Comparing Data Store Performance for Full-Text Search: to SQL or to NoSQL?

George Fotopoulos,[2] Paris Koloveas,[1] Paraskevi Raftopoulou,[1] and Christos Tryfonopoulos[1]
[1]*Department of Informatics & Telecommunications, University of the Peloponnese, Tripolis, Greece*
[2]*WITSIDE (Intelligence for Business Ltd), Athens, Greece*
georgios.fotopoulos@witside.com, {pkoloveas, praftop, trifon}@uop.gr

Keywords: database performance, text search, NoSQL data stores, relational databases, performance comparison

Abstract: The amount of textual data produced nowadays is constantly increasing as the number and variety of both new and reproduced textual information created by humans and (lately) also by bots is unprecedented. Storing, handling and querying such high volumes of textual data have become more challenging than ever and both research and industry have been using various alternatives, ranging from typical Relational Database Management Systems to specialised text engines and NoSQL databases, in an effort to cope with the volume. However, all these decisions are, largely, based on experience or personal preference for one system over another, since there is no performance comparison study that compares the available solutions regarding full-text search and retrieval. In this work, we fill this gap in the literature by systematically comparing four popular databases in full-text search scenarios and reporting their performance across different datasets, full-text search operators and parameters. To the best of our knowledge, our study is the first to go beyond the comparison of characteristics, like expressiveness of the query language or popularity, and actually compare popular relational, NoSQL, and textual data stores in terms of retrieval efficiency for full-text search. Moreover, our findings quantify the differences in full-text search performance between the examined solutions and reveal both anticipated and less anticipated results.

1 Introduction

In contrast to keyword search, which is a method of searching for documents that include the exact keywords specified by a user, *full-text search* refers to searching some text inside computer-stored documents (or a collection of documents in a data store) and returning results that contain *some or all* of the words from the query posed by a user (Schuler et al., 2009). *Full-text queries*, which can include either simple words and phrases or multiple forms of a word or phrase, perform linguistic searches against text data and return any documents that contain at least one match (Microsoft, 2023).

Relational (or else SQL) databases excel at storing data in tables, rely on a static schema to define the structure and relationships between tables, and are vertically scalable. Most relational databases provide support for keyword search, even in the case when some field in a record includes *free-form text* (like a product description); this approach gives results that miss the precision of the relevancy ranking provided by full-text search systems (Lucidworks, 2019). Non-relational (or else NoSQL) data stores, on the other hand, store data in a variety of formats, such as collections, documents, key-value pairs, and graph databases, use a dynamic schema, and are horizontally scalable, being more flexible and adaptable to changing data structures. Text search systems and non-relational data stores are better for quickly searching high volumes of structured, unstructured, or semi-structured textual data according to a specific word, bag of words, or phrase, since they provide rich text search capabilities and give sophisticated relevancy ranking of results (McCreary and Kelly, 2013).

Several works in the literature have evaluated the performance and have compared relational to non-relational data stores, both in the context of features, but also with respect to the advantages and disadvantages of each system type. In their survey, Nayak et al. (Nayak et al., 2013) analyse the different types and characteristics of SQL and NoSQL systems, while in (Mohamed et al., 2014; Sahatqija et al., 2018), SQL/NoSQL data stores are compared in terms of main features, such as scalability, query language, security issues, etc. Along the same lines, (Jatana

et al., 2012) provides a general comparison of relational and non-relational data stores, while (Lourenco et al., 2015) reviews NoSQL data stores in terms of the consistency and durability of the data stored, as well as with respect to their performance and scalability; the results indicate that MongoDB can be the successor of SQL databases, since it provides good stability and consistency of data.

Query execution time and related system performance measurements have also been examined in the relevant literature. The work in (Brewer, 2012) compares the performance of HBase and MySQL, and concludes that under the same scenarios, the NoSQL alternative is faster than the relational one, while (Li and Manoharan, 2013) compares several NoSQL databases and SQL Server Express, to observe that NoSQL databases are not always faster than the SQL alternative. The work in (Fraczek and Plechawska-Wojcik, 2017) is a comparative analysis of relational and non-relational databases in the context of data reading and writing in web applications, and demonstrates that MongoDB performs better at reading data, while PostgreSQL at writing. In (Truica et al., 2015), the performance of document data stores and relational databases is compared, concluding that CouchDB is the fastest during insertion, modification and deletion of data, while MongoDB is the fastest at reading. Similarly, the work in (Čerešňák and Kvet, 2019) compares popular relational database storage architectures with non-relational systems and shows that MongoDB outperforms its competitors.

Despite the explosion in textual data over the last decade, mainly driven by the contribution of textual content in social networks in the form of posts, comments and forum threads, surprisingly, there is no study concerning the performance in terms of the efficiency of popular data stores when it comes to the retrieval of big (unstructured) textual data. It is worth noting that the available comparisons of full-text search enalbed systems are mainly technical news articles and research works (Lucidworks, 2019; AnyTXT, 2021; Carvalho et al., 2022) that focus on the qualitative characteristics of the systems (e.g., expressiveness of the query language, popularity, maturity, ease of use, and community support), and neither consider any quantitative elements nor perform any kind of efficiency comparison. In this work, we present a *comparative study* among four popular relational and non-relational solutions that offer full-text search capabilities, *highlighting* the overall characteristics of each data store technology *in text retrieval*. In the light of the above, our contributions can be summarised as follows:

- We systematically compare four popular data stores (namely PostgreSQL, MongoDB, ElasticSearch, Apache Solr) in full-text search scenarios and report their performance efficiency across datasets of different sizes, various full-text search operators, and a wide variety of parameters.
- We quantify the performance differences between the examined systems, highlight the best option for each full-text search scenario and provide guidelines for the system and parameter setup.
- We openly distribute[1] our experimental setup and configuration both in the form of source code and as a ready-to-use virtual machine that may be used by the research community to replicate, verify and/or expand our findings.

Note that, although vector space queries are an important part of full-text search, we have deliberately left them out of our comparison since most of the examined systems have no inherent support for them and any custom implementation from our side would interfere with the objectivity of the reported results.

The paper is structured as follows. In Section 2, we outline the textual datasets utilised for the performance comparison, the systems under examination along with some of their important features relevant to our task, and the necessary configurations to create the testbed, i.e. the parameters and indexes used for each system. In Section 3, we present the experimental setup, including the full-text queries that we use in our evaluation, and present the experimental results of our study. Finally, Section 4 outlines the work and gives some future directions.

2 Data and System Configuration

In this section, we introduce the datasets used, present the systems compared and outline their *querying* capabilities, focusing specifically to *full-text retrieval*, and provide technical specifications on the testbed, including the machine characteristics and specialised software that was used to execute the comparison.

2.1 Datasets configuration

The first dataset used is the *Crossref* (CR) dataset; it consists of publication metadata and it has been published by the Crossref organisation on January 2021.[2] The dataset is 38.5GB, contains 12.1 million records, and each record contains the fields: *doi*, *ti-*

[1] https://github.com/pkoloveas/DATA2023-FKRT-resources

[2] https://www.crossref.org/

Dataset	Size (Records)	Fields	Avg words per record
Crossref	4.1GB (3M)	'title', 'abstract'	228
Yelp reviews	1.8GB (3M)	'review_id', 'review'	104

Table 1: Dataset characteristics

tle, *year_published*, *date_published*, *author*, *journal*, *domain*, and *abstract*.

The second dataset used is the *Yelp reviews* (Yelp) dataset; it consists of reviews and recommendations written and posted by consumers, suggesting restaurants, hotels, bars, shopping, etc. The dataset has been downloaded from Kaggle,[3] is 3.8GB, contains 5.3 million records, and each record contains the fields: *review_id*, *user_id*, *business_id*, *stars*, *date*, *review*, *useful*, *funny*, and *cool*.

Since our task was to measure full-text search performance, not all the aforementioned fields were required. We used the *title* and *abstract* fields from the *Crossref* dataset and the *review_id* and *review* fields from the *Yelp* dataset. The resulting datasets and their key characteristics are summarised in Table 1. Finally, in order to perform the experiments on *varying sizes*, we randomly split the (constructed) datasets into smaller batches in the range of 500K to 3M records ($\mathcal{D}$) with a 500K step.

2.2 Systems configuration

For the performance comparison we selected four popular systems, namely, *PostgreSQL*, *MongoDB*, *ElasticSearch*, and *Apache Solr*, that incorporate full-text capabilities and are typical representatives of the broader SQL, NoSQL, and search engine categories. The main criteria of our selection were the usability and robustness of the systems, their wide acceptance among the users, and their ability to execute full-text queries on unstructured text. In the following sections, we briefly introduce each system, outline its technical configurations with respect to the performance comparison carried out, and state any system-specific indexes that were used.

PostgreSQL A free and open-source object-relational database management system that uses the SQL language. Postgres features transactions with ACID properties and is designed to handle a range of workloads, from single machines to data warehouses and 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.

For our testbed, we installed and used version 13.3. We also created the databases for each dataset. Then, in each database we created tables for the different batches of the datasets, i.e., for 500K-3M records. To configure Postgres for full-text search use, we created a new column for each table, named "*document*" of data type "*tsvector*", containing the fields we wanted to use for text search.

Postgres provides two index types that can be used to index tsvector data types: Generalized Inverted Index (*GIN*) and Generalized Search Tree (*GiST*). We used the *GIN* index for our setup since it is recommended by Postgres as the default index type for full-text searching, especially for long documents.

MongoDB A source-available cross-platform document-oriented data store; it is classified as a NoSQL system and stores data in JSON-like documents with dynamic schema in the form of *(field: value, pair)* rather than tabular form. Mongo provides high performance, high availability, easy scalability, auto-sharding, and out-of-the-box replication.

For our testbed, we installed and used version 4.4.2. Mongo uses text indexes to support text search queries and to perform full-text search in a document. Like most non-relational data stores, Mongo stores data in collections instead of tables; an index in Mongo is created after creating the database and the respective collections. Similarly to the PostgreSQL case, we created one collection for each batch of the datasets. Subsequently, we created the appropriate text indexes to facilitate full-text search; Mongo text indexes come with a limitation of only one 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.

ElasticSearch A search engine, based on the Apache Lucene library; it provides a distributed multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. ElasticSearch emphasises scalability and resilience via the distribution of the stored data and their respective indexes. It also, supports real-time GET requests, which makes it suitable as a NoSQL datastore although it lacks distributed transactions.

For our testbed, we installed and used version 7.10.0 (together with Kibana 7.10.0 as part of the ELK stack). In ElasticSearch, each index is created when each dataset is imported on the Elasticsearch server. Indexes are used to store the documents in dedicated data structures corresponding to the data types of their respective fields. The process of defining how each document and the corresponding fields are stored and indexed is called *mapping*. By using the *get-mapping* API, we can view the mappings for the indexes that were created after data insertions.

[3] https://www.kaggle.com/datasets/luisfredgs/yelp-reviews-csv

As previously, we created indexes for the different batches of our datasets. For further editing of indexes and mappings, we used the Kibana user interface.

Apache Solr An open-source, enterprise search platform built on top of the Apache Lucene library. All of Lucene's search capabilities are provided to Solr through HTTP requests. Its main features include full-text and faceted search, real-time indexing, hit highlighting, advanced analysis/tokenization, dynamic clustering, and database integration.

For our testbed, we installed and used version 8.6.3. To setup Solr for our experiments, we created Solr *cores*; a Solr core runs an instance of a Lucene index that contains all the Solr configuration files required to use it. We created as many cores as the number of batches that the datasets were split in. To index data under the created cores, we mapped the respective text fields from the datasets and specified the type for each field as *text_general*. The field type is used to inform Solr on how to interpret and query it; in our case we define the fields of type *text* that are appropriate for performing full-text search.

2.3 Technical details & specifications

In this section, we discuss the technical specifications and the requirements for each of the systems above.

All system versions were the latest stable ones at the time of the initialisation of the performance comparison. Programming operations and query executions were implemented in Python and a commodity system (with Core i3 2GHz processor, 6GB RAM) was used to run the experiments.

A number of Python libraries and modules were also used to achieve the connections 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 *pymongo*, a native Python driver for Mongo. To establish a connection with the ElasticSearch server in order to index and search for data, we used a Python client for ElasticSearch from the *elasticsearch* package, while for HTTP connections with Apache Solr and REST API tasks we resorted to the *urllib.requests* module. For operations regarding data preprocessing such as combining, merging and handling missing data, we utilised the Python *pandas* and *nltk* libraries.

3 Experimental Evaluation

In this section, we present the experimental evaluation of our study. Initially, we discuss the query generation process and subsequently present the experimental comparison of the systems in terms of query execution time (across various parameters), data insertion time, and memory utilisation.

3.1 Query design and generation

Since no database of queries was available to us, we initially designed and generated appropriate queries, based on the methodology in (Zervakis et al., 2017; Tryfonopoulos et al., 2009; Tryfonopoulos, 2018), that varied in type (phrase matching, wildcard search, boolean queries) and selectivity (high, medium low). To do so, we extracted the appropriate keywords from the datasets and used them to form different groups of queries. Initially, we extracted the most frequent words from each dataset, after performing punctuation removal and case folding. To avoid utilising different queries for the different batches of documents, we selected terms that were *frequent* across all different batches of data. Similarly, we selected terms that were *infrequent* across all batches of data. Subsequently, we composed multi-word terms by extracting keyword pairs present in the datasets.

The frequency of occurrence of single- and multi-word terms has an effect on the selectivity of the created query. In our setup, we define three levels of selectivity, i.e. *high*, *medium*, and *low*; a highly selective query executed over any of the datasets would result in returning (on average) a low number of relevant records; similarly a query of low selectivity would result in returning a high number of relevant records. To create queries of varying selectivity, we used combinations of single- and multi-word terms; we could foresee their selectivity based on their frequency of occurrence. For example, highly selective queries were constructed by infrequent single- and multi-word terms randomly chosen from a specifically created group of infrequent terms for each dataset.

Query selectivity (σ) is used as a parameter in the query generation process of the various query types (Q). Specifically, for *exact phrase matching*, *wildcard search*, and *boolean search*, the corresponding selectivity values used were the following: (a) *high* – 0.02%, 0.25%, 0.08%, (b) *medium* – 0.2%, 2.5%, 0.8%, and (c) *low* – 2%, 25%, 8%.

Exact phrase matching Exact phrase matching returns database results that match a particular phrase; an exact phrase query is represented as a sequence of terms enclosed in double quotation marks.

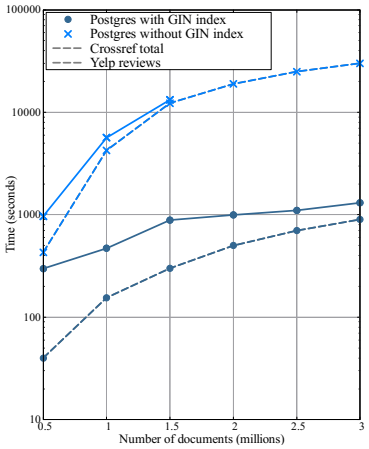

Figure 1: Exact phrase matching – PostgreSQL with/without using GIN index

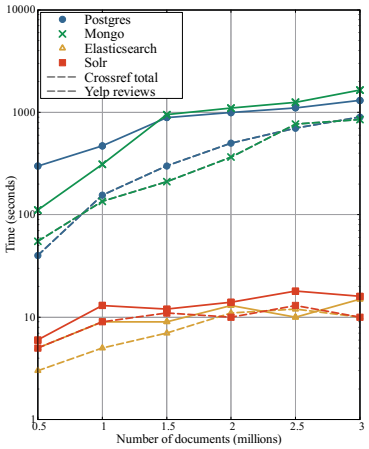

Figure 2: Exact phrase matching

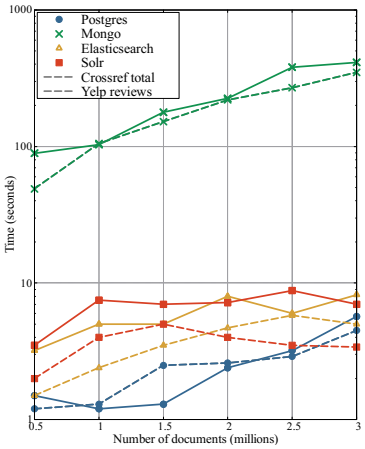

Figure 3: Wildcard search

To issue an exact phrase matching in PostgreSQL, we utilise the *phraseto_tsquery()* function within the *tsvector* and *tsquery* text search operators, after having created a new *document* column. To issue the same query in MongoDB, we use the *$text* operator inside the *find()* method, right after the *abstract* column has been previously indexed as a new text column, in order to perform any full-text operations in MongoDB. Finally, Apache Solr and ElasticSearch execute the exact phrase matching query using the Lucene parser according to the Query DSL syntax.

Wildcard search Wildcard search is used as an advanced search feature to manually stem results or when in doubt of the correct writeup; an asterisk (*) is utilised as the wildcard symbol to represent the existence (or not) of none, one or more other characters.

In wildcard search, we resort to the *to_tsquery()* function to execute the query in PostgreSQL, while in MongoDB we utilise the *$regex* operator that provides us with regular expression capabilities for matching string patterns in queries. To issue a wildcard search query in ElasticSearch, we use the *query_string*, while in Apache Solr we execute the query with the use of the Lucene parser using the Query DSL syntax.

Boolean search In boolean search we have generated only conjunctive queries since other operations like disjunction and negation are just different types of constraints that end up to different levels of selectivity. Executing conjunctive queries is a straightforward operation/function readily supported in all systems under comparison.

3.2 Querying time

In this section, we present a series of experiments that compare the different systems (i.e., PostgreSQL, MongoDB, ElasticSearch, and Solr) in terms of query time under various query types and various levels of query selectivity. The time measured is wall-clock time and the results of each experiment are averaged over five runs to eliminate measurements fluctuations.

Exact phrase matching In this section, we present the findings that derived from the experiments, comparing the systems' query execution time for exact phrase matching. Please note that all diagrams are displayed in logarithmic scale.

Figure 1 presents the execution time for PostgreSQL with respect to the use or not of *GIN index*. Notice the speedup in execution time when utilising the index and also that, PostgreSQL crashes after 1.5M documents when not using it. We thus, used the GIN index for our setup, as also recommended by Postgres for full-text searching.

In Figure 2, the results comparing the different systems with respect to the database size for each one of the datasets are presented. As the database size gets larger the execution time for each system increases as expected. ElasticSearch and Solr though, seem to achieve the best performance compared to MongoDB and PostgreSQL, which both need significantly more time to execute the issued queries. Notice however that, MongoDB and PostgreSQL have in this set of experiments similar performance, an observation that does not hold for the rest of the query types examined (presented in later sections). In addition, for each system, the differences in the runtimes measured for the different datasets are due to the high frequency of occurrence of terms in the *Yelp* dataset as opposed to the lower frequency of occurrence in the *CR* dataset. Note that the results in Figure 2 concerns highly selective queries – the detailed experiments regarding *selectivity* are presented later in the section.

			PostgreSQL		MongoDB		ElasticSearch		Solr	
$\mathcal{Q}$	$\mathcal{D}$	σ	CR	Yelp	CR	Yelp	CR	Yelp	CR	Yelp
Exact Phrase Matching	1M	Low	759	320	492	178	12,2	6,4	16	10,5
		Med	550	264	341	141	10,5	5,2	14	11,4
		High	470	155	310	134	9,2	5,1	13,3	9,3
	2M	Low	1641	1196	1585	671	17,3	11,3	16,8	12,5
		Med	1350	751	1378	465	16,4	10,8	16,3	12,1
		High	996	500	1100	365	13,2	11	14,1	10,4
	3M	Low	2766	2257	2550	1050	18,6	11,5	19,1	12
		Med	2150	1013	2150	960	18,1	11	18,5	11,2
		High	1308	902	1650	849	15,2	10,5	16,5	10
Boolean Search	1M	Low	454	195	581	200	7	3	10,8	7,3
		Med	4,3	2,2	498	139	6	3	7,8	6,6
		High	1,9	3,2	408	110	5,2	2,7	5,4	5
	2M	Low	3379	739	1359	548	10,3	6	11,8	8,4
		Med	7,2	4,6	1267	425	9,2	4,8	8,4	6,7
		High	2,9	4	1060	371	6,6	5,5	5,5	5,7
	3M	Low	10049	1948	4959	3659	11,1	6,4	11,5	8,5
		Med	10,6	7,4	1720	1221	11	5,6	7,7	6,7
		High	4,2	3,7	1398	975	8,4	5,2	7	5,8

Table 2: Performance for different selectivity levels across datasets and query types (in seconds)

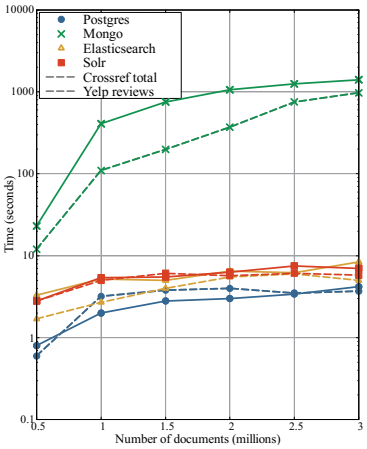

Figure 4: Boolean search

Wildcard search The second set of experiments measures the time needed to query the data stores using wildcard search and the results are shown in Figure 3. As we observe, all systems perform much better in terms of query time when compared to exact phrase matching, with PostgreSQL, ElasticSearch and Solr to process the query load in less than 10 seconds for all datasets. The performance of PostgreSQL, being similar to that of the other systems, is attributed to the use of the *GIN* index that supports the querying of a single term inside a text field. Also notice that, although PostgreSQL achieves the best performance when less than 2.5M documents, it then converges with the performance of the rest of the systems. This demonstrates the expected result of better scalability of non-relational data stores over the relational ones as data grows in size. Finally, MongoDB performs significantly better in this setup, as it achieves almost 50% of the time needed to match the same amount of exact phrase queries (Figure 2).

Boolean search In this set of experiments, we evaluate the query execution time for boolean search performed as conjunctions of keywords. Figure 4 presents the results for the different systems with respect to the database size for each one of the examined datasets. Both ElasticSearch and Solr are fast and are able to execute the boolean searches in under 10 seconds for both datasets. Once again, MongoDB performs significantly slower than its alternatives, whereas PostgreSQL performs fastest among the examined systems. Similarly to our previous findings, this can be attributed to the use of the *GIN* index that is designed mainly for conjunctive queries and provides an advantage over the rest of the systems.

As it has also been observed in the previous sections, in the case of boolean search, query execution times for the *Yelp* dataset are lower compared to those for the *CR* dataset, due to the high frequency of occurrence of terms in the *Yelp* dataset.

Selectivity In this section, we present the evaluation of the systems related to the query selectivity. In the top half of Table 2, we present the results derived after comparing exact phrase matching queries in terms of *low*, *medium* and *high* selectivity with respect to the database size for each dataset, while the lower half of the table concerns boolean search. The results for wildcard search are omitted due to space constraints.

The results illustrate that for all systems the runtimes increase proportionally with the size of the database, as also observed in Figures 2 and 4. Notice though in Table 2 that, ElasticSearch and Solr seem to be relatively agnostic to increasing/decreasing selectivity for exact phase matching (upper part of tha table), while there is a slight difference in their performance when varying selectivity in all database sizes for boolean search (lower part of the table). In addition, MongoDB slightly improves its performance in terms of execution time when increasing the selectivity of the queries executed in all database sizes for both datasets. The setup used though, does not affect the performance of all the systems with the same way; PostgreSQL seem to be greatly affected both by the selectivity and the query type used.

More specifically, PostgreSQL performs faster with queries of high selectivity. Notice in Table 2 that, PostgreSQL needs one order of magnitude less query execution time for high selectivity queries when performing boolean search (lower part of the table) in contrast to low selectivity ones. Also, when compared to MongoDB, for exact phase matching (upper part of tha table) they have similar performance when using the *CR* dataset, while MongoDB outperforms PostgreSQL when using the *Yelp* dataset. However, PostgreSQL performs significantly faster (i.e., one order of magnitude) than MongoDB for boolean search and medium/low selectivity.

		PostgreSQL		MongoDB		ElasticSearch		Solr	
$\mathcal{D}$	$\mathcal{Q}$	CR	Yelp	CR	Yelp	CR	Yelp	CR	Yelp
1M	EPM	470	195	581	200	7	3	10,8	7,3
	WS	1,2	2,2	498	139	6	3	7,8	6,6
	BS	1,9	3,2	408	110	5,2	2,7	5,4	5
2M	EPM	996	500	1100	365	13,2	11	14,1	10,4
	WS	2,4	2,6	225	219	8	4,7	7,2	4
	BS	2,9	4	1060	371	6,6	5,5	6,3	5,7
3M	EPM	1308	902	1650	849	15,2	10,5	16,5	10
	WS	5,7	4,5	412	349	8,2	5	7	3,4
	BS	4,2	3,7	1398	975	8,4	5,2	7	5,8

Table 3: Performance for different query types and high selectivity (in seconds)

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 performance in querying documents for the full-text search operations of exact phrase matching (EPM), wildcard (WS) and boolean search (BS). All tests have been carried out by applying queries of high selectivity.

In Table 3, we can see the results for each system. Notice that, exact phrase matching needs significantly more time to be executed by all systems in most of the cases, while on the contrary, wildcard and boolean search demand less time on average. PostgreSQL needs more than 1000 seconds to query 3M records for the *CR* dataset, while it needs less than 10 seconds for the rest of the operations; a similar behaviour can also be seen for the *Yelp* dataset. ElasticSearch and Solr also need more time to operate with exact phrase matching queries, but still significantly less compared to PostgreSQL (in all datasets) and with minor time differences compared to the other query operations. Finally, MongoDB performs better with wildcard search in comparison to exact phrase and boolean search.

In summary, we conclude that the majority of systems face difficulties in executing exact phrase queries in full-text search, especially with large amounts of data, while on the other hand, wildcard and boolean search are operations faster to execute in most of the examined systems.

3.3 Insertion time & memory usage

In this section, we present the results concerning the time needed to index the documents into each system. In Figure 5, we present the results for all systems across datasets for data insertion time as the size of the database grows. A general observation is that as the database size increases, more time is needed to index new documents for all examined systems. It is worth noting though, that Solr needs less time to index new documents compared to ElasticSearch, whereas PostgreSQL has the fastest indexing times across all examined data stores, with MongoDB being the slowest

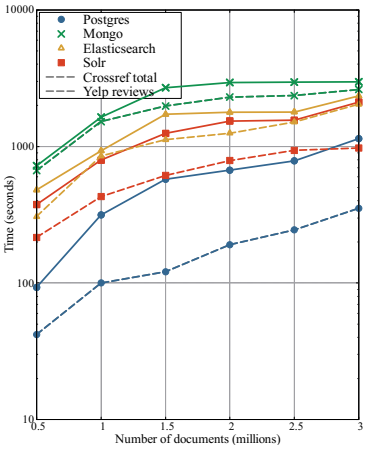

Figure 5: Data insertion time

system in data insertion/indexing.

We have also executed a set of experiments to investigate memory requirements for each system. Solr and ElasticSearch have high memory requirements for data indexing, while MongoDB utilises significantly less memory. Also, PostgreSQL typically needs less memory than the rest of the systems, apart from Solr, especially for smaller database sizes. Regarding the querying process, we conclude that Solr and ElasticSearch consume significantly less memory to execute a set of given queries compared to MongoDB and PostgreSQL. The detailed results are omitted due to space constraints.

3.4 Summary of results

Our extensive experimentation has demonstrated that ElasticSearch and Solr have responded in the experiments with balanced time measurements in proportion to the dataset size and without any fluctuations in most of the query types examined. PostgreSQL proved to perform much better with the use of the GIN index, competing non-relational systems and outperforming MongoDB in the wildcard and boolean search scenarios, but becomes significantly slower and has more memory requirements than its competitors for queries with low selectivity. Memory utilisation is also one of MongoDB drawbacks, especially in smaller database sizes, while performance is not as expected in most of the cases (especially in comparison to the NoSQL alternatives).

Our experimentation has also quantified how query selectivity affects the performance of the different systems. In the exact phrase matching scenario the systems under comparison responded without any serious fluctuation in query answering time, but in the boolean search scenarios significant fluctuations in query execution were observed. Additionally, most

of the examined systems present faster query execution times in the wildcard and boolean search scenarios. Finally, we demonstrated that PostgreSQL needs lower insertion/indexing time over its competitors.

Overall, we conclude that NoSQL and text data stores indeed provide a fast and trustworthy alternative for full-text search that is agnostic to the size of the database. However, MongoDB is the slowest and most sensitive to parameter and query setup among the NoSQL competitors, whereas PostgreSQL performs (surprisingly) well in some query scenarios (mainly wildcard and boolean search) and outperforms some of the NoSQL competitors especially for small and medium database sizes.

4 Future Work

In this work, we compared four popular data stores (PostgreSQL, MongoDB, ElasticSearch, Apache Solr) in full-text search scenarios and reported their performance efficiency across datasets of different sizes, various full-text search operators, and a wide variety of parameters. We also quantified the performance differences between the examined systems, highlighted the best options for each full-text search scenario and provided guidelines for selecting the appropriate system and parameter setup.

In the future, we plan to (i) expand our study with more systems (including CouchDB, Cassandra, Spinx and SQL Server), query types (including proximity, fuzzy and synonyms) and parameter combinations, (ii) incorporate a variety of textual content (web pages, social media posts, emails, publications, audio/video transcripts), and (iii) introduce a machine learning component that will enable the self-designing of text stores depending on the dataset and the workload in the spirit of (Chatterjee et al., 2021).

Acknowledgements

This work was supported in part by project ENIRISST+ under grant agreement No. MIS 5047041 from the General Secretary for ERDF & CF, under Operational Programme Competitiveness, Entrepreneurship and Innovation 2014-2020 (EPAnEK) of the Greek Ministry of Economy and Development (co-financed by Greece and the EU through the European Regional Development Fund).

REFERENCES

AnyTXT (2021). AnyTXT Searcher: Lucene vs Solr vs ElasticSearch, 2021. https://anytxt.net/how-to-choose-a-full-text-search-engine/.

Brewer, E. (2012). CAP twelve years later: How the rules have changed. *Computer*, 45(2).

Carvalho, I., Sá, F., and Bernardino, J. (2022). NoSQL Document Databases Assessment: Couchbase, CouchDB, and MongoDB. In *DATA*.

Čerešňák, R. and Kvet, M. (2019). Comparison of query performance in relational a non-relation databases. *TRPRO*, 40.

Chatterjee, S., Jagadeesan, M., Qin, W., and Idreos, S. (2021). Cosine: A cloud-cost optimized self-designing key-value storage engine. *VLDB Endowment*, 15(1).

Fraczek, K. and Plechawska-Wojcik, M. (2017). Comparative Analysis of Relational and Non-relational Databases in the Context of Performance in Web Applications. In *BDAS*.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., and Gosain, D. (2012). A survey and comparison of relational and non-relational database. *IJERT*, 1(6).

Li, Y. and Manoharan, S. (2013). A performance comparison of SQL and NoSQL databases. In *IEEE PACRIM*.

Lourenco, J. R., Cabral, B., Carreiro, P., Vieira, M., and Bernardino, J. (2015). Choosing the right NoSQL database for the job: a quality attribute evaluation. *J Big Data*, 2.

Lucidworks (2019). Full Text Search Engines vs. DBMS. https://lucidworks.com/post/full-text-search-engines-vs-dbms/.

McCreary, D. G. and Kelly, A. M. (2013). *Finding information with NoSQL search*. Manning.

Microsoft (2023). Full-text search. https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver15.

Mohamed, M., Altrafi, O., and Ismail, M. (2014). Relational vs. nosql databases: A survey. *IJCIT*, 3(3).

Nayak, A., Poriya, A., and Poojary, D. (2013). Type of NOSQL databases and its comparison with relational databases. *IJAIS*, 5(4).

Sahatqija, K., Ajdari, J., Zenuni, X., Raufi, B., and Ismaili, F. (2018). Comparison between relational and NOSQL databases. In *IEEE MIPRO*.

Schuler, K., Peterson, C., and Vincze, E. (2009). *Data Identification and Search Techniques*. Syngress.

Truica, C. O., Radulescu, F., Boicea, A., and Bucur, I. (2015). Performance evaluation for CRUD operations in asynchronously replicated document oriented database. In *CSCS*.

Tryfonopoulos, C. (2018). A Methodology for the Automatic Creation of Massive Continuous Query Datasets from Real-Life Corpora. In *ICAIT*.

Tryfonopoulos, C., Koubarakis, M., and Drougas, Y. (2009). Information Filtering and Query Indexing for an Information Retrieval Model. *ACM TOIS*, 27(2).

Zervakis, L., Tryfonopoulos, C., Skiadopoulos, S., and Koubarakis, M. (2017). Query Reorganisation Algorithms for Efficient Boolean Information Filtering. *IEEE TKDE*, 29(2).