University of Macedonia

Computer Science and Technology

Department of Applied Informatics

Postgraduate Thesis

Full-text indexing in open source DBMS

Michail Chatziparaskevas

Supervisor:

Georgios Evangelidis
Professor

Thessaloniki, June, 2023

Approved by the selection board on 20/6/2023.

Georgios Evangelidis	Koloniari Georgia	Karakasidis Alexandros
Professor	Assistant Professor	Laboratory Teaching Personnel

Michail Chatziparaskevas
Department of Applied Informatics
University of Macedonia

Copyright ©Michail Chatziparaskevas, 2023
All rights reserved.

Copying is prohibited, storage and distribution of the present work, wholly or in part, for commercial purposes. Reprint is allowed, storage and distribution for non-profit purposes, educational or research nature, provided that the source of origin is indicated and to keep this message. Questions concerning the use of work for profit-making purposes should be addressed to the author. The views and conclusions contained in this document express the author and should not be interpreted as representing the official positions of the University of Macedonia.

Abstract

The main objective of this diploma thesis is to analyze and present the features of various open-source SQL engines that perform text indexing for specific documents. We will explain the concept of text indexing and the fields used to enhance efficiency and reduce search time. Additionally, we will compare and enhance the performance of different Database Management Systems (DBMS) used for storing, processing, and searching large amounts of rapidly growing data.

However, collecting and storing massive amounts of data alone does not address the issue of proper data organization and classification. To enable faster query processing and quick access to data records, indexing is necessary. It is crucial to carefully design indexes as they can occupy significant disk space. Hence, selecting the most suitable database management system and effectively managing the database are important considerations.

For this diploma thesis, we will utilize MySQL, PostgreSQL, and SQLite as the database management systems. By using these systems, we aim to gain a comprehensive understanding of their construction and how text indexing can be implemented for specific documents. We will follow a consistent approach for all engines, covering fundamental text searching commands, text searching characteristics, auxiliary functions, and the creation of text indexing tables.

Once we have gathered all the necessary data, we will conduct data visualization analysis to examine the response time of each system. This analysis will enable us to compare different functions not only within the same system but also across different SQL engines.

Keywords: open-source SQL engines, text indexing, DBMS, indexing, MySQL, PostgreSQL, SQLite, text searching commands, text searching characteristics, auxiliary functions
english

Contents

Abstract i

Contents iv

1 Intro 1
- 1.1 Text Indexing - Preface . . . 1
- 1.2 Full-text searching advantages . . . 2
- 1.3 Overview of Full-Text Searching in SQLite, PostgreSQL and MySQL . . . 3
- 1.4 Comparing traditional search techniques with full text searching for Handling of word variations and misspellings . . . 5
- 1.5 Disadvantages of full text searching . . . 6

2 PostgreSQL - Full text Indexing 9
- 2.1 Basic Full-text searching functionality . . . 10
- 2.2 Auxiliary functions . . . 11
- 2.3 Optimizing ts_rank function . . . 13
- 2.4 Dictionaries . . . 15
- 2.5 Full-text indexing in PostgreSQL . . . 16
  - 2.5.1 Example - GIN & GiST . . . 18
  - 2.5.2 Advantages and Disadvantages - Gist vs GIN . . . 19

3 MySQL - Full text Indexing 23
- 3.1 Types of full text searching in MySQL . . . 23
- 3.2 Natural Language Full-Text Search . . . 25
  - 3.2.1 Examples . . . 25
  - 3.2.2 Limitations . . . 26
  - 3.2.3 Rank Function . . . 28
- 3.3 Boolean Mode . . . 28
  - 3.3.1 Examples . . . 29
  - 3.3.2 How MySQL calculate the ranking . . . 31
  - 3.3.3 Limitations . . . 32
- 3.4 Query Expansion Mode . . . 32
  - 3.4.1 Optimize Natural Language Mode using Query expansion . . . 33
- 3.5 Full text Indexing in My SQL . . . 33
  - 3.5.1 InnoDB and MyISAM . . . 34
  - 3.5.2 Full Text Indexing in MySQL . . . 35

4 SQLite - Full text Indexting 37
- 4.1 Basic Full text searching . . . 38
- 4.2 Advanced Full text searching . . . 39

4.2.1 FTS5 Prefix Queries 39
4.2.2 FTS5 Initial Token Queries 40
4.2.3 FTS5 NEAR Queries 41
4.2.4 FTS5 Column Filters 41
4.3 Auxiliary Functions 42
4.4 Full text indexing in SQLite 45

5 Comparison Analysis - Word Phrase Searching **47**
5.1 Methodology 47
5.1.1 Description of the dataset 48
5.1.2 Overview of full-text searching 49
5.1.3 F1 Score methodology 50
5.2 Intro of the analysis 50
5.2.1 Evaluating the search performance of three different systems - PostgreSQL, MySQL, and SQLite 52
5.2.2 Experimental Results 56
5.2.3 Comparison of word and phrase searching capabilities with the enhancements 58
5.2.4 Full text searching for synonyms 60
5.3 Conclusion 61

6 Conclusion **63**

Chapter 1

Intro

In the field of database management, full-text searching and indexing have become essential tools for efficient data retrieval. Databases like SQLite, PostgreSQL, and MySQL now offer full-text indexing and searching capabilities, allowing users to search for keywords and phrases within large volumes of data. Full-text indexing is a process that creates an index of the words within a document or database, allowing for fast and accurate searching. This index can then be used to perform full-text searches, which can be more powerful and precise than traditional keyword-based searches. With the growth of data and the need for faster, more accurate search capabilities, full-text indexing and searching have become increasingly important in many industries. From e-commerce to healthcare to finance, organizations rely on these tools to quickly find the information they need to make informed decisions and gain a competitive edge. In this context, understanding the capabilities and limitations of databases like SQLite, PostgreSQL, and MySQL, as well as full-text indexing and searching techniques, has become essential for effective data management and analysis.

1.1 Text Indexing - Preface

As the volume of digital information available to us continues to increase rapidly, the necessity for efficient and accurate search and retrieval of that data has become more urgent [3]. Conventional search methods, such as the **LIKE** operator, have limitations in their capacity to handle vast amounts of text data and may be sluggish and inefficient when dealing with complex queries [6].

Although SQL search functions like LIKE are helpful for basic pattern matching, such as locating all records where a field contains a particular string of characters or begins with a certain letter, they may be relatively slow for large datasets, as they require a full table scan to match records. Moreover, LIKE searches rely on exact matches to a given pattern and do not consider variables like synonyms or the proximity between search terms [14]. This is where full-text searching comes into play, providing a range of capabilities that make it a valuable tool for contemporary databases.

In full-text searching, techniques like compression and full-text indexing are utilized to enhance search efficiency and enable more comprehensive retrieval of information from digital libraries [11]. Full-text searching goes beyond simple pattern matching and incorporates features such as synonym detection and proximity analysis, which contribute to improving search accuracy and relevance [1]. These capabilities make full-text searching an essential component in dealing with the growing volume of digital information and the complexities of modern databases.

Full-text searching offers a significant advantage over traditional search methods by allowing users to enter natural language queries, making it easier to find the desired information without the need for complex syntax or Boolean logic. This is especially useful for e-commerce websites where users can search for products using everyday language, such as ”red shoes for women”.

This not only streamlines the search process for a wider audience but also enhances the overall user experience, resulting in higher customer satisfaction and retention [3].

Moreover, full-text searching incorporates relevance ranking algorithms to prioritize search results based on their relevance to the user's query. This increases the likelihood of users finding the required information quickly and efficiently, making the search process more effective overall [11]. Additionally, full-text searching offers extensive customization options such as search filters, allowing users to specify search parameters, such as particular phrases, or exclude specific words or characters [14]. Users can also search within specific fields or columns, further enhancing the precision of search results and offering more relevant data.

Furthermore, by leveraging the capabilities of full-text searching, organizations can develop more sophisticated and powerful data-driven applications that improve business intelligence, customer experience, and decision-making. By extracting insights from vast amounts of text data, organizations can gain a deeper understanding of customer needs, market trends, and competitive landscapes. This deeper understanding can lead to more informed business strategies, increased revenue, and improved customer satisfaction.

In conclusion, the significance of full text searching in contemporary databases cannot be overstated. Its ability to handle natural language queries, incorporate relevance ranking algorithms, and offer customizable search options and filters makes it a valuable tool for businesses and organizations across a wide range of industries. As digital information continues to grow in both volume and complexity, the importance of efficient and precise search and retrieval will only increase, making full text searching an essential capability for modern databases. Full text searching is a game-changer for modern databases, offering unparalleled search capabilities that can help businesses and organizations unlock the full potential of their data. With its ability to handle natural language queries, incorporate relevance ranking algorithms, offer customizable search options and filters, and integrate with other database functionalities, full text searching is a must-have for any organization that deals with large volumes of text data.

1.2 Full-text searching advantages

In today's data-driven world, organizations are collecting and storing vast amounts of information from a variety of sources [11]. However, the sheer volume of data can make it difficult to extract meaningful insights and valuable information. This is where full-text searching becomes an important tool. Full-text search engines use complex algorithms that allow users to search through large volumes of data and quickly find the information they need [3]. With the ability to search for keywords, phrases, and even concepts, full-text searching has become a critical tool for businesses and organizations across a range of industries, including healthcare, finance, e-commerce, and more. By providing fast, accurate, and relevant search results, full-text searching helps organizations make better decisions, improve productivity, and gain a competitive edge [11]. The ability to quickly and efficiently search through vast amounts of data enables organizations to uncover valuable insights and extract meaningful information. This, in turn, supports informed decision-making processes, allowing organizations to respond effectively to market trends and customer needs [22]. Additionally, full-text searching aids in improving productivity by reducing the time and effort required to find relevant information [11]. With faster access to information, employees can focus on analyzing data and deriving actionable insights.

Saying that, please find the advantages that Full-text searching provides in the below list:

1. **Improved search accuracy**: Full-text search algorithms can handle complex queries with multiple keywords and search terms. It can also search for synonyms and related words, which can help to improve the accuracy of search results.

2. **Faster searching**: Full-text search engines are designed to quickly search through large volumes of data. They use indexing techniques to store information about the content and location of words in documents, which makes searching faster and more efficient.

3. **Increased productivity**: Allows users to quickly find relevant information within large documents, reducing the time needed to manually search through documents.

4. **Increased precision**: Provide more precise results than traditional keyword-based searches. This is because full-text search algorithms can take into account the proximity of words and the context in which they appear, which can help to filter out irrelevant results.

5. **More flexibility**: Full-text searching can be used to search a wide range of content types, including text-based documents, multimedia files, and structured data. This makes it a versatile tool for many different types of applications

6. **Customizable search options**: Search engines can be customized to meet specific search requirements, including filters, ranking criteria, and search scopes. This allows for more tailored and accurate search results.

7. **Improved user experience**: Full-text search can provide a more user-friendly experience by returning more relevant results and allowing users to quickly find the information they need.

8. **Scalability**: Full-text search engines can handle large volumes of data and are designed to scale with growing data volumes, making them a good choice for applications that require high-performance search capabilities.

In conclusion, full-text searching is a powerful tool that offers numerous advantages over traditional keyword-based searching. By analyzing the content of documents and identifying relevant keywords, full-text searching enables users to find the information they need quickly and easily. It is particularly useful for searching large databases and collections of documents, as it allows users to narrow down search results to only those that are relevant. Additionally, it can help users discover new information by suggesting related topics and providing additional context. With the growing volume of data available online, full-text searching is becoming an increasingly important tool for businesses, researchers, and individuals alike.

1.3 Overview of Full-Text Searching in SQLite, PostgreSQL and MySQL

Full-text searching is a powerful and essential feature offered by many SQL database management systems, enabling efficient searching within large text-based datasets [18]. It proves especially valuable for handling unstructured or semi-structured data like articles, documents, and web pages [14]. By leveraging full-text searching, applications can swiftly search through vast amounts of textual information, delivering faster and more accurate search results.

To employ full-text searching in SQL, a crucial step is creating a full-text index on the relevant columns [22]. This index is a specialized data structure optimized for rapid text searching, designed to store and retrieve textual data efficiently. Once the full-text index is established, SQL operators can be utilized to perform searches on the indexed columns, allowing for complex queries and retrieval of relevant information.

The full-text index enhances the search capabilities of SQL databases by enabling features like stemming, proximity matching, and relevance ranking [11]. Stemming involves recognizing and indexing word variants, allowing for more comprehensive searches [1]. Proximity matching takes into account the proximity of search terms, enabling more precise and context-aware searches [11].

Relevance ranking algorithms prioritize search results based on their relevance to the query, further enhancing the accuracy of search results [11].

By incorporating full-text searching in SQL databases, applications can efficiently handle large text-based datasets and provide users with faster and more accurate search functionality. This, in turn, improves the overall performance and usability of data-driven applications that rely on textual data

From an open database management systems (DBMS) perspective, prominent options like SQLite, PostgreSQL, and MySQL offer robust support for full-text searching. Integrating full-text searching capabilities into applications can significantly enhance search functionalities, enabling users to efficiently navigate and retrieve information from large volumes of text-based data.

Each of these DBMS platforms implements full-text searching with its own unique set of features and capabilities. In SQLite, full-text searching is accomplished using a virtual table module called FTS (Full-Text Search) [3]. PostgreSQL utilizes the tsvector and tsquery modules, which provide advanced full-text search capabilities, including language-aware search and ranking [1]. MySQL incorporates a built-in full-text search engine, offering features such as relevance ranking and Boolean search queries [13].

These full-text search implementations in SQLite, PostgreSQL, and MySQL enable developers to leverage powerful search capabilities within their applications. By utilizing the specific functionalities provided by each DBMS, developers can tailor the full-text search experience to suit the requirements of their application and optimize search performance.

Incorporating full-text searching in applications powered by SQLite, PostgreSQL, or MySQL allows users to efficiently search and retrieve information from large text-based datasets, enhancing the overall usability and effectiveness of the applications

Here's a brief overview of full-text searching in SQLite, PostgreSQL, and MySQL:

1. **Full-Text Searching in SQLite**:

   SQLite, a self-contained and lightweight database management system, boasts an impressive full-text searching feature. This feature is enabled by the FTS (Full-Text Search) virtual table module that creates full-text search indexes on table columns. The FTS module is equipped with advanced functionalities such as tokenization, stemming, and stop-word removal, making it even more powerful.

   To utilize full-text searching in SQLite, you must create an FTS virtual table that contains the columns to be searched. Subsequently, you can insert data into the table and generate a full-text search index using the FTS module. Once the index is created, you can execute full-text searches using the MATCH operator in SQL.

   ```sql
   SELECT * FROM articles_fts WHERE articles_fts MATCH 'database';
   ```

2. **Full-Text Searching in PostgreSQL**:

   PostgreSQL is a widely-used open-source relational database management system with support for full-text searching. It comes with a built-in full-text search engine called tsvector and tsquery that enable you to create full-text search indexes on columns within a table. The tsvector and tsquery modules offer a range of advanced capabilities, such as stemming, synonymy, and stop-word removal, which can improve the accuracy and relevance of search results.

   To use full-text searching in PostgreSQL, you must first create a tsvector column that includes the columns you want to search. After that, you can add data to the table and generate a full-text search index utilizing the tsvector and tsquery modules. Once the index is established, you can conduct full-text searches using the @@ operator in SQL.

```
SELECT * FROM documents WHERE to_tsvector('english', title || ' ' ||
    content) @@ to_tsquery('english', 'database');
```

3. **Full-Text Searching in MySQL**:

   MySQL is a renowned relational database management system that facilitates full-text searching. The system comes equipped with a built-in full-text search engine capable of creating full-text search indexes on columns in a table. It allows various features, such as natural language mode, Boolean mode, and phrase searching, to enhance the search experience.

   To utilize full-text searching in MySQL, you need to first create a full-text search index on the columns you intend to search using the FULLTEXT index type. Once the index is created, you can conduct full-text searches using the MATCH operator in SQL. Additionally, MySQL offers a range of functions to manipulate and analyze the search results, such as the MATCH() function and the AGAINST() function.

```
SELECT * FROM articles WHERE MATCH (title, content) AGAINST ('database');
```

Each of the full-text search solutions for MySQL, SQLite, and PostgreSQL has its own strengths and weaknesses. MyISAM in MySQL provides basic full-text search capabilities but lacks support for advanced features and may not be the best option for high-performance applications. FTS5 in SQLite is a lightweight and flexible solution but does not provide relevance ranking out of the box. tsvector in PostgreSQL is a robust and scalable solution with support for multiple languages and advanced text processing but can be resource-intensive. Ultimately, the best solution will depend on your specific requirements and use case.

In the next sections, we will explain in more detail the advantages and disadvantages of each full-text search solution for MySQL, SQLite, and PostgreSQL providing all the capabilities.

1.4 Comparing traditional search techniques with full text searching for Handling of word variations and misspellings

The ability of modern databases to incorporate full-text searching brings a significant advantage in managing word variations and misspellings. Unlike conventional search functions such as LIKE, which only match exact search terms, full-text search engines employ techniques like stemming and fuzzy matching to address this challenge [3].

Stemming is a method employed by full-text search engines to reduce words to their root form or stem [1]. This enables the search engine to match variations of the word. For example, a search for ”run” could also match ”running” and ”runner” since they share the same stem. Stemming algorithms can be language-specific and handle common inflections and conjugations, ensuring a wider range of relevant search results [1].

In addition to stemming, full-text searching utilizes fuzzy matching techniques to accommodate misspellings and variations in word order. Fuzzy matching assigns a similarity score to potential matches based on how closely they resemble the search term. This allows the search engine to provide results even if the search term is misspelled or if the word order is slightly different. Fuzzy matching enhances the search experience by considering potential matches that may have slight deviations from the original search term [11].

By incorporating stemming and fuzzy matching techniques, full-text search engines overcome the limitations of exact matching, enabling more flexible and comprehensive searches [3]. These techniques ensure that users can find relevant information even when they enter variations, misspellings, or slightly different word forms. Consequently, the accuracy and usability of the search

functionality are improved, providing a better experience for users in locating the desired information [11].

The combination of stemming and fuzzy matching in full-text searching not only addresses common challenges such as word variations and misspellings but also enhances the search capabilities of applications and databases. Users can rely on the search engine to deliver accurate and relevant results, even when the input deviates from the exact match.

Let's assume that we have a database of restaurant reviews and we want to search for all reviews containing the term "delicious." If we employ a conventional search function like **LIKE**, our query might look like this:

```
SELECT * FROM reviews WHERE text LIKE '%delicious%';
```

This query will only identify reviews that contain the precise word "delicious." Nevertheless, it will not identify variations of the term, like "delish," or misspellings such as "delicous."

The above query demonstrates how full-text searching can be used in SQLite to find restaurant reviews containing the word "delicious" and its variations. By using the MATCH operator and the NEAR operator for word proximity, the query can find reviews containing both "delicious" and "delish" within a range of 5 words. Additionally, the full-text search engine's stemming algorithm can identify variations of "delicious" such as "delightful" or "delicacy", allowing for a more comprehensive search.

```
SELECT * FROM reviews WHERE text MATCH 'delicious NEAR/5 delish';
```

Traditional search techniques, such as SQL LIKE and regular expressions, match keywords or phrases exactly as they appear in the database. While these techniques can be effective for finding exact matches, they are less effective at handling variations in spelling or phrasing [6]. In contrast, full-text search techniques, such as those provided by MyISAM in MySQL, FTS5 in SQLite, and tsvector in PostgreSQL, use advanced algorithms to handle variations in spelling and phrasing [22].

These full-text search techniques can also take into account synonyms, stemming, and other linguistic nuances to improve the accuracy of search results. By leveraging these advanced algorithms, full-text search provides a more robust solution for handling variations and linguistic complexities in search queries. By understanding the advantages and disadvantages of each approach, you can choose the best search technique for your specific needs. While traditional search techniques can be effective for exact matches, full-text search is often more powerful when it comes to handling variations in spelling and phrasing [1].

To sum up, traditional search methods like simple string searches, pattern matching, and regular expressions may not be effective in dealing with word variations and spelling errors in vast amounts of text-based data . On the contrary, full-text searching is a potent tool that can efficiently search such data by building search indexes that include advanced features like stemming, synonym recognition, and stop-word elimination.

1.5 Disadvantages of full text searching

Full-text searching is a potent technology that allows users to search large volumes of text data in a fast and efficient manner [11]. It is a substitute for the traditional LIKE function in SQL and other database systems, offering more advanced search capabilities that can manage word variations, spelling mistakes, and intricate natural language queries.

One of the potential challenges is the complexity of implementing and maintaining full-text search functionality. It requires careful consideration of indexing strategies, search algorithms, and query optimization techniques [11]. Furthermore, as the volume of data grows, the speed of full-text search queries can decrease, impacting the overall performance of the system.

Another consideration is the precision of search results. Full-text search engines, while powerful, may occasionally produce false positives or false negatives, impacting the accuracy of the search results. This can be particularly problematic when dealing with critical or sensitive information.

Language support is another aspect to consider. Full-text search engines may vary in their support for different languages, and certain linguistic nuances may not be adequately handled [1]. It's crucial to assess the language-specific capabilities and limitations of the chosen full-text search engine.

Maintenance is an ongoing concern when utilizing full-text search. The indexes used for efficient searching need to be regularly updated as new data is added or modified. Additionally, as the system scales and the data grows, the management and optimization of the full-text search functionality become more complex.

Finally, the learning curve associated with full-text searching should be taken into account. Developers and administrators need to familiarize themselves with the specific syntax, APIs, and configuration options provided by the chosen full-text search engine. This may require additional training or resources to effectively utilize and maintain the full-text search functionality.

By comprehending these potential challenges, organizations can make well-informed decisions about when and how to employ full-text searching and how to enhance its performance to meet their particular requirements [17].

1. **Complexity**: Full-text searching is a complex technology that requires a lot of processing power and storage space to create and maintain indexes. This can be challenging for smaller organizations or for applications with limited resources.

2. **Speed**: Although full-text searching is generally faster than the **LIKE** function, it can still be slower than exact-match searches, especially for large datasets. Searches that require complex natural language processing or ranking algorithms can also be slower.

3. **Accuracy**: Full-text searching is not always accurate, especially when dealing with variations of words, misspellings, or ambiguous phrases. While techniques like stemming and synonym expansion can improve accuracy, they can also produce false positives or miss important results.

4. **Language Support**: Full-text searching is more effective in some languages than others. For example, languages with complex morphology or non-Latin alphabets may require more sophisticated indexing and processing techniques to produce accurate results.

5. **Maintenance**: Full-text search indexes need to be updated regularly to ensure they are up-to-date and accurate. This can be time-consuming and resource-intensive, especially for large databases or frequently changing content.

6. **Learning Curve**: Full-text searching requires some level of expertise in database management and search technology. Organizations may need to invest in training or hire specialized personnel to maintain and optimize full-text search capabilities.

Apart of the above, In full-text searching, several factors can hinder effective information retrieval. Synonyms and variant spellings of words are two such factors that can lead to incomplete retrieval of information. While searching for a particular term, a user might miss out on documents that contain synonyms or variant spellings of the searched term. Similarly, shortened forms of terms like abbreviations and acronyms can also result in incomplete retrieval of information. In addition, searching for a term in one language might not match documents that contain the foreign-language version of that concept[5]. Homonyms, false cognates, and word lists are other factors that can lead to low search precision or irrelevant hits in full-text searches. Name disambiguation is

another problem that occurs when the practice of making each person's name unique in a database is not employed. As a result, name disambiguation is significant in academic libraries because some style guides prescribe the use of initials instead of given names in citations, making a full-text search for an author's name more difficult.

Although full-text searching may have some drawbacks, the advantages it offers usually surpass the limitations, especially for applications with extensive amounts of text data and intricate search requirements. By delivering robust search capabilities, full-text searching can assist users in finding the information they need rapidly and effectively, which can lead to improved decision-making and increased productivity.

Chapter 2

PostgreSQL - Full text Indexing

PostgreSQL is a popular open-source relational database management system that provides support for full-text search. Full-text search is a powerful search capability that allows users to search for words or phrases within a large collection of text documents [1]. PostgreSQL provides a set of functions that enable developers to create flexible and efficient full-text search capabilities in their applications.

The ”to_tsvector” function is used to convert text data into a vector of tokens that can be indexed for full-text searching. The ”to_tsquery” function is used to convert a search query into a vector of tokens that can be used to search the full-text index. The ”@@” operator is used to search the full-text index for matches to a specific search query [1].

To ensure that full-text search in PostgreSQL is optimized for performance, developers must use appropriate data types, optimize the index, and tune the search parameters to balance accuracy and speed. The ”ts_vector” data type represents a document as a sorted list of unique words or lexemes, and the ”to_tsvector” function generates the ts_vector by taking a text document as input. The ts_vector contains information about the position and frequency of each lexeme within the document, as well as its normalization and dictionary lookup status.

```sql
SELECT * FROM documents WHERE content_vector @@ to_tsquery('english',
    'database');
```

The ts_query is a data type that represents a search query as a group of lexemes or words that have been processed and normalized for use in full-text search. The to_tsquery() function generates the ts_query by taking a search query string as input. The ts_query can comprise boolean operators, phrase queries, negation, and prefix operators, providing the ability to conduct intricate and adaptable searches.

```sql
SELECT title, content
FROM articles
WHERE to_tsvector('english', title || ' ' || content) @@ to_tsquery('english',
    'cat & dog');
```

The combination of ts_vector and ts_query in PostgreSQL provides a potent set of tools for searching and examining text data, allowing for advanced functionality such as phrase matching, prefix matching, negation, fuzzy matching, weighting, and ranking. These features make PostgreSQL's full-text search capabilities ideal for a diverse range of applications, including search engines, content management systems, and data analytics platforms

2.1 Basic Full-text searching functionality

PostgreSQL, as a powerful relational database management system, provides developers with a comprehensive set of data types and functions specifically designed to enable flexible and accurate full-text searches [15]. One such data type is the to_tsquery, which offers a wide range of capabilities for refining and customizing searches.

By representing a query string as a processed and normalized set of lexemes, the to_tsquery data type empowers developers to perform intricate searches that consider factors such as word order, proximity, and term frequency. This enables the construction of complex search queries that accurately capture the intent of the user [16]. Furthermore, developers can leverage various operators and functions in combination with to_tsquery to refine and fine-tune the search results [10] [16].

Boolean operators, such as AND, OR, and NOT, allow developers to combine search terms in different ways, providing flexibility in constructing complex search conditions. Additionally, phrase queries enable the search for specific phrases within a document, ensuring precise retrieval of relevant information [16]. The availability of negation and prefix operators further enhances the search capabilities by allowing developers to exclude or specifically search for certain terms.

In addition to these advanced querying options, the to_tsquery data type in PostgreSQL also supports ranking and weighting mechanisms [10]. Developers can adjust the relevance and ranking of search results based on factors like proximity to search terms or other criteria. This functionality ensures that the most pertinent search results are presented to the user, improving the overall search experience.

By offering a rich set of data types and functions for full-text searching, PostgreSQL enables developers to build sophisticated search functionalities that cater to diverse requirements. The to_tsquery data type, along with its associated operators and functions, empowers developers to perform accurate and customized searches, ultimately enhancing the precision, relevance, and usability of search results within PostgreSQL-based applications [10] [15] [16].

Some of the key functionalities provided by to_tsquery in PostgreSQL include:

- **Term matching**: ts_query can be used to match one or more terms in a ts_vector, using the 'AND' or 'OR' operators. For example:

  ```
  SELECT * FROM articles
  WHERE to_tsvector('english', title || ' ' || body) @@
      to_tsquery('english', 'search OR engine');
  ```

  This will return all articles containing either the term 'search' or the term 'engine', or both.

- **Phrase matching**: ts_query can be used to match a phrase or combination of terms in a ts_vector, using the '<->' operator. For example:

  ```
  SELECT * FROM articles
  WHERE to_tsvector('english', title || ' ' || body) @@
      to_tsquery('english', 'search <-> engine');
  ```

  This will return all articles containing the terms 'search' and 'engine' in close proximity to each other, such as 'full text search engine' or 'search and retrieval engine'.

- **Negation**: ts_query can be used to exclude terms or phrases from a ts_vector, using the 'NOT' operator. For example:

  ```
  SELECT * FROM articles
  WHERE to_tsvector('english', title || ' ' || body) @@
      to_tsquery('english', 'search AND NOT engine');
  ```

This will return all articles containing the term 'search' but not the term 'engine', such as 'search algorithm' or 'search techniques'.

- **Prefix matching**: ts_query can be used to match terms with a common prefix in a ts_vector, using the ':*' operator. For example:

```
SELECT * FROM articles
WHERE to_tsvector('english', title || ' ' || body) @@
    to_tsquery('english', 'informat:*');
```

  This will return all articles containing terms starting with the prefix 'informat', such as 'information retrieval' or 'informatics research'.

- **Fuzzy matching**: ts_query can be used to match terms with variations or misspellings, using the 'SIMILAR TO' operator. For example:

```
SELECT * FROM articles
WHERE to_tsvector('english', title || ' ' || body) @@
    to_tsquery('english', 'colour SIMILAR TO "color%"');
```

  This will return all articles containing the term 'colour' or any term starting with the prefix 'color', such as 'colorful' or 'colorization'.

- **Weighting**: ts_query can be used to assign weights to individual terms or phrases in a ts_vector, using the 'A' (for 'A-weight') and 'B' (for 'B-weight') modifiers. For example:

```
SELECT * FROM articles
WHERE to_tsvector('english', title || ' ' || body) @@
    to_tsquery('english', 'search:A & engine:B');
```

  This will return all articles containing both the term 'search' with an 'A' weight and the term 'engine' with a 'B' weight. This allows for more fine-grained control over search results, as certain terms or phrases can be given more or less weight depending on their importance or relevance to the search.

2.2 Auxiliary functions

In addition to the fundamental set of functions used for full-text search in PostgreSQL, the system provides a rich set of auxiliary functions that allow for further customization and optimization of search capabilities within applications.

One crucial set of auxiliary functions in PostgreSQL is the text search configuration functions. These functions empower end users to customize the behavior of full-text search by defining specific rules for text processing and indexing [10]. By utilizing these functions to create custom text search configurations, developers can tailor the search functionality to suit the specific requirements of their application. This includes support for different languages, character encodings, and text normalization techniques. With the ability to customize text search configurations, developers can enhance the accuracy and relevancy of search results, ensuring that the search functionality aligns with the unique needs of their application [10].

Another valuable set of auxiliary functions for full-text search in PostgreSQL is the ranking functions. These functions enable the assignment of relevance scores to search results based on various factors such as keyword frequency, proximity, and context [16]. By employing ranking functions, developers can fine-tune the search results and provide users with more accurate and relevant search outcomes. This optimization enhances the overall usability and effectiveness of

the application's search functionality, ensuring that users can quickly find the most relevant information.

PostgreSQL also offers a range of functions for fuzzy text matching. These functions are particularly useful in scenarios where users may not have precise knowledge of the spelling or wording of the text they are searching for. Fuzzy text matching functions can identify matches that are similar to, but not exact matches to the search query. By incorporating fuzzy text matching, developers can improve the search experience and accommodate variations, misspellings, or alternative phrasings, making it easier for users to find the desired information[2].

With the availability of these auxiliary functions, PostgreSQL provides developers with powerful tools to further customize and optimize full-text search capabilities within their applications. By leveraging text search configuration functions, ranking functions, and fuzzy text matching functions, developers can refine the search functionality to deliver more precise, relevant, and user-friendly search experiences

Please find below some of the most important functions in PostgreSQL

- **ts_headline()**: This function is used to highlight the matching words in the search results. It takes a text input, a tsvector column, and a tsquery object, and returns a text output with the matching words highlighted.

```
SELECT ts_headline('english', 'The quick brown fox jumped over the lazy
    dog',
to_tsquery('english', 'quick & (brown | fox)')) AS highlighted_text;

Output:
The <b>quick</b> <b>brown</b> <b>fox</b> jumped over the lazy dog
```

- **plainto_tsquery()**: This function is used to create a tsquery object from a plain text input. It tokenizes the input text based on the selected text search configuration, and returns a tsquery object.

```
SELECT plainto_tsquery('english', 'quick brown fox');

Output:
'quick' & 'brown' & 'fox'
```

- **phraseto_tsquery()**: This function is used to create a tsquery object from a phrase input. It tokenizes the input text based on the selected text search configuration, and returns a tsquery object that matches the entire phrase.

```
SELECT plainto_tsquery('english', 'quick brown fox');

Output:
'quick' & 'brown' & 'fox'
```

- **websearch_to_tsquery()**: This function is used to create a tsquery object from a web search input. It tokenizes the input text based on the selected text search configuration, and returns a tsquery object that matches the input text as a phrase or individual words.

```
SELECT websearch_to_tsquery('english', 'quick brown fox');

Output:
'quick' & 'brown' & 'fox'
```

- **ts_rewrite()**: This function is used to rewrite a tsquery object using synonym and/or thesaurus dictionaries, to expand the search scope.

```
SELECT ts_rewrite(to_tsquery('english', 'quick & (brown | fox)'),
'english_synonym') AS expanded_query;

Output:
'quick' & ('brown' | 'fox' | 'express')
```

- **ts_lexize()**: This function is used to apply a dictionary rule to a lexeme, to normalize or transform the lexeme for search purposes.

```
SELECT ts_lexize('english_stem', 'jumping');

Output:
{jump}
```

- **ts_filter()**: This function is used to apply a filtering rule to a lexeme, to exclude or include certain words based on specific criteria.

```
SELECT ts_filter('english_stop', 'the');

Output:
null
```

Overall, auxiliary functions provide developers with a powerful set of tools for customizing and optimizing the full-text search capabilities of their PostgreSQL applications. By leveraging these functions, developers can create search functionality that is tailored to the specific needs of their users, providing a more effective and satisfying user experience.

2.3 Optimizing ts_rank function

In PostgreSQL, the ts_rank function is a powerful tool for improving the relevance of search results in full-text search applications. This function calculates a relevance ranking for each search result based on the proximity and frequency of the search terms within the text.

The basic syntax of the ts_rank function is as follows:

```
ts_rank(
  vector tsvector,
  query tsquery [, normalization integer]
) returns double precision
```

Here, the ts_rank function takes in two parameters: a tsvector object representing the text being searched, and a tsquery object representing the search query. The optional "normalization" parameter can be used to specify the normalization method to be used for the search.

By analyzing the proximity and frequency of the search terms within the text, ts_rank assigns a relevance score to each search result, allowing developers to sort the results by relevance and present the most relevant results to the user. This makes ts_rank a critical tool for improving the accuracy and usability of full-text search applications in PostgreSQL.

Here's an example that demonstrates how to use the ts_rank() function to assign a rank to search results based on their relevance to the query:

```
EXPLAIN ANALYZE SELECT paragraph, body, ts_rank(to_tsvector('english', body),
phraseto_tsquery('english', 'right and left')) AS rank
FROM play_table
WHERE to_tsvector('english', body) @@ phraseto_tsquery('english', 'right and
    left')
ORDER BY rank DESC;

"Planning Time: 0.114 ms"
"Execution Time: 735.127 ms"
```

In this example, we're searching for articles that contain the terms right and left and using the ts_rank() function to calculate a relevance score for each article based on how well it matches the query. The ts_rank() function takes two arguments: the ts_vector to rank, and the ts_query to rank it against. It returns a value between 0 and 1, where 1 represents a perfect match and 0 represents no match.

First Way

One optimization you can do is to use a subquery to calculate the rank and only return the rows with a rank greater than 0. This can help to reduce the amount of data that needs to be sorted.

Here's an optimized version of the code:

```
EXPLAIN ANALYZE SELECT paragraph, body, rank
FROM (
SELECT paragraph, body, ts_rank(to_tsvector('english', body),
phraseto_tsquery('english', 'right and left'))
AS rank
FROM play_table
WHERE to_tsvector('english', body) @@ phraseto_tsquery('english', 'right and
    left')
) AS ranked_docs
WHERE rank > 0
ORDER BY rank DESC;

"Planning Time: 0.126 ms"
"Execution Time: 734.840 ms"
```

In this version, the subquery calculates the rank for each document that matches the search query, and only returns rows with a rank greater than 0. The outer query then sorts the results by rank and returns the body and rank columns.

This optimization can improve performance by reducing the amount of data that needs to be sorted and returned.

Second Way

Here is a more optimized version of the query:

```
EXPLAIN ANALYZE
SELECT paragraph, body, ts_rank_cd(content_tsv, query_tsv) AS rank
FROM (
SELECT paragraph, body, to_tsvector('english', body) AS content_tsv,
phraseto_tsquery('english', 'right and left') AS query_tsv
```

```
FROM play_table
) AS subquery
WHERE content_tsv @@ query_tsv
ORDER BY rank DESC;

"Planning Time: 0.106 ms"
"Execution Time: 594.845 ms"
```

In this optimized version, we first create the tsvector and tsquery objects as subqueries in the FROM clause. Then we use the @@ operator to check if the tsvector matches the tsvector‘tsquery. Finally, we calculate the ranking score using the ts_rank_cd function.

Note that we also use the ts_rank_cd function instead of ts_rank. The ts_rank_cd function is faster than ts_rank because it uses the more efficient ”cover density” algorithm to calculate the ranking score.

By optimizing the query in this way, we can significantly improve its performance, especially for large datasets.

Third Way

Here is a possible optimization of the previous code:

```
EXPLAIN ANALYZE
SELECT paragraph, body, ts_rank_cd(tsv, query) AS rank
FROM (
SELECT paragraph, body, setweight(to_tsvector('english', body), 'A') AS tsv
FROM play_table
) d, phraseto_tsquery('english', 'right and left')query
WHERE d.tsv @@ query
ORDER BY rank DESC

"Planning Time: 0.099 ms"
"Execution Time: 537.450 ms"
```

This code takes advantage of several optimizations:

- It uses the **ts_rank_cd** function instead of **ts_rank**, which is more efficient because it does not require normalization.
- It creates the **tsvector** in a subquery, so that it can be reused in the main query without recomputing it for each row.
- It uses the **setweight** function to assign weight to the **body** and field. This allows for more fine-grained ranking based on the relevance of each field.
- It uses the **@@** operator to perform the full-text search directly on the **tsvector** column, rather than on the text fields. This is faster because it avoids the overhead of converting the text to **tsvector** for each query

2.4 Dictionaries

In the context of full-text search, a dictionary refers to a set of rules and regulations that are employed to recognize and standardize words within a given text. These rules are language-specific and serve the purpose of identifying the base or fundamental form of a word, also known as its stem.

The utilization of dictionaries plays a crucial role in enhancing the accuracy and effectiveness of searches.

In PostgreSQL's full-text search functionality, the sequence of dictionaries utilized is determined by the text search configuration (TSC) settings [10]. A text search configuration defines the specific dictionaries that are employed for tasks such as tokenization, normalization, and stemming [10]. It also allows for the inclusion of custom rules and regulations, such as stopword lists or synonym dictionaries, to further refine the search process.

During the query processing stage, the text search configuration's dictionaries are accessed in a specific order, as defined within the configuration itself. This order of dictionary access influences the tokenization, normalization, and stemming processes, enabling the search engine to accurately identify and interpret the words within the text being searched.

By carefully configuring the text search configuration and specifying the appropriate sequence of dictionaries, developers can ensure that the full-text search functionality in PostgreSQL aligns with the specific linguistic requirements of their application. This allows for more precise and accurate searches, leading to improved search results and a better overall user experience.

The default TSC in PostgreSQL is **pg_catalog.english**, which uses several built-in dictionaries and rules for English language text. The order of dictionaries used in this TSC is:

1. **Word segmentation**: This dictionary breaks the text into words based on whitespace and punctuation.
2. **Lowercase normalization**: This dictionary converts all words to lowercase.
3. **Stopword filtering**: This dictionary removes common words like ”the” and ”and” from the text.
4. **English stemming**: This dictionary reduces words to their base form, such as converting ”jumped” to ”jump”.
5. **Synonym dictionary**: This dictionary maps equivalent words to a single term. For example, ”car” and ”automobile” might both map to the term ”vehicle”.
6. **Thesaurus dictionary**: This dictionary provides a hierarchy of related terms for a given word. For example, a thesaurus might indicate that ”dog” is a type of ”animal”.
7. **Soundex dictionary**: This dictionary maps words to a phonetic code, which can be used to find similar-sounding words in the search index.
8. **Trigram dictionary**: This dictionary breaks words into groups of three letters (trigrams), which can be used to find similar words based on partial matches. For example, a trigram dictionary might find ”cat” and ”hat” as similar words.

The process of full-text searching involves several distinct steps, including tokenization, stop word removal, stemming, synonyms, thesaurus, matching and scoring, and returning search hits. By breaking down the search query into individual tokens and applying various linguistic techniques like stemming and synonym mapping, PostgreSQL is able to provide accurate and relevant search results to the user. Overall, full-text search in PostgreSQL is a valuable tool for anyone who needs to search large volumes of text and extract meaning from unstructured data.

2.5 Full-text indexing in PostgreSQL

PostgreSQL provides several full-text indexing methods, each designed to address specific requirements and offers unique advantages. These methods include GIN (Generalized Inverted In-

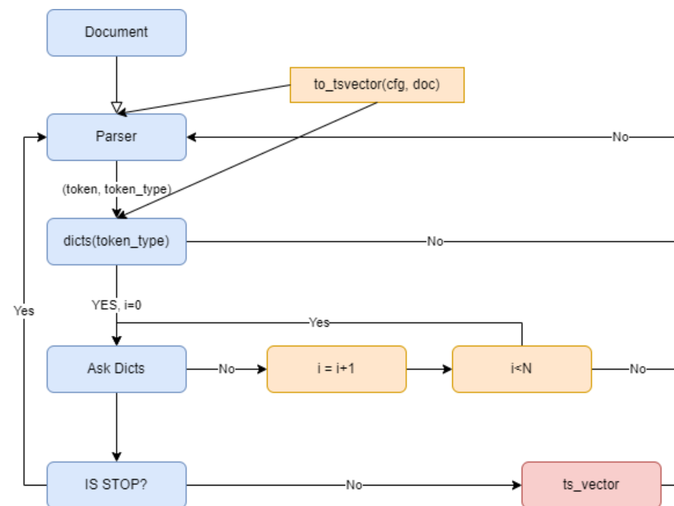

Figure 1. FTS PostgreSQL Dictionary Flow

dex), GIST (Generalized Search Tree), BRIN (Block Range Index), and SP-GiST (Space-Partitioned Generalized Search Tree) [4] [17].

The GIN indexing method is well-suited for large and frequently updated text data [4]. It excels at handling partial matches and provides good overall performance. With GIN, you can efficiently search for and retrieve relevant results, even when the search terms are not exact matches.

If you require fast searching of exact matches and a wide range of search operators, the GIST indexing method is an excellent choice. It utilizes a generalized search tree structure, making it suitable for various data types and offering a broad set of search capabilities [17].

On the other hand, the BRIN indexing method is specifically designed for large datasets with ordered data. It performs particularly well on range queries, making it an efficient choice for scenarios where you need to search within specific value ranges [4].

For complex geometric data, the SP-GiST indexing method is the ideal option [17]. It provides fast searching capabilities for complex queries involving geometric shapes and spatial data.

When deciding on the best full-text indexing method for your application, it's crucial to consider your specific needs. Factors such as the size and nature of your data, the types of queries you'll be running, and the frequency of data updates should be taken into account. By carefully selecting the appropriate indexing method and tuning the search parameters, you can build highly efficient and accurate search functionality in your PostgreSQL applications.

Please see PostgreSQL indexing methods with more details:

1. **GIN (Generalized Inverted Index)** - a high-performance index type that can handle complex queries and supports full-text search as well as range and equality queries.

2. **GIST (Generalized Search Tree)** - a flexible index type that supports a wide range of queries, including full-text search, spatial search, and custom search operations.

3. **Brin (Block Range INdex)** - a lightweight index type that is designed for very large tables and can be used for full-text search as well as other types of queries.

4. **SP-GiST (Space-Partitioned Generalized Search Tree)** - an index type that is optimized for spatial data and can also be used for full-text search.

2.5.1 Example - GIN & GiST

Let's see a detailed example regarding how we can create a full text indexing using GIN and GiST.

1. **Create a sample table**:

```
CREATE TABLE articles (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  body TEXT NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT NOW()
);
```

This creates a simple table with an auto-incrementing ID, a title column, a body column, and a created_at column.

2. **Insert some sample data**:

```
INSERT INTO articles (title, body)
VALUES
  ('How to make a great cup of coffee', 'In this article, we will share
     our tips for making the perfect cup of coffee.'),
  ('The benefits of meditation', 'Meditation has been shown to reduce
     stress and improve mental clarity.'),
  ('10 healthy snack ideas', 'Here are 10 snack ideas that are both
     delicious and nutritious.');
```

This inserts some sample data into the "articles" table.

3. **Install the full-text search extension and the unaccent extension**:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE EXTENSION IF NOT EXISTS unaccent;
```

This installs the "pg_trgm" extension, which provides trigram-based text indexing and similarity search capabilities, and the "unaccent" extension, which removes accents from text to improve search quality.

4. **Create a full-text index**:

```
CREATE INDEX articles_fulltext_idx ON articles USING
    gin(to_tsvector('english', unaccent(title || ' ' || body)));
```

This creates a full-text index on the "title" and "body" columns of the "articles" table using the "gin" index type, the "to_tsvector" function to convert text to a search vector, and the "unaccent" function to remove accents from text. The 'english' parameter specifies the language to use for text parsing and stemming.

5. **Perform a basic full-text search**:

```
SELECT * FROM articles WHERE to_tsvector('english', unaccent(title || ' '
    || body)) @@ to_tsquery('english', 'coffee');
```

This performs a simple full-text search for the term "coffee" in the "title" and "body" columns using the "to_tsvector" and "to_tsquery" functions. The "@@" operator checks if the search vector matches the search query

6. **Perform a phrase search**:

```
SELECT * FROM articles WHERE to_tsvector('english', unaccent(title || ' '
    || body)) @@ to_tsquery('english', 'great & cup & coffee');
```

   This performs a phrase search for the phrase ”great cup coffee” in the ”title” and ”body” columns using the ”to_tsquery” function and the ”&” operator to specify that the search terms must appear in sequence.

7. **Perform a ranking search**:

```
SELECT *, ts_rank_cd(to_tsvector('english', unaccent(title || ' ' ||
    body)), to_tsquery('english', 'coffee')) AS rank
FROM articles
WHERE to_tsvector('english', unaccent(title || ' ' || body)) @@
    to_tsquery('english', 'coffee')
ORDER BY rank DESC;
```

   This performs a ranking search for the term ”coffee” in the ”title” and ”body” columns using the ”to_tsvector”, ”to_tsquery”, and ”ts_rank_cd” functions. The ”ts_rank_cd” function calculates a ranking score between the search vector and the search query. The results are sorted by the ranking score in descending order to prioritize the best matches.

Similar to GIN, we can also use the GiST index type for full-text search in PostgreSQL. The main difference between GIN and GiST is that GiST is optimized for indexing overlapping data structures, whereas GIN is optimized for non-overlapping data structures.

The choice between GIN and GiST indexing methods in PostgreSQL for full-text indexing depends on the specific use case and data characteristics. GIN indexes are generally faster to build and update, and work well with small to medium-sized text documents. They are also better suited for exact matches and prefix searches.

On the other hand, GiST indexes are slower to build and update, but they perform better with larger documents and complex queries that involve multiple keywords and phrases.

In general, if the dataset consists of small to medium-sized documents and exact matches or prefix searches are the most common queries, GIN indexing may be the better option. However, if the dataset contains large documents and complex queries with multiple keywords and phrases are common, GiST indexing may be more suitable.

2.5.2 Advantages and Disadvantages - Gist vs GIN

When it comes to full-text indexing in PostgreSQL, there are two popular indexing methods available: GIN (Generalized Inverted Index) and GIST (Generalized Search Tree). The choice between these methods depends on the specific requirements of your application, as each method has its own advantages and disadvantages.

The GIN index method is well-suited for applications that necessitate fast searching of large volumes of data [4]. It efficiently handles complex queries, including phrase searches, prefix searches, and fuzzy searches. Moreover, the GIN index performs particularly well in scenarios where the indexed data undergoes frequent updates or additions. However, it’s worth noting that the GIN index may require more disk space compared to the GIST index. Additionally, building the GIN index can be slower compared to the GIST index, especially for large datasets.

On the other hand, the GIST index method is an excellent choice for applications that require more advanced search functionality beyond the standard full-text search capabilities offered by PostgreSQL. The GIST index can handle complex queries that involve spatial relationships or

geographic proximity, making it suitable for applications dealing with location-based data. Additionally, the GIST index is well-suited for applications that require efficient indexing of frequently updated data. However, it's important to consider that the GIST index may not perform as optimally as the GIN index for large datasets. It may also be less efficient for simple queries that do not require the additional capabilities provided by the GIST index.

Here are some pros and cons of GIN and GiST in PostgreSQL's full-text indexing:

GIN:
Advantages:

- Faster indexing for larger data sets and larger documents: GIN is optimized for indexing larger data sets and documents, making it a good choice for applications with a high volume of textual data.
- Good for read-heavy workloads where queries are frequently executed: GIN's indexing structure is optimized for fast queries, making it a good choice for read-heavy workloads.
- More space efficient than GiST for indexing arrays and multiple columns: GIN uses less storage space than GiST for indexing arrays and multiple columns, making it more efficient for these scenarios.
- Supports indexing of arrays and multiple columns: GIN can index arrays and multiple columns, making it a versatile option for indexing complex data types.
- Better performance for queries that match more common terms: GIN performs well for queries that match more common terms, such as frequently used words.
- Good for full-text search scenarios that require ranking, highlighting, and fuzzy matching: GIN provides support for advanced search features like ranking, highlighting, and fuzzy matching, making it a good choice for full-text search scenarios.

Disadvantages

- Slower for small data sets or small documents: GIN's indexing structure is optimized for larger data sets, so it may not perform as well for smaller data sets or documents.
- More memory consumption during indexing and query execution: GIN requires more memory during indexing and query execution, which may be a consideration for applications with limited resources.
- Not as suitable for write-heavy workloads, as updates to the indexed data require more resources: GIN's indexing structure makes updates to the indexed data more resource-intensive, so it may not be the best choice for write-heavy workloads.
- Not ideal for prefix matching or range queries: GIN is not optimized for prefix matching or range queries, which may impact performance in these scenarios.

GiST
Advantages

- Good for prefix matching, range queries, and complex data types: GiST is optimized for prefix matching, range queries, and complex data types, making it a good choice for certain types of search scenarios.
- Supports custom indexing schemes and user-defined operators: GiST provides flexibility for developers to create custom indexing schemes and user-defined operators.

- Less memory consumption during indexing and query execution: GiST requires less memory than GIN during indexing and query execution.
- Suitable for write-heavy workloads, as updates to the indexed data are more efficient: GiST's indexing structure makes updates to the indexed data more efficient than GIN, making it a good choice for write-heavy workload
- Supports geometric and spatial data types: GiST supports indexing for geometric and spatial data types.

Disadvantages:

- Slower indexing performance for larger data sets and larger documents: GiST may not perform as well as GIN for indexing larger data sets and documents.
- Larger index size than GIN: GiST's indexing structure requires more storage space than GIN, which may be a consideration for applications with limited storage.
- Not as efficient for read-heavy workloads as GIN: GiST may not perform as well as GIN for read-heavy workloads.
- Not as space efficient as GIN for indexing arrays and multiple columns: GIN is more space efficient than GiST for indexing arrays and multiple columns, so it may be a better choice for these scenarios.
- Less suitable for full-text search scenarios that require ranking, highlighting, and fuzzy matching: GiST is not as optimized for advanced search features like ranking, highlighting, and fuzzy matching, so it may not be the best choice for full-text search scenarios that require these features.

It's worth noting that the choice between GIN and GiST depends on the specific use case and workload of the application. In general, GIN is a better choice for full-text search scenarios, while GiST is better suited for prefix matching, range queries, and complex data types.

Chapter 3

MySQL - Full text Indexing

Full-text searching in MySQL is a powerful tool that enables users to search for words or phrases within text fields in a database [7]. This feature is particularly valuable when dealing with large amounts of unstructured or semi-structured text data, such as news articles, product descriptions, or user comments.

MySQL supports full-text searching for MyISAM and InnoDB tables, providing flexibility in choosing the appropriate table type for your specific needs. When performing a full-text search in MySQL, the system creates an index of the words in the targeted text field [7]. This index enables MySQL to quickly locate relevant information during searches, resulting in significantly faster search operations compared to traditional pattern-matching techniques.

One of the key advantages of full-text searching in MySQL is its ability to return search results based on relevance [7]. MySQL utilizes a ranking algorithm that assigns a relevance score to each search result, considering factors such as the frequency of the search term within the text, the proximity of search terms to each other, and the length of the text being searched [7] [21]. This ranking algorithm ensures that the most relevant search results are displayed first, saving valuable time when working with extensive data sets.

However, it is important to be aware that full-text searching can be resource-intensive, particularly for large datasets. The process of creating and updating the full-text index consumes system resources, and the performance impact should be considered when implementing full-text search functionality [23]. It is crucial to assess the specific search requirements of your application and choose the appropriate search solution accordingly. While full-text indexing is a powerful tool, alternative search methods may be more suitable for certain use cases.

By understanding the capabilities and considerations of full-text searching in MySQL, developers can make informed decisions when implementing search functionality in their applications. Assessing the performance implications, considering the specific search requirements, and choosing the right approach will result in an efficient and effective search solution

3.1 Types of full text searching in MySQL

Full-text searching in MySQL is a powerful technique that allows you to search for text data stored in a database using keywords or phrases. It offers three types of full-text searching: Natural Language Full-Text Search, Boolean Full-Text Search, and Query Expansion Full-Text Search [7] [28]. These types provide different capabilities and functionalities, enabling you to perform searches based on specific requirements.

The Natural Language Full-Text Search is designed for searching natural language text [7]. It utilizes a language-based approach to determine the relevance of search results. It considers factors such as word proximity, stemming, and stop words to improve the accuracy of the search. This

type is well-suited for applications that require user-friendly and intuitive search experiences.

The Boolean Full-Text Search, on the other hand, allows you to construct complex search queries using Boolean operators like AND, OR, and NOT. This type is useful when you need precise control over the search results, enabling you to combine search terms and conditions to refine the query and obtain specific matches [7] [13].

The Query Expansion Full-Text Search expands the original search query by adding related terms. It helps to broaden the search scope and retrieve additional relevant results. This type is beneficial when you want to improve recall and discover related content.

To utilize these full-text search types in MySQL, you can specify the desired search mode within the MATCH() function using the corresponding modifiers: ”IN NATURAL LANGUAGE MODE,” ”IN BOOLEAN MODE,” or ”WITH QUERY EXPANSION.”

It's important to understand the differences between these search types and their implications. Consider factors such as the nature of your data, the desired search behavior, and the trade-off between precision and recall. By selecting the appropriate search type, you can enhance the accuracy and efficiency of your search queries, improving the overall search experience for your application users [7] [23].

To use these types of full-text searching in MySQL, you can specify the search type in the MATCH() function using the ”IN NATURAL LANGUAGE MODE”, ”IN BOOLEAN MODE”, or ”WITH QUERY EXPANSION” modifiers.

```
search_modifier:
{
IN NATURAL LANGUAGE MODE
| IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION
| IN BOOLEAN MODE
| WITH QUERY EXPANSION
}
```

There are three types of full-text searching in MySQL:

1. **Natural Language Full-Text Search**: This type of full-text searching is designed to work with natural language text, such as articles, blog posts, or product descriptions. It uses a ranking algorithm to determine the relevance of each search result based on factors such as word proximity, keyword density, and the presence of stop words (common words like ”a”, ”an”, ”the”, etc.). Natural Language Full-Text Search is the default type of full-text search in MySQL.

```
SELECT * FROM products WHERE MATCH (description) AGAINST ('search query'
    IN NATURAL LANGUAGE MODE);
```

2. **Boolean Full-Text Search**: This type of full-text searching is designed to work with structured data, such as databases, tables, or forms. It uses Boolean operators (AND, OR, NOT) and other special characters to allow for more precise search queries. For example, you can use the ”+” symbol to require a particular word or phrase, or the ”-” symbol to exclude a word or phrase from the search results.

```
SELECT * FROM articles WHERE MATCH (title,body) AGAINST ('search query1
    search query2' IN BOOLEAN MODE);
```

3. **Query Expansion Full-Text Search**: This type of full-text searching is designed to improve the relevance of search results by automatically expanding the search query to include related words and phrases. For example, if you search for ”dog”, Query Expansion Full-Text Search might automatically include related words like ”puppy”, ”canine”, or ”pet” in the search results.

```
SELECT * FROM books WHERE MATCH (description) AGAINST ('search query'
    WITH QUERY EXPANSION);
```

To use these types of full-text searching in MySQL, you can specify the search type in the MATCH() function using the ”IN NATURAL LANGUAGE MODE”, ”IN BOOLEAN MODE”, or ”WITH QUERY EXPANSION” modifiers.

3.2 Natural Language Full-Text Search

The ”IN NATURAL LANGUAGE MODE” clause in MySQL is a feature that allows for full-text searches to be performed in a more natural language style. It enables users to enter a search query using common phrases and sentences, rather than specific keywords or complex search syntax.

When using ”IN NATURAL LANGUAGE MODE” for full-text searching in MySQL, the search engine will attempt to match the search query with the most relevant results from the specified table or column(s). This is done by analyzing the text and considering factors such as the frequency and proximity of words, and the presence of synonyms and related terms [7] [21]. The clause can be used in conjunction with other search parameters, such as Boolean operators and wildcards, to further refine the search results. It is important to note that the effectiveness of natural language searching may depend on the quality and completeness of the data being searched, as well as the specific search algorithms and settings used by MySQL.

3.2.1 Examples

”In natural language mode” is a specialized search mode in MySQL that is designed to improve the accuracy and relevance of full-text search results by taking into account the natural language usage of the search terms. It allows you to search for text data in a more intuitive and natural way, using keywords, phrases, and synonyms that reflect the user’s intent.

The benefits include more accurate search results, better handling of synonyms and stemming, improved ranking of results by relevance, and the ability to construct more complex search queries using Boolean operators. This can help to make your application more user-friendly, by making it easier for users to find the information they need, and can improve the overall user experience.

Here are some examples of how each capability of ”IN NATURAL LANGUAGE MODE” can be used in MySQL:

1. **Phrase searching**: Suppose you have a table named ”articles” with a column named ”content” that contains the text of various articles. You can search for the exact phrase ”customer satisfaction survey” like this:

   ```
   SELECT * FROM articles WHERE MATCH(content) AGAINST('"customer
       satisfaction survey"' IN NATURAL LANGUAGE MODE);
   ```

2. **Synonym matching**: Suppose you have a table named ”products” with a column named ”description” that contains various product descriptions. You can search for products that are ”big” like this:

   ```
   SELECT * FROM products WHERE MATCH(description) AGAINST('big' IN NATURAL
       LANGUAGE MODE);
   ```

3. **Ranking results by relevance**: Suppose you have a table named ”documents” with a column named ”text” that contains various documents. You can search for the word ”database” like this:

```
SELECT *, MATCH(text) AGAINST('database' IN NATURAL LANGUAGE MODE) AS
    relevance FROM documents WHERE MATCH(text) AGAINST('database' IN
    NATURAL LANGUAGE MODE) ORDER BY relevance DESC;
```

4. **Stemming:**:

   Suppose you have a table named "words" with a column named "word" that contains various words. You can search for words that have the same stem as "run" like this:

   ```
   SELECT * FROM words WHERE MATCH(word) AGAINST('run' IN NATURAL LANGUAGE
       MODE);
   ```

5. **Stopword handling:**:

   Suppose you have a table named "texts" with a column named "content" that contains various texts. You can search for the phrase "the quick brown fox" like this:

   ```
   SELECT * FROM texts WHERE MATCH(content) AGAINST('+quick +brown +fox' IN
       NATURAL LANGUAGE MODE);
   ```

6. **Boolean operators:**:

   Suppose you have a table named "posts" with a column named "content" that contains various forum posts. You can search for posts that contain the words "MySQL" and "database" like this:

   ```
   SELECT * FROM posts WHERE MATCH(content) AGAINST('+MySQL +database' IN
       NATURAL LANGUAGE
   ```

Here's an example SQL query that uses IN NATURAL LANGUAGE MODE and the COUNT function to retrieve the matching blog posts and the number of times the search term appears in each post:

```
SELECT id, title, author, created_at,
MATCH (content) AGAINST ('MySQL' IN NATURAL LANGUAGE MODE) AS relevance,
COUNT(*) - LENGTH(REPLACE(content, 'MySQL', '')) AS term_count
FROM blog_posts
WHERE MATCH (content) AGAINST ('MySQL' IN NATURAL LANGUAGE MODE)
GROUP BY id
ORDER BY relevance DESC, created_at DESC;
```

In this query, we use the MATCH function to search for the search term "MySQL" in the content column using IN NATURAL LANGUAGE MODE. We also use the AS keyword to give the relevance score a name (relevance) so that we can refer to it later.

We then use the COUNT function to count the number of times the search term appears in each matching row. We subtract the length of the string with all occurrences of the search term removed from the length of the original string to get the number of times the search term appears.

We retrieve the id, title, author, created_at, relevance, and term_count columns from the blog_posts table, and use the GROUP BY keyword to group the results by the id column (since we only want to count each blog post once). We then use the ORDER BY keyword to sort the results by the relevance column in descending order and then by the created_at column in descending order.

3.2.2 Limitations

While "IN NATURAL LANGUAGE MODE" is a powerful and useful tool for full-text search in MySQL, it is essential to be aware of its limitations before deciding to use it in your application.

These limitations can impact the accuracy, precision, and performance of your search queries, and it may be necessary to consider alternative search modes or techniques to achieve the desired results.

One of the primary limitations of ”IN NATURAL LANGUAGE MODE” is its limited language support. It may not provide optimal results for languages other than English or for complex linguistic structures [7]. If your application involves multilingual content or requires precise language-specific search capabilities, you may need to explore other search modes or language-specific full-text search techniques.

Another limitation is the lack of precision for short queries. ”IN NATURAL LANGUAGE MODE” tends to produce less accurate results for shorter search queries, as it primarily focuses on the relevance of individual words rather than the context or meaning of the entire phrase [7]. If your application frequently involves short search terms or phrases, you might consider using alternative search modes or techniques that can provide more precise results.

Moreover, ”IN NATURAL LANGUAGE MODE” offers limited control over search behavior. It relies on a predefined set of rules and algorithms to determine relevance, and it may not allow fine-grained customization or advanced query manipulation. If your application requires more granular control over search behavior, such as boosting certain terms, filtering results, or specifying complex search conditions, you might need to explore other search modes or implement custom search logic.

Performance can also be a concern with ”IN NATURAL LANGUAGE MODE” [23]. In some cases, particularly when dealing with large datasets or complex queries, the performance of natural language search may not be optimal [7] [23]. It may require additional optimizations or indexing strategies to achieve satisfactory search speed and scalability. If performance is a critical factor for your application, you should consider benchmarking and profiling different search modes to identify the most efficient option.

Lastly, ”IN NATURAL LANGUAGE MODE” may have limitations when searching for special characters [7]. Depending on the configuration and version of MySQL, certain special characters or punctuation marks may not be processed correctly or may not be searchable at all. If your application requires searching for specific characters or symbols, it is important to understand the behavior of ”IN NATURAL LANGUAGE MODE” with regards to special characters and consider alternative search modes or techniques if necessary [7] [21].

Understanding these limitations of ”IN NATURAL LANGUAGE MODE” will enable you to make informed decisions about its suitability for your application. By evaluating your specific requirements and considering alternative search modes or techniques, you can optimize your search queries to achieve maximum accuracy, precision, and performance

1. **Limited language support**: ”IN NATURAL LANGUAGE MODE” is designed primarily for English language text. While it may work for other languages, it may not be as effective at handling stemming, synonyms, and stop words in languages other than English

2. **Lack of precision for short queries**: For short queries, the ”IN NATURAL LANGUAGE MODE” may not be precise enough to return accurate results. This is because the relevance ranking algorithm relies heavily on the frequency of the search terms in the text, and short queries may not have enough terms to generate an accurate relevance score.

3. **Limited control over search behavior**: Unlike Boolean mode, ”IN NATURAL LANGUAGE MODE” does not allow fine-grained control over the search behavior. This can be a limitation when trying to construct complex search queries that require precise control over the search terms and operators.

4. **Performance issues**: Using ”IN NATURAL LANGUAGE MODE” can be resource-intensive, especially when dealing with large datasets or complex queries. This can lead to slower search performance and increased server load.

5. **Inability to search for special characters**: "In natural language mode" does not allow searching for special characters, such as quotation marks or parentheses, which can limit its usefulness for certain types of searches. It is important to keep these limitations in mind when deciding whether to use "IN NATURAL LANGUAGE MODE" in your MySQL application. Depending on your specific use case and requirements, it may be necessary to explore other search modes or techniques to achieve the desired search results.

3.2.3 Rank Function

In this article, we present some queries that demonstrate the capabilities of Natural Language Mode for ranking in MySQL. By using this feature, you can search for specific keywords or phrases within a large dataset and rank the results based on their relevance to the search term. In this context, we explore how to optimize the search query to reduce the overall execution time by using a subquery to calculate the relevance score first and then joining it with the articles table.

Here are some queries that demonstrate the capabilities of Natural Language Mode for ranking in MySQL:

Search for articles about "machine learning" and order the results by relevance score:

```
SELECT title, MATCH (title, content) AGAINST ('machine learning' IN NATURAL
    LANGUAGE MODE) AS relevance_score
FROM articles
WHERE MATCH (title, content) AGAINST ('machine learning' IN NATURAL LANGUAGE
    MODE)
ORDER BY relevance_score DESC;
```

Optimized Way

Another way to optimize the query is to use a subquery to calculate the relevance score first and then join it with the articles table. This way, we can avoid calling the **MATCH** function twice and reduce the overall query execution time.

Here's the optimized query using a subquery:

```
SELECT a.title, r.relevance_score
FROM articles AS a
JOIN (
    SELECT id, MATCH (title, content) AGAINST ('machine learning' IN NATURAL
        LANGUAGE MODE) AS relevance_score
    FROM articles
    WHERE MATCH (title, content) AGAINST ('machine learning' IN NATURAL
        LANGUAGE MODE)
) AS r ON a.id = r.id
ORDER BY r.relevance_score DESC
LIMIT 10;
```

In this example, we're using a subquery to calculate the relevance_score for each article in the articles table. Then, we join the subquery with the articles table based on the id column and fetch only the top 10 results using the LIMIT clause.

By using a subquery, we avoid calling the **MATCH** function twice and reduce the overall query execution time.

3.3 Boolean Mode

Boolean Full-Text Searches in MySQL provide a powerful mechanism for constructing complex search queries using Boolean operators, such as AND, OR, and NOT [7]. This feature allows

users to combine multiple search terms and conditions to create highly customizable search queries [12].

When utilizing Boolean Full-Text Searches in MySQL, you can employ various operators to refine your search results. The + operator indicates that a search term must be present in the search results, while the - operator indicates that a search term must not be present. Additionally, you can use parentheses to group search terms together, enabling the creation of more intricate and specific search conditions [7].

One of the key advantages of Boolean Full-Text Searches is the level of precise control it offers over search results compared to other types of full-text searches. By utilizing Boolean operators, users can fine-tune their queries to ensure that specific search terms or conditions are met. This granular control allows for more targeted and accurate search results [7] [12].

However, it's important to note that constructing search queries for Boolean Full-Text Searches requires careful attention. Even small changes to the query can have a significant impact on the search results. Due to the increased complexity and potential sensitivity to query construction, it is recommended to thoroughly test and validate the queries to ensure the desired results are obtained.

By leveraging the capabilities of Boolean Full-Text Searches in MySQL, users can create sophisticated and tailored search queries that precisely match their requirements. This level of control empowers developers and users to achieve more accurate and focused search results within their MySQL databases.

Please find the basic operators using IN BOOLEAN MODE Operators

Here are some of the operators you can use in Boolean Full-Text Searches in MySQL:

- **+**: Indicates that the term must be present in the search results.
- **:** Indicates that the term must not be present in the search results.
- **AND**: Indicates that both terms must be present in the search results.
- **OR**: Indicates that either term (or both) can be present in the search results.
- **NOT**: Indicates that the following term must not be present in the search results.
- **()** or parentheses: Used to group terms together and create more complex search conditions.
- **> <**: Used to perform proximity searches. For example, search_term1 <5> search_term2 would match if the two terms appear within 5 words of each other in the document.
- **:** Used for wildcard searches. For example, search_te* would match any terms that begin with ”search_te”.
- **” ”**: Used for exact phrase searches. For example, ”search term” would only match documents where the two terms appear together as an exact phrase.
- : Used for fuzzy searches. For example, search_term would match documents that contain terms that are similar to ”search_term” based on the Levenshtein distance.
- |: Used to represent ”or”. For example, search_term1 | search_term2 would match documents that contain either term.

3.3.1 Examples

By using BOOLEAN MODE, you can search for rows that contain specific keywords or phrases, or even synonyms of a word. In this context, we present several examples of BOOLEAN MODE queries that demonstrate its capabilities.

- **Simple Boolean Query**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('+word1 -word2' IN BOOLEAN MODE);
```

This query will search for rows that contain the word ”word1” but not the word ”word2” in the column ”my_column”.

- **Boolean Query with Multiple Keywords**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('+word1 +word2' IN BOOLEAN MODE);
```

This query will search for rows that contain both the words ”word1” and ”word2” in the column ”my_column”.

- **Boolean Query with Phrase**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('"word1 word2"' IN BOOLEAN MODE);
```

This query will search for rows that contain the exact phrase ”word1 word2” in the column ”my_column”.

- **Boolean Query with Parentheses**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('+word1 (word2 | word3)' IN BOOLEAN
    MODE);
```

This query will search for rows that contain the word ”word1” and either the word ”word2” or ”word3” in the column ”my_column”.

- **Boolean Query with Negation**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('+word1 -word2' IN BOOLEAN MODE);
```

This query will search for rows that contain the word ”word1” but not the word ”word2” in the column ”my_column”.

- **Boolean Query with Required and Optional Terms**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('+word1 ~word2' IN BOOLEAN MODE);
```

This query will search for rows that contain the word ”word1” and either the word ”word2” or another related term in the column ”my_column”.

- **Boolean Query with Proximity Search**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('"word1 word2"~3' IN BOOLEAN MODE);
```

This query will search for rows that contain the exact phrase ”word1 word2” with up to 3 words between them in the column ”my_column”.

- **Boolean Query with Thesaurus Expansion**

```
SELECT * FROM my_table
WHERE MATCH (my_column) AGAINST ('~word1' IN BOOLEAN MODE);
```

This query will search for rows that contain synonyms of the word "word1" in the column "my_column".

These examples demonstrate the flexibility and power of "IN BOOLEAN MODE" in MySQL, allowing for more precise and complex search queries than "IN NATURAL LANGUAGE MODE".

3.3.2 How MySQL calculate the ranking

The calculation of relevance ranking involves two main values: term frequency (TF) and inverse document frequency (IDF). The TF value represents the number of times a specific word appears in a document. The IDF value, on the other hand, is calculated using the following formula[7]:

$$\$IDF = log10(\$total_records/\$matching_records) \tag{3.1}$$

where $total_records is the total number of records in the collection and $matching_records is the number of records that contain the search term. When a document contains a word multiple times, the IDF value is multiplied by the TF value.

Using the calculated values for TF and IDF, the formula for relevance ranking can be calculated by multiplying.

$$\$rank = \$TF * \$IDF * \$IDF. \tag{3.2}$$

This formula helps in determining the relevance of a document to a given search query.

Suppose we have a collection of articles and we want to search for articles containing the term "MySQL" and "database". We want to order the results by relevance ranking, where the most relevant articles are those with the highest value of the formula: $TF * $IDF * $IDF

To achieve this in Boolean mode, we could use the following query:

```
SELECT *, MATCH (article_content) AGAINST ('+MySQL +database' IN BOOLEAN MODE)
    AS relevance_rank
FROM articles
WHERE MATCH (article_content) AGAINST ('+MySQL +database' IN BOOLEAN MODE)
ORDER BY relevance_rank DESC;
```

In this example, the MATCH function is used with the BOOLEAN MODE modifier to search for articles that contain both the terms "MySQL" and "database". The query also calculates the relevance ranking for each article using the formula:

$$\$TF * \$IDF * \$IDF \tag{3.3}$$

Note that in this example, the TF value is the number of times that the term "MySQL" and "database" appears in the article, the IDF value is calculated using the formula $IDF = log10($total_records / $matching_records) where $total_records is the total number of articles in the collection and $matching_records is the number of articles that contain the search term, and the $rank is calculated using the formula $TF * $IDF * $IDF.

3.3.3 Limitations

IN BOOLEAN MODE is a full-text search mode in MySQL that provides flexibility in searching for data by allowing users to use Boolean operators such as AND, OR, and NOT to specify complex search queries. While it offers some advantages, such as flexibility and speed, there are also limitations to using this mode [34] [40].

One of the main limitations is the lack of linguistic analysis, which can lead to imprecise search results. In this mode, MySQL treats the search terms as individual words without considering linguistic variations or relationships. As a result, the search may not take into account synonyms, stemming, or language-specific rules. This can impact the accuracy and relevance of the search results, especially when dealing with languages with complex linguistic structures or multiple word forms.

Additionally, IN BOOLEAN MODE does not provide relevance ranking [34] [40]. The search results are based on the presence or absence of the search terms, rather than their relevance or significance within the document. This means that all matching documents are considered equally relevant, which may not always reflect the actual importance or quality of the results.

Furthermore, the search query syntax in IN BOOLEAN MODE can be more complex and less intuitive compared to other full-text search modes. Constructing advanced queries with multiple Boolean operators and parentheses requires careful attention to syntax and can be error-prone.

There are some limitations of using the IN BOOLEAN MODE in MySQL for full-text searching. Here are a few:

1. **Lack of Linguistic Analysis**: The IN BOOLEAN MODE does not perform any linguistic analysis on the search terms. This means that the search results may not always be accurate or relevant to the user's query.

2. **Limited Precision and Recall**: has limited precision and recall compared to other full-text search modes. This is because it only returns exact matches or partial matches based on the presence or absence of search terms in the document.

3. **No Relevance Ranking**: Unlike the IN NATURAL LANGUAGE MODE, the IN BOOLEAN MODE does not provide any relevance ranking for the search results. This means that it does not consider the relevance of the search terms in the document.

4. **Limited Operators**: Limited set of operators, such as AND, OR, and NOT. This can limit the flexibility of complex search queries.

5. **No Stop Words**: IN BOOLEAN MODE does not support stop words, which are common words that are often excluded from search queries to improve search performance. This can lead to irrelevant search results in some cases.

While BOOLEAN MODE can be a useful tool for keyword-based searches, it's important to understand its limitations and consider other options such as the natural language mode or external search engines for more advanced search functionality.

3.4 Query Expansion Mode

Query expansion is a technique used to improve the quality and relevance of search results by automatically expanding the original search query to include additional keywords or phrases [7]. In query expansion mode, the full-text search engine uses a combination of the original search query and additional information about the indexed data to generate a list of related keywords or

phrases. These related keywords are then added to the original search query to broaden the search and improve the chances of finding relevant results.

Query expansion can be beneficial in several ways. It helps to overcome the limitations of an exact match search by considering synonymous terms or related concepts. By including these additional keywords or phrases, the search results become more comprehensive and encompass a wider range of relevant documents. This approach is particularly useful when dealing with ambiguous queries or when users may use different terminology than what is present in the indexed data [7].

3.4.1 Optimize Natural Language Mode using Query expansion

Let's say we have a database of movie titles and descriptions, and we want to search for movies that are about "artificial intelligence."

First, let's look at a query that uses Natural Language Mode:

```
SELECT movie_id, title, description
FROM movies
WHERE MATCH (title, description) AGAINST ('artificial intelligence' IN NATURAL
    LANGUAGE MODE);
```

This query will return results that match the exact phrase "artificial intelligence" in the movie title or description.

Now, let's look at a query that uses Query Expansion:

```
SELECT movie_id, title, description
FROM movies
WHERE MATCH (title, description) AGAINST ('+artificial intelligence' WITH
    QUERY EXPANSION);
```

This query will return results that not only match the exact phrase "artificial intelligence," but also include related keywords or phrases. For example, if the search engine identifies "machine learning" or "robotics" as related keywords, it will include movies that contain those terms as well.

Using Query Expansion in this case can optimize the search because it will return a wider range of relevant results, instead of just exact matches. However, it's worth noting that in some cases, Natural Language Mode might be more appropriate if we want to specifically search for an exact phrase or word.

3.5 Full text Indexing in My SQL

Full-text indexing in MySQL provides a valuable mechanism for efficiently searching character-based data stored in columns such as CHAR, VARCHAR, and TEXT. By default, MySQL utilizes the MyISAM storage engine for full-text indexing, but the InnoDB storage engine can also be employed for this purpose, offering flexibility based on specific needs [28].

When a full-text index is created in MySQL, a separate table is generated by the database engine. This table contains the indexed words and their respective locations within each row of the original table [20]. This index table is optimized for fast searching, enabling rapid retrieval of rows that contain the desired words or phrases during full-text searches.

To conduct a full-text search in MySQL, the MATCH() function is utilized. This function takes one or more search terms as input and returns the rows that contain those terms [7]. Additionally, modifiers such as BOOLEAN can be applied, allowing the usage of Boolean operators (AND, OR, NOT) within the search, while NATURAL LANGUAGE provides more advanced language-based search capabilities [12].

Full-text indexing is a powerful tool that facilitates efficient searching of large datasets containing text-based information in MySQL. By creating a full-text index on columns containing textual data, users can perform fast and accurate searches for specific words or phrases.

3.5.1 InnoDB and MyISAM

The InnoDB and MyISAM storage engines are two of the most commonly used storage engines in MySQL, and both support full-text indexing on CHAR, VARCHAR, or TEXT columns [9]. While the basic concept of full-text indexing is similar across both engines, there are some differences in their implementation and performance characteristics [21].

MyISAM has traditionally been the preferred storage engine for full-text indexing in MySQL because it provides faster indexing and searching capabilities for large text data . In contrast, InnoDB has been used more for transactional data and is known for its strong data consistency and reliability [11]. However, InnoDB has been enhanced in recent versions of MySQL to provide better support for full-text indexing, including support for the Boolean full-text search and proximity search operators, and full-text search indexes that can be combined with other types of indexes for even better query performance [15] [18].

Choosing between InnoDB and MyISAM for full-text indexing depends on the specific needs of your application, including factors such as query performance, data consistency requirements, and transactional support. It's important to carefully evaluate your application requirements and choose the storage engine that best meets your needs.

1. **Query performance**: MyISAM provides faster indexing and searching capabilities for large text data, which makes it a better choice for read-heavy applications that require fast query performance. In contrast, InnoDB is optimized for transactional support, and may be a better choice for applications that require a high level of data consistency and durability.

2. **Data consistency requirements**: InnoDB is known for its strong data consistency and reliability, while MyISAM does not provide transactional support and can be more prone to data corruption or loss. If data consistency is a top priority for your application, InnoDB may be a better choice.

3. **Transactional support**: If your application requires transactional support, then InnoDB is the recommended choice. InnoDB provides support for ACID-compliant transactions, which ensures that data is reliably stored and maintained in the face of failures or concurrent access.

4. **Storage space**: MyISAM typically requires less storage space than InnoDB, which can be an advantage if you have limited storage resources available.

5. **Concurrent access**: InnoDB is designed to support high concurrency and provides features such as row-level locking, which can help improve performance in multi-user environments. MyISAM, on the other hand, uses table-level locking, which can lead to performance issues in high-concurrency environments.

6. **Replication**: If you plan to use replication in your MySQL environment, InnoDB may be a better choice. InnoDB provides support for row-level replication, which can reduce the amount of data that needs to be transferred between replicas and improve replication performance.

7. **Backup and recovery**: InnoDB provides better support for backup and recovery compared to MyISAM, due to its transactional model and support for incremental backups.

8. **Compatibility with other databases**: If you need to integrate your MySQL database with other databases that support full-text indexing, you may want to consider MyISAM, as it uses a similar syntax for full-text indexing as other databases such as PostgreSQL and SQLite.

9. **Future development**: InnoDB is the recommended storage engine for modern versions of MySQL, and is likely to receive more attention and development in the future. MyISAM, on the other hand, is considered to be a legacy storage engine and may not receive as much support or development going forward.

Ultimately, the choice between InnoDB and MyISAM for full-text indexing in MySQL depends on your specific requirements and priorities, including factors such as query performance, data consistency, transactional support, concurrent access, replication, backup and recovery, compatibility with other databases, and future development prospects.

3.5.2 Full Text Indexing in MySQL

MySQL provides various search capabilities, including full-text indexing, which enables faster and more precise text search within large datasets. By creating a full-text index on a table column, MySQL can match and rank search results based on their relevance to the search query. In this context, we explore how to create a full-text index on a MySQL database table and how to use the MATCH AGAINST statement to search for text within the indexed columns. We also examine how to refine our search using IN BOOLEAN MODE or WITH QUERY EXPANSION options to find more precise and relevant results.

We will try to provide by examples how we can create full text indexing in MySQL providing steps. Therefore, let's say we have a database of blog posts with the following table:

```
CREATE TABLE blog_posts (
    post_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    author_id INT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

To create a full text index on the title and content columns, we can use the following SQL command:

```
ALTER TABLE blog_posts
ADD FULLTEXT(title, content);
```

This creates a full text index on the title and content columns of the blog_posts table. Now we can use the MATCH AGAINST statement to search for text within these columns.

For example, let's say we want to search for blog posts that mention the phrase "machine learning." We can use the following query:

```
SELECT post_id, title, author_id, created_at
FROM blog_posts
WHERE MATCH(title, content) AGAINST('machine learning');
```

This query will return all blog posts that mention the phrase "machine learning" in the **title** or **content** columns. Note that the MATCH AGAINST statement will return a relevance score for each result, based on how well it matches the search query.

We can also use the IN BOOLEAN MODE or WITH QUERY EXPANSION options to refine our search further. For example, let's say we want to search for blog posts that mention either "machine learning" or "artificial intelligence." We can use the following query with IN BOOLEAN MODE:

```
SELECT post_id, title, author_id, created_at
FROM blog_posts
WHERE MATCH(title, content) AGAINST('+machine +learning OR +artificial
    +intelligence' IN BOOLEAN MODE);
```

This query will return all blog posts that mention either ”machine learning” or ”artificial intelligence” in the **title** or **content** columns. Note that we’re using the + symbol to indicate that both keywords must appear in the result.

We can also use the **WITH QUERY EXPANSION** option to search for related keywords or phrases. For example, let’s say we want to search for blog posts that mention ”machine learning,” but we’re also interested in related topics like ”deep learning” and ”neural networks.” We can use the following query with **WITH QUERY EXPANSION**:

```
SELECT post_id, title, author_id, created_at
FROM blog_posts
WHERE MATCH(title, content) AGAINST('machine learning' WITH QUERY EXPANSION);
```

This query will return all blog posts that mention ”machine learning,” as well as related topics identified by the search engine. Note that the WITH QUERY EXPANSION option can help us find relevant results that we might have missed with a more specific search. However, it’s worth noting that the results may also be less relevant or more broad.

Chapter 4

SQLite - Full text Indexting

SQLite's full-text search module is built upon the FTS3 and FTS4 virtual table modules, which provide efficient storage and indexing of text data [8]. These modules enable the creation of specialized tables for full-text searching, offering powerful features to enhance search capabilities. The full-text search module in SQLite incorporates advanced functionalities, including stemming support, result ranking, and term highlighting within the search results.

The FTS3 and FTS4 virtual table modules enable the creation of virtual tables dedicated to full-text search operations in SQLite [8]. These virtual tables differ from regular tables as they do not store data on disk directly. Instead, they store a tokenized version of the data in memory, leveraging it for efficient searching operations [8].

Both FTS3 and FTS4 modules offer similar functionality, but FTS4 is the newer and more advanced version. These modules provide support for tokenizing text data, stemming, and ranking search results. They also offer advanced features such as phrase searching, prefix searching, and proximity searching, enabling users to fine-tune their search queries.

To create an FTS3 or FTS4 virtual table in SQLite, the schema of the table needs to be specified using the CREATE VIRTUAL TABLE statement. Additionally, the name of the tokenizer, which determines how the text data is tokenized, must be provided. SQLite offers various built-in tokenizers, including the simple tokenizer, which splits text into individual words based on whitespace, and the porter tokenizer, which applies the Porter stemming algorithm to normalize words.

Here's an example of creating an FTS4 virtual table:

```sql
CREATE VIRTUAL TABLE mytable USING FTS4(content);
```

This creates a virtual table called ”mytable” with a single column called ”content”, which will be used to store the text data.

Once you've created an FTS3 or FTS4 virtual table, you can perform full text search operations on it using the MATCH operator. Here's an example of a simple full text search query:

```sql
SELECT * FROM mytable WHERE content MATCH 'search term';
```

This will return all rows from the ”mytable” virtual table where the ”content” column contains the phrase ”search term”.

In general, to use the full-text search module in SQLite, you first need to create a table with the FTS3 or FTS4 module. This table will contain the text data that you want to search. You can then insert text data into the table, and use the MATCH operator to perform searches. The MATCH operator allows you to specify the search term or terms that you want to search for. SQLite's full-text search module supports several search modes, including simple, phrase, and proximity searching.

4.1 Basic Full text searching

Full-Text Search (FTS) in SQLite offers a comprehensive set of search functionalities that simplify text searches in various ways. It provides support for stemming, allowing the recognition of different forms of a word and treating them as the same for searching purposes [8]. This ensures that users can search for a term and find all related variations of that term. FTS also supports phrase searching, enabling users to search for a specific sequence of words in a particular order. This feature is particularly useful when searching for exact phrases or specific combinations of words.

Additionally, FTS offers proximity searching, which allows users to search for words that are within a certain distance of each other [8]. This feature is beneficial when users need to find information that is contextually related or when searching for terms that often occur together [8]. Furthermore, FTS supports synonym searching, enabling users to search for words with similar meanings. This feature expands the search scope by including related terms, thereby improving the search experience and increasing the chances of finding relevant information [8].

One of the key strengths of FTS in SQLite is its ranking algorithm, which determines the relevance of search results based on various factors such as word frequency and proximity. This ranking ensures that the most pertinent results are displayed prominently, making it easier for users to quickly find the information they are seeking.

These features include phrase searching, which enables users to search for a specific sequence of words in a specific order. Additionally, stemming allows users to search for different variations of a word using a root form of the word, making it easier to find all related results without having to search for each variation of the word separately. Ranking is another key feature, which orders search results based on their relevance to the search query. Boolean operators such as ”and”, ”or”, and ”not” can be used to combine search terms, while synonym searching enables users to find results that include either the search term or a synonym of the search term. Finally, stopwords are common words that are excluded from full text searching because they do not add value to the search results. By combining all of these features, full text searching in SQLite offers a powerful and efficient solution for searching through large amounts of text data.

1. **Phrase Searching**:

```
SELECT * FROM mytable WHERE content MATCH '"exact phrase"';
```

2. **Stemming**:

```
SELECT * FROM mytable WHERE content MATCH 'stems OR stemmed OR stemming';
```

3. **Ranking**:

```
SELECT * FROM mytable WHERE content MATCH 'search query' ORDER BY rank
    DESC;
```

4. **Boolean Operators**:

```
SELECT * FROM mytable WHERE content MATCH 'word1 AND word2 OR word3 NOT
    word4';
```

5. **Synonyms**:

```
CREATE VIRTUAL TABLE mytable USING fts5(content, tokenize=simple,
    prefix='2 3 4');
INSERT INTO mytable(content) VALUES('important crucial critical');
SELECT * FROM mytable WHERE content MATCH 'important OR crucial OR
    critical';
```

6. **Stopwords**:

```
CREATE VIRTUAL TABLE mytable USING fts5(content, tokenize=porter,
    stopwords='the a an');
INSERT INTO mytable(content) VALUES('The quick brown fox jumps over the
    lazy dog');
SELECT * FROM mytable WHERE content MATCH 'quick fox jumps';
```

4.2 Advanced Full text searching

In addition to the fundamental full-text search functionality, SQLite's FTS5 module offers a range of advanced capabilities that enable users to perform more precise and targeted searches. These features enhance the flexibility and accuracy of the search process, allowing users to find the desired data more efficiently.

FTS5 Prefix Queries are designed to search for words that begin with a specific prefix. This functionality is particularly useful when users want to find all the words in a document or database that start with a certain set of characters. It provides a convenient way to filter and retrieve words based on their initial characters.

FTS5 Initial Token Queries, on the other hand, allow users to search for words that start with a particular letter or character [8]. This feature is beneficial when users need to locate words or terms that begin with a specific alphabet letter or special character. It enables targeted searches that focus on a specific subset of words.

FTS5 NEAR Queries enable users to search for words that are in close proximity to each other within a document. This functionality is valuable for scenarios where the relationship or context between words matters. By specifying a proximity constraint, users can retrieve results that reflect the spatial or temporal relationship between the search terms.

Furthermore, FTS5 Column Filters provide the ability to filter search results based on specific columns in the virtual table. This allows users to narrow down their search to specific attributes or properties associated with the text data. By utilizing column filters, users can refine their search results to match specific criteria or conditions.

These advanced capabilities offered by SQLite's FTS5 module significantly enhance the search experience for users. They provide more precise control over the search process and enable users to locate the data they need more efficiently. By leveraging these features, users can save time and effort by retrieving highly relevant and targeted search results.

4.2.1 FTS5 Prefix Queries

FTS5 Prefix Queries is a feature of SQLite's Full-Text Search module that allows you to search for words in a document that start with a specific prefix. This can be useful in situations where you want to search for words that have a common root, such as ”automobile”, ”automotive”, and ”automation”.

With FTS5 Prefix Queries, you can specify a prefix followed by the wildcard character (*), which matches any character sequence. For example, to search for all words in a document that start with ”auto”, you can use the query ”auto*”. This will match words like ”automobile”, ”automation”, ”autonomous”, and so on.

Here's an example of how to use FTS5 Prefix Queries in SQLite:

Suppose you have a virtual FTS5 table named ”mytable” with a column named ”content” that contains various documents. You want to find all documents that contain at least one word starting with the prefix ”auto”. Here's the SQL query you can use:

```
SELECT * FROM mytable WHERE content MATCH 'auto*';
```

In this example, we're using the MATCH keyword to perform the FTS5 Prefix Query. The query will return all rows where the "content" column matches a word that starts with "auto".

You can also combine FTS5 Prefix Queries with other search conditions, such as searching for documents that contain multiple prefixes or specific keywords. For example:

```
SELECT * FROM mytable WHERE content MATCH 'auto*' AND content MATCH 'car';
```

This query will return all rows where the "content" column matches a word that starts with "auto" and also contains the word "car".

In the below example, we've added an additional condition to the WHERE clause using the FTS5 Column Filter "address MATCH 'city:New York'" to filter the search results based on the value of the "address" column. We've also added the FTS5 Prefix Query "auto*" to search for words that start with the prefix "auto".

```
SELECT * FROM mytable
WHERE content MATCH ?||'' AND address MATCH 'city:New York' AND content MATCH
    'auto'
ORDER BY rank LIMIT 10;
```

Note that we've used a parameter marker "?" in the FTS5 Prefix Query, which means that we can pass in the prefix string as a parameter when executing the query. This can be useful if you want to search for different prefixes without having to modify the SQL query itself.

4.2.2 FTS5 Initial Token Queries

FTS5 Initial Token Queries allow you to search for words that start with a specific letter or character. This can be useful if you want to find all words in your text data that begin with a certain letter or character, without having to specify a complete prefix.

To use FTS5 Initial Token Queries in SQLite, you can use the "^" character followed by the letter or character you want to match. Here's an example SQL query that searches for all words in a virtual FTS5 table named "mytable" that start with the letter "a":

```
SELECT * FROM mytable
WHERE content MATCH '^a'
ORDER BY rank LIMIT 10;
```

In this example, we've used the FTS5 Initial Token Query "^a" in the WHERE clause to search for all words in the "content" column that start with the letter "a". The "*" character is not necessary in this case, since we're only looking for words that start with "a" and don't care about the rest of the word.

Here's an example that searches for all documents that contain the phrase "apple pie" and at least one word that starts with the letter "c" :

```
SELECT * FROM mytable
WHERE content MATCH '("apple pie") INTERSECT (content:c^)'
ORDER BY rank LIMIT 10;
```

In this example, we've used the FTS5 Phrase Query "apple pie" to search for documents that contain the exact phrase "apple pie". We've also used the FTS5 Prefix Query content:c* to search for words that start with the letter "c". The INTERSECT operator is used to combine the two conditions, so we're only returning documents that meet both criteria. By using FTS5 Compound Query feature we can create more complex queries that are easier to read and maintain.

4.2.3 FTS5 NEAR Queries

FTS5 NEAR Queries allow you to search for words that appear near each other in a document. This can be useful if you want to find documents where certain words are mentioned in close proximity to each other.

To use FTS5 NEAR Queries in SQLite, you can use the ”NEAR” operator followed by a number indicating the maximum distance between the two words you want to match. Here’s an example SQL query that searches for all documents in a virtual FTS5 table named ”mytable” where the words ”apple” and ”pie” appear within 5 words of each other:

```
SELECT * FROM mytable
WHERE content MATCH 'apple NEAR/5 pie'
ORDER BY rank LIMIT 10;
```

In this example, we’ve used the FTS5 NEAR Query ’apple NEAR/5 pie’ in the WHERE clause to search for all documents where the words ”apple” and ”pie” appear within 5 words of each other. The ”/” character is used to separate the NEAR operator from the maximum distance (in this case, 5 words).

We can use FTS5 Matchinfo function to calculate the distance between two search terms and filter the results based on that distance. Here’s an example that searches for all documents that contain the phrase ”apple pie” within 10 words of the word ”recipe”:

```
SELECT * FROM mytable
WHERE content MATCH 'apple NEAR/10 pie NEAR/10 recipe'
AND matchinfo(mytable, 'pcnalx') LIKE '%2__2__%'
ORDER BY rank LIMIT 10;
```

In this example, we’ve used the FTS5 Query ’apple NEAR/10 pie NEAR/10 recipe’ to search for documents that contain the word ”apple” within 10 words of the word ”pie”, which in turn is within 10 words of the word ”recipe”. We’ve used FTS5 Matchinfo function with the ’pcnalx’ option to calculate the distance between the three search terms in terms of the number of words between them. The LIKE clause is used to filter the results and only return documents where the distance between the first two search terms is 2 or less, and the distance between the last two search terms is also 2 or less, which means the word ”recipe” appears within 10 words of the phrase ”apple pie”. By using the FTS5 Matchinfo function instead of Boolean operators or FTS5 Compound Query, we can create more complex queries that are more efficient and faster to execute.

4.2.4 FTS5 Column Filters

FTS5 Column Filters allow you to filter search results based on specific columns in the virtual table. This can be useful if you have a large dataset with multiple columns, and you only want to search for matches in certain columns.

To demonstrate this, suppose we have a database table called **articles** with three columns: **id**, **title**, and **content**. We want to search for articles that contain the word ”database” in the **content** column, but we only want to see the **id** and **title** columns in the search results.

Here’s an example query:

```
SELECT id, title FROM articles WHERE content MATCH 'database' ORDER BY rank;
```

In this query, we are using the FTS5 MATCH operator to search for the word ”database” in the content column of the articles table. We then specify that we only want to see the id and title columns in the search results by including them in the SELECT clause.

Note that the **rank** column is still included in the query, as it is needed to order the search results. If you don’t need to order the search results, you can omit the ORDER BY clause.

You can also use the **MATCH** operator in conjunction with other SQL operators, such as **AND**, **OR**, and **NOT**, to create more complex search queries that filter results based on multiple columns. Here's an example:

```
SELECT id, title FROM articles WHERE content MATCH 'database' AND title MATCH
    'management' ORDER BY rank;
```

In this query, we are searching for articles that contain the word "database" in the content column and the word "management" in the title column. The search results will only include the id and title columns, and will be ordered by relevance (rank).

4.3 Auxiliary Functions

SQLite provides several auxiliary functions that can be used in conjunction with the full text searching capabilities. Here are some of them:

1. **snippet()** - Returns a snippet of text that contains the search term.

   One useful auxiliary function provided by SQLite for full text searching is the **snippet()** function. This function returns a snippet of text that contains the matching search terms. The function takes three arguments: the name of the FTS5 table, the rowid of the document to retrieve a snippet from, and optional start and end strings to wrap around the matching terms.

   Here's an example query that uses the **snippet()** function:

   ```
   SELECT snippet(mytable, 1, '<strong>', '</strong>') AS snippet_text
   FROM mytable
   WHERE mytable MATCH 'search term';
   ```

   This query returns a snippet of text from the document with a rowid of 1 that contains the matching search term. The matching terms are wrapped in **** and **** tags to highlight them.

   The **snippet()** function is useful for presenting search results to users in a way that highlights the relevant content.

2. **offsets()** - Returns the byte offsets of the search term within the text.

   The offsets() function in SQLite is used to retrieve the byte offsets of the full text matches in the search result. This can be useful when you want to highlight the matched text in your application or perform some other processing on the matched text.

   ```
   SELECT content, offsets(articles) FROM articles WHERE content MATCH
       'database';
   ```

   This query will return a result set with two columns - content and offsets(articles). The content column will contain the matched text and the offsets(articles) column will contain a comma-separated list of byte offsets for each match. For example, if there are two matches in a row with byte offsets 56 and 78, the offsets(articles) value will be "56, 78". You can use this information to highlight the matched text in your application or perform other processing on the matched text.

3. **bm25()** - Calculates the BM25 ranking score for a search result.

   bm25() is a built-in SQLite auxiliary function that is used to calculate relevance scores for full-text searches. It implements the BM25 ranking algorithm, which is a popular method for determining the relevance of documents in information retrieval systems.

The function takes two arguments: the frequency of a term in the document, and the number of documents in the corpus that contain the term. It returns a relevance score that takes into account both the term frequency and the document frequency.

Here is an example of how to use **bm25()** in a full-text search query:

```
SELECT title, bm25(mytable) as relevance
FROM mytable
WHERE mytable MATCH 'search query'
ORDER BY relevance DESC;
```

In this example, the **bm25()** function is used to calculate the relevance score for each document in the **mytable** virtual table that matches the search query. The results are then ordered by relevance in descending order, so that the most relevant documents appear first.

Note that **bm25()** is an auxiliary function, which means that it needs to be registered with SQLite using the **sqlite3_create_function()** API function before it can be used in SQL queries.

4. **highlight()** - Returns the text with HTML tags added to highlight the search term.

   The **highlight()** function is an auxiliary function in SQLite's FTS5 module that is used to add HTML tags to highlight the matching words in the search results. It takes three arguments: the column name, the start tag (e.g., ****), and the end tag (e.g., ****).

   Here's an example of how to use the **highlight()** function:

   Assume we have a table named **documents** with an FTS5 virtual table named **doc_fts** that contains a column named **content** which stores the document content.

   To search for the word ”database” and highlight the matching words in the search results, we can use the following query:

```
SELECT highlight(documents, 'content', '<b>', '</b>') AS
    highlighted_content
FROM doc_fts
WHERE content MATCH 'database';
```

   This query returns a single column **highlighted_content**, which contains the highlighted search results with the matching words wrapped in **** and **** tags.

   The **highlight()** function can be used in combination with other FTS5 functions and operators to build more complex queries that return highlighted search results.

5. **matchinfo()** - Returns information about the search term's occurrence within the text.

   matchinfo() is an auxiliary function in SQLite's FTS5 module that returns information about the matches found in the full-text search.

   The **matchinfo()** function takes the following parameters:

   - The name of the FTS5 table or virtual table.
   - The optional column number to retrieve match information for (defaults to 0).
   - The optional matchinfo command to execute (defaults to 0).

   The matchinfo command parameter is an integer that determines the type of information returned by the function. There are several commands available, each of which returns a different set of information. Here are some examples:

- matchinfo('table'): Returns an array of integers with information about each match, including the number of occurrences and offsets.
- matchinfo('table', 1): Returns an array of integers with information about each match for the second column in the FTS5 table.
- matchinfo('table', 0, 'bm25'): Returns an array of integers with information about each match using the BM25 ranking function.

Here's an example of using matchinfo() to retrieve information about the matches found in a full-text search:

```
CREATE VIRTUAL TABLE documents USING fts5(title, content);

INSERT INTO documents(title, content) VALUES ('Document 1', 'This is the
    content of document 1.');
INSERT INTO documents(title, content) VALUES ('Document 2', 'This is the
    content of document 2.');
INSERT INTO documents(title, content) VALUES ('Document 3', 'This is the
    content of document 3.');

SELECT * FROM documents WHERE documents MATCH 'content';
```

This query will return all three documents since they all contain the word "content". Now let's add the matchinfo() function to retrieve information about the matches:

```
SELECT matchinfo('documents') FROM documents WHERE documents MATCH
    'content';
```

This query will return an array of integers for each document, with information about the matches found. The array contains pairs of integers representing the number of occurrences and offsets for each match. For example, if there are two occurrences of the word "content" in a document, the array will contain the integers **(2, offset1, offset2)**. The exact format of the array depends on the matchinfo command used.

Using **matchinfo()** can be helpful when analyzing search results and implementing custom ranking algorithms.

6. **rank()** - Returns the relevance ranking score for a search result.

   rank() is a built-in function in SQLite's FTS5 module that is used to calculate a ranking score for each match in the search results. The ranking score is based on how well the document matches the search query, with higher scores indicating a better match.

   The **rank()** function takes in a single argument, which is the matchinfo blob returned by the **matchinfo()** function. This blob contains information about the matches in the search results, such as the number of times the search terms appear in each match and the positions of those occurrences.

   Here's an example of using **rank()** to calculate the ranking score for each match in the search results:

```
SELECT *, rank(matchinfo(mytable)) AS score
FROM mytable
WHERE content MATCH 'example query'
ORDER BY score DESC;
```

In this example, we're selecting all columns from the **mytable** virtual table, using the **MATCH** operator to search for the phrase "example query" in the **content** column. We're then using

the **rank()** function to calculate the ranking score for each match, and aliasing the result as **score**.

Finally, we're ordering the results by the score column in descending order, so that the matches with the highest ranking score appear first.

4.4 Full text indexing in SQLite

Full text indexing in SQLite offers an efficient solution for searching words or phrases within extensive text data stored in the database [8]. SQLite's implementation of full text indexing is built on the FTS5 (Full-Text Search version 5) extension, which provides a robust and flexible mechanism for indexing and searching text data [13]. By creating a full text index on one or more columns in a table, you can significantly accelerate text-based searches compared to traditional methods like using the LIKE operator or regular expressions [9].

The FTS5 extension in SQLite is designed to handle large volumes of text data and provides enhanced performance for search operations [8]. By leveraging full text indexing, applications that require searching through substantial amounts of text data, such as search engines, e-commerce websites, or content management systems, can benefit from significantly faster search capabilities [41].

By creating a full text index, SQLite organizes and structures the text data in a way that enables efficient searching. This allows you to perform queries that involve complex search conditions or require proximity-based matching. The indexing process optimizes the search performance by pre-processing the text data and creating an index structure that facilitates fast retrieval of relevant information.

Compared to traditional approaches like the LIKE operator or regular expression searches, full text indexing in SQLite provides a more streamlined and efficient solution for text-based searches. It eliminates the need for manual pattern matching and enables developers to focus on creating powerful search functionalities without compromising performance.

Here is a step-by-step example of how to create a full text index in SQLite:

1. First, you need to make sure that the FTS5 extension is enabled in SQLite. You can do this by running the following command when you connect to the database:

   ```
   SELECT load_extension('fts5');
   ```

2. Next, create a table that you want to create a full text index on. For example, let's say you have a table called my_table with the following schema:

   ```
   CREATE TABLE my_table (
       id INTEGER PRIMARY KEY,
       name TEXT,
       description TEXT
   );
   ```

   This table has two columns, name and description, and an integer primary key column id.

3. Now, create a virtual table that will store the full text index. You can do this by running the following command:

   ```
   CREATE VIRTUAL TABLE my_table_fts USING fts5(name, description);
   ```

   This will create a new virtual table called my_table_fts that has the same columns as my_table and a full text index on the name and description columns.

4. Copy the data from my_table to my_table_fts. You can do this by running the following command:

```
INSERT INTO my_table_fts(docid, name, description) SELECT id, name,
    description FROM my_table;
```

   This will copy all the data from my_table to my_table_fts and create the full text index.

   Note that we're using docid instead of rowid as the first column in the INSERT INTO statement. This is because FTS5 requires a document ID column to be present in the virtual table, and docid is the default name for this column.

5. You can now use the full text index in your queries. For example, if you want to search for rows in my_table that contain the word "example" in either the name or description column, you can run the following command:

```
SELECT * FROM my_table WHERE id IN (SELECT docid FROM my_table_fts WHERE
    my_table_fts MATCH 'example');
```

   This will return all the rows from my_table that contain the word "example" in either the name or description column.

Full text indexing is an important feature of SQLite that allows you to efficiently search for text data within your database. By creating a full text index on one or more columns in a table, you can perform fast and accurate text-based searches on large amounts of data, which can be particularly useful for applications that need to search through large amounts of text data. SQLite's implementation of full text indexing with the FTS5 extension provides a powerful and flexible way to index and search text data, making it an ideal choice for a wide range of applications that deal with text data. Whether you're building a search engine, an e-commerce website, or a content management system, full text indexing in SQLite can help you provide a more efficient and effective search experience for your users.

Chapter 5

Comparison Analysis - Word Phrase Searching

5.1 Methodology

PostgreSQL, MySQL, and SQLite are widely used database management systems known for their robustness and versatility. When it comes to implementing full text searching capabilities, these systems offer various features and techniques to deliver accurate search results. Full text searching plays a crucial role in modern applications where users rely on efficient and precise retrieval of information. Whether it's a content management system, e-commerce platform, or search engine, the ability to find relevant data quickly and accurately is of utmost importance. This comparison analysis focuses on evaluating and comparing the accuracy of word and phrase search results obtained from PostgreSQL, MySQL, and SQLite, with a specific emphasis on the F1 Score.

The F1 Score is a widely accepted evaluation metric that combines precision and recall to provide a comprehensive measure of search result accuracy. It takes into account both the relevance of the retrieved documents (precision) and the completeness of the retrieved documents (recall). By focusing on the F1 Score, this analysis aims to provide a balanced evaluation of the accuracy of word and phrase search results obtained from PostgreSQL, MySQL, and SQLite.

PostgreSQL, MySQL, and SQLite each have their own mechanisms for handling full text search operations, including word and phrase searches. They employ different algorithms, indexing structures, and query optimization strategies to enhance search accuracy specifically for words and phrases. Understanding how these systems perform in terms of accurately retrieving specific words and phrases based on the F1 Score metric is vital for developers, administrators, and organizations making decisions about their database technology.

This comparison analysis aims to explore various factors that can impact the accuracy of word and phrase search results. These factors include the ability to handle complex queries, support for features such as stemming and synonym matching, relevance ranking algorithms, the effectiveness of search indexes specifically for word and phrase queries, and their overall impact on the F1 Score.

By evaluating the word and phrase search capabilities of these database systems based on the F1 Score, this analysis will provide valuable insights to developers and decision-makers. Understanding the accuracy of search results offered by PostgreSQL, MySQL, and SQLite specifically for words and phrases, as measured by the F1 Score, can help organizations choose the most suitable system based on their specific requirements, performance expectations, and the nature of their data.

In the following sections, we will delve into the specific aspects of word and phrase search accuracy and compare how PostgreSQL, MySQL, and SQLite perform in each area. Through a

comprehensive assessment of their features, techniques, and performance benchmarks, we aim to provide a thorough understanding of the accuracy of word and phrase search results delivered by these database systems, as measured by the F1 Score.

5.1.1 Description of the dataset

To conduct a comprehensive comparison analysis of the accuracy of word and phrase search results in PostgreSQL, MySQL, and SQLite, we will utilize a dataset consisting of movies with 10,000 records. The dataset includes several relevant attributes that provide context and contribute to the accuracy of the search results.

The table structure for this analysis will be defined as follows:

```
CREATE TABLE movies10000 (
  title VARCHAR(255),
  overview TEXT,
  original_language VARCHAR(50),
  vote_count INT,
  vote_average DECIMAL(5, 2)
);
```

Columns:

- **title**: This column will store the title of each movie in the dataset. The title serves as a unique identifier for each movie and allows users to search for movies based on their specific titles
- **overview**: The overview column will contain a summary or description of each movie. It provides valuable context for the content of the movie and serves as the primary source for the full text search. Full text indexing will be performed on this column, enabling efficient and accurate retrieval of movies based on their content.
- **original_language**: This column will capture the original language in which each movie was produced. Language can be a significant factor in search queries, especially for multilingual applications or users searching for movies in a specific language.
- **vote_count**: The vote_count column will store the number of votes received by each movie. Although not directly related to the full text search, it can provide additional insights into the popularity and relevance of movies when combined with search results.
- **vote_average**: The vote_average column will record the average rating assigned to each movie by users or critics. Similar to the vote_count, it can offer additional relevance indicators for search results.

By incorporating these attributes into the analysis, we aim to evaluate the accuracy of word and phrase search results in PostgreSQL, MySQL, and SQLite in a realistic and meaningful context. The overview column, in particular, will serve as the focal point for full text indexing and retrieval, as it encapsulates the content and essence of each movie.

Through the comparison of these three database systems, we seek to provide insights into the effectiveness of their full text indexing capabilities for word and phrase searches. By analyzing the accuracy of search results based on the provided dataset and these specific attributes, we aim to determine which system performs best in terms of accurately retrieving movies that match the search criteria.

In the subsequent sections of the analysis, we will delve into the implementation details of full text indexing in PostgreSQL, MySQL, and SQLite, perform benchmark tests, and assess the accuracy of word and phrase search results based on the F1 Score metric. The findings will shed

light on the strengths and limitations of each system, aiding developers and decision-makers in choosing the most suitable database system for their specific full text search requirements.

5.1.2 Overview of full-text searching

In this comparison analysis, we will be examining the full text searching capabilities of three popular database management systems: PostgreSQL, MySQL, and SQLite. Full text searching is a powerful feature that allows for efficient and accurate searching of textual data within database tables. By utilizing this feature, we can perform searches based on word and phrase matching to retrieve relevant records.

- **PostgreSQL**, basic full text searching queries can be performed using the to_tsvector and to_tsquery functions:

  Word Search:

  ```
  SELECT * FROM movies
  WHERE to_tsvector('english', overview) @@ to_tsquery('english', 'word');
  );
  ```

  Phrase Search:

  ```
  SELECT * FROM movies
  WHERE to_tsvector('english', overview) @@ to_tsquery('english', 'phrase
      search');
  );
  ```

- In **MySQL**, basic full text searching queries can be performed using the MATCH...AGAINST syntax. For word searching, we can use the MATCH...AGAINST syntax with the IN NATURAL LANGUAGE MODE option

  Word Search:

  ```
  SELECT * FROM movies
  WHERE MATCH(overview) AGAINST ('word' IN NATURAL LANGUAGE MODE);
  ```

  Phrase Search:

  ```
  SELECT * FROM movies
  WHERE MATCH(overview) AGAINST ('"+phrase +search"' IN BOOLEAN MODE);
  ```

- In **SQLite**, full text searching can be enabled using the FTS (Full Text Search) extension.

  Word Search:

  ```
  SELECT * FROM movies_fts
  WHERE overview MATCH 'word';
  ```

  Phrase Search:

  ```
  SELECT * FROM movies_fts
  WHERE overview MATCH '"phrase search"';
  ```

In addition to the native full text searching capabilities provided by each system, we will also explore the use of traditional search functions like LIKE to add multiple variations, synonyms, and other search criteria. This will allow us to compare the results obtained from the traditional search approach with the corresponding queries executed in each system's full text search functionality.

Therefore, In this analysis, we will focus on the basic full text searching capabilities of each system and compare them with the results obtained from the traditional search approach. By evaluating the accuracy, relevance of these searches, we aim to gain a comprehensive understanding of the capabilities of PostgreSQL, MySQL, and SQLite when it comes to efficient and accurate text searching.

5.1.3 F1 Score methodology

When comparing the F1 Score in the context of full text search capabilities, we are evaluating the performance of different systems or models in retrieving relevant results based on their search capabilities. The F1 Score allows us to assess the effectiveness and balance between precision and recall for each system, providing insights into their search accuracy and comprehensiveness.

Let's consider a scenario where we have three systems: System A (e.g., PostgreSQL), System B (e.g., MySQL), and System C (e.g., SQLite), all equipped with their respective full text search capabilities. We perform a comparison analysis to understand how well each system performs in retrieving relevant results.

To evaluate the full text search capabilities, we utilize the F1 Score as a metric. We define a set of search queries or criteria that include multiple variations, synonyms, and relevant terms. These queries are designed to test the ability of each system to retrieve accurate and comprehensive results.

For each system, we execute the queries using traditional search functions like LIKE, along with the necessary variations and synonyms, to mimic the behavior of the full text search capabilities of the respective system. We retrieve a set of documents or records from each system based on these queries.

Next, we determine the true positives, false positives, and false negatives for each system. True positives represent the number of relevant documents or records correctly retrieved by the system. False positives indicate the number of irrelevant documents or records incorrectly retrieved, and false negatives represent the relevant documents or records that were not retrieved by the system.

Using these values, we calculate the precision and recall for each system. Precision measures the accuracy of the retrieved results, while recall measures the completeness of the retrieved results.

Finally, we compute the F1 Score for each system by taking the harmonic mean of precision and recall. The F1 Score provides a comprehensive assessment of the search capabilities, taking into account both the accuracy and completeness of the retrieved results.

By comparing the F1 Scores of System A, System B, and System C, we can determine which system demonstrates superior full text search capabilities in terms of accuracy and comprehensiveness. A higher F1 Score indicates a better balance between precision and recall, reflecting a more effective and reliable system for retrieving relevant results.

This comparison perspective based on the F1 Score allows us to objectively evaluate the full text search capabilities of different systems, enabling us to identify the strengths and weaknesses of each system and make informed decisions about their suitability for specific search requirements or applications.

5.2 Intro of the analysis

In this analysis, we will evaluate the precision and recall of three different database systems, namely PostgreSQL, MySQL, and SQLite, in retrieving relevant documents from a movies dataset. The goal is to assess the effectiveness of these systems' full-text search capabilities when searching for commonly used words and phrases related to movie genres, such as drama, comedy, action, and more. To conduct this analysis, we will utilize a set of carefully chosen words and phrases that are frequently employed by users when searching for movies. These terms represent a wide range

of genres and encompass common variations, synonyms, and related words associated with each genre. By utilizing the LIKE function in SQL, we can construct queries that capture multiple variations and synonyms for a given genre, enabling us to compare the results with the full-text searching capabilities offered by the database systems.

The analysis will focus on measuring two key metrics: precision and recall. Precision refers to the proportion of retrieved documents that are relevant, while recall represents the proportion of relevant documents that are successfully retrieved. By calculating these metrics for each system and comparing the results, we can gain insights into the effectiveness of their search capabilities when confronted with a diverse range of user queries.

The Precision is calculated using the formula:

$$Precision = (retrieved_relevant_documents)/(retrieved_documents) \tag{5.1}$$

The Recall is calculated using the formula:

$$Recall = (retrieved_relevant_documents)/(total_relevant_documents) \tag{5.2}$$

It is important to note that in this analysis, we will primarily use the LIKE function in conjunction with the OR statement to encompass various variations, synonyms, and related terms for each genre. This approach allows us to simulate a comprehensive search strategy commonly employed by users seeking movies within a specific genre. By comparing the results obtained through this method with the full-text search capabilities of PostgreSQL, MySQL, and SQLite, we can make informed conclusions regarding their respective strengths and weaknesses in retrieving relevant movie documents based on genre-related queries.

Through this analysis, we aim to provide a comprehensive understanding of the precision and recall performance of PostgreSQL, MySQL, and SQLite in the context of full-text searching for movies. By evaluating their effectiveness in retrieving relevant documents for common genre-related queries, we can guide users and developers in selecting the most suitable database system for their specific requirements.

In addition to evaluating precision and recall, we will also calculate the F1 score at the end of the analysis for both word and phrase searches. The F1 score is a widely used metric that combines precision and recall into a single value, providing a more comprehensive measure of the overall performance of each database system's full-text search capabilities.

The F1 score is calculated using the formula:

$$F1Score = 2 * (Precision * Recall)/(Precision + Recall) \tag{5.3}$$

By considering both precision and recall simultaneously, the F1 score allows us to assess the balance between the completeness and correctness of the retrieved results. A higher F1 score indicates a better overall performance, with a balance between precision and recall.

By calculating the F1 score for both word and phrase searches, we can further compare the effectiveness of each system in accurately retrieving relevant documents based on different types of search queries. This comprehensive evaluation will provide valuable insights into the overall performance of PostgreSQL, MySQL, and SQLite in meeting the information retrieval needs of users searching for movies within specific genres.

In summary, the analysis will not only encompass precision and recall calculations but will also incorporate the F1 score to provide a holistic evaluation of the full-text search capabilities of each database system for both word and phrase searches. This approach will enable us to draw informed conclusions about the relative strengths and weaknesses of PostgreSQL, MySQL, and SQLite in accurately retrieving relevant movie documents based on genre-related queries.

5.2.1 Evaluating the search performance of three different systems - PostgreSQL, MySQL, and SQLite

In this section, we will focus on evaluating the search performance of three different systems - PostgreSQL, MySQL, and SQLite - in retrieving relevant documents for queries related to overview of movies. Our goal is to assess the accuracy and completeness of the search results provided by each system. To achieve this, we will first determine the set of relevant documents for each query based on our knowledge or expectations. These relevant documents represent the desired search results that we expect the systems to retrieve.

- Relevant documents for Word Search fun. As you can see, the ILIKE function includes not only the word that the user searches but also some synonyms of the word fun

```
SELECT * FROM movies10000 WHERE overview ILIKE overview ILIKE '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%' AS retrieved_relevant_documents
```

- Relevant documents for Phrase Search like Romance and Music (synonyms and variations)

```
SELECT *
FROM movies10000
WHERE (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%');
```

We will proceed to calculate the precision, recall, and F1 score for each query. Precision measures the proportion of retrieved documents that are relevant, providing an indication of the system's accuracy in retrieving the desired results. Recall, on the other hand, assesses the system's ability to retrieve all relevant documents, thus capturing its completeness.

To obtain the precision and recall values, we will count the number of retrieved relevant documents and the total number of retrieved documents for each query. These values will then be used to calculate precision and recall using the respective formulas. By analyzing the precision and recall for each query, we can gain insights into the strengths and weaknesses of each system's search capabilities. In the below example, we use some sample search words and phrases as a reference

- Precision Query for Word fun
  - PostgreSQL

```
SELECT COUNT(*) AS retrieved_documents, SUM(CASE WHEN overview ILIKE
    '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%' THEN 1 ELSE 0 END) AS
    retrieved_relevant_documents
FROM movies10000
WHERE to_tsvector('english', overview) @@ to_tsquery('english',
    'fun');
```

– MySQL

```
SELECT COUNT(*) AS retrieved_documents, SUM(CASE WHEN overview ILIKE
    '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%' THEN 1 ELSE 0 END) AS
    retrieved_relevant_documents
FROM movies10000
WHERE MATCH(overview) AGAINST ('fun' IN NATURAL LANGUAGE MODE);
```

– SQLite

```
SELECT COUNT(*) AS retrieved_documents, SUM(CASE WHEN overview ILIKE
    '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%' THEN 1 ELSE 0 END) AS
    retrieved_relevant_documents
FROM movies10000
WHERE overview MATCH 'fun';
```

- Recall Query for Word fun

– PostgreSQL

```
SELECT COUNT(*) AS total_relevant_documents, SUM(CASE WHEN
    to_tsvector('english', overview) @@ to_tsquery('english', 'fun')
    THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE overview ILIKE '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%';
```

– MYSQL

```
SELECT COUNT(*) AS total_relevant_documents, SUM(CASE WHEN
    MATCH(overview) AGAINST ('fun' IN NATURAL LANGUAGE MODE) THEN 1
    ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE overview ILIKE '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%';
```

– SQLite

```
SELECT COUNT(*) AS total_relevant_documents, SUM(CASE WHEN overview
    MATCH 'fun' THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE overview ILIKE '%fun%'
OR overview ILIKE '%joy%'
OR overview ILIKE '%comedy%'
OR overview ILIKE '%laugh%';
```

- Precision Query for Phrase Search: music and romance

– PostgreSQL

```
SELECT COUNT(*) AS retrieved_documents,
  SUM(CASE WHEN (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%')
   THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE to_tsvector('english', overview) @@ to_tsquery('english',
    'music & romance');
```

– MySQL

```
SELECT COUNT(*) AS retrieved_documents,
  SUM(CASE WHEN (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%')
   THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE MATCH(overview) AGAINST ('+music +romance' IN BOOLEAN MODE);
```

– SQLite

```
SELECT COUNT(*) AS retrieved_documents,
  SUM(CASE WHEN (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%')
   THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE overview MATCH 'music romance';
```

- Recall Query for Phase Search: music and romance

– PostgreSQL

```
SELECT COUNT(*) AS total_relevant_documents,
  SUM(CASE WHEN to_tsvector('english', overview) @@
      to_tsquery('english', 'music & romance') THEN 1 ELSE 0 END) AS
      retrieved_relevant_documents
FROM movies10000
WHERE (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
```

```
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%');
```

– MYSQL

```
SELECT COUNT(*) AS total_relevant_documents,
  SUM(CASE WHEN MATCH(overview) AGAINST ('+music +romance' IN
      BOOLEAN MODE) THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000
WHERE (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%');
```

– SQLite

```
SELECT COUNT(*) AS total_relevant_documents,
  SUM(CASE WHEN overview MATCH 'music romance') THEN 1 ELSE 0 END)
      AS retrieved_relevant_documents
FROM movies10000
WHERE (overview ILIKE '%Music%'
OR overview ILIKE '%Song%'
OR overview ILIKE '%Melody%'
OR overview ILIKE '%Harmony%')
AND
(overview ILIKE '%Emotional%'
OR overview ILIKE '%Relationship%'
OR overview ILIKE '%Romance%');
```

Furthermore, we will calculate the F1 score, which combines precision and recall into a single metric. The F1 score provides a balanced measure of the system's overall search performance, taking into account both false positives and false negatives in the search results. It allows us to effectively compare the systems and identify the one that achieves the best balance between precision and recall. In the next session, we will provide the final results of the analysis based on the Word and Phrase queries using a set of 20 words (please see below the results for each word and phrase in more detail) and 18 phrases so that we can calculate the F1 Score and identify the most accurate system. Through this analysis, we aim to provide a comprehensive assessment of PostgreSQL, MySQL, and SQLite in terms of their ability to retrieve relevant documents for product description queries.

Words: drama, funny, entrepreneur, technology, mystery, fear, innovation, criminal, car, speed, computers, comedy, fantasy, adventure, romance, sports, science, animation, biography, musical

Phrases: drama and historical, drama and romantic, sports and comedy, technology and innovation, car and speed, business and man, horror and fear, adventure and biography, science and biography, musical and romance, fantasy and comedy, drama and adventure, science and adventure, romance and biography, sports and team, crime and war, family and school, action and fantasy

Please see in the below table the Precision and Recall results for words search as a reference

Word: Precision and Recall accordingly

Word	PostgreSQL	MySQL	SQLite
drama	1	1	1
funny	1	1	1
entepreneur	1	1	1
tecnhology	1	1	1
mystery	1	1	1
fear	1	1	1
innovation	1	1	1
criminal	1	1	1
car	1	1	1
speed	1	1	1
computers	0.08	1	1
comedy	1	1	1
fantasy	1	1	1
adventure	1	1	1
romance	1	1	1
sports	1	1	1
science	1	1	1
animation	0.38	1	1
biography	1	1	1
musical	1	1	1

Word	PostgreSQL	MySQL	SQLite
drama	0.67	0.65	0.65
funny	0.93	0.89	0.89
entepreneur	1	0.72	0.72
technology	0.96	0.77	0.77
mystery	1	0.19	0.19
fear	0.76	0.36	0.36
innovation	1	0.14	0.14
criminal	1	0.69	0.69
car	0.18	0.16	0.16
speed	0.14	0.05	0.05
computers	0.02	0.05	0.05
comedy	0.8	0.78	0.78
fantasy	0.59	0.43	0.43
adventure	0.93	0.59	0.59
romance	0.58	0.53	0.36
sports	0.34	0.16	0.16
science	0.16	0.15	0.15
animation	0.96	0.19	0.19
biography	0.18	0.16	0.02
musical	0.76	0.23	0.23

5.2.2 Experimental Results

Throughout this analysis, we have conducted an in-depth evaluation of the search capabilities of three popular database systems: PostgreSQL, MySQL, and SQLite. Our aim was to assess the precision and recall of each system's full-text search functionality when retrieving relevant documents for a set of 20 words and 18 phrases commonly used in movie search queries.

For each system, we executed the same set of queries related to movie descriptions and compared the search results obtained. By utilizing the traditional search function 'LIKE' with OR statements, we ensured that our queries encompassed multiple variations, synonyms, and related terms commonly used by users when searching for products.

In order to measure precision and recall, we followed a rigorous methodology. Firstly, we manually determined the set of relevant documents for each query based on our knowledge and expectations. These relevant documents represented the desired search results that we expected the systems to retrieve. We tried to use the traditional search function like LIKE adding multiple OR statements so that we can find out all the relevant words for the words.

Next, we executed the queries on each system and counted the number of retrieved relevant documents and the total number of retrieved documents. By applying the respective precision and recall formulas, we calculated the precision and recall values for each query. This allowed us to evaluate the accuracy and completeness of the search results provided by each system.

To obtain a comprehensive assessment of the search capabilities of each system, we also calculated the average precision and recall separately. This involved summing up the precision and recall values across all queries and dividing them by the total number of queries. These average values provided us with a more holistic view of the systems' overall performance in terms of accuracy and completeness.

These average values is denoted by the symbol $\sum$ and for a set of Precision and Recall values $x_1, x_2, \cdots, x_n$, their sum is defined as $\sum_{k=1}^{n} x_k$. That is

$$P = \frac{\sum_{k=1}^{20} Precision_k}{number_of_queries} = \frac{Precision_1 + Precision_2 + \cdots + Precision_{n-1} + Precision_n}{number_of_queries}$$

$$R = \frac{\sum_{k=1}^{20} Recall_k}{number_of_queries} = \frac{Recall_1 + Recall_2 + \cdots + Recall_{n-1} + Recall_n}{number_of_queries}$$

Finally, to consolidate the evaluation and provide a single metric for comparison, we calculated the F1 score. The F1 score combines the precision and recall values into a single measure, taking into account both false positives and false negatives. This score allowed us to assess and compare the effectiveness of the systems' search capabilities for the given set of words and phrases.

$$F1_score = 2 * \frac{P * R}{P + R}$$

By conducting this analysis and obtaining the precision, recall, average precision, and average recall values, as well as the F1 score for each system, we have gained valuable insights into their respective abilities to retrieve relevant documents in response to product search queries. These findings will serve as a basis for informed decision-making regarding the selection of the most suitable database system based on specific search requirements.

Now, let us delve into the detailed analysis and present the results of each system, highlighting their individual strengths and areas for improvement in terms of precision, recall, and overall search effectiveness.

- **F1 Score for Word Search**

  For PostgreSQL, the F1 score is 0.75. This indicates a relatively good balance between precision and recall, suggesting that PostgreSQL's full-text search capabilities for movie descriptions are effective in retrieving relevant documents. The system demonstrates a high level of precision, meaning that a significant portion of the retrieved documents are relevant to the search queries. However, the recall is somewhat lower, indicating that there is room for improvement in terms of capturing all the relevant documents available in the dataset.

On the other hand, MySQL and SQLite exhibit lower F1 scores of 0.56 and 0.54, respectively. This implies that these systems may have some limitations in terms of their search effectiveness compared to PostgreSQL. While both systems show a perfect precision score of 1.0, indicating that all retrieved documents are relevant to the queries, the recall values are considerably lower. This suggests that MySQL and SQLite may miss out on a significant number of relevant documents in the dataset.

System	Precision	Recall	F1 Score
PostgreSQL	0.92	0.64	0.75
MySQL	1	0.39	0.56
SQLite	1	0.37	0.54

- **F1 Score for Phrase Search**

  For PostgreSQL, the F1 score is 0.41. This indicates a moderate balance between precision and recall, suggesting that PostgreSQL's full-text search capabilities for phrases in movie descriptions have some limitations in retrieving relevant documents. The system demonstrates a moderate level of precision, meaning that a reasonable portion of the retrieved documents are relevant to the search phrases. However, the recall is relatively low, indicating that PostgreSQL may miss out on a significant number of relevant documents in the dataset when performing phrase-based searches.

  Both MySQL and SQLite exhibit lower F1 scores of 0.26 and 0.28, respectively. This implies that these systems may have limitations in their ability to effectively retrieve relevant documents for the provided set of phrases compared to PostgreSQL. Both systems show similar precision scores of 0.66, indicating that a substantial portion of the retrieved documents are relevant. However, the recall values are considerably lower, suggesting that MySQL and SQLite may miss out on a significant number of relevant documents in the dataset when performing phrase-based searches.

  The main reason these models are missing the synonyms words. As we will see in the next section, this will increase the score but depends on some advanced configurations.

System	Precision	Recall	F1 Score
PostgreSQL	0.66	0.3	0.41
MySQL	0.66	0.17	0.26
SQLite	0.66	0.18	0.28

These results provide insights into the effectiveness of each system's phrase search capabilities. However, it is important to consider that these results are specific to the dataset, query set, and evaluation criteria used in this analysis. The performance of each system may vary in different scenarios and with different datasets.

To make informed decisions regarding the selection of the appropriate database system for phrase-based search scenarios, it is recommended to further evaluate the systems based on specific use cases, requirements, and potentially explore additional tuning or configuration options to improve the retrieval of relevant documents. Additionally, considering other factors such as scalability, indexing options, and system compatibility may also be important when making a comprehensive assessment.

5.2.3 Comparison of word and phrase searching capabilities with the enhancements

In this section, we will delve into the comparison of word and phrase searching capabilities with the enhancements using the variation functionality of full-text searching. By incorporating variations of words during the search process, we aim to improve the accuracy and effectiveness of retrieving relevant documents.

Now, we will utilize the F1 score analysis to determine the updated scores for precision, recall, and F1 score. With the enhancements made to the queries for word search capabilities in each DBMS system (PostgreSQL, MySQL, SQLite), we have expanded the scope of relevant documents by considering variations of the word during the full-text search. These improvements enable us to capture a more comprehensive set of relevant documents.

Through this analysis, we aim to identify the strengths and weaknesses of the word capabilities in each system with the inclusion of variations. This will enable us to understand the impact of the enhancements and determine which system performs better in retrieving relevant documents.

Let's proceed with the analysis and explore the results to gain insights into the effectiveness of the variation functionality in improving the search capabilities of PostgreSQL, MySQL, and SQLite

The analysis

In order to compare the word and phrase searching capabilities of the three DBMS systems (PostgreSQL, MySQL, SQLite) with the enhancements, we have made improvements to the code

For PostgreSQL, we have modified the query to use the to_tsvector and to_tsquery functions with the addition of the :* wildcard. The updated query for PostgreSQL is:

```
SELECT COUNT(*) AS total_relevant_documents,
      SUM(CASE WHEN to_tsvector('english', overview) @@ to_tsquery('english',
          'word:*') THEN 1 ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000;
```

For MySQL, we have adjusted the query to use the MATCH function with the AGAINST clause in BOOLEAN MODE, and added the * wildcard to match variations of the word. The updated query for MySQL is:

```
SELECT COUNT(*) AS total_relevant_documents,
      SUM(CASE WHEN MATCH(overview) AGAINST ('word:*' IN BOOLEAN MODE) THEN 1
          ELSE 0 END) AS retrieved_relevant_documents
FROM movies10000;
```

For SQLite, we have modified the query to use the MATCH operator with the LIKE keyword and the * wildcard to match variations of the word. The updated query for SQLite is:

```
SELECT COUNT(*) AS total_relevant_documents,
      SUM(CASE WHEN overview MATCH 'word*' THEN 1 ELSE 0 END) AS
          retrieved_relevant_documents
FROM movies10000;
```

These enhancements allow us to capture a wider range of relevant documents by considering variations of the word during the full-text search. We will use these enhanced queries to evaluate the word search capabilities of each DBMS system and analyze the results using precision, recall, and F1 score.

Please find in the below table the results based on the new enhancements:

System	Precision	Recall	F1 Score
PostgreSQL	0.92	0.72	0.80
MySQL	1	0.56	0.71
SQLite	1	0.52	0.68

For PostgreSQL, we see that the precision remains the same at 0.92, indicating a high level of accuracy in retrieving relevant documents. However, the recall value increases to 0.72, which means that a higher percentage of relevant documents are successfully retrieved. This improvement

in recall leads to an increase in the F1 score to 0.80, indicating an overall enhancement in the system's performance.

Similarly, in the case of MySQL, the precision remains perfect at 1, indicating precise retrieval of relevant documents. The recall value improves to 0.56, indicating a higher percentage of relevant documents successfully retrieved. As a result, the F1 score increases to 0.71, indicating an improved overall performance compared to Table 1.

In SQLite, we also observe an improvement in the recall value, which increases to 0.52 from the previous 0.37 in Table 1. This signifies that a higher proportion of relevant documents are successfully retrieved. The F1 score increases to 0.68, indicating an overall enhancement in the system's performance.

These results demonstrate that the inclusion of variations in the search queries has a positive impact on the search capabilities of all three systems. By considering variations, the systems are able to capture a broader range of relevant documents, thereby increasing their recall and overall effectiveness in retrieving the desired information.

It is important to note that even with these enhancements, the precision values remain high for all systems, indicating that the accuracy of retrieving relevant documents is maintained. This implies that the systems are still able to effectively filter out irrelevant results, ensuring the quality of the retrieved documents.

5.2.4 Full text searching for synonyms

Although the enhancements in variations have shown improvements in the precision, recall, and F1 score of the search systems, it is important to note that these enhancements primarily focus on capturing variations of the specified word. However, they may not effectively address the challenge of finding synonyms during full text searching.

Finding synonyms can be a complex task, as it requires understanding the context and meaning of words in relation to the query. While variations can help capture different forms of a word (e.g., plurals, different tenses), they may not cover the entire range of synonyms associated with the query term.

To address this limitation and further improve the search capabilities, each system can consider implementing additional techniques. Here are some suggestions for each system:

- PostgreSQL:
  - Utilize a thesaurus: Integrate a thesaurus or synonym dictionary within the full text search configuration. This allows the system to map query terms to their synonyms and retrieve relevant documents accordingly.
  - Expand search terms: Apply techniques like query expansion or term rewriting to automatically include synonymous terms during the search process.
- MySQL
  - Implement synonym mapping: Create a custom mapping of synonyms using a synonym table or file. This mapping can be used to expand the search query and retrieve documents containing synonymous terms.
  - Leverage external libraries or plugins: Explore the use of external libraries or plugins that offer advanced linguistic analysis, including synonym identification and retrieval.
- SQLite
  - Preprocess data for synonyms: Before indexing the data, preprocess it to identify synonyms and create additional columns or tables that include these synonymous terms. This allows for expanded search capabilities

– Implement custom functions or extensions: Develop custom functions or extensions to handle synonym matching and retrieval during full text searching.

By incorporating these suggestions, the systems can enhance their ability to capture synonymous terms during full text searching. This would result in improved precision, recall, and overall search performance, as a wider range of relevant documents would be retrieved.

It is worth noting that handling synonyms can be a complex task, and the effectiveness of these approaches may vary depending on the specific requirements and characteristics of the search system. It is recommended to carefully evaluate and test these enhancements to determine their impact on the search results and user experience.

5.3 Conclusion

The provided analysis discusses the F1 scores for word search and phrase search using different database management systems (PostgreSQL, MySQL, and SQLite). The F1 score is a measure of the balance between precision and recall in retrieving relevant documents.

For word search, PostgreSQL demonstrates a relatively good balance with an F1 score of 0.75, indicating effective retrieval of relevant documents. However, MySQL and SQLite have lower F1 scores (0.56 and 0.54, respectively), suggesting limitations in their search effectiveness compared to PostgreSQL.

For phrase search, PostgreSQL shows a moderate balance with an F1 score of 0.41. While it retrieves a reasonable portion of relevant documents, there is room for improvement in recall. MySQL and SQLite have lower F1 scores (0.26 and 0.28, respectively) and also exhibit lower recall values.

In the next section, it is mentioned that the models are missing synonyms, which can impact the scores. Advanced configurations and techniques are recommended to improve performance, such as utilizing thesauri, expanding search terms, implementing synonym mapping, leveraging external libraries or plugins, and preprocessing data for synonyms.

Overall, the results indicate that including variations and synonyms in search queries can enhance the performance of the systems by improving recall and capturing a broader range of relevant documents while maintaining high precision values. Among the evaluated database management systems (PostgreSQL, MySQL, and SQLite), PostgreSQL stands out with a better F1 score (precision and recall). For both word search and phrase search, PostgreSQL demonstrates superior performance compared to MySQL and SQLite. With an F1 score of 0.75 for word search and 0.41 for phrase search, PostgreSQL exhibits a more effective balance between precision and recall, indicating its stronger capability in retrieving relevant documents. MySQL and SQLite, on the other hand, exhibit lower F1 scores, suggesting limitations in their search effectiveness. To further enhance the search capabilities of each system, specific techniques such as the utilization of thesauri, query expansion, synonym mapping, and preprocessing of data for synonyms are recommended.

Chapter 6

Conclusion

In this diploma thesis, we embarked on a comprehensive exploration of text indexing and full-text searching in three widely used database management systems: SQLite, PostgreSQL, and MySQL. Our goal was to understand their capabilities, advantages, and disadvantages, and to compare their performance in handling word and phrase searching. Additionally, we aimed to identify potential areas for improvement and future research in the field of full-text searching.

Throughout our research, we have demonstrated the significant advantages of full-text searching over traditional search techniques. The ability to handle word variations, misspellings, and complex query requirements is crucial in many real-world applications. Full-text searching provides a more flexible and intuitive approach to information retrieval, enabling users to find relevant results even in large datasets.

In our analysis of PostgreSQL, we explored its comprehensive set of full-text indexing features. We examined the basic functionality, auxiliary functions, and the optimization techniques for the ts_rank function. Moreover, we investigated the role of dictionaries in influencing search results and compared the performance and characteristics of the GIN and GiST indexing methods. The examples provided a practical understanding of how PostgreSQL can be leveraged for efficient full-text searching.

Moving on to MySQL, we examined its different types of full-text searching. We delved into the natural language full-text search, which offers a convenient and user-friendly way to perform searches. We also explored the Boolean mode and query expansion mode, understanding their functionalities and limitations. Additionally, we discussed the implementation of full-text indexing in MySQL, considering the storage engines InnoDB and MyISAM. This analysis helped us comprehend the specific nuances and trade-offs involved in utilizing MySQL for full-text searching.

In the case of SQLite, we investigated its basic and advanced full-text searching techniques. We explored the FTS5 module, which provides powerful features such as prefix queries, initial token queries, NEAR queries, and column filters. We also discussed auxiliary functions available in SQLite, such as snippet and offsets, which enhance search capabilities. The examination of full-text indexing in SQLite shed light on its implementation details and integration within the database management system.

To evaluate the search performance of the three database management systems, we conducted a thorough comparison analysis focused on word and phrase searching. We devised a methodology based on the F1 score, a widely accepted measure for information retrieval evaluation. By applying this methodology to our datasets, we were able to quantitatively assess the search performance of each system. The experimental results provided insights into the strengths and weaknesses of the three systems, offering valuable guidance for selecting the most suitable solution for specific use cases.

Furthermore, we compared the word and phrase searching capabilities among the systems,

specifically examining the enhancements provided by each system. This analysis allowed us to understand the extent to which the systems could handle complex search queries and retrieve relevant results. We also explored the potential of full-text searching for synonyms, which is crucial for improving search precision and recall. In the next paragraphs we will provide several potential avenues for future improvement and research in this field

Firstly, expanding the comparative analysis to include additional database management systems, such as Oracle or Microsoft SQL Server, would provide a more comprehensive evaluation of full-text searching capabilities across different platforms. Expanding the comparative analysis to include additional database management systems, such as Oracle or Microsoft SQL Server, would enable a more comprehensive evaluation of full-text searching capabilities across a wider range of platforms. Each DBMS may employ different techniques and algorithms for handling synonyms, such as synonym dictionaries or ontologies, to enhance search results. Exploring and comparing these approaches would contribute to a more nuanced understanding of synonym support in different systems.

Additionally, conducting performance benchmarking tests on larger datasets would allow for a more in-depth investigation of scalability and efficiency. Exploring the integration of natural language processing (NLP) techniques with full-text searching could enhance the relevance and understanding of search queries. The integration of natural language processing (NLP) techniques with full-text searching represents an exciting area for future improvement. NLP techniques, such as part-of-speech tagging, semantic analysis, and entity recognition, can enhance the relevance and understanding of search queries by extracting meaning and context from textual data. Investigating how NLP can be effectively integrated into full-text searching algorithms in different DBMS systems would open up new possibilities for more accurate and context-aware search results.

Furthermore, studying the impact of different languages and character encodings on full-text searching effectiveness in multilingual environments would be valuable. Languages vary in terms of word morphology, syntax, and semantic nuances, which can pose challenges for search algorithms. Investigating how different DBMS systems handle multilingual indexing, tokenization, and language-specific features would provide valuable insights into their performance and potential areas for improvement.

In addition to the potential avenues mentioned above, the exploration of synonyms and their handling in full-text searching across different database management systems presents an intriguing area for future research. Synonyms play a crucial role in improving search precision and recall by capturing the inherent variability in language and user queries. Investigating how various DBMS systems incorporate synonym functionality would provide valuable insights into their capabilities and potential enhancements.

Lastly, conducting user studies or surveys to gather feedback and insights on the usability and user satisfaction with full-text searching capabilities would contribute a user-centric perspective. Understanding the practical challenges and requirements faced by users in real-world scenarios can guide the development of user-friendly interfaces, query expansion techniques, and relevance ranking algorithms.

In conclusion, this diploma thesis serves as a valuable resource for practitioners and researchers interested in implementing text indexing and full-text searching in SQLite, PostgreSQL, and MySQL. By comprehensively exploring their features and conducting a performance comparison, we have provided a solid foundation for making informed decisions and driving advancements in this field. Moreover, future research directions involving synonym handling, integration with NLP techniques, multilingual support, and user-centric studies have the potential to further enhance the capabilities and usability of full-text searching across different DBMS systems.

Bibliography

[1] Aleksander Alekseev. “Full Text Search in PostgreSQL”. In: *Presentation* 916 (), pp. 1–30.

[2] Jimmy Angelakos. “The State of (Full) Text Search in PostgreSQL 12”. In: *Presentation* 916 (2020), pp. 1–40.

[3] Oleg Bartunov. “Full-Text Search in PostgreSQL, A Gentle Introduction”. In: *Oleg Bartunov and Teodor Sigaev* (2001), pp. 1–32.

[4] Kat Batuigas. “Postgres Full-Text Search: A Search Engine in a Database”. In: *website* (). URL: https://www.crunchydata.com/blog/postgres-full-text-search-a-search-engine-in-a-database.

[5] Jeffrey Beall. “The Weaknesses of Full-Text Searching”. In: *The Journal of Academic Librarianship* 34 (2008), pp. 438–444.

[6] Charles Bell. “Expert MySQL”. In: *Book* 916 (2012), pp. 57–105.

[7] Max Bramer. “Web Programming with PHP and MySQL”. In: *Book* 916 (2015), pp. 197–243.

[8] Compose. “Indexing for full text search in PostgreSQL”. In: *website* (). URL: https://www.compose.com/articles/indexing-for-full-text-search-in-postgresql/.

[9] ROB CONERY. “CREATING A FULL TEXT SEARCH ENGINE IN POSTGRESQL, 2022”. In: *website* (2022). URL: https://robconery.com/postgres/creating-a-full-text-search-engine-in-postgresql-2022/.

[10] Anthony DeBarros. “PRACTICAL SQL A Beginner’s Guide to Storytelling with Data”. In: *Book* 916 (2018), pp. 336–348.

[11] MySQL Documentation. “MySQL - Full-Text Search Functions”. In: *website* (). URL: https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html.

[12] SQLite Documentation. “SQLite FTS5 Extension”. In: *website* (). URL: https://www.sqlite.org/fts5.html.

[13] Paul Dubois. “MySQL”. In: *Book* 916 (2015), pp. 197–243.

[14] Omar Farooq. “SQLite Full Text Search”. In: *website* (). URL: https://linuxhint.com/sqlite-full-text-search/.

[15] W. Jason Gilmore and Robert H. Treat. “Beginning PHP and PostgreSQL 8 From Novice to Professional”. In: *Book* 916 (2006), pp. 573–593.

[16] Mike Owens Grant Allen. “The Definitive Guide to SQLite”. In: *Book* 916 (2010), pp. 87–124.

[17] The PostgreSQL Global Development Group. “PostgreSQL 11.16 Documentation”. In: *Documentation* 916 (2022), pp. 347–380.

[18] Jon Heller. “Pro Oracle SQL Development”. In: *Book* 916 (2019), pp. 3–29.

[19] Anna Bailliekova Henrietta Dombrovskaya Boris Novikov. “PostgreSQL Query Optimization”. In: *Book* 916 (2021), pp. 1–132.

[20] Timothy C. Bell Ian H. Witten Alistair Moffat. “Compression and full-text indexing for Digital Libraries”. In: *Digital Libraries Current Issues* 916 (1995), pp. 182–201.

[21] ELIZABETH INERSJÖ. “Comparing database optimisation techniques in PostgreSQL”. In: *Book* 916 (2021), pp. 1–35.

[22] javapoint. “MySQL FULLTEXT SEARCH (FTS)”. In: *website* (2022). URL: https://www.javatpoint.com/mysql-fulltext-search.

[23] Michael Kofler. “The Definitive Guide to MySQL5”. In: *Book* 916 (2005), pp. 3–47.

[24] Jay A. Kreibich. “Using SQLite”. In: *Book* 916 (2010), pp. 169–171.

[25] Jesper Wisborg Krogh. “MySQL 8 Query Performance Tuning”. In: *Book* 916 (2022), pp. 311–344.

[26] Frank M. Kromann. “Beginning PHP and MySQL From Novice to Professional Fifth Edition”. In: *Book* 916 (2018), pp. 567–603.

[27] MICHAEL KRUCKENBERG and JAY PIPES. “Pro MySQL”. In: *Book* 916 (2005), pp. 57–62.

[28] Kevin Languedoc. “Build iOS Database Apps with Swift and SQLite”. In: *Book* 916 (2016), pp. 11–21.

[29] MariaDB. “Full-Text Indexes”. In: *website* (2022). URL: https://mariadb.com/kb/en/full-text-indexes/.

[30] NEIL MATTHEW and RICHARD STONES. “SQL Primer An Accelerated Introduction to SQL Basics”. In: *Book* 916 (2005), pp. 352–356.

[31] Regina O. Obe and Leo S. Hsu. “PostgreSQL: Up and Running, A Practical Guide to the Advanced Open Source Database”. In: *Digital Libraries Current Issues* 916 (2018), pp. 133–148.

[32] Michael Owens. “The Definitive Guide to SQLite”. In: *Book* 916 (2006), pp. 73–171.

[33] Andrey Volkov Salahaldin Juba. “Learning PostgreSQL 11 Third Edition”. In: *Book* 916 (2019), pp. 322–329.

[34] Hans-Jürgen Schönig. “Mastering PostgreSQL 12 Third Edition”. In: *Book* 916 (2019), pp. 92–98.

[35] Hans-Jürgen Schönig. “Troubleshooting PostgreSQL”. In: *Book* 916 (2015), pp. 35–41.

[36] Baron Schwartz. “High performance SQL”. In: *Book* 916 (2012), pp. 137–199.

[37] Elvis C. Foster with Shripad Godbole. “Database Systems A Pragmatic Approach”. In: *Book* 916 (2016), pp. 451–460.

[38] Introduction to SQLite full-text search. “Getting Started with SQLite Full-text Search”. In: *website* (). URL: https://www.sqlitetutorial.net/sqlite-full-text-search/.

[39] Jason Strate. “Expert Performance Indexing in SQL Server 2019”. In: *Book* 916 (2019), pp. 1–27.

[40] MySQL Tutorial. “Creating FULLTEXT Indexes for Full-Text Search”. In: *website* (). URL: https://www.mysqltutorial.org/activating-full-text-searching.aspx.

[41] SQLite Tutorial. “Getting Started with SQLite Full-text Search”. In: *website* (2022). URL: https://www.sqlitetutorial.net/sqlite-full-text-search/.

[42] Robert H. Treat W. Jason Gilmore. “Beginning PHP and PostgreSQL 8”. In: *Novice to Professional* (2006), pp. 749–764.

[43] w3resource. “MySQL Full text search”. In: *website* (2022). URL: https://www.w3resource.com/mysql/mysql-full-text-search-functions.php.

[44] Mark Watson. “Scripting Intelligence”. In: *Web 3.0 Information Gathering and Processing* 34 (2009), pp. 192–204.

[45] Adam Zegelin. “PostgreSQL® Full-Text Search”. In: *website* (2022). URL: https://www.instaclustr.com/blog/postgresql-full-text-search/.