- *Machine-Generated Data*: The world is full of machine-generated data which can add a lot of value if it is searchable. As companies generate data by logging events and user engagement or system performance. The data is mostly available in an aggregated format. For example, understanding user traffic and tracking system performance for dashboards and other applications [ML6].

Overview of the Common Components of a Search Application

This section provides insight into the common components of a search application, that is designed to help users fulfil their need for the desired information.

- *Ingestion*: The process of ingestion is the "extract" step of the ETL data processing pipeline to make the content searchable. Either by connecting to a primary data store via default connections or APIs or web scraping. The other steps "transform" and "load" are handled during indexing, where the raw data is transformed into a representation, which the search engine will use to make it searchable [Teo19].

- *Search Box*: is the area in the application, that is used by users to write down their search queries in the form of keywords, questions or using a specific syntax. It is also the place where the search engine provides features like auto-suggest and spell-checks because the user doesn't fully know how to express what they are looking for or they are not yet fully aware of search works in our application [Teo19].

- *Autocomplete*: also known as typeahead or autosuggest, has become a standard feature to help users complete their queries either to save the effort of typing the whole query or help them in the case that they are not fully sure about what they are looking for. Autocomplete suggestions are based on old user queries, also machine learning techniques can be used to predict what a user might type [Teo19].

- *Spelling Correction*: In search engines, it's possible to identify and make corrections for queries that are misspelt by combining dictionaries and analyzing query logs along with indexed contents. Traditionally, spelling correction was done using statistical approaches. Modern Search Engines will correct spelling errors using recent machine-learning techniques that offer better accuracy. Also, they might offer a "did you mean" suggestion.

- *Results*: The search engine results page is the place where relevant results are returned to the user, and they can interact with the results [Teo19].

- *Query Refinement and Faceting*: refers to creating a collection of different independent attributes that classify a document in the search index, also known as faceted search[Tun09], when a user selects a facet value, the search engine treats it as a constraint and filters the returned results. For example, an e-commerce site might have facets like the type of product, brand, size, colour etc. It is beneficial to refine the initial search queries when they are broad or ambiguous and return large sets of results that are heterogeneous. Facets are a way of summarizing or aggregating search results.

Key Search Concepts

In general, every search engine has three subcomponents: an index component that represents the information for retrieval and aggregation, a query component to fetch and sort query results from the index, and an aggregation component that produces or can aggregate results from the retrieved query results. An overview of a general search engine can be found in the diagram Figure 2.1. We will discuss each of these components in detail below.

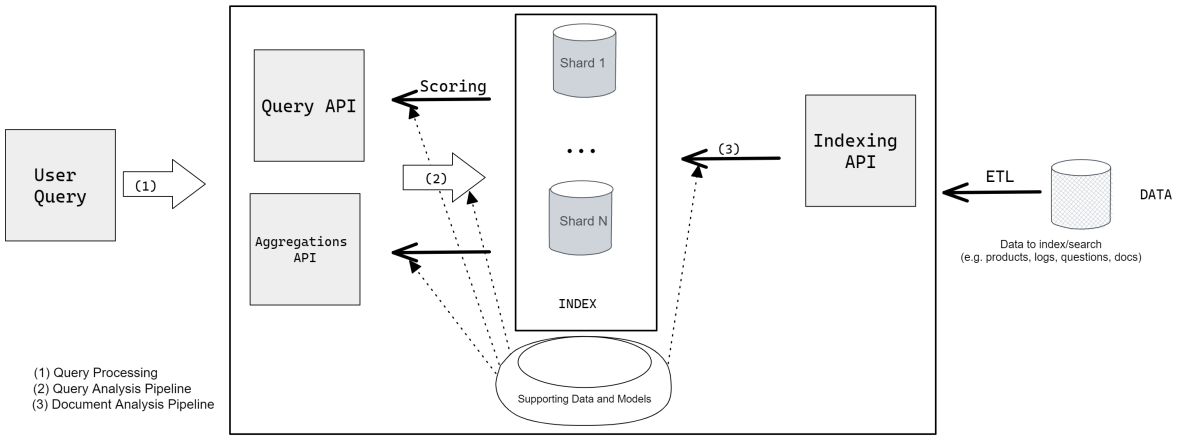

Figure 2.1: An Overview of the Search Architecture [Teo19]

Relevance Relevance is a key driver of search. It is responsible for guiding a search engine to retrieve results that satisfy the user's need for information. Therefore, one of the most important functions of a search engine is to find which documents stored in the index are relevant to the search query entered by the user. Since every user is unique, relevance can vary from user to user and is considered subjective because two users with the same query may not completely agree on which results are relevant. Relevance is a key factor for ranking search results because the user wants to see results that are relevant to their search queries [Teo19]. There are also other considerations while ranking that are beyond relevance such as using popular or recent results. Ranking might also consider user-specific factors like information about a user's location or some user preferences. Search applications also need to consider the relevance of results when they allow the user to override the default ranking results, e.g. by sorting results of a product based on price or based on recency by retrieving the most recent documents. Search engines take a user query as input, and retrieve the results that match the query with the data stored in the index, then rank results to present the top-ranked results to the user. Therefore, they need to ensure relevance in two places: retrieval and ranking.

During the retrieval phase, the raw search query entered by the user is transformed so that it represents the search intent of the user and that representation is used to match the documents stored in the index. With the help of Query Understanding, we ensure that the representation accurately reflects the intent of the user. In a perfect scenario, the retrieved results are exactly the set of relevant results [Teo19]. In practice, there is a tradeoff between precision (the fraction of retrieved results that are relevant) and recall (the fraction of relevant results that are retrieved).

Ranking is responsible for scoring the retrieved results. The scoring function is a combination of different factors which are often broadly differentiated as query-dependent or query-independent. The Query-dependent factors signify the relevance of a document to the query, e.g. which tokens in the query match which fields within the document or whether certain tokens in the query occur within the document as a phrase [Teo19]. Query-Independent factors just focus on the features of the document that signify value, like popularity or recency.

Indexing Indexing refers to the process of ingesting raw content and making it available for search. Traditional search engines do this by creating an inverted index where they map each token(word) to the documents and to the specific fields where those tokens occur [Teo19]. The simple data structure of an inverted index is to map each of the tokens to a sorted array of document IDs. This allows a search application to retrieve documents that contain a specific token or a combination of tokens in a fast and efficient manner. Let's take a look at an example of an inverted index as shown in Table 2.1 for a collection of some short documents:

Document A: Matt loves playing soccer. Document B: People love going to a stadium to watch soccer. Document C: All football players must love to train every day. Assuming that we use an analyzer that converts all strings to lowercase and replaces punctuation with spaces, and the tokenizer splits the document into spaces. We will have the following inverted index.

Word	Occurrence Mapping
matt	A
loves	A
playing	A
soccer	A, B
people	B
love	B, C
going	B
stadium	B
watch	B
all	C
football	C
players	C
must	C
train	C
every	C
day	C
to	B
a	B

Table 2.1: Inverted Index

The last few tokens (to, a) are known as *stopwords* [Teo19], they are considered to be very common that they can be ignored very often, without affecting the meaning of a search query or a document. It depends highly on the use case for the search application, but one

should be cautious about discarding information that might be potentially useful. From this example, one can see how an inverted index is appropriate for **Keyword search** (this is also known as **Lexical Search**):

- An inverted index that is small enough to fit in memory can be represented as a hash table, in which tokens are represented by keys and the sorted arrays of document ids as values.
- In an inverted index, retrieval of all the documents that include a specific token (e.g. "soccer") is a constant time lookup in the table.
- Documents that include multiple tokens (e.g. "soccer" AND "playing") can be retrieved efficiently because computing the intersection of two or more sorted arrays can be done efficiently. Performing an OR operation ("soccer" OR "love") on sorted arrays is also efficient.
- An inverted index can also be compressed using techniques like run-length encoding.

Pros and Cons of a traditional (Keyword/Lexical Search) System

- Traditional search systems based on Keyword/Lexical matching have existed for a long time, so it is a mature technology.
- Since traditional search systems have existed for a long time, they have a well-developed ecosystem that can be easily integrated into existing applications to enable search.
- An inverted index creates a separate entry for "love" and "loves" since they are different tokens. But someone searching for "love" and "loves" might also want to see documents that include "loves" and vice versa.
- An inverted index treats synonyms i.e. words with a similar meaning like "football" and "soccer" as different entries.
- On the contrary, a token(word) with multiple meanings, like "watch", is only given a single entry in the inverted index. Unless we use word sense disambiguation which has its challenges.
- The words "Matt" and "People" are uppercase but the index, specific to the chosen analyzer will lose this information as it converts all tokens to lowercase.
- An inverted index will not guide us on how to rank the returned results. It just gives documents sorted by document id. Therefore, the inverted index is mainly intended for retrieval and not for ranking.
- A simple inverted index does not store any positional information related to the words stored in a document, e.g., information that could inform us that the word "playing" comes before the word "soccer" in Document A. Positional information is very useful for phrase queries (e.g. "playing soccer") which requires that the two tokens appear in a consecutive sequence in the document.

- We used a basic tokenizer in our example, that splits tokens on whitespace and punctuation. It works appropriately for English but for languages like Chinese or Japanese, we cannot use whitespace to split tokens. Tokenization is a crucial concept that defines how text is handled in a search application for the purpose of creating an index and making the information available for search.

- There is no concept of assigning weight or importance to a token for a specific document.

2.1.2 Characteristics of the Research Problem

As for any Q/A platform, it has become crucial to organize their knowledge repositories in a specific way, to enable efficient retrieval and provide an optimized search experience to their users by providing them with relevant information at very fast speeds. Thus, satisfying the user's interest and information needs to build trust by providing accurate and relevant search results. Also, the users represent different countries across the world, using different languages. So, the search functionality should also be able to handle multiple languages.

The task of identifying semantically similar questions is complex due to two main reasons:

1. A single question can be rephrased in many various ways. For example:
   Q1: "How can I merge two branches in Git?"
   Q2: "What is the process for combining branches in Git?"

2. Two questions might be asking different things but looking for the same answer. For Example:
   Q1: "What should I do if I accidentally deleted an important file on my computer?"
   Q2: "How can I recover a file that I mistakenly removed from my PC?"

Additionally, the complexity of providing search functionality in different languages adds more complexity to the problem. We discussed the drawbacks of Lexical Search in paragraph 2.1.1. First, let's explore the challenges related to building a multilingual search application using a traditional keyword-based approach. After examining these challenges, we can better understand the need for alternative approaches such as semantic search.

- *Multilingual Support Complexity*: Building a search application that can handle multiple languages, we need to add a separate inverted index for each language adding more complexity as shown in Figure 2.2.

- *Storage*: Traditional approaches require more storage because as we add more languages, separate indexes are required.

- *Engineering Challenges* It increases engineering challenges, as we need to build a separate data processing and indexing pipeline for each language where each of them might use different tokenizers, stemmers etc.

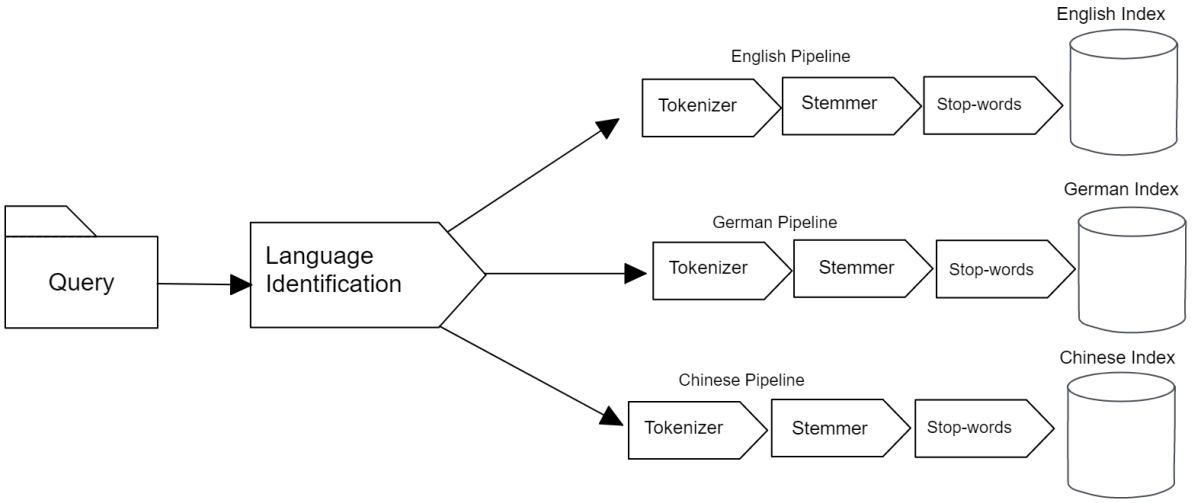

Figure 2.2: Multilingual Lexical Search

- *Maintenance*: It also brings additional challenges related to maintaining and updating search applications potentially leading to higher operational costs and infrastructure requirements.
- *Latency*: It increases the search latency, as we first need to identify the language and then route the search query to the appropriate index based on the identified language. This process extra time and can negatively impact user experience.

In summary, the challenges and complexities in building a multilingual search application using a traditional keyword-based approach highlight the need for a more effective alternative. In the following sections, we will explore semantic search as a potential alternative approach that can overcome these issues. By understanding the advantages offered by semantic search, we aim to demonstrate, that its a more suitable approach for handling multilingual search in Q/A platforms.

2.1.3 Related Work

In this section, we discuss several research works that are relevant to this thesis work. The concepts have provided valuable insights and ideas that have aided in the successful completion of this thesis work.

[WGM+13] presents a detailed analysis of Quora as a social question-answer (Q/A) platform and provides insights into how its internal structure contributes to its success to become one of the most popular knowledge-sharing platforms. The paper also discusses Stack Overflow as a basis for comparison. The paper discusses the impact of three different connection networks (or graphs) inside Quora, a graph connecting topics to users, a social graph connecting users and a graph connecting related questions. Their results show that diversity in user and questions graph plays a huge role in maintaining the quality of Quora's knowledge repository. One graph is responsible for drawing the attention of users and the other points the users towards a smaller subset of trending and interesting questions.

[AS19] presents a data science approach to identify duplicate questions on Quora to ensure the quality and quantity of content provided to the users for enriching the overall user experience. The paper discusses the task of identifying duplicate questions on Quora's question pair dataset using feature engineering and feature importance techniques. The experiments made use of seven machine-learning classifiers namely K-Nearest neighbors, Decision Trees, Random Forest, Extra Trees, Adaboost, Gradient Boosting Machine and XGBoost. They also used four deep-learning models. Accuracy, F1-Score and Log Loss were used for evaluation with training and testing data split into 80/20 for the experiments. The best deep learning model had an accuracy of 85.82%. The paper is focused on a classification task to identify duplicate questions from the perspective of cleaning up Quora's knowledge repository.

[VSP+17] proposed a new deep learning model architecture called the 'Transformer' based on attention mechanisms replacing the recurrent layers frequently used in encoder-decoder architectures. that required less time to train, allowed more parallelism and had superior quality on experiments for two machine translation tasks of WMT 2014 English to German and WMT 2014 English to French establishing a new start of the art. The paper discusses two models for experiments - Transformer (base model) and Transformer(big). These biggest transformer models showed state-of-the-art performance with a BLEU(bilingual evaluation understudy) score of 28.4 for the English-to-German translation task and a BLEU score of 41.0 for the English-to-French task.

[YCA+19] presents two pre-trained multilingual sentence encoding models focused on retrieval, based on the Transformer and CNN (Convolutional Neural Networks) model architectures. The models had the ability to embed text from 16 different languages into a single semantic space using a multi-task trained dual encoder that was trained using translation-based bridge tasks [CYC+18]. The paper also discusses the performance of the models on semantic retrieval (SR), translation pair bitext retrieval(BR) and retrieval question answering(ReQA), where they show competitive performance with state of the art. The models approach and in some cases outperforms English-only, sentence embedding models for English transfer learning tasks.

[RG19] presents Sentence-BERT (SBERT) a modified version of the pre-trained BERT network [DCLT18]. SBERT uses siamese and triplet network structures to obtain semantically meaningful sentence embeddings that can be compared using cosine similarity. SBERT helped reduced the effort for finding similar pairs from 65 hours with BERT to about 5 secs with SBERT while maintaining the accuracy from BERT. Three different pooling strategies (MEAN, MAX, CLS) were evaluated as part of the ablation study to get a better understanding of their importance. Mean pooling outperformed the others. SBERT is more efficient in terms of computation and is 55% than Universal Sentence Encoder [YCA+19].

[RG20] presents an easy and efficient method to extend the available sentence embedding models to new languages, to create new multilingual versions from previous monolingual models. The Training is based on the core idea that a translated sentence should be placed in the same location in the vector space as the original sentence. The paper further discusses the approach to training, taking a monolingual model to generate sentence embeddings for the source language and then training a new model on the translated sentences to mimic the original model. The paper demonstrates the effectiveness of the

approach for 50+ languages. The approach is called Multilingual Knowledge distillation.

[MK16] The paper discusses the existing works on text similarity by dividing them into three different approaches; String-based, Knowledge-based and Corpus-based similarities. String-based measures are applicable for measuring similarity at character levels such as Levenshtein distance and Term-level such as cosine similarity and Jaccard similarity. and others are also discussed. Corpus-based similarity determines the similarity between words based on the information gained in the corpus such as Latent Semantic Analysis (LSA). The knowlege-based similarity is described as similarity based on the degree of similarity between words and information derived from semantic networks like WordNet.

[BA15] presents different evaluation measures that are available for information retrieval. The paper discusses many evaluation measures by categorizing them into two subsets - one for unranked retrieval systems and the other for ranked retrieval systems. The following measures are discussed in the paper - Recall, Precision, Inverse Recall, Inverse Precision, F-measure, Prevalence, Accuracy, Error rate, fallout, and Miss rate for unranked retrieval systems. Precision at k, R-Precision, Average Precision, Mean Average Precision, Mean Reciprocal Rank and Normalized Discounted Cumulative Gain.

[TRR+21] introduces a benchmark known as Benchmarking-IR(BEIR) for robust and heterogeneous evaluation of models for information retrieval to address the problem of limited insights into out-of-distribution (OOD) generalization capabilities of neural information retrieval (IR) models. The benchmark provides 18 datasets that are publicly available. The paper also provides a discussion on the evaluation of 10 state-of-the-art retrieval systems. The benchmark was developed to allow researchers to better evaluate and understand retrieval systems.

[GLX+22] The paper discusses the development of **Manu** a cloud-native vector database, due to the rise of learning-based embedding models, embeddings vectors are widely used for analysing and searching unstructured data. Since vector collections can exceed billion-scale, the development of fully managed and horizontally scalable vector databases is necessary. The paper also discusses the basic concepts and system architecture of **Manu** in detail. The paper provides a discussion on features of **Manu**, its use cases and its evaluation by comparing it against **Milvus** [WYG+21] their previous vector database.

2.2 Natural Language Processing for Semantic Search

In this section, we introduce important concepts related to semantic search. Semantic Search has been a key element in the technology stack of major companies like Amazon, Google and Netflix for many years. However recent advancements in technology have made these technologies available to the general public. As a result, it has led to the adoption of semantic search in organizations around the globe because of its ability to unlock the potential for many applications. Semantic Search has a variety of applications like search engines, autocorrect, translation, and recommendation engines among others.

Semantic search is supported by two pillars: vector search and NLP. In this section, we will focus on the NLP pillar and the role it plays in bringing *semantic* to semantic search.

2.2.1 Dense Vectors

One of the most significant contributors to the success of modern NLP technology is the ability to represent language as vectors. The rise of NLP was sparked by the introduction of word2vec in 2013 [MCCD13b]. Word2vec was one of the earliest examples of representing text using dense vectors.

We use dense vectors Figure 2.3 to represent text so that computers can understand the human-readable text, so we convert text into a machine-readable format. The language contains a lot of information, hence we need a massive amount of data to represent even small amounts of text. Vectors are ideal for representing language as they handle huge amounts of data in an efficient way.

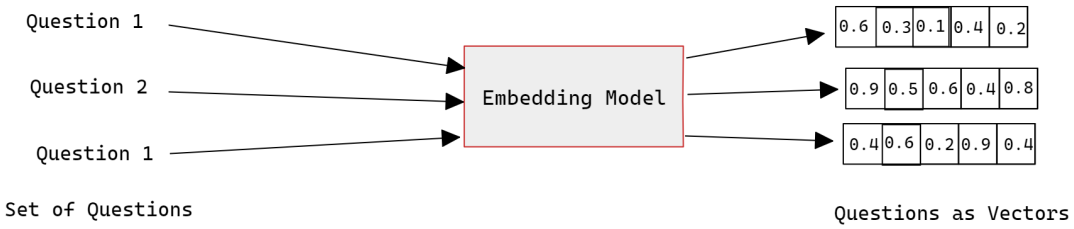

Figure 2.3: Generating Dense Vectors [Pinb]

ML algorithms work with numbers like most other software algorithms. Sometimes, we have data with columns containing only numerical values or values that can be converted into them (ordinal, categorical etc) [Pinb]. But other times we have columns with text or an entire paragraph of text. We can create vector embeddings, which are lists of numerical values, for this kind of data to perform different operations. This type of data can be transformed into a vector [Pinb]. Due to this representation of real-world objects like images, audio recordings, news articles and more as vector embeddings, we are able to quantify the semantic similarity between these objects by their closeness to each other as points in vector spaces [Pinb].

There are two options for vector representation; sparse vectors and dense vectors as shown in Figure 2.4. *Sparse* vectors are stored more efficiently and enable us to perform comparisons based on the syntax of two sentences. Let's look at an example, we have two sentences; *Peter ran from the cat toward the dog* and *Peter ran from the dog toward the cat*, which will give us an exact (or nearly perfect) match. Even though the meaning of the two sentences is different, they are made up of the same syntax (e.g. words).

While sparse vectors represent text syntax, we can think of *dense* vectors as numbers that represent semantic meaning. The words are encoded into dense, high-dimensional vectors that represent the abstract meaning and relationship of words by using numerical values [Pinb].

Sparse vectors are known as sparse because vectors are populated with information sparsely. One has to look at a thousand zeros to find a few ones(useful information). As a result, they can contain a large number of dimensions, going up to tens of thousands[Pinb]. Dense vectors also have a large number of dimensions but each dimension holds relevant information determined by a neural network. They use more memory because compressing them involves a more complex process.

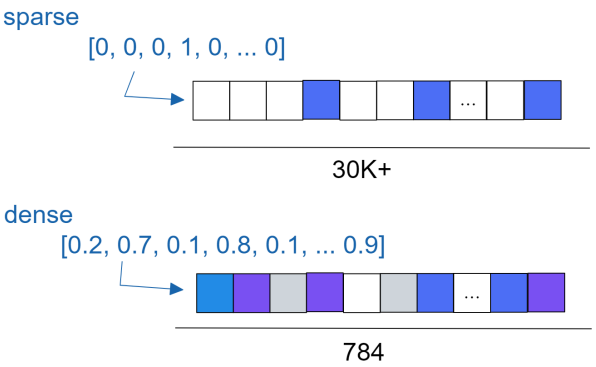

Figure 2.4: Comparison of sparse and dense vectors[Pinb]

Let's imagine we created dense vectors for each word in a book, reduced the dimensionality of those vectors and visualized them, it will give us the ability to identify relationships. As an example, the days of the week may be close to each other Figure 2.5. Also, we can perform 'word-based' arithmetic as shown in Figure 2.6. These can be achieved by using complex neural networks, which have the ability to extract patterns from a huge corpus of text data and transform them into dense vectors.

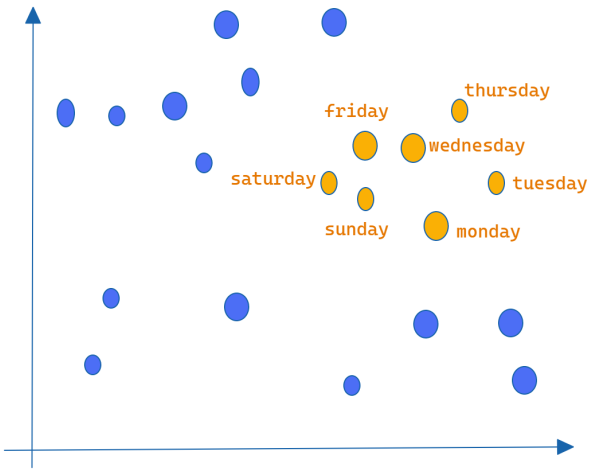

Figure 2.5: Example of the clustering of related keywords as is typical with word embeddings such as word2vec[Pinb]

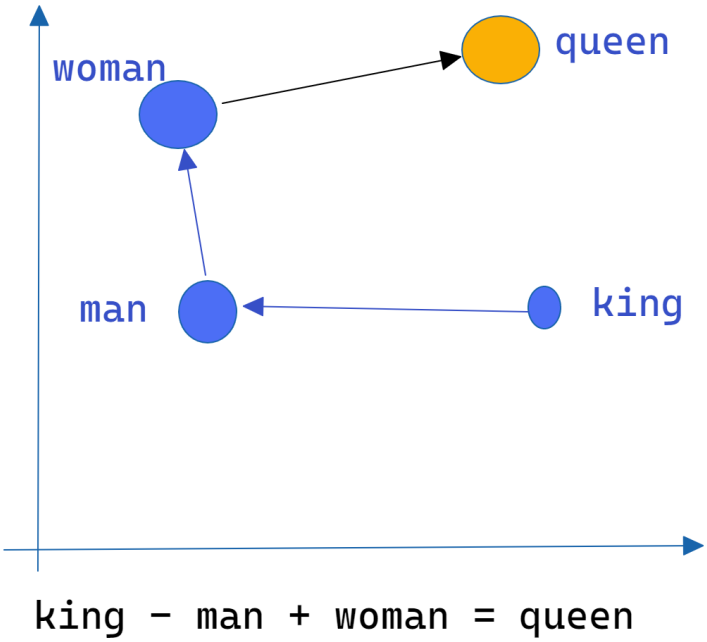

Figure 2.6: Arithmetic on word vectors [MCCD13b]

Dense Vector Generation

There are different technologies available to build dense vectors, including representing words or sentences as vectors, Major League Baseball players [Alc17] and also using both text and images to generate dense vectors. Generally, we use a pre-trained model available publicly to generate dense vectors. There are a lot of high-performance pre-trained models available for every use case, which makes it easier and faster to generate accurate dense vector representations. In some cases, one might need to fine-tune or train a model from scratch for creating embeddings that are specifically required for a particular industry or language, although it is not that common these days.

We will explore the following two methods:

- The '2vec' techniques
- Sentence Transformers

Word2Vec

Nowadays, various methods are available for building embeddings, but first, let's have an overview of the word2vec method [Pinb]. Given a sentence as input, word embeddings will be created by taking a particular word (transforming it to a one-hot encoded vector) and using an encoder-decoder neural network to map it to the surrounding words.

Figure 2.7 refers to the skip-gram method of word2vec, in which for a given word *fox*, it will try to predict the surrounding words (its context). After training, we keep the dense vector situated in the middle which can be used to embed the word *fox* for downstream language models.

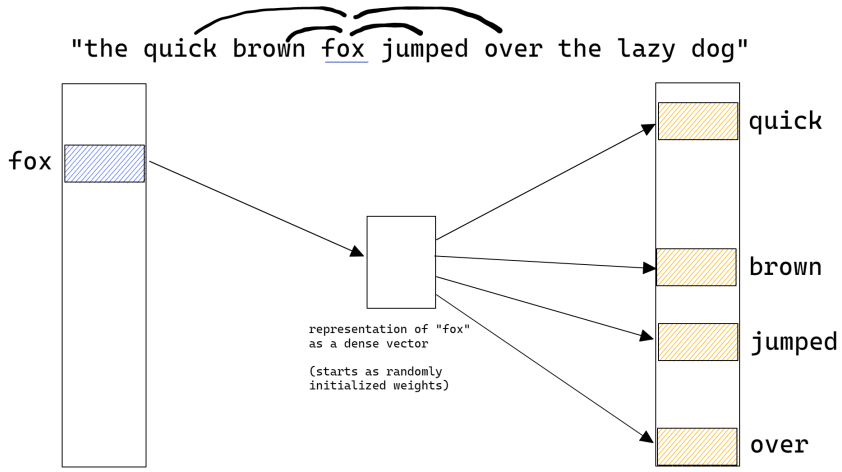

Figure 2.7: The skip-gram approach to build dense vectors embeddings in word2vec [Pinb]

In Figure 2.8 we use the continuous bag of words(CBOW) approach, which focuses on predicting the word based on its context by switching the direction of the prediction. In this case, we produce an embedding for the word *fox* located on the right-hand side.

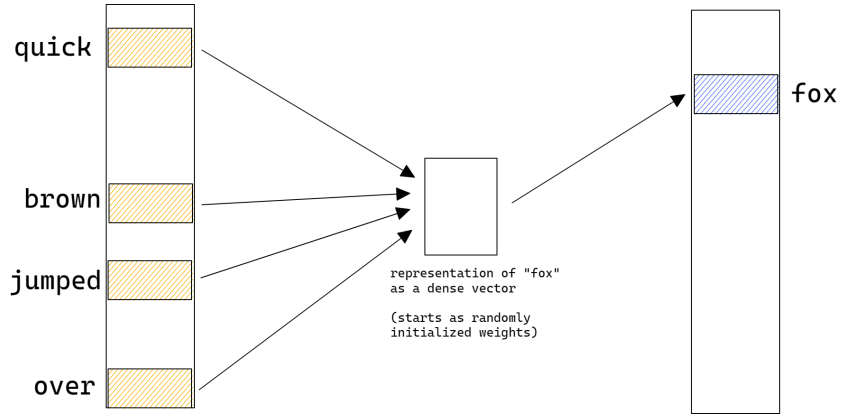

Figure 2.8: The continuous bag of words (c-bow) approach to building dense vector embeddings in word2vec [Pinb]

Both of these approaches are similar in the sense that they produce a dense embedding vector from the middle hidden layer of the encoder-decoder network. Word2vec enabled several advancements in the field of NLP. But when it came to the representation of longer chunks of text using single vectors, word2vec was inadequate. Although it enabled us to encode single words(or n-grams), it couldn't handle the representation of longer chunks by using a single vector so it could only be represented using many vectors. This limitation led to the rise of embedding approaches like sentence2vec and doc2vec, to allow for the comparison of longer chunks of text by representing it using a single vector.

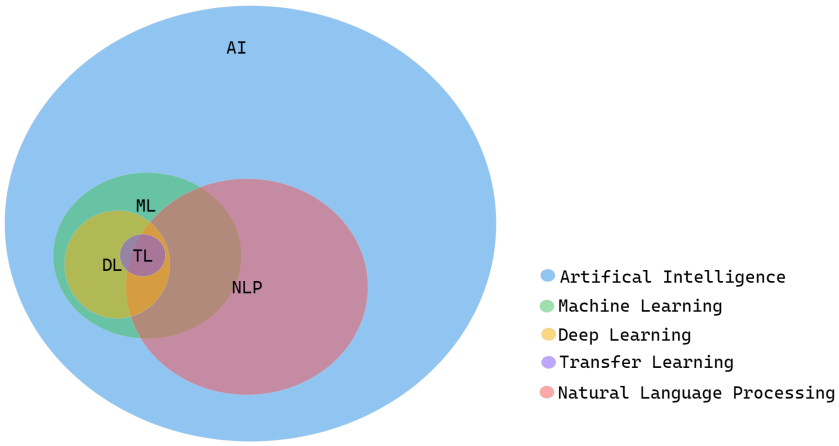

Figure 2.9: Depiction of relationships between areas for Transfer Learning

2.2.2 Transfer Learning

Transfer learning has emerged as a powerful technique combining NLP and DL as depicted in Figure 2.9. It allows researchers and practitioners to leverage pre-trained models that have already been trained on vast amounts of data, often generalizing well to a wide range of tasks. In the context of NLP, transfer learning has revolutionized the field by enabling the development of more sophisticated and accurate models with minimal or no fine-tuning. The rise of transfer learning has significantly contributed to advancements in sentence similarity tasks. The evolution of Transformer models since their introduction in 2017 [VSP+17], and their ability to produce rich embeddings have made them the dominant modern-day language models. They provide highly informative dense vectors that are useful for a variety of applications such as sentiment analysis, question answering, and sentence similarity among many others. One of the most popular of these transformer architectures is Bidirectional Encoder Representations from Transformers (BERT).

BERT produces vector embeddings for every word(token) same as word2vec. However, these embeddings are richer because of deeper neural networks and the attention mechanism that provides the ability to encode the context of words [Ping]. It uses the attention mechanism to prioritize which of the context words should have more impact on a particular embedding by considering the alignment of those context words (think of it as BERT paying attention to particular words based on the context) [Ping].

Here by 'context' we mean, where word2vec would produce the same vector for 'bank' regardless of whether it was "a grassy bank" or "the bank of Germany" - instead BERT would modify the encoding for *bank* based on the surrounding context with the help of attention mechanism and produce different vector in different contexts. But since we want to focus on comparing sentences and not words and BERT embeddings are produced for each token, it doesn't help, we need to represent our sentences or paragraphs using a single vector like sentence2vec. The first transformer built for this purpose was Sentence-Bidirectional Encoder Representations from Transformers (SBERT), a modified version of BERT [RG19]

Both BERT and SBERT use a WordPiece tokenizer - which means that each word is equal to one or more tokens. SBERT has the ability to create a single vector embedding

for sequences that contain up to 128 tokens. The limit is not ideal but more than enough to compare sentences or short paragraphs. The latest models allow for longer sequences of text too. One can create sentence embeddings quickly using the *sentence-transformers* library [RG21].

Sentence Transformers

Transformers were solely responsible for rebuilding the landscape of NLP [VSP+17]. Before that, there were decent translation and language classifying systems using Recurrent Neural Network (RNN), but they had limited ability to comprehend language which led to frequent minor mistakes, and it was not possible to have coherence over larger segments of text. After the introduction of the first transformer model, NLP left RNNs and moved to models like BERT and Generative Pre-trained Transformer (GPT), enabling semantic search and other applications[Ping]. Although, the final parts of these models are similar to RNNs, often using feedforward neural networks that give predictions as the output of the model.

The major difference is that the input that is fed to these layers has changed. These transformers are able to generate more information-rich, dense embeddings that offer significant improvements in performance despite the similar final outward layers[Ping]. These information-rich sentence embeddings are useful for comparing sentence similarity for different use cases, like:

- **Semantic Textual Similarity(STS)**: to compare pairs of sentences, which can be used to identify patterns in datasets, often used for benchmarks[Ping].
- **Semantic Search**: uses semantic meaning for IR. Given a few sentences, allows us to search using 'query' sentence to identify similar sentences. Unlocking search based on concepts rather than just specific words[Ping].
- **Clustering**: Identify and Divide sentences into groups based on similarity, which is useful for topic modeling[Ping].

In this section, we will an overview of 'sentence transformers' a different category of transformers that provide embeddings that can be applied in various semantic similarity applications. Before that, we will discuss the differences between a vanilla transformer and a sentence transformer to understand the reason behind the embeddings being more information-rich.

Transformers can be thought of as the successors, derived the previous RNN models. The old recurrent models were built by using a lot of recurrent units like Long Short-Term Memory (LSTM)s or Gated Recurrent Units (GRU)s.

The encoder-decoder networks in machine translation for encoding the source language into a context vector and then decoding this vector to the target language [Ping]. They have an issue, that they create an information bottleneck between the models, which arises because we are creating huge amounts of information and compressing it over multiple time steps into a single connection Figure 2.10. It limits the performance of the encoder-decoder model, as most of the information is lost before it reaches the encoder [Ping].

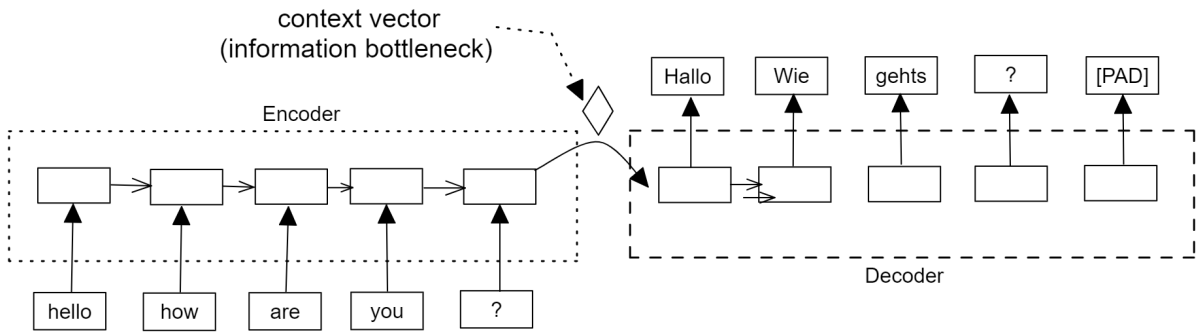

Figure 2.10: Encoder-Decoder architecture uses a context vector shared between two models, that creates an information bottleneck, requiring all information to pass through this single point [Ping]

The attention mechanism provided an answer to the information bottleneck problem by providing an additional route for information to pass [Ping] as shown in Figure 2.11. However, the process was not overloaded because it offered focused attention selectively, only on the relevant information. It passes the context vector from each timestep into the attention mechanism (that produces annotation vectors), which removes the information bottleneck and helps to retain information across longer sequences[Ping]. The model decodes one word/timestep at a time during decoding. For each step, an alignment (e.g. similarity) is computed between the word and all the annotations from the encoder. The high alignment values result in assigning more weight to the annotation from the encoder when output is produced from the decoder, which means the attention mechanism determines which words to focus on from the encoder [Ping]. The RNN encoder-decoders that used the attention mechanism offered the best performance.

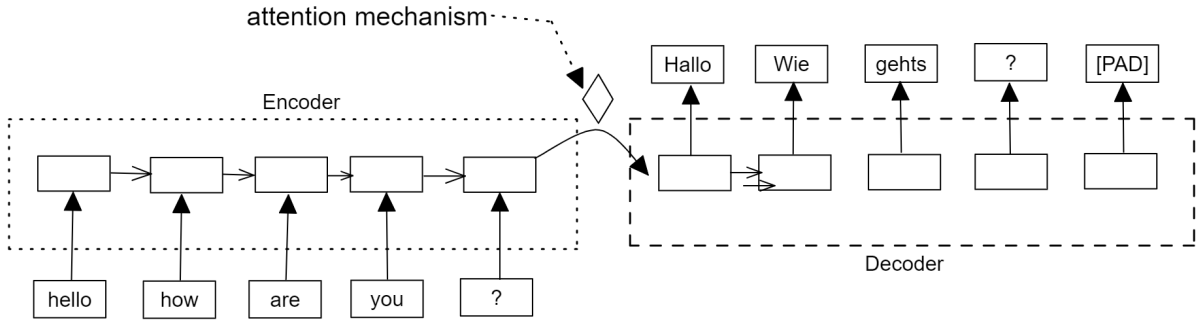

Figure 2.11: Encoder-Decoder with attention mechanism [Ping]

NLP had a major breakthrough in 2017, with the publication of "Attention Is All You Need" [VSP+17] which demonstrated that we do not need the RNN networks and we can just use the attention mechanism with some changes to get great performance. The new model was called a 'transformer', which had great performance and generalization abilities. The 'transformer' architecture has three key components:

- **Positional encoding** compensates for the key advantage of RNNs in NLP, which is the ability to consider the order of a sequence due to its recurrent nature [Ping]. It does so by adding a set of varying sine wave activations to each input embedding according to its position.

- **Self-attention** refers to applying the attention mechanism within a word's own context, such as a sentence or paragraph, as opposed to vanilla attention, which focuses on attention between encoders and decoders [Ping].
- **Multi-head attention** can be viewed as multiple parallel attention mechanisms working in unison [Ping]. This approach allows for the representation of various sets of relationships, as opposed to just a single set.

Pretrained Models

The transformer models had better generalization capabilities than RNN, which were designed for each individual use case. The transformer models made it possible to use a similar 'core' of a model and just swap some of the last layers for distinct use cases(no need to retrain the core). This new feature gave rise to pre-trained models for NLP. These pre-trained transformer models used a massive amount of data for training, which involves huge costs for companies like Google or OpenAI which are then released to be used by the public without any costs. A large number of pre-trained models are hosted on the HuggingFace[WDS+]. BERT is one of the most popular pre-trained models developed by Google AI in 2018 [DCLT18]. Many models were derived from BERT like RoBERTa [LOG+19], distilBERT [SDCW19] and ALBERT [LCG+19], that were built for tasks like classification, Question Answering, and more.

Sentence Similarity using BERT These transformer models worked using a word or token-level embeddings. not embeddings at the sentence level, which is an issue when building sentence vectors [Pind]. The approach to solving this issue, for calculating sentence similarity with BERT was to use the cross-encoder structure. Therefore, passing two sentences to BERT [Pind], that are processed in the same sequence separated by [SEP] token, following that a feedforward neural network is used as a classification head in front of BERT, to provide a similarity score as output as shown in Figure 2.12.

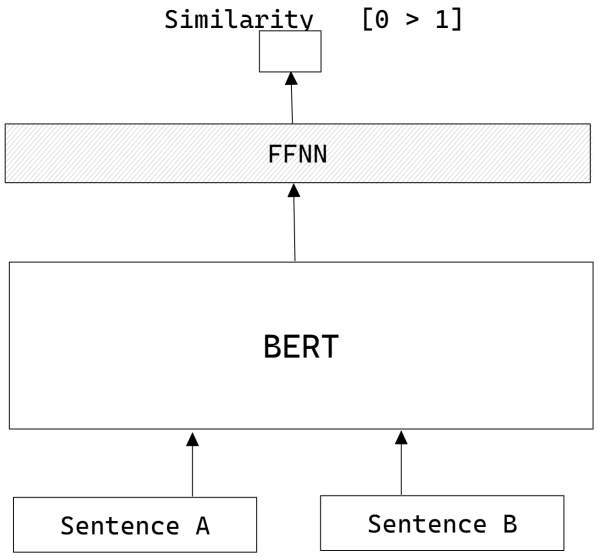

Figure 2.12: Architecture of BERT cross-encoder [Ping]

This network performed well but does provide scalability, for example, to perform a similarity search for 100k sentences, the computation for inference would need to be completed 100K times. We would prefer to pre-compute the sentence vectors, store them and use them as needed. If the representations of the vectors are good, we just need to calculate the cosine similarity between each vector. The BERT family of transformers can help to build sentence embeddings by taking the average of values across the output of all token embeddings(the input of 512 tokens, will give the output of 512 embeddings) or by using the output of the first [CLS] token(a token specific to BERT, where the output embedding is used in classification tasks) [RG19]

These two approaches provide sentence embeddings to store and compare quickly reducing the latency for search, but the accuracy of the embeddings is not good. [RG19].To solve this problem and provide an accurate model with reduced latency, SBERT was introduced and designed by Nils Reimers and Iryna Gurevych, along with the sentence transformers library [RG19]. They demonstrated a massive increase in speed in 2019. While using BERT it took 65 hours to find the most similar pairs of sentences, whereas using SBERT, embeddings are created in about 5 seconds and comparison can be done using cosine similarity in 0.01 seconds [RG19]. There are a lot of sentence transformer models that have been developed since, they are trained using similar and dissimilar sentence pairs, and they use loss functions like softmax loss, multiple negative ranking loss or mean squared error margin loss. These models are optimized for producing similar embeddings for similar embeddings and vice versa.

Sentence Transformers We provided an overview of the cross-encoder architecture for sentence similarity with BERT. SBERT is similar but it does not have the final classification head and processes one sentence at a time and uses mean pooling on the output layer to provide a sentence embedding. SBERT is fine-tuned on sentence pairs using a siamese architecture [RG19]. It can be thought of as having two identical BERTs in parallel sharing the same weights as shown in Figure 2.13. Although, in reality, it is a single BERT model.

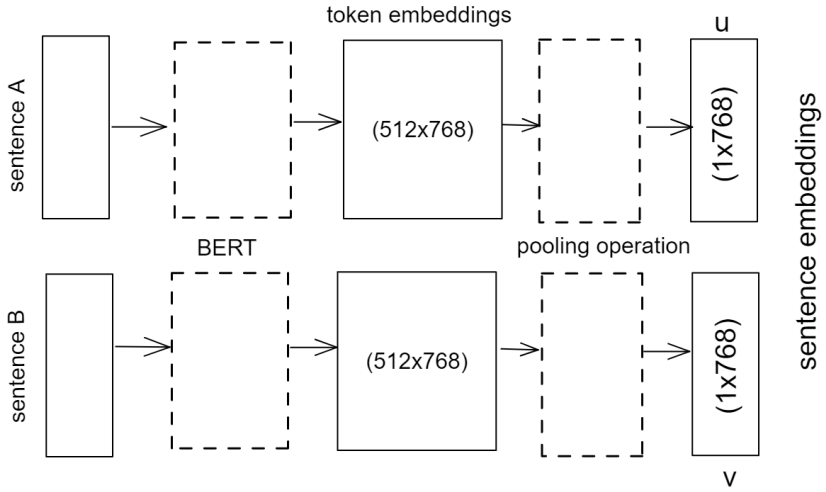

Figure 2.13: SBERT model applied to a pair of sentences. BERT model outputs token embeddings of 512 768-dimensional vectors, which are compressed into a single 768-dimensional sentence vector using a pooling function [Ping].

2.2.3 Multilingual Sentence Transformers

Multilingual Sentence Transformers can be used to map similar sentences from various languages to similar vector spaces [Pind]. For example, the sentence "I love Football" and the German equivalent "Ich liebe Fußball" (I love Football), would be viewed as the same by an ideal multilingual sentence transformer model. The model would consider "Ich mag Fußball"(I like Football) as more similar to "I love Football" than "Ich habe einen Hund" (I have a dog) as shown in Figure 2.14.

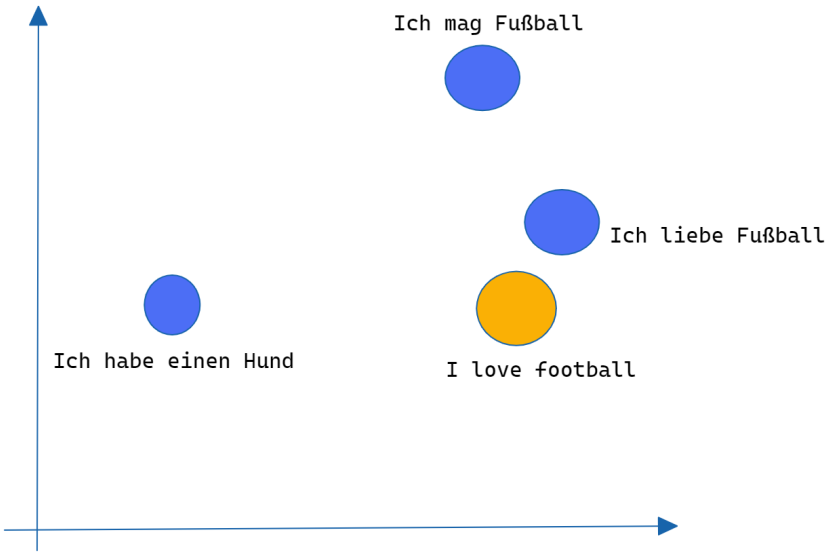

Figure 2.14: A multilingual model maps sentences from different languages into the same vector space [Pind]

Approaches to Training

The most commonly used training methods for these models make use of a contrastive training function. When the model is given a pair of sentences that are highly similar, it is optimized to provide highly similar sentence vectors. The training data is available in abundance for common languages, mainly English. Finding data for other languages is challenging. The following two training approaches rely on translation pairs rather than similarity pairs, partly or fully as they are easier to find.

Bridge Tasks based on Translation : The training is performed using two alternate datasets in a multi-task training setup:

1. A dataset in English that contains question answer (or anchor-positive) pairs (anchor-positive refers to two high similarity sentences).
2. A Parallel data that contains cross-language pairs (English sentence, German Sentence).

The idea behind this approach is for the model to learn the relationships of the monolingual sentence pairs through the dataset of a source language. Later, translate what it learnt into a multilingual scope using parallel data [CYC+18].

This approach works, but there is an alternative approach that might be better because of a few reasons:

- It requires a huge amount of data for training. The multilingual universal sentence encoder model was trained on over a billion sentence pairs [YCA+19].
- The architecture used - multi-task dual encoder [YCA+19] that optimizes two models in parallel which is harder as there must be balance in both training tasks. Optimizing a single model is hard enough.
- The results can be mediocre, if there are no hard negatives [RG20]. Hard negatives are sentences that appear similar but are not relevant or contradictory to the anchor sentence (the sentence used for comparison). Since they are difficult for the model to identify as dissimilar, the model improves by training on them.

Multilingual Knowledge Distillation The multilingual knowledge distillation method was introduced in 2020 by Nils Reimers and Iryna Gurevych [RG20]. In this method, two models are used for fine-tuning, the teacher and student models. The teacher model is already fine-tuned on a single language (most likely English) used for creating embeddings. The student model is a transformer model that has been trained on a multilingual corpus, capable of understanding multiple languages.

Training a transformer model has two stages. During the pretraining stage, the model is trained using techniques like Masked Language Modeling (MLM) to develop a language model. Whereas in Fine-tuning, the core model is further trained for a specific task such as semantic similarity, classification etc to refine it for the task. Sometimes, they are also commonly referred to as pre-trained.

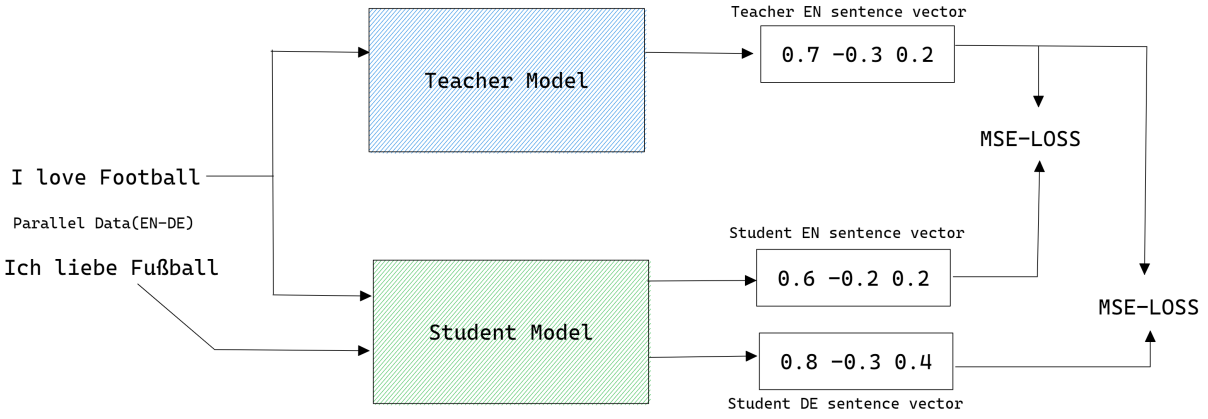

Figure 2.15: Knowledge distillation training approach [RG20]

In Knowledge Distillation, we use a parallel dataset (translation pairs) that contains translations of the sentences [RG20]. These translations are the input for the teacher and student models. For example- Let's say we have English-German pairs as depicted in Figure 2.15. The English sentence is input into the teacher and student models, which produces English sentence vectors. Then we input the German sentence into the student model, we calculate the Mean Squared Error (MSE) loss between the one vector produced by the teacher model and the two vectors we got from the student model to

optimize the student model using this loss. The student model learns to mimic the monolingual teacher model but for multiple languages [RG20]. This approach is known as multilingual knowledge distillation, it offers a great way to add more languages using existing pre-trained models. In comparison to training from scratch, it requires much fewer data since it uses data that can be found more easily - translated sentence pairs. This approach offers an efficient and effective way to extend the capabilities of sentence transformers to multiple languages by using already existing models and multilingual data.

Fortunately, it is rarely needed to fine-tune our own model. We can load and use many high-performing multilingual models from Hugging Face [WDS+], which is what we did in our thesis work. We selected the multilingual models available on Sentence-Transformers[1] [RG21], which can be found in Table 2.2. The "quora-distilbert-multilingual" is not featured on Sentence-Transformers but can be found on Hugging Face[2] [WDS+].

In the table Table 2.2 below, the column headers represent the following information about the models[RG21]:

- **Model**: the name of the model.
- **Performance Sentence Embeddings (14 Datasets)**: The average performance of the model on encoding sentences for 14 diverse tasks in different domains. Higher values indicate better performance
- **Performance Semantic Search (6 Datasets)**: The performance of models on 6 Diverse tasks for semantic search: queries/questions and paragraphs up to 512-word pieces were encoded. Higher values mean better performance.
- **Avg. Performance**: Average sentence performance and semantic search performance. Higher values reflect better performance.
- **Speed**: The speed of encoding sentences per second on a V100 GPU. Higher speed values are better.
- **Size**: The model size in Megabytes (MB).

These models listed in Table 2.2 are capable of generating aligned vector spaces, which means that inputs that are similar in different languages will be mapped closer to each other in a vector without the need to specify the language [RG20]. These models support many languages, to find semantically similar sentences but in our thesis work, we focus on English and German.

[1] https://sbert.net/docs/pretrained$_m odels.html multi-lingual-models$

[2] https://huggingface.co/sentence-transformers/quora-distilbert-multilingual

Model	Performance Sentence Embeddings (14 Datasets)	Performance Semantic Search (6 Datasets)	Avg. Performance	Speed	Model Size
paraphrase-multilingual-MiniLM-L12-v2	64.25	39.19	51.72	7500	420 MB
distiluse-base-multilingual-cased-v1	61.30	29.87	45.59	4000	480 MB
paraphrase-multilingual-mpnet-base-v2	65.83	41.68	53.75	2500	970 MB

Table 2.2: Performance Comparison of SBERT Pretrained Multilingual Models [RG21]

In the table Table 2.3, we provide the details about the teacher model and student model used for training the multilingual models and the max sequence length refers to the maximum number of tokens (words or subwords), a model can process in a single input sentence.

Model	Teacher Model	Student Model	Max Sequence Lengthl
paraphrase-multilingual-MiniLM-L12-v2	paraphrase-MiniLM-L12-v2	microsoft/Multilingual-MiniLM-L12-H384	128
distiluse-base-multilingual-cased-v1	Multilingual universal sentence encoder	distilbert-base-multilingual	128
paraphrase-multilingual-mpnet-base-v2	paraphrase-mpnet-base-v2	xlm-roberta-base	128

Table 2.3: Multilingual models with their teacher and student models and max input length[RG21]

In the table Table 2.4, we provide the number of languages a model can handle and the number of dimensions refers to the output dimensions of the model, e.g. 384 dimensions means the model represents each sentence by a vector of 384 numbers.

Model	Languages	Dimensions
paraphrase-multilingual-MiniLM-L12-v2	50+	384
distiluse-base-multilingual-cased-v1	50+	768
paraphrase-multilingual-mpnet-base-v2	15	512

Table 2.4: SBERT Pretrained Multilingual Models Supported Languages and Output Dimensions [RG21]

We selected these SBERT pretrained multilingual models because they support a wide range of languages and offer high performance in sentence embeddings and semantic search tasks. Additionally, these models have been trained using transfer learning, which enables them to generalize well to various tasks and domains. The models' speed and model size are also significant factors that make them attractive choices for our work. We can use these models to generate aligned vector spaces for inputs that are similar in different languages, enabling us to find semantically similar sentences efficiently. This discussion addresses our fourth research question as mentioned in Section 3.1.

Introduction to Zero-shot Evaluation

Zero-shot evaluation is a way to evaluate NLP models focused on IR. Previously, these models have been evaluated only on the specific tasks and domains for which they were designed. It only brought limited insights into their capability to generalize on new unseen data, also called out-of-distribution (OOD). Benchmarking-IR (BEIR) framework was created by [TRR+21] in 2021 to address these challenges and make the process of evaluating zero-shot transfer capabilities of dense retrieval models more easier and effective, by providing 18 datasets that are available publicly for different information retrieval tasks and covering different domains.

Zero-shot evaluation allows us to evaluate the generalization capabilities of the models used in our experiments, to assess how they might perform on the new tasks since that task was not part of their training data.

2.3 Semantic Search: From Keyword to Context - The Evolution of Search Methods

Semantic Search also known as Similarity search has grown rapidly [ZRB+16] due to the rise AI and ML.It aims to improve the accuracy of search [RG] by understanding the content of the search query. Traditional Search engines used to rely on lexical matches to find documents but semantic search is also able to find synonyms. The core idea for semantic search is to embed all data in your corpus(data you have) such as documents, paragraphs, and sentences to a vector space. During search time, the query from the user is also embedded into the vector space [RG], and the documents that have similar embeddings from the corpus are returned to the user. It can be used to compare data quickly and efficiently (where data can be in any format - text, audio, images, or video) and it returns relevant results. In this section, we will discuss some traditional and recent techniques for similarity search. A key difference in deciding the setup for search applications is symmetric vs. asymmetric semantic search[RG]. These are mentioned below:

1. **Symmteric semantic search**: When the search query and the data entries in the corpus are identical in length and have the same amount of content [RG]. For example, searching for similar questions: the search query could be "How can I learn Python online?" and you want to find an entry such as "How to learn Python on the internet?". In Symmetric tasks, you can also treat each entry in the corpus as a potential query, flipping the corpus and queries[RG].
2. **Asymmetrci Semantic Search**: When the search query is short (keyword or a question) and we need to find a longer paragraph that answers the query. For example, a search query - "What is Python" and we find the answer such as "Python is a programming language, used worldwide... It is high-level, general-purpose ... " [RG].

2.3.1 Traditional Similarity Search

In this section, we will cover three popular metrics to use with Similarity Search.[Pinf]

- **Jaccard Similarity**: It is a simple but powerful metric. When we have two sentences p and q, first find the number of shared tokens/words between them and divide it by the total number of tokens in both sentences.[AoE13].

$$J(p,q) = \frac{|p \cap q|}{|p \cup q|}$$

Example:

p: *I love playing football in the rain.*

q: *I love watching football at Old Trafford.*

To calculate the Jaccard similarity, we convert each sentence into a set of words.

p = {I, love, playing, football, in, the, rain}

q = {I, love, watching, football, at, Old Trafford}

Next, we calculate the intersection and union of the two sets:

$|p \cap q| = 3$

$|p \cup q| = 8$

Finally, we can calculate the Jaccard similarity using the formula:

$$J(p, q) = \frac{|p \cap q|}{|p \cup q|} = \frac{3}{8} = 0.375$$

Therefore, the Jaccard similarity is 0.375.

- **w-Shingling**: uses a similar logic as Jaccard similarity but with 'shingles'. For example, A 2-shingle of sentence "I love playing football" would like 'I love', 'love playing', 'playing football' Let's calculate the 2-shingling similarity between two sentences p and q that we used above:

  p: *I love playing football in the rain.*

  q: *I love watching football at Old Trafford.*

  Convert each sentence into a set of 2-shingles.

  p 2-shingles: {I love, love playing, playing football, football in, in the, the rain.}

  q 2-shingles: {I love, love watching, watching football, football at, at Old, Old Trafford.}

  calculate the intersection and union of the two sets of 2-shingles:

  $|p \cap q| = 3$

$|p \cup q| = 11$

calculate the 2-shingling similarity

$$J(p, q) = \frac{|p \cap q|}{|p \cup q|} = \frac{3}{11} \approx 0.27$$

The 2-shingling similarity is approximately 0.27.

There are also other metrics like Pearson Similarity, Levenshtein distance and Normalized Google Distance [Pinf]

2.3.2 Vector Similarity Search

There are several vector-building methods for vector-based search mentioned below, in this section, we will discuss TF-IDF, BM25, and BERT as they are the most commonly used ones alongside Approximate Nearest Neighbors (ANN) to power semantic/vector search.

- **TF-IDF**: born in the 1970s, is considered the grandfather of vector similarity search, which contains two parts, Term Frequency (TF) and Inverse Document Frequency (IDF). For TF, we count the number of times a word/term appears in a document and divide by the total number of items in that document [QA18]. But it doesn't make a difference between frequently occurring words and rare words like 'the' might same relevance in a sentence with the word 'football' in a sentence. It works fine until the comparison is done on longer documents or using longer queries. Ranking words like 'the', 'is', 'it' etc as high as football does not make sense. The rarer words should be scored higher, that's where we can use IDF, which measures how common is the word across all documents [Pinf]. And vector similarity search comes into play when we calculate TF-IDF for every word in our data. The vectors created by TF-IDF are sparse, so we cannot encode semantic meaning [Pinf]. For example, Let's illustrate the concept of TF-IDF in two short sentences: The cat is on the mat, and The dog is on the floor. Now, we will calculate the TF and IDF for the unique words, then compute the TF-IDF value for each word in each sentence.

- **BM25** Okapi BM25 is the successor of TF-IDF, the result of optimizing TF-IDF by normalizing the results based on the length of the document. TF-IDF does not work well when comparing several mentions [Pinf]. for example, if we have two articles with 500 words, and find that article "A" mentions "Football" six times and article B mentions "Football" twelve times - should article A be considered as half as relevant? Likely Not [Pinf]. Bm25 solves this problem by modifying TF-IDF. It leads to the TF-IDF score increasing linearly as the number of relevant tokens

Word	TF (1)	TF (2)	IDF	TF-IDF (1)	TF-IDF (2)
the	2/5	2/6	log(3/3)	0	0
cat	1/5	0/6	log(3/2)	1/5 * log(3/2)	0
is	1/5	1/6	log(3/3)	0	0
on	1/5	1/6	log(3/3)	0	0
mat	1/5	0/6	log(3/2)	1/5 * log(3/2)	0
dog	0/5	1/6	log(3/2)	0	1/6 * log(3/2)
floor	0/5	1/6	log(3/2)	0	1/6 * log(3/2)

Table 2.5: Term Frequency and Inverse Document Frequency

increases. The BM25 vectors are also sparse vectors, not able to encode semantic meaning [Pinf].

$$\text{BM25}(d,q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{\text{TF}(q_i,d) \cdot (k_1+1)}{\text{TF}(q_i,d) + k_1 \cdot (1 - b + b \cdot \frac{\text{DL}(d)}{\text{avgDL}})} \tag{2.1}$$

- **BERT**: BERT is a very popular transformer model, used for a lot of things in NLP. It has 12(or so) encoder layers to encode massive amounts of information into dense vectors. A dense vector usually contains 768 values, and for each sentence encoding, we have 512 of these vectors. These vectors are numerical representations of language. These vectors can be extracted from different layers but usually from the final layer. Then we can use a similarity metric like cosine similarity to calculate their semantic similarity. But since each sentence is represented by 512 vectors and not one vector, it is a problem. To solve that problem BERT was adapted to Sentence-BERT[RG19] that allowed to create a single vector that represents the sequence in full, also known as a sentence vector [Pinf].

Pros and Cons of Semantic Search

- *Improved relevance*: Semantic search goes beyond simple keyword matching and takes into account the context, meaning, and intent behind the user's query. This results in more accurate and relevant search results.
- *Handling synonyms and ambiguity*: Semantic search can handle synonyms, enabling it to provide relevant results even when users use different terms to describe the same concept. It can also better handle misspelled queries.
- *Multilingual support*: Semantic search can handle multiple languages by using pre-trained models to obtain embeddings, making it a better alternative for building multilingual search applications.
- *Scalability*: Semantic search can scale well when new languages and data are added as it does not require separate indexes or pipelines for each language.
- *Query expansion*: Semantic search can automatically expand queries with related terms or concepts, improving recall and ensuring that the relevant information is returned to the users.

- *Computational resources*: Semantic search requires reasonably more hardware resources, as it depends on pre-trained neural networks to obtain embeddings which require more computing power. *Training data*: Developing accurate semantic search models often requires large amounts of labelled training data, which can be difficult and time-consuming to obtain. *Model interpretability*: Deep learning models used in semantic search can be challenging to interpret, making it difficult to understand why certain results are returned and how to improve them.
- *Maintenance*: As new data and languages are added, the models have to be retrained or fine-tuned, which can be time-consuming and resource-intensive.

Semantic search offers a significant improvement over traditional keyword-based search, as it goes beyond simple keyword matching and considers the meaning and intent behind a user's query. This results in more accurate and relevant search results, and better handling of synonyms and ambiguity. Furthermore, semantic search is well-suited for multilingual applications, as it can easily handle multiple languages without the need for separate indexes or pipelines for each language. This makes semantic search a more suitable and efficient approach for building a multilingual search application and provides an optimized search experience to the user.

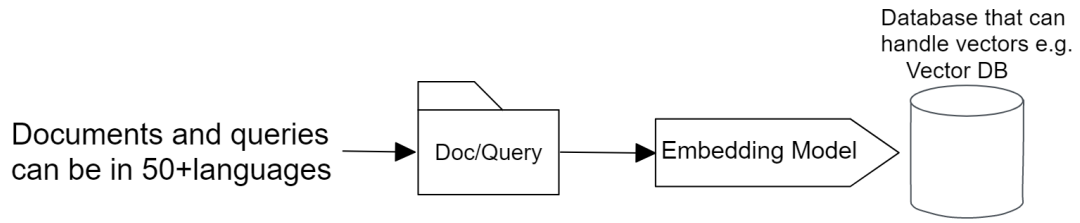

Figure 2.16: Multilingual Semantic Search

2.3.3 Nearest Neighbour Indexes for Similarity Search

Search is easy when we take one item at a time to compare it to another item. Although, when there are millions(or billions) of 'other' items for comparison, it becomes complex [Pinf]. For Search to be useful, it is essential that search is fast and accurate [Pinf]. Our primary interest is to have a more efficient search process. In this section, we will look at different indexes for storing vectors and then using them for smart similarity search and we will also look at the ML algorithm K-Nearest Neighbor (KNN) that forms the basis of similarity search.

K-Nearest Neighbor Algorithm

In ML the concepts of similarity and proximity are concepts that are closely related. It means that the closer the two data points are, they become more similar to each other than the rest of the data points [Zha16]. KNN is a supervised learning algorithm used for both classification and regression. It predicts outcomes for new data points(testing data) based on their similarity with known data (training) points. It assumes that data with similar characteristics belong together and uses a distance measure to calculate similarity.

It is a non-parametric model, as it does not learn any parameters from the training data and just works by remembering the training data in memory [Pinc]. KNN is used in many areas because of its flexibility like Agriculture, Finance, Healthcare and Internet. The algorithm captures the whole dataset and makes a prediction for each new data point based on similar data points that exist in the data set. It makes new predictions using "known" data only [Pinc]. It calculates the distance between the new data point and the other data points in the dataset using a distance measure, the data points that are closer to the new data point become the "K-neigbors". "K" is the number of neighbors the algorithm will use to make a prediction for the new data point [Pinc]. Some downsides include choosing the K and distance metrics being crucial for performance. It is very sensitive to outliers and imbalanced datasets [Pinc].

ANNS Indexes

An index is used to store the vector representations of the data to enable vector similarity search. When choosing an index, factors such as the size of the dataset, search frequency, search quality and search speed [Pinh]. The core idea of ANNS is to focus on improving retrieval efficiency by sacrificing a little bit within an acceptable range [Zil]. It differs from accurate retrieval where the most accurate results are returned which takes time, instead, it searches for the nearest neighbors of the target [Zil]. The ANNS vector indexes can be divided into four categories based on their implementation - Tree- based index, Graph-based index, Hash-based index, and Quantization-based index [Zil].

1. **FLAT**: indexes are 'flat', as the vectors that are inserted into the index are not modified because there is no clustering or approximation of vectors involved [Pinh]. A Flat index produces the most accurate results, so it provides you high level of search quality but the search times are on the higher side. The search query is compared against every other vector stored in the index by using a distance measure. After the distances are calculated, the nearest k matches will be returned as results, the K-nearest neighbors (KNN) search [Pinh]. Flat indexes are used when search quality has a high priority and search time does not matter OR for smaller datasets.

   But for scenarios when the search needs to be faster, there are two possible approaches:

   - Reduce vector size - by using techniques such as dimensionality reduction or representing vector values with fewer bits. [Pinh].
   - Reduce search scope - The second approach is to use clustering or organizing tree-based structures based on similarity, distance or particular attributes to minimize the search scope that will restrict the search to clusters that are closer or filter using similar branches, it reduces the number of vectors that are used for comparison with the search query.

   These approaches are based on using the ANN search rather than using exhaustive nearest neighbors search. Thus, allowing us to bring a balanced approach to search applications and prioritize both search quality and search speed.

Index	Classification	Scenario
FLAT	N/A	• Relatively small dataset • Requires a 100
IVF_FLAT	Quantization-based index	• High-speed query • Requires a recall rate as high as possible
IVF_SQ8	Quantization-based index	• High-speed query • Limited disk and memory capacity • Has CPU • Minor compromise in recall rate
IVF_PQ	Quantization-based index	• Very high-speed query • Limited memory resources • Substantial compromise in recall rate
HNSW	Graph-based index	• High-speed query • Requires a recall rate as high as possible
Annoy	Tree-based index	• Low dimensional vectors

Table 2.6: Approximate nearest neighbor Indexes

2. **Hierarchical Navigable Small World Graphs(HNSW)**: Hierarchical Navigable Small World Graphs (HNSW)-based ANN search are high performing and consistent indexes [B^+] adapted from navigable small world (NSW) graphs - a graph structure that contains vertices connected using edges to their nearest neighbors[Pinh]. The Navigable Small World (NSW) in the graph means that all vertices inside the graph have a short average length path to other vertices in the graph, although they might not be directly connected [Pinh]. For example, In 2016 Facebook could consider each user as a vertex and their friends as the nearest neighbors. Although there are 1.59 billion active users, the average number of steps required to travel from one user to another was 3.57 [BK]. This is one example of high connectivity in large networks. We can construct a HNSW graph by taking a NSW graph and dividing it into multiple layers [Pinh] where each layer removes intermediate connections between vertices. This provides a hierarchical structure where vertices in each layer are connected to only a subset of vertices in the previous layer. Thus, making the nearest neighbor search more efficient by restricting the search to a smaller number of vertices in every layer, as the required number of distance computations is minimized. HNSW is one of the best-performing indexes for bigger datasets and high dimensionality [Pinh]. HNSW provides high-quality search at very fast search speeds but requires a higher amount of memory [Pinh].

3. **Inverted File Index(IVF)**: reduces search scope by clustering. It's a popular index that provides high search quality and reasonable search speed [Pinh]. It uses a concept called Voronoi diagrams - known as Dirichlet tessellation. It has the following types:

    - **IVF_FLAT**: divides vector data into 'nlist' clusters and compares the distance between the target input vector(the search query) and the centre of each cluster[Zil]. By setting the number of clusters to query 'nprobe', the similarity search results are returned based on comparisons between the target input with the vectors in the most similar clusters only, it reduces query time significantly [Zil].

    - **IVF_SQ8**: IVF_SQ8(Scalar Quantization) is a better option than IVF_FLAT when disk, CPU or GPU memory resources are limited. It can convert each FLOAT(4 bytes) to UINT8(1 byte) using scalar quantization reducing disk, CPU, and GPU memory consumption by 70-75% [Zil]. For example, if the original 1B SIFT dataset is 476 GB, its IVF_FLAT index files will be 470 GB whereas IVF_SQ8 index files require just 140GB of storage [Zil].

    - **IVF_PQ**: The PQ (Product Quantization) method decomposes the high dimensional vector space into cartesian products of m low-dimensional vector spaces and quantizes them [Zil]. It calculates the distance between the target vector and the clustering centre of each low dimensional space which reduces the time and space complexity of the algorithm. It performs IVF index clustering before quantizing the product of vectors. It has an even smaller index file than IVF_SQ8 at the cost of losing accuracy during vector search.

4. **ANNOY**: ANNOY (Approximate Nearest Neighbors Oh Yeah) index that uses a hyperplane to partition a high-dimensional space into multiple subspaces which are stored in a tree structure [Zil]. The two parameters that can be optimised for ANNOY are 'n_trees' and 'search_k'. 'n_trees' is set during the index building

phase so it influences the build time and size of the index [Zil]. A higher value leads to more accurate results but increases the index size [Zil]. 'search_k' is set at run time and affects search performance. A higher value returns result with greater accuracy but take more time [Zil]. It defaults to 'n*n_trees' if 'search_k' is not provided where n is the number of approximate nearest neighbors [Zil]. 'search_k' and 'n_trees' are independent, i.e the value of 'n_trees' will not affect search time if 'search_k' is constant and vice versa [Zil]. Setting 'n_trees' to a large value given the amount of memory available and also setting 'search_k' to a large value given the time constraints for the queries is recommended [Zil].

In this section, we discussed the importance of various indexes for storing vectors used in similarity search. We explored the KNN algorithm, a widely-used supervised learning algorithm, as the basis for similarity search. We also examined different ANN indexes, including FLAT, Hierarchical Navigable Small World Graphs (HNSW), Inverted File Index (IVF), and ANNOY, each with their unique characteristics. These indexes enable a balanced approach to search applications by prioritizing both search quality and search speed, depending on the specific requirements and constraints of the application.

2.3.4 Evaluation Measures

Evaluation of Search Applications (Information Retrieval systems) is key to making informed decisions about the performance of the system. Evaluation measures help us to measure the performance and quality of the search system so that we can understand what works well and what does not work in retrieval. There are two categories: online and offline metrics as shown in Figure 2.17.

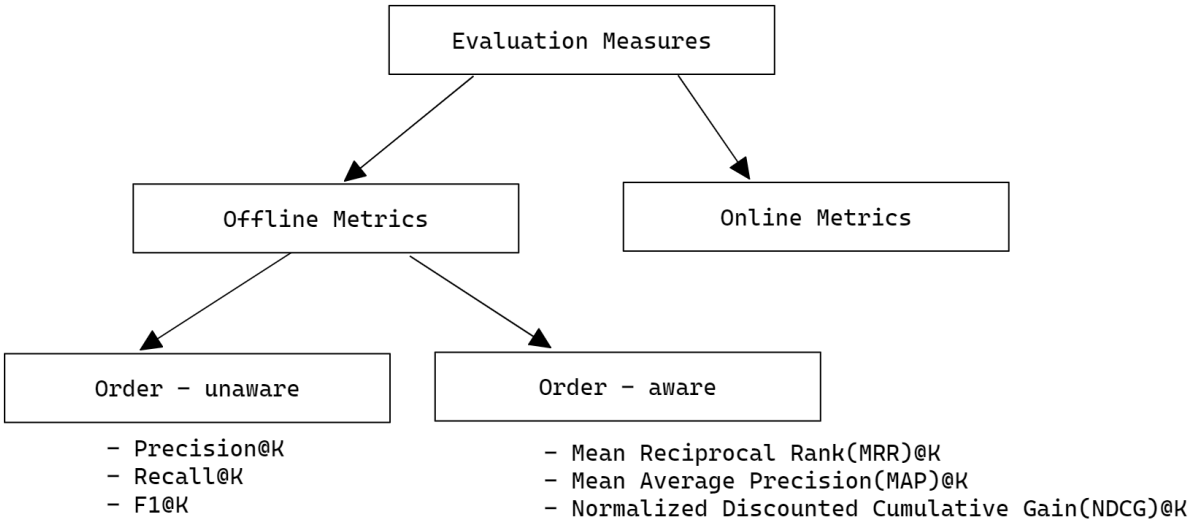

Figure 2.17: Evaluation Measures [BA15]

Online Metrics are tracked when the IR system is live and being utilized by users. These metrics take into consideration the user interactions, like whether the user clicked on a particular ad or a recommended item from Amazon, also known as Click Through Rate (CTR). There are a lot of online metrics, but all of them are related to some form of user interaction [CZL+17].

Offline Metrics are calculated before the deployment of a IR system. These metrics assess whether the system returns relevant results when searching for items. There are many offline metrics available to assess the performance of a search system as discussed in [BA15]. In our work, we discuss the selected offline metrics for evaluation in our own experiments. They can be further divided into two categories; *order-aware* and *order-unaware*. We selected two metrics Recall@K and MRR@K according to our use case and requirements for our experiments. They are discussed below:

1. **Recall@K**: is very popular because of its simple interpretation [Pine]. It measures the number of items that are relevant were returned by the system against the total number of relevant items that exist in the dataset. In the equation below: the true positives are the number of items that are relevant and were correctly retrieved while the false negatives are the number of relevant items that were not retrieved by the system. K refers to the number of items that are returned by the system. The returned value is between 0 and 1.

$$Recall@K = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{2.2}$$

For example: For K = 2, TruePositives = 1 and FalseNegatives = 3; the recall@2 score is calculated as

$$Recall@2 = \frac{1}{1+3} = 0.25 \tag{2.3}$$

As the value of K increases, it is able to return more relevant results, therefore the performance of the system improves as the scope of retrieved items increases. The following are Positive and negative points about using Recall@k

- It is easy to interpret. With a perfect score of 1.0 indicating that all the relevant items are returned. A smaller value of K makes it harder for the system to do well with Recall@k.
- Since setting K to near or the maximum number of items in the dataset will return a perfect score, which can be deceptive. It is better to use Recall@k in combination with other evaluation metrics.
- There is another problem that is an order unaware metric [BA15]. For example, if we used Recall@4 and got one result that is relevant at rank one, we would get the same value if we get the result that is relevant at rank four. Obviously, it is better to get the result that is relevant at a higher rank but it does not account for this order.

2. **Mean Reciprocal Rank(MRR@K)**: is an order-aware metric, which means returning a result that is relevant at rank one will give a better score [BA15], than returning a relevant result at rank 4.

$$MRR = \frac{1}{Q}\sum_{q=1}^{Q}\frac{1}{rank_q} \tag{2.4}$$

Q is the total number of queries, q is a particular query, and rank_q is the rank of the first result that is actually relevant for the query q. For example: Let's say, we have a total number of Queries = 3. and their ranked results as:

- Query 1: relevant results ranked at positions 3, 6, 7, 12, 15, 20.
- Query 2: relevant results ranked at positions 1, 2, 4, 5, 8, 11.
- Query 3: relevant results ranked at positions 2, 5, 8, 9, 10, 13.

To calculate MRR@3, we compute the reciprocal rank of the first relevant result for each query and then do the average for all queries:

- Query 1 has the first relevant result at position 3, so the reciprocal is $\frac{1}{3}$.
- Query 2 has the first relevant result at position 1, so the reciprocal is $\frac{1}{1}$.
- Query 3 has the first relevant result at position 2, so the reciprocal is $\frac{1}{2}$.

so, MRR@3 is the average of these:

$$MRR@3 = \frac{\frac{1}{3}+\frac{1}{1}+\frac{1}{2}}{3} = 0.78 \tag{2.5}$$

Therefore, it indicates that on average the first relevant result is ranked at position 1.28 (1/0.78), which shows good performance for a search application. Following are some advantages and disadvantages of Mean Reciprocal Rank (MRR):

- It is aware of the order, which is helpful in use cases where the rank of the first relevant result is important like chatbots or question answering.[Pine]

2.3.5 Similarity Metrics

In this section, we discuss the three most common similarity metrics used for comparing the similarity between vectors. The metrics are mentioned in Table 2.7 along with the properties of vectors that influence the metric[Pini]. We will also try to understand what it means when we say two vector embeddings are similar, in the context of specific use cases. For example, two vectors that represent words may be considered similar if they are used in similar contexts or represent similar ideas in the field of NLP. Whereas in recommender systems, two vectors represent choices of the user can be similar if they made the same choices or have the same interests.

Name	Vector Properties Considered
Euclidean distance	Magnitudes and direction
Cosine similarity	Only direction
Dot product similarity	Magnitudes and direction

Table 2.7: List of Similarity Metrics

- **Euclidean Distance**: is the distance measured by a straight line between two vectors in an n-dimensional space. It is calculated as the square root of the sum of

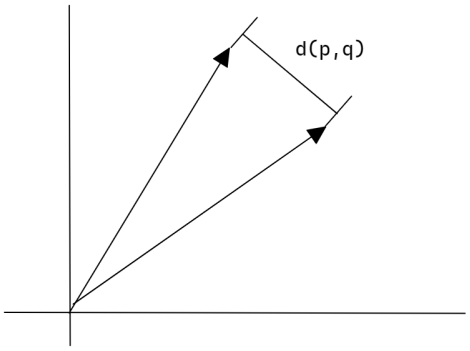

Figure 2.18: Euclidean distance in two dimensions [Pini]

the squares of the differences between the corresponding component of the vectors. Here p and q are two vectors, the equation below calculates the difference between the first components of two vectors until the n^{th} components. [MK16].

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

Euclidean distance is sensitive to both the scale and location of the vectors. Thus, larger values of vectors will result in a larger Euclidean distance than smaller values

of vectors, even though the vectors might be similar [Pini]. It is a simple and straightforward similarity metric which measures the distance between the values of the vectors that are being compared. If Euclidean distance is small, it means that the values of the coordinate in the vectors are closer [Pini]. When the embeddings consist of information such as counts or measures of things [Pini] it is more helpful because it is sensitive to magnitudes.

- **Dot Product Similarity**: is the sum of the products of the vectors components. The dot product for p and q vectors can be calculated using the following equation:

$$\mathbf{p} \cdot \mathbf{q} = \sum_{i=1}^{n} p_i q_i = p_1 q_1 + p_2 q_2 + \cdots + p_n q_n$$

p and q are the vectors that are being compared. p_i and q_i are the components for the particular vectors. The dot product can also be represented with the following equation that is mentioned below:

$$\mathbf{p} \cdot \mathbf{q} = |\mathbf{p}|\,|\mathbf{q}| \cos\theta$$

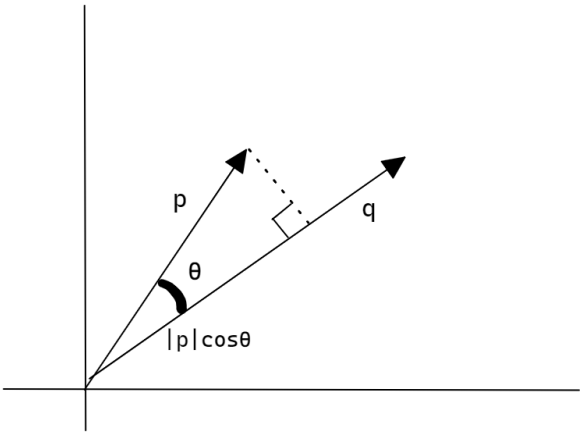

Figure 2.19: Dot Product in two dimensions [Pini]

It is a scalar value, either positive, negative or zero that depends on the angle between the vectors. If the angle is less than 90 degrees, it is positive and if it is greater than 90 degrees then it is negative. If the two vectors are at a right angle to each other, then the dot product is zero [Pini]. It can be influenced by the length and direction of the vectors. Vectors that have identical lengths but different directions, will result in a larger dot product if the two vectors are pointing in the same direction and smaller if they are pointing in opposite directions [Pini]. For example, if the word "happy" has a larger magnitude than the word "content", it may suggest that "happy" is more intense or commonly used than "content", even though they are synonyms.

- **Cosine Similarity**: measures the angle between two vectors, calculated by taking the dot product of vectors and dividing by the product of their magnitudes [MK16]. It is only influenced by the angle between the vectors, the size of the vector has no influence which means that vectors with different values but the same direction will

have the same cosine similarity value. Cosine similarity for two vectors p and q can be represented by the following equation:

$$sim(p, q) = \frac{p \cdot q}{||p|| \cdot ||q||}$$

Here p and q are the vectors that are being compared. "." refers to dot product, and the denominator represents the lengths of the vectors. Cosine similarity ranges from -1 to 1, where 1 indicates an angle of 0 (similar vectors), o means perpendicular and -1 means that vectors in opposite directions(dissimilar).[Pini] If a model has been

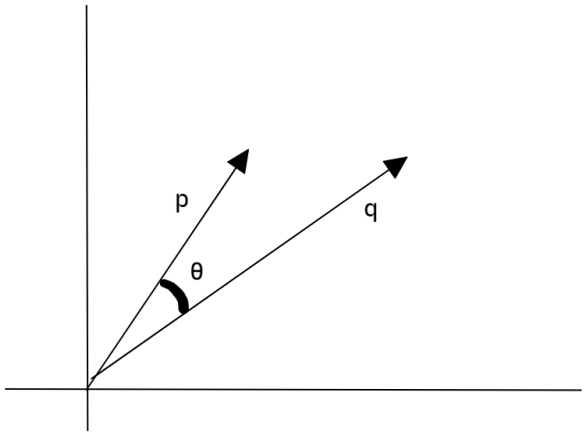

Figure 2.20: Cosine Similarity in two dimensions [Pini]

trained using cosine similarity, then use cosine similarity or a normalized version of the dot product as a similarity metric. They are the same mathematically, the choice is specific to the use case [Pini]. for example, Semantic Search is an ideal use case for using cosine similarity [Pini] as well as for document classification problems. Also, recommendation systems can use cosine similarity to recommend new items to users based on their past choices. But, it is not suitable for use cases where magnitude is important for finding similarity, e.g. finding similar images based on pixel values.

The general principle is to use the same similarity metric to build the index for your semantic search use case, which was used to train your embedding model [Pini]. Therefore, we have chosen Cosine Similarity as a similarity metric for our experiments in this thesis work. This discussion addresses our third research question as mentioned in Section 3.1.

2.4 Databases

In our thesis work, we compare relational databases and vector databases to understand how they differ specially in terms of handling vector data. In this section, we briefly discuss relational databases, and how they enable vector search and handle vector data, then we discuss vector databases in detail, comparing different solutions that are available to enable vector search. Firstly, let's discuss the different types of data generated in the world. In today's world, we as a society generate huge amounts of data by using and interacting with various platforms every day. This data is generally categorized into three types:

- **Structured data**: When data can be logically expressed and stored in a table, it is known as structured data. Structured data follows a strict pattern and specifications. It is mainly managed and stored using relational databases [WMW+22].
- **Unstructured data**: When data has no regular or complete structure nor a predefined model [WMW+22] and cannot be managed or stored using tables in databases, it is called unstructured data. Some examples are text, images, audio, video etc.
- **Semi-structured data**: It lies between structured and unstructured data. Examples include Extensible markup language (XML), JavaScript Object Notation (JSON) and log files. Semi-structured data does not conform to a rigid structure, but it does contain relevant tags that can be used to separate semantic elements to stratify records and fields [WMW+22].

A database is a collection of data or information that is stored in a computer system that follows a particular structure [ČK19]. A database offers persistent storage of data and is managed by a software called Database Management System (DBMS) using different query languages that allow efficient access to that data to perform different operations like insertion, deletion, updation or extraction according to the requirements of users. There are different databases that have been developed over the years, to handle different needs and requirements. They are mainly categorized into two categories - Relational databases and Non-Relational Databases. We will briefly discuss Relational Databases in this thesis work since they are fully matured and widely used for handling and storing data in production applications.

SQL Databases SQL databases are popularly known as relational databases. They store data in the form of tables with rows and columns where a row represents a single record of data. The data records are uniquely identified with a column known as the primary key which contains unique values. For linking two tables together, foreign keys are used, that are stored in both tables [KKH04]. They use SQL(Structured Query Language) as their primary query language. They are widely used and highly popular in every industry to handle structured data, popular ones are Oracle, MySQL, Microsoft SQL Server and PostgreSQL [Sta]. The major advantages of relational databases in comparison to non-relational databases are as follows:

- SQL databases offer a simple query language for data operations that is very powerful and advanced [KKH04].

- SQL databases follow ACID (atomicity, consistency, isolation, and durability) properties to ensure data stability and security [KKH04].

- SQL databases provide the ability for normalization of data, to prevent redundant data [KKH04].

SQL databases natively do not offer support to store and manage vector data. Postgres plugins [Gro21] were developed to enable vector search. Also two OLAP database systems, AnalyticDB-V [WWW+20] and PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension (PASE) [YLFW20] were developed to support vector data using a table column to store them but these options lack optimizations that are specifically tailored for managing and querying data [GLX+22]. In our thesis, we decided to use an extension built on core ideas from [YLFW20] and is open source and available at PostgreSQL Extension Network [Anka] and on github [Ankb] for further comparison with a vector database.

2.4.1 Introduction to Vector Databases

Vector data management solutions initially went through two stages of development [GLX+22]. In the first stage, the solutions were libraries (e.g., Facebook Faiss[JDJ21], Microsoft SPTAG[CWL+18], HNSWlib[Mal18], Annoy[Spo21]) and plugins (e.g. Elasticsearch plugins [Ela21] and Postgres plugins [Gro21]) for vector search. For current applications, they are insufficient as full-fledged management functionalities are required e.g. recovering failures, online data update and distributed execution for scalability [GLX+22]. The two OLAP systems [WWW+20, YLFW20] also do not cover the required optimizations for vector data. In the second stage, solutions are full-fledged vector databases such as Vearch, Vespa[Ves22], Weaviate[ST22], Vald[vda22], Qdrant[Qdr22], Pinecone [Pin22] and Milvus [WYG+21]. In this section, we will provide an introduction to a vector database, its practical applications and the required capabilities of a vector database.

As complex data keeps growing at a rapid speed and growing integration of AI technologies for building various applications, it has become difficult for traditional relational databases that were built for the purpose of storing structured data [Pinj]. With the rapid adoption of Machine Learning techniques that help to represent this complex data by converting them into vector embeddings. These vector embeddings have the ability to describe complex data objects as numerical values in hundreds or thousands of different dimensions [Pinj]. This has led to the rise of vector databases, which can store and provide the ability to use this complex unstructured data like documents, images, videos, etc for adding vector search capabilities to different applications to provide a better search experience for the end user [Pinj]. They have been built for the purpose of handling the unique structure of vector embeddings. They help to provide superior search capability for an application by enabling easy search by indexing vectors i.e. storing them in an organized manner for efficient retrieval by comparing values and determining values that are most similar to each other [Pinj].

Vector Databases A vector database is designed for indexing and storing vector embeddings for providing quick retrieval and similarity search, as well as Create, Read, Update and Delete capabilities, along with metadata filtering and horizontal scaling [Pinj]. Vector databases excel at performing similarity search, also called vector search. Vector search allows users to find what they are looking for by finding similar information stored in the index, without needing to know the exact keywords or other metadata classifications related to the stored data. Vector search can return similar or nearest-neighbour matches, by giving more comprehensive results which otherwise might have remained hidden [Pinj].

- vector: an array of numerical measurements used in machine learning to describe and represent the various characteristics of an object [Pinj].
- database: a huge collection of data organized by computer for quick search and retrieval [Pinj].

Applications of Vector Databases

The most common reason to use a vector database is to perform a vector search. Vector search is used to compare the similarity of many objects to a search query. In order to find similar objects, the search query is converted into a vector using the same ML embedding model used to create vector embeddings for all the objects [Pinj]. The vector database uses similarity to find similar objects and provide accurate results and discard irrelevant results that traditional search technology might have returned [Pinj].

1. Semantic Search: Generally, traditional search engines for text and documents use methods like Lexical search i.e. relying on finding patterns and matching exact words or matching phrases of words; whereas modern search engines use an approach known as semantic search that aims to understand the meaning of a search query or question and its context [Pinj]. By using vector databases to store and index vector embeddings that are derived by using NLP models which help computers to understand language i.e. the meaning and context of strings of text, words, sentences or whole documents to return more precise and useful search results [Pinj]. Using natural language queries offers a superior user experience, and helps the user to find the desired information quickly.
2. Using Similarity Search for Images, Audio, Video, JSON and other unstructured data: Datasets containing images, audio, video and other unstructured forms of data are difficult to classify and store in a conventional database [Pinj]. It requires manually applying keywords, descriptions, and metadata to each object. Different people may classify objects differently; which leads to searching for complex data unreliable. It requires the user that is searching for information, to have knowledge of how data is stored and to create queries that are aligned with the data model[Pinj].
3. Recommender Systems: creating a recommendation system for an online retailer to make recommendations on buying similar items based on purchase history or for media streaming companies to create a list of next song recommendations based on the past history of songs, the user has listened to are great use cases to the use the power of a vector database as a solution[Pinj].

4. Finding Duplicate Documents: another area where we can use vector similarity search for identifying duplicate records or documents and remove them depending on the use case of the application[Pinj].

5. Detecting Anomalies: So far we have discussed that Vector databases are good at finding similar objects but they can also find objects that are not similar to a particular record or document, making them ideal to use for applications like anomaly detection for bank transactions, in areas of threat assessment and more[Pinj].

Required Capabilities of a Vector Database

1. **Vector Indexes**: Vector databases use algorithms that are specifically designed for indexing vectors and retrieving vectors efficiently. Every use case is different and prioritizes different requirements from accuracy, latency or memory usage which can be fined tuned using different algorithms [Pinj]. There are also similarity and distance metrics such as Euclidean distance, cosine similarity etc, some have better recall and precision performance than others. Vector databases use "nearest neighbor" indexes to measure similarity between objects [Pinj] to another object or a search query. When there is a large index, it is problematic for traditional nearest neighbor search that compares every vector with the search query which takes a lot of time. ANN solves this problem by using approximation and returning the best guess of vectors [Pinj]. HNSW, Inverted File Index (IVF) or Product Quantization (PQ) are popular techniques to build efficient ANN indexes. Each technique focuses on improving a specific performance property, PQ focuses on reducing memory, and HNSW or IVF focuses on fast but accurate search times [Pinj].

2. **Single-Stage Filtering**: Filtering allows to limit results based on the metadata of the vector which can help in improving relevance. *Post-filtering* applies ANN search first and then limits returned results using metadata filter restrictions [Pinj]. ANN returns similar results but has no idea how many (if any) will match the metadata filtering criteria. It is fast but might return a small number of vectors to match the filter if any at all. In comparison, *Pre-filtering* vectors with metadata shrink the dataset [Pinj] and might return relevant results but it slows the performance because the filtering criteria is applied to each vector in the index first. *Single-stage filtering* is a must for effective vector databases [Pinj]. It provides the benefits of both filtering techniques above by integrating vector and metadata indexes into a unified index, delivering relevant results at high speeds.

3. **Data Sharding**: ANN algorithms are able to search vectors with great efficiency but regardless of the efficiency, hardware limits what's possible on a single machine. Scaling vertically by increasing the capacity of the machine is one option but eventually, there is a limit in terms of cost and availability of high-capacity machines [Pinj]. A vector database that lacks scalability will pose limitations on the efficiency of ANN algorithms. So we need horizontal scaling to divide vectors into shards and replicas to scale across multiple machines, search each shard and combine the results from all shards to return relevant results. This improves scalability and provides cost-effective performance [Pinj].

4. **Replication**: Vector databases will need to handle a lot of requests. Shards allow us to use many pods in parallel to perform vector search faster. Replicas allow us to handle more requests in parallel by replicating the whole sets of pods [Pinj]. Replicas improve availability, even in case of failure by spreading replicas to different availability zones.

5. **Hybrid Storage**: Vector search is typically run in-memory(RAM). For organizations with over a billion items, the memory costs will make vector search too expensive to even consider. Some vector search libraries provide an option to store on disk but at the expense of latency in search results. In Hybrid Storage, a compressed version of the vector index is stored in memory and the original vector is stored on disk[Pinj]. The in-memory index is then used to find a smaller set of vectors to search within the original index on disk. this enables fast and accurate search and reduces infrastructure costs by up to 10x[Pinj].

6. **API**: Vector databases should take the burden of building and maintaining vector search applications away from developers by providing an API to enable easier usage and management of vector databases from other applications [Pinj]. It should be easy to make API calls to the database for performing operations like insertion, deletion, updating and retrieving query results. REST APIs add flexibility by allowing the functionality of the database to be accessed from any environment that can make HTTPS calls. So that developers can access it directly through client languages like python, java and Go [Pinj].

2.4.2 Comparison of Vector Library and Vector Database

In this section, we provide an overview of the key differences between a vector library and a vector database summarized in Table 2.8 where Pinecone has been used as an example for a vector database.

Vector Libraries The vector Libraries use in-memory indexes to store vector embeddings to perform similarity search. They have the following characteristics:

- *Store Vectors Only*: They store vectors only and not the related objects that were used to generate them. When you make a search query, the vector library will return the relevant vectors and object ids [Wea]. It is a limitation since the main information is stored in the object and not the id. To overcome this issue, the objects need to be stored in secondary storage, so that they can be used to match the objects with the returned ids from the query to get the actual results [Wea].

- *Immutable Data*: The vector libraries produce immutable indexes, which means once the index has been built using the data, you cannot modify it (no new inserts, deletes or changes). The index has to be rebuilt from scratch to reflect any changes [Wea].

- *Query during Import Limitation*: In most vector libraries, it is not possible to query the data while it is being imported [Wea]. All the data has to be imported first and then the index is built, it is a limitation for applications that require importing huge amounts of data [Wea].

- Some of the examples are Facebook Faiss [JDJ21], Spotify Annoy[Spo21], Google ScaNN [GSL+20], NMSLIB[BN13] and HNSWLIB[Mal18].These libraries perform vector similarity search using the ANN algorithm. The ANN algorithm has different implementations that depends on the vector library. For instance, Faiss uses the clustering method, Annoy uses trees and ScaNN uses vector compression, there is a performance tradeoff in each implementation. The vector libraries are most commonly used for applications where the data does not change [Wea].

Vector Databases The key feature that differentiates vector databases from vector libraries is the ability to store and update data. Vector Databases provide the full CRUD (create, read, update, and delete) functionality to solve the limitations of vector libraries and are more focused on deployments in production at the enterprise level [Wea]. They have the following characteristics:

- *Store Objects and Vectors*: The vector databases store both data objects and vectors. As they store both, databases combine vector search with structured filters that allow you to make sure the nearest neighbors match the filter from metadata [Wea].
- *CRUD Functionality*: They allow to modify entries in the index after it has been created, which is necessary when working with data that changes continuously [Wea].
- *Real-time Search*: Vector databases allow us to query and modify your data during the import process. As you are importing millions of objects, the imported data remains fully accessible for querying but the queries will not return objects that have not been imported yet [Wea].
- Some examples are Pinecone, Weaviate, Qdrant, and Milvus. Vector databases are great for applications where the data is changing [Wea].

The comparison of vector databases and vector libraries led to the main conclusion that vector libraries are better suited for applications where data doesn't change, and fully-fledged vector databases are more suited for applications where data changes continuously.

Attributes	Vector Library	Vector Database (Pinecone)
Filtering (combined with Vector Search)	No	Yes
CRUD Support	No (some do, e.g. hnswlib)	Yes
Incremental importing, concurrent reading while importing	No (some do, e.g. hnswlib)	Yes
Stores objects and vectors	No	Yes
Speed	Typically faster than full-blown database	Typically slower than pure library
Performance optimized for	In-memory similarity search	End-to-end callstack, includes vector search, object retrieval, filtering, network requests, etc.
Durability, crash recovery	No	Yes
Persistence	Only at explicit snapshot	Immediate (after each insert, update, or delete)
Sharding	No (there are some 3rd-party projects that wrap sharding around existing libraries)	Yes
Replication	No	Yes
Deployment ecosystem	No (you have to build it yourself)	Yes
SDKs / language clients	No (although most have python bindings)	Yes (Python, Node.js, Ruby(community-maintained))
Execution	Embedded (can be turned into standalone service if you build a simple wrapper app around it)	Standalone service
Communication with app	Foreign Function Interface (FFI) / Language Bindings	Network calls (RESTAPI, gRPC)
Multi-tenancy	No	Yes
Hybrid BM25+dense vector search	No	Yes (from v2.2)

Table 2.8: Comparison between Vector Library and Vector Database [Wea]

2.4.3 Comparison of Vector Databases

We compared Qdrant, Milvus, Weaviate, and Pinecone vector databases to compare their features and functionalities for supporting vector/semantic search. We present them and their unique value proposition below. Some of the features are summarized in Table 2.9.

1. **Qdrant**: is a vector similarity engine that offers extended support for filtering. It implements dynamic query planning and payload data indexing [Kan]. It was entirely developed in Rust, and implements dynamic query planning and payload data indexing. It supports a large variety of datatypes and query conditions as well

as filtering conditions [Kan].

2. **Milvus**: is a cloud-native vector database that manages unstructured data. It pays attention to the scalability of the entire search engine i.e. how to index and reindex vector data efficiently, and scale the search part. A unique value is an ability to index the data with multiple ANN algorithms [Kan].

3. **Weaviate**: supports an expressive query syntax through a GraphQL-like interface, to enable explorative data science querying on rich entity data [Kan]. Its core features include vector search, object storage, and an inverted index for boolean keyword searches, which prevent data drift and reduce latency between different databases storing vector data separately from objects/inverted index [Kan].

4. **Pinecone**: is a fully managed vector database that supports vector search for unstructured data [Kan]. It offers single-stage filtering to search objects and filter by metadata in one query [Kan]. Its exact KNN is powered by FAISS, ANN is powered by a proprietary algorithm[Kan]. It provides a distributed infrastructure for benefits like reliability and speed. Pinecone also handles security through AWS AND GCP environments, isolated containers and encryptions [Kan].

In conclusion, there are a lot of common functionalities offered by vector databases, the major differences lie in the availability of different algorithms for creating vector search indexes and the distance metrics available to compare the vectors for semantic search, although there are three common metrics that are available in all vector databases namely - Euclidean, Cosine and Dot Product or Inner product. Qdrant uses native HNSW algorithm that has limitations like durability requirements, CRUD support, pre-filtering etc. Milvus does not offer the functionality to update a vector. Weaviate and Pinecone are quite similar but differ in the case of filtering with vector search. Pinecone uses single-stage filtering which is more efficient, hence we decided to use Pinecone in our experiments for comparison with relational databases. This discussion addresses our first research question as mentioned in Section 3.1.

After the comparison of vector databases, Pinecone emerges as a strong choice for building a semantic search application due to several key features that are discussed below:

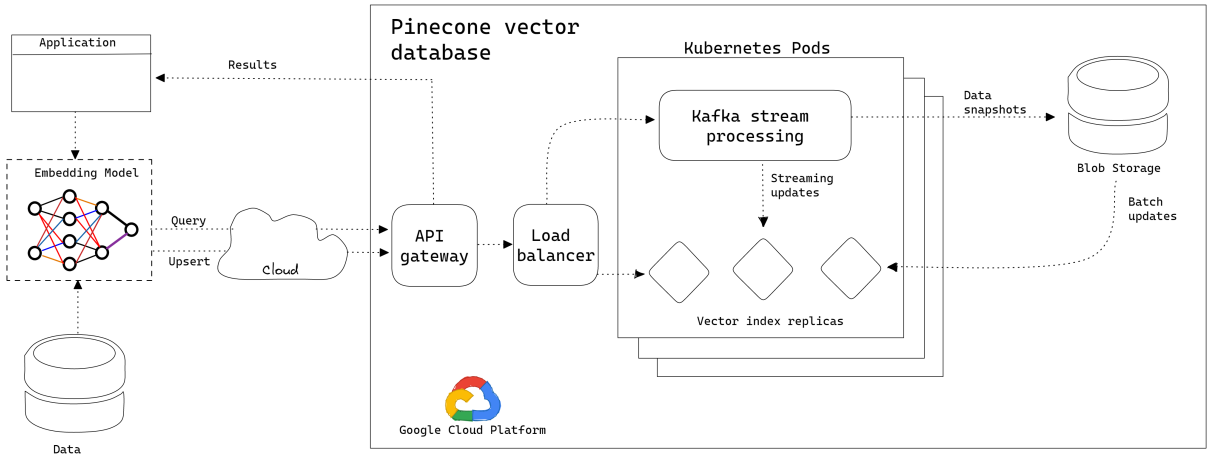

Figure 2.21: Pinecone Architecture [Kan]

Feature	Qdrant	Milvus	Weaviate	Pinecone
Latest Version	1.1.0	2.2.4	1.18.2	2.2.0
Programming Language	rust	Go	Go	Rust
Consistency Model	Eventual Consistency	Strong Consistency	Eventual Consistency	Eventual Consistency
Support for GraphQL	N/A	N/A	Yes	N/A
Sharding	Yes	Yes	Yes	Yes
Replication	Yes	Yes	Yes	Yes
Pagination	Yes	Yes	Yes	N/A
Metric Types	Euclidean (L2), Cosine Similarity, Dot Product	L2, Inner Product, Jaccard, Tanimoto, Hamming, Superstructure, Substructure	cosine, dot, l2-squared, hamming, manhattan	euclidean, cosine, dot product
Max vector dim	N/A	32,768	N/A	20,000
Index types	HNSW	FLAT, IVF_FLAT, IVF_SQ8, IVF_PQ, HNSW, ANNOY	Custom HNSW	Proprietary Graph based
Deployment Type	Managed/self-hosted	Self-hosted	Managed/self-hosted	Managed
Code	open source	open source	open source	close source
Scalability	Horizontal and Vertical	Horizontal	Yes	Horizontal and Vertical
filtering with ANN	N/A	N/A	pre-filtering	single-stage filtering
Hybrid BM25+dense vector search	Yes	Yes	Yes	Yes
SDKs/ language clients	python, Rust, Go	Python, Go, Java, Node.js	Python, Go, Java, JavaScript	Python, Node.js, Ruby(community-maintained)

Table 2.9: Comparison of Vector databases [far]

1. **Single-stage filtering**: Pinecone provides single-stage filtering, i.e. it allows for efficient vector search by filtering objects and metadata in one query. It helps to reduce latency and makes search more efficient contributing to an improved user experience.

2. **Scalability**: Pinecone supports both horizontal and vertical scalability, ensuring that the database can handle growing data volumes and increasing query loads. This makes Pinecone an ideal choice for applications that need to handle large

amounts of data.

3. **Managed Service**: Pinecone is a fully managed vector database, which means that the database infrastructure, maintenance, and updates are handled by the service provider. This allows us to focus on building applications without worrying about database management, making it easier to deploy and scale the application.

4. **Consistency Model**: Pinecone offers an eventual consistency model, which balances performance and consistency, making it suitable for applications that prioritize low latency and high availability.

5. **Supported Metrics**: Pinecone supports the most popularly used distance metrics such as Euclidean, Cosine, and Dot Product to enable accurate similarity search.

6. **Graph-based Index**: Pinecone's proprietary graph-based index offers a unique approach to nearest neighbor search, delivering high-speed search with very good quality even as the number of vectors increases to millions or billions.

7. **Security**: Pinecone is SOC2 Type II certified. The certification is based on the COSO framework and has been audited by an external Big4 CPA firm (EY). The scope of the program includes Information Security, Availability, and Confidentiality [pina].

2.4.4 Comparison of Relational and Vector Databases

In this section, we explore the terminologies related to vector databases using Pinecone as an example and later compare their advantages and limitations with relational databases. Firstly, we will describe the way different operations are executed in Pinecone and PostgreSQL pgvector because they enable semantic search capabilities.

1. Insert query of Pinecone and Pgvector

   - Pinecone

     ```
     # connect to the index
     index = pinecone.Index("pinecone-index")

     # insert data as  list of (id, vector tuples)
     index.upsert([("A", [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]),
                   ("B", [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2])])
     ```

   - pgvector

     ```
     embedding = np.array([1, 2, 3])
     cur.execute('INSERT INTO item (embedding) VALUES (%s)', (embedding,))
     ```

2. Select query of Pinecone and Pgvector

   - Pinecone

     ```
     index.query(vector = new_vector, top_k = 5, include_values = True)
     ```

   - pgvector

```
cur.execute('SELECT * FROM item ORDER BY embedding <-> %s LIMIT 5',
(embedding,))
```

3. Update query of Pinecone and Pgvector
    - Pinecone

      ```
      # updates the value of a item
      index.upsert([("id-3", [3.3, 3.3])])
      ```

    - pgvector

      ```
      query = text(
      "UPDATE update_benchmark SET embeddings = :updated_vector WHERE ←
      id = 1"
      conn.execute(query, updated_vector=new_vector)
      ```

4. Delete query of Pinecone and Pgvector
    - Pinecone

      ```
      index.delete(ids=["id-1", "id-2"], namespace='example-namespace')
      ```

    - pgvector

      ```
      query = text("DELETE FROM delete_benchmark WHERE id = 1")
      conn.execute(query)
      ```

5. Create Index for Pinecone and Pgvector
    - Pinecone

      ```
      pinecone.create_index("example-index", dimension=128, metric="cosine",
      pods=4, pod_type="s1.x1")
      ```

    - pgvector

      ```
      CREATE INDEX ON items USING ivfflat (embedding vector_cosine_ops);
      ```

In Table 2.10, we compare the different features and functionalities provided by these three vector databases: PostgreSQL, Pinecone, and PostgreSQL with pgvector extension. PostgreSQL and PostgreSQL with pgvector extension are relational databases that offer traditional CRUD operations, PostgreSQL pgvector extension supports vector operations, providing an inverted file index algorithm for nearest neighbor search, while PostgreSQL does not support vector operations, thus cannot be used for semantic search. Pinecone, on the other hand, is a proprietary vector database that uses a graph-based index algorithm for nearest neighbor search, with support for Euclidean, cosine, and dot product distance metrics. Pinecone has a lower selectivity of queries and is CPU-bound, whereas PostgreSQL and pgvector are highly selective and IO-bound. Pinecone and PostgreSQL both offer low latency and consistency, with Pinecone being eventually consistent and PostgreSQL providing strong consistency. In terms of scalability, all three databases offer both horizontal and vertical scaling, with sharding and replication capabilities. Pinecone is a managed service, while PostgreSQL and pgvector can be self-hosted or managed. Overall, Pinecone is purpose-built for vector data, while PostgreSQL is a general-purpose database that has an extension pgvector that supports vector data.

Feature	PostgreSQL	Pinecone	PostgreSQL with pgvector
Data types	SQL standard types + JSON/blob	Vectors	Standard + vectors
Geometric filters	No	Yes	Yes
Updates, deletes, traditional (i.e. metadata) filters	Yes	Yes	Yes
Freshness, low latency	Yes	Yes	Yes
Query language	SQL	model(Query, X) > threshold	SQL
Selectivity of queries	Highly selective, IO-bound	Low selectivity, CPU bound	-
Vector dimensions limit	-	20,000	16,000
Distance metrics	-	Euclidean, cosine, dot product	Euclidean, cosine, dot product
ANN Based algorithm	-	Proprietary Graph-based	Inverted File Index
Programming Language	C	Rust	C
Consistency Model	Strong Consistency	Eventual Consistency	Strong Consistency
Sharding	Yes	Yes	Yes
Replication	Yes	Yes	Yes
Deployment Type	Managed/self-hosted	Managed	N/A
Code	open source	close source	open source
Scalability	Horizontal and Vertical	Horizontal and Vertical	N/A

Table 2.10: Comparison of Relational and Vector Databases

2.5 Summary

In this chapter, we started with Fundamental concepts and Background in which we discussed the search fundamentals, key concepts, applications of search, challenges of traditional lexical search, characteristics of the research problem, related work, NLP for semantic search are discussed in detail. We then moved to discuss semantic search, nearest neighbor indexes, evaluation metrics, zero-shot evaluation, and similarity metrics. Finally, we discussed vector databases.

3 Design

In Chapter 2, the objective was to provide an overview of the important background related to the concepts for our thesis work. This chapter presents the specific information of our work, accompanied by the research questions and our approach. The chapter contains the following sections:

- The first section outlines the research questions we seek to address with our work (Section 3.1).
- In the next section (Section 3.2), we provide the methodology used to address the second and fifth research questions as stated in Section 3.1.
- The last section, (Section 3.3) provides the summary of the discussions and work presented in this chapter.

3.1 Research Questions

Based on the research work, conducted during this thesis, we identified the following research questions that are important for our work. The first, third and fourth questions have been already addressed in Chapter 2 that provided valuable insights and understanding of the capabilities of vector databases that enable semantic search, similarity measures used to compute the similarity between vectors and the pre-trained models that are available to perform semantic search and understand their performance. In this chapter, we outline the methodology used to address the second and fifth research questions that will provide insights into the performance of the selected databases and models. This assessment will allow us to identify the most suitable solutions for multilingual semantic search applications, highlighting our contribution.

1. What are the vector databases that are currently available and how do they differ in their capabilities?
2. How does a vector database perform as compared to a PostgreSQL database (with and without pgvector-python extension)?
3. What are some of the similarity functions that help to measure the similarity of vectors?
4. What are the multilingual models that are currently available that can perform a semantic search?
5. How do these multilingual models differ in terms of Zero-shot Evaluation of their retrieval capabilities and comparison of their inference speed?

3.2 Methodology

In this section, we describe the methodology used to address the second and fifth research questions described in Section 3.1. There are two main components: assessing the performance of storage backends for semantic search and examining the performance of multilingual models. The following sections describe the methodologies for each of these components.

3.2.1 Benchmark PostgreSQL and Pinecone for Semantic Search

In order to evaluate the performance of the different storage backends (PostgreSQL, Pinecone, and PostgreSQL with pgvector-python extension), we conducted experiments focusing on the core tasks of Create, Read, Update, Delete, (i.e. CRUD functionality) and Batch Insertion. Each experiment was designed to evaluate the performance of the storage backends in terms of their ability to handle different aspects of the semantic search application to address the second research question. We evaluated the following storage backends as mentioned in Table 3.1 after examining their features and functionalities related to semantic search in comparison to the alternatives as discussed in Chapter 2.

Name	Client version	Database version
PostgreSQL	1.4.46 (sqlalchemy)	15.1
Pinecone	2.1.0 (python)	2.0
PostgreSQL (pgvector-python extension)	0.1.6	15.1

Table 3.1: List of Storage backends

Core Tasks

1. **Create:** By inserting records into each database, we aim to evaluate their performance with a varying number of rows and embedding dimensions, to assess their efficiency for handling data insertion and different dimensions to simulate the use of diverse models for practical applications. We measure the time taken to insert data in seconds Table 3.2.

Row Size	**Embedding Size**
100	384
200	512
300	768
400	1024

Table 3.2: Insertion Performance using the mentioned Row and Embedding Sizes

2. **Read:** Given a database with a table that has *400* rows with the embedding size of *768*, retrieve existing records in the range of the following K values and measure the time taken for retrieval in seconds to evaluate the performance of the database for read operations:

K
3
5
10
100
250

Table 3.3: Different Values of K for Retrieval

3. **Update:** Given a database with a table that has *400* rows with the embedding size of *768*, update the existing record by id, capture the time taken in seconds for doing so 20 times and calculate the average time taken in seconds to assess performance for update tasks.

4. **Delete:** Delete a record by id given we have a table that has *400* rows with the embedding size of *768*, and capture the time taken in seconds to assess performance for the deletion task.

5. **Batch Insertion:** Measure the insertion time taken in minutes to insert records into the database with a batch size of *100* rows and embedding size of *768* for the number of rows mentioned in Table 3.4. This task evaluates the performance of each database when handling large-scale data insertion, as data size in real-world applications can vary significantly.

Total Number of Rows
10000
30000
50000
70000
100000

Table 3.4: Batch Insertion with different row sizes

3.2.2 Benchmark Multilingual Model Performance

In order to assess the performance of different multilingual models that were selected after examining their features and performance as discussed in Chapter 2, we conducted two sets of experiments using the multilingual models from the library *sentence-transformers [RG19], version 2.2.2* mentioned in Table 3.5. These experiments helped in assessing the performance of these models in terms of their retrieval capabilities and inference speed, addressing the fifth research question.

1. **Zero-shot evaluation**: We performed zero-shot evaluation[TRR+21] of the models to evaluate retrieval performance using two metrics Recall and Mean Reciprocal Rank at different values of K mentioned in Table 3.6 for two datasets mentioned in table Table 3.7.

Model Name	max_seq_length	embedding_dimension
paraphrase-multilingual-MiniLM-L12-v2	128	384
distiluse-base-multilingual-cased-v1	128	512
paraphrase-multilingual-mpnet-base-v2	128	768
quora-distilbert-multilingual	128	768

Table 3.5: List of Multilingual Models

Recall@K & MRR@K
1
3
5
10
100

Table 3.6: Values of K for Evaluation measures - Recall and Mean Reciprocal Rank

Dataset	Language	Website	BEIR-Name	Type	Queries	Corpus
Quora	English	Homepage	quora	dev, test	10,000	523K
Germandpr-beir processed	German	Homepage	N/A	train, test	10,300	26K

Table 3.7: List of Datasets

2. **Inference Speed Comparison**: We compared the average inference speed of the models using a sample of *100000* rows from the Quora dataset. The experiment involved encoding each query (10000) and returning the results at $K = 10$ using cosine similarity as a measure of similarity. The time taken was measured in milliseconds, providing a basis for comparison of their inference speed.

3.3 Summary

This chapter outlines the research questions and describes the methodology used to address the second and fifth research questions to assess the performance of different databases and models, that will provide valuable insights into the most suitable solutions for the semantic search application.

4 Experimental Setup

In this chapter, we provide the necessary details required to replicate the results of our performance benchmark experiments and also the prototype for the multilingual semantic search application using the Quora Dataset. The chapter covers the following sections:

- Section 4.1 provides the overview of the data used to Benchmark the Performance of different storage backends and different multilingual models.
- Section 4.2 provides details about the multilingual semantic search application prototype.
- In Section 4.3 we provide the specifications of the hardware and software used during this work.
- Finally, the summary of the chapter.

4.1 Datasets

4.1.1 Data for Benchmarking Performance of Different Storage Backends

The data was created with the following structure as mentioned in Table 4.1 using two libraries *faker [Fak], version 16.6.1 and pandas [pdt20], version 1.5.3*. The *'sentence'* column contains text, pandas encodes text columns as object datatype and also *'embeddings'* column is a list of floating point numbers which is also encoded as object datatype in pandas. The *'object'* datatype is a flexible datatype which can hold various types of Python objects. Generating fake data provides us with the ability to generate data for different embedding sizes and different numbers of rows for our performance benchmark.

Name	Datatype
id	integer
sentence	object
embeddings	object

Table 4.1: List of Data types

The different storage backends used the following datatypes for the performance benchmark as shown in Table 4.2. The PostgreSQL uses datatype - *ARRAY(Float)* to store embeddings, whereas PostgreSQL pgvector extension uses *Vector(size)* where size = the size of embedding dimension, e.g. Vector(384) stores embeddings where a single embedding has a size of 384 dimensions. Pinecone only accepts *id* with string datatype.

Name	id	sentence	embeddings
PostgreSQL	Integer()	Text()	ARRAY(Float)
Pinecone	String	String	vector
PostgresSQL (pgvector-python extension)	Integer()	Text()	Vector(size)

Table 4.2: List of Datatypes used by Storage backends

4.1.2 Data for Evaluating Performance of Multilingual Models

We used two open-source datasets to perform a Zero-shot evaluation of the multilingual models. Every BEIR dataset must contain a corpus, queries, and qrels(relevance judgements file). The datasets follow the default dataset structure, as required by the BEIR benchmark [TRR+21]. The folder structure of any BEIR dataset is as follows, e.g.:

- scifact/
  - corpus.jsonl
  - queries.jsonl
  - qrels/
    - train.tsv
    - dev.tsv
    - test.tsv

We used the test split of the *qrels* judgements to evaluate our models. The format of the corpus, queries and qrels (relevance judgements file) must be as follows; we also describe the datatypes of each column and provide a description of the columns and a high-level example.

1. **corpus file:** a .jsonl (jsonlines) file that includes a list of dictionaries, where each dictionary contains three fields:
   - _id a unique document identifier of datatype - string.
   - *title* denotes the title of the document (optional) of the datatype - string.
   - *text* document paragraph or passage of datatype - string.

   For example:

```
{"_id": "doc7", "title": "Jeremy Howard",
"text": "Jeremy Howard is an expert in the field of deep learning..."}
```

2. **queries file:** also a .jsonl (jsonlines) file with a list of dictionaries, where each dictionary contains two fields:
    - _id unique query identifier of datatype - string
    - *text* denotes the text of the query, also of datatype - string

    For example:

    ```
    {"_id": "q7", "text": "Who is Jeremy Howard?"}
    ```

3. **qrels file:** a .tsv (tab-separated) file with a header in the first row, it includes a list of dictionaries that have three columns in the following order:
    - query-id represents the id of the query, of datatype - string
    - corpus-id represents the document id, of datatype - string
    - score denotes the relevance judgement between a query and document, of datatype - int32

    For example:

    ```
    {"q7": {"doc7": 1}}
    ```

BEIR Quora Dataset

The quora dataset [bei] from BEIR is in English and has the following specifications:

- Total Size of Corpus: **522931**
- A single entry in the *corpus dictionary* has the id of the document containing a text field, which is the document in our corpus, and a title field empty for all questions. Here is an example.

```
{'44241': {'text': 'Can an F-1 Visa student start Online startup-↩
    company?',
'title': ''}}
```

- Total Queries: **10000**
- A single entry in *queries dictionary* has the unique id of the query and the query/question asked by the user, it looks like this:

```
{'44241': 'Can you start an online business/store while on the F-1 visa↩
     in the US?'}
```

- Total Query Document Relevance Judgements: **10000**
- A entry in *qrels dictionary* has query id as the key of the dictionary which is also mapped to a dictionary that contains the document id and the judgement score of 1 or 0, which represents if the document in the corpus is relevant to our query(1) or not(0). In the example below, we can see that for the *query id = 44242*, the document id *44241* is relevant since it has a score of *1*.

```
{'44242': {'44241': 1}}
```

Germandpr-beir Dataset

The original dataset germandpr [MRP21] was created by the team at deepset.ai. The Geramndpr-beir [PA] was created by converting the original germandpr dataset into the BEIR format as required by the framework. The Language of the datasets is German. The Germandpr-beir has two variants, described in Table 4.3.

Variant	Split	Text Preprocessing	Keep Title
Original	test/train	False	True
Processed	test/Train	True	False

Table 4.3: Description of Germandpr-beir dataset

We used the *'test'* split of the *processesd variant* of the dataset for the experiments. The processed variant does not have the *title* column and removes the special formatting in the texts. which has the following specifications:

- Total Size of Corpus: **2875**
- Each entry in the *corpus dictionary* has the id of the document, a text field which is the document in our corpus, and also a title column that has been removed during preprocessing. Here is an example below.

```
'c279': {'text': "Tuberkulose = Symptome = Grunds tzlich wird der
Erkrankungsverlauf bei der Tuberkulose in verschiedene Stadien
eingeteilt. Krankheitszeichen, die sich direkt nach der Infektion
manifestieren, werden als ''Prim rtuberkulose'' bezeichnet. Da die
Bakterien aber auch bei intakter Immunabwehr ohne
Krankheitszeichen oder nach durchgemachter Prim rtuberkulose
lebenslang im K rper schlummern und jederzeit wieder reaktiviert
werden k nnen, spricht man bei einer nicht zur Erkrankung
f hrenden Erstinfektion von einer latenten Tuberkuloseinfektion
(LTBI) bzw. nach einer Ersterkrankung von einer ''postprim ren
Tuberkulose'' oder auch ''Sekund rtuberkulose''. Da sich die
Infektion zwar zumeist an der Lunge, aber eben prinzipiellauch an
jedem anderen Organ abspielen kann, wird au erdem die
''Lungentuberkulose'' von der ''Organtuberkulose'' unterschieden.",
'title': ''}
```

- Total Queries: **1025**
- Every entry in *queries dictionary* has the unique id of the query and the query/question asked by the user, that looks like:

```
{'q966': 'Durch was wird Tubrekulose ausgel st?'}
```

- Total Query Document Relevance Judgements: **1025**

- For every entry in *qrels dictionary* there is query id, which is mapped to a dictionary that contains the document id and the judgement score of 1 or 0, which represents if the document in our corpus is relevant to the query(1) or not(0). In the example below, we can see that for the *query id = q966*, the document id *c279* is relevant since it has a score of *1*.

```
{'q966': {'c279': 1}}
```

4.2 Prototype: Multilingual Semantic Search Application

To address the task of finding semantically similar questions, We need a search application that satisfies the following requirements:

- *Language support*: The system should be able to handle multiple languages and provide accurate and relevant search results for each language.
- *Semantic understanding*: The application should have a strong semantic understanding of the languages supported, enabling it to recognize synonyms, translations, and related terms across languages.
- *Efficient indexing*: Implement an efficient indexing mechanism such as Approximate Nearest Neighbor Search (ANNS) indexes to ensure fast search speeds and low latency without sacrificing search quality.
- *Scalability*: The system should be able to scale with the growth of data, supporting an increasing number of languages, documents, and user queries.
- *High recall*: The search algorithm should be optimized for high recall, ensuring that users do not miss relevant information while searching.
- *Low latency*: The system should be designed to provide quick response times, ensuring a seamless user experience.

By addressing these requirements, a multilingual search application ensures a seamless, efficient and optimized search experience for users across various languages. To meet the requirements we developed a prototype for the multilingual semantic search application.

The prototype of the multilingual semantic search application uses the *Pincone* vector database to store the vector embeddings and the *paraphrase-mutltilingual-mpnet-base-v2* model to encode the Quora questions data to obtain the embeddings. The architecture of the application is depicted in Figure 4.1. The Quora dataset can be found on the Hugging Face Hub [Dat]. The data preparation details can be found in the following section.

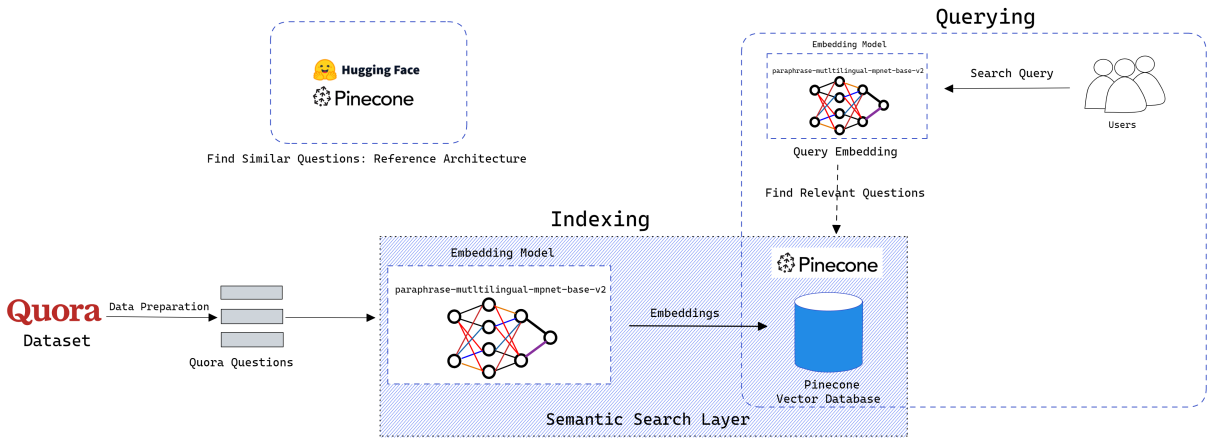

Figure 4.1: Multilingual Semantic Search Application Architecture

4.2.1 Data Preparation and Pre-processing

The Quora dataset is composed of Question Pairs with the following data fields:

- questions: a dictionary feature that contains the following data and datatypes:
  - id: int32
  - text: string
- is_duplicate: bool

The dataset has **404290** rows. An example row looks like:

```
{'questions': {'id': [1, 2], 'text': ['What is the step by step guide to invest
in share market in india?', 'What is the step by step guide to invest in
share market?']}, 'is_duplicate': False}
```

The following transformation steps were performed to prepare the data before indexing data for semantic search:

1. Filter on the is_duplicate column, to keep only unique question pairs. The number of rows after filtering - **255027**.
2. Flatten the data and remove 'is_duplicate' column.
3. Create separate id columns for 'id1' and 'id2' from 'id' in the questions dictionary.
4. Create separate question columns for 'question1' and 'question2' from 'text' in the questions dictionary.
5. Now, we have four columns in the dataset 'id1', 'id2', 'question1', 'question2', with **255027** rows.
6. Concatenate the two id columns into a single 'id' column.
7. Concatenate the two questions into a single 'question' column.
8. Drop duplicate rows using 'id'.
9. The dataset contains **413109** rows. Identifed **67** outliers.

10. The final cleaned dataset contains **413042** rows with **2** columns namely *'id' and 'question'*.

11. The first **30000** questions were indexed and stored in the vector database and made available for search in the prototype.

4.3 Experimental Environment

This section provides information about the software and hardware resources used during our work for the purpose of reproducibility.

4.3.1 Benchmark Performance for Storage Backends

Machine Configuration

- **Operating System** Microsoft Windows 10 Home
- **Processor** Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s)
- **Memory** 8.0 GB RAM
- **Graphics** NVIDIA® GeForce® 920MX (2 GB DDR3 dedicated)

Programming Framework

- **Programming Languages** Python (Version 3.9.12)
- **Programming Tools** Visual Studio Code (Version 1.76.1), Jupyter Notebook (Version 6.5.2)
- **Libraries** Faker (Version 16.6.1), matplotlib (Version 3.6.3), numpy (Version 1.23.5), pandas (Version 1.5.3), pinecone (Version 2.1.0), pgvector (Version 0.4.0), sqlalchemy (Version 1.4.46)
- **Containerization Platform Tools** Docker (Version 20.10.22), Docker Compose (Version 2.15.1)
- **Miscellaneous** Pinecone provides a simple API to connect to its vector database. Therefore, an API is required to use Pinecone, which can be accessed by signing up at [Pin22]. The experiments use the *'us-east1-gcp'* environment on Google Cloud and using a single pod of pod type - *p1*.

Limitations of the Experimental Setup

- PostgreSQL instance was accessed using Docker which may reflect the same performance as an instance running on a local machine.
- Pinecone runs in the cloud which adds overhead like network, authentication and parsing among others. Therefore the experiments and results obtained during this work, are an approximate measure of Pinecone's performance.

- Since pgvector is not currently available on google cloud [Goo23], we ruled out creating a PostgreSQL instance on the Google Cloud Platform.

4.3.2 Benchmark Performance for Multilingual Models

The experiments for the model performance were performed on Google Colab.

Machine Configuration

- **Operating System** Ubuntu 20.04.5 LTS (Focal Fossa)
- **Processor** Intel(R) Xeon(R) CPU @ 2.30GHz, 2300 Mhz, 2 Core(s), 4 Logical Processor(s)
- **Memory** 11.0 GB RAM
- **Graphics** NVIDIA A100-SXM (40 GB)
  - Driver Version: 525.85.12
  - CUDA Version: 12.0

Programming Framework

- **Programming Languages** Python (Version 3.9.16)
- **Programming Tools** google-colab (Version 1.0.0)
- **Libraries** beir (Version 1.0.1), gradio (3.21.0), matplotlib (Version 3.5.3), numpy (Version 1.22.4), pandas (Version 1.4.4), pinecone (Version 2.1.0), sentence_transformers (Version 2.2.2)

4.4 Summary

This chapter provided an overview of the datasets used for the experiments to benchmark performance, the architecture of the prototype application and the necessary data preparation details for the quora dataset used for the application. We concluded this chapter by providing the resource specifications.

5 Evaluation and Results

We addressed the first, third and fourth research questions in Chapter 2. In this chapter, we discuss the results of experiments that were conducted to answer the second and fifth research questions as mentioned in Section 3.1. To address the second research question, we present the results of the comparison between a PostgreSQL database, an extension that enables semantic search in the PostgreSQL database called pgvector-python and Pinecone, a vector database. The fifth research question is answered by comparing multilingual models based on their zero-shot retrieval capability and inference speed. The chapter contains the following sections:

1. In Section 5.1, we provide a discussion about the comparison of the different databases.
2. Section 5.2 discusses the results of our model comparison.
3. In Section 5.3, we provide the summary of the results derived from the previous two sections.

Moving forward, we refer to the PostgreSQL database sometimes as **postgres** and the PostgreSQL pgvector extension solely as **pgvector** in the following discussions. Also, the terms **embedding size**, **dimensions** and **dimension size**, all of which refer to the size of the vector/embedding are used interchangeably.

5.1 Comparison of Different Storage Backends

This section will present the comparison of the different storage backends, that were explored in this thesis work, using data mentioned in Subsection 4.1.1 on the core tasks mentioned in Section 3.2.

1. Create
2. Read
3. Update
4. Delete
5. Insertion in Batches

5.1.1 Performance Evaluation for *Create* Task

For this task, we measure the time taken in seconds for the completion of the data insertion process using different numbers of rows and various dimension sizes of the embeddings to view the impact of a growing number of rows and also the impact of different dimension sizes of the embeddings on time taken to insert data in the storage backends. The PostgreSQL database instance was run using Docker, while the Pinecone Vector database is a managed, cloud-native database that provides a REST API to interact with the underlying instance. Pinecone has a limit of the max size of 2MB for an upsert (inserting data) request. Thus, a smaller number of rows were chosen to allow us to use higher dimensional embedding sizes like 1024 without sending data in batches for the first task.

Comparison of insertion time for different embedding size

In this section, we discuss the impact of inserting a certain number of rows across different dimensions. In Figure 5.1 when inserting 100 rows for different embedding sizes, we can see that the pgvector database that uses specially designed *VECTOR(size)* Datatype performs better than Postgres using the datatype *ARRAY(FLOAT)* and Pinecone. Pgvector takes the least amount of time for inserting 100 rows across all embedding sizes. While Pinecone takes the most time with 0.82 seconds for an embedding size of 1024. PostgreSQL takes the most time for the embedding size of 768 with 0.83 seconds. Pinecone performs better than postgreSQL for 768 embedding size by taking 0.69 seconds. Overall Pgvector keeps it under 0.5 seconds for each embedding size for 100 rows, whereas Pinecone keeps insertion under 1 second as shown in Table 5.1. Postgres and Pgvector individually took more time for insertion for an embedding size of 768 than when inserting for an embedding size of 1024 dimensions. Pinecone used more time for insertion as the embedding size grew when inserting 100 rows.

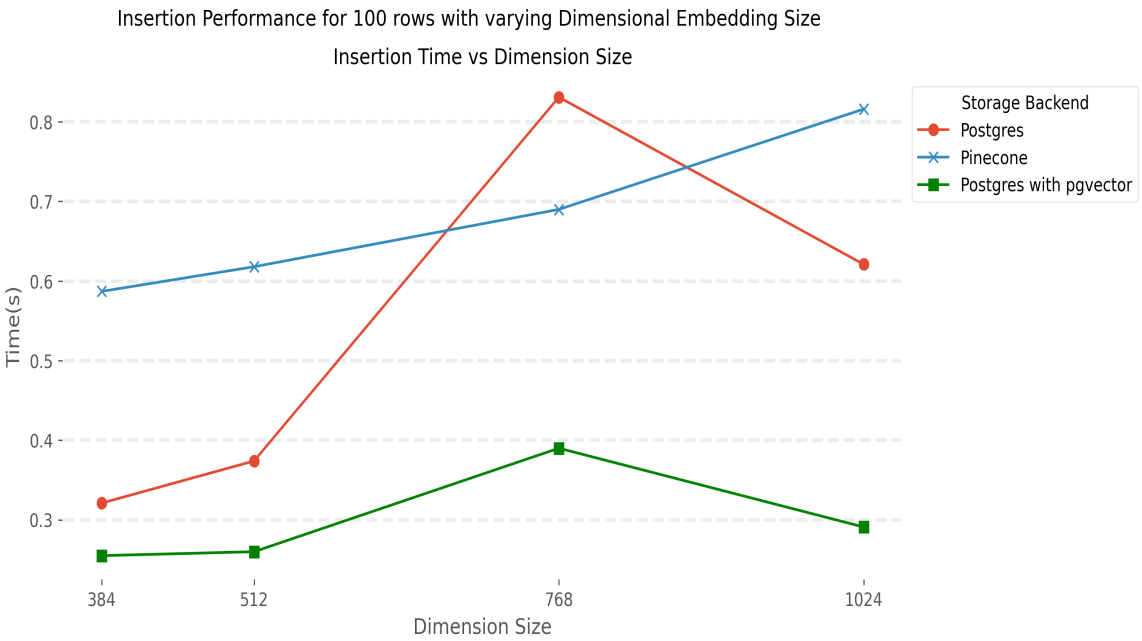

Figure 5.1: Comparison of Insertion time for *rows*=100

	Embedding Size	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Rows=100	384	0.321	0.587	0.255
	512	0.374	0.618	0.260
	768	0.831	0.690	0.390
	1024	0.621	0.816	0.291

Table 5.1: Comparison of insertion time for 100 rows

In Figure 5.2 below, inserting 200 rows sees a decline in performance for the Postgres database, as it performs worse than both Pgvector and Pinecone. Postgres took more time for an embedding size of 768 again, and used less time for an embedding size of 1024, suggesting it performs worse for 768 dimensions among the embedding sizes. Pinecone performs the best for 384 dimensions and beats both Postgres and Pgvector, for 512 dimensions Pinecone performs on the same level as Pgvector and better than Postgres. Pgvector performs better than Pinecone for 768 dimensions, although there is only a difference of 0.254 seconds as depicted in Table 5.2. It performs much better than Pinceone for 1024 dimensions. While it performs considerably better than Postgres for 768 and 1024 dimensions. It can be observed that Pinecone utilizes more time as the embedding size grew again. Overall Pinecone performs better for 384 and 512 dimensions and Pgvector across 768 and 1024 dimensions.

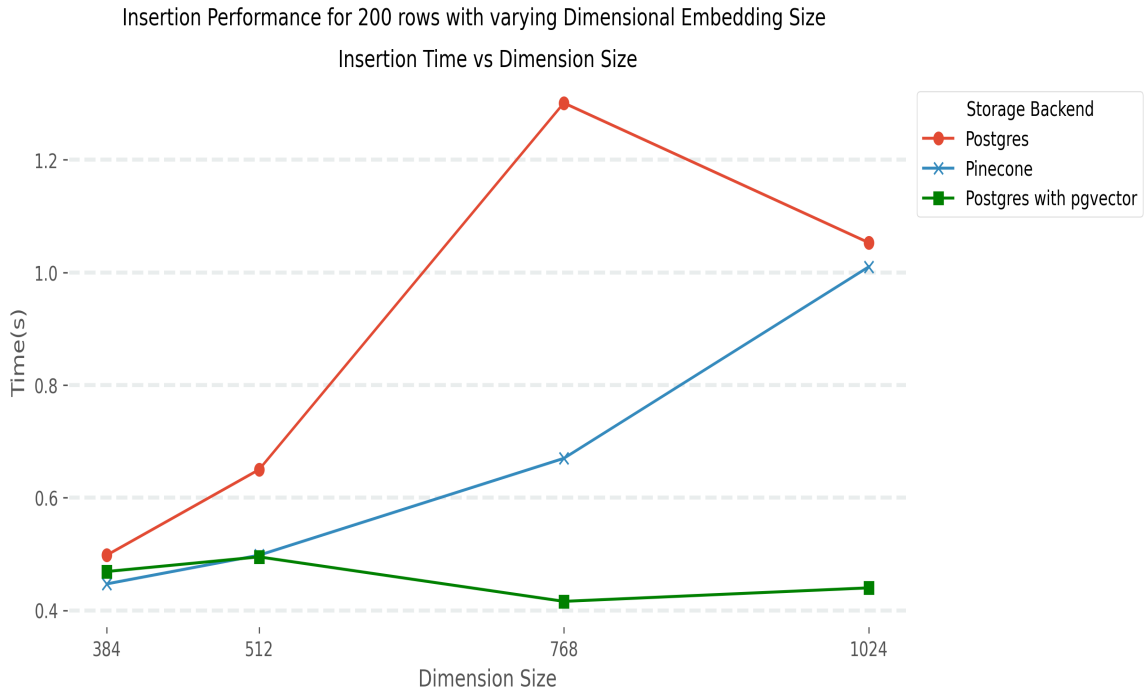

Figure 5.2: Comparison of Insertion time for *rows*=200

	Embedding Size	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Rows=200	384	0.498	0.447	0.469
	512	0.650	0.498	0.495
	768	1.301	0.670	0.416
	1024	1.053	1.010	0.440

Table 5.2: Comparison of insertion time for 200 rows

In the Figure 5.3, while inserting 300 rows, Pgvector performs better than both Postgres and Pinecone across all embedding sizes. While Pinecone performs better than Postgres across 3 embedding sizes but takes more time than Postgres for 512 dimensions. Pinecone is observed to be more stable across different dimensions. This time Postgres takes more time for 1024 dimensions than 768, breaking its previous observed pattern. Postgres also performs better than Pinecone for 512 dimensions but as the dimension size increases it struggles to keep up with Pinecone. Overall Pgvector outperforms both Postgres and Pinecone across all dimensions, showing great performance. Postgres recorded the highest time of 1.855 seconds for 1024 dimensions. Pinecone recorded its highest time as 1.206 seconds while inserting data for 1024 dimensions and Pgvector had the highest value of 0.691 seconds while inserting data for 768 dimensions as shown in Table 5.3.

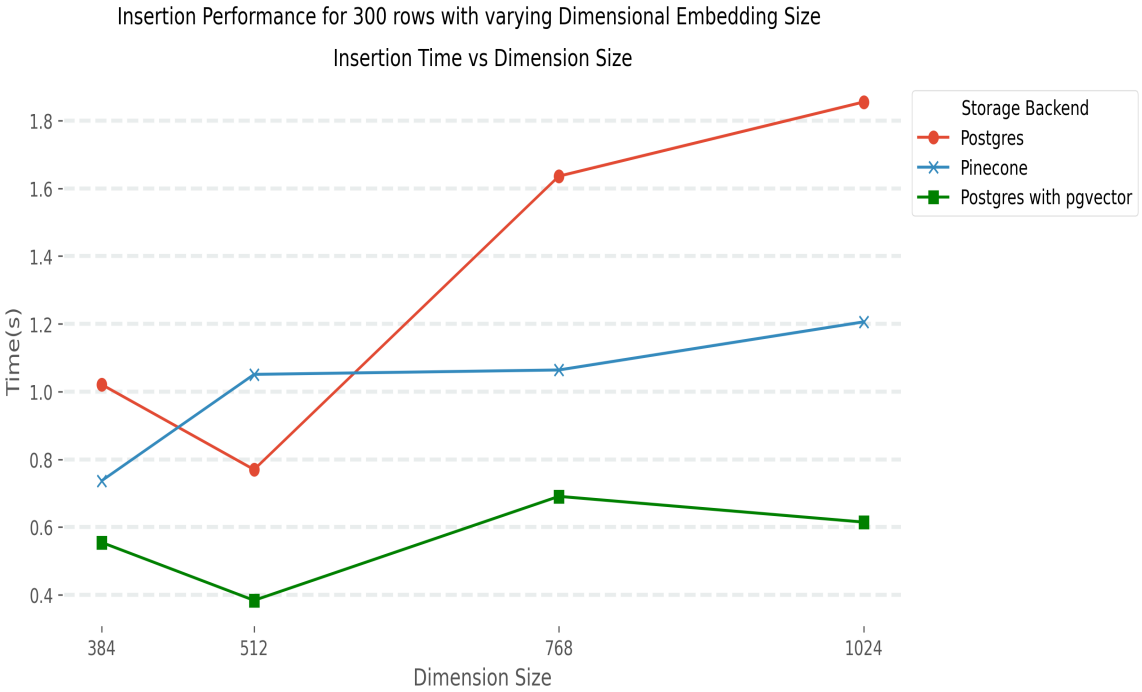

Figure 5.3: Comparison of Insertion time for *rows*=300

	Embedding Size	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Rows=300	384	1.021	0.737	0.555
	512	0.770	1.051	0.384
	768	1.636	1.064	0.691
	1024	1.855	1.206	0.615

Table 5.3: Comparison of insertion time for 300 rows

In Figure 5.4, we evaluate the performance of the different storage backends while inserting 400 rows for different embedding sizes. By analyzing the results, it was observed that Pinecone performs better than Postgres across all different sizes of embeddings while taking more time as the embedding size grows. Pinecone kept the time in the range of 0.9 - 1.7 seconds over different dimensions, which can be seen in Table 5.4. A similar pattern can be observed for Postgres, utilizing more time as the number of dimensions grows but performing badly in comparison to both Pinecone and Pgvector. Postgres recorded the highest time of 2.158 seconds for all iterations when inserting data for 1024 dimensions. Overall Pgvector performs better than both other competitors. It utilized half the time as compared to PostgreSQL without pgvector. Pinecone does a little better than Postgres when competing with the pgvector extension keeping in mind that additional burden of network overhead.

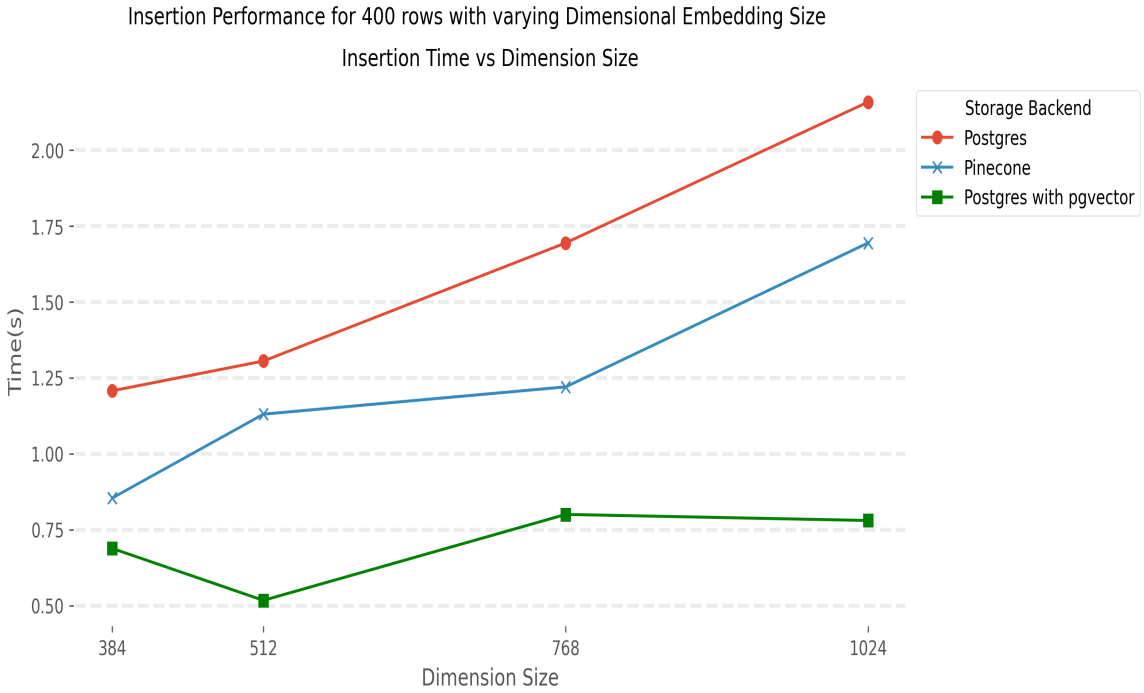

Figure 5.4: Comparison of Insertion time for *rows*=400

	Embedding Size	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Rows=400	384	1.207	0.854	0.688
	512	1.305	1.130	0.517
	768	1.694	1.220	0.800
	1024	2.158	1.694	0.780

Table 5.4: Comparison of insertion time for 400 rows

Comparison of insertion time across a different number of rows

In this section, we observe the impact of inserting a different number of rows for a particular dimension size.

In the Figure 5.5, we can see that for a dimension size of 384; Postgres takes more time as the number of rows increases from 10 to 400, and we can see a notable increase in time for 300 and 400 rows. Pinecone is observed to take more time for 10 rows, which can be attributed to the network overhead, especially significant for the first REST API call to the service. Later, Pinecone shows better performance when inserting 100 and 200 rows with a slight increase in time taken for 300 and 400 rows. We can also see that Pgvector also takes more time for insertion as the number of rows increase. In conclusion, all the storage backends took more time to insert data as the number of rows increased. The time taken by different storage backends measured in seconds is mentioned in Table 5.5.

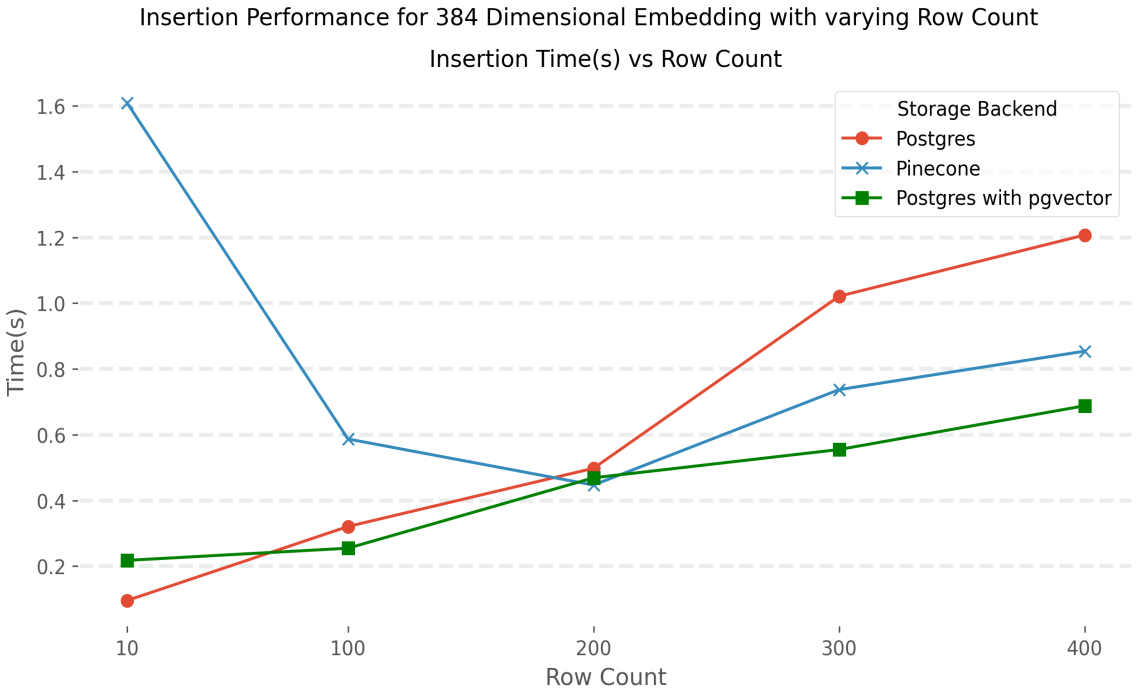

Figure 5.5: Comparison of Insertion time for *Embedding Size*=384

	Rows	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Embedding Size=384	10	0.096	1.609	0.218
	100	0.321	0.587	0.255
	200	0.498	0.447	0.469
	300	1.021	0.737	0.555
	400	1.207	0.854	0.688

Table 5.5: Comparison of insertion time for the embedding size of 384

In Figure 5.6 below, we can observe that for 512 dimensions, Postgres took more to insert data as the number of rows increased. Pinecone is observed to take more time for 10 rows and then it gradually shows better performance for 100 and 200 rows, and then takes more time for the following row sizes. But Pinecone shows better performance than Postgres for 400 rows with a difference of 0.2 seconds. Pgvector shows better performance in comparison to both Postgres and Pinecone, taking significantly less time for all row sizes apart from 200 rows, whereas Pinecone matches the Performance of Pgvector. In summary, Postgres follows the same pattern of taking more time as data size increases, whereas Pinecone follows an irregular pattern showing good performance in some cases. While Pgvector maintains its better performance. The insertion times measured in seconds are recorded in Table 5.6.

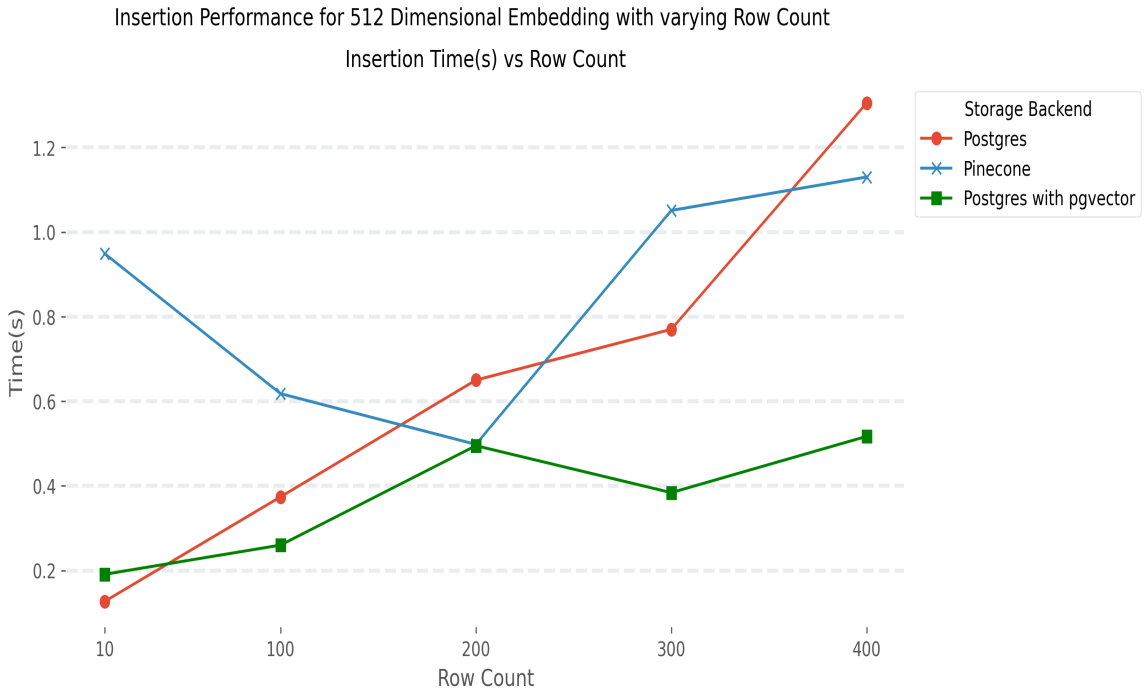

Figure 5.6: Comparison of Insertion time for *Embedding Size*=512

	Rows	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Embedding Size=512	10	0.127	0.949	0.191
	100	0.374	0.618	0.260
	200	0.650	0.498	0.495
	300	0.770	1.051	0.384
	400	1.305	1.130	0.517

Table 5.6: Comparison of insertion time for the embedding size of 512

In Figure 5.7, we can observe the impact of growing row count when using a 768-dimensional vector on insertion time. Postgres degrades in Performance and continues the pattern of taking more time for insertion as the row count increases. Pinecone performs better than Postgres for all the row counts except 10, which can be explained due to the network overhead for the first API call to the service. Pinecone shows steady performance in two cases before we see the impact of increasing row count leading to more time for insertion. Pgvector performs better again but this time we can also observe the impact of an increasing number of rows leading to more time for inserting data. In summary, we can see the time taken by each storage backend is affected by an increasing number of rows, as the row size increases as depicted in Table 5.7.

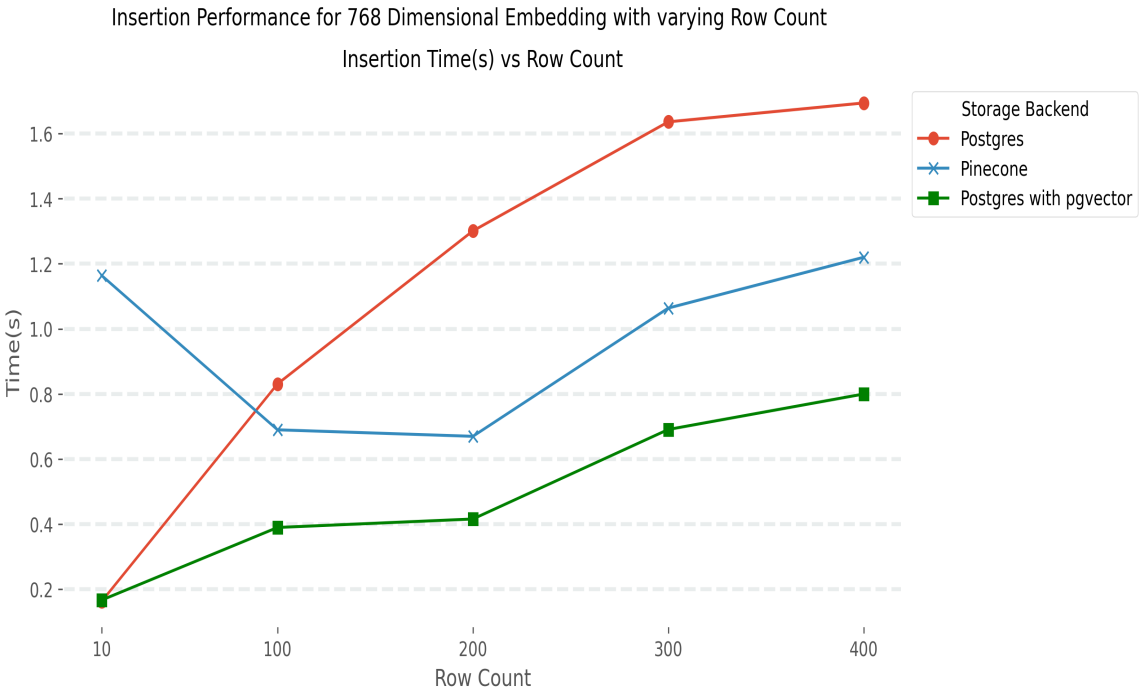

Figure 5.7: Comparison of Insertion time for *Embedding Size*=768

	Rows	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Embedding Size=768	10	0.163	1.164	0.167
	100	0.831	0.690	0.390
	200	1.301	0.670	0.416
	300	1.636	1.064	0.691
	400	1.694	1.220	0.800

Table 5.7: Comparison of insertion time for the embedding size of 768

In Figure 5.8, we observe the performance of the different storage backends for 1024 dimensions. Postgres continues the trend, establishing that it needs more time for data insertion as the number of rows increases. But it performs better than others for the row count of 10. Pinecone, also observes a similar pattern, taking more time for the first insertion of 10 rows and then doing better for the subsequent second and third iterations. It also takes more time for insertion as the data size increases. Pgvector demonstrates a similar behaviour using more time as the data size grows for 1024 dimension vectors. The time recordings are mentioned in Table 5.8. Pinecone does better than Postgres across all different row counts except 10 rows which is considerable taking into account the additional network overhead. Pgvector keeps the recorded time for all row sizes under 1 second.

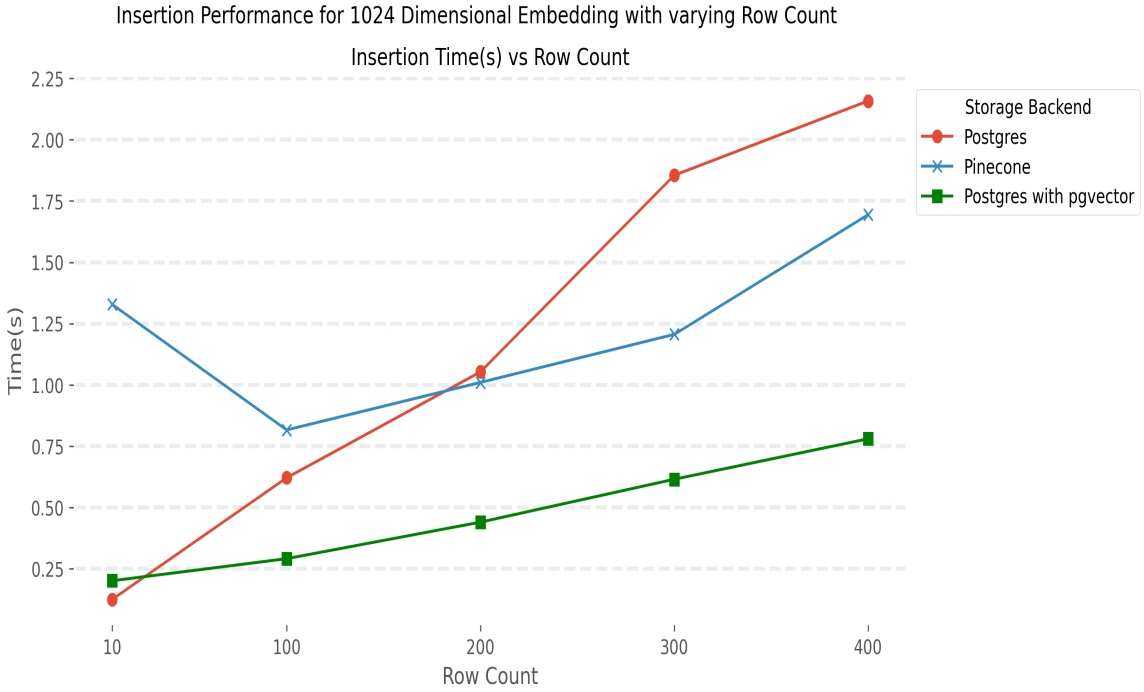

Figure 5.8: Comparison of Insertion time for *Embedding Size*=1024

	Rows	Insertion Time (Seconds)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Embedding Size=1024	10	0.125	1.328	0.201
	100	0.621	0.816	0.291
	200	1.053	1.010	0.440
	300	1.855	1.206	0.615
	400	2.158	1.694	0.780

Table 5.8: Comparison of insertion time for the embedding size of 1024

Table 5.9 summarizes the results of the experiments, showing a comparison of the different storage backends for inserting data(lower is better). The lowest cell value is highlighted for the particular storage backend where it shows superior performance than the others. Pgvector offers superior performance for the insertion task than others, while Pinecone's performance issues can be explained due to network overhead, however, it still keeps the time taken in the range of 0.4 to 1.7 seconds. Postgres only offers better performance for the lowest number of rows, it also crossed over 2 seconds to insert data for the 1024 dimension vectors.

Number of Rows	Embedding Size	PostgreSQL		Pinecone		PostgreSQL with pgvector	
		Time (s)	Time (ms)	Time (s)	Time (ms)	Time (s)	Time (ms)
10	384	0.096	96	1.609	1609	0.218	218
100	384	0.321	321	0.587	587	0.255	255
200	384	0.498	498	0.447	447	0.469	469
300	384	1.021	1021	0.737	737	0.555	555
400	384	1.207	1207	0.854	854	0.688	688
10	512	0.127	127	0.949	949	0.191	191
100	512	0.374	374	0.618	618	0.260	260
200	512	0.650	650	0.498	498	0.495	495
300	512	0.770	770	1.051	1051	0.384	384
400	512	1.305	1305	1.130	1130	0.517	517
10	768	0.163	163	1.164	1164	0.167	167
100	768	0.831	831	0.690	690	0.390	390
200	768	1.301	1301	0.670	670	0.416	416
300	768	1.636	1636	1.064	1064	0.691	691
400	768	1.694	1694	1.220	1220	0.800	800
10	1024	0.125	125	1.328	1328	0.201	201
100	1024	0.621	621	0.816	816	0.291	291
200	1024	1.053	1053	1.010	1010	0.440	440
300	1024	1.855	1855	1.206	1206	0.615	615
400	1024	2.158	2158	1.694	1694	0.780	780

Table 5.9: Insertion Task Summary Across Different Storage Backends

5.1.2 Performance Evaluation for *Read* Task

Extraction is one of the key tasks of a search engine, we need the search engine to be quick in returning results to offer the user a good search experience. Generally, Top-K results are returned to the user which are similar to the user's query. In this task, since PostgreSQL does not support vector operations, a normal *SELECT* query was executed, while for pgvector and pinecone, cosine similarity was used to compute similarity and return results.

Table 5.10 summarizes the performance of different storage backends for extracting different numbers of rows for given values of K. The time taken is measured in seconds (lower is better), where each cell that contains the lowest value is highlighted when a particular storage backend performs better in comparison to others.

Top K	PostgreSQL		Pinecone		PostgreSQL with pgvector	
	Time (s)	Time (ms)	Time (s)	Time (ms)	Time (s)	Time (ms)
3	0.008	8	0.274	274	0.008	8
5	0.007	7	0.278	278	0.094	94
10	0.011	11	0.347	347	0.075	75
100	0.063	63	0.907	907	0.013	13
250	0.135	135	1.560	1560	0.110	110

Table 5.10: Time taken for Retrieval for different values of K

In Figure 5.9, we can observe the performance of different storage backends for the extraction task. We can see that Postgres and Pgvector outperform Pinecone. While Pgvector performs better than Postgres for higher values of K. While Pinecone follows the pattern of taking more time as K increases. In conclusion, for higher values of K, Pgvector performs better. Pinecone keeps time taken in the range of 0.2 to 1.6 seconds without considering the network overhead issues. Postgres outperformed others for three lower values of K.

5.1.3 Performance Evaluation for *Update* Task

The task evaluates the performance of different backends on the time taken to update a record by id. We measured the time taken to execute the task in seconds, repeated it 20 times and then reported the average time in Table 5.11. PostgreSQL performs better than others, although Pgvector only used 0.004 seconds more than PostgreSQL and Pinecone also exhibited good performance using under 0.15 seconds to update a record.

Database Name	Updation Time(s)	Updation Time (ms)
PostgreSQL	0.026	26
Pinecone	0.142	142
PostgreSQL with pgvector	0.030	30

Table 5.11: Duration of update task

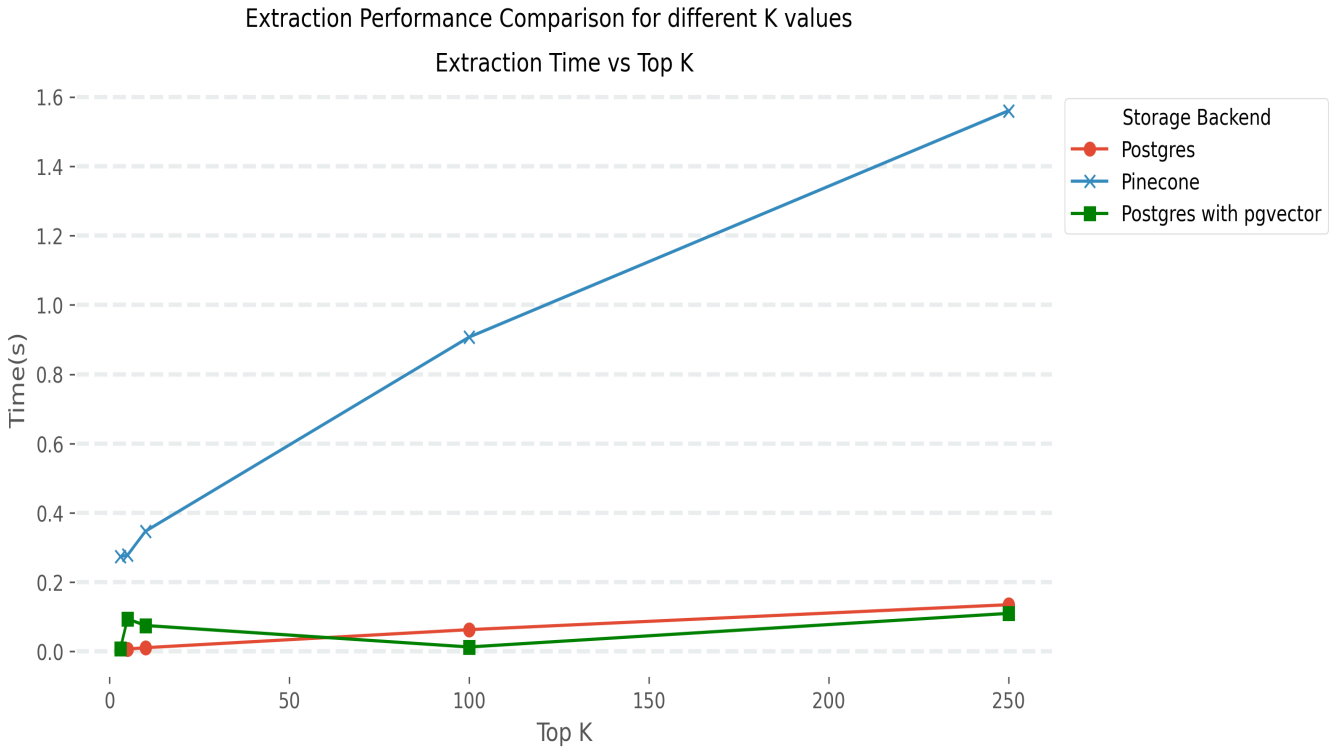

Figure 5.9: Comparison of Time Taken to Return Results

5.1.4 Performance Evaluation for *Delete* Task

This task evaluates the performance of different storage backends to delete a single record by id. The results of the experiment as shown in Table 5.12, tell us that PostgreSQL and Pgvector offer similar performance again. While Pinecone also offers good performance using under 0.14 seconds to delete a record by id, keeping in mind the network overhead issues involved.

Database Name	Deletion Time (s)	Deletion Time (ms)
PostgreSQL	0.020	20
Pinecone	0.136	136
PostgreSQL with pgvector	0.023	23

Table 5.12: Time taken for Deletion task

5.1.5 Performance Evaluation for *Batch Insertion* Task

This task evaluates the performance of the storage backends for data insertion in batches over a larger number of rows to provide better insights into their ability to handle large-scale data insertion tasks efficiently. The embedding dimension size was 768 for all of them.

In Figure 5.10, we can see that, as the number of rows increases, the time taken to insert data in batches also increases for all storage backends. Postgres is observed to be the slowest of all three, taking almost 16 mins as mentioned in Table 5.13 to insert 100000

rows of data. Pinecone offers better Performance than Postgres just using half the time utilized by Postgres for 50000 and 70000 rows of data. Pinecone takes almost 13 minutes to insert 100000 rows of data in batches of 100. Pgvector offers superior performance, by taking just under 8 mins to complete the batch insertion task for 100000 rows of data. Pgvector takes half the time than Postgres for all different row counts. Pinecone's performance is comparable to Pgvector for 70000 rows of data, keeping in mind the additional network overhead issues. Overall, Pgvector is the standout performer for the batch insertion task.

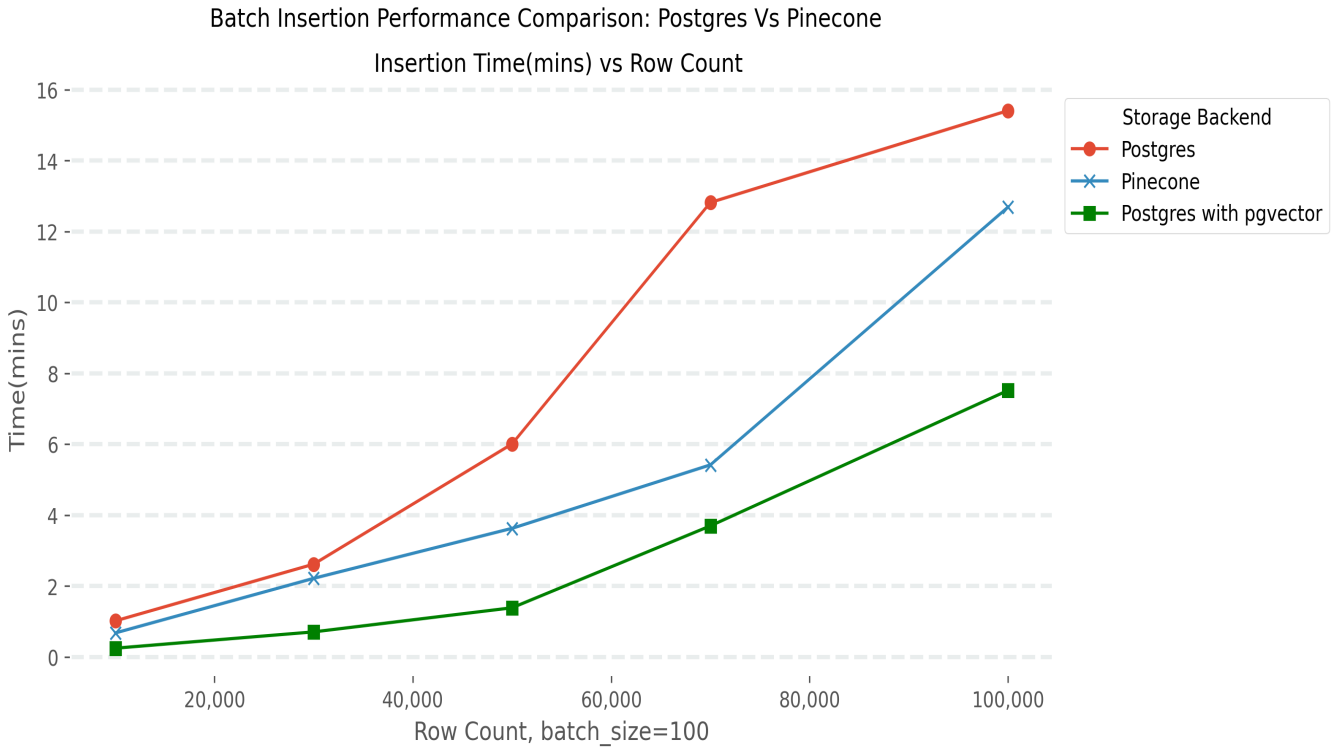

Figure 5.10: Comparison of Storage Backends for Batch Insertion task

	Rows	Insertion Time (Minutes)		
		PostgreSQL	Pinecone	PostgreSQL with pgvector
Batch Size=100	10000	1.02	0.68	0.25
	30000	2.62	2.22	0.71
	50000	6.01	3.63	1.39
	70000	12.82	5.42	3.70
	100000	15.41	12.69	7.52

Table 5.13: Insertion time for different numbers of rows using batch size = 100

Summary

In this section, we compared and evaluated the performance of different storage backends for our semantic search application, namely PostgreSQL - a relational database, Pinecone - a vector database, and an extension of PostgreSQL i.e. PostgreSQL with pgvector for different tasks. The analysis was done to compare the performance of loading data in bulk, extracting data, updating data, deleting data and also inserting data in batches in terms of their execution time. In the first task, where data is inserted in bulk, observed from two different viewpoints - the impact of data that has a different number of dimensions for an embedding/vector and the impact of an increasing number of records/rows i.e. for different data sizes. the performance of the PostgreSQL pgvector extension was better than PostgreSQL and Pinecone, while Pinecone performed better than PostgreSQL and was not far away from the pgvector extension when we consider the additional load of network overhead. Pinecone kept the execution times under 1 second for the majority of the iterations. For the second task, the PostgreSQL and PostgreSQL pgvector extension both outperformed each other in 3 different iterations each. While Pinecone kept the performance under 1 second for all iterations, which is impressive considering Pinecone is a distributed service and the execution times depend on the network since querying an index across the internet adds overhead like network, authentication, parsing etc. In the third and fourth tasks, all storage backends perform really well, although PostgreSQL came out on top with very tiny margins. In the fifth task, PostgreSQL pgvector offers superior performance, although Pinecone also offers very good performance considering the issue of network overhead in comparison to PostgreSQL. It was observed that an increasing number of dimensions and data size has an impact on execution times which affects performance.

Selection of Storage backend

Pinecone and PostgreSQL pgvector extension both are reasonable choices when dealing with vector data. The results from the experiments show that the PostgreSQL pgvector extension offers great performance, while Pinecone comes in second and PostgreSQL third. Since we want to create a semantic search application, where we need to identify similar questions using similarity search, we have to rule it out because it does allow vector operations. PostgreSQL pgvector extension uses IVF for nearest neighbour search compromising search speed for better quality while Pinecone uses a graph-based index that offers high speed with very good quality as discussed in Chapter 2 as the number of vectors increases millions to billions. The upper limit for the number of dimensions for vectors in Pinecone is 20000 whereas pgvector has a limit of 16000. In pgvector, each vector takes 4* dimensions + 8 bytes of storage while in Pinecone each dimension on a single vector consumes 4 bytes of memory and storage per dimension. PostgreSQL pgvector extension is currently at version 0.4.0 whereas Pinecone is at 2.0 which suggests that Pinecone is more actively maintained and new features are added, also Pinecone was designed for vector data and since databases work best when they are designed for the type of data you are storing and representing.

Therefore, we selected Pinecone to tackle the problem, of building a semantic search application - identifying similar questions, where the knowledge repository can be huge to provide users with low latency and relevant results.

5.2 Comparison of Multilingual Models

The multilingual models are evaluated using two datasets provided by the BEIR benchmark for two languages English and German using the following metrics, as discussed in Chapter 2:

- Recall@k: measures the percentage of relevant items that are included in the top-k matches. We want the application to have a higher recall rate so that our users are served with relevant information.
- MRR@k: determines the rank of the first relevant item in relation to the top-k matches. We want the application to have a higher mean reciprocal rank so that the relevant results are close to the top among the list of results that are returned to the user.

For the following discussions, we will refer to each model using the names mentioned below:

- paraphrase-multilingual-MiniLM-L12-v2 as *MiniLM-L12-v2*
- distiluse-base-multilingual-cased-v1 as *distiluse-cased-v1*
- paraphrase-multilingual-mpnet-base-v2 as *mpnet-base-v2*
- quora-distilbert-multilingual as *quora-distilbert*

5.2.1 Evaluation of Performance on BEIR Quora Dataset

The Quora dataset provided by BEIR is in English, here we provide the overview of the performance of the models.

1. Recall@K - The recall rate at different values of K can be observed in Table 5.14. The discussion of model performance can be found below:
    - All models achieve at least 80% Recall@3, which means they are able to retrieve 80% out of all relevant items in the top-3 results. The highest value was achieved by MiniLM-L12-v2 at 85.2% while the lowest was by quora-distilbert at 81.7%. distiluse-cased-v1 achieves 82.2% and mpnet-base-v2 achieves 85.1% with only a small difference of 0.1% with the top performing model.
    - At Recall@5, the performance of models improve a bit more, with the minimum value at 86.7% achieved by quora-distilbert, 87.3% by distiluse-case-v1 and the other two models again the top performing ones with MiniLM-L12-v2 achieving 90.4% recall rate and mpnet-base-v2 right behind with 90.2%.
    - At Recall@10, the model performance increases further, as each model crosses the 90% threshold and offers great performance, with the highest value of 93.3% by mpnet-base-v2 and MiniLM-L12-v2 and the lowest of 91.1% by quora-distilbert while distiluse-cased-v1 gets 92.1%.

- At recall@100, the models offer great performance going close to 100% recall rate which can be explained by the fact that as we allow more items to be retrieved (i.e. increase the value of k), it becomes easier because the model has more opportunities to retrieve relevant items, even if some irrelevant items are retrieved. The higest performing model was mpnet-base-v2 at 99.4%. While MiniLM-L12-v2 got 99.3%, distiluse-cased-v1 got 98.6% and quora-distilbert got 98.4%.

2. MRR@k - The different MRR values achieved by the models can be found in Table 5.14.

    - MRR@3: The models start with an 80% baseline again, and offer reasonable performance in putting the relevant results at the top of the list when results are returned to the user. At MRR@3 the top performing model is MiniLM-L12-v2 with 84.5% and mpnet-base-v2 just behind with 84.3%. The other models are also close to each other with distiluse-cased-v1 at 81.4% and quora-distilbert at 80.8%

    - MRR@5: The models achieve a short boost in performance of about 1%. The top-performer is again MiniLM-L12-v2 with 85.3% with mpnet-base-v2 again close with 85.1% The distiluse-cased-v1 model achieves 82.3% and the quora-distilbert 81.7%.

    - At MRR@10 and MRR@100, the models again only show a short increase in performance. With only a difference in the range of 0.1-0-0.3% for values at 10 and 100. The models arranged top to bottom in terms of performance are: MiniLM-L12-v2, mpnet-base-v2, distiluse-cased-v1 and quora-distilbert.

3. Summary: Model Performance on this dataset is dominated by MiniLM-L12-v2, with mpnet-base-v2 slightly behind with minimal differences for both metrics. The other models perform well in terms of Recall@k but not so well for MRR@k. The performance of the models improved for larger values of K.

5.2.2 Evaluation of Performance on Germandpr-beir Dataset

This dataset is in German, here we provide a discussion on the performance of the models.

1. Recall@K - The recall rate at different values of K can be observed in Table 5.14.

    - All models achieve at least 70% Recall@3, except quora-distilbert which struggles at 59.5%, not doing so well as it means they are able to retrieve just 70% out of all relevant items in the top-3 results. The highest value was achieved by mpnet-base-v2 at 74.6% while the lowest was by quora-distilbert. distiluse-cased-v1 achieves 74% and MiniLM-L12-v2 achieves 71% with a difference of 3.5% with the top performing model.

    - At Recall@5, the performance of models improves significantly, with the minimum value at 70.3% achieved by quora-distilbert, 82% by both distiluse-case-v1 and the mpnet-base-v2 and with MiniLM-L12-v2 achieving 79.7% recall rate.

- At Recall@10, the model performance increases further significantly again, as each model achives the 80% threshold and offers good performance, with the value of 88.2% by mpnet-base-v2 and 86.3% by MiniLM-L12-v2 and the lowest of 79.6% (roughly 80%) by quora-distilbert while distiluse-cased-v1 gets the highest 88.4%.
- At recall@100, the models offer great performance going close to 97% recall rate for MiniLM-L12-v2, mpnet-base-v2, and distiluse-cased-v1, The higest performing model was mpnet-base-v2 at 97.6%. While MiniLM-L12-v2 got 96.4%, distiluse-cased-v1 got 97% and quora-distilbert got 93.9%.

2. MRR@k - The different MRR values achieved by the models can be found in Table 5.14.
   - MRR@3: The models start with a very low baseline of 60% apart from quora-distlbert that offers poor performance with 46.2%, At K = 3, the models do not do well in putting the relevant results at the top of the list when results are returned to the user. At MRR@3 the top performing model is mpnet-base-v2 with 60.9% and distiluse-cased-v1 at 60.5% just behind, while MiniLM-L12-v2 achieves 57.9%.
   - MRR@5: The models do not show significant improvement just achieving improvements in the range of 1-2.5%. The top-performer is mpnet-base-v2 again close with 62.5%, MiniLM-L12-v2 with 60% and the distiluse-cased-v1 model achieves 62.2% and the quora-distilbert 48.7%.
   - At MRR@10 and MRR@100, the models again only show a short increase in performance. With only a difference in the range of 0.1-0-0.6% for values at 10 and 100. The models arranged top to bottom in terms of their performance on MRR are: mpnet-base-v2, distiluse-cased-v1, MiniLM-L12-v2, and quora-distilbert.
3. Summary: Model Performance on this dataset is dominated by mpnet-base-v2, with distiluse-cased-v1, slightly behind with minimal differences for both metrics. All models perform well in terms of Recall@k, but not so well for MRR@k, with quora-distilbert at 51% struggling to provide adequate performance, other models also don't do well for MRR.

Model Name	Evaluation Metric	BEIR-Quora		Germandpr-beir	
		Corpus Size	Queries	Corpus Size	Queries
		522931	10000	2875	1025
paraphrase-multilingual-MiniLM-L12-v2	Recall@3	0.852		0.710	
	Recall@5	0.904		0.797	
	Recall@10	0.943		0.863	
	Recall@100	0.993		0.964	
	Mrr@3	0.845		0.579	
	Mrr@5	0.853		0.600	
	Mrr@10	0.857		0.609	
	Mrr@100	0.858		0.613	
distiluse-base-multilingual-cased-v1	Recall@3	0.822		0.740	
	Recall@5	0.873		0.820	
	Recall@10	0.921		0.884	
	Recall@100	0.986		0.970	
	Mrr@3	0.814		0.605	
	Mrr@5	0.823		0.622	
	Mrr@10	0.827		0.631	
	Mrr@100	0.829		0.635	
paraphrase-multilingual-mpnet-base-v2	Recall@3	0.851		0.746	
	Recall@5	0.902		0.820	
	Recall@10	0.943		0.882	
	Recall@100	0.994		0.976	
	Mrr@3	0.843		0.609	
	Mrr@5	0.851		0.625	
	Mrr@10	0.854		0.634	
	Mrr@100	0.856		0.639	
quora-distilbert-multilingual	Recall@3	0.817		0.595	
	Recall@5	0.867		0.703	
	Recall@10	0.911		0.796	
	Recall@100	0.984		0.939	
	Mrr@3	0.808		0.462	
	Mrr@5	0.817		0.487	
	Mrr@10	0.821		0.499	
	Mrr@100	0.824		0.506	

Table 5.14: Retrieval performance of Multilingual models

5.2.3 Comparison of Inference speed

In this section, we discuss the results of the comparison of the inference speed of the models, to measure how much time it takes a model to encode a query into an embedding and return top-k similar results. We used k=10 and cosine similarity in our experiment. The inference speed of the models is summarized in Table 5.15.

The inference speed of the embedding model plays a crucial role in the overall performance and user experience of the semantic search applications. As depicted in Table 5.15, all the models have impressive inference speeds, which is essential for real-time applications that demand fast and efficient results. Semantic search applications mostly deal with a large amount of data and must provide relevant search results to the user within a reasonable time. The inference speed of the embedding models directly affects the response time of the application, a faster inference speed yields a quicker response to the queries of the user which is desirable for real-time search applications where users expect quick results or when the application has to process numerous simultaneous queries. Additionally, a model with fast inference speed allows the application to handle more queries in a short amount of time, improving the overall scalability of the system which is vital for applications that need to process huge amounts of data and serve a large user base.

All the models perform extremely well in terms of inference speed, using less than 0.1 seconds to encode the query, get the embedding and return results. distiluse-cased-v1 and quora-distilbert take 9 milliseconds since they are optimized for speed as discussed in Chapter 2. In comparison, MiniLM-L12-v2, and mpnet-base-v2 use 17 milliseconds, here mpnet-base-v2 performs equivalent to MiniLM-L12-v2, even though the number of output dimensions for an embedding are more for mpnet-base-v2. We can say the same about distiluse-cased-v1 and quora-distilbert, as quora-distilbert outputs 768 dimensions in comparison to 512 by distiluse-cased-v1. Here, the index sizes are related to the size of the dimensions of the model. We can observe that as the number of dimensions grows for embeddings, they take up more storage space. Therefore, all of the models offer great speed that would allow the semantic search application to return results at blazing-fast speed.

Model Name	**Output Dim.**	**Index Size (MB)**	**Avg Inference Time (ms)**
MiniLM-L12-v2	384	153.60	17
distiluse-cased-v1	512	204.80	9
mpnet-base-v2	768	307.20	17
quora-distilbert	768	307.20	9

Table 5.15: Inference speed of Multilingual models

Selection of Best Model

In conclusion, the models perform reasonably well on the quora beir dataset for Recall@k and MRR@k, MiniLM-L12-v2 is the clear winner for both metrics, while mpnet-base-v2 almost matches the performance of MiniLM-L12-v2. For the german-dpr dataset, mpnet-base-v2 is the clear winner for both metrics, although distiluse-cased-v1 offers similar performance. The models don't do well for MRR@k for this dataset but perform well for Recall@k. In terms of inference speed, all models offer great speed for the process of returning results. We selected mpnet-base-v2 because of its overall good performance in all experiments satisfying our requirements in terms of serving our users with relevant results at great speed.

5.3 Summary

In this chapter, we started by providing the analysis of comparing the different storage backends on five core tasks using the data we created. Then we presented the analysis by comparing the models on two datasets using two information retrieval metrics that are relevant to our problem in building the application prototype. Finally, we also compared the inference speed of the models to better understand their performance.

6 Conclusion and Future Work

In this chapter, we present the conclusions drawn from the thesis work, as well as potential future work to improve the search application and further research directions. The chapter has the following structure:

- In Section 6.1, we discuss the essential conclusions drawn from our thesis work.
- In Section 6.2, we provide potential future work to improve the search application and its features, as well as to further extend our work.

6.1 Conclusion

In this thesis work, we have compared a relational database, a vector database and a relational database with an extension 'pgvector' in terms of their capabilities to store data in the form of vectors and also developed a prototype for a search application that can handle queries in multiple languages for Question and Answering Platforms such as Quora using semantic search. Firstly, we have provided the essential background information required to develop and design a semantic search application. This involved exploring the relevant concepts and techniques of NLP and DL that help in understanding Semantic Search. We compared the three storage backends to understand their performance capabilities on five core tasks related to Data management. We also compared Multilingual models to understand their retrieval capabilities using the BEIR benchmark for zero-shot evaluation and also comparing their speed in returning search results. The following conclusions were drawn from our study:

- *Performance of Storage Backends*: The results from our experiments showed that when storing vectors using ARRAY(FLOAT) datatype in a relational database like PostgreSQL, the Create, Read, Update and Delete (CRUD) operations are not efficient and we cannot perform a comparison of vectors using cosine similarity which is required to perform a semantic search. While the PostgreSQL pgvector extension provides the most efficient performance and allows the comparison of vectors and building a vector index that uses IVF, it is currently in earlier stages with version 0.4.0 and is used for scenarios where high search quality is a requirement at a reasonable speed, whereas Pinecone currently at version 2.0, is a vector database purpose-built for storing vectors and uses a proprietary graph-based index to provide high search quality alongside high speed. The performance of PostgreSQL pgvector in comparison to Pinecone was better, this may be due to the overhead involved in querying the index across the internet such as network, authentication, parsing, etc that affect the speed of Pinecone.

- *Performance of Multilingual Models*: The zero-shot evaluation of Multilingual models on the two datasets from the BEIR benchmark, showed that, with an increasing number of results that are returned, there are improvements in performance for Recall@k, MRR@k. The model 'paraphrase-multilingual-MiniLM-L12-v2' and 'paraphrase-multilingual-mpnet-base-v2' are the best-performing models for the BEIR-Quora dataset in English and 'paraphrase-multilingual-mpnet-base-v2' outperforms other models for Germandpr-beir dataset in German. When comparing their inference speed, the models 'distiluse-base-multilingual-cased-v1' and 'quora-distilbert-multilingual' perform slightly better than the other two models, using only 9 milliseconds to encode the query and return top-k similar results while the other two models took 17 milliseconds.
- The choice of Pinecone as the storage backend for the Prototype of the Semantic search application was based on several factors. Firstly, it offers a simple API to incorporate vector search that offers high performance for a semantic search application. Secondly, since it's a managed, cloud-native vector database, there is no need to set up and maintain the infrastructure that allows for easier development and deployment of the search application. It offers high search speed, which is essential for a search application that needs to handle large amounts of data efficiently. It also provides live index updates and keeps the index fresh and up-to-date enhancing the search functionality of the application. These features serve our requirements for the prototype of the semantic search application. The selection of 'paraphrase-multilingual-mpnet-base-v2' as the embedding model for the prototype of the semantic search application because it performs well for both datasets as compared to other models in the zero-shot evaluation and also offers high inference speed.

6.2 Future Work

In this section, we discuss potential future work to further improve on the approach presented in this thesis work:

- *Selection of Storage Backends*: We have presented the performance comparison of three storage backends, a relational database, a relational database with an extension and a vector database, it would be worthwhile to investigate the performance of other vector databases.
- *Selection of Models*: In model comparison, some of the limitations of our approach are that we did not perform any fine-tuning of the models which would further boost performance for specific domains and also we evaluated the models only for two languages on monolingual datasets. Comparing model performance over more datasets in different languages would also give us more insights into model performance.
- *Comparison of Vector Indexes*: We have used the default vector index offered by Pinecone for our application prototype. It would be worthwhile to investigate the performance of different available vector indexes and their impact in terms of search quality and search speed.

Bibliography

[Aar] Eric Aaron. Search engines that learn from experience. pages 24–25.

[AB07] Dirk Ahlers and Susanne Boll. Location-based web search. *The Geospatial Web: How Geobrowsers, Social Software and the Web 2.0 are Shaping the Network Society*, pages 55–66, 2007.

[Alc17] Michael A. Alcorn. (batter|pitcher)2vec: Statistic-free talent modeling with neural player embeddings, 2017.

[Anka] Ankane. pgvector: Open-source vector similarity search for postgres. https://pgxn.org/dist/vector/#Getting.Started.

[Ankb] Ankane. pgvector: Open-source vector similarity search for postgres. https://github.com/pgvector/pgvector.

[AoE13] S. I. Ao and International Association of Engineers. *International MultiConference of Engineers and Computer Scientists : IMECS 2013 : 13-15 March, 2013, the Royal Garden Hotel, Kowloon, Hong Kong*. Newswood Ltd., 2013.

[AS19] Navedanjum Ansari and Rajesh Sharma. Identifying semantically duplicate questions using data science approach: A quora case study, 2019. URL: http://qim.fs.quoracdn.net/quora_duplicate_questions.tsv.

[B+] Erik Bernholtz et al. ann-benchmarks. https://github.com/erikbern/ann-benchmarks.

[BA15] SSathya Bama and MSIrfan Ahmed. A survey on performance evaluation measures for information retrieval system. *International Research Journal of Engineering and Technology*, 2015. URL: www.irjet.net.

[bei] Beir quora: A dataset for benchmarking information retrieval models on quora. https://huggingface.co/datasets/BeIR/quora.

[BK] Lars Backstrom and Jon Kleinberg. Three and a half degrees of separation. *Facebook Research Blog*.

[BN13] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In Nieves R. Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2013. doi:10.1007/978-3-642-41062-8_28.

[Bro] Andrei Broder. A taxonomy of web search. URL: http://www.nationalcar.com.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. *Proceedings of NSDI 2010: 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 313–327, 2010.

[ČK19] Roman Čerešňák and Michal Kvet. Comparison of query performance in relational a non-relation databases. *Transportation Research Procedia*, 40:170–177, 2019.

[CWL+18] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. *SPTAG: A library for fast approximate nearest neighbor search*, 2018. URL: https://github.com/Microsoft/SPTAG.

[CYC+18] Muthuraman Chidambaram, Yinfei Yang, Daniel Cer, Steve Yuan, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Learning cross-lingual sentence representations via a multi-task dual-encoder model. 10 2018. URL: http://arxiv.org/abs/1810.12836.

[CZL+17] Ye Cheny, Ke Zhouz, Yiqun Liuy, Min Zhangy, and Shaoping May. Meta-evaluation of online and offlineweb search evaluation metrics. pages 15–24. Association for Computing Machinery, Inc, 8 2017. doi:10.1145/3077136.3080804.

[Dat] Hugging Face Datasets. Quora. https://huggingface.co/datasets/quora.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 10 2018. URL: http://arxiv.org/abs/1810.04805.

[Ela21] Elastic. Elasticsearch: Open source, distributed, restful search engine. https://github.com/elastic/elasticsearch, 2021.

[Exc] Stack Exchange. Stack exchange. URL: https://stackexchange.com/.

[Fak] Faker: A python library for fake data generation. URL: https://faker.readthedocs.io/en/master/index.html.

[far] Powering ai with vector databases: A benchmark (part i). https://www.farfetchtechblog.com/en/blog/post/powering-ai-with-vector-databases-a-benchmark-part-i/.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GLX+22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. Manu: a cloud native vector database management system. *arXiv preprint arXiv:2206.13843*, 2022.

[Goo23] Google. Issue 265172065: Bibtex export for groups and project library. https://issuetracker.google.com/issues/265172065, 2023.

[Gro21] PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/, 2021.

[GSL+20] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020. URL: https://arxiv.org/abs/1908.10396.

[Gui21] First Site Guide. Big data statistics and facts (editor's choice). https://firstsiteguide.com/big-data-stats/, 2021.

[Inc] Foundation Inc. 31+ quora statistics that you need to know in 2021. URL: https://foundationinc.co/lab/quora-statistics/.

[JDJ21] Jeff Johnson, Matthijs Douze, and Herve Jegou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2021. arXiv:1702.08734, doi:10.1109/TBDATA.2019.2921572.

[Kan] Dmitry Kan. Milvus, pinecone, vespa, weaviate, vald, gsi: What unites these buzz words and what makes each of them unique. https://towardsdatascience.com/milvus-pinecone-vespa-weaviate-vald-gsi-what-unites-%these-buzz-words-and-what-makes-each-9c65a3bd0696.

[Kar] Andrej Karpathy. Software 2.0. medium. com (2017).

[KKH04] Kevin Kline, Daniel Kline, and Brand Hunt. *SQL in a nutshell: a desktop quick reference*. " O'Reilly Media, Inc.", 2004.

[KSS+03] Masaru Kitsuregawa, Betty Salzberg, David Simmen, Tapas Nayak, Ricardo Baeza-yates, Erkki Sutinen, Jorma Tarhio, Panagiotis G Ipeirotis, H V Jagadish, Nick Koudas, S Muthukrishnan, Lauri Pietarinen, and Divesh Srivastava. Ø ò ò ö ò. 24(4), 2003.

[LBH15] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning, 5 2015. doi:10.1038/nature14539.

[LCG+19] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. 9 2019. URL: http://arxiv.org/abs/1909.11942.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. 7 2019. URL: http://arxiv.org/abs/1907.11692.

[Mal18] Yu. A. Malkov. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 31–33, 2018. URL: https://github.com/nmslib/nmslib{%}0Ahttp://ann-benchmarks.com/hnsw(nmslib).html.

[MCCD13a] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality arxiv : 1310 . 4546v1 [cs . cl] 16 oct 2013. *arXiv preprint arXiv:1310.4546*, cs.CL, 2013.

[MCCD13b] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, pages 1–12, 2013. arXiv:1301.3781.

[MK16] Vijaymeena M.K and Kavitha K. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal*, 3:19–28, 3 2016. doi:10.5121/mlaij.2016.3103.

[ML6] ML6. Semantic search: Introduction and business applications. https://www.ml6.eu/knowhow/semantic-search-intro-and-business-applications.

[MRP21] Timo Möller, Julian Risch, and Malte Pietsch. Germanquad and germandpr: Improving non-english question answering and passage retrieval, 2021. arXiv:2104.12741.

[Ove] Stack Overflow. Stack overflow. URL: https://stackoverflow.com/.

[PA] PM-AI. Germandpr for beir. https://huggingface.co/datasets/PM-AI/germandpr-beir.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020. doi:10.5281/zenodo.3509134.

[pina] Pinecone security. URL: https://www.pinecone.io/security/.

[Pinb] Pinecone. Dense vector embeddings for nlp. https://www.pinecone.io/learn/dense-vector-embeddings-nlp/.

[Pinc] Pinecone. K-nearest neighbor.

[Pind] Pinecone. Multilingual transformers. https://www.pinecone.io/learn/multilingual-transformers/.

[Pine] Pinecone. Offline evaluation. URL: https://www.pinecone.io/learn/offline-evaluation/.

[Pinf] Pinecone. Semantic search. https://www.pinecone.io/learn/semantic-search/.

[Ping] Pinecone. Sentence embeddings: What, why, and how. https://www.pinecone.io/learn/sentence-embeddings/.

[Pinh] Pinecone. Vector indexes. URL: https://www.pinecone.io/learn/vector-indexes/#indexes-in-search.

[Pini] Pinecone. Vector similarity.

[Pinj] Pinecone. What is a vector database? https://www.pinecone.io/learn/vector-database/.

[Pin22] Pinecone. Pinecone. https://www.pinecone.io/, 2022.

[QA18] Shahzad Qaiser and Ramsha Ali. Text mining: Use of tf-idf to examine the relevance of words to documents. *International Journal of Computer Applications*, 181:25–29, 7 2018. doi:10.5120/ijca2018917395.

[Qdr22] Qdrant. Qdrant. https://qdrant.tech/, 2022.

[Quo] Quora. Quora for business. URL: https://business.quora.com/.

[Red] Reddit. Reddit. URL: https://www.reddit.com/.

[RG] Nils Reimers and Iryna Gurevych. Semantic search with sbert. https://sbert.net/examples/applications/semantic-search/README.html. Accessed on March 24, 2023.

[RG19] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL: https://arxiv.org/abs/1908.10084.

[RG20] Nils Reimers and Iryna Gurevych. Making monolingual sentence embeddings multilingual using knowledge distillation. 4 2020. URL: http://arxiv.org/abs/2004.09813.

[RG21] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. https://www.sbert.net/, 2021.

[Rog19] Ian Rogers. The Google Pagerank Algorithm and How It Works or PhD. pages 1–16, 2019.

[SDCW19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. 10 2019. URL: http://arxiv.org/abs/1910.01108.

[Spo21] Spotify. Annoy: Approximate nearest neighbors oh yeah. https://github.com/spotify/annoy, 2021.

[SSKZ18] Parikshit Sondhi, Mohit Sharma, Pranam Kolari, and Chengxiang Zhai. A taxonomy of queries for e-commerce search. pages 1245–1248. Association for Computing Machinery, Inc, 6 2018. doi:10.1145/3209978.3210152.

[ST22] Semi-Technologies. Weaviate. https://github.com/semi-technologies/weaviate, 2022.

[Sta] Statista. Worldwide popularity ranking of database management systems. https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/.

[Teo19] Tommaso Teofili. *Deep Learning for Search*. Manning Publications, 2019.

[TRR+21] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL: https://openreview.net/forum?id=wCu6T5xFjeJ.

[Tun09] Daniel Tunkelang. *Introduction: What Are Facets?*, pages 3–9. Springer International Publishing, Cham, 2009. doi:10.1007/978-3-031-02262-3_1.

[vda22] vdaas. Vald. https://github.com/vdaas/vald, 2022.

[Ves22] Vespa. Vespa. https://vespa.ai/, 2022.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 6 2017. URL: http://arxiv.org/abs/1706.03762.

[WBB] Wang Wei, Payam M Barnaghi, and Andrzej Bargiela. Search with meanings: An overview of semantic search systems. URL: http://www.w3.org/TR/owl-guide/.

[WDS+] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Hugging face's transformers: State-of-the-art natural language processing. https://huggingface.co/models.

[Wea] Weaviate. Vector library vs vector database. https://weaviate.io/blog/vector-library-vs-vector-database/.

[WGM+13] Gang Wang, Konark Gill, Manish Mohanlal, Haitao Zheng, and Ben Y Zhao. Wisdom in the social crowd: an analysis of quora. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1341–1352, 2013.

[Whi15] Martin White. *Enterprise search: enhancing business performance.* " O'Reilly Media, Inc.", 2015.

[WMW+22] Bo Wang, Cristian Mitroi, Feng Wang, Shubham Saboo, and Susana Guzmán. *Neural Search - From Prototype to Production with Jina.* Packt Publishing, 2022.

[WWW+20] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020. doi:10.14778/3415478.3415541.

[WYG+21] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. pages 2614–2627. Association for Computing Machinery, 2021. doi:10.1145/3448016.3457550.

[YCA+19] Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Multilingual universal sentence encoder for semantic retrieval. 7 2019. URL: http://arxiv.org/abs/1907.04307.

[YLFW20] Wen Yang, Tao Li, Gai Fang, and Hong Wei. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2241–2253, 2020. doi:10.1145/3318464.3386131.

[Zha16] Z Zhang. Introduction to machine learning: k-nearest neighbors. *Annals of Translational Medicine*, 2016. doi:10.21037/atm.2016.03.37.

[Zil] Zilliz. Milvus. https://milvus.io/docs/index.md.

[ZRB+16] Ye Zhang, Md Mustafizur Rahman, Alex Braylan, Brandon Dang, Heng-Lu Chang, Henna Kim, Quinten McNamara, Aaron Angert, Edward Banner, Vivek Khetan, Tyler McDonnell, An Thanh Nguyen, Dan Xu, Byron C. Wallace, and Matthew Lease. Neural information retrieval: A literature review. 11 2016. URL: http://arxiv.org/abs/1611.06792.

Appendices

A. Benchmark Performance for Different Storage backends

Imports

```
from dotenv import load_dotenv
load_dotenv()

from faker import Faker
fake = Faker()
Faker.seed(0)

import itertools

import matplotlib.pyplot as plt
plt.style.use("ggplot")
plt.rcParams["axes.grid"] = True
plt.rcParams["figure.facecolor"] = "white"
plt.rcParams["axes.facecolor"] = "white"

import numpy as np

import os

import pandas as pd
pd.set_option("display.max_columns", 10)
pd.set_option("display.float_format", "{:.2f}".format)
pd.set_option("display.width", 200)
import pinecone
import psycopg2
from pgvector.sqlalchemy import Vector

import random
random.seed(42)

import sqlalchemy
from sqlalchemy import create_engine, inspect
from sqlalchemy.sql import text
from sqlalchemy.types import Integer, Float, Text, ARRAY
import sys

import time

import warnings
warnings.simplefilter("ignore", category=FutureWarning)
```

Connection Credentials

```
DB = os.getenv("POSTGRES_DB")
USER = os.getenv("POSTGRES_USER")
PW = os.getenv("POSTGRES_PW")
HOST = "localhost"

PINECONE_API_KEY = os.getenv("PINECONE_API_KEY")
```

Utility Functions

```
def postgres_connection(db: str, user: str,
                        pw: str, host: str) -> sqlalchemy.engine.base.Engine:

    engine = create_engine(f"postgresql+psycopg2://{USER}:{PW}@{HOST}:5432/{DB}")
    return engine

def pinecone_connection(api_key: str,
                        environment: str = "us-east1-gcp",
                        index_name: str = "benchmark_db",
                        distance_metric: str = "cosine",
                        vector_dimension: int = 7):

    pinecone.init(api_key=api_key, environment=environment)
    pinecone.create_index(index_name, dimension=vector_dimension,
                          metric=distance_metric, pod_type="p1.x1")
    index = pinecone.Index(index_name)
    return index

def create_fake_dataframe(num_rows: int, vector_dimension: int) -> pd.DataFrame:

    data = {
            'id': [i for i in range(num_rows)],
            'sentence': [fake.sentence(nb_words = 30, variable_nb_words = True) for _ ↩
                in range(num_rows)],
            'embeddings': [[round(random.random(), 5) for _ in range(vector_dimension)]↩
                 for _ in range(num_rows)]    # to replicate the embeddings from ↩
                different models
            }
    df = pd.DataFrame(data)
    assert df.shape[0] == num_rows, f'Expected {num_rows} rows, but got {df.shape[0]}'
    assert len(df['embeddings'][0]) == vector_dimension, f'Expected {vector_dimension} ↩
        elements in embeddings column, but got {len(df["embeddings"][0])}'
    return df

def postgres_chunks(iterable, batch_size=100):
    """A helper function to break an iterable into chunks of size batch_size."""
    it = iter(iterable)
    chunk = list(itertools.islice(it, batch_size))
    while chunk:
        yield pd.DataFrame(chunk)
        chunk = list(itertools.islice(it, batch_size))

def pinecone_chunks(iterable, batch_size=100):
    """A helper function to break an iterable into chunks of size batch_size."""
    it = iter(iterable)
    chunk = tuple(itertools.islice(it, batch_size))
    while chunk:
        yield chunk
        chunk = tuple(itertools.islice(it, batch_size))
```

Benchmark Performance for Bulk Insertion(Create) - Task 1

```
ROW_SIZE = [10, 100, 200, 300, 400]
DIMENSION_SIZE = [384, 512, 768, 1024]
```

```
def benchmark_postgres_insertion(num_rows: list[int], vector_dimension: list[int]) -> ↩
    pd.DataFrame:

    postgres_insert_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "embedding_dimension",
            "insertion_time(s)"
        ]
    )

    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Connected to Postgres")
    time.sleep(30)

    print("start benchmarking")
    for vector_size in vector_dimension:
        for rows in num_rows:

            data_set = create_fake_dataframe(num_rows = rows, vector_dimension = ↩
                vector_size)

            print(f"Inserting data into Postgres with {rows} rows and {vector_size} ↩
                dimensions")

            start_time = time.perf_counter()
            data_set.to_sql("insert_benchmark", con=conn, if_exists="replace", index=↩
                False,
                dtype={
                    "id": Integer(),
                    "sentence": Text(),
                    "embeddings": ARRAY(Float)
                },
            )
            end_time = time.perf_counter()

            print(f"Insertion of {rows} rows and {vector_size} dimensions into Postgres↩
                 complete")
            print("-" * 70)
            insertion_time = end_time - start_time

            postgres_insert_df = postgres_insert_df.append(
                {
                    "database_name": "Postgres",
                    "row_count": rows,
                    "embedding_dimension": vector_size,
                    "insertion_time(s)": f"{insertion_time:.3f}"

                },
                ignore_index=True,
            )

        conn.execute("DROP TABLE insert_benchmark")
        print("Table dropped")
        time.sleep(10)


    print("Postgres Insertion Benchmarking complete")


    conn.close()


    postgres_insert_df.to_csv("benchmarking/benchmark_results/
    01_postgres_data_insertion_results.csv", index=False,)

    print("Results saved")

    return postgres_insert_df
```

```
def benchmark_pinecone_insertion(num_rows: list[int], vector_dimension: list[int]) -> ↩
    pd.DataFrame:
    pinecone_insert_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "embedding_dimension",
            "insertion_time(s)"
        ]
    )
    for vector_size in vector_dimension:
        index = pinecone_connection(api_key = PINECONE_API_KEY, index_name = "pinecone-↩
            insertion-benchmark", vector_dimension = vector_size)
        print("Connected to Pinecone")
        time.sleep(60)

        for rows in num_rows:
            df = create_fake_dataframe(num_rows = rows, vector_dimension = vector_size)
            upsert_vectors = [
                (
                    str(row["id"]),
                    row["embeddings"],
                    {
                        "sentence": row["sentence"],
                    },
                )
                for _, row in df.iterrows()
            ]

            print(f"Inserting data into Pinecone with {rows} rows and {vector_size} ↩
                dimensions")
            start_time = time.perf_counter()
            index.upsert(vectors = upsert_vectors)
            end_time = time.perf_counter()

            print("Data Insertion complete")

            insertion_time = end_time - start_time

            pinecone_insert_df = pinecone_insert_df.append(
                {
                    "database_name": "Pinecone",
                    "row_count": rows,
                    "embedding_dimension": vector_size,
                    "insertion_time(s)": f"{insertion_time:.3f}"
                },
                ignore_index=True,
            )

        pinecone.delete_index("pinecone-insertion-benchmark")
        print(f"Index deleted for {vector_size} dimension vector")
        print("-" * 70)
        time.sleep(10)


    print("Pinecone Insertion Benchmarking complete")

    pinecone_insert_df.to_csv("benchmarking/benchmark_results/
    01_pinecone_insertion_benchmark.csv",index=False)
    print("Results saved")

    return pinecone_insert_df

def benchmark_postgres_pgvector_insertion(
    num_rows: list[int], vector_dimension: list[int]
) -> pd.DataFrame:
    postgres_insert_pgvector_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "embedding_dimension",
            "insertion_time(s)"
```

```
        ]
    )

    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Connected to Postgres")
    conn.execute(text('CREATE EXTENSION IF NOT EXISTS vector'))
    print("Created extension")
    time.sleep(30)

    print("start benchmarking")
    for vector_size in vector_dimension:
        for rows in num_rows:
            data_set = create_fake_dataframe(
                num_rows=rows, vector_dimension=vector_size
            )

            print(
                f"Inserting data into Postgres with {rows} rows and {vector_size} ↩
                    dimensions"
            )

            start_time = time.perf_counter()
            data_set.to_sql(
                "insert_pg_vector_benchmark",
                con=conn,
                if_exists="replace",
                index=False,
                dtype={
                    "id": Integer(),
                    "sentence": Text(),
                    "embeddings": Vector(vector_size),
                },
            )
            end_time = time.perf_counter()

            print(
                f"Insertion of {rows} rows and {vector_size} dimensions into Postgres ↩
                    complete"
            )
            print("-" * 70)
            insertion_time = end_time - start_time

            postgres_insert_pgvector_df = postgres_insert_pgvector_df.append(
                {
                    "database_name": "Postgres_pgvector_extension",
                    "row_count": rows,
                    "embedding_dimension": vector_size,
                    "insertion_time(s)": f"{insertion_time:.3f}",
                },
                ignore_index=True,
            )

        conn.execute("DROP TABLE insert_pg_vector_benchmark")
        print("Table dropped")
        time.sleep(10)

    print("Postgres Insertion Benchmarking complete")

    conn.close()

    postgres_insert_pgvector_df.to_csv(
        "benchmarking/benchmark_results/
        01_postgres_pgvector_data_insertion_results.csv",
        index=False,
    )

    print("Results saved")

    return postgres_insert_pgvector_df
```

Benchmark Performance for Extraction/Read - Task 2

```
ROWS = 400
VECTORS = 768
EXTRACT_TOP_K = [3, 5, 10, 100, 250]

def benchmark_postgres_extraction(rows: int, vectors: int, top_k: list[int]) -> pd.↩
    DataFrame:
    postgres_extract_df = pd.DataFrame(
        columns=[
            "database_name",
            "k",
            "extraction_time(s)",
            "row_count",
            "embedding_dimension"
        ]
    )

    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    time.sleep(30)

    print(f"Creating dataframe with {rows} rows and having embedding size of {vectors} ↩
        dimensions")
    data_set = create_fake_dataframe(num_rows = rows, vector_dimension = vectors)

    data_set.to_sql(
        "extract_benchmark",
        con = conn,
        if_exists = "replace",
        index = False,
        dtype={
            "id": Integer(),
            "sentence": Text(),
            "embeddings": ARRAY(Float)
        },
    )

    print("Inserting data into Postgres completed")

    print("Start Benchmarking")
    for k in top_k:
        start_time = time.perf_counter()
        query = f"SELECT * FROM extract_benchmark LIMIT {k}"
        df = pd.read_sql(query, conn)
        end_time = time.perf_counter()

        extraction_time = end_time - start_time

        # Append results to the dataframe
        postgres_extract_df = postgres_extract_df.append(
            {
                "database_name": "Postgres",
                "k": k,
                "extraction_time(s)": f"{extraction_time:.3f}",
                "row_count": rows,
                "embedding_dimension": vectors
            },
            ignore_index=True,
        )

    print("Benchmarking complete")

    conn.execute("DROP TABLE extract_benchmark")
    print("Table dropped")

    conn.close()
    print("Postgres connection closed")

    postgres_extract_df.to_csv("benchmarking/benchmark_results/
    02_postgres_data_extraction_results.csv", index = False)
    print("Results saved")

    return postgres_extract_df
```

```
def benchmark_pinecone_extraction(rows: int, vectors: int, top_k: list[int]) -> pd.↩
    DataFrame:
    pinecone_extract_df = pd.DataFrame(
        columns=[
            "database_name",
            "k",
            "extraction_time(s)",
            "row_count",
            "embedding_dimension"
        ]
    )

    index = pinecone_connection(api_key = PINECONE_API_KEY, index_name = "pinecone-↩
        extraction-benchmark", vector_dimension = vectors)
    time.sleep(30)

    df = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

    upsert_vectors = [
        (
            str(row["id"]),
            row["embeddings"],
            {
                "sentence": row["sentence"],
            },
        )
        for _, row in df.iterrows()
    ]

    index.upsert(vectors = upsert_vectors)
    print(f"Data added")

    for k in top_k:
        new_vector = np.random.rand(vectors).tolist()
        new_vector = [round(x, 5) for x in new_vector]
        start_time = time.perf_counter()
        index.query(vector = new_vector, top_k = k, include_values = True,)
        end_time = time.perf_counter()

        extraction_time = end_time - start_time

        pinecone_extract_df = pinecone_extract_df.append(
            {
                "database_name": "Pinecone",
                "k": k,
                "extraction_time(s)": f"{extraction_time:.3f}",
                "row_count": rows,
                "embedding_dimension": vectors
            },
            ignore_index=True,
        )

    print("Pinecone Extraction Benchmarking complete")

    pinecone.delete_index("pinecone-extraction-benchmark")
    print("Index deleted")

    pinecone_extract_df.to_csv("benchmarking/benchmark_results/
    02_pinecone_extraction_benchmark.csv",index=False,)
    print("Results saved")

    return pinecone_extract_df

def benchmark_postgres_pgvector_extraction(rows: int, vectors: int, top_k: list[int]) ↩
    -> pd.DataFrame:
    # Dataframe to store results
    postgres_pgvector_extract_df = pd.DataFrame(
        columns=[
            "database_name",
            "k",
            "extraction_time(s)",
            "row_count",
```

```
            "embedding_dimension"
        ]
    )

    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    conn.execute(text('CREATE EXTENSION IF NOT EXISTS vector'))
    print("Created extension")
    time.sleep(30)

    print(f"Creating dataframe with {rows} rows and having embedding size of {vectors} ↩
        dimensions")
    data_set = create_fake_dataframe(num_rows = rows, vector_dimension = vectors)

    data_set.to_sql(
        "extract_pgvector_benchmark",
        con = conn,
        if_exists = "replace",
        index = False,
        dtype={
            "id": Integer(),
            "sentence": Text(),
            "embeddings": Vector(vectors)
        },
    )
    example = data_set["embeddings"][0]
    print("Inserting data into Postgres completed")
    conn.execute("CREATE INDEX ON extract_pgvector_benchmark USING ivfflat (embeddings ↩
        vector_cosine_ops);")
    print("Start Benchmarking")
    for k in top_k:
        start_time = time.perf_counter()
        _ = conn.execute(f"SELECT * FROM extract_pgvector_benchmark ORDER BY embeddings↩
            <=> '{example}'  LIMIT {k};").fetchall()
        end_time = time.perf_counter()

        extraction_time = end_time - start_time

        # Append results to the dataframe
        postgres_pgvector_extract_df = postgres_pgvector_extract_df.append(
            {
                "database_name": "Postgres_pgvector_extension",
                "k": k,
                "extraction_time(s)": f"{extraction_time:.3f}",
                "row_count": rows,
                "embedding_dimension": vectors
            },
            ignore_index=True,
        )

    print("Benchmarking complete")

    conn.execute("DROP TABLE extract_pgvector_benchmark")
    print("Table dropped")

    conn.close()
    print("Postgres connection closed")

    postgres_pgvector_extract_df.to_csv("benchmarking/benchmark_results/
    02_postgres_pgvector_data_extraction_results.csv", index = False)
    print("Results saved")

    return postgres_pgvector_extract_df
```

Benchmark Performance for Update - Task 3

```
ROWS = 400
VECTORS = 768



def benchmark_postgres_updation(rows: int, vectors: int) -> pd.DataFrame:
    postgres_update_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "updation_time(s)",
            "embedding_dimension"
        ]
    )

    # create a connection to the Postgres database
    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    time.sleep(30)

    print(f"Creating dataframe with {rows} rows and having embedding size of {vectors} ↩
        dimensions")
    data_set = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

    data_set.to_sql(
        "update_benchmark",
        con=conn,
        if_exists="replace",
        index=False,
        dtype={
            "id": Integer(),
            "sentence": Text(),
            "embeddings": ARRAY(Float),
        },
    )

    print(f"Inserting data into Postgres")
    print(
        pd.read_sql_query(
            "SELECT embeddings as old_embedding FROM update_benchmark WHERE id = 1",
            con=conn,
        )
    )


    print("Start Benchmarking")
    time_taken = []
    for i in range(20):
        print(f"Starting {i} loops of 20 for update query")
        new_vector = np.random.rand(vectors).tolist()
        new_vector = [round(x, 5) for x in new_vector]
        start_time = time.perf_counter()
        query = text(
            "UPDATE update_benchmark SET embeddings = :updated_vector WHERE id = 1"
        )
        conn.execute(query, updated_vector=new_vector)
        end_time = time.perf_counter()
        time_taken.append(end_time - start_time)

    updation_time = sum(time_taken) / len(time_taken)
    print(
        pd.read_sql_query(
            "SELECT embeddings as updated_embedding FROM update_benchmark WHERE id = ↩
                1",
            con=conn,
        )
    )

    postgres_update_df = postgres_update_df.append(
```

```
        {
            "database_name": "Postgres",
            "row_count": rows,
            "updation_time(s)": f"{updation_time:.3f}",
            "embedding_dimension": vectors
        },
        ignore_index=True,
    )

    print("Postgres Updation Benchmarking complete")

    conn.execute("DROP TABLE update_benchmark")
    print("Table dropped")

    conn.close()
    print("Postgres connection closed")

    postgres_update_df.to_csv("benchmarking/benchmark_results/
    04_postgres_updation_benchmark_results.csv", index=False)
    print("Results saved")
    return postgres_update_df

def benchmark_pinecone_updation(rows: int, vectors: int) -> pd.DataFrame:
    pinecone_update_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "updation_time(s)",
            "embedding_dimension"
        ]
    )

    # Create index
    index = pinecone_connection(api_key = PINECONE_API_KEY, index_name = "pinecone-↩
        update-benchmark", vector_dimension = vectors)
    print("Connected to Pinecone")
    time.sleep(30)

    df = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)
    print("Dataframe created")

    upsert_vectors = [
        (
            str(row["id"]),
            row["embeddings"],
            {
                "sentence": row["sentence"],
            },
        )
        for _, row in df.iterrows()
    ]

    index.upsert(vectors=upsert_vectors)

    print(f"Data Insertion complete")

    old_vector = index.fetch(ids=['1'])
    print("Old vector:", old_vector['vectors']['1']['values'][:10])

    time_taken = []
    for i in range(20):
        print(f"Starting loop {i} out of 20 loops for update query")
        new_vector = np.random.rand(vectors).tolist()
        new_vector = [round(x, 5) for x in new_vector]
        start_time = time.perf_counter()
        index.update(
            id='1',
            values=new_vector
        )
        end_time = time.perf_counter()
        time_taken.append(end_time - start_time)

    updation_time = sum(time_taken) / len(time_taken)
```

```
    new_vector = index.fetch(ids=['1'])
    print("New vector", new_vector['vectors']['1']['values'][:10])

    pinecone_update_df = pinecone_update_df.append(
        {
            "database_name": "Pinecone",
            "row_count": rows,
            "updation_time(s)": f"{updation_time:.3f}",
            "embedding_dimension": vectors
        },
        ignore_index=True,
    )

    print("Pinecone Updation Benchmarking complete")

    pinecone.delete_index("pinecone-update-benchmark")
    print("Index deleted")

    pinecone_update_df.to_csv("benchmarking/benchmark_results/
    03_pinecone_updation_benchmark.csv", index=False)
    print("Results saved")

    return pinecone_update_df

def benchmark_postgres_pgvector_updation(rows: int, vectors: int) -> pd.DataFrame:
    postgres_pgvector_update_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "updation_time(s)",
            "embedding_dimension"
        ]
    )

    # create a connection to the Postgres database
    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    conn.execute(text('CREATE EXTENSION IF NOT EXISTS vector'))
    print("Created extension")
    time.sleep(30)

    print(f"Creating dataframe with {rows} rows and having embedding size of {vectors} ↩
        dimensions")
    data_set = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

    data_set.to_sql(
        "update_pgvector_benchmark",
        con=conn,
        if_exists="replace",
        index=False,
        dtype={
            "id": Integer(),
            "sentence": Text(),
            "embeddings": Vector(vectors),
        },
    )

    print(f"Inserting data into Postgres")
    print(
        pd.read_sql_query(
            "SELECT embeddings as old_embedding FROM update_pgvector_benchmark WHERE id↩
                = 1",
            con=conn,
        )
    )


    print("Start Benchmarking")
    time_taken = []
    for i in range(20):
        print(f"Starting {i} loops of 20 for update query")
```

```
        new_vector = np.random.rand(vectors).tolist()
        new_vector = [round(x, 5) for x in new_vector]
        start_time = time.perf_counter()
        query = text(
            "UPDATE update_pgvector_benchmark SET embeddings = :updated_vector WHERE id↩
                = 1"
        )
        conn.execute(query, updated_vector=new_vector)
        end_time = time.perf_counter()
        time_taken.append(end_time - start_time)

    updation_time = sum(time_taken) / len(time_taken)
    print(
        pd.read_sql_query(
            "SELECT embeddings as updated_embedding FROM update_pgvector_benchmark ↩
                WHERE id = 1",
            con=conn,
        )
    )

    postgres_pgvector_update_df = postgres_pgvector_update_df.append(
        {
            "database_name": "Postgres_pgvector_extension",
            "row_count": rows,
            "updation_time(s)": f"{updation_time:.3f}",
            "embedding_dimension": vectors
        },
        ignore_index=True,
    )

    print("Postgres Updation Benchmarking complete")

    conn.execute("DROP TABLE update_pgvector_benchmark")
    print("Table dropped")

    conn.close()
    print("Postgres connection closed")

    postgres_pgvector_update_df.to_csv("benchmarking/benchmark_results/
    04_postgres_updation_pgvector_benchmark_results.csv", index=False)
    print("Results saved")
    return postgres_pgvector_update_df
```

Benchmark Performance for Delete - Task 4

```
ROWS=400
VECTORS=768

def benchmark_postgres_deletion(rows: int, vectors: int) -> pd.DataFrame:
    postgres_delete_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "deletion_time(s)",
            "embedding_dimension"
        ]
    )

    # create a connection to the Postgres database
    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    time.sleep(30)

    print(f"Creating dataframe with {rows} rows and having embedding size of {vectors} ↩
        dimensions")
    data_set = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

    data_set.to_sql(
        "delete_benchmark",
        con=conn,
```

```
        if_exists="replace",
        index=False,
        dtype={
            "id": Integer(),
            "sentence": Text(),
            "embeddings": ARRAY(Float),
        },
    )

    start_time = time.perf_counter()

    query = text("DELETE FROM delete_benchmark WHERE id = 1")
    conn.execute(query)
    end_time = time.perf_counter()

    deletion_time = end_time - start_time


    postgres_delete_df = postgres_delete_df.append(
        {
            "database_name": "Postgres",
            "row_count": rows,
            "deletion_time(s)": f"{deletion_time:.3f}",
            "embedding_dimension": vectors
        },
        ignore_index=True,
    )

    print("Postgres Deletion Benchmarking complete")

    conn.execute("DROP TABLE delete_benchmark")
    print("Table dropped")

    conn.close()
    print("Postgres connection closed")

    postgres_delete_df.to_csv("benchmarking/benchmark_results/
    05_postgres_deletion_benchmark_results.csv", index=False)
    print("Results saved")
    return postgres_delete_df

def benchmark_pinecone_deletion(rows: int, vectors: int) -> pd.DataFrame:
    pinecone_delete_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "deletion_time(s)",
            "embedding_dimension"
        ]
    )

    # Create index
    index = pinecone_connection(api_key = PINECONE_API_KEY, index_name = "pinecone-↩
        delete-benchmark", vector_dimension = vectors)
    print("Connected to Pinecone")
    time.sleep(30)

    df = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)
    print("Dataframe created")

    upsert_vectors = [
        (
            str(row["id"]),
            row["embeddings"],
            {
                "sentence": row["sentence"],
            },
        )
        for _, row in df.iterrows()
    ]

    index.upsert(vectors=upsert_vectors)
```

```
    print(f"Data Insertion complete")

    start_time = time.perf_counter()
    index.delete(ids=['1'])
    end_time = time.perf_counter()
    print("Deletion complete")
    deletion_time = end_time - start_time

    pinecone_delete_df = pinecone_delete_df.append(
        {
            "database_name": "Pinecone",
            "row_count": rows,
            "deletion_time(s)": f"{deletion_time:.3f}",
            "embedding_dimension": vectors
        },
        ignore_index=True,
    )

    print("Pinecone Deletion Benchmarking complete")

    pinecone.delete_index("pinecone-delete-benchmark")
    print("Index deleted")

    pinecone_delete_df.to_csv("benchmarking/benchmark_results/
    04_pinecone_deletion_benchmark.csv", index=False)
    print("Results saved")

    return pinecone_delete_df

def benchmark_postgres_pgvector_deletion(rows: int, vectors: int) -> pd.DataFrame:
    postgres_pgvector_delete_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "deletion_time(s)",
            "embedding_dimension"
        ]
    )

    # create a connection to the Postgres database
    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    conn.execute(text('CREATE EXTENSION IF NOT EXISTS vector'))
    print("Created extension")
    time.sleep(30)

    print(f"Creating dataframe with {rows} rows and having embedding size of {vectors} ↩
        dimensions")
    data_set = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

    data_set.to_sql(
        "delete_pgvector_benchmark",
        con=conn,
        if_exists="replace",
        index=False,
        dtype={
            "id": Integer(),
            "sentence": Text(),
            "embeddings": Vector(vectors),
        },
    )



    print("Start Benchmarking")

    start_time = time.perf_counter()
    query = text("DELETE FROM delete_pgvector_benchmark WHERE id = 1")
    conn.execute(query)
    end_time = time.perf_counter()
    deletion_time = end_time - start_time
```

```
    postgres_pgvector_delete_df = postgres_pgvector_delete_df.append(
        {
            "database_name": "Postgres_pgvector_extension",
            "row_count": rows,
            "deletion_time(s)": f"{deletion_time:.3f}",
            "embedding_dimension": vectors
        },
        ignore_index=True,
    )

    print("Deletion Benchmarking complete")

    conn.execute("DROP TABLE delete_pgvector_benchmark")
    print("Table dropped")

    conn.close()
    print("Postgres connection closed")

    postgres_pgvector_delete_df.to_csv("benchmarking/benchmark_results/
    05_postgres_deletion_pgvector_benchmark_results.csv", index=False)
    print("Results saved")
    return postgres_pgvector_delete_df
```

Benchmark Performance for Batch Insertion - Task 5

```
BATCH_SIZE = 100
BATCH_ROW_SIZE = [10000, 30000, 50000, 70000, 100000]
DIMENSION_SIZE = 768

def benchmark_postgres_batch_insertion(num_rows: list[int], vectors: int, batch_size: ↩
    int) -> pd.DataFrame:
    postgres_batch_insert_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "batch_size",
            "embedding_dimension",
            "insertion_time(mins)"
        ]
    )

    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    time.sleep(30)

    for rows in num_rows:
        time.sleep(60)
        print(f"Creating dataframe with {rows} rows and having embedding size of {↩
            vectors} dimensions")
        data_set = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

        print("Inserting data into Postgres")
        print(f"Total number of batches: {rows // batch_size}")
        batch_count = 0
        time_taken = []
        for data_chunk in postgres_chunks(data_set.itertuples(index = False), ↩
            batch_size):
            start_time = time.perf_counter()
            data_chunk.to_sql(
                "insert_batch_benchmark",
                con=conn,
                if_exists="append",
                index=False,
                chunksize=batch_size,
                dtype={
                    "id": Integer(),
                    "sentence": Text(),
                    "embeddings": ARRAY(Float),
                },
            )
```

```
            end_time = time.perf_counter()
            time_taken.append(end_time - start_time)
            batch_count += 1
            if batch_count % 50 == 0:
                time_for_50 = sum(time_taken[-50:])
                time_in_minutes = time_for_50 / 60
                print(f"time taken for batch {batch_count}: {time_in_minutes:.2f} ↩
                    minutes")


        insertion_time = sum(time_taken) / 60
        print(f"Total time taken for {rows} rows: {insertion_time:.2f} minutes")

        postgres_batch_insert_df = postgres_batch_insert_df.append(
            {
                "database_name": "Postgres",
                "row_count": rows,
                "batch_size": batch_size,
                "embedding_dimension": vectors,
                "insertion_time(mins)": f"{insertion_time:.2f}"
            },
            ignore_index=True,
        )

    conn.execute("DROP TABLE insert_batch_benchmark")
    print("Table dropped")

    print("Benchmarking complete")

    conn.close()
    print("Postgres connection closed")

    postgres_batch_insert_df.to_csv("benchmarking/benchmark_results/
    03_postgres_batch_insertion_results.csv", index=False)
    print("Results saved")

    return postgres_batch_insert_df

def benchmark_pinecone_batch_insertion(num_rows: list[int], vectors: int, batch_size: ↩
    int) -> pd.DataFrame:
    pinecone_batch_insert_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "batch_size",
            "embedding_dimension",
            "insertion_time(mins)"
        ]
    )
    index = pinecone_connection(api_key = PINECONE_API_KEY, index_name = "pinecone-↩
        batch-benchmark", vector_dimension = vectors)
    time.sleep(30)
    for rows in num_rows:
        time.sleep(60)
        df = create_fake_dataframe(num_rows=rows, vector_dimension = vectors)
        upsert_vectors = [
            (
                str(row["id"]),
                row["embeddings"],
                {
                    "sentence": row["sentence"],
                },
            )
            for _, row in df.iterrows()
        ]
        time_taken = []
        print(f"Total number of batches: {rows // batch_size}")
        batch_count = 0
        for data_chunk in pinecone_chunks(upsert_vectors, batch_size=batch_size):
                start_time = time.perf_counter()
                index.upsert(vectors=data_chunk)
                end_time = time.perf_counter()
                time_taken.append(end_time - start_time)
```

```
                batch_count += 1
                if batch_count % 50 == 0:
                    time_for_50 = sum(time_taken[-50:])
                    time_in_mins = time_for_50 / 60
                    print(f"time taken for batch {batch_count}: {time_in_mins:.2f} ↩
                        minutes")

        insertion_time = sum(time_taken) / 60
        print(f"Total time taken to insert data into Pinecone: {insertion_time:.2f} ↩
            minutes")

        pinecone_batch_insert_df = pinecone_batch_insert_df.append(
            {
                "database_name": "Pinecone",
                "row_count": rows,
                "batch_size": batch_size,
                "embedding_dimension": vectors,
                "insertion_time(mins)": f"{insertion_time:.2f}"
            },
            ignore_index=True,
            )

    print("Pinecone Batch Insertion Benchmarking complete")

    pinecone.delete_index("pinecone-batch-benchmark")
    print("Index deleted")

    pinecone_batch_insert_df.to_csv("benchmarking/benchmark_results/
    03_pinecone_batch_insertion_results.csv", index = False)
    print("Results saved")

    return pinecone_batch_insert_df

def benchmark_postgres_pgvector_batch_insertion(num_rows: list[int], vectors: int, ↩
    batch_size: int) -> pd.DataFrame:
    postgres_pgvector_batch_insert_df = pd.DataFrame(
        columns=[
            "database_name",
            "row_count",
            "batch_size",
            "embedding_dimension",
            "insertion_time(mins)"
        ]
    )

    engine = postgres_connection(db=DB, user=USER, pw=PW, host=HOST)
    conn = engine.connect()
    print("Postgres connection created")
    conn.execute(text('CREATE EXTENSION IF NOT EXISTS vector'))
    print("Created extension")
    time.sleep(30)

    for rows in num_rows:
        time.sleep(60)
        print(f"Creating dataframe with {rows} rows and having embedding size of {↩
            vectors} dimensions")
        data_set = create_fake_dataframe(num_rows=rows, vector_dimension=vectors)

        print("Inserting data into Postgres")
        print(f"Total number of batches: {rows // batch_size}")
        batch_count = 0
        time_taken = []
        for data_chunk in postgres_chunks(data_set.itertuples(index = False), ↩
            batch_size):
            start_time = time.perf_counter()
            data_chunk.to_sql(
                "insert_pgvector_batch_benchmark",
                con=conn,
                if_exists="append",
                index=False,
                chunksize=batch_size,
                dtype={
                    "id": Integer(),
```

```
                    "sentence": Text(),
                    "embeddings": Vector(vectors),
                },
            )
            end_time = time.perf_counter()
            time_taken.append(end_time - start_time)
            batch_count += 1
            if batch_count % 50 == 0:
                time_for_50 = sum(time_taken[-50:])
                time_in_minutes = time_for_50 / 60
                print(f"time taken for batch {batch_count}: {time_in_minutes:.2f} ↩
                    minutes")


        insertion_time = sum(time_taken) / 60
        print(f"Total time taken for {rows} rows: {insertion_time:.2f} minutes")

        postgres_pgvector_batch_insert_df = postgres_pgvector_batch_insert_df.append(
            {
                "database_name": "Postgres_pgvector_extension",
                "row_count": rows,
                "batch_size": batch_size,
                "embedding_dimension": vectors,
                "insertion_time(mins)": f"{insertion_time:.2f}"
            },
            ignore_index=True,
        )

    conn.execute("DROP TABLE insert_pgvector_batch_benchmark")
    print("Table dropped")

    print("Benchmarking complete")

    conn.close()
    print("Postgres connection closed")

    postgres_pgvector_batch_insert_df.to_csv("benchmarking/benchmark_results/
    03_postgres_pgvector_batch_insertion_results.csv", index=False)
    print("Results saved")

    return postgres_pgvector_batch_insert_df
```

Visualizations

```
def plot_row_benchmark(
        df_1: pd.DataFrame,
        df_2: pd.DataFrame,
        df_3: pd.DataFrame,
        row_size: int,
        save_path: str = "insertion_plot.png",
        title: str = "Insertion Time vs Dimension Size",
        sup_title = "Insertion Performance of Postgres and Pinecone for varying ↩
            Dimension Size",
        x_label: str = "Dimension Size",
        y_label: str = "Time(s)"
        ):

    plt.figure(figsize=(10, 5))
    df_1 = df_1[(df_1["row_count"] == row_size)]
    df_2 = df_2[(df_2["row_count"] == row_size)]
    df_3 = df_3[(df_3["row_count"] == row_size)]

    times_1 = df_1["insertion_time(s)"]
    dimensions_1 = df_1["embedding_dimension"]

    times_2 = df_2["insertion_time(s)"]
    dimensions_2 = df_2["embedding_dimension"]

    times_3 = df_3["insertion_time(s)"]
    dimensions_3 = df_3["embedding_dimension"]
```

```
    plt.plot(dimensions_1, times_1, marker = "o", label="Postgres")
    plt.plot(dimensions_2, times_2, marker = "x", label="Pinecone")
    plt.plot(dimensions_3, times_3, marker = "s", label="Postgres with pgvector", color↩
        ="green")

    plt.title(title, fontsize=12)
    plt.suptitle(sup_title, fontsize=12)

    plt.xticks(dimensions_1)
    plt.xlabel(x_label)

    plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.2)

    plt.ylabel(y_label)
    plt.yticks()

    plt.legend(title='Storage Backend',
               labels= ["Postgres", "Pinecone", "Postgres with pgvector"],
               fontsize=10,
               bbox_to_anchor=(1, 1),
                borderaxespad=0.7,
                framealpha=0.7
                )

    plt.savefig(save_path, bbox_inches='tight', dpi=300)
    plt.show()

def plot_vector_benchmark(
        df_1: pd.DataFrame,
        df_2: pd.DataFrame,
        df_3: pd.DataFrame,
        vector_size: int,
        save_path: str = "insertion_plot.png",
        title: str = "Insertion Time(s) vs Row Count",
        sup_title = "Insertion Performance for 384 Dimensional Embedding with varying ↩
            Row Count",
        x_label: str = "Row Count",
        y_label: str = "Time(s)"
        ):

    plt.figure(figsize=(10, 5))
    df_1 = df_1[(df_1["embedding_dimension"] == vector_size)]
    df_2 = df_2[(df_2["embedding_dimension"] == vector_size)]
    df_3 = df_3[(df_3["embedding_dimension"] == vector_size)]


    times_1 = df_1["insertion_time(s)"]
    rows_1 = df_1["row_count"]

    times_2 = df_2["insertion_time(s)"]
    rows_2 = df_2["row_count"]

    times_3 = df_3["insertion_time(s)"]
    rows_3 = df_3["row_count"]

    plt.plot(rows_1, times_1, marker = "o", label="Postgres")
    plt.plot(rows_2, times_2, marker = "x", label="Pinecone")
    plt.plot(rows_3, times_3, marker = "s", label="Postgres with pgvector", color="↩
        green")


    plt.title(title, fontsize=12)
    plt.suptitle(sup_title, fontsize=12)

    plt.xticks(rows_1)
    plt.xlabel(x_label)

    plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.2)

    plt.ylabel(y_label)
    plt.yticks()
```

```
    plt.legend(title='Storage Backend', labels = ["Postgres", "Pinecone", "Postgres ↩
        with pgvector"],
               fontsize=10, bbox_to_anchor=(1, 1), borderaxespad=0.7, framealpha=0.7)
    plt.savefig(save_path, bbox_inches='tight', dpi=300)
    plt.show()

def plot_topk_benchmark(df_1: pd.DataFrame,
                        df_2: pd.DataFrame,
                        df_3: pd.DataFrame,
                        save_path: str = "01_extraction_plot.png",
                        title: str = "Extraction Time vs Top K",
                        sup_title: str = "Extraction Performance Comparison for ↩
                            different K values",
                        x_label: str = "Top K",
                        y_label: str = "Time(s)"):

    plt.figure(figsize=(10, 5))

    times_1 = df_1["extraction_time(s)"]
    topk_1 = df_1["k"]

    times_2 = df_2["extraction_time(s)"]
    topk_2 = df_2["k"]

    times_3 = df_3["extraction_time(s)"]
    topk_3 = df_3["k"]


    plt.plot(topk_1, times_1, marker = "o", label="Postgres")
    plt.plot(topk_2, times_2, marker = "x", label="Pinecone")
    plt.plot(topk_3, times_3, marker = "s", label="Postgres with pgvector", color = "↩
        green")


    plt.title(title, fontsize=12)
    plt.suptitle(sup_title, fontsize=12)


    plt.xlabel(x_label)
    plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.2)

    plt.ylabel(y_label)
    plt.yticks()

    plt.legend(title='Storage Backend',
               labels= ["Postgres", "Pinecone", "Postgres with pgvector"],
               fontsize=10,
               bbox_to_anchor=(1, 1),
                borderaxespad=0.7,
                framealpha=0.7
                )

    plt.savefig(save_path, bbox_inches='tight', dpi=300)
    plt.show()

def plot_batch_benchmark(df_1: pd.DataFrame,
                        df_2: pd.DataFrame,
                        df_3: pd.DataFrame,
                        save_path: str = "insertion_plot.png",
                        title: str = "Insertion Time(mins) vs Row Count ",
                        x_label: str = "Row Count, batch_size=100",
                        y_label: str = "Time(mins)"):

    plt.figure(figsize=(10, 5))

    times_1 = df_1["insertion_time(mins)"]
    rows_1 = df_1["row_count"]

    times_2 = df_2["insertion_time(mins)"]
    rows_2 = df_2["row_count"]

    times_3 = df_3["insertion_time(mins)"]
```

```
    rows_3 = df_3["row_count"]

    plt.plot(rows_1, times_1, marker = "o", label="Postgres")
    plt.plot(rows_2, times_2, marker = "x", label="Pinecone")
    plt.plot(rows_3, times_3, marker = "s", label="Postgres with pgvector", color = "↩
        green")


    plt.title(title, fontsize=12)
    plt.suptitle("Batch Insertion Performance Comparison: Postgres Vs Pinecone", ↩
        fontsize=12)


    plt.xlabel(x_label)
    plt.xticks()

    fmt = mtick.StrMethodFormatter("{x:,.0f}")
    plt.gca().xaxis.set_major_formatter(fmt)
    plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.2)
    plt.ylabel(y_label)
    plt.yticks()

    plt.legend(title='Storage Backend', labels= ["Postgres", "Pinecone", "Postgres with↩
         pgvector"], fontsize=10, bbox_to_anchor=(1, 1),
            borderaxespad=0.7, framealpha=0.7)

    plt.savefig(save_path, bbox_inches='tight', dpi=300)
    plt.show()
```

B. Benchmark Multilingual Model Performance

Imports

```
from beir import util, LoggingHandler
from beir.datasets.data_loader import GenericDataLoader
from beir.retrieval import models
from beir.retrieval.search.dense import DenseRetrievalExactSearch as DRES
from beir.retrieval.search.dense import util as utils
from beir.retrieval.evaluation import EvaluateRetrieval

import datetime
import datasets

import json

import logging

import numpy as np

import os

import pandas as pd
import pathlib

import random
random.seed(42)

import sys
from sentence_transformers import SentenceTransformer

import torch
from typing import List, Dict, Tuple
import time
```

Utility Functions

```
def download_dataset(dataset_name: str) -> None:
  url = f"https://public.ukp.informatik.tu-darmstadt.de/thakur/BEIR/datasets
  /{dataset_name}.zip"
  os.chdir('/content/drive/MyDrive')
  output_dir = os.path.join(os.getcwd(), "thesis_datasets")
  data_path = util.download_and_unzip(url, output_dir)
  print(f"Dataset downloaded here: {data_path}")

def load_data(data_path: str, data_split: str) -> Tuple[Dict, Dict, Dict]:
  corpus, queries, qrels = GenericDataLoader(data_path).load(split=data_split)
  print(f"Total Size of Corpus: {len(corpus)}")
  print(f"Total Queries: {len(queries)}")
  print(f"Total Query Document Relevance Judgements: {len(qrels)}")
  return corpus, queries, qrels
```

Benchmark Model performance for Retrieval

```
def evaluate_retrieval_models(data_paths: List[str], model_list : List[str], batch_size
    : int = 256, data_split: str = "test", similarity_function: str = "cos_sim"):
  retrieval_models_df = pd.DataFrame(
        columns=[
            "model_name",
            "batch_size",
            "data_set",
            "corpus_size",
            "queries",
            "query_relevance_judgements",
            "distance_function",
            "data_split_used"

        ]
    )

  model_results_df = pd.DataFrame(columns= [
      "model_name",
      "recall@1",
      "recall@3",
      "recall@5",
      "recall@10",
      "recall@100",
      "mrr@1",
      "mrr@3",
      "mrr@5",
      "mrr@10",
      "mrr@100"
      ]
  )

  for data_path in data_paths:
    dataset_name = data_path.split("/")[-1]
    corpus, queries, qrels = load_data(data_path = data_path, data_split = data_split)
    for model_name in model_list:
      print(f"Starting run for {model_name} on {dataset_name} dataset")
      model = DRES(models.SentenceBERT(model_name), batch_size = batch_size)
      retriever = EvaluateRetrieval(model, score_function = similarity_function)
      results = retriever.retrieve(corpus, queries) # format of results is identical to
          qrels

      # Evaluation Metrics
      ndcg, _map, recall, precision = retriever.evaluate(qrels, results, retriever.
          k_values)
      mrr = retriever.evaluate_custom(qrels, results, retriever.k_values, metric = "mrr
          ")
      print(f"Finished run for {model_name} on {dataset_name} dataset")
      print('*' * 60)
      model_results_df = model_results_df.append(
          {
              "model_name": model_name,
              "recall@1": f"{recall['Recall@1']:.3f}",
              "recall@3": f"{recall['Recall@3']:.3f}",
```

```
                "recall@5": f"{recall['Recall@5']:.3f}",
                "recall@10": f"{recall['Recall@10']:.3f}",
                "recall@100": f"{recall['Recall@100']:.3f}",
                "mrr@1": f"{mrr['MRR@1']:.3f}",
                "mrr@3": f"{mrr['MRR@3']:.3f}",
                "mrr@5": f"{mrr['MRR@5']:.3f}",
                "mrr@10": f"{mrr['MRR@10']:.3f}",
                "mrr@100": f"{mrr['MRR@100']:.3f}"
            },
            ignore_index=True
        )

        retrieval_models_df = retrieval_models_df.append(
            {
              "model_name": model_name,
              "batch_size": batch_size,
              "data_set": dataset_name,
              "corpus_size": len(corpus),
              "queries": len(queries),
              "query_relevance_judgements": len(qrels),
              "distance_function": similarity_function,
              "data_split_used": data_split
            },
            ignore_index=True
      )
    model_results_df.to_csv("model_results.csv", index=False)
    retrieval_models_df.to_csv("experiment_description.csv", index=False)
    print("Results saved")
    return [retrieval_models_df, model_results_df]
```

Evaluate Model Performance for Inference Speed

```
def evaluate_inference_speed( model_list: List[str], batch_size:int = 256, data_path: ↩
    str = "/content/drive/MyDrive/thesis_datasets/quora", data_split: str = "test"):
    inference_results_df = pd.DataFrame(
        columns=[
            "model_name",
            "batch_size",
            "data_set",
            "corpus_size",
            "queries",
            "distance_function",
            "data_split_used",
            "embedding_dimension",
            "index_size(MB)",
            "Avg_time_for_inference(s)",
            "k"
        ]
    )

    corpus, queries, qrels = corpus, queries, qrels = load_data(data_path = data_path, ↩
        data_split = data_split )
    corpus_ids, query_ids = list(corpus), list(queries)

    sample_docs = 100000
    sample_corpus = [corpus[corpus_id] for corpus_id in corpus_ids[:sample_docs]]

    for model_name in model_list:
      model = models.SentenceBERT(model_path=model_name)
      print("Extracting Embeddings...")
      embeddings = model.encode_corpus(sample_corpus, batch_size=batch_size, ↩
          convert_to_tensor=True, normalize_embeddings=True)

      time_taken_all = {}
      print(f"Measuring Time Taken for Inference with {model_name}")
      print("*" * 60)

      for query_id in query_ids:
          query = queries[query_id]
          start_time = time.perf_counter()
```

```
            query_embedding = model.encode_queries([query], batch_size=1, ↩
                convert_to_tensor=True, normalize_embeddings=True, show_progress_bar=↩
                False)
            similarity_scores = utils.cos_sim(query_embedding, embeddings)
            top_k, top_k_idx = torch.topk(similarity_scores, 10, dim=1, largest=True, ↩
                sorted=True)
            end_time = time.perf_counter()

            time_taken = (end_time - start_time)
            time_taken_all[query_id] = time_taken

        time_taken = list(time_taken_all.values())
        avg_time = sum(time_taken)/len(time_taken_all)

        embeddings = embeddings.cpu()
        cpu_memory = sys.getsizeof(np.asarray([emb.numpy() for emb in embeddings]))
        inference_results_df = inference_results_df.append(
            {
                "model_name": model_name,
                "batch_size": batch_size,
                "data_set": "quora",
                "corpus_size": sample_docs,
                "queries": len(query_ids),
                "distance_function": "cos_sim",
                "data_split_used": data_split,
                "embedding_dimension": len(embeddings[0]),
                "index_size(MB)": f"{cpu_memory*0.000001:.2f}",
                "Avg_time_for_inference(s)": f"{avg_time:.3f}",
                "k": 10
            },
            ignore_index = True
        )
    inference_results_df.to_csv("inference_results.csv", index=False)
    print("Results saved")
    return inference_results_df
```

C. Semantic Search Application Prototype

Imports

```
import gradio as gr

import itertools

import os

import pandas as pd
import pinecone

from sentence_transformers import SentenceTransformer, util

PINECONE_API_KEY = os.environ.get("PINECONE_API")
```

Data Cleaning

```
quora_dataset = load_dataset("quora", split="train")


quora_dataset = quora_dataset.filter(lambda x: x["is_duplicate"] is not True)
print("Number of rows after filtering duplicate question paris", len(quora_dataset))

quora_dataset_flat = quora_dataset.remove_columns("is_duplicate").flatten()
```

```
def create_seperate_id_cols(example):
  return {"id1": example["questions.id"][0],
          "id2": example["questions.id"][1]}

def create_seperate_question_cols(example):
  return {"question1": example["questions.text"][0],
          "question2": example["questions.text"][1]}

quora_dataset_updated = quora_dataset_flat.map(create_seperate_id_cols)
quora_dataset_updated = quora_dataset_updated.map(create_seperate_question_cols)

quora_dataset_updated = quora_dataset_updated.remove_columns(["questions.id", "↩
    questions.text"])
quora_dataset_updated.set_format("pandas")
quora_df = quora_dataset_updated[:]

id = pd.concat([quora_df['id1'], quora_df['id2']], ignore_index=True)
question = pd.concat([quora_df['question1'], quora_df['question2']], ignore_index=True)

quora_df_updated = pd.concat([id, question], ignore_index=True, axis=1)
quora_df_updated = quora_df_updated.rename(columns={0: 'id', 1: 'question'})
quora_df_updated = quora_df_updated.sort_values(by='id', ignore_index=True)
quora_df_updated = quora_df_updated.drop_duplicates(subset="id").reset_index(drop=True)
quora_df_updated.to_csv("/content/drive/My Drive/quora_df_updated.csv", index=False)
quora_dataset_updated = Dataset.from_pandas(quora_df_updated)

bad_indices = []

num_tokens = np.array(quora_tokens)
sus_indices = np.arange(len(quora_dataset_updated))[num_tokens == 237]
bad_indices.extend(sus_indices.tolist())


sus_indices = np.arange(len(quora_dataset_updated))[num_tokens == 0]
bad_indices.extend(sus_indices.tolist())

sus_indices = np.arange(len(quora_dataset_updated))[num_tokens == 1]
bad_indices.extend(sus_indices.tolist())

print(f'Number of outliers: {len(bad_indices)}.')

print(f'Original dataset has {len(quora_dataset_updated)} rows.')

quora_dataset_cleaned = quora_dataset_updated.filter(lambda x: len(x["question"].split↩
    ()) not in (0,1,237))

print(f'Cleaned dataset has {len(quora_dataset_cleaned)} rows.')
quora_dataset_cleaned.set_format("pandas")
quora_df_cleaned = quora_dataset_cleaned[:]
quora_df_cleaned.to_csv("/content/drive/My Drive/quora_df_cleaned.csv", index=False)
```

Indexing Data

```


INDEX_NAME, INDEX_DIMENSION  = "quora-semantic-search", 768
MODEL_NAME = "paraphrase-multilingual-mpnet-base-v2"

index = pinecone_connection(api_key = PINECONE_API_KEY, environment="us-east1-gcp", ↩
    index_name = INDEX_NAME,
                            distance_metric = "cosine", vector_dimension = ↩
                                INDEX_DIMENSION)

def encode_quora_data_and_upsert(model_name: str, dataset_path: str, NUM_SAMPLES: int =↩
     30000):
  quora = pd.read_csv(dataset_path)
  model = SentenceTransformer(model_name, device = "cpu")
  quora_small_dataset = quora[:NUM_SAMPLES].copy()

  print('Encoding questions...')
  encoded_questions = model.encode(quora_small_dataset['question'].tolist(), ↩
      show_progress_bar=True)
```

```
  quora_small_dataset['question_vector'] = encoded_questions.tolist()
  quora_small_dataset['id'] = quora_small_dataset['id'].apply(str)
  upsert_vectors = [
                (
                    row["id"],
                    row["question_vector"],
                    {
                        "question": row["question"],
                    },
                )
                for _, row in quora_small_dataset.iterrows()
            ]
  for data_chunk in chunks(upsert_vectors):
    index.upsert(vectors = data_chunk)
  print("Data Added to Pinecone. Semantic Search Enabled")
```

app.py

```
def display_df_as_table(result: tuple):
      df = pd.DataFrame(result, columns = ['id', 'question', 'score'])
      return df

def semantic_search(query: str = None, top_k: int = 3):
  if not query:
      return "Please enter a search query"
  vector_embedding = model.encode(query).tolist()
  response = index.query([vector_embedding], top_k = top_k, include_metadata=True)
  matches = response['matches']
  result = [(match['id'], match['metadata']['question'], round(match['score'],4)) for ↩
      match in matches]
  return result

title = """<h1 id="title">Search Similar Questions in Multiple Languages for Quora ↩
    Dataset</h1>"""

description = """
Semantic Search is a way to generate search results based on the actual meaning of the ↩
    query instead of a standard keyword search.
- The App generates embeddings using [paraphrase-multilingual-mpnet-base-v2]
(https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet
-base-v2) model from Sentence Transformers.
- The model encodes the query and the question field from the [Quora Dataset](https://↩
    huggingface.co/datasets/quora) dataset which contains question pairs.
- The Storage Backend for storing the embeddings is the Pinecone Vector Database [↩
    Pinecone](https://www.pinecone.io/)
- Similarity scores are generated, from highest to lowest using Cosine Similarity. The ↩
    user can select how many similar questions they need from the vector database.
- You will see 1 table generated, that displays the search results.
Enjoy and Search like you mean it!!
"""
css = '''
h1#title {
  text-align: center;
}
'''
demo = gr.Blocks(css=css)
with demo:
  gr.Markdown(title)
  gr.Markdown(description)
  top_k = gr.Slider(minimum=3,maximum=10,value=3,step=1,label='Number of Similar ↩
      Questions to Retreive')
  with gr.Row():
    query = gr.Textbox(placeholder = "Enter your search query here", label="Search")
  with gr.Row():
    output = gr.DataFrame(headers=['Question_id', 'Question', 'Similarity Score'], ↩
        label=f'Top-{top_k} Similar Questions to your query', wrap=True)
  semantic_search_btn = gr.Button("Search")
  semantic_search_btn.click(fn=semantic_search, inputs=[query, top_k], outputs=output)
  gr.Markdown("## Example Search Queries")
  gr.Examples(
        ["How can I learn Python online?",
```

```
        "Wie kann ich Python online lernen?",
        query,
        output,
        semantic_search,
        cache_examples=True
    )

demo.launch(debug=True, enable_queue=True)
```