UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Comparison of full-text search performance in relational and non-relational database systems

by

Georgios Fotopoulos

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Christos Tryfonopoulos, Associate Professor

Examination Committee: Spiros Skiadopoulos, Professor
Anastasia Krithara, Post-Doctoral Researcher

Athens, November 2022

Comparison of full-text search performance in relational and non-relational database systems

Georgios Fotopoulos

MSc. Thesis, MSc. Programme in Data Science

University of the Peloponnese & NCSR "Democritos", November 2022

UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Comparison of full-text search performance in relational and non-relational database systems

by

Georgios Fotopoulos

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Christos Tryfonopoulos, Associate Professor

Examination Committee: Spiros Skiadopoulos, Professor
Anastasia Krithara, Post-Doctoral Researcher

Approved by the examination committee on November, 2022.

(Signature)	(Signature)	(Signature)
.....	.....	.....
Christos Tryfonopoulos Associate Professor	Spiros Skiadopoulos Professor	Anastasia Krithara Post-Doctoral Researcher

Athens, November 2022

UNIVERSITY OF THE PELOPONNESE & NCSR "DEMOCRITOS"
MSC PROGRAMME IN DATA SCIENCE

Declaration of Authorship

(1) I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

(2) I confirm that this thesis presented for the degree of Bachelor of Science in Informatics and Telecommunications, has

   (i) been composed entirely by myself

   (ii) been solely the result of my own work

   (iii) not been submitted for any other degree or professional qualification

(3) I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification except as specified.

(Signature)

.....

Georgios Fotopoulos

Athens, November 2022

Acknowledgments

I would like to express my gratitude to my professor and primary supervisor Mr. Christos Tryfonopoulos, for his guidance and feedback throughout this project.
I would also like to thank my friends and family who supported me and offered deep insight into the study.

To my family.

Abstract

In this work, we aim to compare and report the performance between relational and non-relational database systems in text retrieval, and more specifically in full-text search. This comparative study is focused on the characteristics of some of the most popular database management systems, regarding the overall performance when querying text documents. We also pose and highlight the features, the technical specifications and the differences of the database systems, through several comparative tests and theoretical exploration.

For the evaluation of the perfomance in full-text search, both relational and non-relational database systems have been selected to be compared in this study. Our research comprises of studying PostgreSQL on the one hand, as a representative system from the relational database systems family, wich supports a wide variety of full-text search capabilities and advanced features, while on the other hand, we study non-relational databases like MongoDB, Apache Solr and Elasticsearch.

Our findings indicate that non-relational and text search database systems provide a trustworthy alternative in full-text search, regardless the amount of data they operate with, where on the other hand, the relational models are sensitive to their parameters as it can operate well under specific conditions.

Contents

List of Tables

List of Figures

List of Abbreviations

DBA	Database Administrator
RDBMS	Relational Database Management System
SQL	Structured Query Language
NoSQL	Not Only Structured Query Language
CRUD	Create, Read, Update, Delete
ACID	Atomicity, Consistency, Isolation, Durability
JSON	Javascript Object Notation
XML	Extensible Markup Language
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
DWH	Data Warehouse
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
DSL	Domain Specific Language
REST	Representational State Transfer
GIN	Generalized Inverted Index
GiST	Generalized Search Tree

Chapter 1

Introduction

In text retrieval, **full-text search** refers to techniques for searching a single computer-stored document or a collection in a database. In a full-text search, a search engine examines all of the words in every stored document as it tries to match search criteria [1]. Full-text queries, while been executed in several types of database systems, perform linguistic searches against text data, by operating on words and phrases based on particular rules. Full-text queries can include simple words and phrases or multiple forms of a word or phrase and return any documents that contain at least one match [2].

Relational databases excel at storing and manipulating structured data – anything that is in a table format of rows and columns. They support flexible search of multiple record types for specific values of specific fields, and can be great for quickly and securely updating specific individual records. However, some of the fields in a database's records can be free-form text (like a product description). Most relational databases provide support for doing keyword searching on these unstructured fields. But, the relevancy ranking of results coming out of a database do not have the same quality or sophistication as the best full-text search systems. Further, unless the Database Administrator (DBA) knows what questions the user will ask, the performance of a Relational Database Management System (RDBMS) will be quite slow and provide a poor user experience.

Text search systems and non-relational databases on the other hand, are better

for quickly searching high volumes of structured, unstructured, or semi-structured data according to a specific word, bag of words or phrase. They provide rich text search capabilities and sophisticated relevancy ranking for ordering results on how well they match a potentially "fuzzy" search query (words that don't quite match, like typos or homonyms).
Some key differences between relational and non-relational database systems are summarized in Table 1.1.

Type	**Relational**	**Non-relational**
Language	Structured Query Language	Unstructured Query Language
Data	Data stored in Tables, Columns, Rows	Data stored in Collections, Fields, Documents
Schema	Static	Dynamic
Structure	Table-based	Document-based, key-value pairs, graph databases, wide-column stores
Scalability	Vertically scalable	Horizontally scalable
Flexibility	Rigid schema bound to relationship	Non-rigid schema and flexible
Transaction	ACID properties	CAP theorem

Table 1.1: Relational vs non-relational databases.

In this work, we aim to present a comparative study between relational and non-relational database models, highlighting the overall characteristics of each database technology in text retrieval, especially for non-relational systems as a new solution over relational databases.

1.1 Problem description

Experimenting with the performance of and comparing SQL and NoSQL databases, have been made in various research papers. Both advantages and disadvantages, as well as the overall features for each of the systems, have been posed and highlighted through several studies. In their survey, Nayak et al. [3], analyze the different types and characteristics of NoSQL database systems, and list their advantages and disadvantages over relational databases whereas in [4] and [5], the authors conduct

a survey addressing the concepts of NoSQL databases, their main application areas and the security issues concerning these databases, compared with the traditional relational models.
But what is the performance of the database systems when it comes to execute queries in large amount of unstructured textual data? How much time is needed for such operations and what is the cost? Can relational databases perform as well as non-relational and text search systems do, in terms of time saving?
In the light of the above, we concentrate in this work on textual performance in an attempt to answer the above research questions by executing full-text queries on datasets of varying sizes, on both relational and on non-relational databases, thus presenting the results obtained during performance comparison tests.

1.2 Thesis structure

This document is structured as follows. On Chapter 2 we review related work by looking at related papers and other surveys comparing relational and non-relational database management systems. On Chapter 3 we outline some useful definitions and theory about the relational and non-relational models including the systems that we study. On Chapter 4 we present the experimental setup giving an overview of the datasets and the full-text queries that we use in our experiments, we describe the technical specifications of each one of the systems that we examine, we discuss the databases and the indices structure, and we present the experimental results of our study. Finally, on Chapter 5, we outline the conclusions and the future work that arises from them.

Chapter 2

Related Work

The vast majority of databases can be categorized as either relational or non-relational, with the main difference between these to be the way they store information. For several years, relational databases are accepted as the traditional database systems for storage. However, with the increase of the internet usage and the huge amounts of structured, semi-structured and unstructured data that are being produced in a daily basis, a need for an alternative, like non-relational systems, is essential.

Relational models represent how data is stored in relational databases. A relational database stores data in the form of relations (tables of values). Every row in a table represents a collection of related data values and the columns of a table hold attributes of the data. A system used to maintain relational databases is a relational database management system (RDBMS) [6]. Relational database systems are equipped with the option of using the Structured Query Language (SQL) for querying and maintaining the database and therefore relational databases are also widely known as SQL databases [7].

A non-relational database, on the other hand, stores data in a non-tabular form and it does not follow the relational model provided by traditional relational database management systems. Instead, the non-relational database uses a storage model optimized for specific requirements of the type of data being stored [8]. Non-relational databases often perform faster because a query doesn't have to view several

tables in order to deliver an answer, as relational systems often do and therefore, they are ideal for applications that handle many different kinds of data [9]. Some popular non-relational database types are: document data store, column-oriented database, key-value store and graph database. Non-relational systems are often called "non-SQL" or "Not only SQL" (NoSQL) to emphasize that they may support SQL-like query languages or sit alongside SQL databases [10].

There are several early studies that have devoted their attention on the comparison between relational and non-relational database models (or SQL and NoSQL), evaluating their performance as well. Jatana et al. [11] report the comparison between these two leading types of database models by studying various relational and non-relational systems and listing the attributes and features of both. In order to showcase the main advantages and disadvantages of the models, the authors present some of the most extensively used tools along with their features, they present the categories that the non-relational databases are classified and finally they conclude and highlight the major differences between the two types. Moreover, Sahatqija et al. [12] compare in their study, some SQL and NoSQL features such as scalability and performance, flexibility, query language, security, data management including storage and availability of data and present their advantages and disadvantages over each other.
Beyond the investigation in the theoretical background in the aforementioned studies, our contribution in this work is a practical take on the comparison of the databases in the field of full-text search through several experiments we conducted.

In their work Ceresnak and Kvet [13], present the database storage architectures, principles and differences and describe the manipulation operations (insert, update, delete and select), choosing for their evaluation some well known relational databases such as Oracle, MySQL and MS SQL to be consecutively compared with some non-relational oriented systems such as Mongo, Redis and Cassandra. After conducting several experiments on the databases' query performance, they decided on the speed and the effectiveness of NoSQL database systems, especially in the higher number of records. In our study we take a step further, carrying out a number of experiments, as we compare the textual performance of the database systems on several full-text

search operations.

Since MongoDB is one of the most popular NoSQL database models, a number of papers have been involved with comparative studies between MongoDB and the traditional relational database systems. Results show that Mongo performs equally as well or better than most of the relational databases [14]. Mongo provides lower execution times in basic operations (insert, select, update, delete) compared to the relational MySQL [15], as well as for various CRUD operations (create, read update, delete) for small and large datasets [16]. Moreover, Mongo is generally optimized for key-value store implementations and performs faster in most operations, while SQL databases are not. Yishan et al. clearly state this aspect in their research [17]. In our work, we examine how Mongo performs with full-text search operations and we compare it with a relational model, as well as with some of the standard NoSQL and Text search systems.

Text search systems excel at efficiently searching large volumes of text, including unstructured or semi-structured content. Full-text search engines also have relevancy ranking capabilities to determine the best match for a query. In Lucidworks.com [18], the author inspects the capabilities of such systems compared to relational database models. The most efficient full-text search engines are Apache Solr and Elasticsearch, both built on Apache Lucene Java library. The AnyTXT Searcher [19] web article, introduces the differences and similarities of Apache Solr and Elasticsearch, as well as their advantages and disadvantages. We also make an attempt to put these systems in comparison while performing with some of the most common full-text search operations such as exact phrase matching, wildcards search and conjunctions, setting also side by side the outcomes of similar experiments from some of the most known relational and non-relational models.

Chapter 3

Theory and Definitions

In this chapter we cover the theoretical background of this work, before we present the experimental evaluation in the chapter that follows.

3.1 Theoretical background

In this section we discuss the theory and definitions concerning the database systems, with reference to the two leading types of database storage components in the industry, *relational* and *non-relational*, and we examine the features and overall characteristics of each one of the database technologies with regard to ease of use, installation, querying the data and full-text searching.
The systems that we have selected to report and investigate are:

- **Relational database systems (SQL)**
  PostgreSQL [1]
- **Non-relational database systems (NoSQL)**
  MongoDB [2], Elasticsearch [3]
- **Text search systems**
  Apache Solr [4]

[1] https://www.postgresql.org/
[2] https://www.mongodb.com/
[3] https://www.elastic.co/
[4] https://solr.apache.org/

The main criteria of our selection are the usability and the robustness of the systems, the wide acceptance among the users, and their capabilities in full-text search.
After presenting the theoretical framework, we outline the technical configurations and the setup of the database systems under investigation, in the upcoming sections.

3.1.1 Relational database systems (SQL)

PostgreSQL [5] (or simply Postgres) is a free and open-source object-relational database management system that uses the SQL language. Postgres features transactions with ACID properties (Atomicity, Consistency, Isolation and Durability) and is designed to handle a range of workloads, from single machines to data warehouses or web services. Postgres has been proven to be highly extensible and scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate [20], [21].

Ease of Use and Installation. Postgres is an easy to use database with its full stack of RDBMS database features and capabilities that can handle structured and unstructured data. Postgres supports many data types, ranging from traditional ones (integer, date, timestamp) to complex ones (JSON, XML, TEXT). Postgres has replication and clustering capabilities and can ensure data operations are distributed horizontally [22].
Before Postgres can be used, first need to be installed. Postgres installation is a process that can be very easily achieved, regardless of operating system and the following steps are essential to complete the installation:

1. Download PostgreSQL installer
2. Install PostgreSQL
3. Verify the installation

Querying Data. Postgres is a fully SQL-compliant database and supports all SQL standard features. It is a database of high demand among developers who have

[5] https://www.postgresql.org/

to write complex queries and it makes it a popular choice for online transaction processing (OLTP), online analytical processing (OLAP), and data warehouse (DWH) environments.
Like any other relational database, the data is stored in tables. To retrieve data, the database is queried with the use of an SQL *'SELECT'* statement. The statement is divided into a select list (the part that lists the columns or attributes to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). In what follows, we present a typical example of an SQL query.

SELECT select_list FROM table_name;

Full-text searching. Postgres has ~, ~*, LIKE, and ILIKE operators for textual data types, but they lack many essential properties required by modern information systems like Solr and Elasticsearch. Using these operators, full-text search is supported in Postgres. A typical example of an operator is the following.

SELECT FROM table_name WHERE column LIKE 'xxxx_';

The full-text and phrase search features in Postgres are very powerful and fast. Full-text searches can be accelerated using indexes. Indexing allows documents to be preprocessed and an index is saved for later rapid searching. Preprocessing includes:

1. Parsing documents into tokens.
2. Converting tokens into lexemes.
3. Storing preprocessed documents optimized for searching.

A document is the unit of searching in a full-text search system. For searches within Postgres, a document is normally a textual field within a row of a database table or a combination of such fields, stored in several tables.
Postgres provides two data types that are designed to support full-text search: **to_tsvector()** and **to_tsquery()**. A data type tsvector is provided for storing preprocessed documents, along with a type tsquery for representing processed queries. The tsvector type represents a document in a form optimized for text search which is used to parse and normalize a document string. The tsquery type similarly rep-

resents a text query and contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, and NOT operators. The to_tsvector() function breaks up the input string and creates tokens out of it, which is then used to perform full-text search using the to_tsquery() function.
There are many functions and operators available for these data types, the most important of which is the match operator **@@**. The match operator @@ returns true if a tsvector matches a tsquery [23].

SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat');

Table 3.1 summarizes the basic operators that are provided for full-text searching in Postgres [24].

Operator	**Return Type**	**Description**
@@	boolean	tsvector matches tsquery ?
@@@	boolean	deprecated synonym for @@
||	tsvector	concatenate tsvectors
&&	tsquery	AND tsquerys together
||	tsquery	OR tsquerys together
!!	tsquery	negate a tsquery
<->	tsquery	tsquery followed by tsquery
@>	boolean	tsquery contains another ?
<@	boolean	tsquery is contained in ?

Table 3.1: Postgres text search operators.

3.1.2 Non-relational database systems (NoSQL)

MongoDB [6] (or simply Mongo) is a source-available cross-platform document-oriented database program. It is classified as a NoSQL database and stores data in JSON-like documents with dynamic schema in the form of the structure (field: value, pair) rather than tabular form. It provides high performance, high availability, easy scalability, auto-sharding and out-of-the-box replication [25]. An example of a document that can be stored in Mongo, is the following.

[6] https://www.mongodb.com/

```
{
    "id": 3,
    "name": "John Doe",
    "age": 20,
    "hobbies": {
        "indoor": [
            "Chess"
        ],
        "outdoor": [
            "Basketball"
        ]
    }
}
```


Ease of Use and Installation. Mongo is always preferred when the main concern is the deal with large volume of data with a high performance. As a document store database, makes it easy to store structured, semi-structured or unstructured data. Also, horizontal scaling in case of Mongo is very easy, since it is a schema less database. This type of work can be directly handled by the application automatically and there is no need to any type of administrative work for performing any type of horizontal scaling [26], [27].
Installing Mongo is an easy process, following the steps mentioned below:

1. Download the MongoDB installer package.

2. Install MongoDB.

3. Create the directory (...\data\db) where MongoDB will store it's files.

Querying Data. Mongo queries provide the simplicity in process of fetching data from the database, it is similar to SQL queries in SQL database language. Mongo supports field, range query, and regular-expression searches. Queries can return specific fields of documents and also include user-defined JavaScript functions. Queries can also be configured to return a random sample of results of a given size.

Mongo provides the function names as *'db.collection_name.find()'* to operate query operations on a database. The *find()* method displays the database collection in non-structure form ({Key:value}) including auto-created key "id" by Mongo and collection data inserted by the user or the admin [28]. Below we present some query examples.

Syntax: *db.collection_name.find()*

db.student.find(StudentName : "Smith")

db.student.find(score:math: 230, science: 234)

db.student.find().limit(10)

Full-text searching. Mongo supports query operations that perform a text search of string content. To perform text search queries on a collection, Mongo uses a text index and the **$text** operator.

db.collection.find({text : {search: "phrase"} })

Text indices can include any field whose value is a string or an array of string elements and support efficient execution of queries by avoiding full collection scans, resulting in a very efficient query.

When creating a text index on a field, Mongo tokenizes and stems the indexed field's text content and sets up the indexes accordingly. A collection can have a text index created on a single field but in case text search is being used on multiple fields of a document, multiple fields can be indexed by enabling compound text indexing. The $text operator tokenizes the search string using whitespace and most punctuation as delimiters, and performs a logical OR of all such tokens in the search string [29].

db.stores.find({text : {search: "java coffee shop"} })

Elasticsearch [7] is a search engine, which is based on the Lucene library [8]. It is developed in Java and provides a distributed multitenant-capable full-text search engine with a hypertext transfer protocol (HTTP) web interface and schema-free JSON documents. It provides scalable search, has near real-time search and supports multitenancy. Elasticsearch is distributed, which means that indices can be divided

[7] https://www.elastic.co/

[8] https://lucene.apache.org/

into shards and each shard can have zero or more replicas. Elasticsearch supports real-time GET requests, which makes it suitable as a NoSQL datastore but it lacks distributed transactions [30].

Ease of Use and Installation. Elasticsearch is one of the most popular enterprise search engines, is easy to install and configure but it's quite a bit heavier than Apache Solr, which we will introduce later in this section. Elasticsearch can be used to search any kind of documents and Java is the primary prerequisite for installing. Together with Elasticsearch, Logstash [9] and Kibana [10] are the main components of the Elastic stack (also known as ELK). Kibana is a free user interface which allows for visualization of Elasticsearch data and navigation in the Elastic stack.

Querying Data. The search application programming interface (API) is used for searching and aggregating data stored in Elasticsearch data streams or indices. The search API returns search hits that match a query defined in the request. The API's query request body parameter accepts queries written in Query DSL (Domain Specific Language). Query DSL supports a variety of query types, such as match, term, range and bool queries that can be mixed and matched to get the desirable results. The full Query DSL that Elasticsearch provides, is based on JSON to define queries [31].
A query is made up of two clauses:

- Leaf Query Clauses: these clauses are match, term or range, which look for a specific value in a specific field. An example of a Leaf Query is:
  POST /schools/_search { "query": { "match" : { "rating":"4.5" } } }*

- Compound Query Clauses: these queries are a combination of leaf query clauses and other compound queries to extract the desired information. An example of a Compound Query is:
  POST /schools/_search { "query": { "bool" : { "must" : { "term" : { "state" : "UP" } }, "filter": { "term" : { "fees" : "2200" } }, "minimum_should_match" : 1, "boost" : 1.0 } } }

[9] https://www.elastic.co/logstash/
[10] https://www.elastic.co/kibana/

A query starts with a query keyword and then has conditions and filters in the form of JSON object. The filter context - as the name suggests - simply filters out documents that do not match the conditions in the syntax [32]. Documents that match a search's queries are returned in the hits or search results of the response. Elasticsearch sorts the results by a relevance score that represents the quality of the match. Below we present some query examples.

Syntax: *GET /my-index/_search { "query": { "match": {"user.id": "kimchy"} } }*

GET /employees/_search { "query": { "match": { "phrase": { "query" : "heuristic" } } } }

POST /employees/_search { "query": { "match_phrase": {"phrase": { "query": "roots heuristic coherent" } } } }

Full-text searching. As we have already mentioned, Elasticsearch stores documents in JSON format. By default, it indexes all fields in a document, and they become instantly searchable. It provides a distributed, full-text search engine with an HTTP web interface and schema-free JSON documents.

The high level full-text queries are usually used for running queries on full-text fields. They understand how the field being queried is analyzed and apply each field's analyzer to the query string before executing [33].

The queries that belong in the full-text search group are:

match query, match_phrase query, match_phrase_prefix query, multi_match query, query_string query, common terms query.

3.1.3 Text search systems

Apache Solr [11] is an open-source, enterprise search platform written in Java, built on the top of Apache Lucene library that provides all of Lucene's search capabilities through HTTP requests [34]. Its major features include powerful full-text search, faceted search, real-time indexing, hit highlighting and advanced analysis/tokenization capabilities, dynamic clustering and database integration.

Ease of Use and Installation. Solr is designed for scalability and fault tol-

[11] https://solr.apache.org/

erance and is widely used for enterprise search and analytics use cases. It has REST-like (Representational State Transfer), HTTP/XML and JSON APIs that make it usable from most popular programming languages [35]. Solr's external configuration allows it to be tailored to many types of applications without Java coding.

It is easy to get up and running with Solr. Installation of Solr on Unix-compatible or Windows servers generally requires simply extracting (or, unzipping) the download package which will suffice as an initial development environment. Similarly to Elasticsearch, Java is prerequisite for installing.

Querying Data. The default Solr query syntax used to search an index, uses a superset of the Lucene query syntax. Standard Solr query parser is the default (registered as the "Lucene" query parser) [36]. The key advantage of the standard query parser is that it supports a robust and fairly intuitive syntax, allowing to create a variety of structured queries. The main query for a solr search is specified via the **'q'** parameter. The 'q' parameter defines a query using standard query syntax. This parameter is mandatory. In what follows, we present some typical examples of the standard query parser.

Syntax: *http://localhost:8983/solr/query?q=test*

http://localhost:8983/solr/techproducts/select?q=id:SP2514N

http://localhost:8983/solr/techproducts/select?q=id:SP2514N&fl=id+name

A query to the standard query parser is broken up into terms and operators. There are two types of terms: single terms and phrases. Multiple terms can be combined together with boolean operators that are supported by the standard query parser, to form more complex queries. For example, to search documents that contain "Apache" and "Lucene", we can use either of the following queries:

"Apache" AND "Lucene", *"Apache" && "Lucene"*

Full-text searching. Apache Solr runs as a standalone full-text search server, and it uses Lucene at its core for full-text indexing and searching [37]. It supports near real-time searches and takes advantage of all of Lucene's search capabilities. It also offers the ability of writing very complex text search queries that are unavailable among the rest of the database systems. Data has to be indexed on the server, so

that it becomes searchable. To index and search data, a core must be created first. The results, after searching, are sorted by relevance score. In Apache Solr, we can index various document formats such as XML, CSV, PDF, etc. and once we have the documents indexed in our repository, we can search for keywords, phrases, date ranges and geospatial data.

Chapter 4

Experimental Evaluation

In this chapter we present the experimental evaluation of our study along with the essential set up of the data and query sets, the data procesing, data indexing and the technical configuration of the database systems that we have selected to compare. We also provide the reader with some examples of the datasets and the keywords we extracted to experimented with and the syntax of the full-text queries for each system.

4.1 Creating databases and indexing data

Before we began testing, we start with the creation of the databases and data indexing for each one of the database systems of our study.

Postgres. Before uploading data in Postgres, we first had to create the databases and the tables. We created as many databases as the number of the datasets that we had to import, this means one database for the small, one for the medium and one for the large dataset respectively. We also created as many tables as the number of batches that we split the datasets (250k, 500k, 1m, 1,5m, 2m, 2,5m and 3m). For example, the tables for the Crossref-total database are: crossref-total250, crossref-total500, crossref-total1, crossref-total1half, crossref-total2, crossref-total2half and crossref-total3.
To import data from the csv files to the databases that we created, we used the **'copy'** statement. In Postgres, the 'copy' statement is used to make duplicates of

tables, records and other objects, but it is also useful for transferring data from one format to another, for example, it can be used for inserting csv data into a table as Postgres records [38].
Finally, we should configure Postgres properly for full-text search use. For this reason, we created a new column, for each table, named **'document'** of data type *tsvector*, adding the fields we want to use for text search in the queries. As an example, with the following SQL command, we create a 'document' column in the 'crossref-total250' table of the Crossref-total database, adding the fields 'title' and 'abstract':

ALTER TABLE crossref-total250 ADD COLUMN document tsvector;
UPDATE crossref-total250 SET document = to_tsvector(title, abstract)

For indexing, Postgres provides two index types that can be used to index tsvector data types: **GIN** (Generalized Inverted Index) and **GiST** (Generalized Search Tree) indexes [39]. GIN is recommended by Postgres as the default index type for full-text searching and specifically for long documents. That's because the inverted index facilitates rapid matching and retrieval. For this reason in our work, we used the GIN index to speed up text searches. The syntax to create a GIN index is:

CREATE INDEX document_idx ON crossref-total250 USING GIN (document);

Mongo. As we have mentioned before, Mongo uses text indices to support text search queries and to perform full-text search in a document. An index in Mongo is created after creating the database and the collections. Mongo's **'use database_name'** command is used to create a database. The command will create a new database if it doesn't exist, otherwise it will return the existing database [40]. Like most non-relational databases, Mongo stores data in collections instead of tables. To create a collection, the method **'db.createCollection()'** is used. Similarly with Postgres, we created as many collections as the number of batches that we split the datasets (250k, 500k, 1m, 1,5m, 2m, 2,5m and 3m).
After that, we had to create the appropriate text indices for full-text search and for this reason we used the **'db.collection.createIndex()'** method. Mongo text

indexes come with a limitation of only one text index per collection. In order to index the fields of the collections that contain string elements, we specified the string literal **'text'** in the index documents. In the following example, we create a text index for the fields 'title' and 'abstract' of the 'crossref-total250' collection:

db.crossref-total250.createIndex(title:"text", abstract:"text")

Elasticsearch. In Elasticsearch, each index is created at the same time each dataset is imported on the Elasticsearch server. Indices are used to store the documents in dedicated data structures corresponding to the data type of fields. For example, text fields are stored inside an inverted index. Mapping is the process of defining how each document, and the fields it contains, are stored and indexed [41]. By using the **'Get mapping'** API, we can view the mappings for the indices that were created after data insertions. For example, the mapping definitions for the index 'crossref-total250' can be retreived with the HTTP request:

GET /crossref-total250/_mapping

For the purposes of our study, we created indices for 250k, 500k, 1m, 1,5m, 2m, 2,5m and 3m documents, for all three datasets that we imported. For further editing of indices and mappings, we used the Kibana user interface.

Apache Solr. After installing and starting Apache Solr, we first had to create cores and then index the data. A Solr core is running an instance of a Lucene index that contains all the Solr configuration files required to use it, and it can be created using the **'bin/solr'** script [42]. We created as many cores as the number of batches that we split the datasets (250k, 500k, 1m, 1,5m, 2m, 2,5m and 3m). With the following command we create a new core for 250k documents named 'crossref-total250', of the Crossref-total dataset in Apache Solr:

C:\Apache solr\solr-8.6.3\bin>solr create –c crossref-total250

Created new core 'crossref-total250'

After that, to index any data under all the cores that we created, we had to map the respective text fields from the datasets (in this case 'title' and 'abstract'), and

specify the type for each field as **'text_general'**. The field type tells Solr how to interpret the field and how it can be queried, where in the case of our study, we define the fields of type 'text' appropriate for full-text search.

4.2 Experimental setup

In this chapter we discuss the data and query sets, the underlying algorithmic configuration, and the metrics employed in our evaluation.
To form the full-text queries for executing in order to carry out the experimental process, we needed data to be used as incoming documents. For our workload purposes we chose to use initially two different datasets; the first one composed of long text records and a second one composed of short text records. In the following section, we briefly present these datasets along with their content and the transformations we made.

4.2.1 Datasets and data processing

The first dataset that we used in the experiments is the **Crossref dataset.** It consists of publications' metadata and it has been published by the Crossref organization on January 2021 [1,2]. The dataset has a size of 38.5GB, 12.1 million rows and is composed of the fields: 'doi', 'title', 'year_published', 'date_published', 'author', 'journal', 'domain' and 'abstract'. We consider this dataset as the long text dataset of our experiments.
The second dataset that we used in the experiments is the **Yelp reviews dataset.** It consists of reviews and recommendations written and posted by consumers, suggesting restaurants, hotels, bars, shopping, etc. The dataset, which has been downloaded from Kaggle [3], has a size of 3.8GB, 5.3 million rows and is composed of the fields: 'review_id', 'user_id', 'business_id', 'stars', 'date', 'review', 'useful', 'funny', and 'cool'. We consider this dataset as the short text dataset of our experiments.

[1]https://www.crossref.org/
[2]Special thanks to Mr. Christos Tryfonopoulos for providing us with the dataset.
[3]https://www.kaggle.com/datasets

Table 4.1 summarizes some key characteristics for both datasets.

Dataset	**Crossref**	**Yelp reviews**
Format	CSV	CSV
Size	38.5GB	3.8GB
Rows	12.1M	5.3M
Fields	'doi', 'title', 'year_published', 'date_published', 'author', 'journal', 'domain', 'abstract'	'review_id', 'user_id', 'business_id', 'stars', 'date', 'review', 'useful', 'funny', 'cool'
Average vocabulary size (words)	243	111

Table 4.1: Datasets key characteristics.

Below we present some record examples for each one of the datasets.

Crossref

1. *10.1001/2013.jamafacial.65 Zonal Analysis of Facial Asymmetry and Its Clinical Significance in Facial Plastic Surgery 2013 2013-3-1—2013-3-1—N/A Karan Dhir——William Lawson——William J. Binder JAMA Facial Plastic Surgery——JAMA Facial Plastic Surgery Abstract Objectives: To describe common patterns of facial asymmetry and to augment the facial analysis paradigm for improved preoperative counseling and surgical planning. Methods: We conducted a frontal photographic analysis of 50 patients who were seeking various types of facial cosmetic surgical procedures. The horizontal zonal thirds of the face were analyzed, and the bilateral data points were compared in regard to brow height, width of midface at maximum distance, malar eminence height,nasal alar height, and mandible width measured from the oral commissure to the gonial angle. Results: Forty-five patients demonstrated measurable asymmetry of the midface. The malar eminence was found to be more superiorly positioned and defined on the narrower side of the face in all cases. In contrast, the contralateral wider side of the face appeared flatter, with a more hypoplastic, inferiorly positioned malar eminence.Also, the wider side of the face more often demonstrated a wider mandibular dimension and a superiorly displaced ala. The upper third of the face,*

in regard to brow height, was the most variable and showed little correlation to the lower two-thirds of the face. Conclusion: This facial analysis exercise can assist the surgeon in (1) preoperative counseling, (2) managing expectations,(3) choosing appropriate-sized implants for improved symmetry, and (4) offering a more detailed assessment during the counseling of patients before faceliftsurgery.

2. *10.1016/s0012-821x(97)00079-4 The influence of silicate melt composition on distribution of siderophile elements among metal and silicate liquids 1997 1997-8—1997-8—N/A Dipayan Jana——David Walker Earth and Planetary Science Letters——Earth and Planetary Science Letters N/A Abstract Liquid metal-liquid silicate partitioning of Fe, Ni, Co, P, Ge, W and Mo among a carbon-saturated metal and a variety of silicate melts (magnesian-tholeiitic-siliceous-aluminous-aluminosiliceous basalts) depends modestly to strongly upon silicate melt structure and composition. Low valency siderophile elements, Fe, Ni and Co, show a modest influence of silicate melt composition on partitioning. Germanium shows a moderate but consistent preference for the depolymerized magnesian melt. High valency siderophile elements, P, Mo, and W, show more than an order of magnitude decrease in metal-silicate partition coefficients as the silicate melt becomes more depolymerized. Detailed inspection of our and other published W data shows that polymerization state, temperature and pressure are more important controls on W partitioning than oxidation state. For this to be true for a high and variable valence element implies a secondary role in general for oxidation state, even though some role must be present. Equilibrium core segregation through a magma ocean of 'ultrabasic' composition can provide a resolution to the 'excess' abundances of Ge, P, W and Mo in the mantle, but the mantle composition alone cannot explain the excess abundances of nickel and cobalt in chondritic proportions.*

Yelp reviews

1. *vkVSCC7xljjrAI4UGfnKEQ, bv2nCi5Qv5vroFiqKGopiw, AEx2SYEUJmTxVVB18LlCwA, 5, 2016-05-28, "Super simple place but amazing nonetheless. It's been around since the 30's and they still serve the same thing they*

started with: a bologna and salami sandwich with mustard. Staff was very helpful and friendly.", 0,0,0

2. *5Clsq9QSbz4GWCS8DGDhQQ, s2tUilH-0FHdBQL8fAzv2w, C-cvl8Mf2vpxHwUon3sVGg, 4, 2015-03-31, "Decided to give a chance since it was close by. Decided on the Pollo Fundido based on the reviews and it did not disappoint. Very large portion and very tasty. I would not hesitate to order this again. Based on one meal, I would give it a try.", 0,0,2*

For the purposes of our workload and to measure the performance of querying and insertion time for each database system, we conducted the experiments under different sizes of the initial datasets. For this reason, we formed three new kinds of datasets out of the initial ones, with regard to the datasets' size: small, medium and large, including also those fields with textual content. At first, we consider the dataset **'Crossref-total'**, which consists of the fields 'title' and 'abstract' of the initial Crossref dataset as the **large** one, the dataset **'Yelp reviews'**, which consists of the fields 'review_id' and 'review' as the **medium** one, and finally, the dataset **'Crossref-title'**, which includes only the field 'title' of the initial Crossref dataset, as the **small** one of the datasets.
Furthermore, in order to perform the experiments on varying sizes of the datasets, we randomly split the three datasets, each one, into smaller parts (batches) of 250k, 500k, 1m, 1,5m, 2m, 2,5m and 3m items (rows), setting the 250k items and the 3m items as the lowest and the upper limits respectively.
The resulted datasets and their key characteristics are summarized in Table 4.2.

Dataset	**Crossref-total** (large)	**Yelp reviews** (medium)	**Crossref-title** (small)
Format	CSV	CSV	CSV
Size	4.1GB	1.8GB	286MB
Rows	3M	3M	3M
Fields	'title', 'abstract'	'review_id', 'review'	'title'
Avg vocabulary size (words)	228	104	14

Table 4.2: Large, medium and small datasets characteristics.

Further operations of data massaging were applied to the datasets such as combining, merging and handling missing data. For all the data processing operations we used the Python's **pandas** and **nltk** libraries.

4.2.2 Full-text search queries

To set up the process of comparison for measuring the querying time by executing the full-text queries, first we had to extract the appropriate keywords from the datasets and to form different groups of queries with those keywords. To do this, we first chose to extract the words with the most frequent occurence from each one of the initial datasets, after removing any stopwords, symbols, punctuation marks etc., in order to set up the basis for our experiments in terms of selectivity (see Chapter 4 - Subsection 4.3.5. *Selectivity*). In addition, to avoid the possibility of having divergent results from the queries we executed, we chose the words with the most frequent occurence among all batches of data (i.e., from the range of 250k records to 3m records) and then we selected a subset of these words (the common words from all batches) to put in performance. Following this procedure, we then extracted the keywords with the least appearance in the datasets.
A sample of the top-15 keywords, arranged in descending order of occurrence, is shown in Table 4.3. (the numbers in the parentheses indicate the number of appearances at 3m records).

Crossref	study (329.610), results (301.070), using (293.848), patients (255.385), data (230.021), different (216.734), model (190.041), analysis (178.774), new (174.924), based (173.308), high (172.030), significant (142.823), method (141.957), compared (141.034), paper (140.946).
Yelp reviews	place (956.687), food (915.323), great (834.980), like (826.913), good (792.220), one (706.965), service (660.446), really (648.134), time (647.751), go (616.856), back (548.831), always (470.391), best (455.288), love (447.525), nice (445.925).

Table 4.3: Keywords with the most frequent occurence.

Notice that, the keywords extracted from the Yelp reviews dataset are three times

more than those extracted from the Crossref dataset which makes sense, considering the fact that those keywords are most commonly used. We assume that this may have an impact in the comparison process and we will examine any potential of this finding in the experiments that follow.

Next, we composed pairs of keywords (collocations) that match together and are present in sentences. The majority of these pairs derived from the keywords that have been extracted previously. An example of these pairs, is presented in Table 4.4.

Crossref	study results, different species, negative effects, new treatment, early studies, clinical data, increased number, paper presents, high number, system based.
Yelp reviews	great place, good service, nice people, friendly staff, next time, great food, new restaurant, every time, wonderful dinner, go back.

Table 4.4: Collocations.

As we have mentioned before, the selection of keywords and collocations has taken place with regard to selectivity. The selectivity of a specific value is the number of rows with that value, divided by the total number of rows. To test the database systems we selected the appropriate keywords and collocations in order to form three levels of selectivity: **high**, **medium** and **small** and put in performance queries of high, medium and small selectivity. For example, if we operate with the queries of small selectivity on the database size of 3m total records, this would return an estimated 60.000 records which results in a selectivity value of 0.02 (2%). Similarly, queries of medium selectivity (6.000 records) will result in a selectivity value of 0,002 (0,2%) and high selective queries (600 records) will result in a selectivity value of 0,0002 (0,02%). To do this, we formed high selective queries with groups of keywords and collocations that appear a few times inside texts (the keywords with the least appearance in the datasets we have previously extracted), queries of small selectivity with groups of keywords and collocations with the most frequent appearances and queries of medium selectivity in between. These groups consist of 10 keywords and 10 collocations for each level of selectivity.

Afterwards, in order to conduct the experiments for measuring the performance of each database system, we used these queries under the following query operations:

- **Exact phrase matching**
- **Wildcards search**
- **Conjunctive queries**

Based on the example we have given previously, Table 4.5 presents the different levels of selectivity we applied for each one of the query operations.

Selectivity	**High**	**Medium**	**Small**
Exact phrase matching	0,02%	0,2%	2%
Wildcards search	0,25%	2,5%	25%
Conjunctive queries	0,08%	0,8%	8%

Table 4.5: Selectivity.

Notice that, for the wildcards search operation the proportion of the records that have been queried in the database is much larger than the other two operations. The reason for this, is that in wildcards search we put in performance groups of keywords and not groups of collocations. We will examine later in the experiments section any potential of this aspect.

Exact phrase matching. Querying a database using exact phrase matching will return results that match a phrase, containing all components of the keywords, word by word, in the same order. To form a query for exact phrase matching we should enclose the entire phrase in quotation marks. For instance, the syntax to query the phrase *'study results'* in the 'abstract' field of the Crossref-total dataset, for each one of the database systems, would be:

- **Postgres:** *SELECT abstract FROM crossref-total250 WHERE document @@ phraseto_tsquery('study results')*

- **Mongo:** *find({"$text": {"$search":"\"study results\""}})*
- **Elasticsearch:** *"query": {"match_phrase": {"query": "study results", "type": "phrase", "fields": "abstract"}*
- **Apache Solr:** *abstract: "study results"*

Postgres uses the **tsvector** and **tsquery** data types. We have previously created a new **'document'** column setting it to the tsvector data type (see Chapter 4 - Section 4.1. *Creating databases and indexing data*). Especially for the exact phrase matching query group, we introduce the *phraseto_tsquery()* function, which supports phrases text search instead of using the *to_tsquery()* function. To perform the same query in Mongo, we use the **$text** operator inside the *find()* method. The 'abstract' column has been previously indexed as a new text column, in order to perform any full-text operations in Mongo (see Chapter 4 - Section 4.1. *Creating databases and indexing data*). Finally, Apache Solr executes the search query with the use of the 'Lucene' parser as well as the Elasticsearch does, also using the Query DSL syntax.

Wildcards search. Wildcards search is an advanced technique that can be used to maximize the search results. Wildcards are used in search terms to represent one or more other characters. An asterisk (*) is used as the wildcard symbol, to specify any number of characters. Supposing we wish to query the postfix wildcard *'spec*'* in the 'abstract' field of the Crossref-total dataset, the syntax for each database system would be:

- **Postgres:** *SELECT abstract FROM crossref-total250 WHERE document @@ to_tsquery('spec*')*
- **Mongo:** *find({"abstract": {"$regex": "spec*", "$options":"i"}})*
- **Elasticsearch:** *"query": {"query_string": {"query": "spec*", "fields": "abstract"}*
- **Apache Solr:** *abstract: spec**

We observe that Postgres in this case, executes the query using the *to_tsquery()* function. In Mongo we use the **$regex** operator instead of the $text operator. The $regex operator provides regular expression capabilities for pattern matching strings in queries. We have also used the 'i' option with the query, which performs case-insensitive regular expression matches inside documents. To submit the same query in Elasticsearch we use the *query_string*, instead of the *match_phrase*, which better perfoms in wildcards search. Apache Solr executes the query once more with the use of the 'Lucene' parser, with as the same syntax as in the exact phrase matching.

Conjunctive queries. A conjunctive query is a restricted form of first-order queries using the logical conjunction operator *AND* ($\wedge$). The logical conjunction operator is used to check that two or more conditions are true. To execute the conjunctive query *'negative AND effects'* in the 'abstract' field of the Crossref-total dataset, the syntax for each database system would be:

- **Postgres:** *SELECT abstract FROM crossref-total250 WHERE document @@ to_tsquery('negative & effects')*
- **Mongo:** *find({"$text": {"$search": "\"negative\" \"effects\""}})*
- **Elasticsearch:** *"query": {"query_string": {"query": "negative AND effects", "fields": "abstract"}*
- **Apache Solr:** *abstract: 'negative' AND abstract: 'effects'*

What we observe in this case is that, except for Mongo, the logical conjunction operator *AND* ($\wedge$), is used by the rest of the systems for executing the conjunctive query. In Mongo, the AND operator is used indirectly between the search terms, by combining quotes and space.

Note: As we have already mentioned, several Python clients have been used for the connection to the databases and execution of the queries for each system in the experiments (see Chapter 3 - Section 3.2. *Technical configurations*). For this reason,

the syntax in the scripts of the experimental evaluation (which has been written in Python) is slightly different.

4.2.3 Technical configurations

In order to perform our experimental evaluation, we first had to install and configure the database systems that we introduced in the previous chapter. In this section we discuss the technical configurations and the requirements for each one of the database systems that we installed, used and compared in our experiments.
The database systems versions that we used in our machine locally for the performance evaluation are:

- **PostgreSQL:** 13.3
- **MongoDB:** 4.4.2
- **Elasticsearch:** 7.10.0 (installed together with Kibana 7.10.0 as part of the ELK stack)
- **Apache Solr:** 8.6.3

All programming operations and query executions were implemented in the Python language and a PC (Core i3, 2GHz processor, 6GB RAM, Windows 10) was used to run the experiments.
A number of Python libraries and modules also used to achieve the connections with the database systems and execute the queries for each system. More specifically, to connect to Postgres and execute the SQL queries, we installed and used the **'psycopg2'** database adapter for Python. To be able to connect to MongoDB we used a MongoClient by installing a native Python driver for Mongo, **'pymongo'**. To settle a connection with Elasticsearch server in order to index and search for data, we used a Python client for Elasticsearch and more specifically the **'elasticsearch'** package. Finally, to achieve HTTP connections with Apache Solr and perform tasks with REST APIs, we used the **'urllib.requests'** module.

4.3 Comparing querying time

In this section, we present a series of experiments that compare the four database systems presented in the previous chapters, i.e. Postgres, Mongo, Elasticsearch and Solr. The comparison of the database sytems involves three fundamental operations in terms of querying time: exact phrase matching, wildcards search and conjunctive queries. Our experiments measure the performance on these operations on all three datasets, that we have also analysed (i.e., small, medium and large), for the chosen databases.

We have also conducted experiments comparing extended conjunctive queries, in which more than two words are included in conjunctions (conjunctions with three and four words specifically) and experiments comparing queries' selectivity. In that case, we test how much selective a query can be, as the database systems perform with queries of high, medium and small selectivity (see Chapter 4 - Subsection 4.3.5. *Selectivity*). Finally, we have conducted experiments in which we compare the performance between the aforementioned operations (exact phrase matching, wildcards search and conjunctive queries), where in this thesis, we entitle *"query groups"*.

The baseline setup in the experimental evaluation comprises a database size of 3m items and we study the performance of each system as the database size increases (see Chapter 4 - Subsection 4.2.1. *Datasets and data processing*). The time shown in the graphs is wall-clock time and the results of each experiment are averaged over five runs to eliminate fluctuations in time measurements. Finally, in the sections that follow, we present experiments involving data insertion time and memory usage for data indexing and querying for each one of the database systems. All results are summarized in the conclusion section.

4.3.1 Exact phrase matching

In this section we present the findings that derived from the experiments, comparing the systems' quering time on exact phrase matching. For this set of queries we use phrases that consist of pairs of words (or collocations, see Chapter 4 - Subsection

4.2.2. *Full-text search queries*). After we executed these operations, we obtained the results that are shown in the figures [4] that follow.

In Figures 4.1, 4.2 and 4.3, we can see the results comparing the database systems in regard to the database size for each one of the datasets, (i.e., small, medium and large). As the database size gets larger the execution time for each system increases, which is a logical outcome at first. Moreover we can see that the gap in time measurements between the text search systems (i.e., Elasticsearch and Solr) and the rest of the them (i.e., Postgres and Mongo) increases, while operating with the medium and the large datasets something which not happens with the small dataset, where time measurements are relatively close. This fact indicates that the systems can be quite sufficient while operating with small amounts of data. Elasticsearch and Solr though, seem to achieve the best performance among all the database systems in general. We also observe that Mongo and Postgres need significantly more time compared to Elasticsearch and Solr to execute the queries.

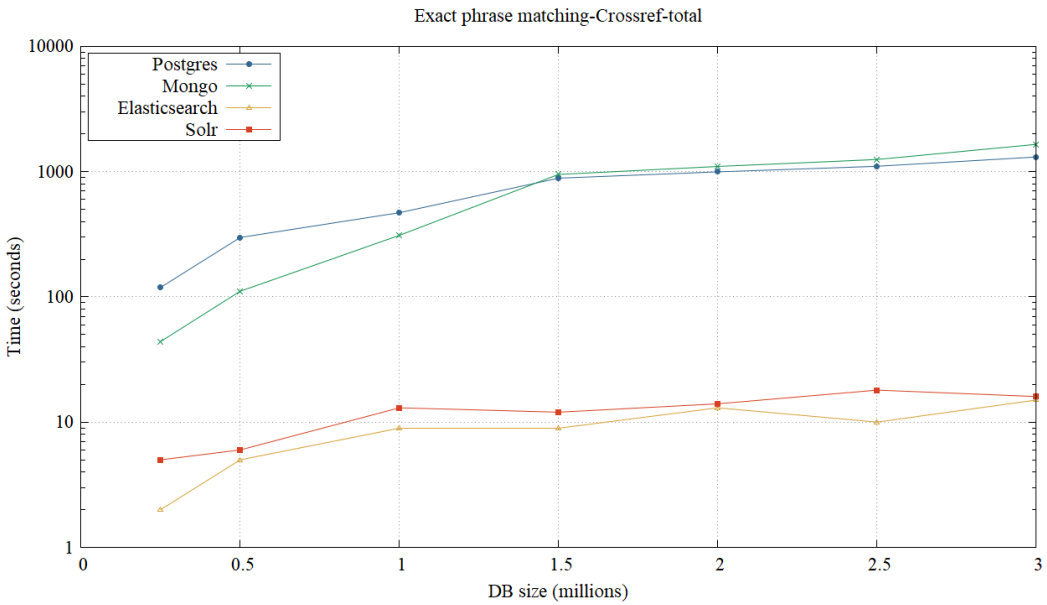

Figure 4.1: Exact phrase matching Crossref total.

[4] All diagrams are displayed in logarithmic scale.

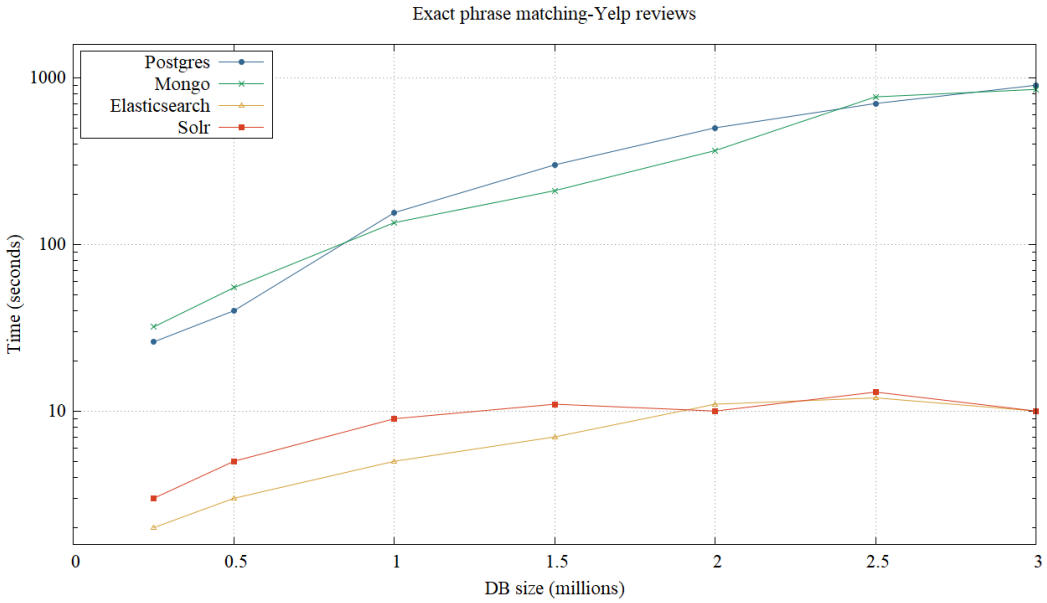

Figure 4.2: Exact phrase matching Yelp reviews.

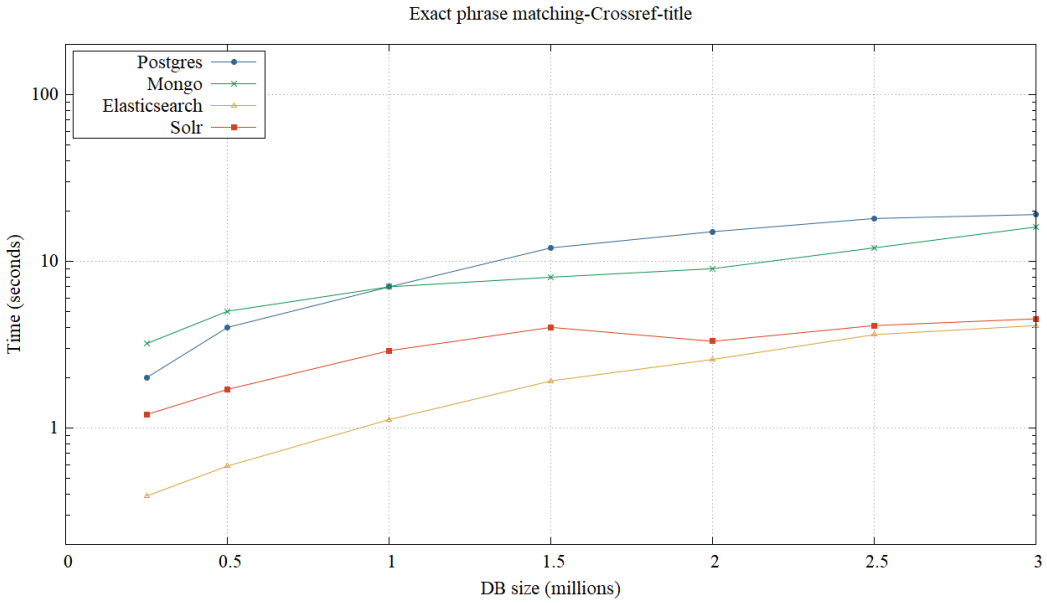

Figure 4.3: Exact phrase matching Crossref title.

In Figures 4.4, 4.5, 4.6. and 4.7, we can see an alternative for the previous diagrams and we present the time measurements for each database system separately. We notice that there is a linear increase in time measurements as the database size gets larger. All systems achieve better results with the small dataset, as we have identified previously, Elasticsearch and Solr though, perform same as well with the

medium and large datasets. We also observe that the runtimes measured for the medium dataset for each system, are relatively close to those of the large one. This fact confirms the assumption we made previously about the words occurence in the Yelp reviews dataset compared to those of the Crossref dataset and the impact that this fact would have in the experiments.

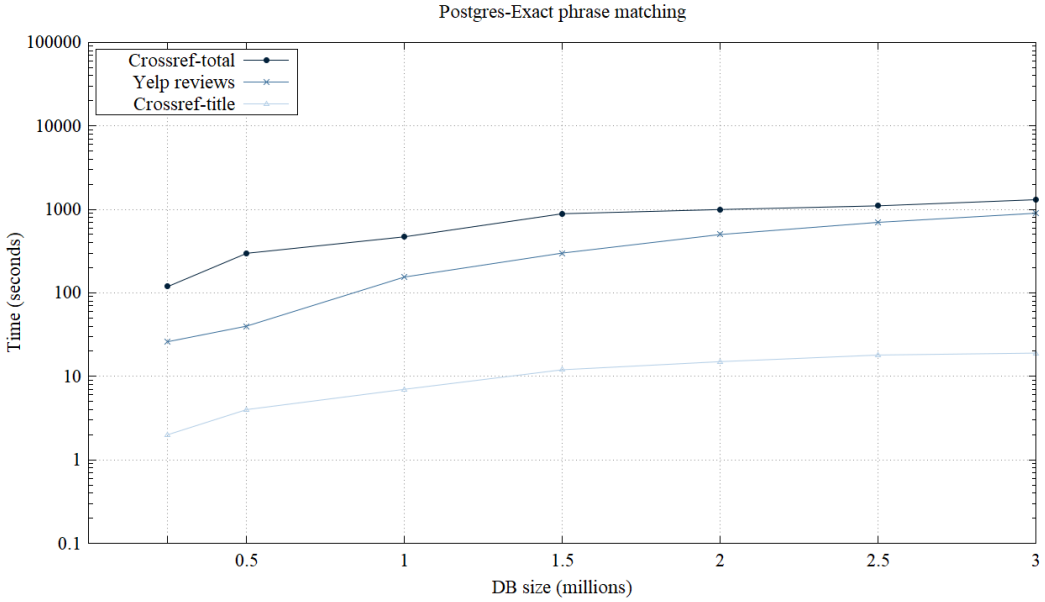

Figure 4.4: Postgres Exact phrase matching.

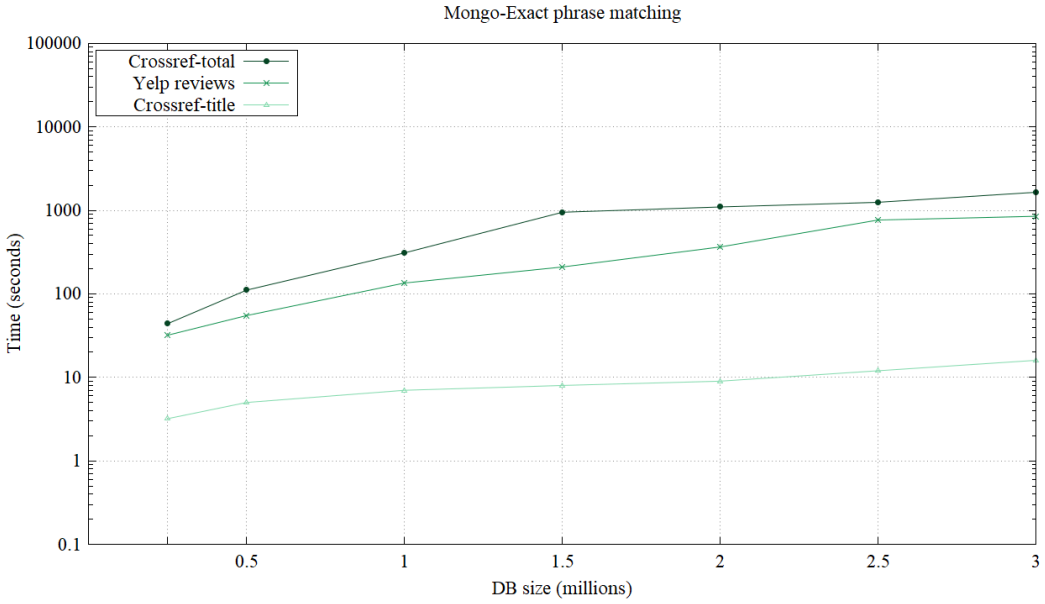

Figure 4.5: Mongo Exact phrase matching.

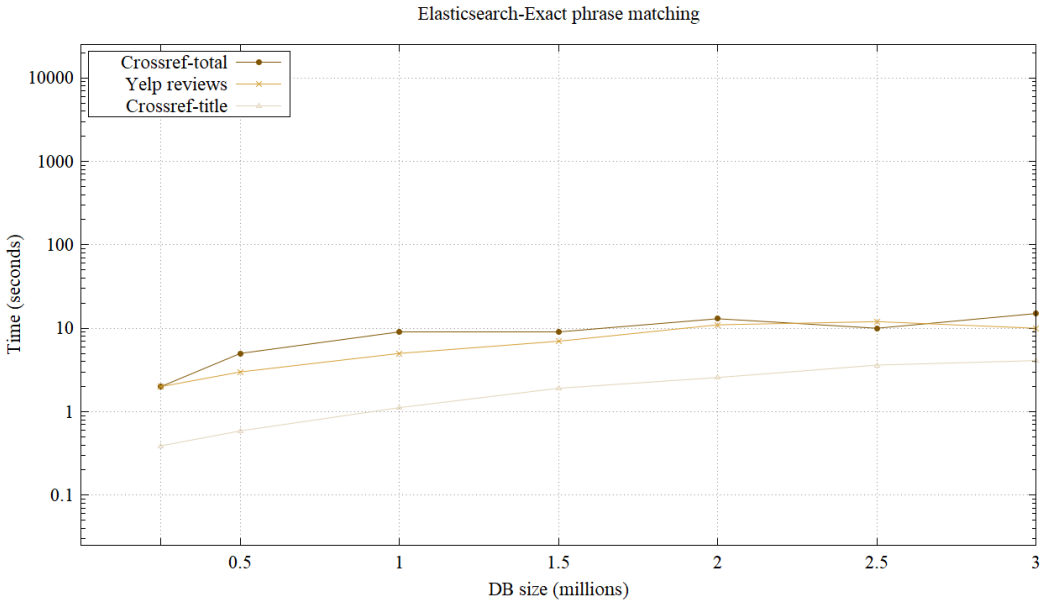

Figure 4.6: Elasticsearch Exact phrase matching.

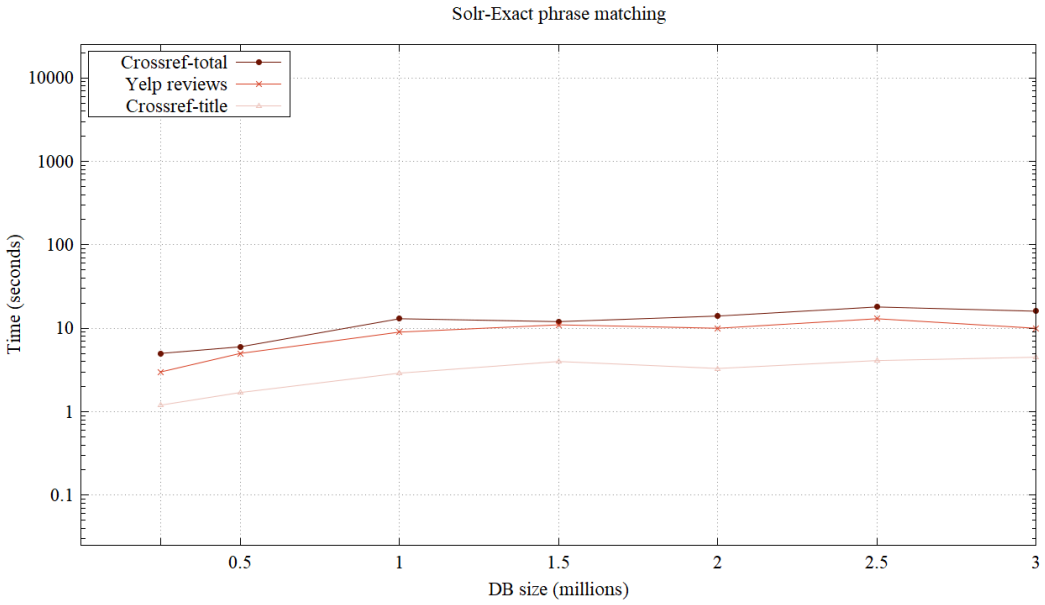

Figure 4.7: Solr Exact phrase matching.

However, Postgres operates much faster with the use of the GIN index to query the data, in contrast to operating without the use of it. As an example, we highlight this difference in performance in Figures 4.8, 4.9 and 4.10, for each dataset size, where we can see the lower time measurements that Postgres with GIN index can achieve. It is also noteworthy the fact that Postgres cannot respond in the experiments while performing without the use of the GIN index with the large

dataset, after the limit of 1.5m documents. Due to these reasons, for the rest of the operations that we have experimented with Postgres in this work, we examine its performance only by using the GIN index.

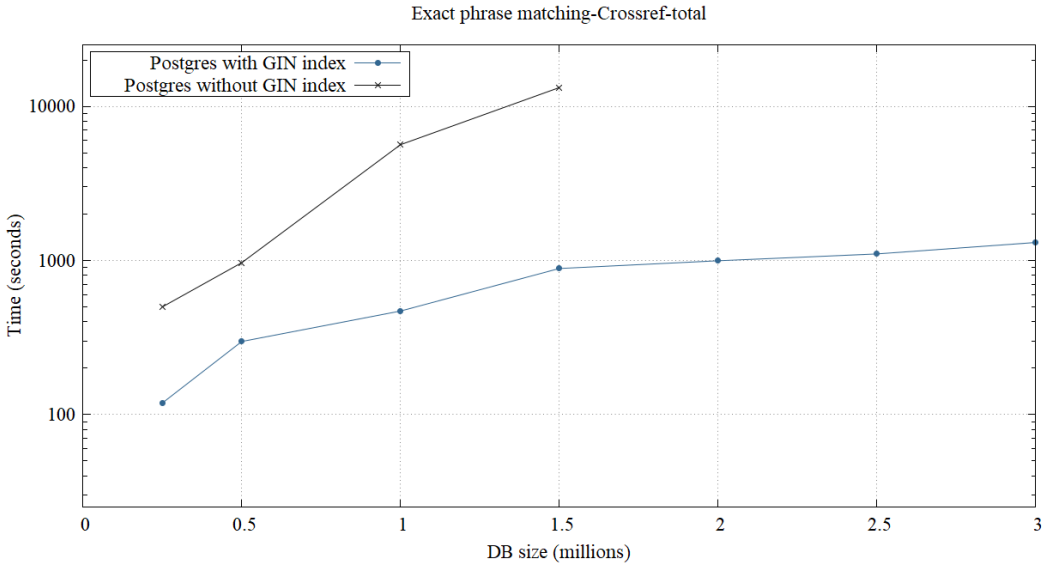

Figure 4.8: Postgres GIN index Crossref total.

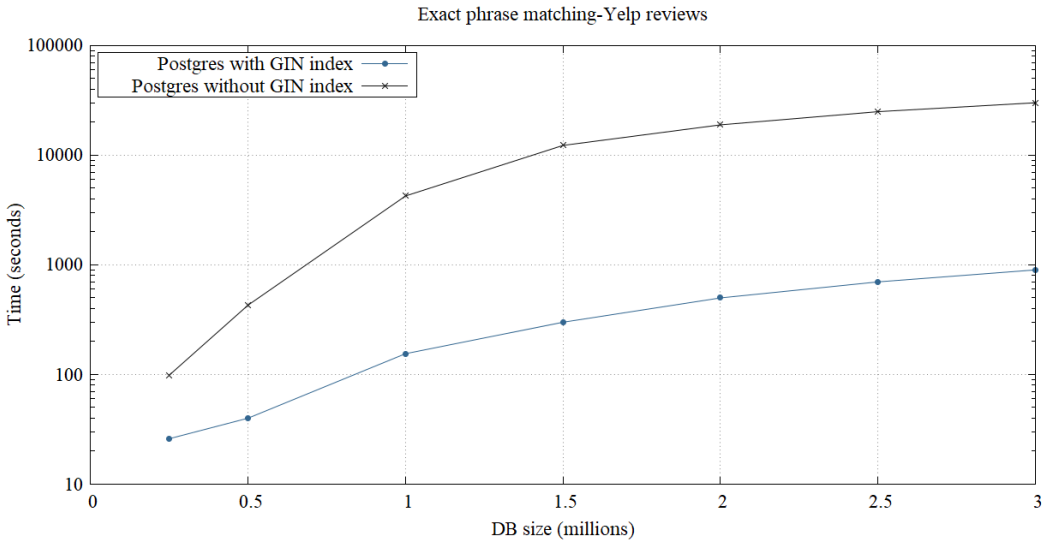

Figure 4.9: Postgres GIN index Yelp reviews.

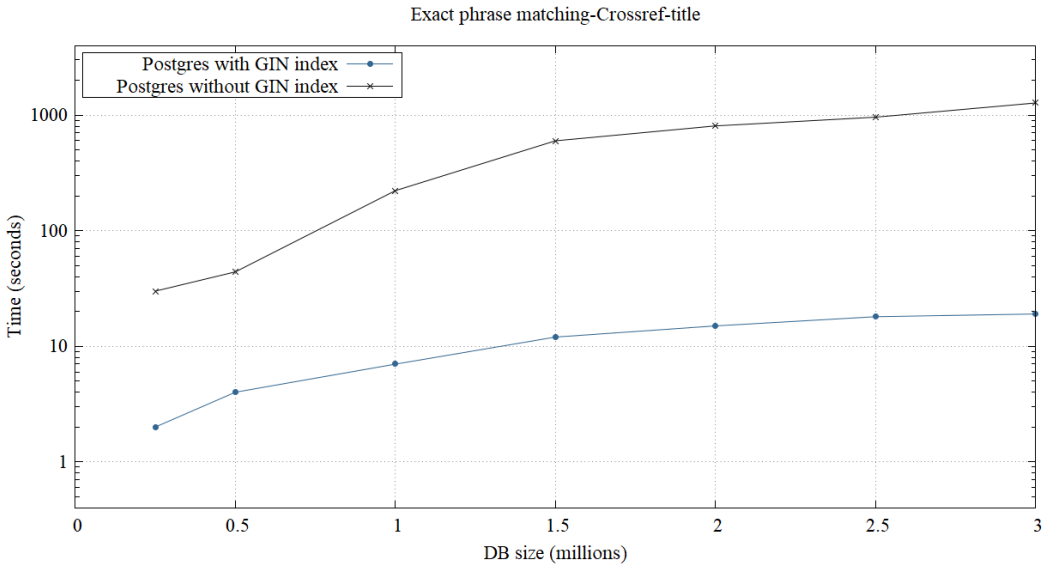

Figure 4.10: Postgres GIN index Crossref title.

For the exact phrase matching set of queries, as well as for wildcards search and conjunctive queries, we test the databases while operating with highly selective queries (the experiments regarding *Selectivity* will be presented in a following section, as we have already mentioned). It is interesting though, to test the systems' performance, while operating with queries of small selectivity this time. Testing the systems with queries of small selectivity means that we query the 2% of the database, in contrast to high selective queries where we query the 0,02% of the database, as we have previously described in subsection 4.2.2. Full-text search queries.
The results from this set are shown in Figures 4.11, 4.12 and 4.13.

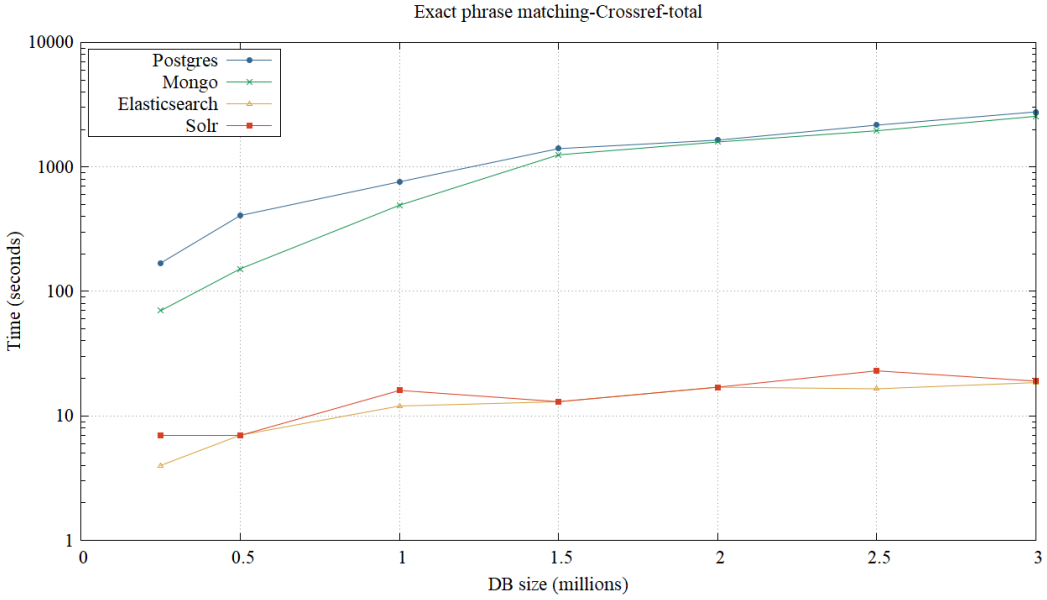

Figure 4.11: Exact phrase matching Crossref total (small selectivity).

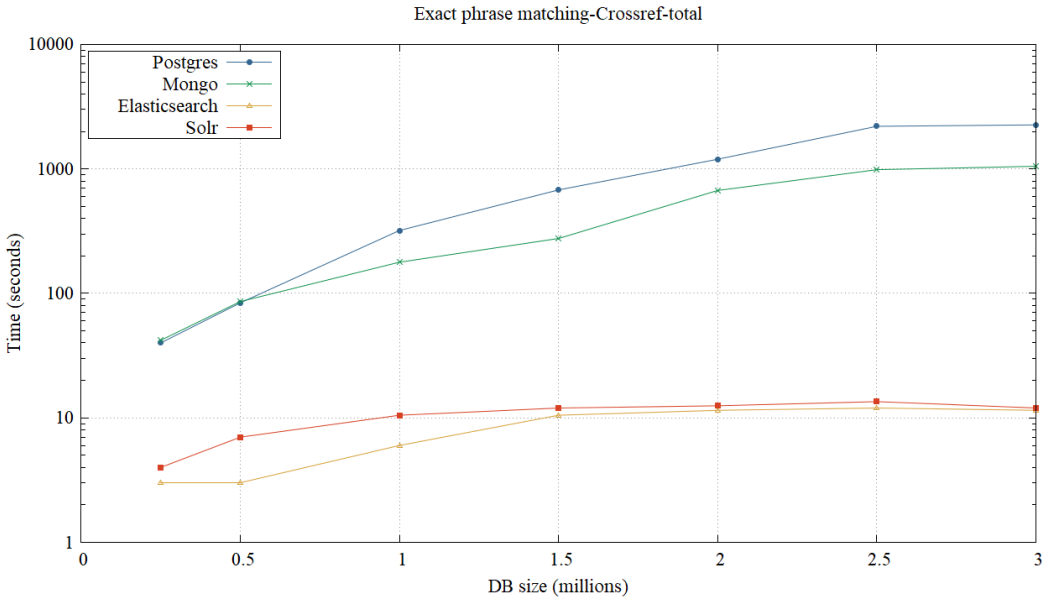

Figure 4.12: Exact phrase matching Yelp reviews (small selectivity).

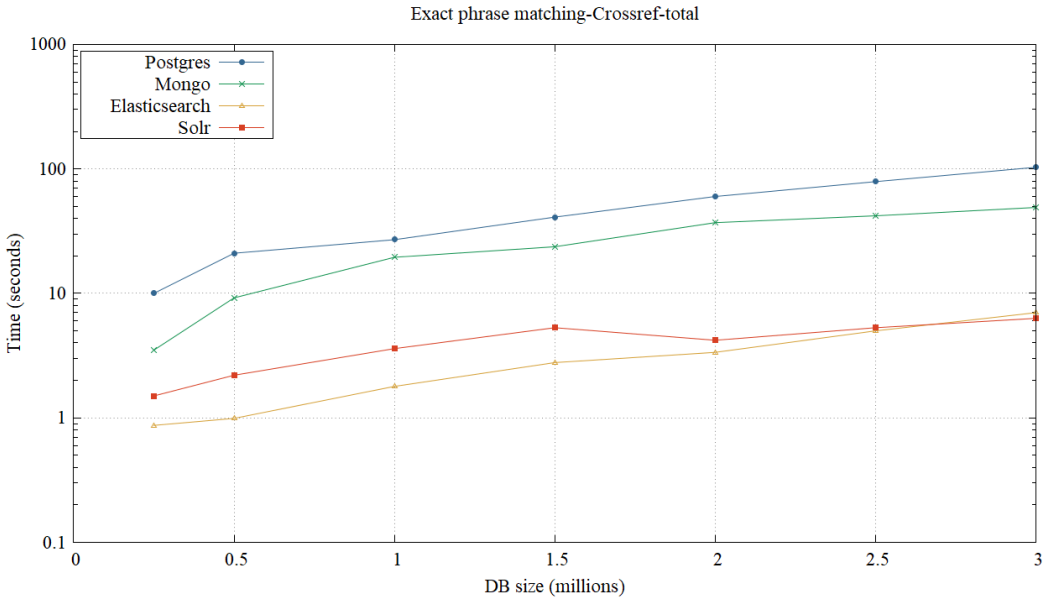

Figure 4.13: Exact phrase matching Crossref title (small selectivity).

In this set of experiments Postgres needs more time to operate. The difference in time measurements is significantly larger compared to those of high selective queries, especially in the medium and small datasets. There are minor differences though in time measurements regarding the rest of the systems which keep steadily the time variance among the datasets, compared to the tests we executed with the high selective queries. We also notice that Postgres and Mongo make an entry from the same starting point at 250k documents, ending up at 3m documents with a significant time difference while performing in the medium dataset. The exact opposite happens in the large dataset where the systems are relatively close at 3m documents. Postgres and Mongo are pretty much close in time measurements while performing in the exact phrase matching operation, something that not happens in the rest of the operations we will examine later.

4.3.2 Wildcards search

Our second experiment measures the time needed to query the databases with wildcards search. For this set of queries we use the groups of keywords we have previously formed (see Chapter 4 - Subsection 4.2.2. *Full-text search queries*), followed by a wildcard symbol (an asterisk), witch specifies any number of characters.

The graphical representations of the results are shown in Figures 4.14, 4.15 and 4.16. At first place, we compare the database systems in regard to the database size for each one of the datasets, (i.e., small, medium and large). As we observe, all systems perform much better compared to exact phrase matching, despite the large proportion of the records that it query in wildcards search (see Chapter 4 - Subsection 4.2.2. *Full-text search queries*). Elasticsearch and Solr still need less than 10 seconds to execute the queries for all datasets. It is also stunning how fast Postgres performs in this set of queries and proves to be same as fast as the rest of the systems. With the use of the GIN index Postgres seems to perform much better when querying a single word inside a text, in contrast to exact matches in the experiments we have examined before. Also notice that, as Postgres achieves better time measurements at 250k documents than Elasticsearch and Solr in the large dataset, it converges with the systems on the limit of 3m documents. This could be a proof that non-relational and text search databases prevail over the relational ones as data grow in size. Mongo finally, also performs much better in this set, as it achieves half of the time in the medium and large datasets, compared to the time measured in exact phrase matching.

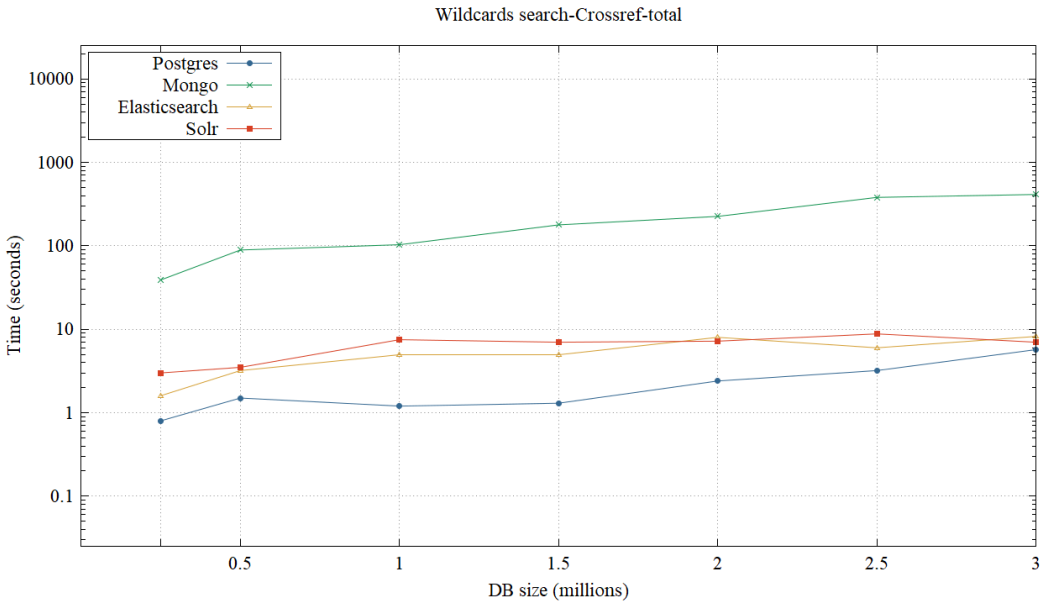

Figure 4.14: Wildcards search Crossref total.

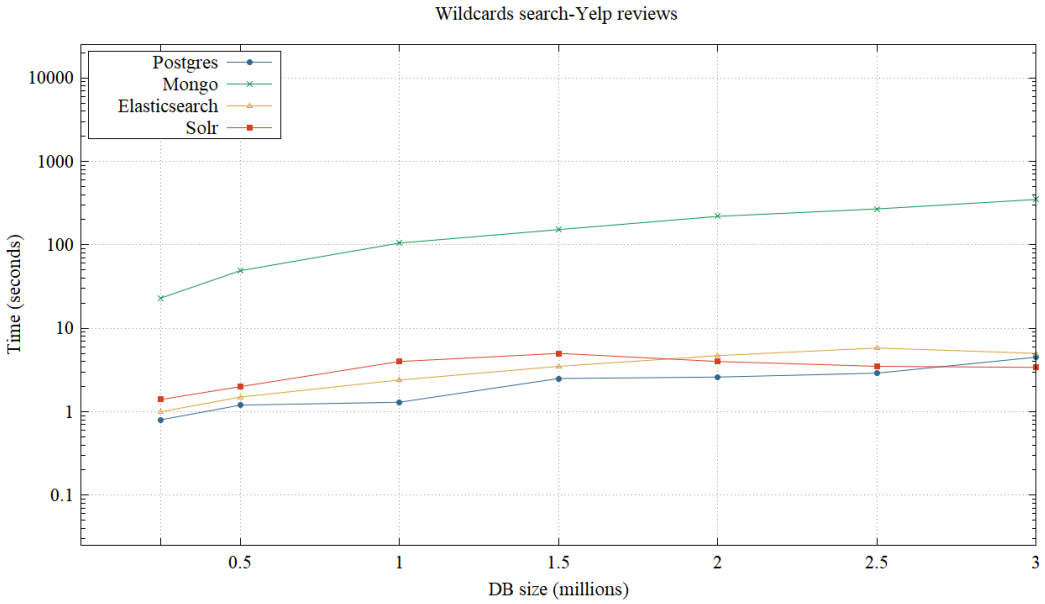

Figure 4.15: Wildcards search Yelp reviews.

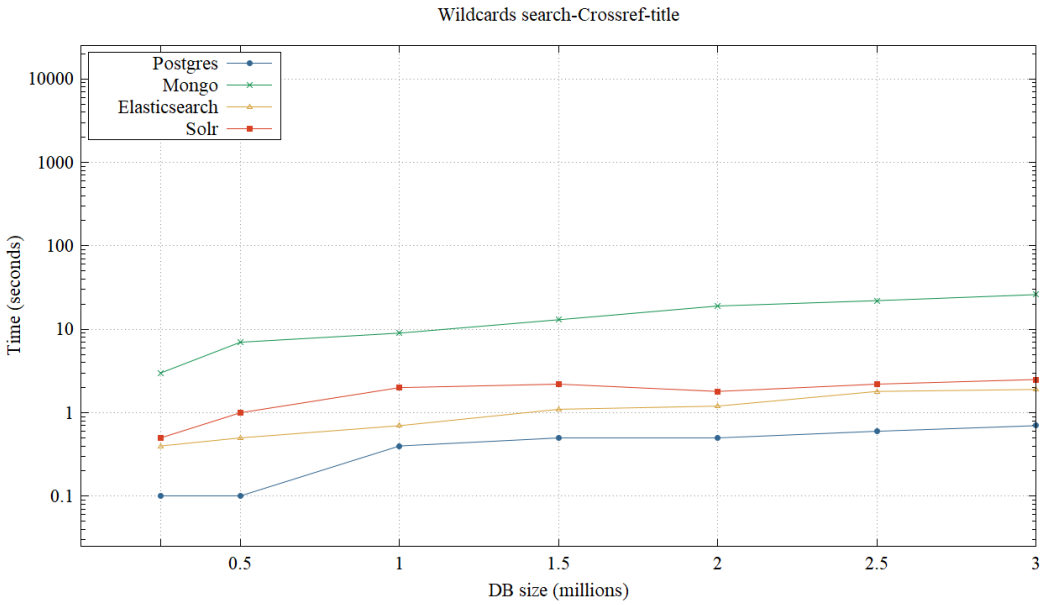

Figure 4.16: Wildcards search Crossref title.

In Figures 4.17, 4.18, 4.19. and 4.20, we can see an alternative view for the previous experiments, comparing the time needed to query the datasets (i.e., small, medium and large), in regard to the database size for each system respectively. Again, for the reasons explained previously, we can see Postgres's performance in detail and verify the low time measurements, which in the case of the small dataset needs 0,1 second to query 500k records.

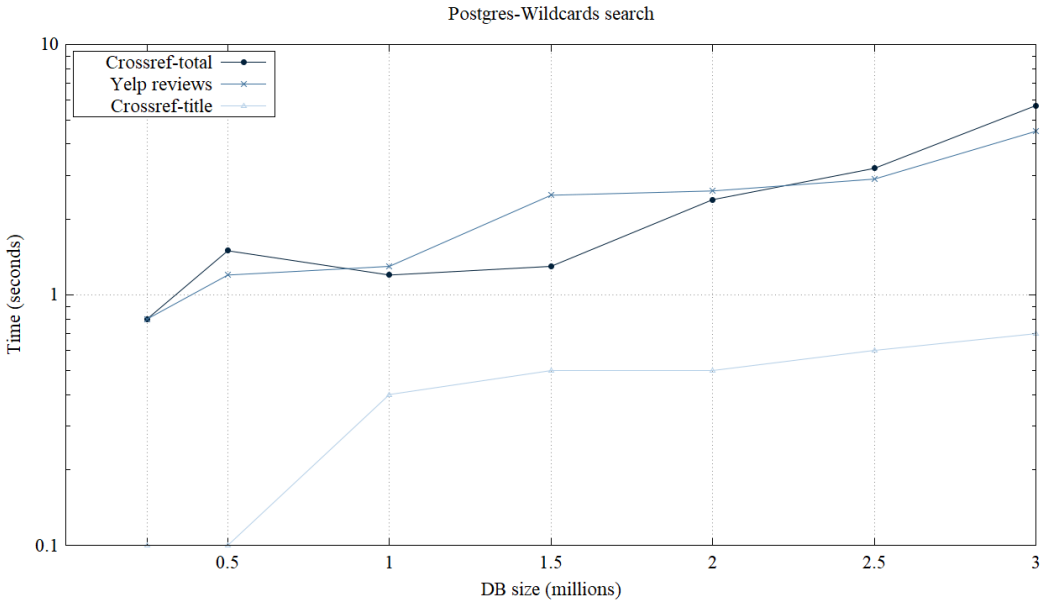

Figure 4.17: Postgres Wildcards search.

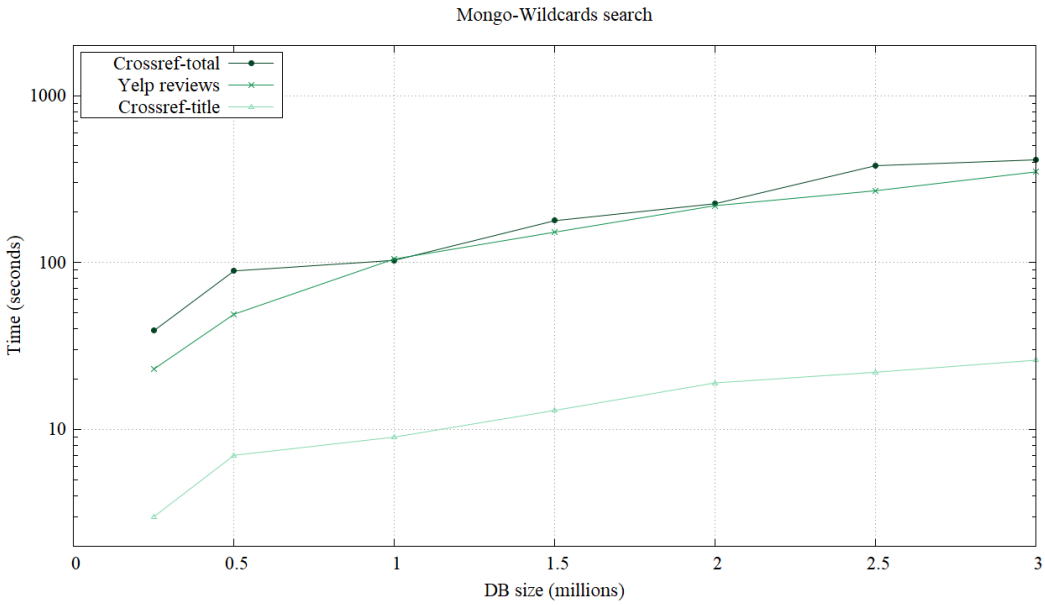

Figure 4.18: Mongo Wildcards search.

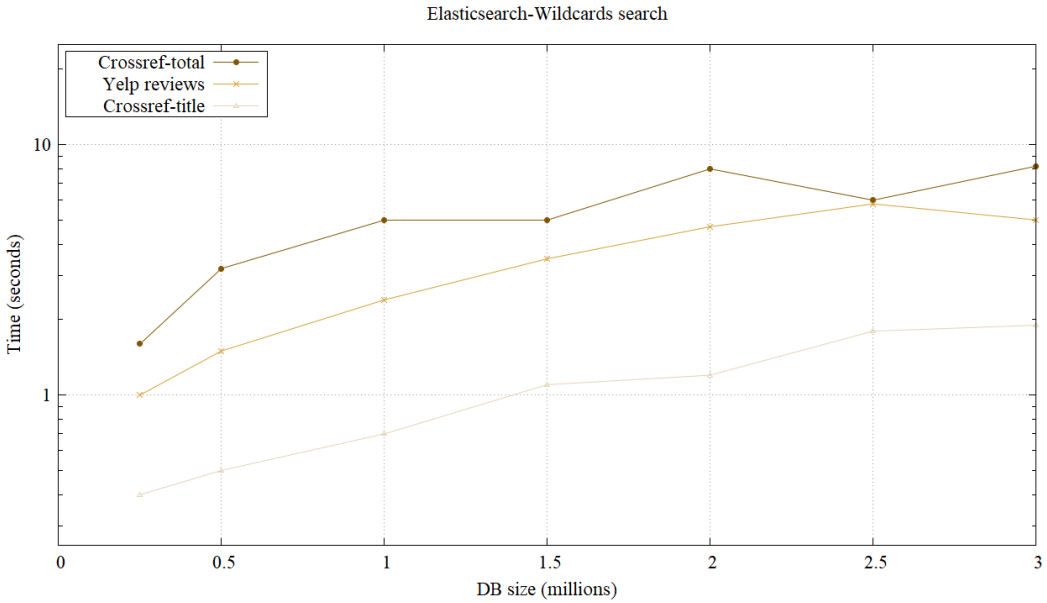

Figure 4.19: Elasticsearch Wildcards search.

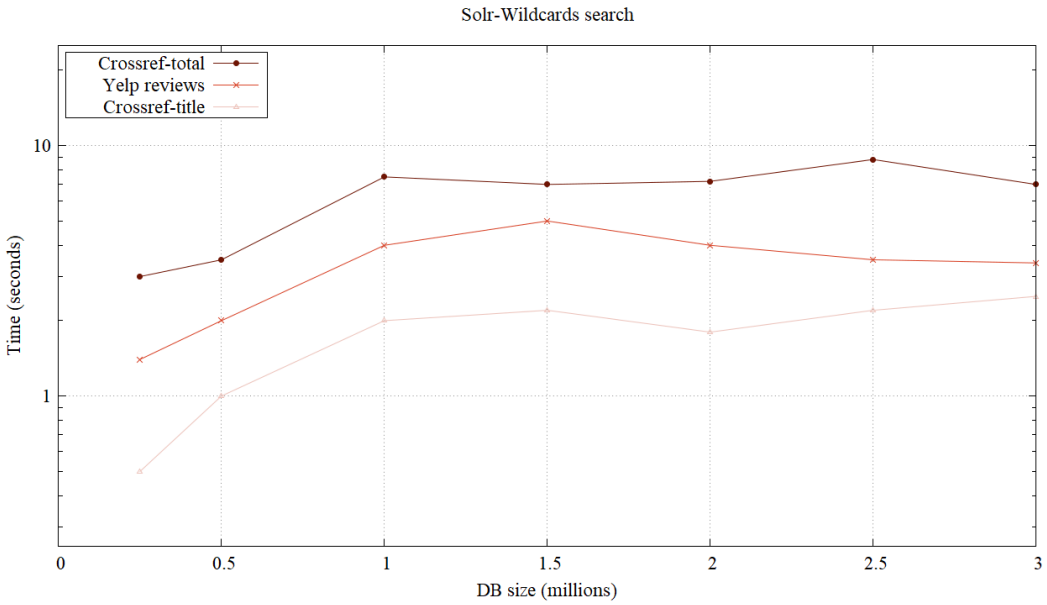

Figure 4.20: Solr Wildcards search.

Once more we can see that the runtimes measured for the medium dataset for each system, are close (or overlap in the case of Postgres) to those of the large one, due to the words occurence in the medium dataset.

Similarly with the exact phrase matching operation, we have tested the systems' performance in the wildcards search while operating with queries of small selectivity. The results that we obtain are presented in Figures 4.21, 4.22 and 4.23.

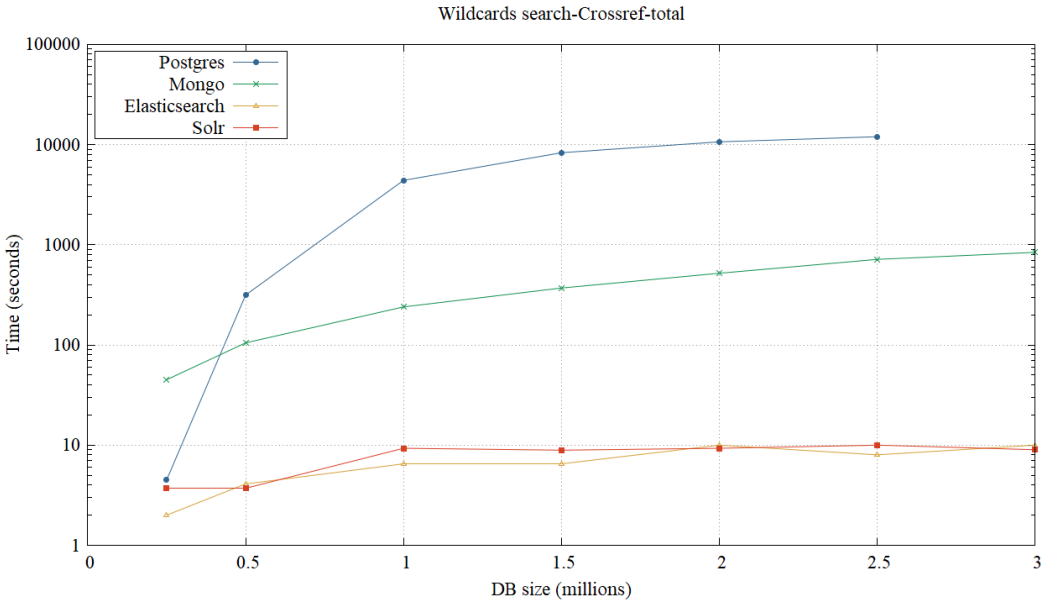

Figure 4.21: Wildcards search Crossref total (small selectivity).

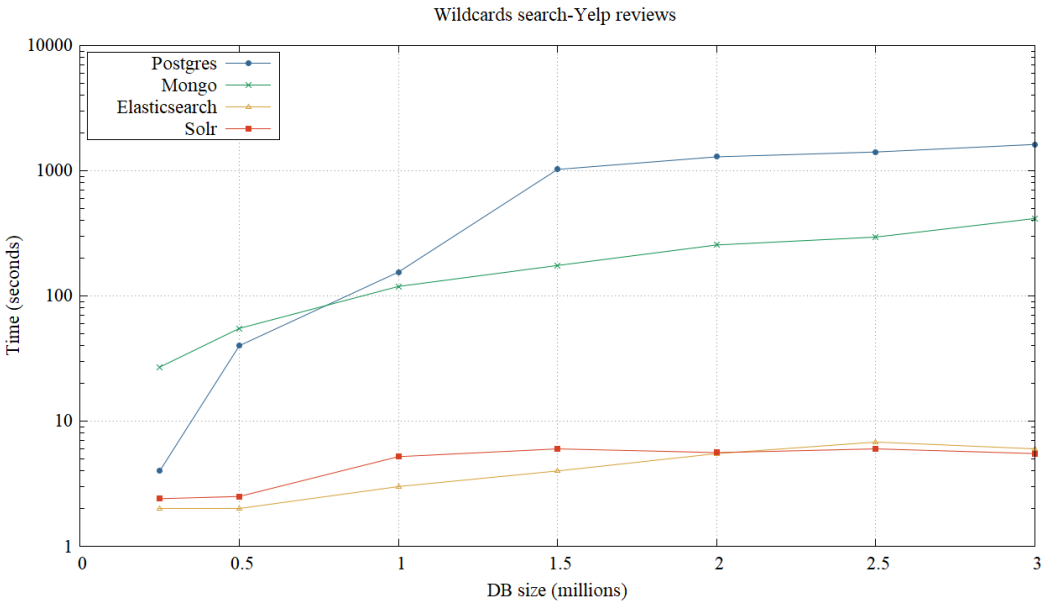

Figure 4.22: Wildcards search Yelp reviews (small selectivity).

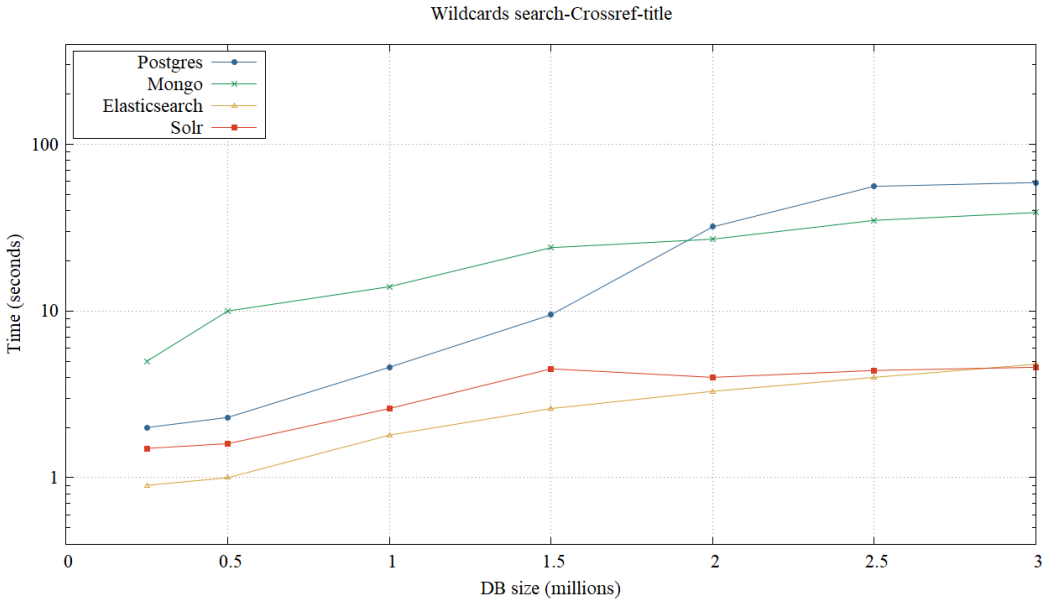

Figure 4.23: Wildcards search Crossref title (small selectivity).

Similar to the experiments of exact phrase matching, Postgres also needs more time to operate with queries of small selectivity in the wildcards search. In this scenario Mongo, Elasticsearch and Solr outperformed Postgres and seem to have an advantage as the data grow in size. Notice that, Postgres cannot respond after the limit of 2,5m records while performing in the Crossref-total dataset (i.e., the large one).

4.3.3 Conjunctive queries

In the next experiment, we evaluate the time taken to query the databases using conjunctions and conjunctive queries. For this set of queries a logical conjunction operator (*AND*) is used to form conjunctions between pairs of words (or collocations, see Chapter 4 - Subsection 4.2.2. *Full-text search queries*) using the groups we have previously formed.

Figures 4.24, 4.25 and 4.26, show the results for the database systems in regard to the database size for each one of the datasets, (i.e., small, medium and large). Both Elasticsearch and Solr execute the conjunctive queries in less than 10 seconds. Once again, Mongo performs much slower and reaches the limit of 1000 seconds in the medium and large datasets. Postgres is this time also the fastest one among

all the systems. This is explained by the fact that it makes use of the GIN index and seems to take an advantage of it over the rest of the databases. Like we discovered previously in the wildcards search operation, Postgres also perfoms well with conjunctions compared to exact matching, where query performance is affected by bounded phrases. We can conclude that some relational databases can operate much better in certain query operations and for specific amount of data.

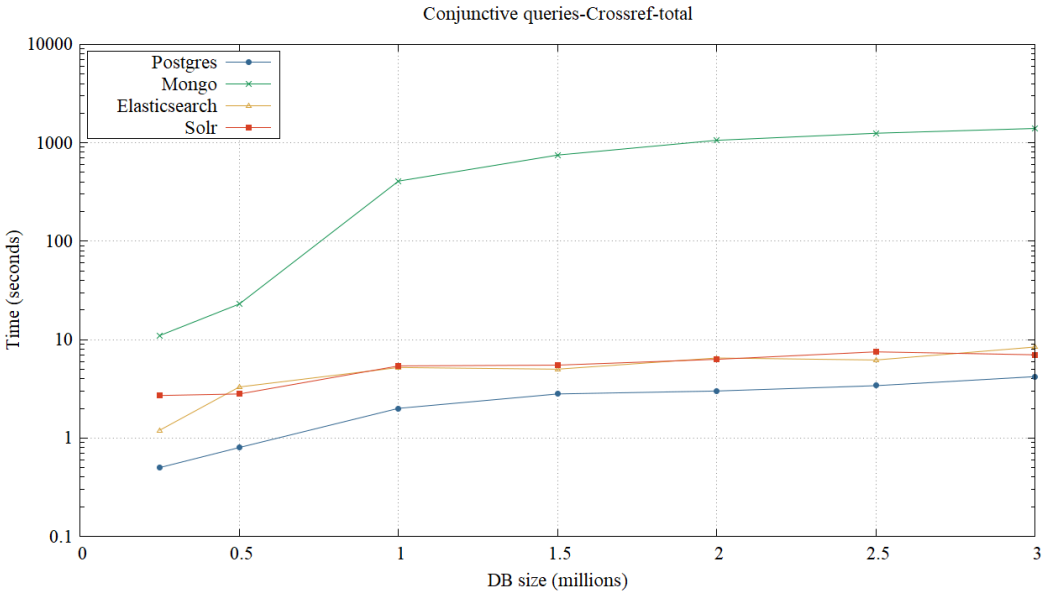

Figure 4.24: Conjunctive queries Crossref total.

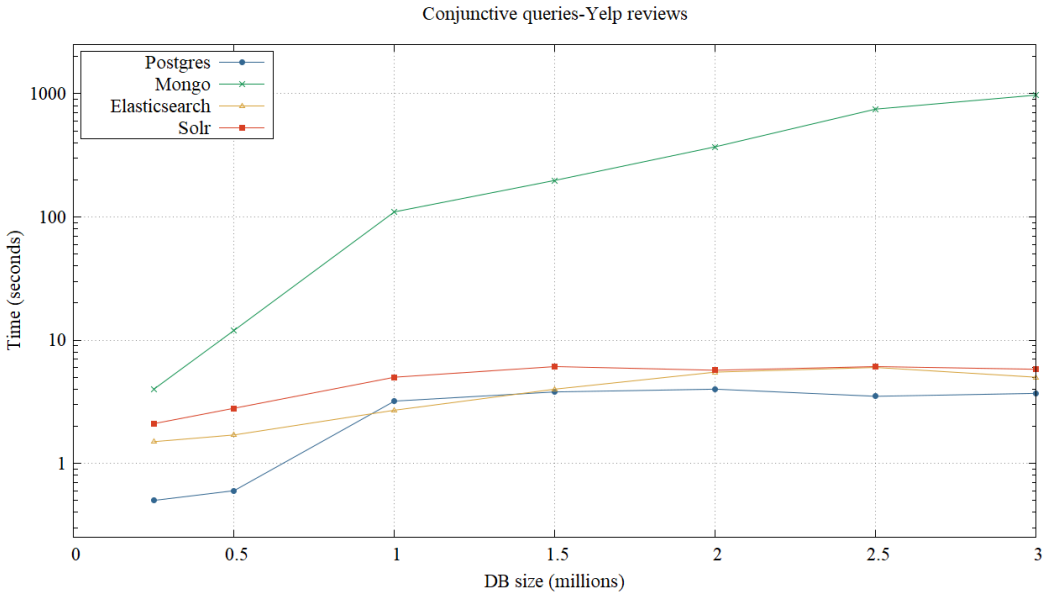

Figure 4.25: Conjunctive queries Yelp reviews.

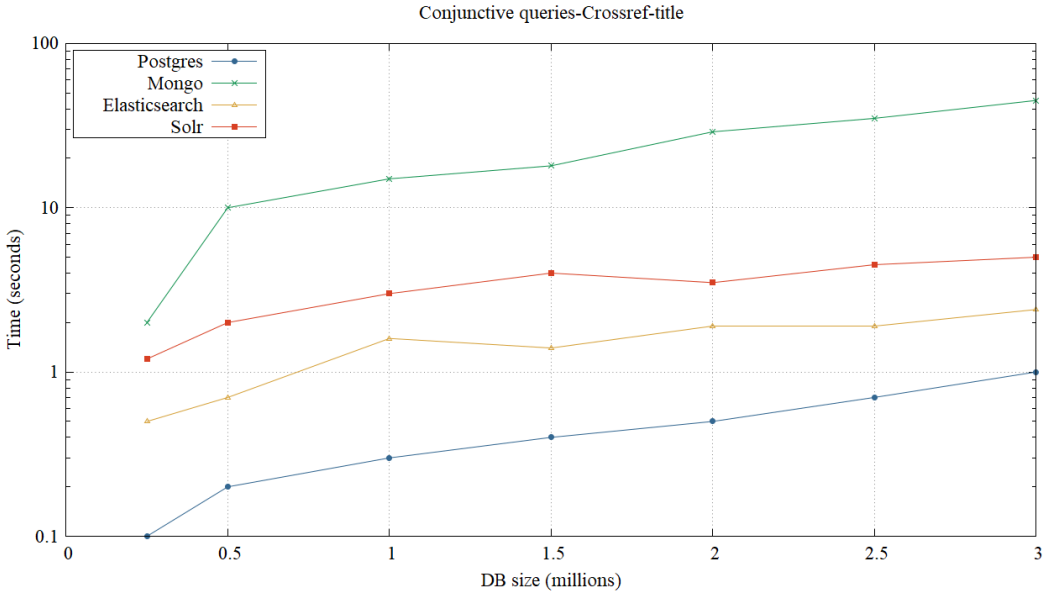

Figure 4.26: Conjunctive queries Crossref title.

Similar to the previous operations, in Figures 4.27, 4.28, 4.29. and 4.30, we can see an alternative view for the results, comparing the time needed to query the datasets (i.e., small, medium and large), in regard to the database size for Postgres, Mongo, Elasticsearch and Solr respectively. Notice that, all systems achieve better time measurements while querying small amount of data (i.e., the small dataset). The time measurements increase accordingly with the number of records in the cases

of Mongo and Elasticsearch. On the other hand, the time measurements overlap in the cases of Solr and Postgres in the medium and large datasets. However, all systems eventually need more time to query 3m documents in the large dataset.

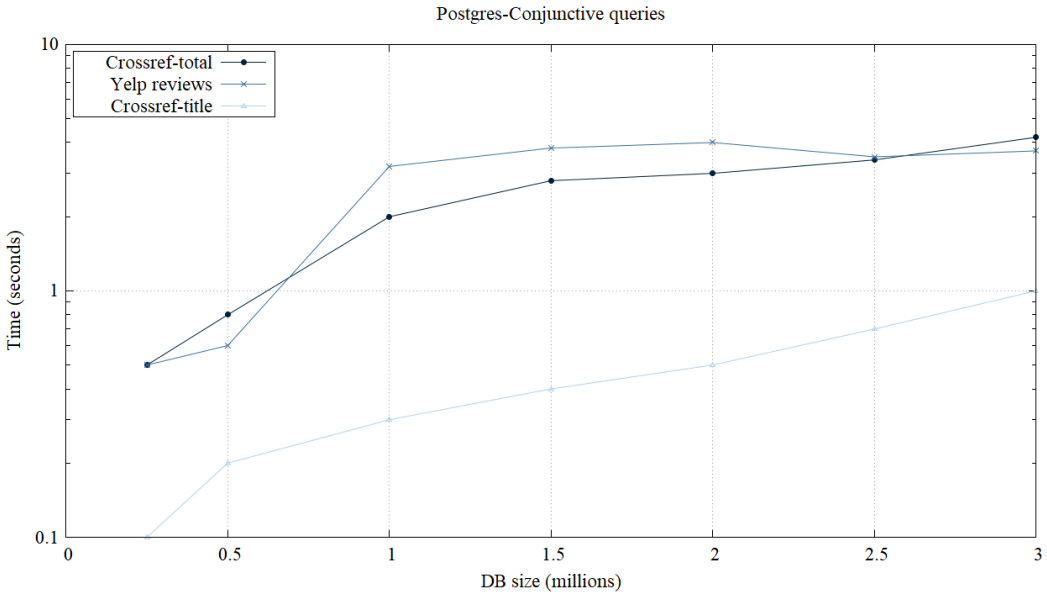

Figure 4.27: Postgres Conjunctive queries.

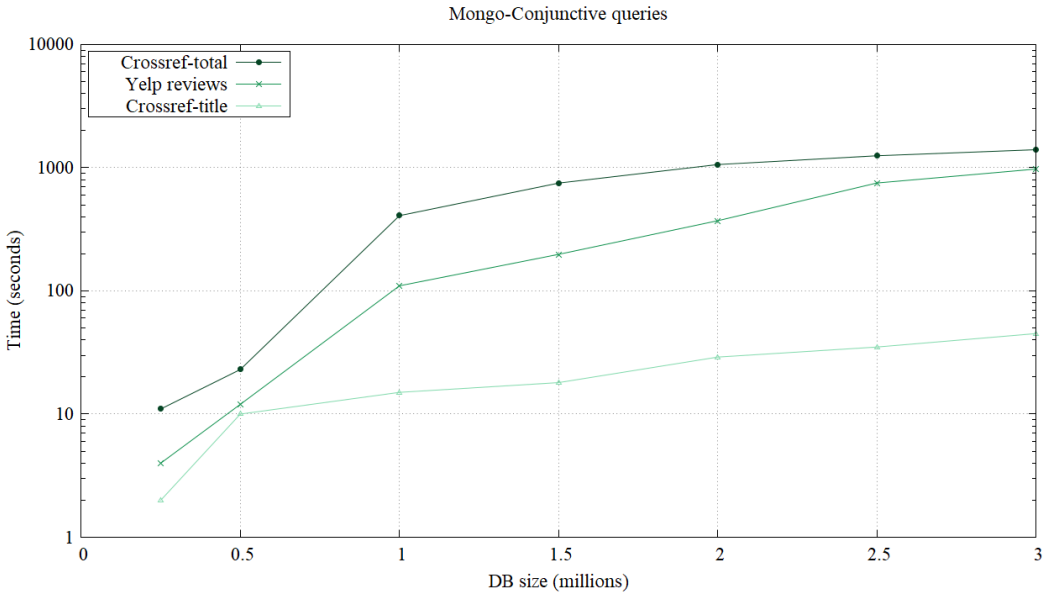

Figure 4.28: Mongo Conjunctive queries.

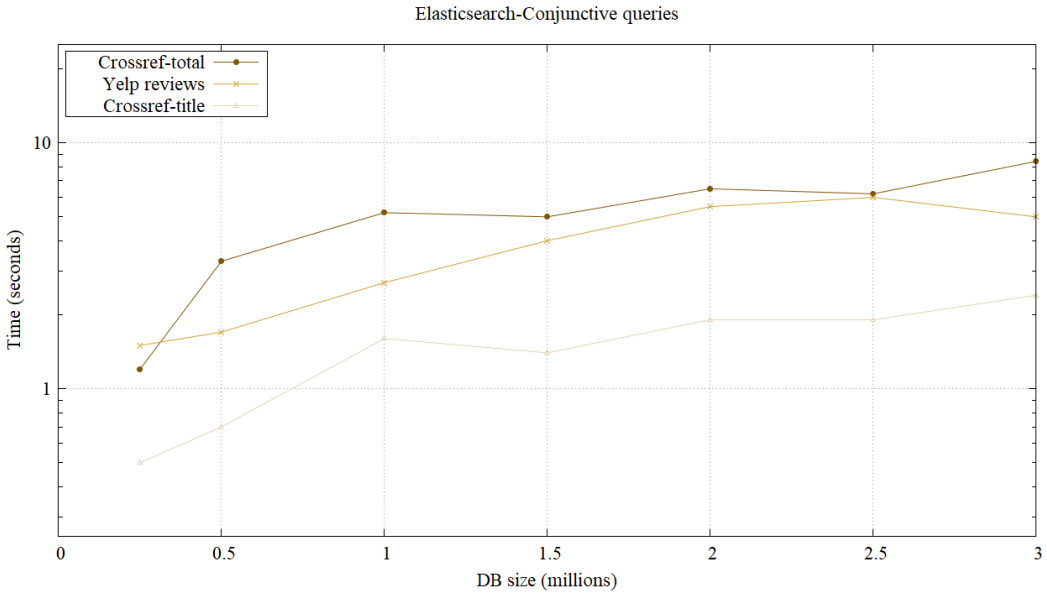

Figure 4.29: Elasticsearch Conjunctive queries.

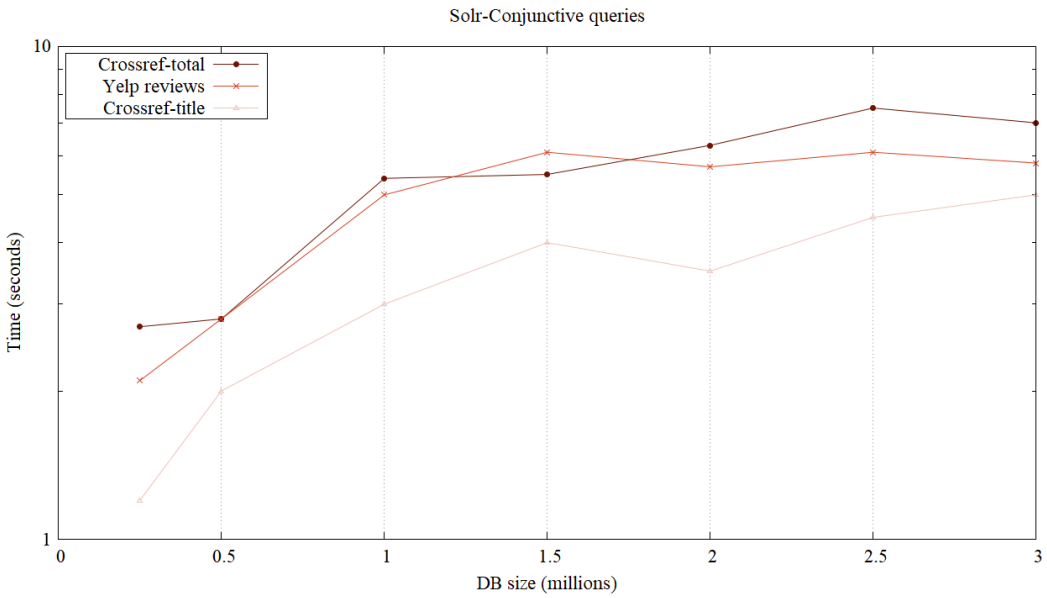

Figure 4.30: Solr Conjunctive queries.

As we observed in the previous query operations, the same conclusion stands also for the conjunctive queries, that is, the runtimes measured for the medium dataset are close to those of the large one, due to the words occurence in the medium dataset.

Like in the previous sections, we have also tested the systems' performance by choosing queries of small selectivity to operate. The results that we obtain for all three datasets are shown in Figures 4.31, 4.32 and 4.33.

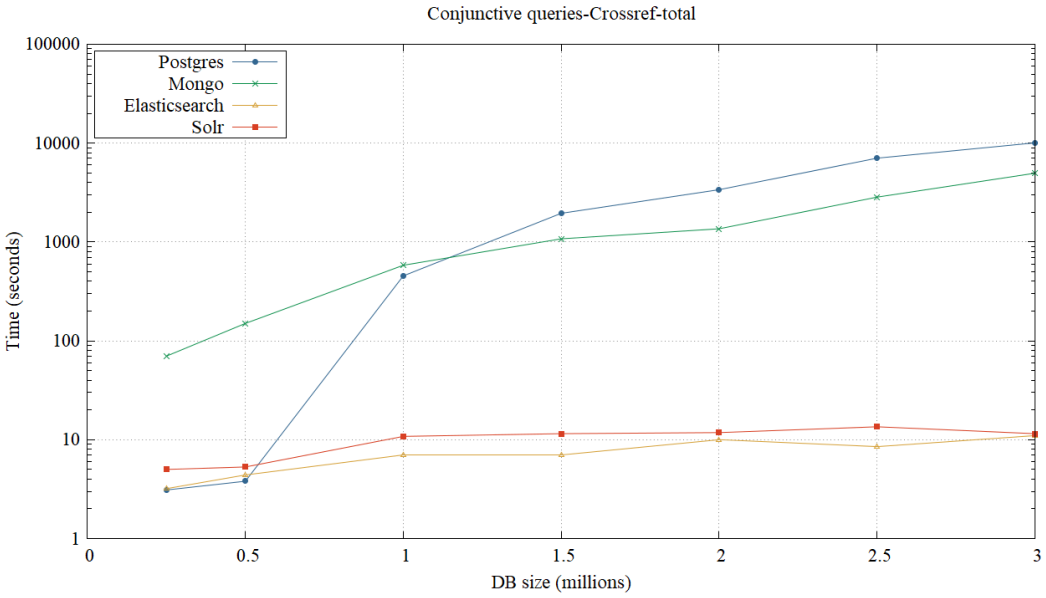

Figure 4.31: Conjunctive queries Crossref total (small selectivity).

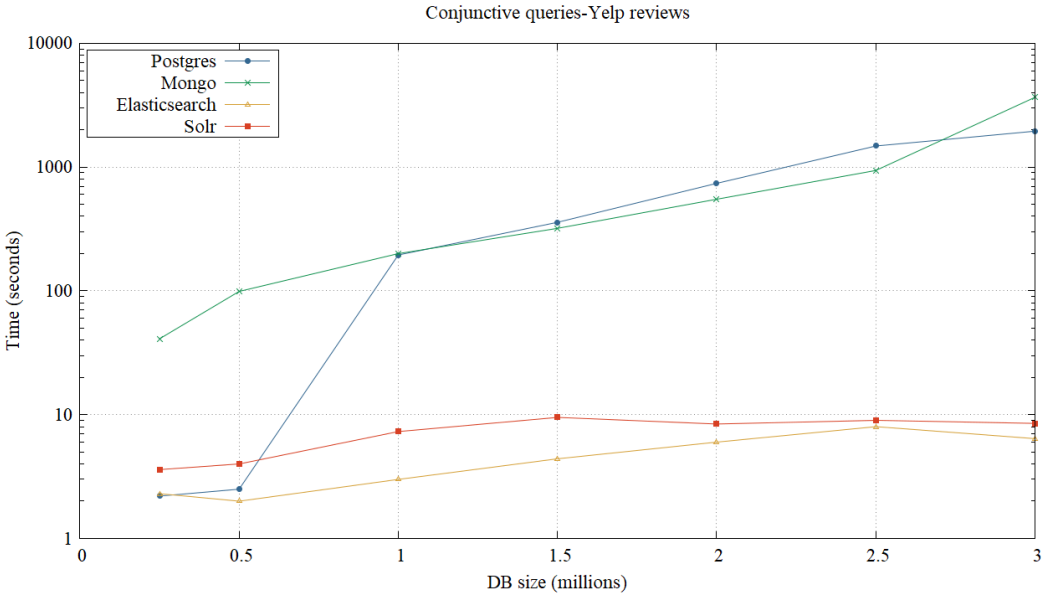

Figure 4.32: Conjunctive queries Yelp reviews (small selectivity).

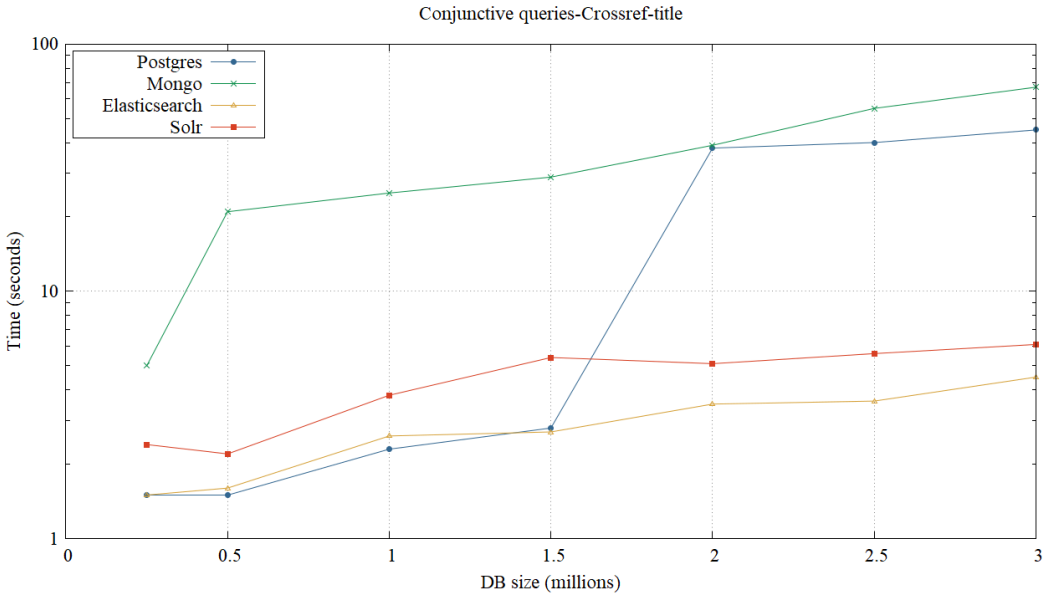

Figure 4.33: Conjunctive queries Crossref title (small selectivity).

Once again we observe that Postgres performs much slower, compared to queries of high selectivity, as the database size increases. Notice also that, after the limit of 500k and up to 1m records, there is an increase of more than two orders of magnitude in time measurements while performing in the medium and large datasets. This same increase happens after the limit of 1.5m and up to 2m records in the small dataset. We believe this is due to the combination of the inefficiency of Postgres with small selective queries and its use of memory, which increases considerably after a limit of records.

4.3.4 Extended conjunctive queries

In this set of experiments we compare the database systems while performing with extended conjunctive queries. This group of queries is composed of conjunctions in which more than two words (i.e., conjunctions with three and four words) are taking part. The words were chosen by using the groups of keywords and collocations we have previously formed (see Chapter 4 - Subsection 4.2.2. *Full-text search queries*). Find below some example queries with words from both datasets.

Crossref

queries with three words

recent AND clinical AND results

using AND patients AND data

new AND different AND species

queries with four words

new AND research AND paper AND show

using AND data AND patient AND cases

recent AND potential AND risk AND effects

Yelp reviews

queries with three words

service AND nice AND place

good AND food AND stuff

great AND place AND stuff

queries with four words

good AND time AND wrong AND place

great AND place AND much AND money

every AND time AND something AND special

In the following figures (Figures 4.34 to 4.45), we can see the diagrams that derived for each one of the results. At this point, it must be noted that conjunctions with three words are indicated as "-Dataset- plus 1" inside the diagram legends, whereas conjunctions with four words are indicated as "-Dataset- plus 2" (e.g. "Crossref-total plus 2"). We compare the extended conjunctive queries with the simple conjunctive queries that we have presented in the previous set of experiments. The results are shown for each database system separately.

At first, we notice that adding one more word in conjunctions does not look to affect the systems' performance in some cases like those of Postgres and Solr, while adding two more words increase the time measurements. This behaviour is observed in the medium and the large datasets whereas in the small dataset the same systems show

a slightly proportional behaviour. Besides that, Mongo's performance does not look to be affected with extended conjunctions, moreover, it responds irrespectively to the augmentation of the words in queries.

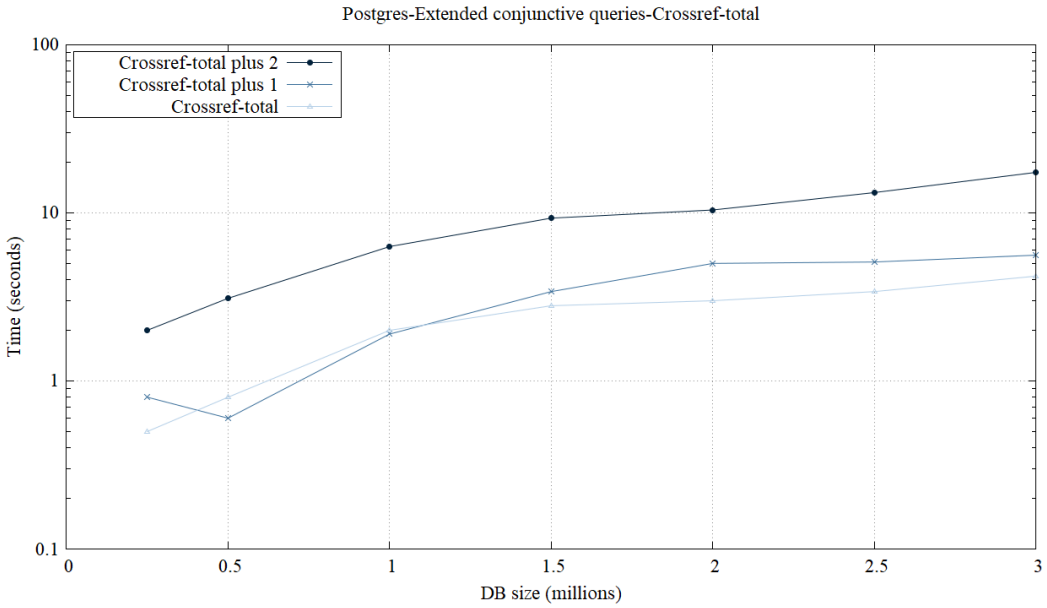

Figure 4.34: Postgres Extended Conjunctive queries Crossref total.

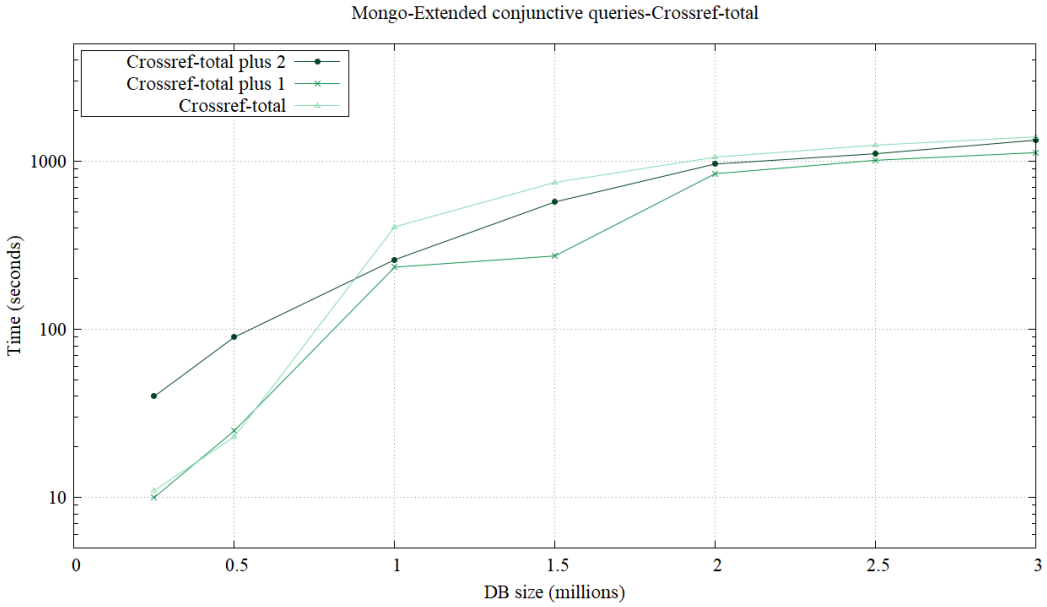

Figure 4.35: Mongo Extended Conjunctive queries Crossref total.

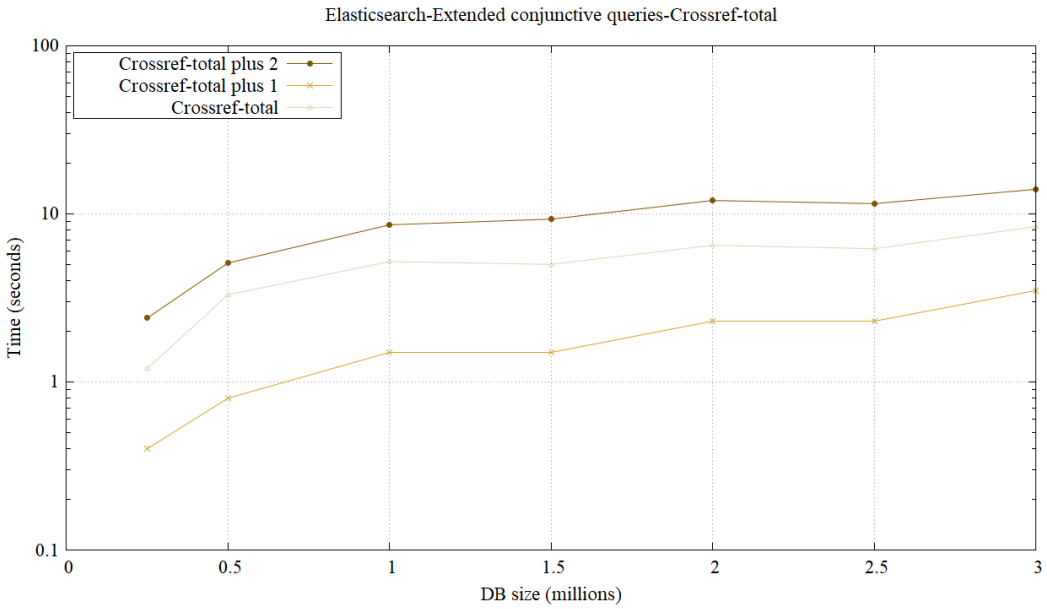

Figure 4.36: Elasticsearch Extended Conjunctive queries Crossref total.

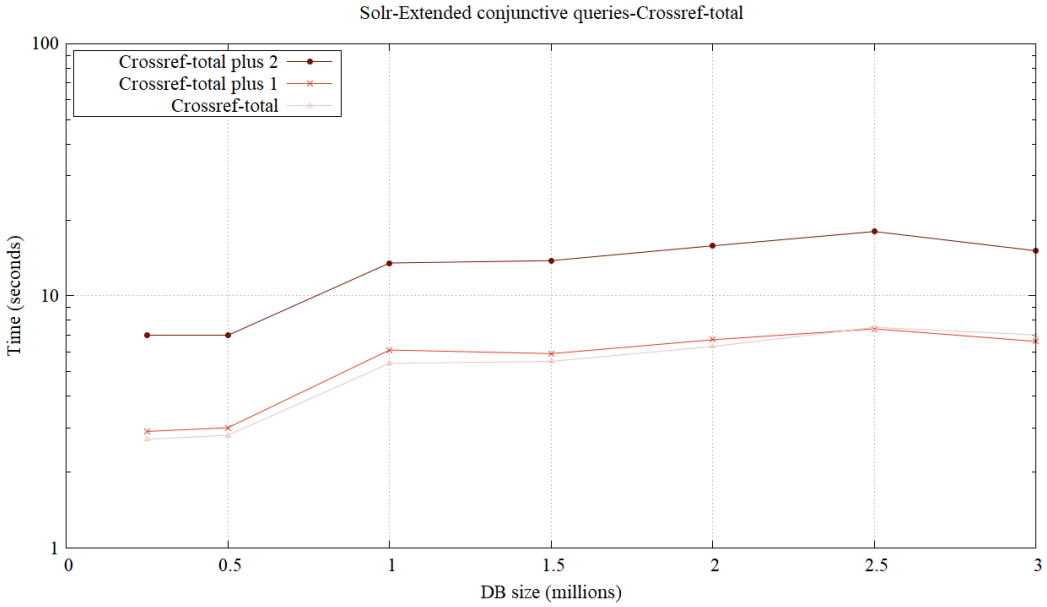

Figure 4.37: Solr Extended Conjunctive queries Crossref total.

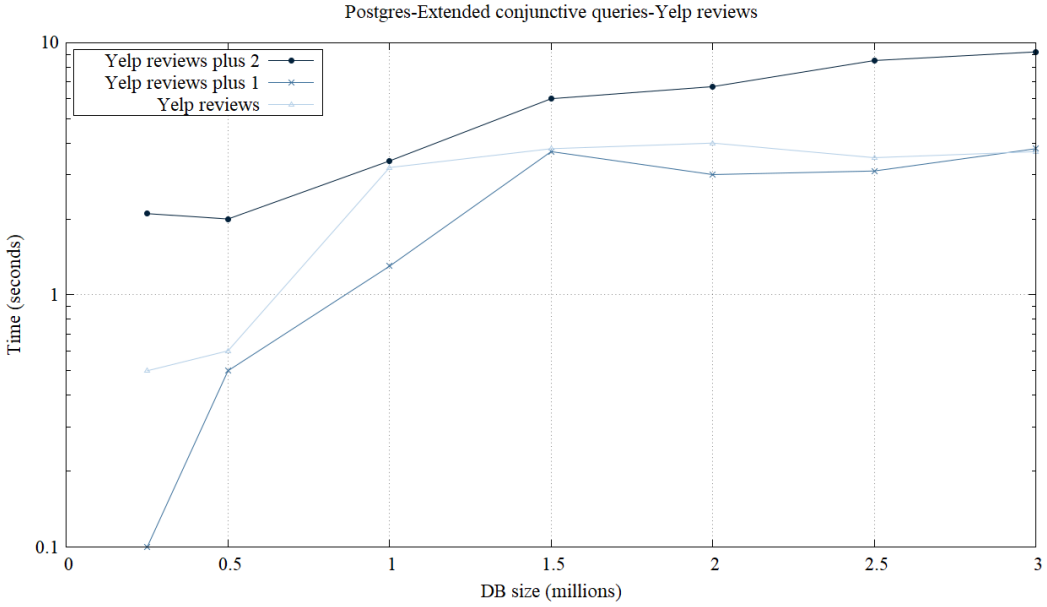

Figure 4.38: Postgres Extended Conjunctive queries Yelp reviews.

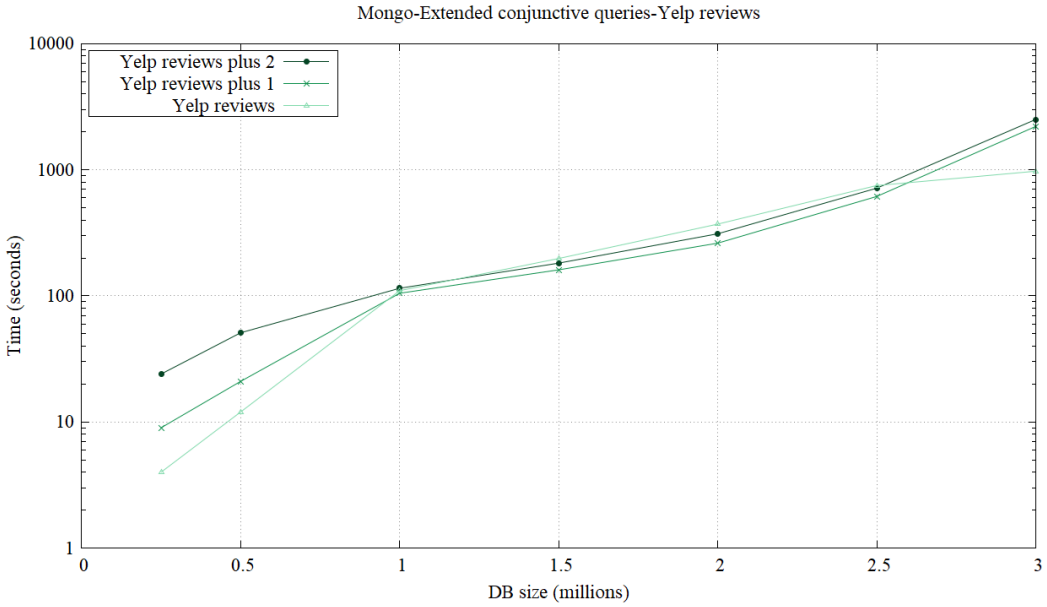

Figure 4.39: Mongo Extended Conjunctive queries Yelp reviews.

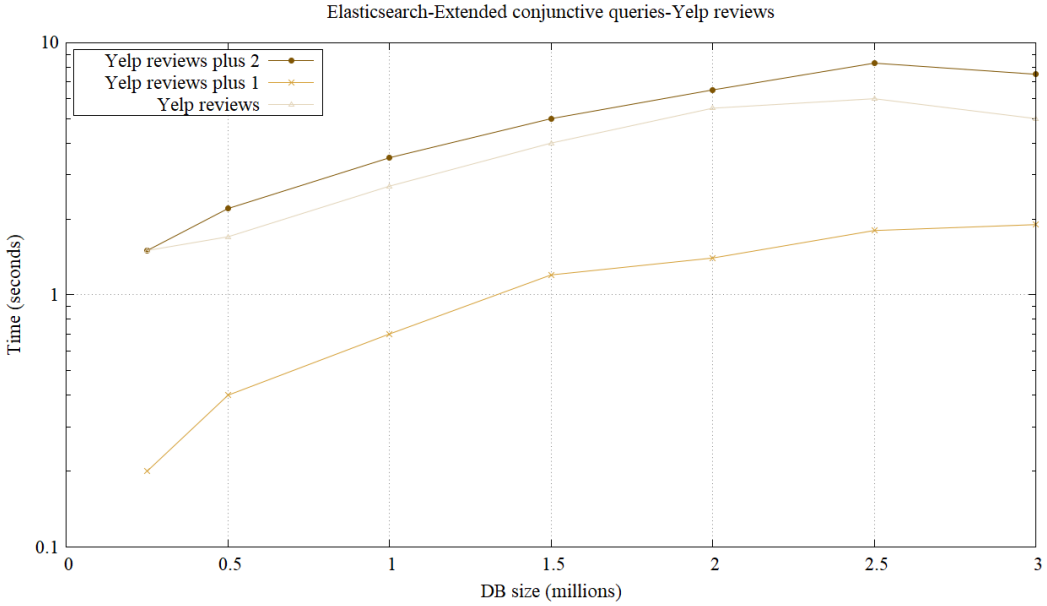

Figure 4.40: Elasticsearch Extended Conjunctive queries Yelp reviews.

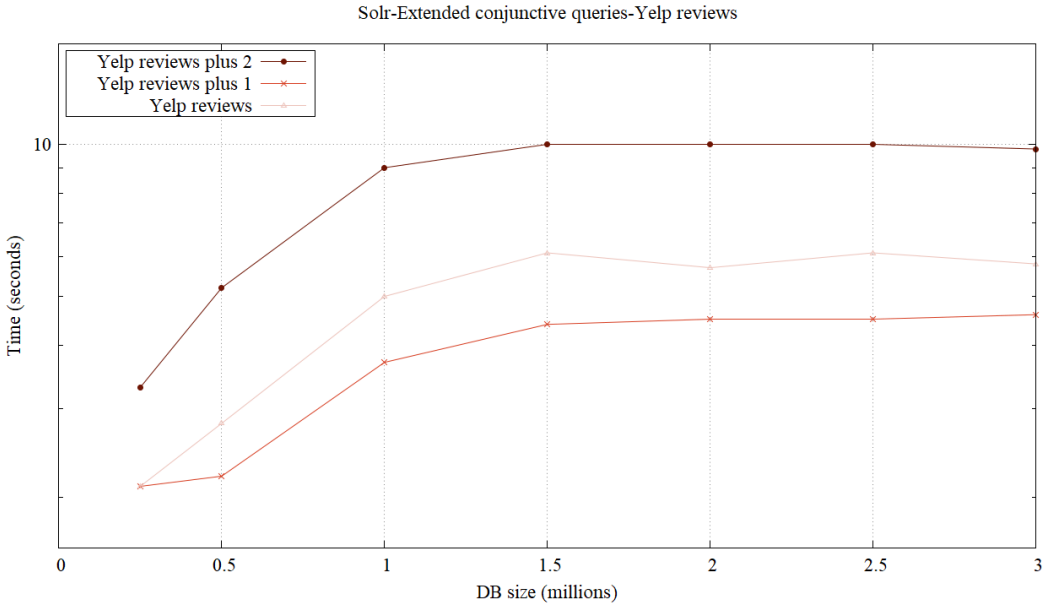

Figure 4.41: Solr Extended Conjunctive queries Yelp reviews.

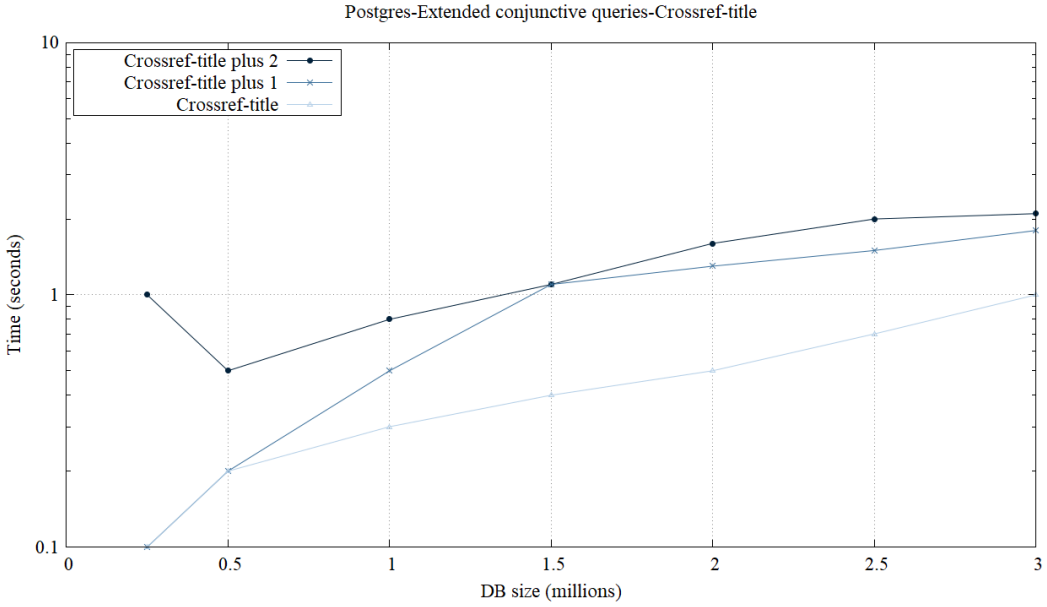

Figure 4.42: Postgres Extended Conjunctive queries Crossref title.

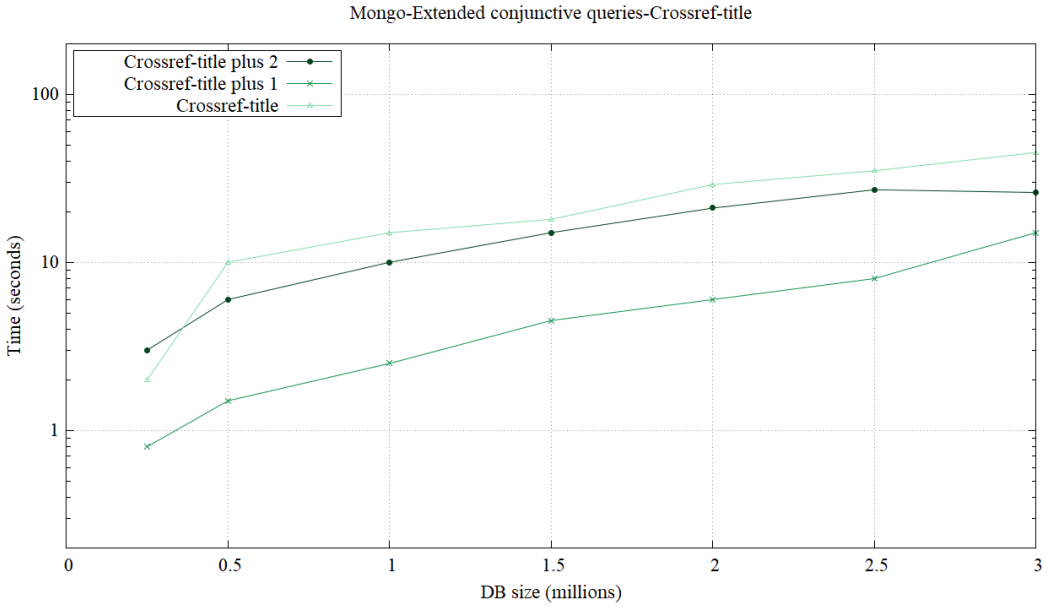

Figure 4.43: Mongo Extended Conjunctive queries Crossref title.

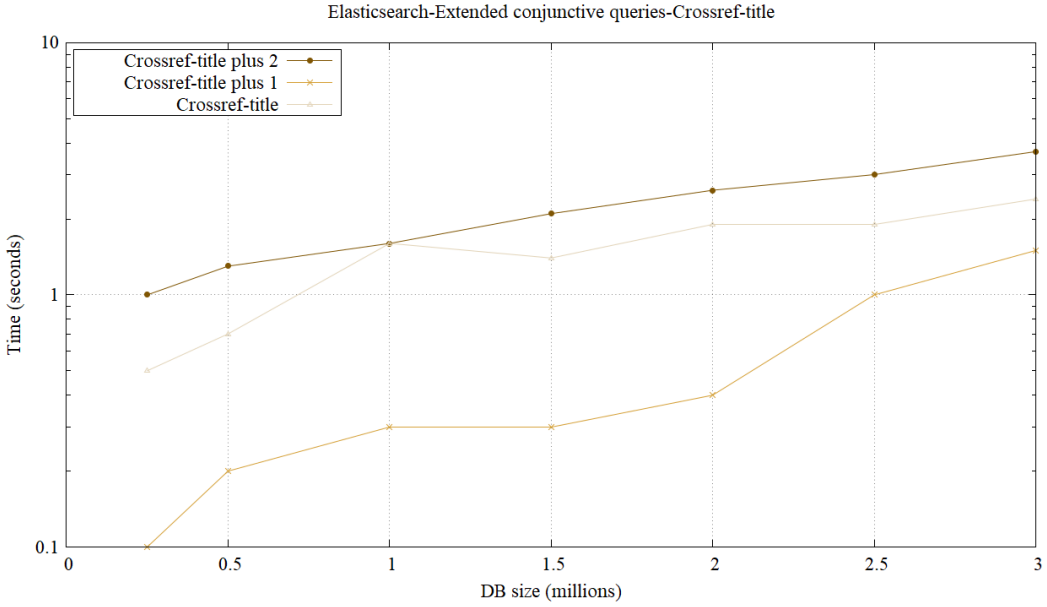

Figure 4.44: Elasticsearch Extended Conjunctive queries Crossref title.

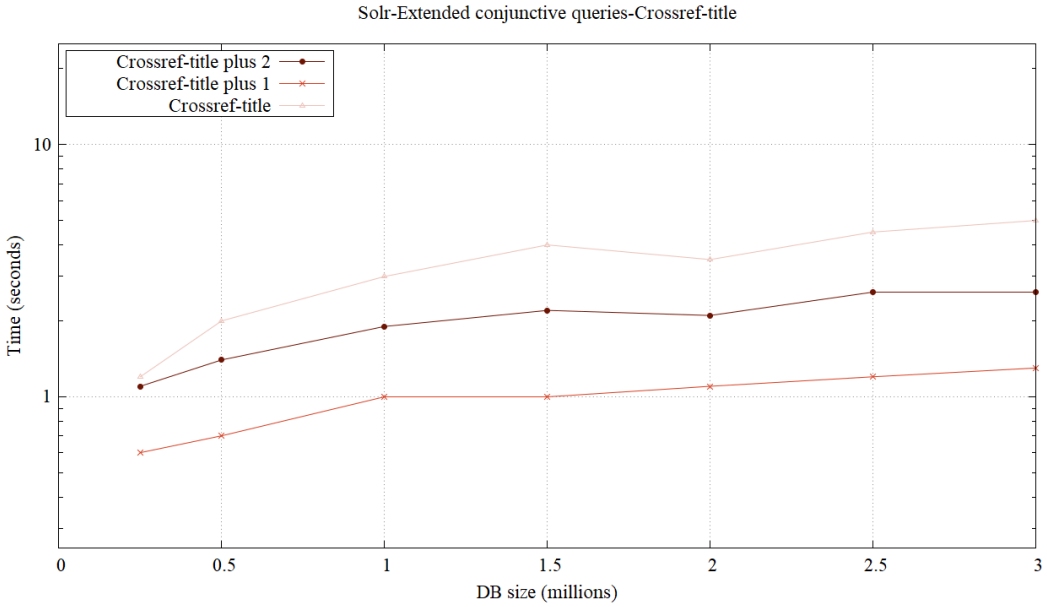

Figure 4.45: Solr Extended Conjunctive queries Crossref title.

Another observation that we can make from this set of experiments, is the results we obtain for Elasticsearch, where despite the fact that time increasements are proportional in all datasets, extented conjunctions with three keywords return better results than the simple conjunctions. A similar behaviour can be seen in cases of Mongo and Solr in the small dataset.
Due to this unexpected behaviour of this set, either because of the exploitation of the keywords inside texts which seems to play a role, or because of the selection of the datasets, we could deduce that extended conjunctions give a different prospective in our performance evaluation unlike the results we arrived so far from the previous query operations.

4.3.5 Selectivity

The database selectivity is a measure of how selective a query is; a lower selectivity value results in a "selective" query (or a query of high selectivity), which selects fewer rows to scan and filter relatively to the number of rows in the table. On the other hand, a large selectivity value results in a "non selective" query (or a query with small selectivity) which selects much more rows to scan and filter. In subsection 4.2.2. (Chapter 4 - Subsection 4.2.2. *Full-text search queries*) we have set the proportions for each selectivity level (i.e., small, medium and high) that we used in our tests. Furthermore, in the experiments we have carried out in the previous sections to measure querying time for all the query operations, we higlighted the difference between selective and non-selective queries by applying high and small selectivity in a varying database size and eventually we determined how database selectivity affects the performance of a system.

In this section, we present the evaluation related to selectivity in more detail. Figures 4.46 to 4.57, show the results that derived after comparing queries in terms of small, medium and high selectivity in regard to the database size for each one of the datasets, (i.e., small, medium and large). Each diagram depicts the results for Postgres, Mongo, Elasticsearch and Solr and the databases were tested performing with exact phrase matching queries.

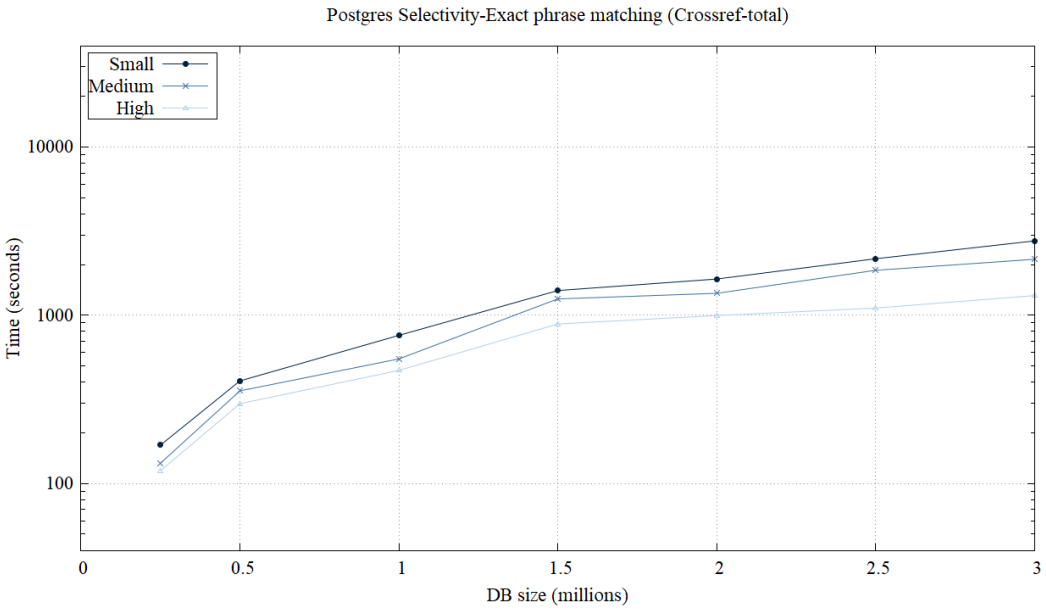

Figure 4.46: Postgres Selectivity Exact phrase matching Crossref total.

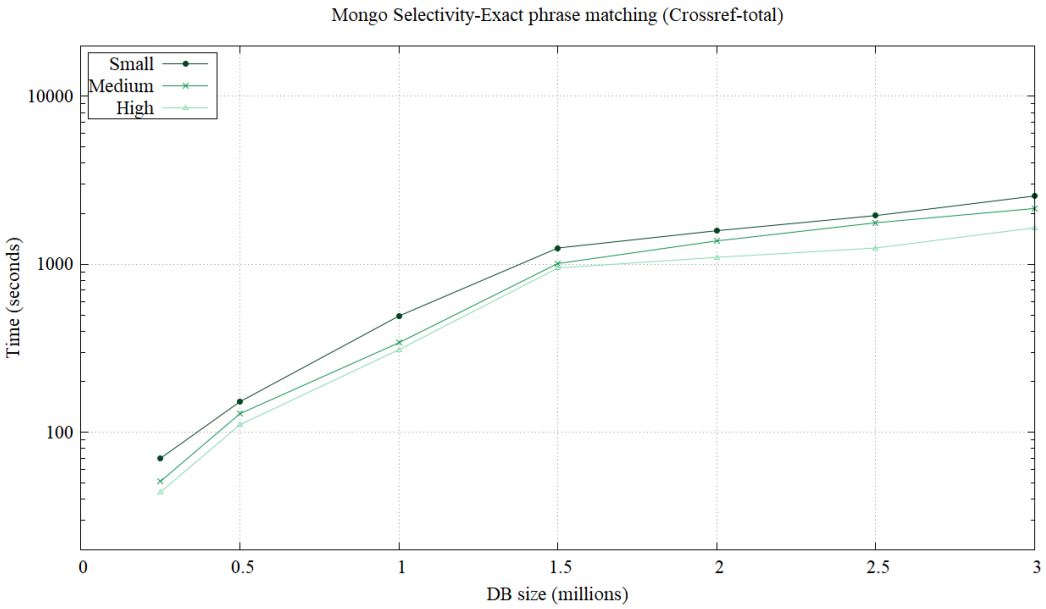

Figure 4.47: Mongo Selectivity Exact phrase matching Crossref total.

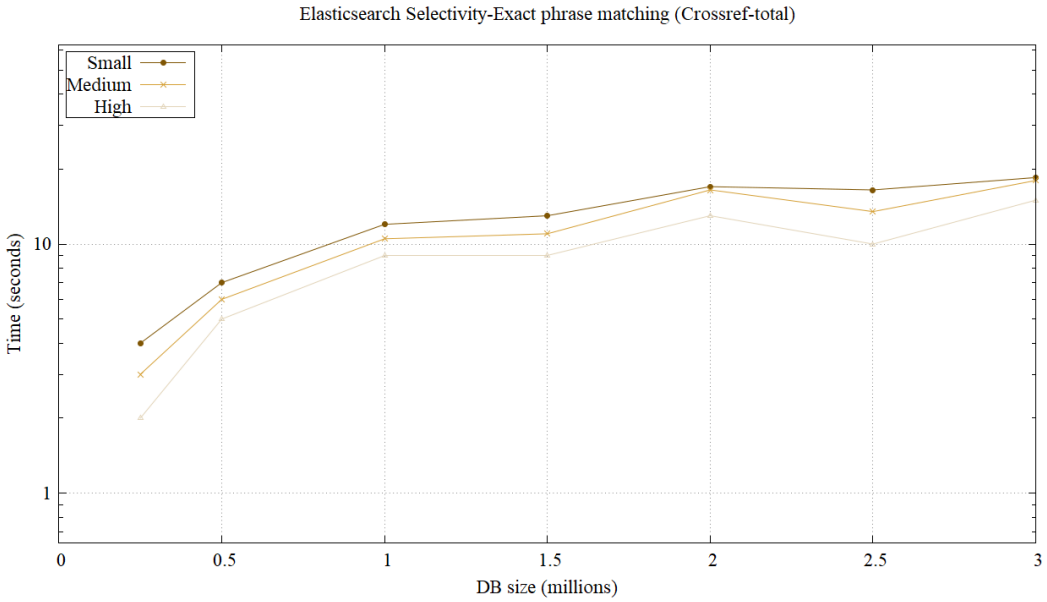

Figure 4.48: Elasticsearch Selectivity Exact phrase matching Crossref total.

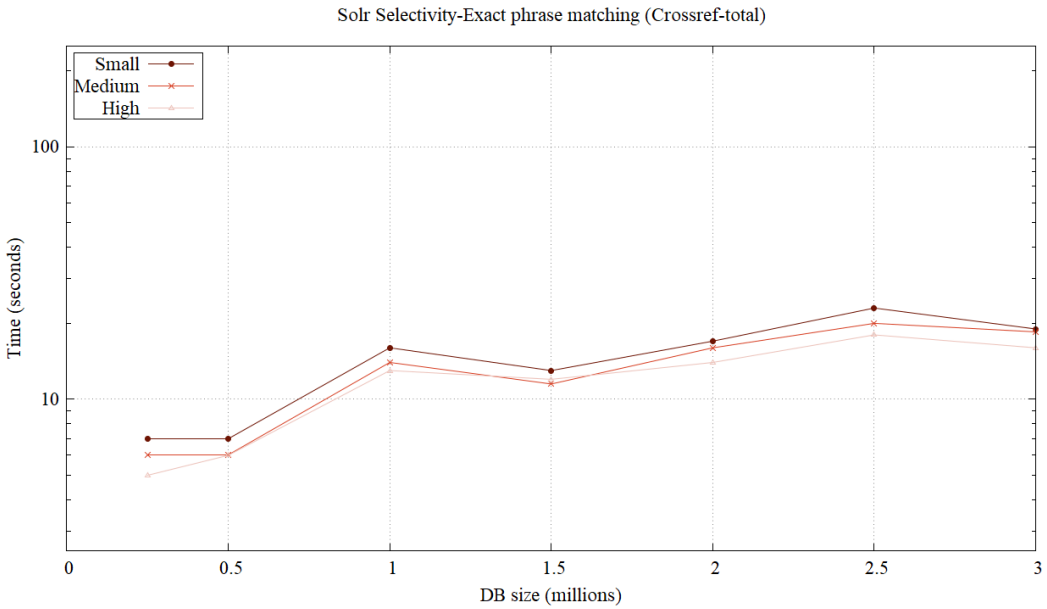

Figure 4.49: Solr Selectivity Exact phrase matching Crossref total.

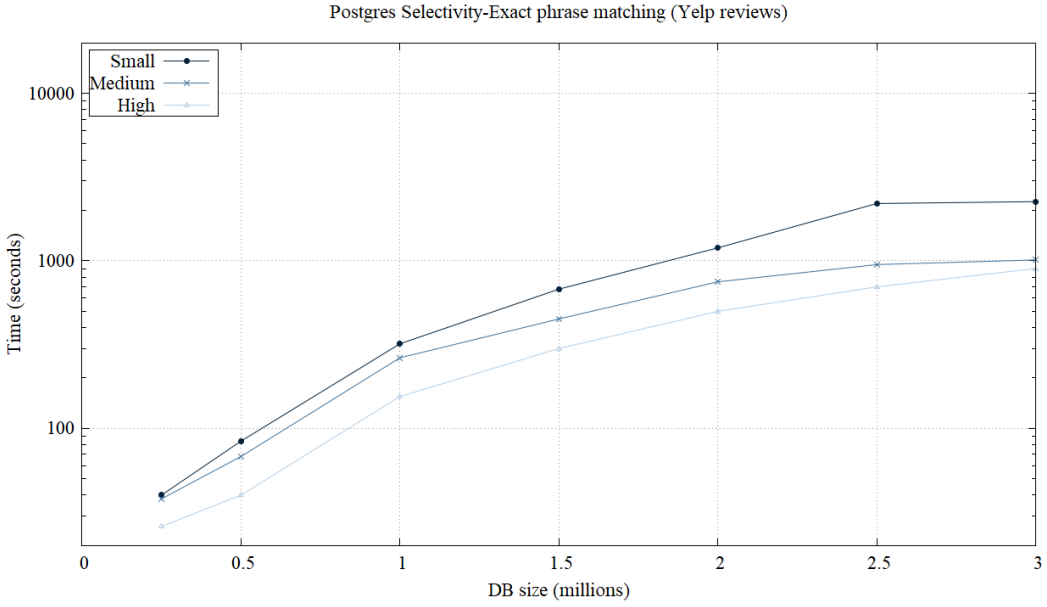

Figure 4.50: Postgres Selectivity Exact phrase matching Yelp reviews.

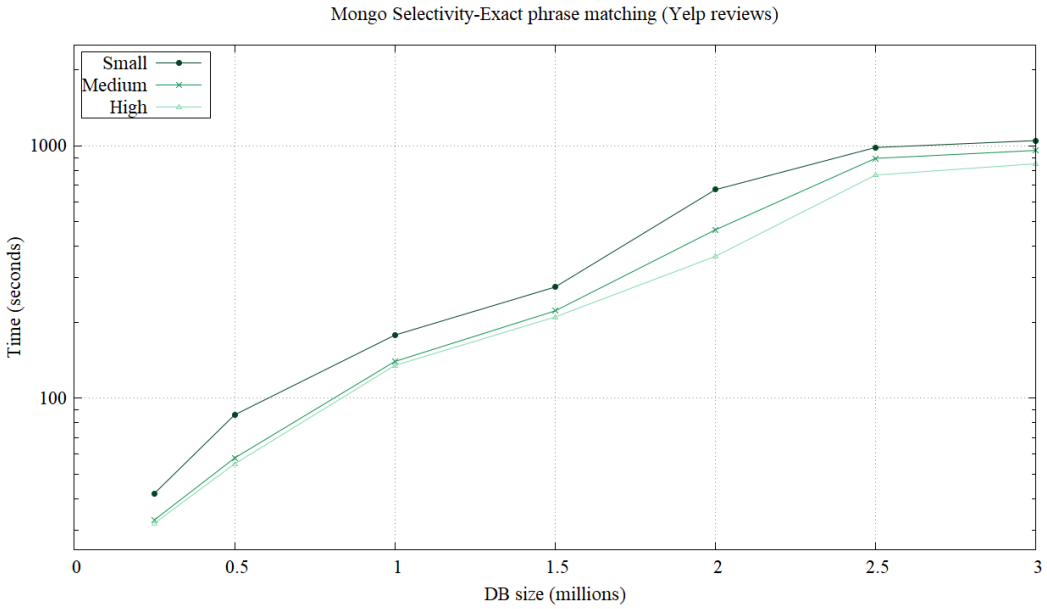

Figure 4.51: Mongo Selectivity Exact phrase matching Yelp reviews.

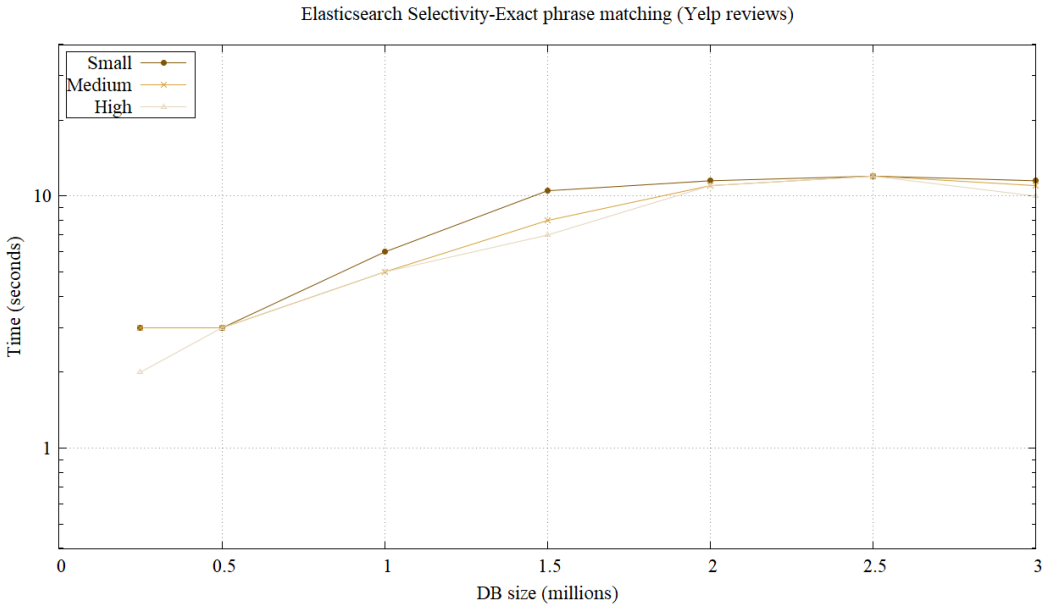

Figure 4.52: Elasticsearch Selectivity Exact phrase matching Yelp reviews.

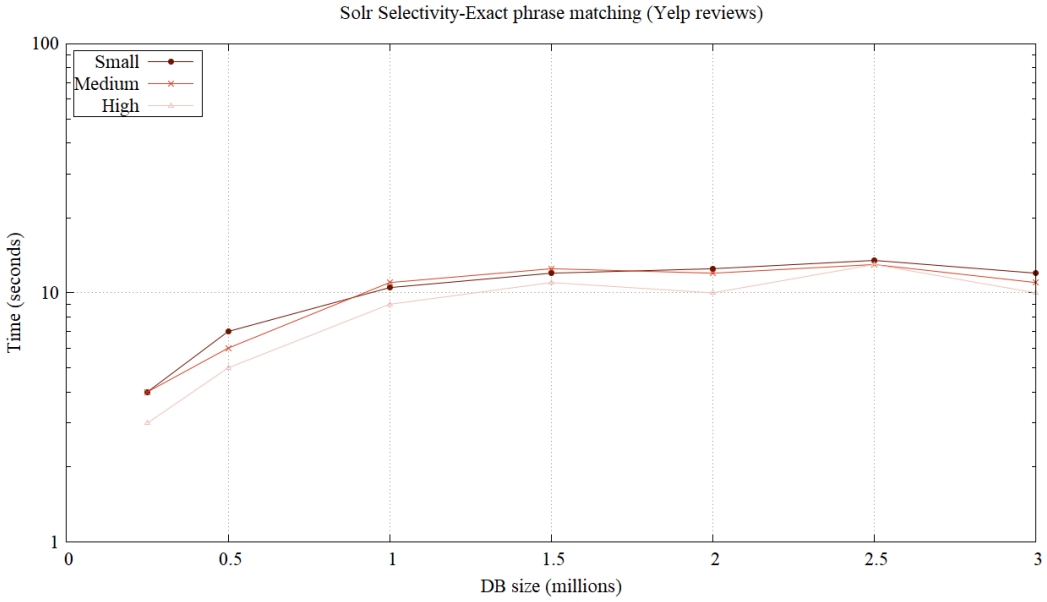

Figure 4.53: Solr Selectivity Exact phrase matching Yelp reviewss.

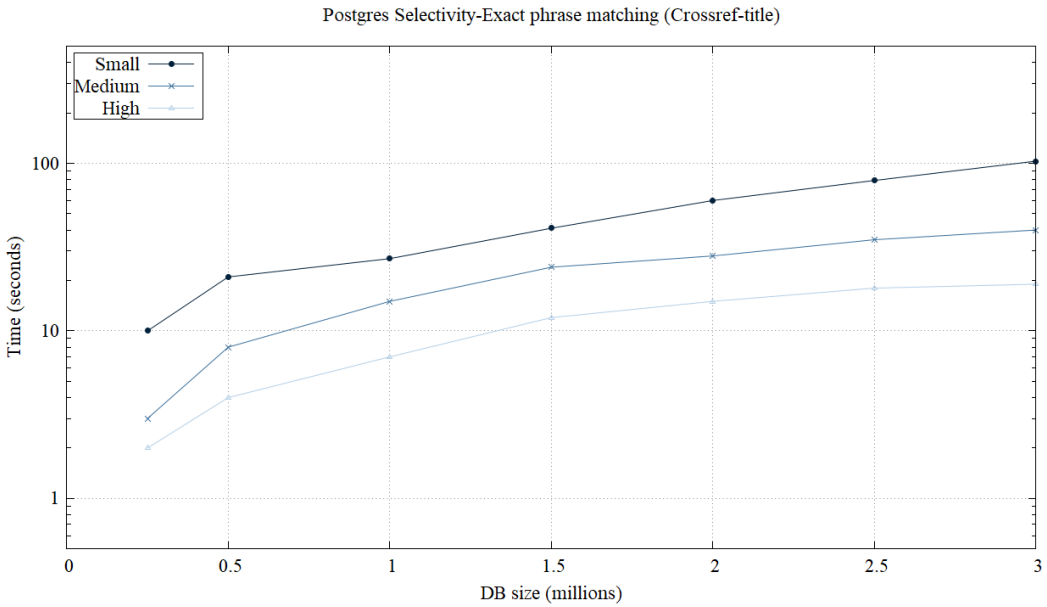

Figure 4.54: Postgres Selectivity Exact phrase matching Crossref title.

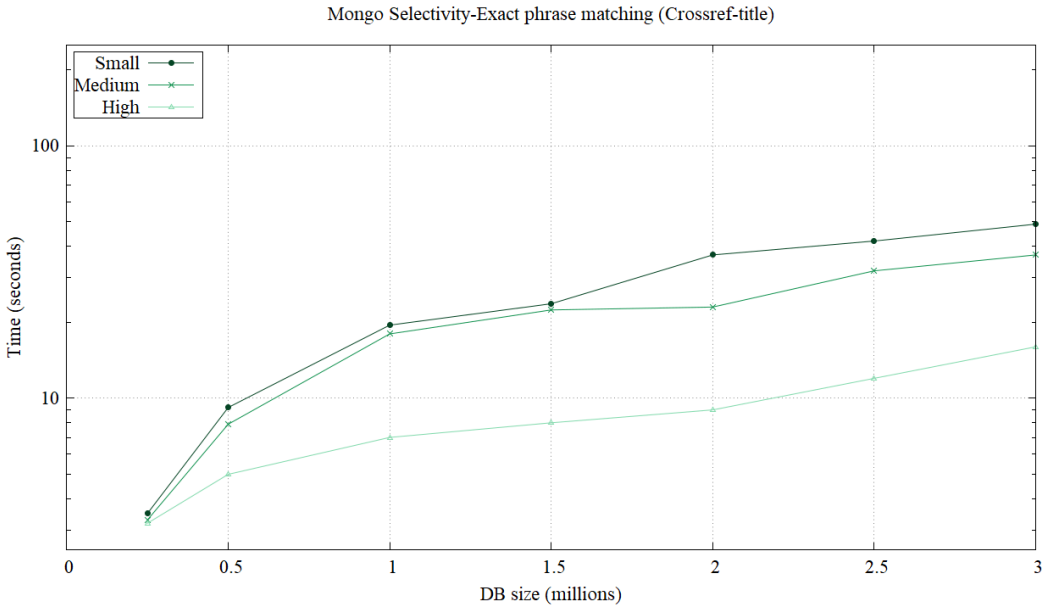

Figure 4.55: Mongo Selectivity Exact phrase matching Crossref title.

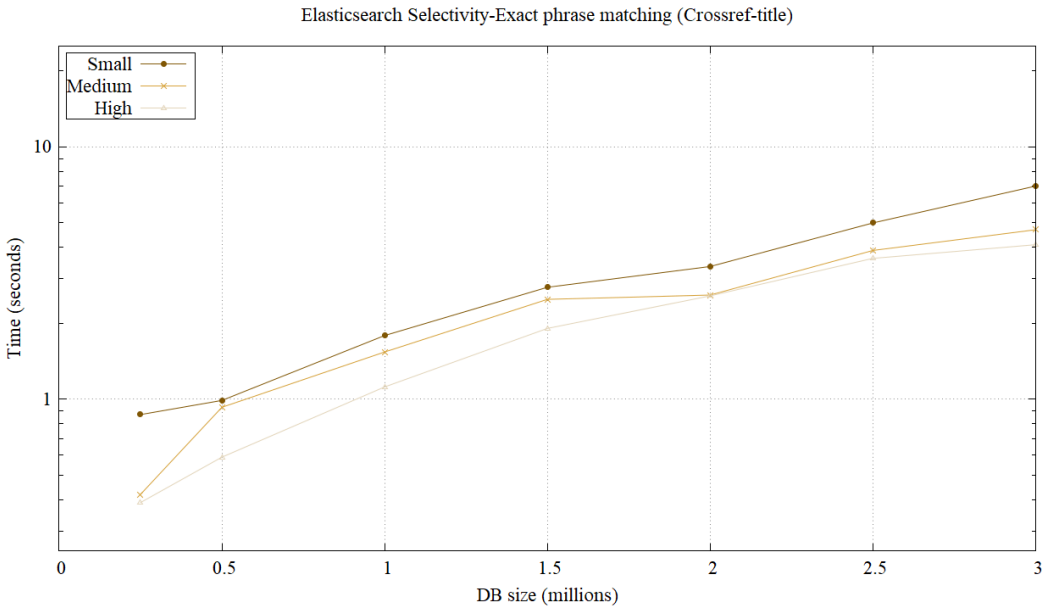

Figure 4.56: Elasticsearch Selectivity Exact phrase matching Crossref title.

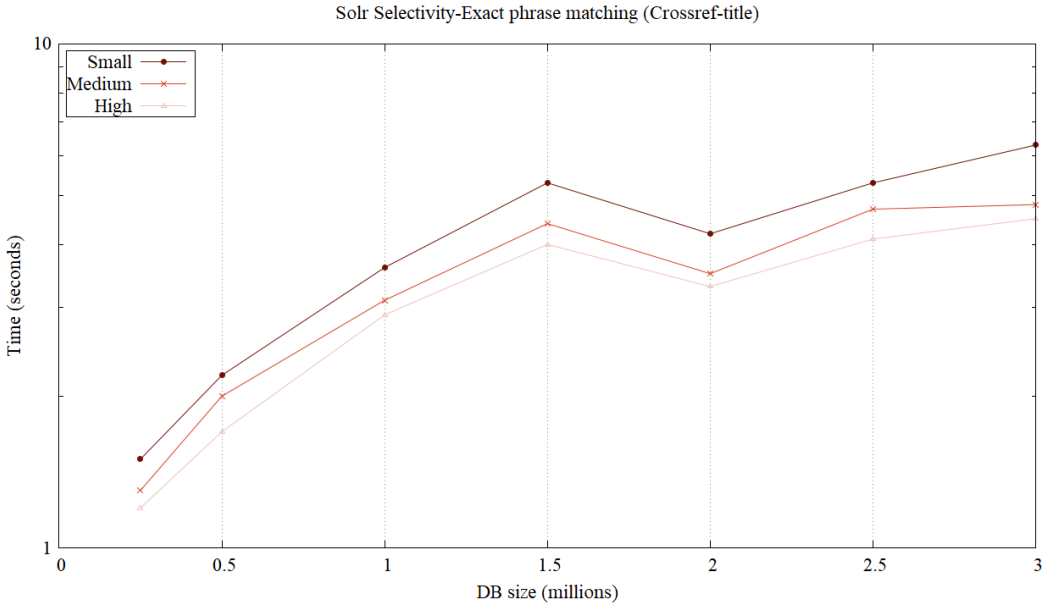

Figure 4.57: Solr Selectivity Exact phrase matching Crossref title.

Figures 4.46 to 4.49 illustrate that all systems behave in the same way while performing with the large dataset. The runtimes increase proportionally with the size of the database and there are minor differences between selectivity levels. In Figures 4.50 to 4.53, where we test with the medium dataset, Postgres and Mongo exhibit an increase between selectivity levels which becomes more sizeable in the small dataset (Figures 4.54 to 4.57). On the other hand, Elasticsearch and Solr seem not to be affected importantly to such an extend.

From this set of experiments, we conclude that selectivity does not definitely affect the performance for some systems, during the exact phrase match operation. The measure of selectivity though can be more clear while executing queries in different query operations. As we have already stated, database systems like Postgres behave differently while executing wildcards or conjunctions instead of exact phrase match queries. To showcase this, we evaluate of how selectivity affects the performance of a system, but this time with the use of conjunctive queries. In the following figures (Figures 4.58 to 4.69), we can see these results.

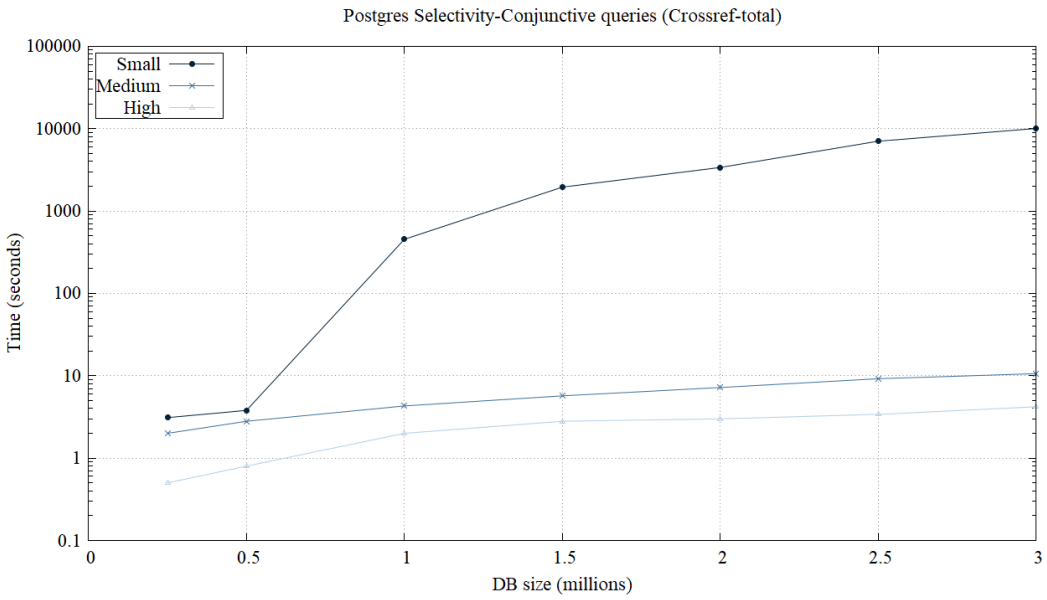

Figure 4.58: Postgres Selectivity Conjunctive queries Crossref total.

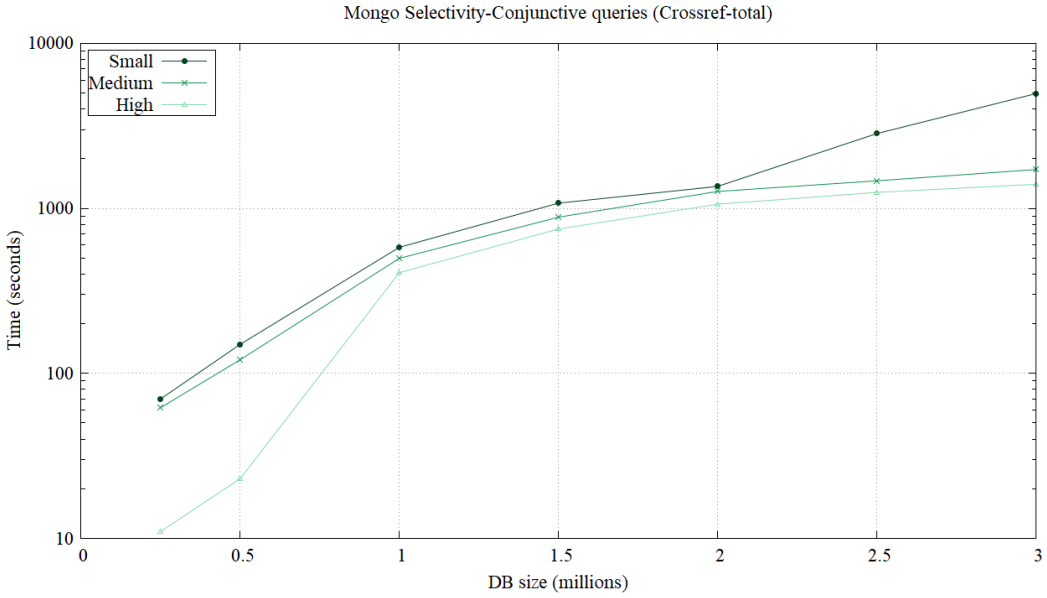

Figure 4.59: Mongo Selectivity Conjunctive queries Crossref total.

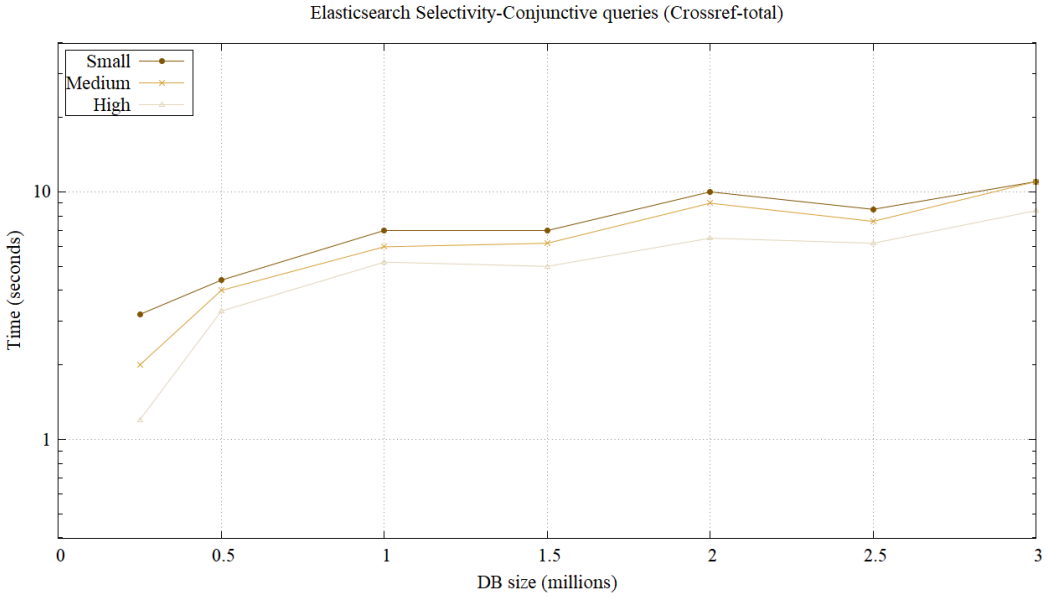

Figure 4.60: Elasticsearch Selectivity Conjunctive queries Crossref total.

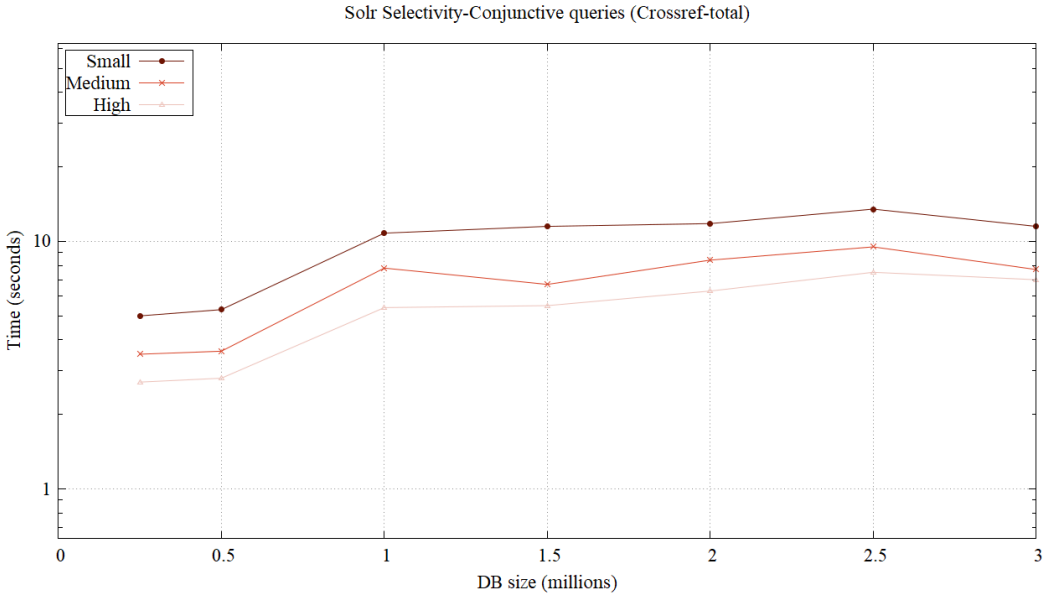

Figure 4.61: Solr Selectivity Conjunctive queries Crossref total.

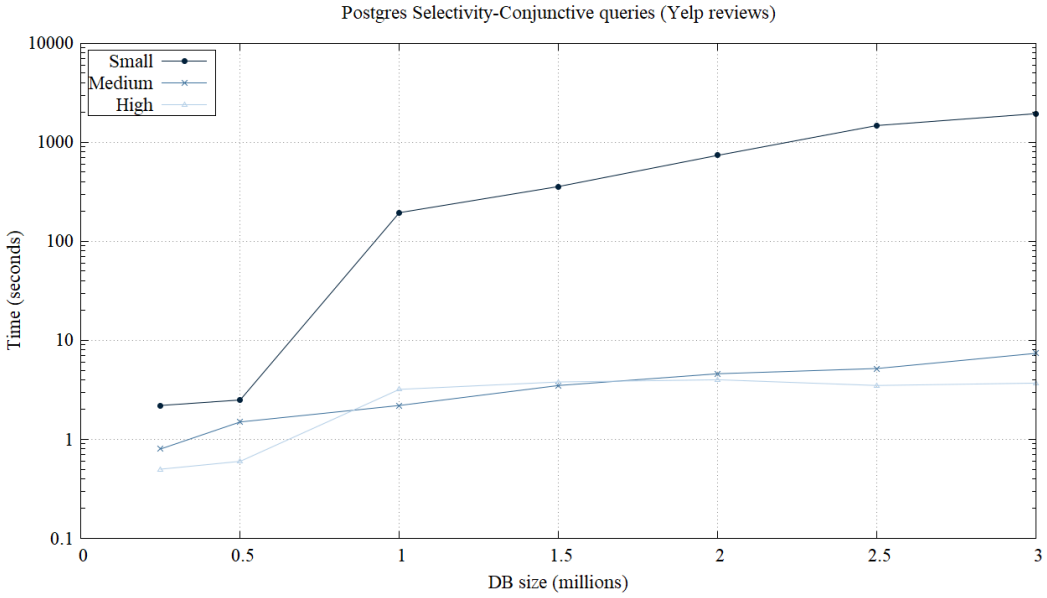

Figure 4.62: Postgres Selectivity Conjunctive queries Yelp reviews.

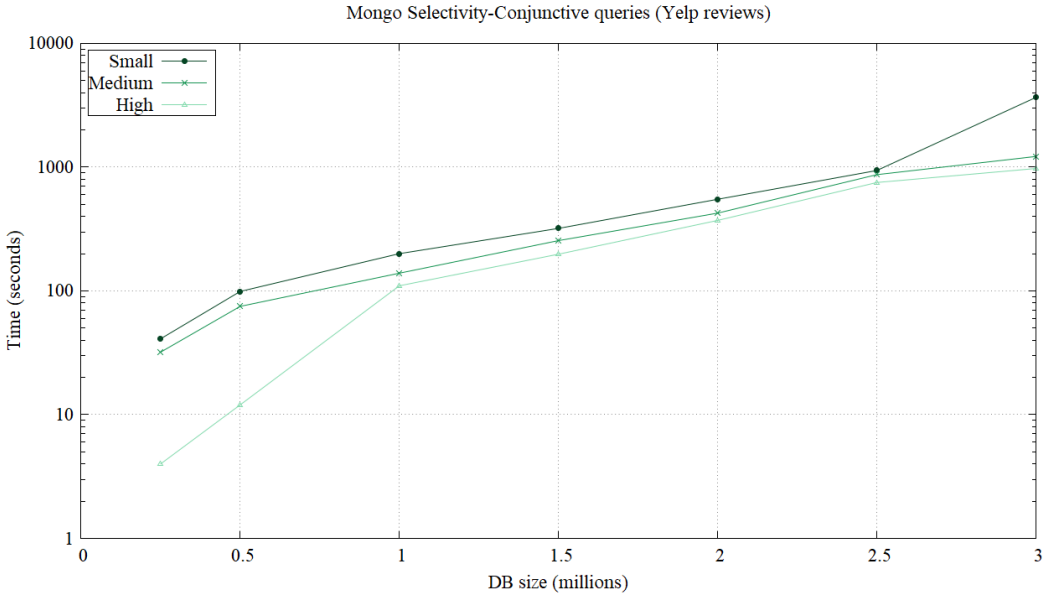

Figure 4.63: Mongo Selectivity Conjunctive queries Yelp reviews.

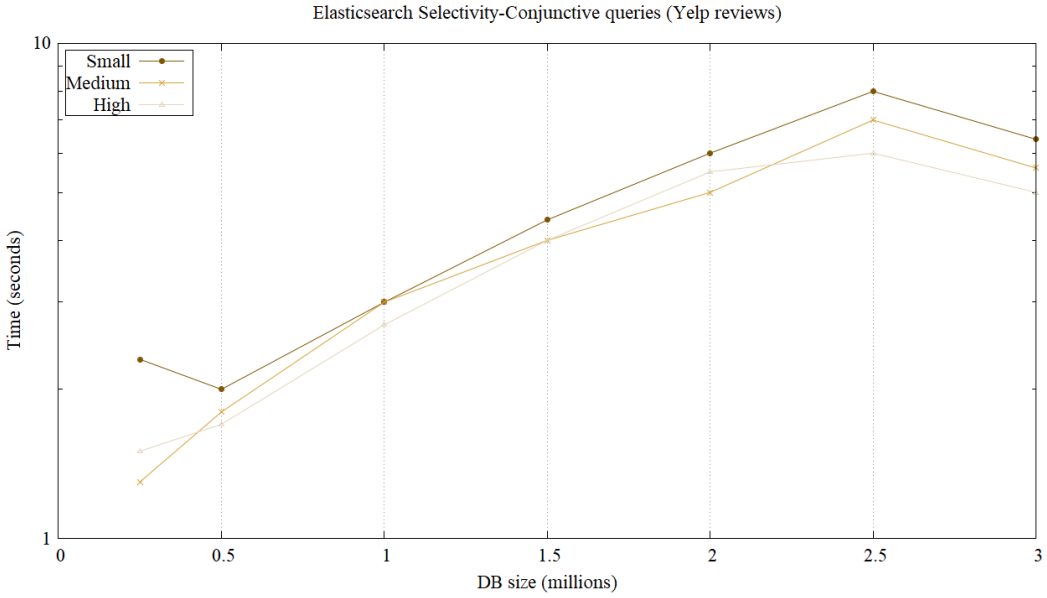

Figure 4.64: Elasticsearch Selectivity Conjunctive queries Yelp reviews.

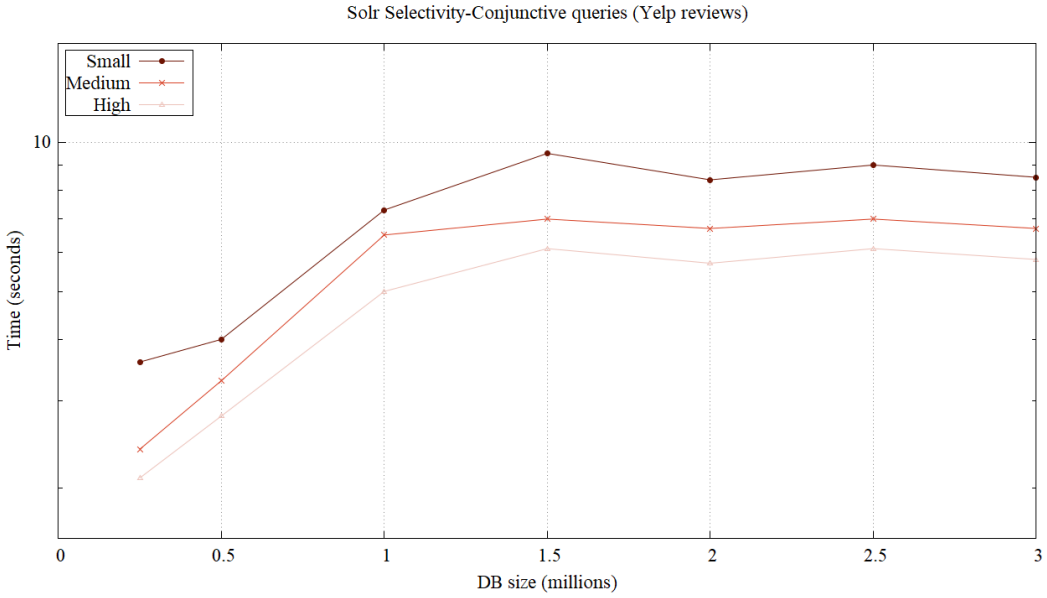

Figure 4.65: Solr Selectivity Conjunctive queries Yelp reviewss.

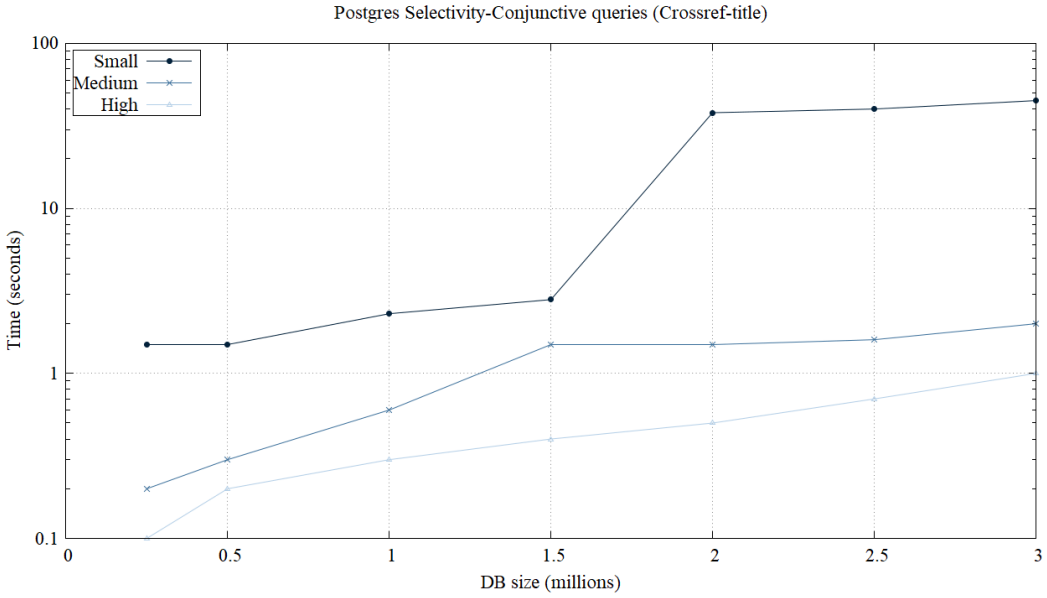

Figure 4.66: Postgres Selectivity Conjunctive queries Crossref title.

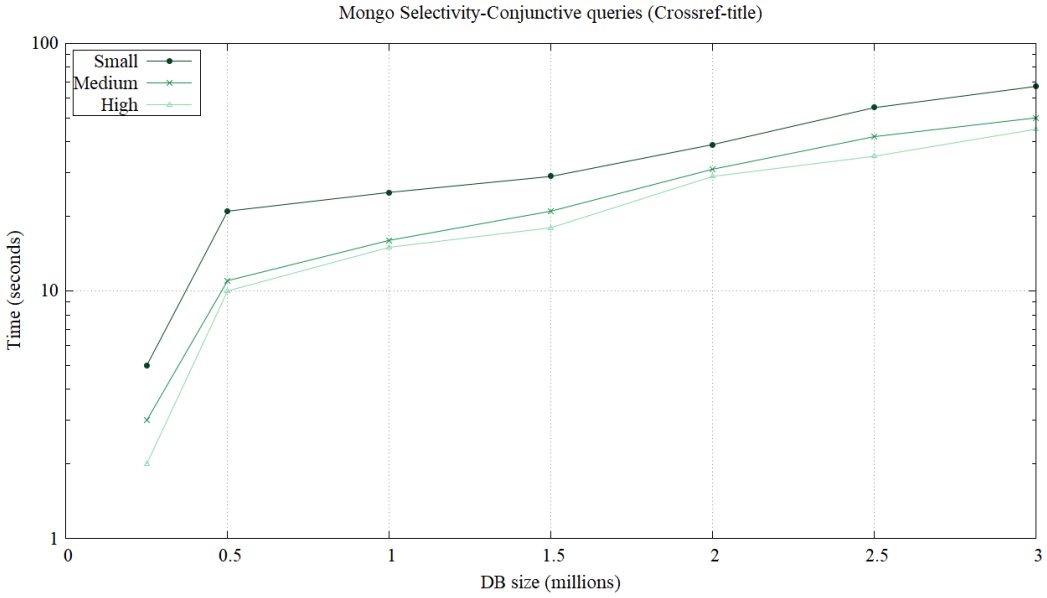

Figure 4.67: Mongo Selectivity Conjunctive queries Crossref title.

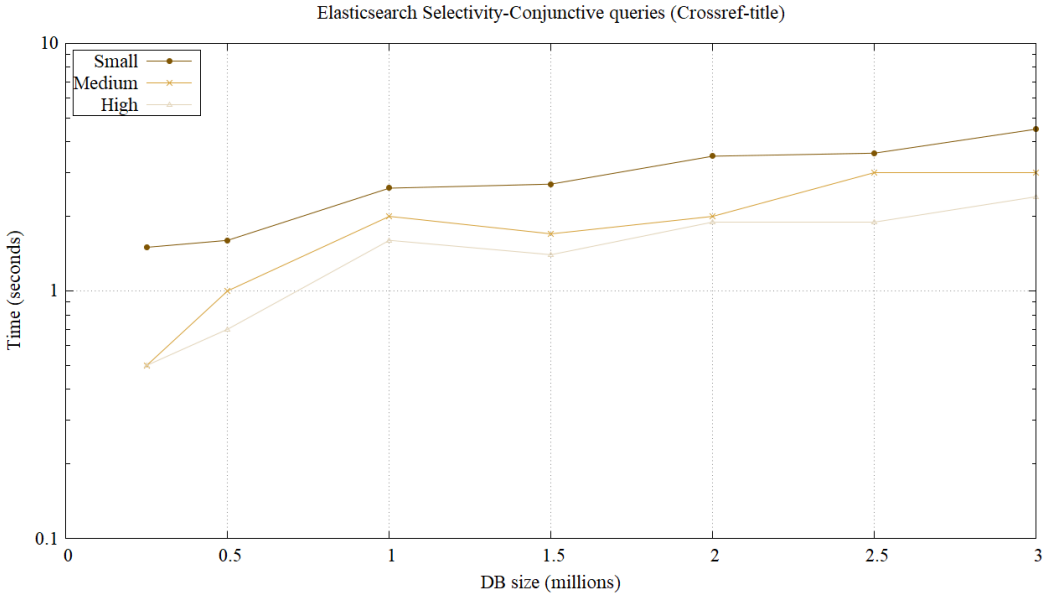

Figure 4.68: Elasticsearch Selectivity Conjunctive queries Crossref title.

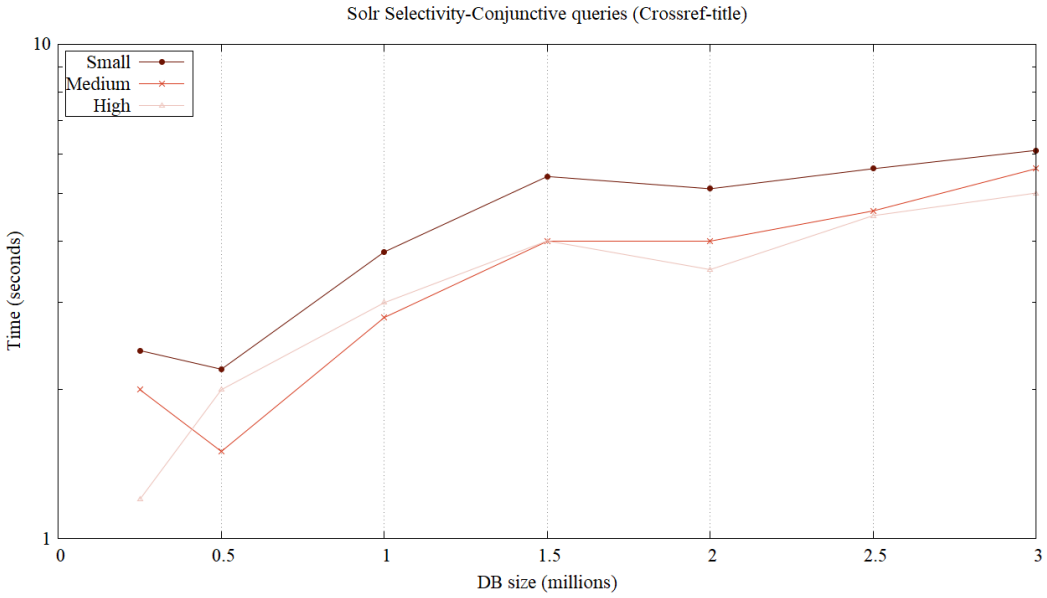

Figure 4.69: Solr Selectivity Conjunctive queries Crossref title.

We observe that Postgres performs significantly slower with queries of small selectivity, indicating a sudden increase in the limits of 1m and 2m records, for the reasons explained in section 4.3.3 regarding the conjunctive queries experiments. Performing in the large dataset, Mongo starts to operate from approximately 10 seconds with high selective queries while with small selective queries it needs something less than 100 seconds. A similar behaviour is observed in the medium dataset. Notice also, that the is a slight difference between selectivity levels in all database sizes for Elasticsearch and Solr, compared to those of exact phrase match operation.

4.3.6 Query operations

For this set of experiments, we investigate the time needed for each one of the systems to query the databases, when using different query operations. We compare each system's perfrormance in querying documents for the full-text search operations of exact phrase matching, wildcards search and conjunctive queries. All tests have been carried out by applying queries of high selectivity.

In Figures 4.70 to 4.81, we can see the results for each database system. We notice that exact phrase matching queries need significantly more time to be executed by the systems in most cases, while on the contrary, wildcards search and

conjunctive queries demand less time. Postgres for example, needs more than 1000 seconds to query 3m records performing in the large dataset, while it needs less than 10 seconds for the rest of the operations. A similar behaviour can be seen in the medium dataset, however these runtimes are reduced significantly in the small dataset. Elasticsearch and Solr also, need more time to operate with exact phrase matching queries but with much better results compared to Postgres (about 10 seconds in all three datasets) and with minor time differences compared to the other query operations. On the other hand, Mongo perfoms with exact phrase matching in the small dataset better than conjunctions and wildcards, except when it comes to perfom in the medium and large datasets where it needs nearly 1000 seconds to query 3m documents, which is about the same runtime with conjunctive queries.

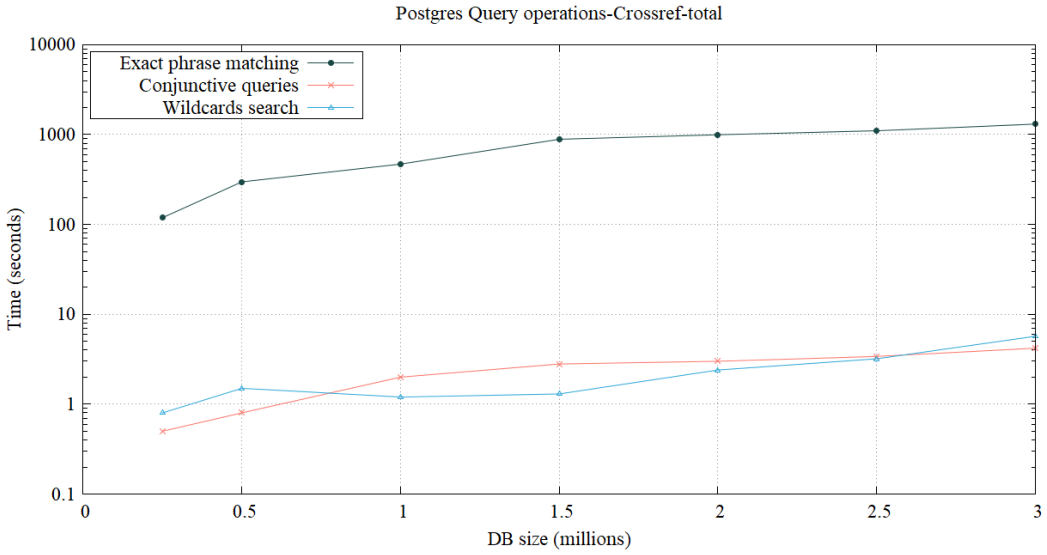

Figure 4.70: Postgres Query operations Crossref total.

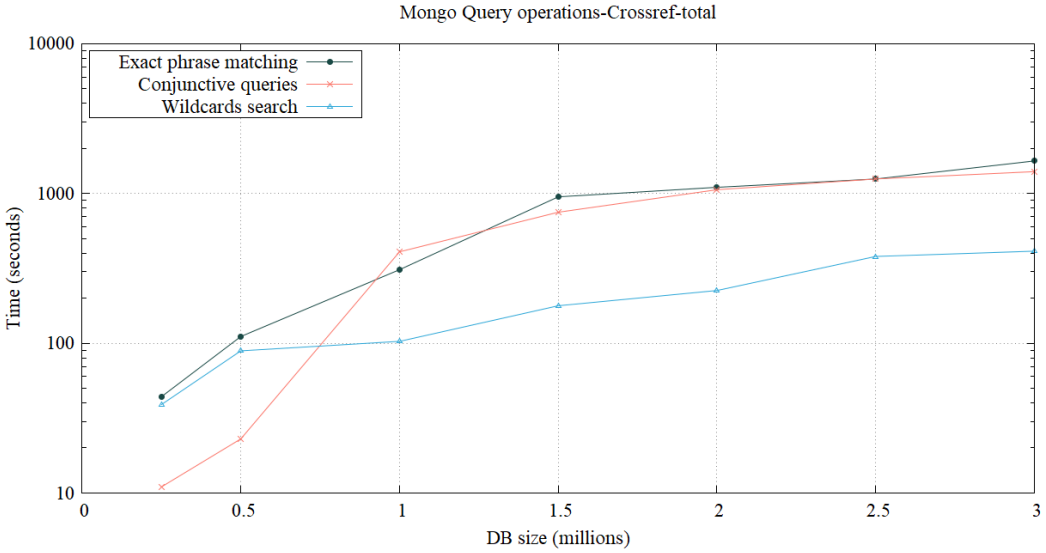

Figure 4.71: Mongo Query operations Crossref total.

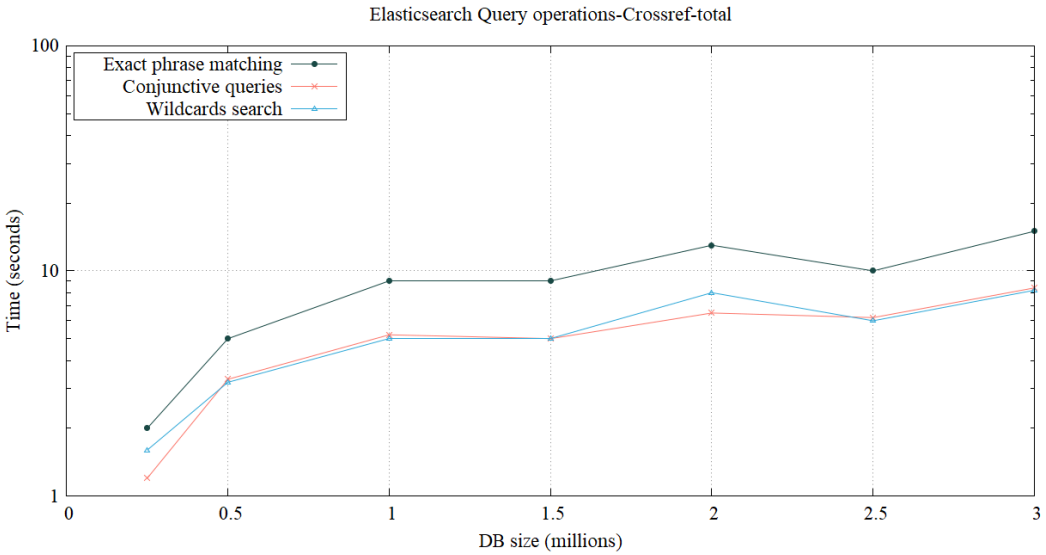

Figure 4.72: Elasticsearch Query operations Crossref total.

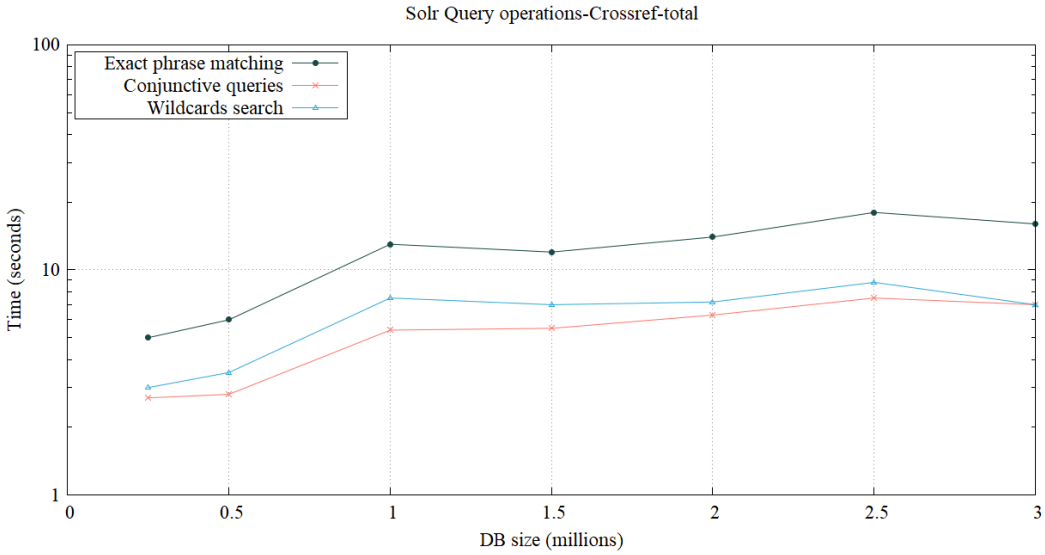

Figure 4.73: Solr Query operations Crossref total.

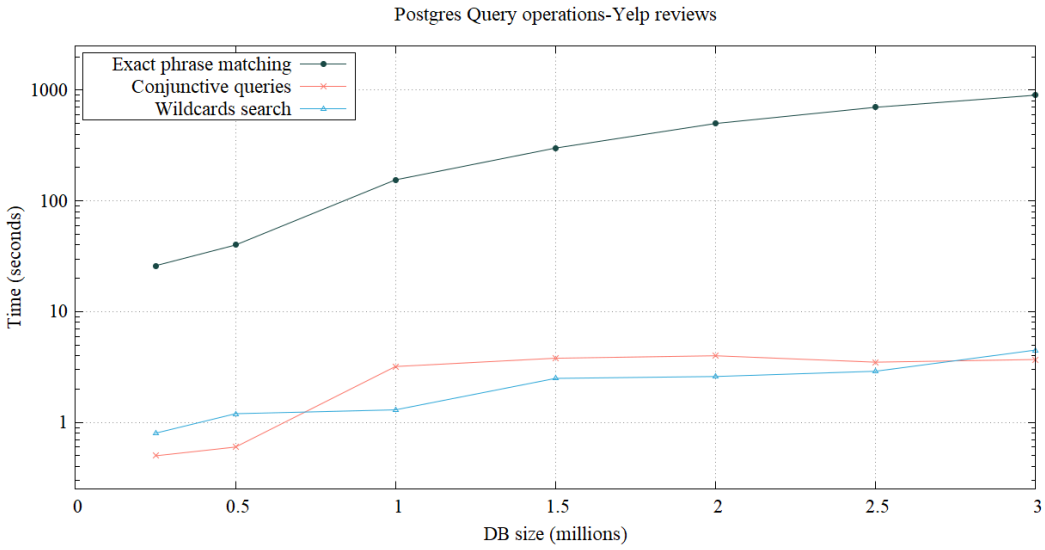

Figure 4.74: Postgres Query operations Yelp reviews.

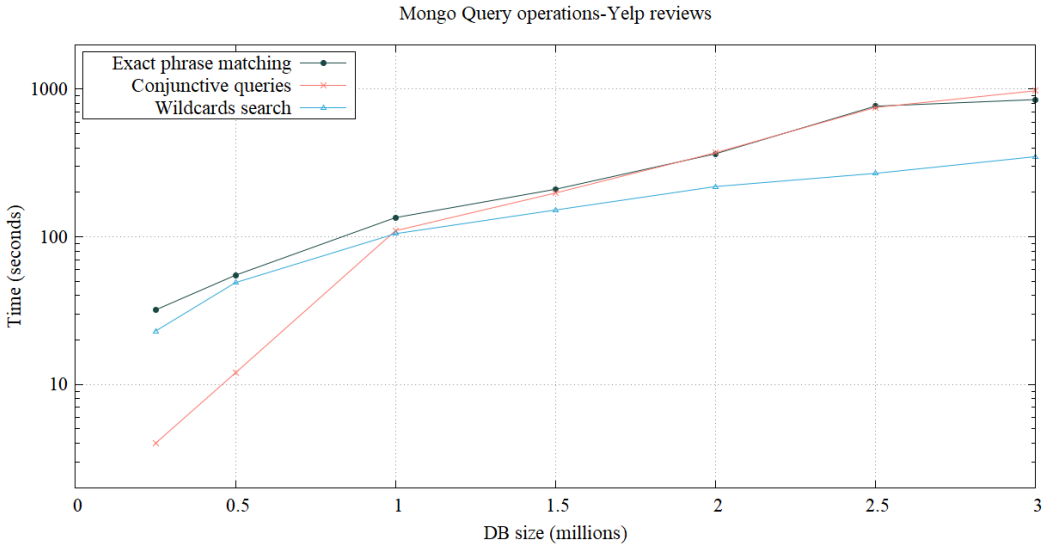

Figure 4.75: Mongo Query operations Yelp reviews.

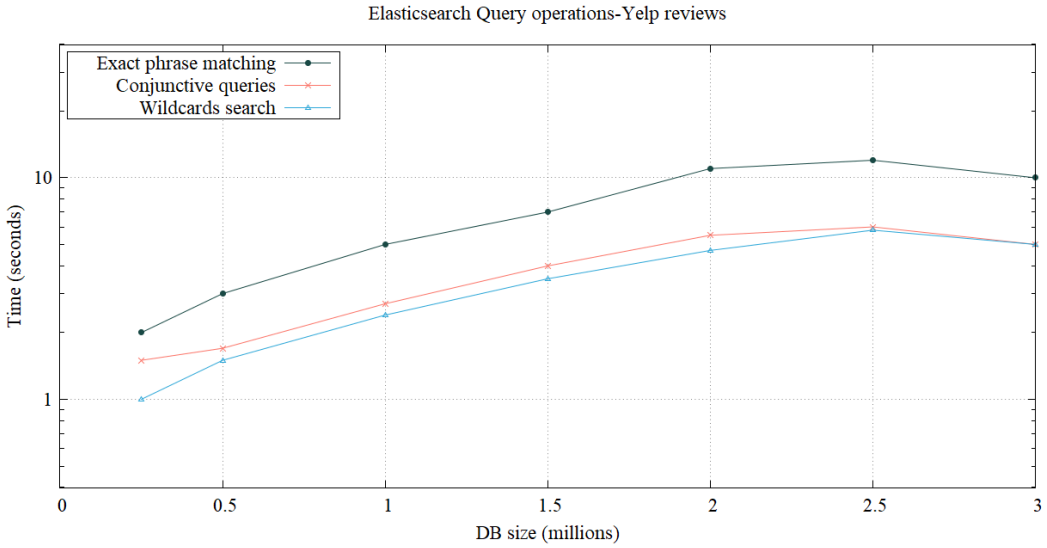

Figure 4.76: Elasticsearch Query operations Yelp reviews.

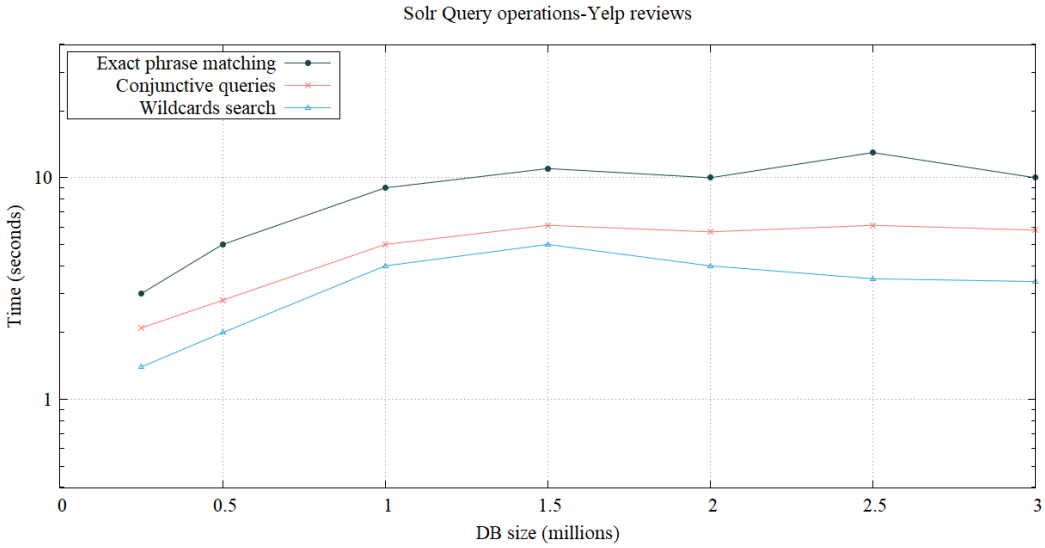

Figure 4.77: Solr Query operations Yelp reviews.

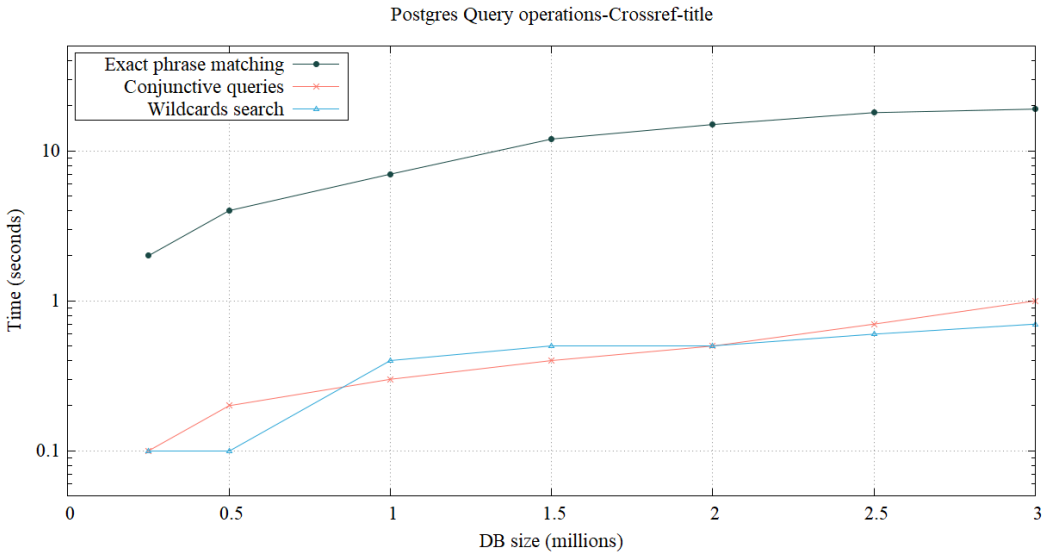

Figure 4.78: Postgres Query operations Crossref title.

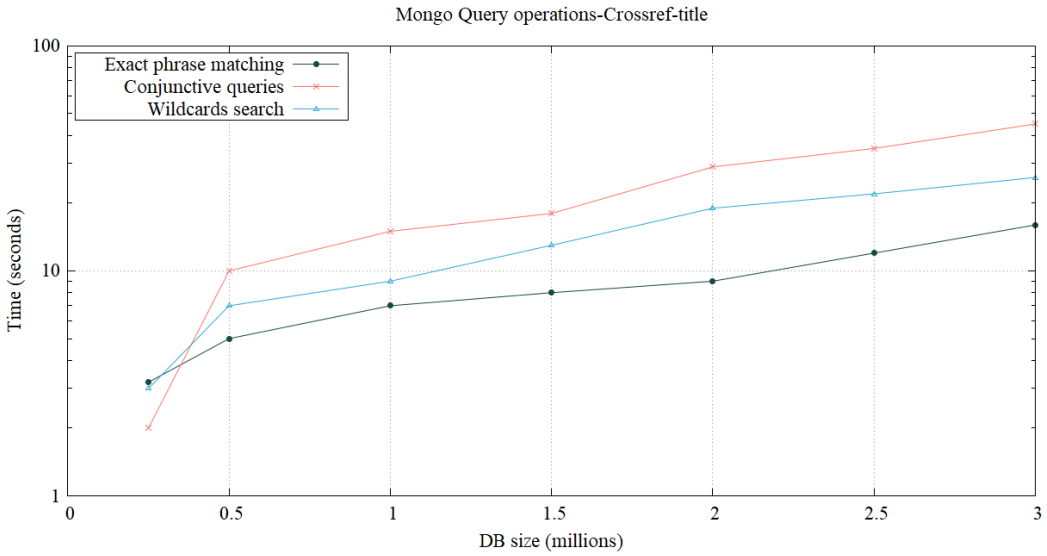

Figure 4.79: Mongo Query operations Crossref title.

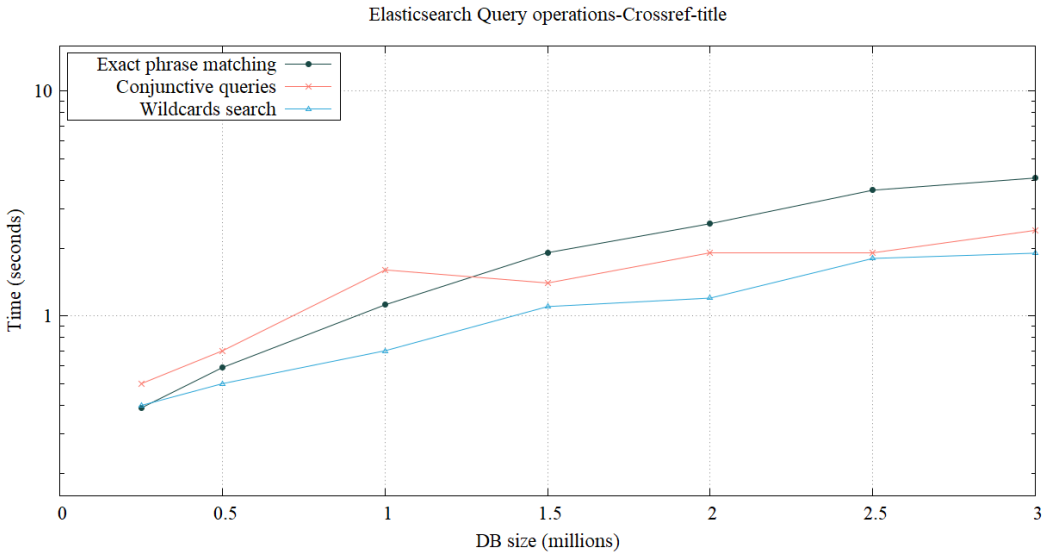

Figure 4.80: Elasticsearch Query operations Crossref title.

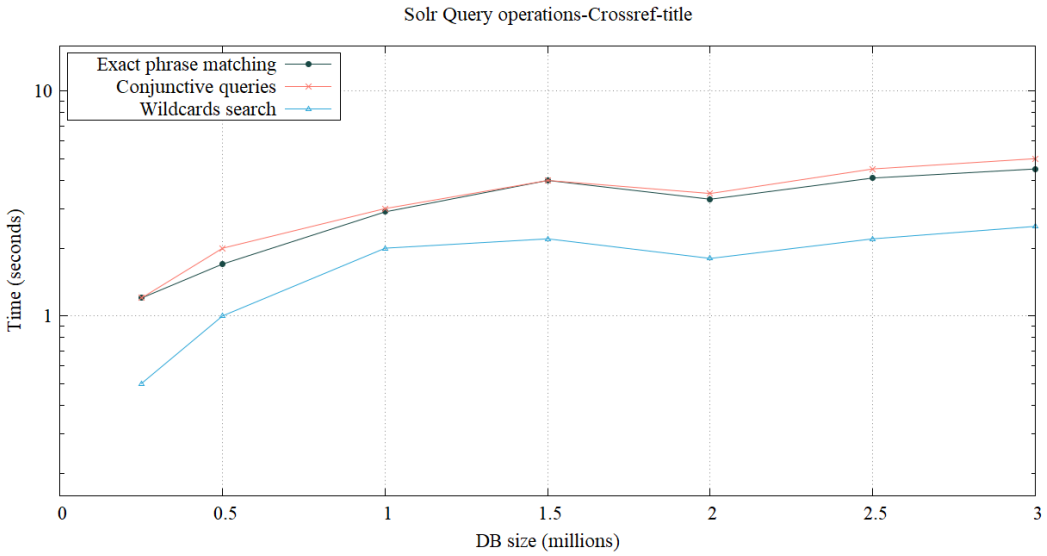

Figure 4.81: Solr Query operations Crossref title.

What we conclude is that, the majority of databases face difficulties while executing exact phrase matching queries in full-text search especially with large amount of data, while on the other hand, wildcards search and conjunctive queries seem more slightly operations as the systems achieve better results.

4.4 Comparing data insertion time

In this chapter, we present the results concerning the time needed to import the documents into the database systems. Hence, the data insertion time, is the time taken for each dataset to be uploaded into the databases.

In Figures 4.82, 4.83 and 4.84, we can see the results for all the database systems' data insertion time in regard to the database size, for each one of the datasets (i.e., small, medium and large). We observe that as the database size increases, the more time is needed to insert new documents into the systems, without any special fluctuations. The same behaviour holds for all database systems.

Notice that, unlike the experiments we conducted on the querying time in the previous sections, Solr now seems to need less time than Elasticsearch to insert new documents and operates relatively fast, whereas Postgres, is the fastest among all databases. For Mongo, it takes more time to operate at data insertion compared to

the rest of the systems.

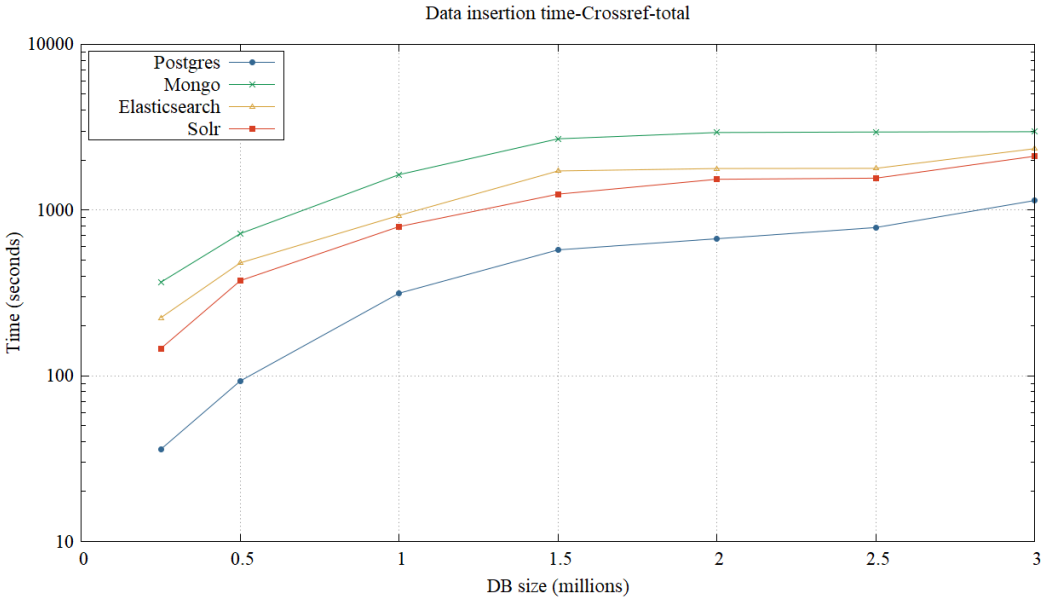

Figure 4.82: Data insertion time Crossref total.

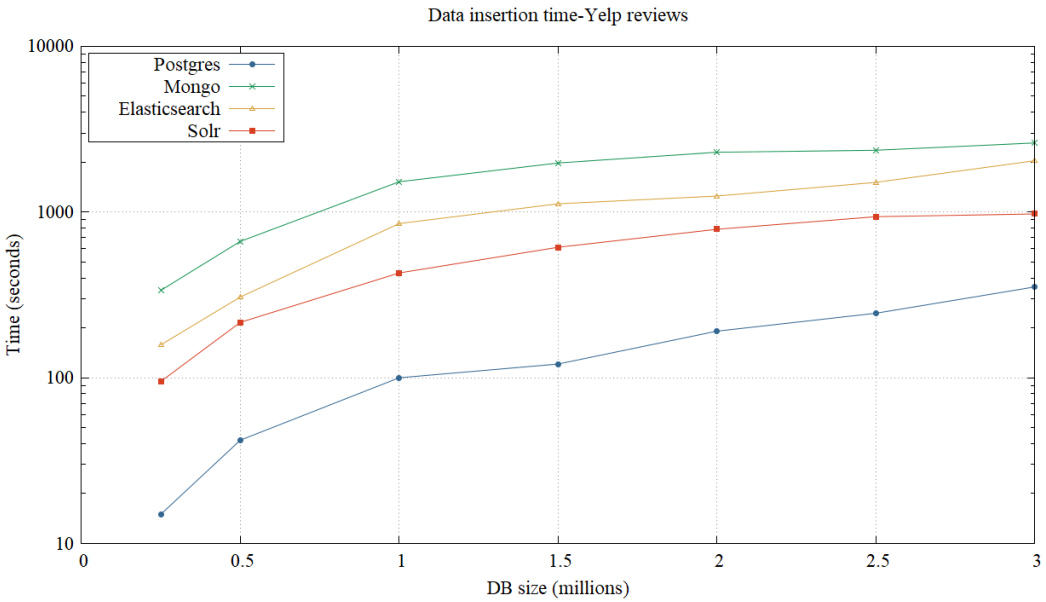

Figure 4.83: Data insertion time Yelp reviews.

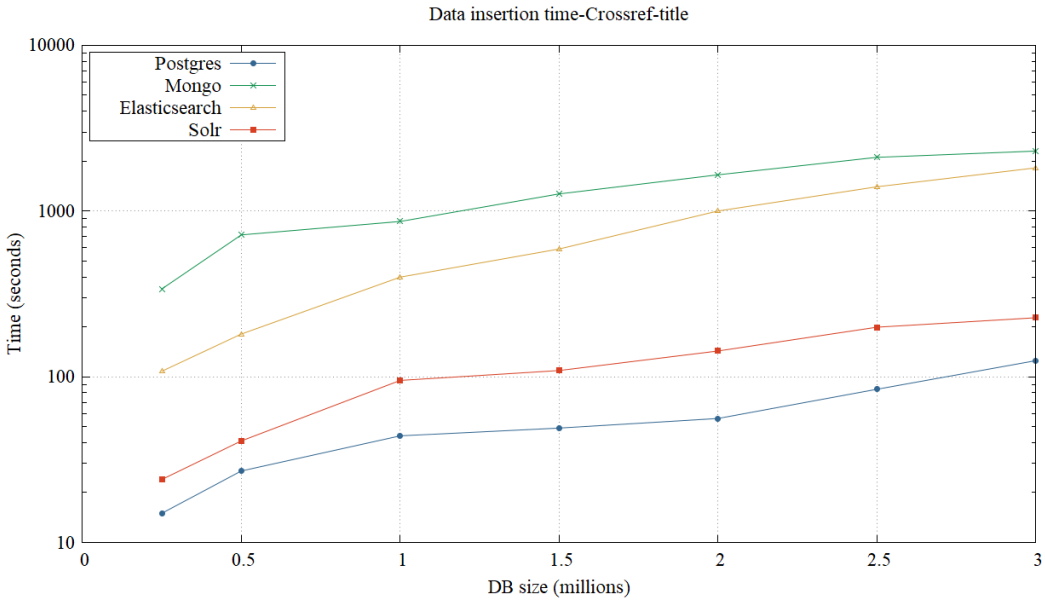

Figure 4.84: Data insertion time Crossref title.

In Figures 4.85, 4.86, 4.87. and 4.88, we can see an alternative view of the previous results, comparing the time needed to import the datasets (i.e., small, medium and large), in regard to the database size for Postgres, Mongo, Elasticsearch and Solr respectively. We observe that Postgres and Solr reach the limit of 3m documents with a significant variance for each dataset size, as Mongo and Elasticsearch seem to converge at this limit. Moreover, Mongo starts from the same point of approximately 500 seconds to insert 250k documents for all three datasets, but this time difference is increased as it moves to even higher values.

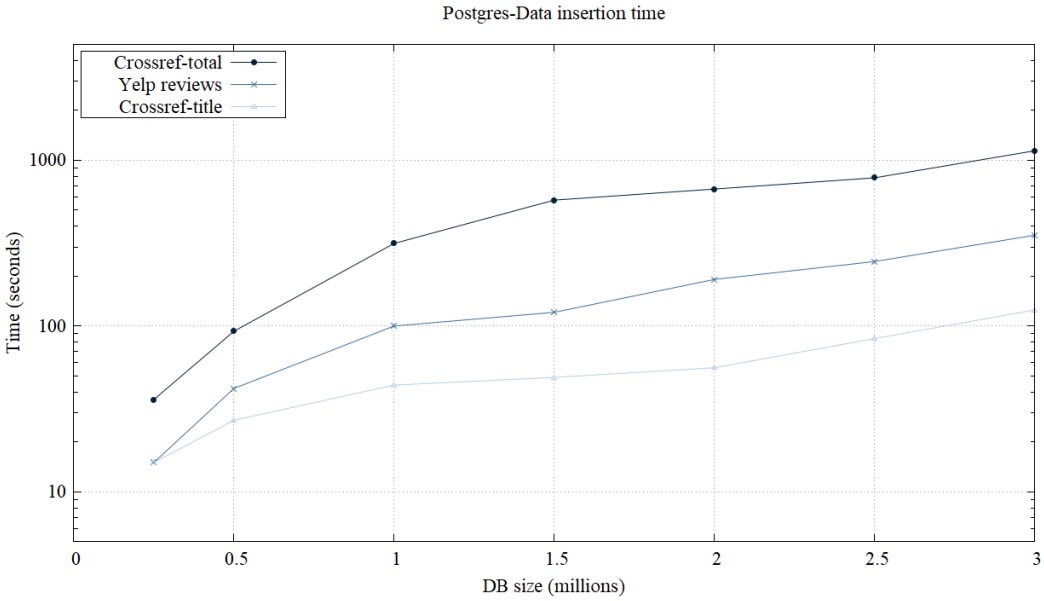

Figure 4.85: Postgres Data insertion time.

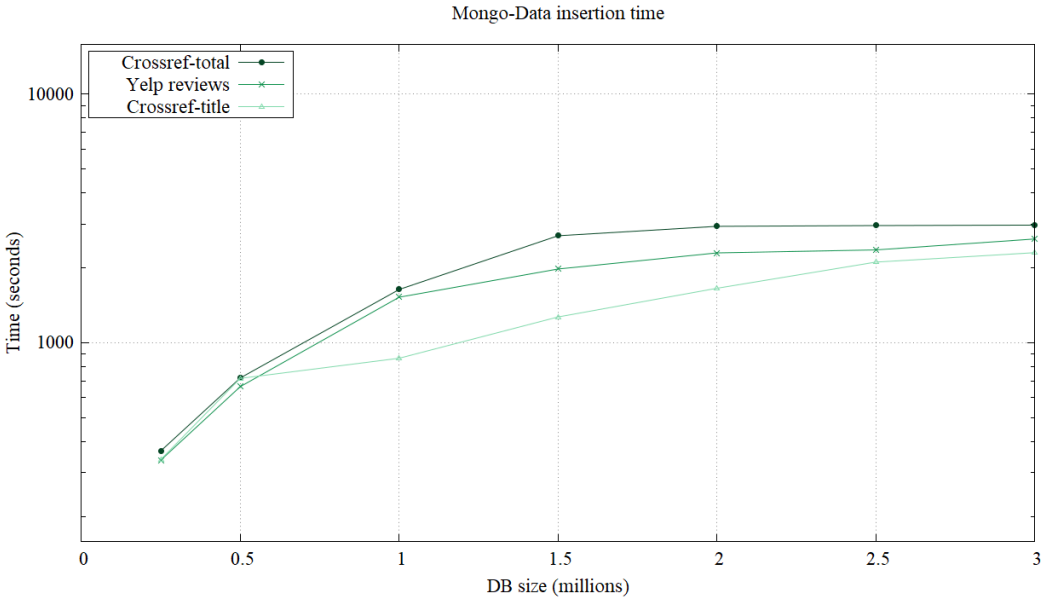

Figure 4.86: Mongo Data insertion time.

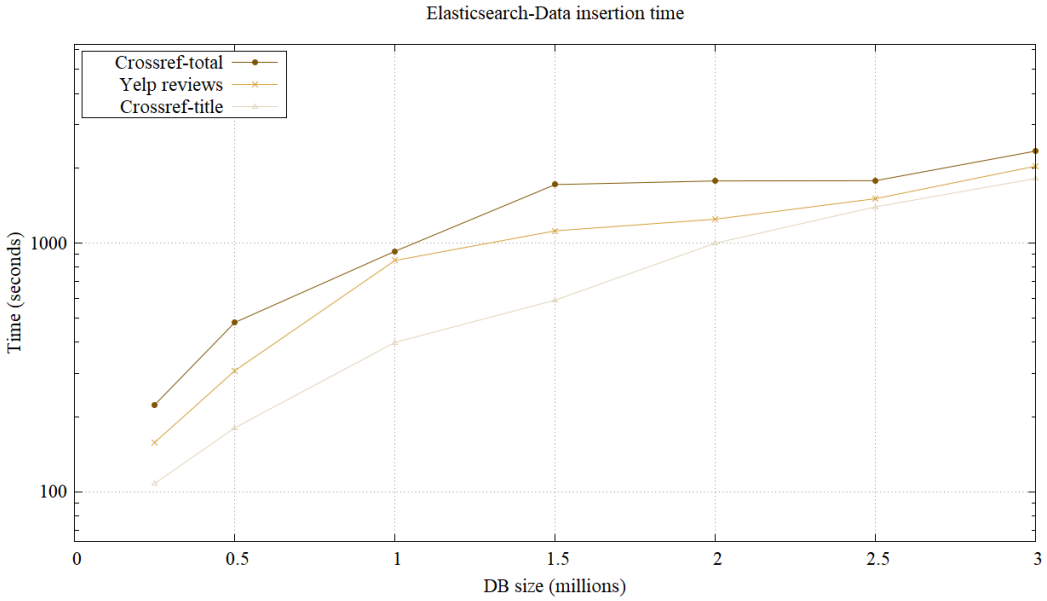

Figure 4.87: Elasticsearch Data insertion time.

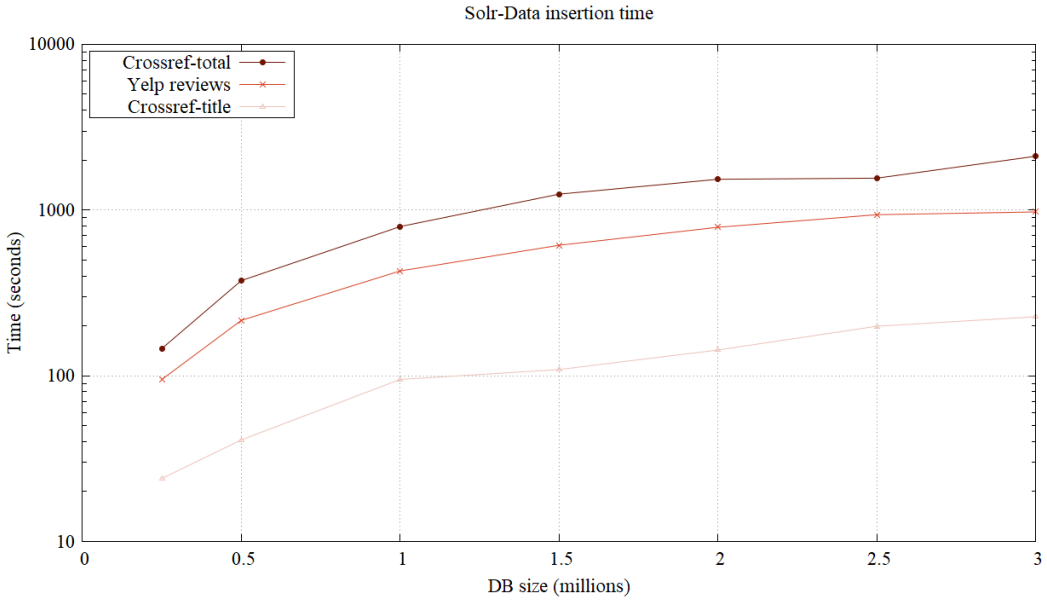

Figure 4.88: Solr Data insertion time.

4.5 Comparing memory usage

We have also executed experiments to specify memory requirements for each of the presented database systems of our study. To specify memory usage, we have measured the amount of memory that have been consumed for the processes of indexing and querying.

Figures 4.89, 4.90 and 4.91 exhibit an overview of the results for memory consumption of the indexing process. What we observe is that Solr and Elasticsearch have high memory requirements for data indexing and both reach the limit of 2000MB for the Crossref-total dataset (the large dataset). The memory consumption corresponds to the time needed for data insertion for both of the systems. On the other hand, Mongo consumes significantly less memory compared to the other systems. The memory requirements of Mongo database system do not correspond to the time needed for data insertion and indexing. On the other hand, Postgres needs on average 1000 to 1500MB of memory usage for the same operations and proves consistent with the previously presented results.

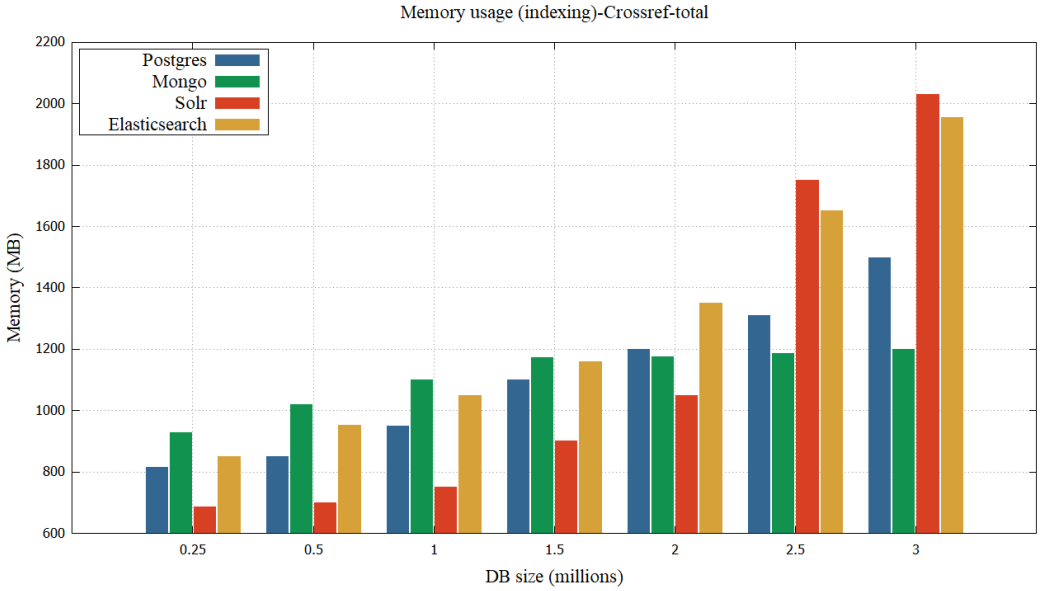

Figure 4.89: Memory usage (indexing) Crossref total.

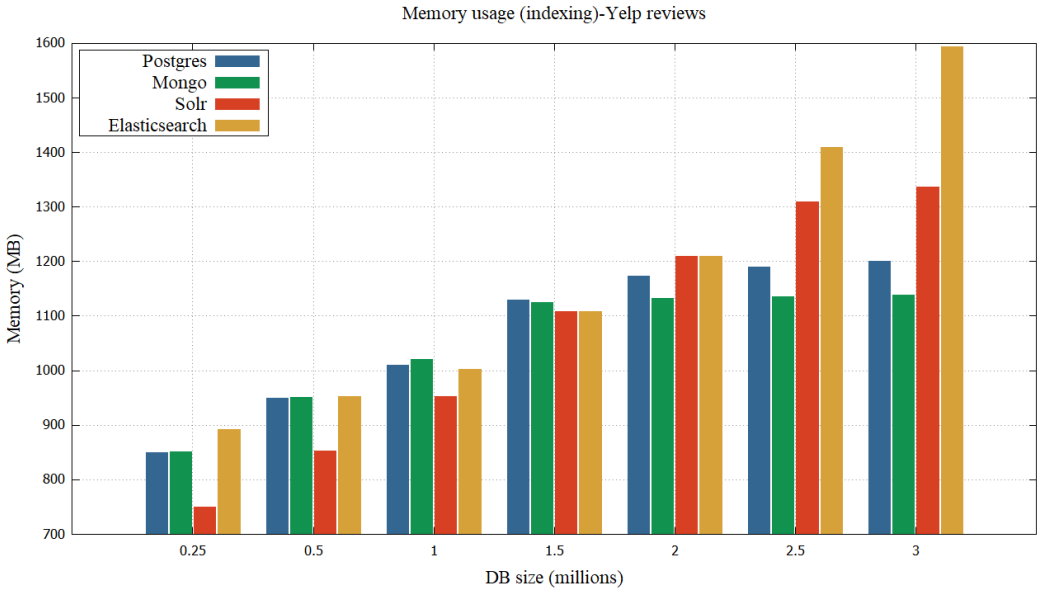

Figure 4.90: Memory usage (indexing) Yelp reviews.

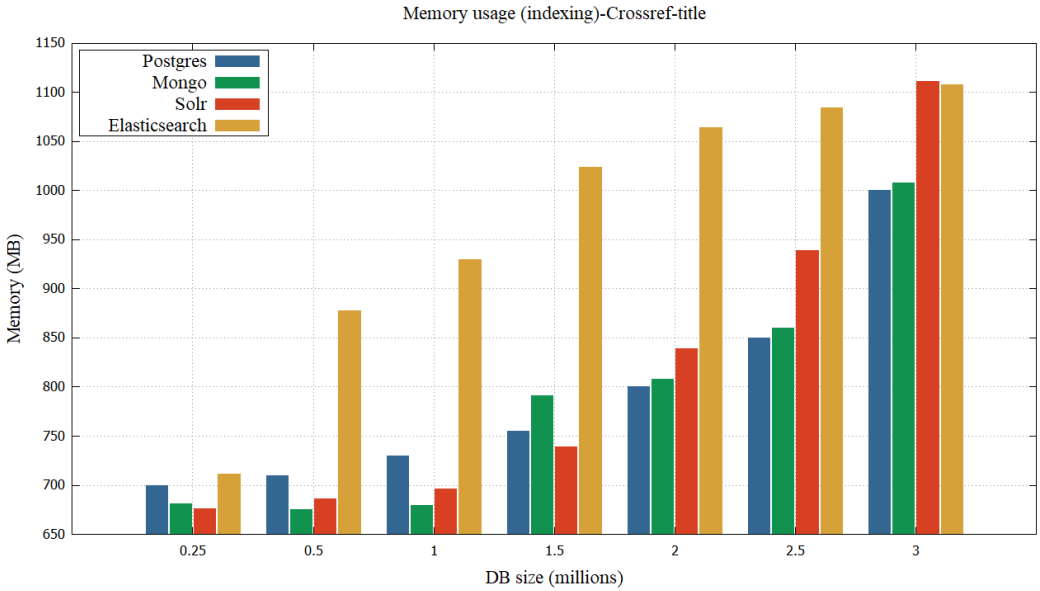

Figure 4.91: Memory usage (indexing) Crossref title.

The final set of figures (Figures 4.92, 4.93 and 4.94) exhibit an overview of the results for memory consumption of the querying process. In this case, the difference in memory usage is clearly obvious among the database systems. Solr and Elasticsearch consume significantly less memory compared to Mongo and Postgres, accordingly to the time needed for the querying operations that we previously analysed in our experiments.

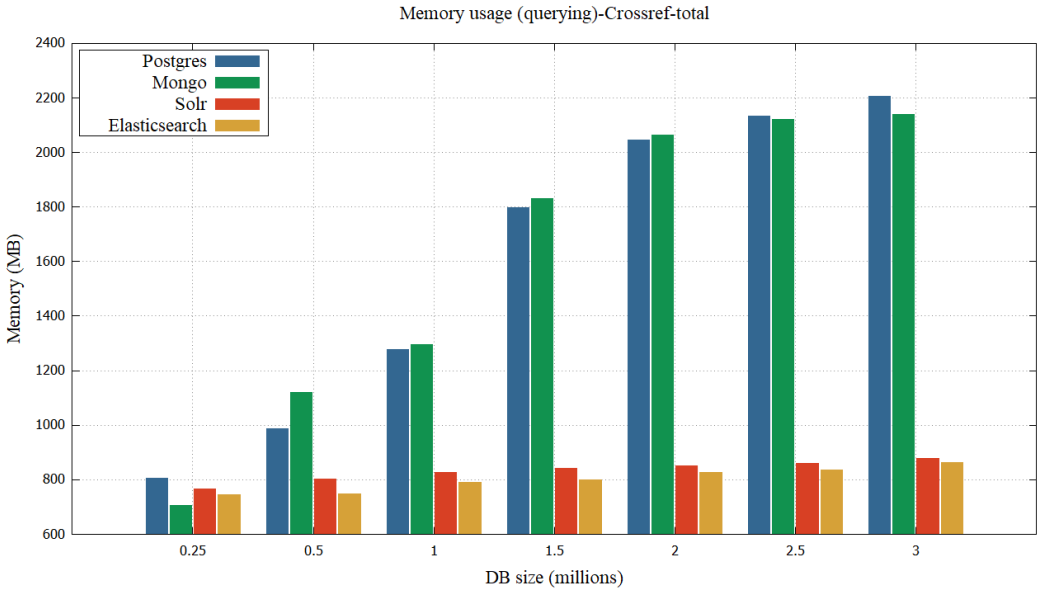

Figure 4.92: Memory usage (querying) Crossref total.

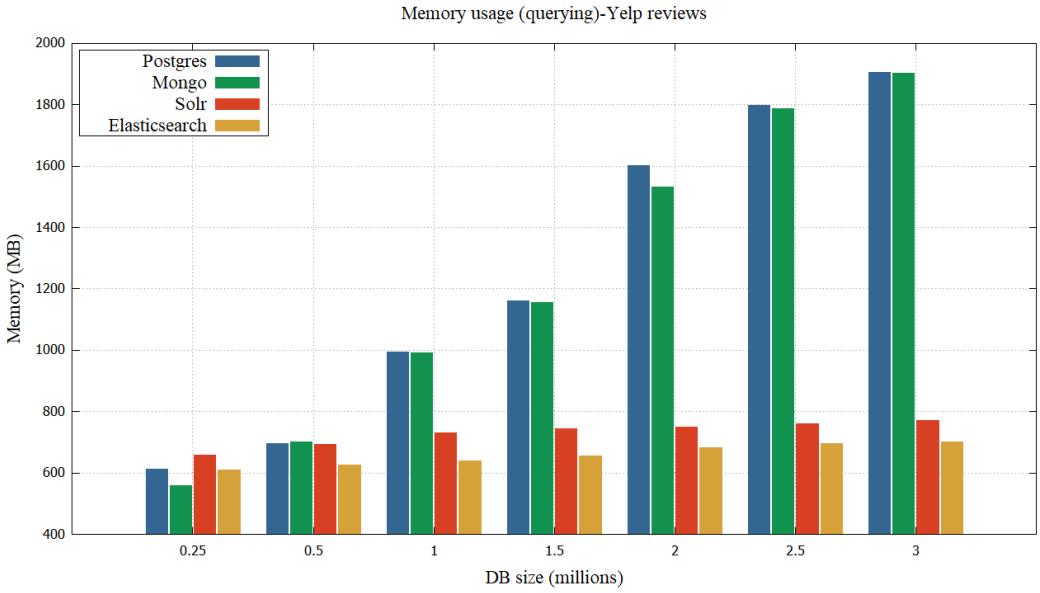

Figure 4.93: Memory usage (querying) Yelp reviews.

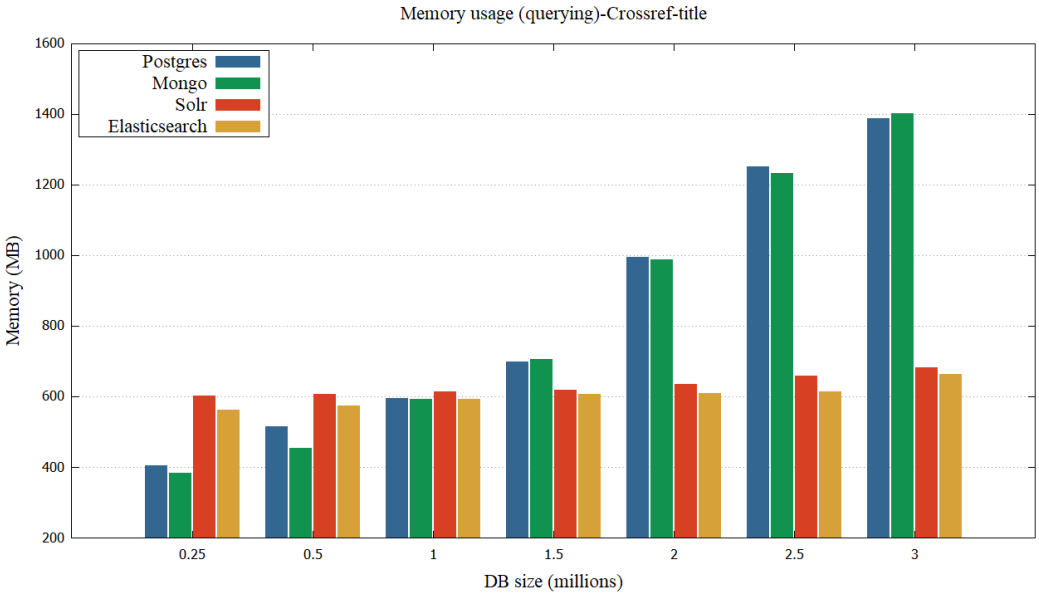

Figure 4.94: Memory usage (querying) Crossref title.

Chapter 5

Conclusions and Future Work

In this paper we studied and compared some of the most used relational and non-relational database systems and we evaluated their performance in full-text search experimenting with datasets of varying sizes.
After presenting the theoretical background, we tested the systems through a series of experiments by executing the operations of exact phrase matching, wildcards search, conjunctive queries and extended conjunctive queries on a range of 250k to 3m records for each one of the datasets. We have also tested the systems in terms of selectivity applying high, medium and small selective queries on each one of the query operations. Finally, we measured the time needed for data insertion and we examined the memory consumption of the systems for the processes of indexing and querying.

For the different query operations we have tested the systems, we observed the following. Elasticsearch and Solr have responded in the experiments with balanced time measurements in proportion to the dataset size and without any fluctuations in the totality of cases. Postgres proved to perform much better with the use of the GIN index, competing non-relational systems and outperforming Mongo in the query oprations of wildcards search and conjunctions, but struggles when it has to query a large proportion of data coupled with a high memory consumption. Memory usage also seems to affect Mongo which performed significantly slow compared to the rest of the non-relational systems.

We have also deduced how selectivity affects the performance in full-text search. We first showcased the example of exact phrase matching operation where the systems responded without any serious influence during the querying process, whereas with conjunctive queries the databases showed a dissimilar behaviour while operating with high or small selective queries in a different proportion of the selected data. Next we noticed that, the majority of the systems perform better with wildcards search and conjunctive queries, while on the other hand, face difficulties while executing exact phrase match queries on a large amount of data. Finally, we inspected the performance during the data insertion operation where we determined that he relational system has an advantage over the non-relational and the text search databases.

What we conclude from these experiments, is that non-relational and text search databases provide a trustworthy alternative in full-text search while performing with small or huge amount of data. We also observe that the relational model is sensitive to its parameters, which in this case is data indexing, despite the size of the dataset that was utilized. We can conclude that some relational databases can operate well in certain query operations and in specific amount of data under conditions.

For future work on this study, one thing we would like to see would be testing the systems in a larger amount of data, for example 10m or 100m documents. The limit of 3m documents acted as a threshold in some cases where databases achieved the same time measurements and we would like to see the runtimes beyond that limit. Apart from this, we would also like to test the systems on datasets with free-form text or imbalanced data such as web pages, social media content, emails, dates, etc. Finally, selecting different database systems to put in comparison and test their capabilities could also be left as future work. Some example databases are CouchDB, Cassandra, Spinx and SQL Server.

References

[1] Full-text search.
https://en.wikipedia.org/wiki/Full-text-search

[2] Full-Text search queries.
https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver15

[3] Ameya Nayak, Anil Poriya, and Dikshay Poojary. *Type of NOSQL Databases and its Comparison with Relational Databases*. International Journal of Applied Information Systems 5(4):16-19, March 2013.

[4] Mohamed, Mohamed A., Obay G. Altrafi, and Mohammed O. Ismail. *"Relational vs. Nosql databases: A survey"*. International Journal of Computer and Information Technology 3.03 (2014): 598-601.

[5] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi and F. Ismaili, *"Comparison between relational and NOSQL databases"*, 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 0216-0221, doi: 10.23919/MIPRO.2018.8400041.

[6] Relational Data Model in DBMS.
https://www.guru99.com/relational-data-model-dbms.html

[7] Relational database.
https://en.wikipedia.org/wiki/Relational_database

[8] What Is a Non-Relational Database?.
https://www.mongodb.com/databases/non-relational

[9] Non-relational data and NoSQL.
https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data

[10] NoSQL.
https://en.wikipedia.org/wiki/NoSQL

[11] Jatana, Nishtha, et al. *"A survey and comparison of relational and non-relational database"*. International Journal of Engineering Research & Technology 1.6 (2012): 1-5.

[12] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi and F. Ismaili, *"Comparison between relational and NOSQL databases"*, 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 0216-0221, doi: 10.23919/MIPRO.2018.8400041.

[13] Čerešňák R., Kvet M. *Comparison of query performance in relational and non-relational databases*. Transportation Research Procedia, 40 (2019), pp. 170-177, 2352-1465.

[14] Z. Parker, S. Poe, and S. V. Vrbsky. *Comparing nosql mongodb to an sql db*. Proceedings of the 51st ACM Southeast Conference, ACM, April 2013.

[15] Cornelia Győrödi, Robert Győrödi, George Pecherle, Andrada Olah. *A comparative study: MongoDB vs. MySQL*. IEEE - 13th International Conference on Engineering of Modern Electric Systems (EMES), 2015, Oradea, Romania, 11-12 June 2015, ISBN 978-1-4799-7649-2, pag. 1-6.

[16] Aghi, R., Mehta, S., Chauhan, R., Chaudhary, S., Bohra, N. *A comprehensive comparison of SQL and MongoDB databases*. Int. J. Sci. Res. Publ. 5(2), (2015).

[17] Yishan Li and Sathiamoorthy Manoharan. *A performance comparison of SQL and NoSQL databases*. 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2013, pp. 15-19, doi: 10.1109/PACRIM.2013.6625441.

[18] Lucidworks:Full Text Search Engines vs. DBMS.
https://lucidworks.com/post/full-text-search-engines-vs-dbms/

[19] AnyTXT Searcher:Lucene vs Solr vs ElasticSearch 2021.
https://anytxt.net/how-to-choose-a-full-text-search-engine/

[20] PostgreSQL About.
https://www.postgresql.org/about/

[21] PostgreSQL Wikipedia.
https://en.wikipedia.org/wiki/PostgreSQL

[22] PostgreSQL vs. MySQL: A 360-degree Comparison [Syntax, Performance, Scalability and Features]
https://www.enterprisedb.com/

[23] PostgreSQL. Full Text Search Tutorial.
https://linuxhint.com/postgresql-full-text-search-tutorial/

[24] PostgreSQL. Text Search Functions and Operators
https://www.postgresql.org/docs/12/functions-textsearch.html

[25] Mongo Wikipedia.
https://en.wikipedia.org/wiki/MongoDB

[26] MongoDB.com: Why Use MongoDB and When to Use It?
https://www.mongodb.com/why-use-mongodb

[27] Dotnettricks: Learn MongoDB PostgreSQL vs. MySQL: A 360-degree Comparison [Syntax, Performance, Scalability and Features]
https://www.dotnettricks.com/learn/mongodb/

[28] GeeksforGeeks: What is a MongoDB Query?
https://www.geeksforgeeks.org/what-is-a-mongodb-query/

[29] Mongo Text Search.
https://docs.mongodb.com/manual/text-search/

[30] Elasticsearch Wikipedia.
https://en.wikipedia.org/wiki/Elasticsearch

[31] Elasticsearch Query DSL.
https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html

[32] Getting Started with Elasticsearch Query DSL.
https://towardsdatascience.com/getting-started-with-elasticsearch-query-dsl-c862c9d6cf7f

[33] Elasticsearch: Full text queries.
https://www.elastic.co/guide/en/elasticsearch/reference/6.8/full-text-queries.html

[34] Apache Solr org.
https://solr.apache.org

[35] Solr vs. Elasticsearch: Who's The Leading Open Source Search Engine?
https://logz.io/blog/solr-vs-elasticsearch/

[36] The Standard Query Parser.
https://solr.apache.org/guide/7_0/the-standard-query-parser.html

[37] Apache Solr Wikipedia.
https://en.wikipedia.org/wiki/Apache_Solr

[38] PostgreSQL Copy Example.
https://kb.objectrocket.com/postgresql/postgresql-copy

[39] GiST and GIN Index Types.
https://www.postgresql.org/docs/9.1/textsearch-indexes.html

[40] How to create a database in MongoDB.
https://www.mongodb.com/basics/

[41] Elasticsearch mapping.
ttps://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html

[42] Solr Cores and solr.xml.
https://solr.apache.org/guide/6.6/solr-cores-and-solr-xml.html