Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Angra-DB: Indexing and Query Language

Fernando T. F. C. Nunes

Project presented as a partial requirement for
the conclusion of the Course of Computer Engineering

Supervisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Angra-DB: Indexing and Query Language

Fernando T. F. C. Nunes

Project presented as a partial requirement for
the conclusion of the Course of Computer Engineering

Prof. Dr. Rodrigo Bonifácio de Almeida (Supervisor)
CIC/UnB

Prof. Dr. Maristela Terto de Holanda
Universidade de Brasília

Jairo Fonseca
Lightbase

Prof. Dr. José Edil Guimarães de Medeiros
Coordinator of the Course of Computer Engineering

Brasília, 24 de january de 2019

Dedication

This work is dedicated to the ones that supported me throughout every challenge that I have faced along the way and shared all the small victories with me. My parents, Fernando and Alda, that provided me with undying support and love during all my life. My sister, Alexia, that was always there for me. My uncle Eudo and my aunt Auremeire, that helped me to adapt into a new city in every possible way. My friends, Fábio Marques, Bruno Rodrigues, Ismael Medeiros and Bruno Bergamaschi, that never refused to share a laughter with me, but also always helped me with the difficulties that were presented to us during this path.

Acknowledgements

Firstly, I would like to thank Professor Dr. Rodrigo Bonifácio de Almeida, who always trusted my work and has given all the guidance needed to develop this project. I would like also to express my thanks to the contributors of the Angra-DB project, which always provided me with useful discussions about the project's development. I would also like to thank Professor Guilherme Novaes Ramos, that first introduced me to academic research and whose advices were always remembered when writing this work.

Abstract

This work presents the process of development of a query language into a NoSQL documental database called Angra-DB. Angra-DB is an open source project that is developed at University of Brasilia and has requirements about its features extensibility, modularity and ease of use. The steps required in order to enable the query language building according to the project's principles involved the creation of an indexing feature for the database and the establishment of a common querying API, which would be used by query languages in order to perform queries into the stored documents. At the end of this implementation a query language inspired in JSON Query Language's specifications was built, allowing rich queries to be performed into the database. The testing of the proposed architeture showed that, while its performance during insertions of new documents lacked in comparison with other document-oriented databases such as CouchDB, the execution times of the queries obtained similar results. Moreover, the query language implementation focused more in its readability and ease of use, when compared to other query languages used in documental databases. Future works involves the further improvement of the architecture's performance, the improvement of the querying capabilities and the creation of new query languages, possibly to specific data domains.

Keywords: NoSQL, Query Language, Information Retrieval, Erlang

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rise of the internet made data a valuable resource. As we are all connected to the global network in various ways, a constant and huge amount of data is created about ourselves everyday[1] and if any economic or social activity wants to be successful it must know a way to extract information from the available data of its users or target audience [2].

Data organization for efficient retrieval is a long dated computer science problem that has been explored since the 1940s and faced many challenges as not only the amount of information increased, but the complexity of its content changed and applications developed different needs in data usage. Different types of databases were created to suit these needs in data processing, eventually leading to relational databases and then to NoSQL databases, as the Big Data problems (related to incredibly huge amounts of non-structured data) started to become relevant [3].

Non-relational databases, or NoSQLs (Not Only SQL, as opposed to the relational databases), are databases that handle data that does not fit the relational model i.e. are not organized primarily in tables. They are designed to store and process a large quantity of data, generally in a distributed computing system [4]. They also use other languages than SQL to query and manipulate the stored data [4]. These types of databases emerged alongside big companies such as Google and Facebook, which started to struggle with the amount of data generated by its users.

Since 2016, an open-source project started at University of Brasilia where students started to develop a NoSQL database called Angra-DB. This project aimed to tackle the different challenges that building a non-relational database pose. This database was built using primarily Erlang, a functional language specialized for building distributed applications using lightweight processes.

To further contribute to this project, this work aims to design and implement a search

mechanisms for AngraDB, including a new indexing module and a query language, following common techniques used for information retrieval.

1.1 Problem Statement

In order to enable the Angra-DB project to fit a diverse field of applications, a query language, a crucial feature present in many databases, needed to be developed. However, as one of the project's principles, each new feature should be a module as loosely coupled as possible, i.e. the modules should be able to be developed independently and should be allowed to be replaced to fit specific needs. That said, a specific architecture needed to be proposed to cope with these demands.

1.2 Objective

The general goal of this work is to design and implement a search engine and a query language for Angra-DB. In order to achieve that, we must address some specific objectives, including:

- Design, implement and integrate an indexing module into Angra-DB's project, following the project's principles of modularity and extensibility.
- Specify, design, and implement an initial query language for Angra-DB. Due to the extensibility requirements, it is expected that other query languages could be also integrated lately.
- Conduct an empirical study to (a) understand the performance implications of using the search engine to execute queries over Angra-DB and (b) compare the initial query language to existing alternatives with respect to expressiveness.

1.3 Methodology

To achieve the objectives above, some changes were needed in the project's organization and new modules needed to be implemented. As Angra-DB is a NoSQL database, the first step in achieving the desired goals was to research about common query and indexing techniques used while processing large quantities of data. With this knowledge, we could choose an approach in order to develop an indexing feature for the Angra-DB project, responsible for the creation and maintenance of the indexes used to access the stored data. To integrate this new module into the project, we reviewed some design decisions of Angra-DB.

To obtain a better understanding of the common use cases of document-oriented databases queries, we reviewed the literature on query languages and query interfaces, giving special emphasis on works detailing how to process JSON documents. This review enabled the assessment of the desired features of the query language and the query capabilities that was expected from the database. Considering the selected features and the query functionalities provided by the indexing module, we implemented an interface to perform queries on the database's documents. This interface supplies the generic querying resources that was used by the query languages. In order to develop the language containing the desired features, we also studied language building tools and chose a parser and a lexical generators. Using these tools, we designed and implemented a query language according to the features and using the query interface provided.

Based on the implementation of the the indexing and query language modules, we conducted empirical studies to assess the limitations and strengths of the developed architecture. The first study compares the performances of different Angra-DB indexing configurations. The second compares the performance of the indexing module and the query execution of Angra-DB and another document-oriented database. Lastly, we compared our query language with existing ones supported by two databases, evaluating the programming language characteristics commonly used in the literature.

Chapter 2

Document-oriented databases and Angra-DB

As the popularity of NoSQL databases increased, multiple types of databases emerged, each of them specialized in different kinds of data [1][4]. This means that NoSQL databases can refer to a great variety of implementations with very different characteristics, unlike the relational databases, which all share a lot of features including the query language [5][6]. Some examples of NoSQL types are:

1. Key-Value Stores: a simple model in which schemaless data can be stored and be retrieved by a key identifier through a simplistic API. Some famous key-value NoSQL databases include DynamoDB, RIAK and LevelDB.
2. Column-Oriented: this type of database models the data as a record consisting of multiple columns in a row. It differs from the relational databases as the data is not split across multiple tables, instead, each key is associated with a set of columns. Google's Big Table is an example of this kind of database.
3. Graph Databases: stores the data in a graph model in which the data is modeled as the nodes and the relationships as the edges. This type of database focuses on the connection between the data.
4. Document-Oriented: each data record is stored in a standard data format such as JSON, XML, etc. and can be referenced by a key. CouchDB and MongoDB are examples of document-oriented databases.

Even though there is a great variation of applications of each of these NoSQL types, there are some characteristics that most of them try to accomplish. To be capable of storing and processing huge amounts of data, NoSQL databases are usually distributed systems, providing horizontal scalability [7][8][**?**]. Also, Non-relational databases do not

ensure the ACID (atomicity, consistency, isolation and durability) principle, instead, they ensure a different set of properties called BASE (basically available, Soft state and eventual consistency)[4][9]. In order to do so, they choose to ensure two of the three guarantees possible according to the CAP theorem:

- Consistency: the same data is available across all the nodes
- Availability: every read/write operation results in a non-error result
- Partition tolerance: the system continues to operate even in case o physical network partitioning

The consistency in NoSQLs is eventual, meaning that at any given moment, the nodes might not contain the same data, but given some amount of time, the data is replicated[8]. Furthermore, as non-relational databases aim to be basically available, i.e the system continues to operate even if some nodes fail, the partition tolerance and availability are ensured to some degree [1].

2.1 Document-Oriented Databases

Document-oriented databases (DODBS) are NoSQL databases designed to handle a large quantity of semi-structured data. The concept of document is central to the design of a DODBS and its definition differs slightly from each implementatio. That said, even though the definition may vary, all DODBS assume that the document is encoded in some standard data format. For instance CouchDB assumes data is saved in JSON format while MongoDB stores data records encoded as BSON [10].

Taking this into consideration, documental databases can be thought as a collection of key-value collections, as each document itself can be seen as a key-value pair collection. This allows for an additional layer of indexing, because not only the documents can be indexed (referred by a key), but its contents too (indexing the documents fields) [10]. Also, each saved document is associated with a document key, which can be used to later retrieve the document [1].

As they are designed to handle Big Data problems, document-oriented databases implementation are distributed systems that stores its data replicated through multiple servers [4]. This means they usually ensure high availability to its users, as the data can be accessed through different nodes on the network.

CouchDB and MongoDB are two of the most used document-oriented databases [10] and were used as guidance during the course of this work. That said, the next two subsections will contain a brief review about their architecture.

2.1.1 MongoDB

MongoDB is an open-source documental NoSQL database built upon collections and documents. In MongoDB, documents are grouped into collections based on their structure. The document structure is a description or a schema of the content of the document. MongoDB is schemaless by default, but provides a way to enforce a schema validation [7].

As stated before, MongoDB stores its documents in BSON format, a binary representation of a JSON document. BSON is a document storage and data interchange format that supports extensions which enables data representations that are not part of JSON format [10]. This data format has three main goals:

1. Be lightweight, minimizing the overhead
2. Be transportable, being easily transferred through systems
3. Be efficient, enabling quick encoding and decoding

The data modeling in MongoDB relies in storing the application data using the minimal number of documents possible, without the need to split the data across multiple relationed subjects [10]. MongoDB also provides different replication strategies based in replica sets, which are clusters of MongoDB servers that implement replication, using sharding, and automated failover [7]. In addition to the replication, MongoDB provides a load balancing feature in order to divide the amount of work demanded from each server running the replica set [7].

Moreover, MongoDB enables querying the data by its fields using range and regular expression queries [10]. Also, it is possible to query elements in nested arrays in documents. The queries are written using mainly Javascript, even though interpreters are written for a multitude of programming languages.

2.1.2 CouchDB

CouchDB is an open-source document-oriented database that focus mainly in ease of use [10]. Data is also organized in collections of documents. In CouchDB, documents are independent, which means that each document stores its own data and schema. Furthermore, CouchDB implements a multi-version concurrency control in order to avoid database locking during write operations[10].

The data format of the documents in CouchDB is JSON. Each node in a CouchDB database stores a complete copy of the documents of a collection, and communication with the CouchDB database can be made with any node. The distributed architecture of CouchDB provides bidirectional synchronization, meaning that the nodes can go offline and resynchronize after reconnection [10].

CouchDB provides a REST API to query and manipulate de database and uses mainly Javascript as its query language. It also provides a map reduce architecture which can be used through the views feature[1] [10].

2.2 Angra-DB

The Angra-DB project started in 2016 in University of Brasilia. It is a document-oriented database inspired in CouchDB and written primarily in Erlang. It uses Erlang language features to provide a distributed and loosely coupled architecture. Appendix A contains a brief explanation of core Erlang features that will be mentioned throughout this work.

In the next subsections, the current Angra-DB architecture will be shown, giving some context about the project state prior to this work.

2.2.1 Database Concepts

Angra-DB has some core concepts that guide the usage of the data stored in its database. These concepts are heavily inspired by the ones used in other document-oriented databases mentioned before, such as MongoDB and CouchDB.

The notion of a collection in other documental databases here is called a database, or simply *db*. An Angra-DB *db* can be viewed as a list of *documents* that can be addressed by a unique identifier called key. The objective of a *db* is to aggregate documents that are similar semantically. For example, if we have a collection of documents that describes the same real-world subject, such as a student, then we should store then in the same *db*.

The document concept is the same as CouchDB's concepts in the sense that it is just as real-world documents. And much like CouchDB, the documents are stored in JSON format, as it is the current standard data transferring and encoding type in the web.

2.2.2 Project Architecture

Following Erlang design patterns, the Angra-DB project uses an architecture of supervision tree and is organized around an application called *adb_core* which provides the core functionality of the Angra-DB database.

Loose coupling is a major principle in the Angra-DB project. That said, every new feature added to the project must be flexible enough to support multiple implementations. To do so, the behaviour pattern of Erlang is of utmost importance since it lets generic and well behavioured code support multiple implementations, leading to extensible and changeable features. Moreover, another way to provide extensibility is by letting other

[1] `https://docs.couchdb.org/en/stable/ddocs/views/intro.html`

applications interact with the database through an easily accessible interface. In that regard, Angra-DB provides a TCP server capability which aims to provide an easy to use interface that can be managed by virtually any programming language. The architecture of Angra-DB is shown in Figure 2.1.

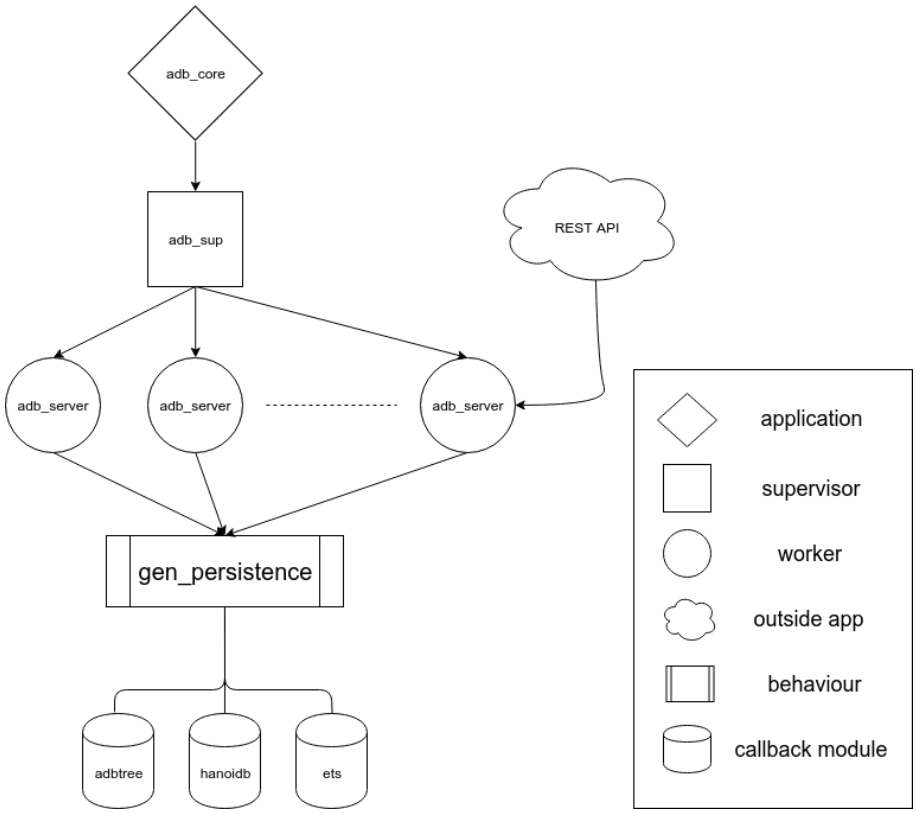

Figure 2.1: The Angra-DB core architecture.

As explained in the previous section, structuring the project in an application behaviour enables the project to use capabilities such as dependency and configuration management. For instance a default Erlang logging app called *lager* is used in order to properly log the database actions for debugging purposes. Also its is possible to set configuration values in a separate app configuration file in a way that these configurations are available in the *adb_core* callback module.

The *adb_core* is an application responsible for the startup the supervision tree with the root supervisor being the *adb_sup* which has the restarting strategy *simple_one_for_one* that enables for simple dynamic instantiation of child processes, with the drawback of only allowing the children to be of the same type of worker.

Next, the *adb_server* is a worker that handles TCP connections to Angra-DB. It parses TCP messages into a set of Angra-DB commands that are able to perform operations like

creating databases and CRUD functions. Here is a list of the Angra-DB commands prior to this work:

create_db creates a new database

connect connects to an existing database. In order for any operation to be performed on a database, a connect command must be issued before

save saves a new document to the connected database. The document key is automatically created and returned if this operation succeeds

save_key saves a new document to the connected database, specifying which document key should be used.

lookup gets the document by key in the connected database

update updates the document with a given key in the connected database

delete deletes the document with a given key in the connected database

Each *adb_server* is responsible for the management of a different TCP connection, maintaining the state of the connected database and, more importantly accessing the storage for read/write operations.

The first extensible point of Angra-DB is its persistence method, i.e how the data will be stored and retrieved from the secondary storage device. This is accomplished through the use of a behaviour called *gen_persistence* as shown in the diagram of Figure 2.1. This behaviour is called in the *adb_server* module and expects that any implementation should contain the functions to perform the CRUD operations and the creation of databases.

As of now, there are three implementations of the *gen_persistence* behaviour. The first uses the ETS module, an Erlang module that implements in-memory hashtables and thus is a memory only database. The second uses a key/value indexed storage engine called HanoiDB. The third and last one is the default *gen_persistence* implementation and is an Angra-DB owned B+Tree implementation heavily inspired by CouchDB's implementation.

Finally, there is another Erlang application implementation that uses the TCP interface in order to provide a REST API service. It has a structure similar to the *adb_core* but receives HTTP requests and contacts the *adb_core* to perform the read/write operations in the databases.

It can be noticed that the API provided above is not far from an API provided by a purely Key-Value Storage NoSQL. Also, the shown architecture lacks mechanisms and

features to manage database access such as concurrency control, permissioning, distribution and more complex data interactions such as queries. This work specifically tackles the last issue. However, other works are in progress in parallel with this one and are trying to provide solutions for the lacking features of Angra-DB.

Chapter 3

Information Retrieval

According to a pioneer in the field, Gerard Salton, information retrieval is a field concerned with the structure, analysis, organization, storage, searching and retrieval of information. Even with the evolution of this field, this definition can still be applied accurately [11].

As the internet has gained a wide usage across different types of activities in the modern society, a huge amount of information is generated in a daily basis and is stored in the network. That said, searching the Web has become a common activity and to make it possible, the information retrieval field has flourished [3].

Since its beginning, information retrieval has been mainly applied to text documents, be they Web pages, scholarly papers, books or news stories [3]. All of these documents have some amount of basic structures, which could be organized as fields in a document. For instance, in the case of news stories, every document contains at least a headline and the story itself, but they can also contain images and have some other predefined structure in its content, like an interview in which there is a sequence of questions and responses [11].

These data formats differ from conventional relational database records because each field is mainly composed of natural language text. To make a comparison, if we would store these documents in a SQL like database, each field would be translated to a column and the database would be more efficiently queried by searching the whole content of a given column value. However, if we take for example a web search engine, the queries submitted by the application's end user will be composed of pieces of information about a Web Page content, not the whole text written in the page [11].

That said, text searching for information is a complex operation in the provided scenarios, since both the document's fields and the submitted query are natural language text. Therefore, one of the main goals of information retrieval is to understand the process used by humans when comparing natural language and convert this process into algorithms to accurately perform this comparison[11].

Moreover, *ad hoc* searches, i.e searches based on a human query, are not the only application of information retrieval. Classification, filtering and question answering [11] can also be included in tasks that information retrieval are applied and some explanation about then is provided below:

- Classification is about labeling texts automatically in order to group then in categories.
- Filtering is a task that monitors documents in search of content related to a specific field of interest, tracking these contents and generating alerts to a user.
- Question answering is a kind of text search in which the submitted query is in the format of a question, thus the correct output is to find an answer in the content of the documents.

Also, it is important to address that, as multiple types of media can be found across the Internet, information retrieval does not solely handles natural language text as it is also applied in tasks involving audio, images and music. Nonetheless, said applications are currently based in text descriptions of the content, even though some studies are developing methods for direct comparisons.

3.1 Key Concepts

Since the beginning of the information retrieval studies, some key issues have been recurrent and can be considered as the core aspects of the field.

Relevance is a concept that is used to describe the documents retrieved from a user query. A relevant document is a document that contains information the user was looking for when the search was submitted. One of the big issues of evaluating the relevance is that it does not depend solely on the document content, but also on other factors such as the age of the document and the previous knowledge of the user about some of the document contents [11].

That said, another important aspect of the query results is the difference between *topical relevance* and *user relevance*[11]. A document is topically relevant if it correctly matches the topic the query was about, while it is user relevant if it matches the desired needs of the person that submitted the query. For instance, if the user submits a query for a paper about "machine-learning state-of-the-art" and the responses are 20 years old or they already seen the method proposed by an article, then these results are not relevant for the user, even though they are topically relevant [11].

The concepts above are the central aspect of information retrieval systems. When retrieving a document list about a given query, information retrieval algorithms has the

goal of ranking the documents based on user relevance. This means that a good ranking algorithm leads to documents that are of interest of the user on its top results. More often than not, these ranking algorithms use statistical data about the documents and the collection as a whole in order to improve its results [11].

3.2 Architecture of an Information Retrieval System

An information retrieval system has mainly two goals to ensure; quality of the query results and efficiency in the processing of the documents [11]:

- Quality: it is desirable to return the most user relevant available documents
- Efficiency: the results should be returned as fast as possible.

Usually, these two properties are antagonic, which means that quality often is reduced in order increase the system's efficiency, and vice versa [11]. That said, the architecture of the system has to balance these two properties in order to provide the desirable results, employing data structures that are more suitable to fast retrievals while storing the statistical data of the stored documents that are needed by their ranking algorithms [11]. In that regard, most information retrieval systems revolves around the creation of indexes containing the documents information for later retrieval when querying the documents.

3.2.1 Building the indexes

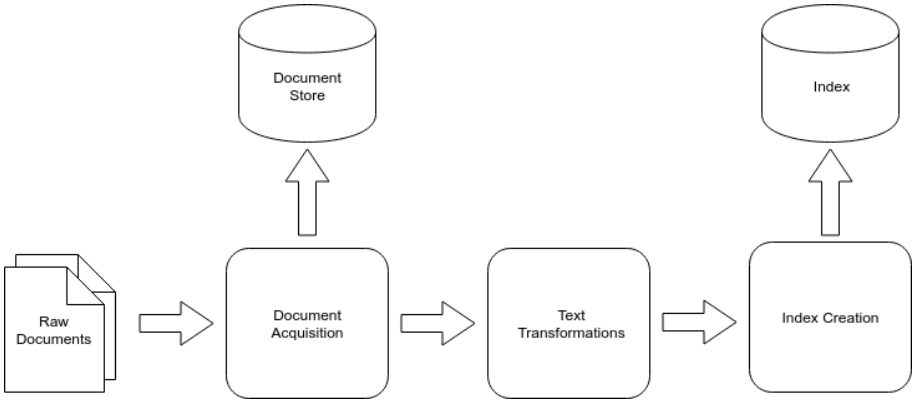

Figure 3.1: Basic information retrieval architecture for building a searching index.

As we can see in Figure 3.1, it all starts with a collection of raw documents that are not in any particular storage. Then we have a process of acquisition of the documents in which the documents are stored in the systems document storage for later retrieval. After

the documents are acquired, they are processed in order to transform its contents in minor parts, also removing unnecessary content for the next step, the index creation. This last part is responsible for the creation of the indexes that will be used while querying the documents [11].

Document Acquisition

Document acquisition is a problem that possess varied complexity across different applications of information retrieval [11]. If the documents are already available in a common format, i.e are encoded according to the same rules, or are already provided in an easily accessible way, as *CSV* files, the task of acquisition can be as simple as an import script that loads the contents in the storage[11].

However, if the data to be acquired can be of different types, is often changed or is continuously generated, then getting the data will increase its complexity greatly[11]. Taking the Web for example, it possess all of the aforementioned characteristics. Content is created every second, some of its content is changed in a regular basis and the content of a web page has a large array of formats. In these scenarios, *crawlers* are used to perform the document acquisition.

Crawlers are automatic web object retrievers, i.e, they have the basic goal of navigating through web pages in order to fetch their contents [12]. To crawlers, the web is a huge graph in which each node is a web page itself and the references between pages (the hyperlinks) are the edges. That said, crawlers basically traverse the graph represented by the network, storing the pages visited. The main goals of crawlers are:

- Discover new content, i.e, pages added to the web
- Monitor the changes in previously visited pages

As the amount of content is huge, the crawlers need to address issues such as scalability (being able to follow the rate at which the network expands) and network load (in order to not overload the network in the crawling process)[12].

As explained above, the crawlers address the issues about the size and mutability of the document collections, but they do not handle the fact that the content might be in a variety of data types. In this regard, the document acquisition task must also include a way to standardize the documents contents. This means that the content retrieved also is transformed into a fixed format that contains the document content plus its metadata before passing it to the next step in the index creation [11].

Text Transformation

In order to be able to generate the index with the information required to perform the querying processing, the contents of the documents must be parsed into lesser pieces of data, called index terms, that are what is actually stored in the indexes [11].

The parsing process is about recognizing the documents contents and classifying them into structures that are relevant to the querying process. It is the parser's job to distinguish structures such as titles, fields, and the words into comparable tokens. By comparable tokens, it is meant that they should be processed into representations where the equality means that they are actually related[11]. As example, for a given search engine, the words *online* and *on-line* might be considered as equal in order that queries containing one of them, would return results that contain either words.

The recognition of the documents structure in pursuit of metadata content should also be handled by the parser. This means that documents that have some structure, as XML and JSON files, should have their tags or fields recognized in order to access information that might be used later while performing queries[11]. For instance, words that are found in the title of a paper are probably more relevant to its contents than words that are found in a footnote, this type of inference might be used in some ranking algorithms.

Moreover, not everything of the documents content adds information that are required in the index, as some special words are on the text just to add to the structure of the sentences. For instance, words like *of, to* and *from* do not contribute to the meaning of most phrases, so they are usually not relevant while performing a query [11]. These words are called the stopwords and the process of removing them is called stopping. The stopping process is often done using a stopword list, which contains a list of common stopwords. However, generating such lists poses a challenge as the meaning of these words might have different relevance in different sentences, also if multiple languages are taken into consideration, the problem of defining the stopwords increases its complexity.

Besides the stopwords, another transformation is needed while parsing the documents contents as some words are created out of common *stems*. For example words like *fishes, fishing, fish* are all derived from the same *stem*. That said, the parser must also group these words as they probably refer to the same subject and therefore, might contribute to the topic relevance of the query results[11]. The process of grouping such words is called *stemming* and, just like stopping, have varied complexity across different languages and usually improves the results of the ranking algorithm.

Index Creation

The whole point of the processing made in the last two steps is to serve the content in the documents as input to the index creation component[11]. In this component, with the parsed document structure provided by the last step, the index is actually built.

The creation process do not only store the words and structures parsed in a data structure, but it also performs some pre-calculations in order to increase the performance of the querying process[11].

In that sense, one of the most important features of the index creation component is the computing of statistical data about the documents being indexed. For instance, an index creation component might store the number of occurrences of a word for each document, the position in which the terms occurred and the length of the documents. The statistics stored are defined by the needs of the ranking algorithm, meaning that any information that is needed in order to rank the documents relevance should be stored during the index creation, if possible[11].

Another relevant data that should be stored is the weighting of the words of a particular document. The *weight* of a word is a way to measure the importance of said word in the document and its calculation is dependant of the ranking algorithm. For instance, a common weight type that is used in older retrieval models is the *term frequency, inverse document frequency*, which combines the frequency of a term in a document with the frequency of the term in the entire collection for the computing of a term weight[11].

It is also important to note that the index creation component might also handle the distribution of the index. Given that the documents are stored in multiple computers, it is more efficient, i.e reduces the amount of time of query processing, if each computer indexes a subset of the entire collection[11]. That way, a query can be sent to multiple computers which will be dividing the querying process since they are handling different documents that are part of a larger set.

The indexing structure used is usually that of an inverted index. It is a common data structure that is specialized in fast retrieval and can be easily update with new content[11]. Section 3.3 will give a more thoroughly explanation about inverted indexes.

3.2.2 Querying

The index building process approached in the last subsection is meant to provide as much data about the documents as possible in order to reduce the amount of computing necessary while performing the queries[11].

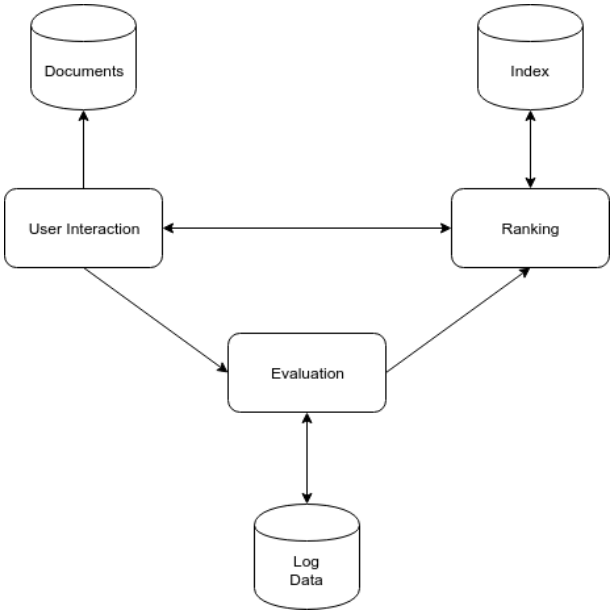

Figure 3.2: Basic information retrieval architecture for performing queries.

Figure 3.2 shows an architecture of the query processing in an information retrieval system. In this architecture, there are three core components that communicate with each other and are responsible for continuously improvements to the query search results.

The first is the *user interaction* component. It is the component that is responsible for the query text acquisition and for the first processing of the query. First, the query text is parsed into tokens just like the documents content. Also, the same operations performed with the documents must be used in the query text such as stemming and stopping. This should be done to simplify the comparisons that will be necessary when searching the indexes previously created[11]. Besides these basic transformations, some search engines might also further process the submitted query in an effort to provide better results, like correcting the spelling of a potential misspelled word or suggesting related queries that might help to narrow the query subject[11]. This component must also be tightly coupled with the document storage in order to provide the documents output that are returned by the other component[11].

The second component in the scheme is the *Ranking* component. This is the component where the retrieval model will be applied. This means that in this component the ranking algorithm will compare the submitted query with indexed documents and, using data stored in the indexes created with the steps explained in the last section, will create a rank of documents in order of relevance, according to the retrieval model established[11].

The last component, the *Evaluation* component is the component responsible for the improvement of the query results. This can be achieved by using the log data that is

generated by the user interaction component, which can be processed in order to infer interests of the end user [11]. For instance, using the data of the user preference of a previously searched term might provide some insight about the user's interest in subsequent queries. Again, how the usage data is manipulated when generating a query result is part of the job of the ranking algorithm.

3.3 Inverted Indexes

An *inverted index* is a data structure that is specialized in both fast retrieval and insertion. In a traditional index, there would be a list of documents where each document is associated with the words that it contains, that said, inverted indexes are called this way because of the fact that they are lists of words (or *terms*) and each word is associated with the documents in that they appear. This is an efficient way of storing the documents contents in order to find in which documents a specific word can be found[11].

As explained in the last section, an inverted index can be built from a list of tokens that indicates the occurrence of a word in a document. The list of tokens can be obtained from the list of documents with our text transformation component[11]. For illustration, given the following documents:

1. Bees are flying insects.
2. Wasps are insects that can sting and fly.
3. The sting of an ant can kill.

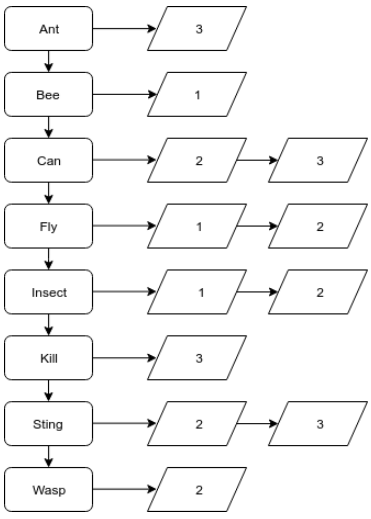

Figure 3.3: An example of the most basic inverted index.

We could generate the inverted index shown in Figure 3.3. Notice that some words were excluded with the stopping process and the word *flying* was contracted to its stem *fly*, in the stemming process.

The index shown above is an inverted index in its simplest form. Basically, a inverted index is a list of lists, where the first one (the vertical list in the image) contains all the words that were found in the document collection, after the process of stemming and stopping were performed. Each word contains a list of occurrences of this word on the document collection. For instance, the word *Bee* could only be found in the document number *1*, therefore, the list referenced by *Bee* contains just one item that refers to document *1*. Similarly, the word *Sting* occurs in both documents *2* and *3*, thus, the list contained in *Sting* refers to these documents. Each node that refers to a word occurrence is called *posting*.

It is also possible to notice that not only the words list is ordered alphabetically, but also the postings lists of each word are ordered according to the identifier of the documents. This is not a required property of inverted indexes, but if implemented, enables the merging of two inverted indexes using a merge algorithm (which has a complexity of $O(n)$) for both the words and postings lists[11]. This way it is possible, for example, to partition the document collection during the indexing process and merge the results in a single index in a later moment.

As inverted indexes are supposed to handle large collections of word occurrences, they are mainly stored in secondary storage such as HDs and SSDs. That said, the organization of an inverted index helps the fast retrieval and insertion in such storages[11]. Both the insert and retrieve operations must always find the list of postings that are associated with a given word. That can be done using hash tables that holds references to the location of the list of postings of a word in a file. This process is illustrated in Figure 3.4. This way it is possible to rapidly access the postings of a term for retrieval[11].

To insert new document data in the index, i.e. new word occurrences, the list of postings of some words must be updated. However, as the index is stored in a file, it is not possible to simply insert a new term or posting in the middle of the corresponding list. A strategy to deal with this problem is to create a smaller index with the new terms to be inserted in the final index and then merge the smaller index into the bigger one. As the merging operation of the smaller index would be costly, another strategy might be coupled with the previous one where multiple documents are stored in the smaller index until it reaches a given size, and only then, perform the merge[11]. This way it is also possible to handle deleted documents since its postings would not be included in the resulting merged index.

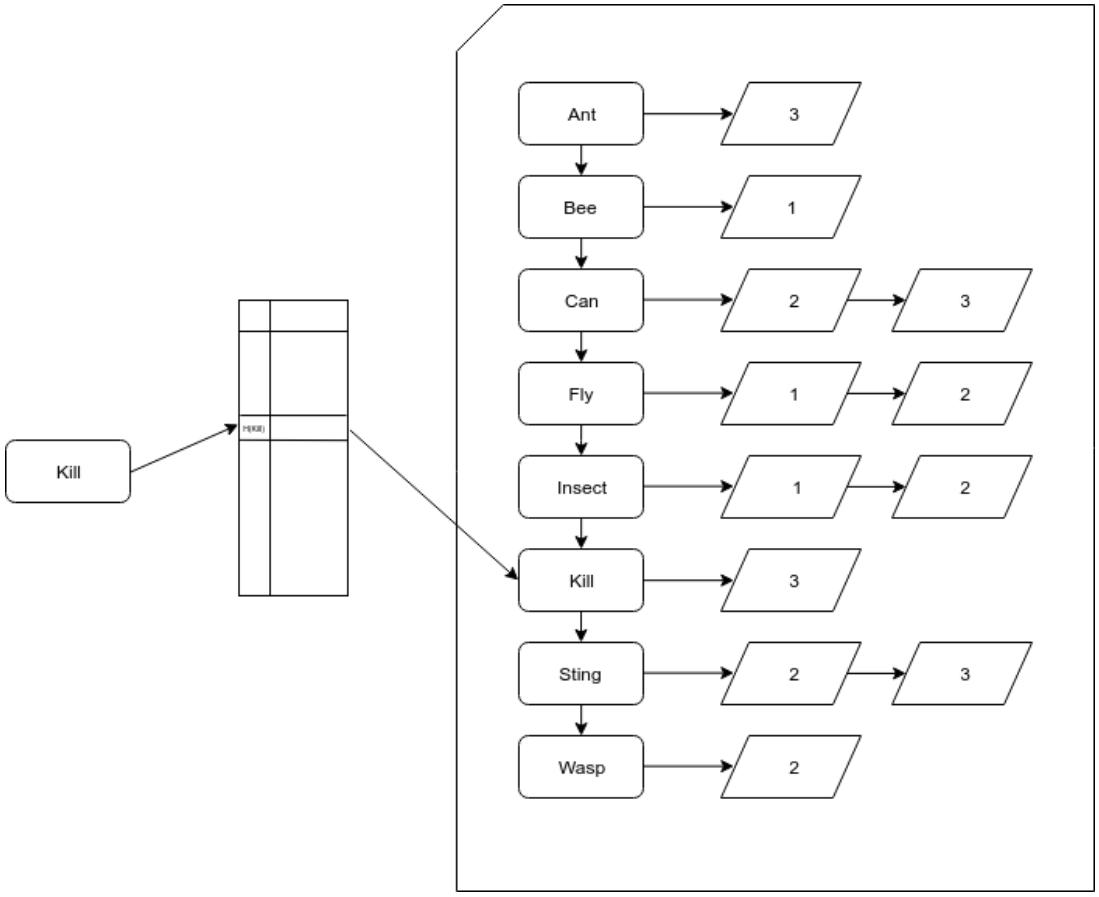

Figure 3.4: Accessing a word's posting list using hash tables.

3.3.1 Auxiliary Data

The inverted indexes presented so far contains a limited amount of information about the word occurrences. Nonetheless, as explained before, statistical data as well as structure data are important while performing queries. That said, it is also possible to store such metadata in the inverted index to be retrieved while querying for terms, therefore, reducing the computation needed while performing queries[11].

The statistical data that should be stored in the index depends on the ranking algorithm that will be used[11]. In that sense, if a ranking algorithm needs the number of occurrences of a word in a given document in order to determine its score, then the word count of each document might be stored in the inverted index because this information is already available while it is being built.

Structure data is of importance because some of the words on a document might have a different meaning in the document encoding context (representing a field, for instance) or the search engine might need this information when presenting the results for the user (in case of occurrence positions)[11].

The word occurrence position is often a useful information when presenting the query results for the user and during the query processing. It can be implemented by just adding the word position in the posting. The word position is the location inside the document where the word can be found and can be thought as an index inside the document.

Moreover, as some words also represents structures inside a document, then it might also be important for the ranking algorithm to retrieve such information. In case of JSON files, some words might represent a field. This can be represented in the inverted index by creating a different kind of posting (called an *extended posting*) that, in addition to the document reference and the word position, also stores the word position at which the field starts and the word position of the last word contained in this field[11]. In Figure 3.5 there is an example of a posting where the word is present as the fourth word of the document. As the information *Field Start* and *Field End* are present, this word represents a field which starts in the fifth word of the document and ends at the eighth.

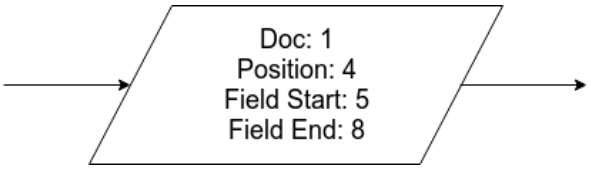

Figure 3.5: A posting example with the structure information..

3.3.2 Processing Queries

The inverted indexes explained above provide a way of rapidly retrieving documents in which a word can be found. Using this simple API, it is possible to perform more complex queries that involves multiple terms and therefore can retrieve all data needed by ranking algorithms, including the statistical and structural data needed that is stored in the inverted index, as explained in the last section.

Querying phrases

One of the simplest queries that a user might want to perform is that a certain phrase is present in a given document. In other words, the query to be performed is to find a document in which the sequence of words can be found[11].

This kind of query can be done using the document position info that was stored alongside the term postings in the inverted index. The query text is striped in terms and each term is searched in the inverted index. Then, with the list of postings of each term, the postings referring to the same documents are grouped and the word positions are verified in order to determine if the queried text is present in the the documents[11].

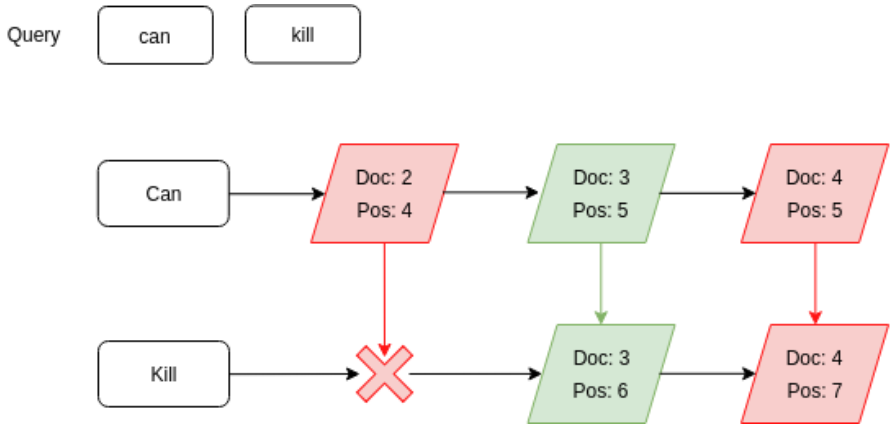

Figure 3.6: An example of a phrase query.

Figure 3.6 roughly shows how a phrase query can be performed using the inverted index. The occurrence of the word *can* in the document *2* is not followed by an occurrence of the word *kill*, therefore there is no match with the queried phrase. Document *3* shows a match, as the word *can* is immediately followed by the occurrence of the word *kill*. In the case of document *4*, both words are present in the document, but the positions do not match as the word *kill* occurs two words later than the word *can*, therefore not resulting in a match with the queried term.

The process described above does not account for the text pre-processing techniques such as stopping and stemming, since both the query phrase and the document text would run through these techniques which could result in different word positions. The word number *6* of document *4* might be a stop word and the word kill might be a result of the stemming process (like in the sentence "can be killed"). The same logic could be applied to the queried phrase which would result in matches not occurring even though the phrases are similar.

This issue can be addressed with thresholds for the distance between the words, increasing the tolerance of the matches as phrases that would not match otherwise now will be included in the query results[11]. In fact, the distance threshold might also be relevant for ranking algorithms that are not after exact matches[11].

Conjuctions and Disjunctions

Using similar concepts as shown in the phrase querying process, it is also possible to retrieve documents based on logic operations such as the conjunction and disjunction. These operators are useful when specifying multiple conditions in a query, allowing for more complex queries to be performed.

In the context of document querying, the conjunction operator aims to find documents in which every sentence in the conjunction is present. For instance, the expression *"can kill"* $\wedge$ *"sting"* (where $\wedge$ is the conjunction operator) will result in documents where both the phrase *can kill* and the term *sting* are present, regardless of the order of occurrence of both terms. Any document containing only one of the operands expressions will not be included in the results.

The disjunction operator returns every document in which at least one of the phrases in the operands occur. In the case of the expression *"can kill"* $\vee$ *"sting"*, each document where the expression *can kill* or the word *sting* are present will be returned in the query result. In other words, only documents where none of these expressions are present will be left out of the query results.

These operators can be implemented by performing each operand as a separate query and then performing list operations in the resulting documents lists[11]. A conjunction will result in the intersection of the documents lists while a disjunction will result in the union of the lists, as shown in Figure 3.7

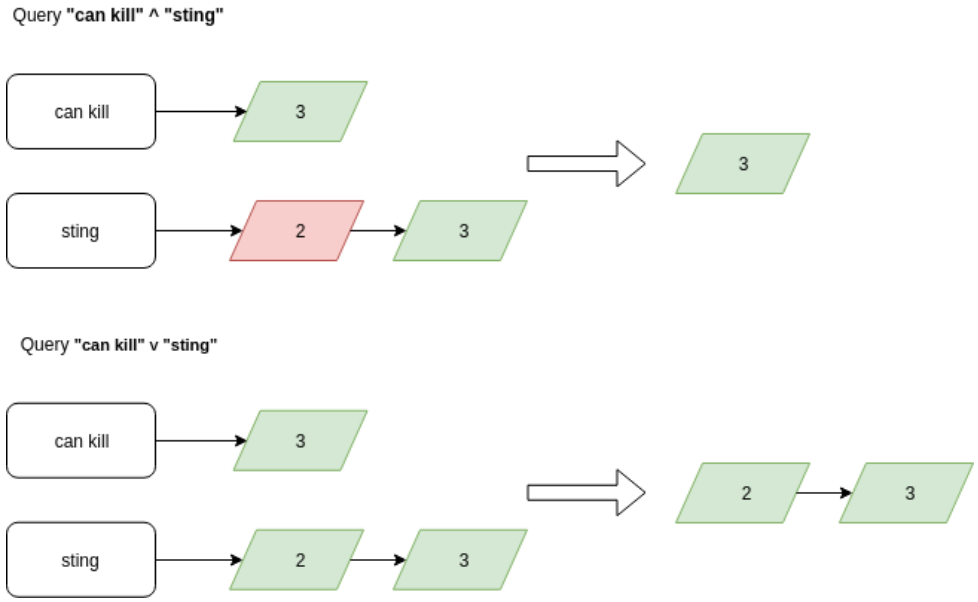

Figure 3.7: Examples of the disjunction and the conjunction queries.

Field Queries

The structural data that was stored in the inverted index, mentioned in the last section, can be useful when querying documents based on a specific field's content. As explained before, if the documents are stored in a semi-structured data encoding type such as JSON,

the fields in a document might contain useful information when performing queries against the database[11].

To perform the field queries, the *extended postings* mentioned in the last section are used in order to find the position of a field content in a document. Also the content is queried without the knowledge of the fields in the document (as if a simple phrase query). Possessing both the position of the occurrence of the queried content and the positions where the field starts and ends, it is possible to compute if the field contains the queried content[11].

As example, supposing the content to be queried is *computer science* and it is desired to find its occurrences in the *department* field of a collection. If the *computer science* expression is found in document *2* starting in position 3 and ending in position 4 and the *extended posting* of the *department* field referring to document *2* points that it starts at position 2 and finishes at position 4. Then, as the interval $[3, 4]$ is contained in the interval $[2, 4]$, the expression *computer science* is present in the field *department* of document *2*. Figure 3.8 illustrates this process.

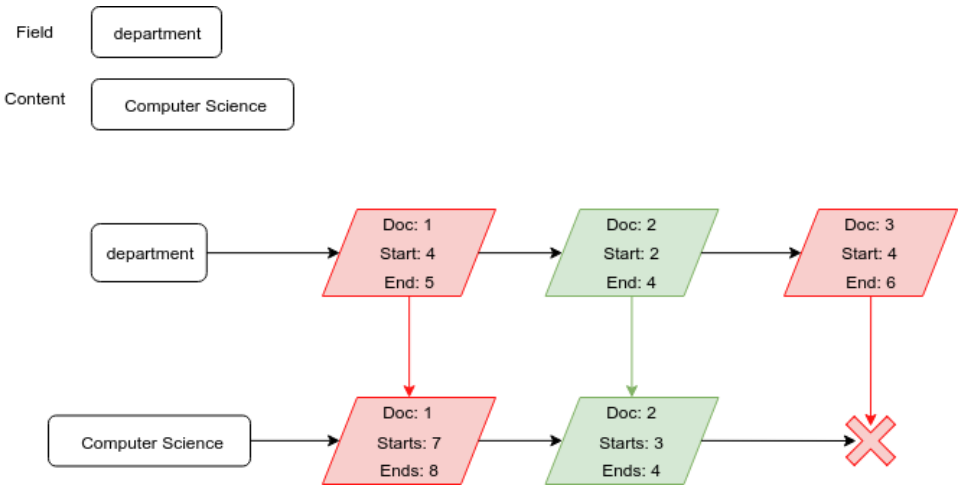

Figure 3.8: Examples of field queries.

Chapter 4

Implementation

This chapter exposes how the different components were implemented in order to enable the development of query languages in the Angra-DB project, following the project's principles of modularity and ease of use.

The first part of the implementation is about the creation of the inverted index module, called the *indexing module*, that provides a way to perform information retrieval of data stored in the documents collection. Next, some modifications were needed in the Angra-DB application in order to handle the new indexing module's new functions and requisites. With the indexing module in place, a *query processing* API was specified and implemented using the indexing module's functionality, providing basic querying capabilities that were to be used by the query languages in order to retrieve documents containing the desired characteristics. Finally, using the developed *query processing* API a query language could be implemented.

The following sections detail how each of these steps were implemented in order to achieve the query language for the Angra-DB project. The query language implemented was an adaptation of the JSON Query Language and will be further explained in the last section.

4.1 Indexing Module

The indexing module was implemented using the concepts of information retrieval and inverted indexes that were explained in the last chapter, with the focus in fast retrieval at the cost of slower insertions. Like the rest of the Angra-DB code, the module was implemented in Erlang.

To improve the flexibility of the indexing module, it was actually implemented into two sub-modules:

Table 4.1: JSON to Erlang data types.

JSON	**Erlang**
String	Binary
Number (integer)	Integer
Number (float)	Float
Object	Proplist
List	List
Null	null (atom)

- A parser module that receives a JSON document and yields a list of tokens that were extracted from that document
- The index manipulation module, which receives the parsed document and inserts the tokens in the inverted index.

This separation was done in order to enable the switching of both the parsing algorithm and the inverted index modules, providing a way to alter the implementation of these two components to achieve the information retrieval and performance needs.

4.1.1 Parsing Module

The parsing module was implemented using the Erlang *jsone*[1] module that provides functions to parse raw JSON documents. This module receives an Erlang list containing the JSON document (encoded in an Erlang binary) and parses the json contents into a list of Erlang terms that describes that JSON document. For example, the JSON document:

```
"{"name": "Fernando", "age": 23}"
```

Is decoded in a list of Erlang tuples that represents each pair in this JSON object, returning the erlang term:

```
[{<<"name">>, <<"Fernando">>}, {<<"age">>, 23}]
```

As can be noticed above, each JSON type is translated to a corresponding Erlang data type, an object is translated to a list of tuples of each key-value pair (also called *proplist*), a string is translated to a binary, a number is converted to an Erlang integer. The full lists of correspondences can be found in Table 4.1

Using Erlang's functional programming features (specially pattern matching), it is possible to obtain each of the JSON's words and also determine the semantics of each word in the JSON encoding structure. For instance, the pattern:

[1] Available at `https://github.com/sile/jsone`

```
json_parser([{<<FieldName/binary>>, Content} | Tail]) ->
    % Token generation code.
```

Identifies that the next element in a JSON returned by the *jsone* module is a key value pair of an object. Thus, it is possible to return a special token that represents a field with name *FieldName* (the first variable in the pattern) and then parse the content of this field (that might also be an object).

That said, the parsing module's returned list of tokens actually contains two types of tokens:

- **Normal Tokens:** tokens that only represent the occurrence of a word in the document, i.e., the occurrence is the content of a field and might be a string, a number or null. They contain the position of the word in the document.
- **Extended Tokens:** tokens that represent the name of a field. Besides the position of this word in the document, they also contain the position of the first word in the field and the position of its last word.

These two types of tokens are used to build and update the inverted index.

4.1.2 Index Manipulation

This module contains all of the index manipulation functions and handles the files needed in order to manage the index updates and searches. The current operations that are supported in the inverted index module is:

1. Insert a list of tokens into the index
2. Retrieve the list of postings associated with a word.
3. Delete the occurrences of a word from a document.

These operations reflect the CRUD operations that might be performed into the AngraDB and might be implemented in a multitude of ways. The following implementation was inspired by the inverted indexes techniques described in the previous chapter.

Architecture

In order to increase the performance of the insertions in the index, the inverted index was split into two. One of the indexes is stored into memory (the memory index) while the other is stored into the secondary storage (the file index).

The update and delete operations are performed first in the index present in memory and only when this index reaches a configured maximum size, it is merged with the index present in the secondary storage. This approach makes most of the insert operations very fast at the cost of a longer merge duration when the memory index is full.

Data Structures

Three main data structures are part of the inverted index module:

- **Normal Postings** are the most common of the word occurrences and represent a occurrence of a word in the content of a JSON field.
- **Extended Postings** are a special type of postings that represent the occurrence of a word that is a field name
- **Term** a term is a list of postings (both normal or extended) associated with a word. It contains all of the occurrences of said word in the document collection, also storing statistical data.

Normal posting contains the key of the document which the word occurrence refers and the word position in the document, as explained in Chapter 3. Extended postings contains the same properties of a normal posting with the addition of two word position that indicates where the field referenced by that word starts and ends. In addition to these fields, a document version number is also stored in the postings in order to determine if it refers to an older version of the document, i.e, the word was removed from it (the usage of this property will be later explained). All word positions are zero based which means that the first word in the document has position *0*. The structure of the postings are represented in Figure 4.1

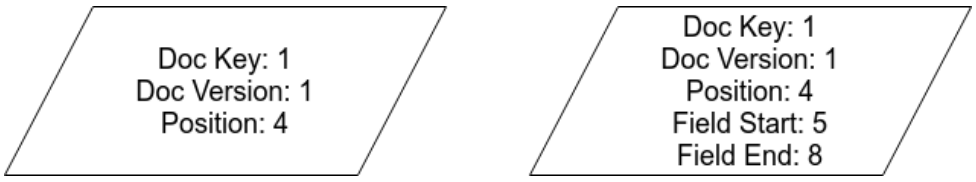

Figure 4.1: On the left, the structure of a normal posting. On the right, the structure of an extended posting.

The term data structure stores the word it is associated, the number of normal postings, the number of extended postings and the list of postings of the stored word. The statistics stored can be used not only in the information retrieval model, but also can help when storing this term in a file on the secondary storage (as will be explained later).

Finally, the last structure is the index itself. It is made of a list of terms that is alphabetically sorted in order to perform the merge algorithm of the memory index with the file index.

Insertion Algorithm

As explained before, all insertions are performed at the memory index. This means that the algorithm for the insert operation does not need to handle writes on the secondary storage and can use some optimizations for searches and insertions of new data.

The insertion expects a list of tokens received from the parser (already classified in normal tokens and extended tokens), the key and the version of the document associated with the received tokens. It then transforms each token in the corresponding posting and inserts it in the inverted index using the following algorithm:

```
search for a term which contains the token's word.
if the term was found then
    add the posting to the term, sorted by document key.
    increment the term's word counters.
else
    create a new term with the token's word.
    add the posting to the new term.
    increment the counters.
```

As the lists are present in memory, it is possible to perform the searches using a binary search, increasing the performance of the insertions.

It is important to remember that with the architecture proposed previously, when the size of the memory index increases to a certain configurable threshold, then it is merged with the file index. However, in this implementation, the index manipulation module was not responsible for the management of the index size, instead, it only provided the function to merge the two indexes that will be used later when integrating the inverted index module with the Angra-DB project, which will manage its size.

File Index

The usage of the memory index helps to improve the insertion's performance, but to obtain a matching performance in the retrieval operations, the inverted index must be stored in the secondary storage in a structure that enables the search of a term with the least amount of disk accesses. In order to do so, the same structure as shown in Figure 3.4 was used.

At the start of the file, a header was stored with the hashtable containing the pointers to the location where corresponding terms were stored. The hash function is passed as an argument to the indexing functions (using Erlang's high order functions feature), which means that the hash function might be configurable. This leads to the problem case where the passed hash function is not collision free, thus requiring a collision handling strategy. For this implementation, the algorithm used is a bucket algorithm in which each term can point to another term's location in case both of them share the same hash, therefore creating a linked list of terms. The file locations were stored as offsets from the beginning of the file.

Each term was stored following the structure:

```
word : normal postings counter : extended postings counter :
next term pointer : postings
```

The colons are merely illustrative, each of these fields were stored with a configurable fixed size of bytes in the file, in order to simplify the reading process. The next term pointer is the location of another term that has the same hash, creating the bucket that was mentioned earlier. The postings were stored contiguously and accordingly to the following structure:

```
posting type : document key : doc position : doc version :
*field start : *field end
```

The posting type field is a field that indicates the type of the stored posting, wether it is a normal or an extended posting. The fields marked with * were included in case of extended postings. Again, the fields were stored with a fixed byte size in order to simplify the postings reading process. This means that only with the number of extended and normal postings, it was possible to find out the length in bytes of the postings list of a term.

Retrieval

The retrieval operation retrieves all of the postings associated with a given word. Therefore, the retrieval operations should lookup in both the memory and file indexes and merge the postings present in both found terms as they both contain the state of the document collection at the retrieval time.

The memory index term can be found in the same way it was found during the insertion algorithm and its postings are already in memory, making this retrieval faster than the file index.

For the file index, first the hashtable present in the file header must be fetched, then using the query word's hash as an index, the location of the document in the file was retrieved from the hashtable. The term reading process was performed in two storage accesses. The first one obtained the word associated with the term and the postings' counters, as well as the next term in the bucket pointer. If the word in the term matched the query's word, then the term was found and another storage access should be performed in order to retrieve the postings (using the postings' counters). Otherwise, the next term in the bucket was analysed until none was left, in which case the word could not be found in the file index.

With both posting lists retrieved and stored in memory, a merge algorithm was performed in order to return the postings ordered by document keys. The merged postings is the result of the retrieval operation.

Merging

The merging function was provided by the inverted index module to enable the merging of the memory index into the file index. The following algorithm was used:

```
start at the first term of each index.
while any of the indexes is not empty:
    fetch both terms
    if memory term is greater than file term
        rewrite the file term
    else if the file term is greater than memory term
        write the memory term
    else (both are equal)
        merge the postings of both terms
        update the counters
        write the result
write the rest of the remaining index.
```

All the comparisons in the above algorithm were made using the alphabetical order of the terms. This means that both the memory index and the file index are considered to be ordered (and are kept sorted by this algorithm). Also, the operations involving terms are made in memory, once the file term is retrieved, it can be treated as a memory term, increasing the performance of the merging process.

Updating and Deleting

The algorithms presented before did not take into consideration the fact that the documents might change or be deleted in the collection. This means that it does not take into account that postings might become invalid and should not be written to the file index and should not be returned during a retrieval. In order to cope with this situation, two auxiliary structures were created.

The first one is a list of deleted document keys. The results of a retrieval is checked against this list of keys and any posting containing any of the deleted keys is excluded from the result. Bear in mind that the postings were not actually excluded from the index yet, they are simply invisible when querying the index. The invalid postings are actually deleted when merging the two indexes. When writing a term (be it a file, a memory or merged term) the postings to be written are checked against the deleted keys, like in the retrieval. The postings are not written to the file, thus resulting in their deletion.

For the update, a similar approach could be used, but instead of the list of deleted keys, a list of the latest document versions was used. The checks during the retrieval and merge were done using the document version stored into the posting and if a later document version was found, then the posting was excluded from the operation.

It is important to notice that not every document's version and deleted keys needed to be stored in the lists, but only the one's deleted/updated after the latest merging of the indexes, since after that, the postings were effectively deleted. The index manipulation module provided all the functions that handle these two lists.

4.2 Changes to Angra-DB

In order to handle the management of the inverted indexes associated with the document collections, changes were made to the *adb_core* application. These changes were mainly in the supervisor tree structure and were made in order to enable a better handling of each collection state. The new *adb_core* structure is shown in Figure 4.2.

In the new architecture, the calls to the *gen_persistence* behaviour were removed from the *adb_server* module. Instead, this module now calls one of the new servers that are handling the connected database and maintain the collection status.

Now, instead of only the *adb_sup* supervisor below the *adb_core* application a *persist_sup* supervisor was added. The purpose of this supervisor is to maintain a child supervisor of type *db_sup* for each collection that was connected in this Angra-DB instance. For example, if the collections *musics* and *books* were connected by any of the TCP connections (by issuing the connect Angra-DB command), then two *db_sup* supervisors would be spawned below the *persist_sup*, each one handling a collection.

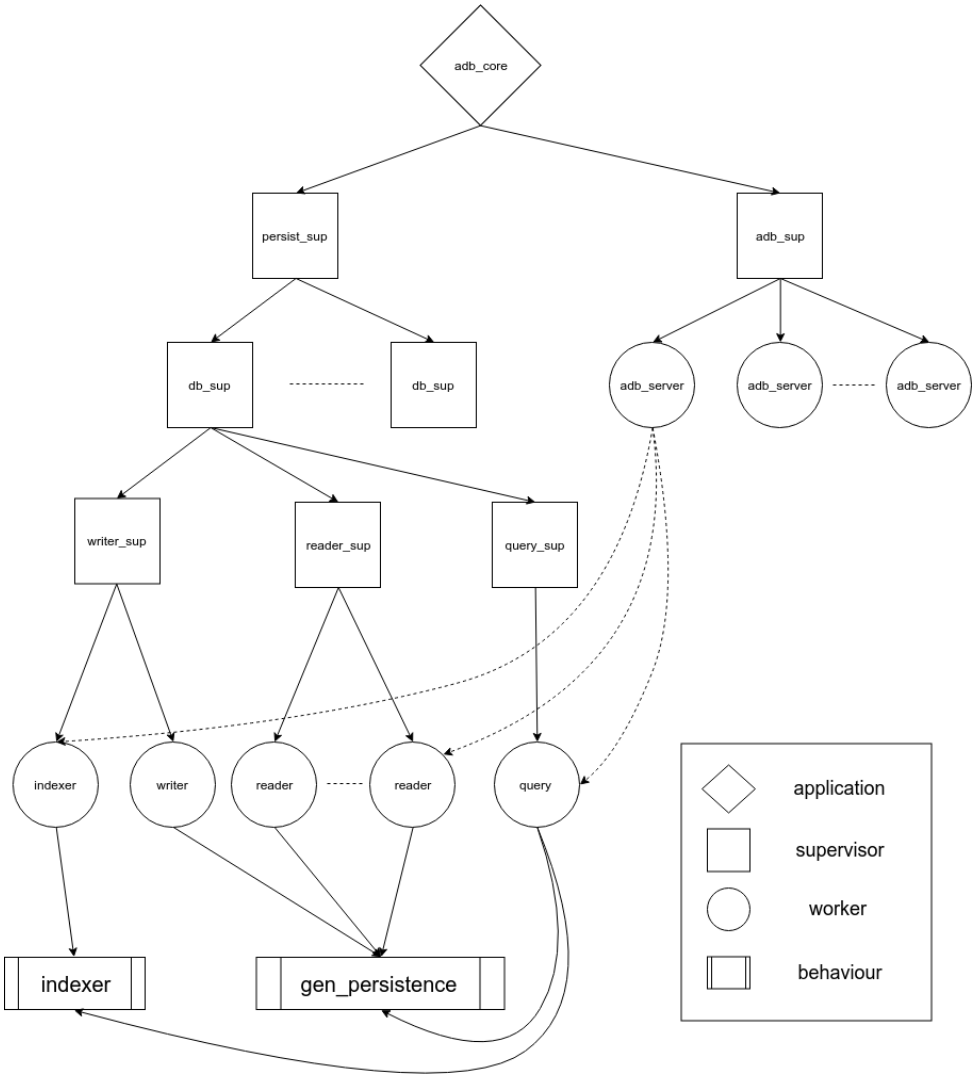

Figure 4.2: The new architecture of the *adb_core* application.

As hinted before, each *db_sup* is responsible for the management of a different collection. It spawns three supervisors that have children that handle different types of operations in the collection.

The *writer_sup* has two children that are responsible for the write operations in the collection. One of its children, the *writer*, is responsible for the insertions, updates and deletions in the collection. As there is only one writer per collection, then it means that the operations are serialized in this worker. The *indexer* worker is responsible for the management of the index manipulation, it uses the functions of the inverted index module to update and maintain the inverted index structure of the collection. Just as the *writer*, that is only one instance of the *indexer* per collection, meaning not only that its operations are serialized, but also that it only has to manage one copy of the inverted index (reducing concurrency problems). The *indexer* is not called directly by the *adb_server*,

instead, it is called by the other workers (*reader*, *writer* and *query*) when they change the state of the collection. For instance, when the *writer* inserts a new document in the collection, it also sends a request to the *indexer* to update the inverted index with the tokens present in that document.

The *indexer* worker was the main purpose of this architecture change, because it enables for the management of the inverted index using the functions provided by the inverted index module presented previously. It maintains the memory inverted index in the server state and updates it according to the requests received, also making updates to deleted documents list and latest versions list. If any operation increases the memory index to a size above a certain threshold configured in the *adb_core* settings, then this worker performs the merge of the file and memory indexes, flushing the memory index and the two list after the merging completes. Also this worker is responsible for starting the memory index in case it was stopped without saving its contents to the file index. To do so, it uses the latest versions list, as it contains all the documents added to the collection after the last merge, retrieving and indexing these documents.

The *reader_sup* manages multiple children (using the *simple_one_for_one* strategy) that has the functions for the retrieval of both documents and term occurrences from the collection. As there are multiple readers available, the reading process can be done concurrently. This way, it is possible to notice that this architecture enables for concurrent reads from the collection (as there are not any concurrency issues in these operations), but the writing operations must be done in a sequential manner.

The *query_sup* will be subject to the next section.

4.3 The Query API

In order to comply to the Angra-DB project principles as much as possible, the architecture to be developed in order to add a query language feature for the database must be extensible and as loosely coupled as possible. With this is mind, the architecture shown in Figure 4.3 was proposed.

This architecture revolves around the query processor component ability to perform generic queries to the database. In other words, the query processor component provides a set of basic query APIs that can be used by the query languages in order to retrieve information of the document collections. To develop a query language for the Angra-DB databases, it is a matter of converting the executed queries in the developed language to a query for the query processor component and then treating the results accordingly.

In this first implementation, the query processor exposes two filtering API's: one for filtering based in phrases occurrences in a documents content, and another that filters the

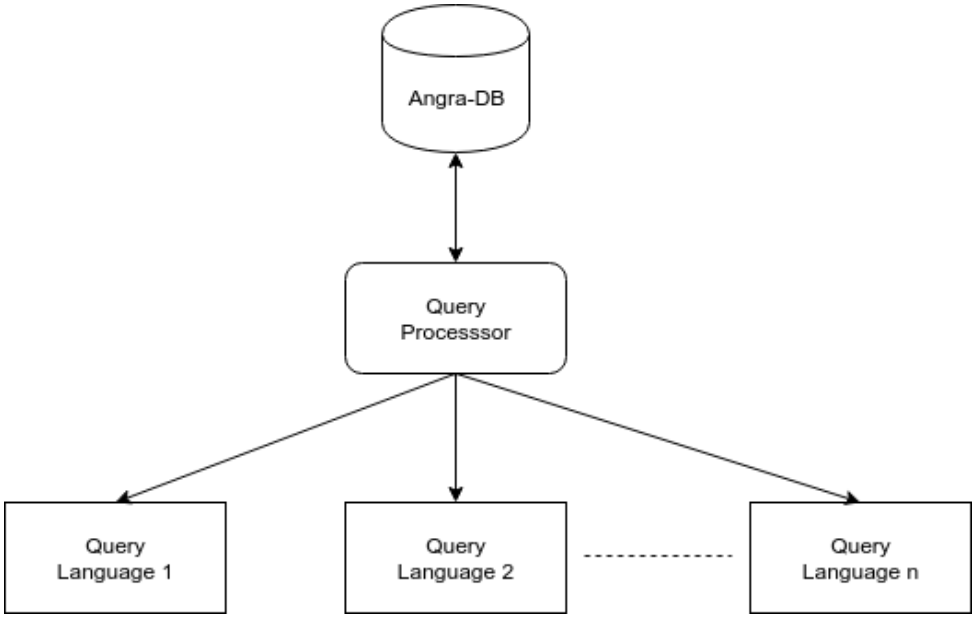

Figure 4.3: The architecture of query languages building.

documents by its fields. Also, two operators are implemented in order to combine filtering results.

4.3.1 The processor in the Angra-DB architecture

The query processor component was implemented as a new command available in the Angra-DB TCP interface. This command accepts one argument that contains the query to be performed and returns a list of documents that were found in the connected collection, along with information of the occurrence locations (extracted from the postings).

The query processor requests were then handled by the *query* server, shown in Figure 4.2. This worker makes requests to the *indexer* server in order to retrieve the information available in the inverted index. It then performs the required operations with the received postings and returns the results of the query.

4.3.2 API

The two filtering API's provided by the query processor are:

- `filter <phrase>: returns all documents that contains "phrase"`
- `filter_field <field names>: returns all documents that contains the field "field name"`

The *filter_field* command when used in isolation only returns the documents in which the queried field is present, as the documents in a collection are semi-structured and might not contain the same fields. However, this command can be used in addition to the *filter* command in order to obtain only the documents in which *phrase* is present in *field name*. This can be done because, in the implementation of the query processor, filtering commands can be chained and each of them might apply additional filtering to the results of the previous filter called.

The *filter* command uses the postings word position in order to find if the documents contain the argument's phrase. It is done using the same algorithm illustrated in Figure 3.6 and does not take into consideration the previous filtering command, thus, it must be the first command in a filtering chain. The result is a list of documents in which the phrase appears, also containing the word positions of the phrase in the documents.

The *filter_field* command uses the algorithm shown in Figure 3.8. The content filtering is done using the results of a previous filter (a *filter* command result) and the extended postings *field start* and *field end* properties to determine if the document is a valid result. Moreover, this command accepts a list of field names as its input, in which case it will verify the contents of nested fields. For instance, the command:

```
filter_field course department name
```

Will look for documents that contains the field name inside a field named department that is also inside a field named course. In an analogy with an object oriented programming language, this command would act as if accessing the field *course.department.name* of an object.

With these two API's it is already possible to execute some complex queries in the documents collections. However, two more commands are also available to enable the combination of multiple filtering results. They are:

- `<filter query 1> and <filter query 2>: returns the documents that are present in the results of both <filter query 1> and <filter query 2>`
- `<filter query 1> or <filter query 2>: returns the documents that are present in the results of either <filter query 1> or <filter query 2>`

With these additional commands, a wider range of queries can be performed in the database.

Moreover, a command for paginating the results of a query, in order to reduce the amount of data transferred through the API is shown below:

`interval <start of page> <num of documents>: returns a subset of the results of a query, skipping <start of page> number of documents and returning <num of documents> documents.`

The results of every query are lists of document keys. In order to efficiently retrieve these documents, a new TCP command was added to retrieve all documents present in a list of document keys. In the end, the TCP commands that were added to Angra-DB were:

- `query <query text>: sends <query text> to the query processor`
- `bulk_lookup <keys>: retrieves all documents that have a key contained in <keys>`

4.4 Query Language

With the query processor in place, the actual query language could be developed. The language implemented is a subset of the JSON Query Language (JSONiq), and was implemented in a separated Erlang application than the *adb_core*. The application was a simple Erlang server that enabled TCP connections, received JSONiq queries and connected to the *adb_core* application in order to retrieve the database's documents using the query processor. Figure 4.4 shows the JSONiq application's architecture.

4.4.1 Lexical Analyser and Parser Generators

In order to build the query language, two Erlang libraries were used. Both of them are part of Erlang's OTP and are Erlang versions of well known GNU libraries.

Yecc, is an Erlang parser generator that is inspired in YACC (Yet another compiler compiler). It has an Erlang like syntax for describing programming language structures, in the form of rules that must be matched by the parser input. As it is inspired in YACC, it has a complete documentation and as it uses Erlang syntax, it also provides a familiar way for handling the rules results. The output of Yecc is an Erlang module that contains a parser with the specified rules. The parser rules are patterns that must be met in order to a language structure to be found. Refer to Yecc documentation[2] for more information on how the parser can be specified.

Leex is an Erlang lexical analyser generator that is inspired in LEX. As with Yecc, it has an extensive documentation and uses Erlang syntax in order to define rules that are used to tokenize an arbitrary input. The output of Leex is a lexical analyser that

[2]Available at `http://erlang.org/doc/man/yecc.html`

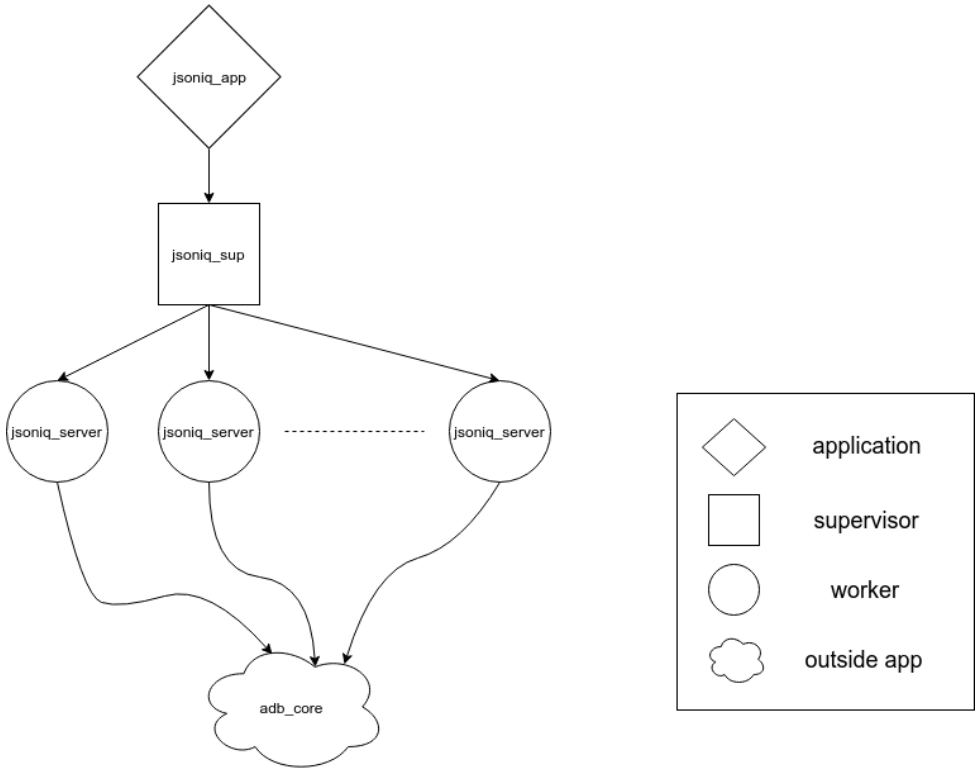

Figure 4.4: Architecture of the JSONiq App.

tokenizes the inputs accordingly to the rules specified. Each rule is a Regex expression that, if matched, produces a certain type of token. The tokens resulting from the lexical analyser are expected in the parser generated by Yecc. The documentation of Leex[3] explains how the lexical analyser can be specified.

4.4.2 JSON Query Language

JSONiq is a query language designed to handle JSON documents. It is inspired in XQuery, a query language that is successfully used to manipulate semi-structure data, specially in XML format. It uses some similar syntax of Javascript but adds functional language structures.

JSON Query Language has a powerful JSON transformation syntax that enables for efficient JSON documents manipulation. Also, a intuitive querying structure called FLOWR is borrowed from XQuery in order to access collections of JSON documents and perform operations in them.

[3]Available at `http://erlang.org/doc/man/leex.html`

JSON Parsing

The most basic JSONiq query is a JSON data. This means that any valid JSON is a valid JSONiq query. All of JSONs primitive types are supported in JSONiq. In Angra-DB's implementation, the following primitive types were implemented:

- **integers:** integer numbers.
- **strings:** sequences of characters enclosed in *""*.
- **boolean:** the values *true* or *false*
- **null:** JSON's *null* value

Also, the compounded types were implemented, like objects and lists. For instance, the following JSON's are valid queries in this implementation:

```
{
    "name": "Fernando",
    "surname": "Nunes",
    "age": "23"
}
```

```
[
    "Bees",
    "Ants",
    "Wasps"
]
```

Operators

JSONiq's specifications provides a wide range of operators for processing and retrieving JSON documents. In this first query language implementation, the following operators were implemented:

- **arithmetic:** $+$, $-$, $*$, *idiv* (integer division), *mod* (modulo)
- **comparison:** $\leq$ (le), $<$ (lt), $\geq$ (ge), $>$ (gt), $=$ (eq), $\neq$ (ne)
- **boolean:** *and*, *or*, *not*

Also, object and array accessing operators were implemented. For instance, given the object:

```
let $student = {
    "name": "Fernando",
    "surname": "Nunes",
    "age": "23"
}
```

The expression *$student.name* would yield the string *"Fernando"*. The expression above also shows another structure of the JSON query language that was implemented, the attribution of variables to JSON values. In this case, the variable *$student* holds the object value:

```
{
    "name": "Fernando",
    "surname": "Nunes",
    "age": "23"
}
```

In case of arrays, the operator *[[ ]]* can be used to access the value of a certain index. For instance, the array:

```
let $insects = [
    "Bees",
    "Ants",
    "Wasps"
]
```

Could have its second value accessed with *$insects[[1]]* (zero based indexes), which would return the value *"Ants"*.

Arrays also possess a predicate operator which filters the array and returns the elements for which the predicate returns *true*. For example, with the above array, the expression:

```
$insects[ it eq "Wasps" or it eq "Ants" ]
```

Would return the filtered array:

```
[ "Ants", "Wasps" ]
```

Inside the predicate, the variable with name *it* refers to the element of the array that is being evaluated by the predicate (in a similar way to a lambda function).

FLOWR and Database Access

So far, every operator was processed with the JSON documents in memory. In order to perform queries in the database, a special JSONiq structure is used, the FLOWR expressions. In this implementation, the syntax of a FLOWR expression is:

```
for <assignment> in collection(<collection name>) <pagination args>
    where <conditional expression>
    return <map function>
```

The first parameter of a FLOWR expression is an assignment that will be used throughout the FLOWR expression to refer to each individual document. The second parameter is the collection name from which the documents must be queried. The third parameter is an optional pagination options that will be explained later. Next, is an optional *where* clause which can be used to apply a filter to the documents in the referred collection. In this first implementation, only tests for equality might be performed (as these are the only querying capabilities of the query processor component of Angra-DB). The return clause maps each resulting document into a new JSON value, which might be the retrieved document itself. For instance, if the following documents are present in collection *musics*:

```
{
    "title": "Smells like teen spirit",
    "album": "Nevermind",
    "artist": "Nirvana"
},
{
    "title": "Black",
    "album": "Ten",
    "artist": "Pearl Jam"
},
{
    "title": "Plush",
    "album": "Core",
    "artist": "Stone Temple Pilots"
},
```

The following FLOWR expression could be written:

```
for $music in collection("musics")
    where $music.title eq "Black" or $music.artist eq "Pearl Jam"
    return $music.album
```

This would yield the list:

```
[ "Nevermind", "Ten" ]
```

Which is the result of the documents filtered by the where clause and mapped by the return clause.

The pagination options are optional arguments that enables the query to specify a subset of the results to be returned. It uses the *interval* command from the query API and has the following format that can be translated into a query for the query processor:

```
[ <start of page>, <num of elements> ]
```

The where clause of FLOWR expressions are translated into a query for the query processor component of *adb_core* and its resulting keys are then fetched from the database using the *bulk_lookup* command. The resulting documents are then parsed in the JSONiq language processor and are made available for the rest of the JSONiq expressions. That said, more operators can be added to the FLOWR expression by implementing the corresponding query in the query processor component of *adb_core*.

Chapter 5

Testing And Evaluation

In order to validate the proposed architecture and implementation of the components, three tests were performed comparing different configurations of the proposed architecture and comparing its performance with a reference documental database, CouchDB. The executed tests were:

1. Assess the impact of the memory index size in the performance of Angra-DB
2. Compare the performance of Angra-DB's document insertion with CouchDB's
3. Compare the querying performance of Angra-DB's new Query Language architecture with CouchDB's

Also, a qualitative analysis of different aspects of the query languages used in both CouchDB and MongoDB is presented at the last section of this chapter.

5.1 The Dataset

The dataset used in order to perform the following tests is a sequence of blocks extracted from Ethereum blockchain [13]. Blockchains are a recent technology that are the basis of cryptocurrencies such as BitCoin and Ethereum. This technology has the property of being an append only distributed ledger, which contains a sequence of transactions that are validated by a network[13].

Basically, a blockchain is composed by a sequence of blocks in which each block contains a cryptographic hash to the previous one in the sequence, this way ensuring that to modify a previous block, all posterior blocks would need to be altered (since all the hashes would change)[14]. Each block is a set of transactions, which is actually just data to be written in the blockchain according to a set of rules. This sequence of blocks is replicated across a network, and each node in it is responsible for the verification that every new

block to be added passes a certain validation rule, meaning that not only the block is valid, but also that the contained transactions are valid according to their rules[14]. This way, in order to add a malicious transaction to the blockchain, an attacker would need to take control of a significant part of the nodes in the network[14].

Further explanation about blockchains can be found in [15][16] and is out of the scope of this work. Blockchains were chosen as a source to the dataset because they provide a constant flow of data that can increase to great proportions. For instance, Ethereum blockchain has about 120 GB in one of its space saving modes at the time of this writing[1]. Moreover, as blockchains can store not only monetary, but about any type of data in its transactions, the querying capabilities of documental databases can also be applied to this technology. For example, Hyperledger Fabric, an open-source framework for blockchain development, provides CouchDB as its storing mechanisms in order to enable querying in its blockchain state[2].

That said, each block extracted from the blockchain contains the following structure, which is a resumed version of the blocks fetched from an ethereum node:

```
{
    "transactions": [
        "0xd7ff2b20864a31...",
        "0x9a99f58c5095d1..."
    ],
    "number": "0x38b033",
    "hash": "0x21ff51da33...",
    "author": null,
    "sealFields": null,
    "parentHash": "0xbcbbcd3a8...",
    "nonce": "0x0000000000000000",
    "sha3Uncles": "0x1dcc4de8dec75d7a...",
    "transactionsRoot": "0xd21c0317f9a85b...",
    "stateRoot": "0x5a2fa547dbef...",
    "receiptsRoot": "0xa546650e3d...",
    "miner": "0x0000000000000000000000000000000000000000",
    "difficulty": "0x2",
    "totalDifficulty": "0x68710a",
    "size": "0xf1c",
```

[1] https://etherscan.io/chart2/chaindatasizefast

[2] https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_tutorial.html?highlight=CouchDB

Table 5.1: Time spent on the insertion of 20.000 documents and insertion and merge average duration.

	Total time	**Avg merge (ms)**	**Avg save (ms)**
2 MB	00:44:49	71180	28.03
5 MB	00:36:59	73967	64.37
10 MB	00:52:03	83024	128.98


```
    "gasLimit": "0x6adf43",
    "gasUsed": "0x18cd54",
    "timestamp": "0x5c4267b9",
    "uncles": []
}
```


5.2 Memory index size tests

One of the configurable aspects of the proposed architecture is related to the maximum size of the inverted index in memory, before it is merged into the file index. This configuration determines the performance of Angra-DB insertions because, as of the current implementation, the database is locked during the merges. A great max size then, would result in a greater amount of documents inserted quickly before a merge occurs, but also results in longer merging times. On the other hand, a small index max size would result into faster but more frequent merges. The balance between frequent and long merge times is what is being assessed in this test.

The tests were performed by inserting 20,000 blocks from the ethereum blockchain using three different configurations for the max index size, 2 MB, 5 MB and 10 MB. From each configuration, the total time of insertion and the average times of merging and saving were measured. These results are shown in Table 5.1.

The table indicates that increasing the memory index not only increases the merging times but also the time required to save a document. In addition, it also shows that a middle term index size does indeed produce the best performance. However, to identify the flaws of each configuration, further data must be analysed.

From the results obtained in the previous test, the total time spent in merging the indexes and saving the documents was calculated, which is presented in table 5.2.

Table 5.2 shows that the smaller index, as expected, spends most of its time in the merging process, which increases its total time since it is a costly operation that involves disk accessing.

The results of the greater index size shows that it spends the majority of its time into saving operations, which are done in memory. This means that this size of index allows

Table 5.2: Total time spent in index merge and document insertion.

	Total merge time	**Total save time**
2 MB	00:35:35	00:09:20
5 MB	00:14:47	00:21:26
10 MB	00:08:18	00:42:58

a number of terms and postings that are beyond what was expected when implementing the memory index insert operations resulting into longer insert operations than with smaller sized indexes. Figure 5.1 shows the progress of saving execution duration through the increasing of the collection size across the three different configurations. As can be noticed in the graphic, when the index is merged the insertion duration is significantly reduced, since the memory index is flushed. Moreover, the graphic also shows that, even tough the merging times were averaged higher in the 10 MB index, they were fewer, which resulted into less time spent into merges.

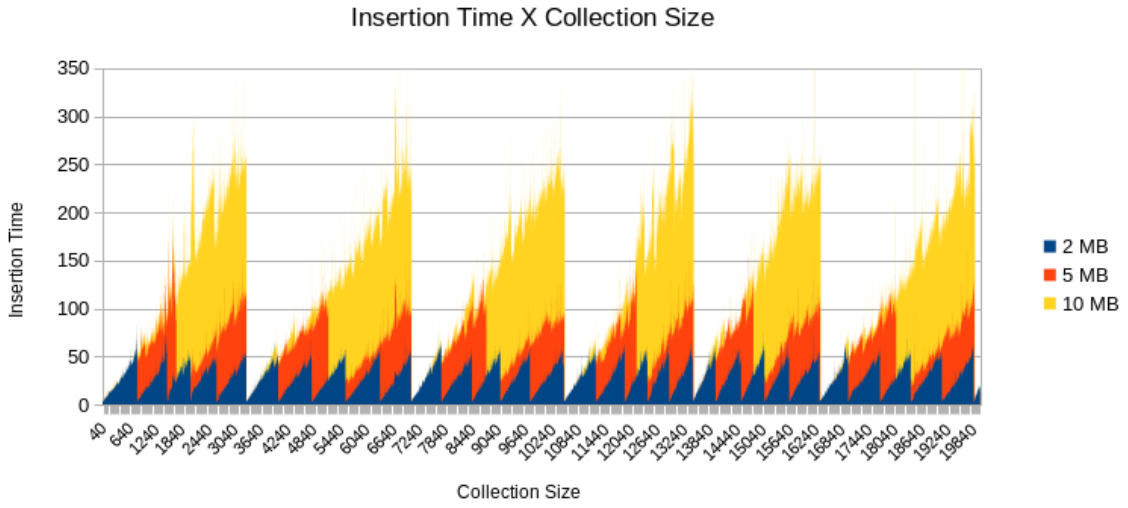

Figure 5.1: Insertion progress accross the configurations.

The results indicates that the performance of the insertion of a great number of documents can be significantly increased by changing the structure of the memory index to allow faster insertions, using, for instance a binary or btree structure.

However, as the tests also show, the merging duration increases with the size of the collection (which is expected since the index file gets larger), which increases the overall response time of the database insertion when it receives a large amount of documents. In order to improve the scalability of the Angra-DB database, a better merging mechanism must be implemented in which the database does not lock during merges.

Table 5.3: CouchDB x Angra-DB: insertion of 40.000 documents.

	Angra-DB	**CouchDB**
1 Thread	02:00:27	00:13:24
4 Threads	01:59:57	00:04:49

5.3 Comparisons with CouchDB insertion

The implemented architecture had its performance tested against CouchDB, which had some characteristics that were not yet present in the current implementation. For instance, CouchDB implements a Multi-Version Concurrency Control (MVCC) which allows for database writes without locks. This means that CouchDB's writing performance could be improved by using concurrent write requests to the database. On the other hand, Angra-DB's current implementation serializes the write requests, not benefiting from concurrency.

The tests that were approached two scenarios where 40,000 documents were submitted to the database:

- A single-threaded one where each document was submitted sequentially
- A multi-threaded scenario where the documents were submitted concurrently using four threads.

The results are presented in Table 5.3 and shows that Angra-DB's performance in insertions is lacking compared to CouchDB's. This result can be attributed mainly to the new indexation mechanisms implemented, as can be noted when comparing this results with the ones found in previous works [17]. Just as the results in the previous section, this means that the indexing module needs further improvements in its indexing architecture. Finally, Angra-DB did not benefit from the parallelism in the second scenario, just as expected, while CouchDB's performance greatly improved.

5.4 Query performance tests

With the collection that resulted from the previous test, two retrieval tests were performed in order to compare the querying performance of the new Angra-DB query language in comparison with CouchDB's Mango query language. The tests performed were:

- Retrieve a document by a unique field content
- Retrieve a page of a query result

Table 5.4: CouchDB x Angra-DB: lookup by unique field.

	Total lookup time	**Avg lookup time**
Angra-DB	00:00:55	555.73
CouchDB	00:01:58	1179,89

Table 5.5: CouchDB x Angra-DB: list 100 documents.

	Total lookup time	**Avg lookup time**
Angra-DB	00:01:58	1185,07
CouchDB	00:01:07	672,43

The first scenario was implemented by searching the blocks with a certain *number* field. The field content was incremented by a step of 300 in order to retrieve documents in different parts of the collection and 100 documents were retrieved. The total time spent in querying all the documents and the average lookup time is shown in Table 5.4

As can be noted in the results, the performance of the lookup by a unique field is better in Angra-DB than in CouchDB. It is important to address that the queried field was not indexed in CouchDB database, since the field specific index feature is not present in Angra-DB yet.

The second scenario tests the retrieval of multiple documents and the pagination feature of each DODBS. It was implemented by querying a page of 100 documents across the database, in a similar way as the first test. The results for this test are shown in Table 5.5.

Figure 5.2: Page retrieval execution time grows in CouchDB as the page offset rises.

The results show that for this type of query, CouchDB initially presents a better

performance than Angra-DB. However, by further inspecting the query execution time for each page, CouchDB's shows that it increases with the page offset of the query, while Angra-DB's execution time stays around the average value. This behaviour can be seen in Figure 5.2

5.5 Query Language Evaluation

Having performed some performance tests with the developed architecture, this section is intended to discuss the developed query language as a programming language, comparing it with other available options that are used in other DODBS.

5.5.1 About the evaluation criteria

Programming languages evaluation criteria are qualitative characteristics [18], meaning that each characteristic relevance is relative to each programmer. In spite of that, three main criteria are agreed to be most relevant [18], which are:

- Readability
- Writability
- Reliability

Readability mostly refers to the ability of the code written in a programming language to be understood easily. Some characteristics of a programming language often improves its readability. For instance, the ability to create data structures to represent specific data types often turns the program more legible [18].

Writability is a measurement of how easily the code for a certain problem can be generated using a given programming language [18]. Expressivity refers to the ability of a language to express relatively complex computation with small language structures. The expressivity of a programming language, for example, is a characteristic that often improves a language writability.

The reliability is a criteria that evaluates if the language provides tools and assists the programmers to create reliable programs. A program is considered reliable if it performs according to its specification under all conditions [18]. For instance, type checking and exception handling capabilities increases the reliability of the code, since they ensure that the program performs as intended, be it by checking that the variables are used as they should (like type checking), or by providing ways to handle undesired outcomes (such as exception handling).

Moreover, as each programming language might have an specific usage domain, i.e, a type of problem that it is specialized in addressing [18], the goal of a program might also have an impact in the programming language evaluation because the language might have special structures that aid the solution to a domain's problem (which would improve its readability and writability) while not having any clear way of solving a problem of other domains. That said, in this section, the Angra-DB query language will only be compared to other languages that are used for the same purpose, which is accessing a database. More specifically, the languages that will be compared are the ones of other documental databases, CouchDB and MongoDB.

5.5.2 Comparing with CouchDB and MongoDB standard query languages

As Angra-DB Query Language (AngraQL) is in its first versions, the comparisons of the query languages will be done using only queries that are already available in the current version. Also, the example queries shown below will use the same dataset used in the performance tests, which is a sequence of blocks retrieved from a blockchain.

In order to obtain a block with the number *0x12c*, the following query could be issued to an Angra-DB database called *eth*:

```
for $block in collection("eth")
    where $block."number" eq "0x12c"
    return $block
```

The same query could be expressed in CouchDB as a Mango query and issued to CouchDB's rest API:

```
{
    "selector": {
        "number": {
            "$eq": "0x12c"
        }
    }
}
```

And using the MongoDB method:

```
db.eth.find( { number: "0x12c" } )
```

It is important to address that in this context, both CouchDB and MongoDB queries are meant to be issued using a javascript environment, while AngraQL is meant to be a language on its own. That said, we are not interested in the capabilities of altering the results using list operations of the languages, but in its ability to express queries to be performed in the database.

In terms of readability both AngraQL and MongoDB's method can be easily understood, while Mango has a unique syntax in relation to other query languages. Moreover, AngraQL can be more readily understood by users that have previous experience with SQL, since its structures can be tied to SQL's SELECT FROM WHERE structure.

Taking writability into account, MongoDB's is the most concise of the three, while retaining readability. AngraQL's structures requires more typing to perform a simple search query. Mango is a middle ground between the two.

However, if we alter the query to search for two different values in two different fields, it is possible to notice that the readability of both Mango and CouchDB's are affected while AngraQL still retains its easily understandable code:

```
AngraQL:
    for $block in collection("eth")
        where $block."number" eq "0x12c" or $block."hash" eq "0x06aa5b..."
        return $block

Mango:
    selector": {
        "$and": [
            {
                "$number": {
                    "$eq": "0x12c"
                }
            },
            {
                "$hash": {
                    "$eq": "0x06aa5b..."
                }
            }
        ]
    },

MongoDB:
    db.inventory.find({$or: [ { number: "0x12c" }, { hash: "0x06aa5b..." }]})
```

MongoDB's retained its good writability and readability, while CouchDB's Mango required more typing than the other two to add a new filter. However, as said before, AngraQL retained almost all of its properties, its as much readable as before and required little code changes to add the new filter.

Just as CouchDB's and MongoDB's queries can use javascript variables to store query results and other data, AngraQL also provides a variable declaration statement. This feature improves the readability of its code, since otherwise hard coded values and structures could now be referred by a readable name. In addition, the variable declaration feature improves the writability since it also allows the programmer to refer to bigger structures by smaller names.

The last criteria of the query languages, the reliability is where AngraQL is mostly lacking in comparison with the other two languages. Since they both are mainly javascript scripts, they can use that language's features of exception handling to handle unexpected results. Although exception handling structures such as *try...catch* blocks are specified in JSON Query Language, they were not implemented in the current language version. However, type checks are implemented in runtime, just like javascript.

Chapter 6

Conclusion

Just as other iterations of the Angra-DB project development, this work exercised the usage of various Erlang components while also explored the various challenges that are faced during the development of a NoSQL database, such as scalability and response time issues.

This work resulted into a well established architecture to the Angra-DB project for the development of new query languages as well as the improvement of the AngraQL. By providing a common API to be consumed by different query language modules in order to perform database queries, the project's principles of modularity and extensibility were respected.

Even though some performance issues were not addressed in this first implementation, the performance tests showed satisfactory results regarding the response times, specially in the retrieval process, which was more thoroughly researched when designing the architecture's components. However, the results in the insertion performance tests into the database showed a significant regression from the results obtained in previous works, which indicates the possibility of new studies regarding the concurrency control in database writes. As the last goal of this work, the developed query language presented an easy to use syntax inspired in the JSON Query Language specification, which, although still limited, provides useful querying capabilities to Angra-DB's collections, a missing feature of the previous Angra-DB versions.

Following this work, other studies should be conducted in the Angra-DB project in order to further improve the current architecture. Besides the concurrency control just mentioned, other approaches should be researched in order to control the merges of the indexes. In addition, in order to enable the same retrieval operations as other DODBSs, such as range queries, further improvements must be applied to the inverted index module and, thereafter, changes should be necessary on the common query API.

Finally, regarding the query language building, the implemented architecture enables

the development of domain specific query languages, which would allow for queries optimized for certain data domains, such as the blockchain domain briefly explored in this work, to be performed into the database's collections. This way, allowing the development of multiple types of applications using Angra-DB.

Bibliography

[1] Moniruzzaman, A B M and Syed Hossain: *Nosql database: New era of databases for big data analytics - classification, characteristics and comparison.* Int J Database Theor Appl, 6, June 2013. 1, 4, 5

[2] McAfee, Andrew and Erik Brynjolfsson: *Big data: The management revolution.* Harvard business review, 90:60–6, 68, 128, October 2012. 1

[3] Singhal, Amit and I Google: *Modern information retrieval: A brief overview.* IEEE Data Engineering Bulletin, 24, January 2001. 1, 11

[4] Zafar, Rashid, Megat Zuhairi, Eiad Yafi, and Hassan Dao: *Big data: The nosql and rdbms review.* May 2016. 1, 4, 5

[5] Lourenço, João Ricardo, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino: *Choosing the right nosql database for the job: a quality attribute evaluation.* Journal of Big Data, 2(1):18, Aug 2015. `https://doi.org/10.1186/s40537-015-0025-0`. 4

[6] Tudorica, B. G. and C. Bucur: *A comparison between several nosql databases with comments and notes.* In *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pages 1–5, June 2011. 4

[7] Jose, Benymol and Sajimon Abraham: *Exploring the merits of nosql: A study based on mongodb.* pages 266–271, July 2017. 4, 6

[8] DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels: *Dynamo: amazon's highly available key-value store.* In *IN PROC. SOSP*, pages 205–220, 2007. 4, 5

[9] Leavitt, N.: *Will nosql databases live up to their promise?* Computer, 43(2):12–14, Feb 2010, ISSN 0018-9162. 5

[10] B. Sundhara Kumar, K, Srividya , and Mohanavalli Subramaniam: *A performance comparison of document oriented nosql databases.* pages 1–6, January 2017. 5, 6, 7

[11] Croft, Bruce, Donald Metzler, and Trevor Strohman: *Search Engines: Information Retrieval in Practice.* Addison-Wesley Publishing Company, USA, 1st edition, 2009, ISBN 0136072240, 9780136072249. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

[12] Sharma, Shruti and Parul Gupta: *The anatomy of web crawlers*. International Conference on Computing, Communication and Automation, ICCCA 2015, pages 849–853, July 2015. 14

[13] Wood, Gavin: *Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07)*, 2017. `https://ethereum.github.io/yellowpaper/paper.pdf`, Accessed: 2018-01-03. 43

[14] Tschorsch, F. and B. Scheuermann: *Bitcoin and beyond: A technical survey on decentralized digital currencies*. IEEE Communications Surveys Tutorials, 18(3):2084–2123, thirdquarter 2016, ISSN 1553-877X. 43, 44

[15] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf*. 44

[16] Johansen, Stefan: *A comprehensive literature review on the blockchain technology as an technological enabler for innovation*. November 2016. 44

[17] Vasconcelos, João Carlos Moura de Mendonça: *Avaliação de performance comparativa do angra-db com uso do yahoo! cloud serving benchmark*. Trabalho de Conclusão de Curso (Licenciatura em Ciência da Computação)—Universidade de Brasília, Brasília, April 2018. `http://bdm.unb.br/handle/10483/19806`. 47

[18] Sebesta, Robert W.: *Concepts of Programming Languages*. Pearson, 10th edition, 2012, ISBN 0273769103, 9780273769101. 49, 50

Appendix A

Erlang

Erlang is a programming language used to build massively scalable systems with requirements on high availability. In fact, Erlang is a functional programming language that has built-in systems to provide distribution, fault tolerance and concurrency.

A.1 Data Types

In Erlang, each piece of data of any type is called an Erlang *term*. Erlang terms can be any one of the following types:

- Number: any numeric literal, which means it can be an integer or a float.
- Atom: a literal, a constant with a name. Similar to Prolog atoms. Their name must start with a lower case character otherwise they must be enclosed in single quotes.
- Bit Strings and Binaries: represents an area of untyped memory. This type of variable can be manipulated using the bit syntax. A bit string that has a number of bits divisible by eight is called a Binary.
- Fun: a reference to a function. It is how a lambda function is called in Erlang.
- Tuples: represents a combination of terms with a fixed amount of elements.
- Lists: represents a combination of terms with a variable number of elements.

There are other data types that represents a variety of Erlang constructs, to learn about all of them, refer to Erlang Docs[1].

Some common data types that are implemented as primitive types in other high level programming languages are just special cases in Erlang. That's the case of Strings, which

[1] `https://www.erlang.org/docs`

are just lists of integers that all of the elements are printable numbers. Booleans are also represented as the atoms *true* and *false*.

A.2 Functional Programming

As Erlang is a functional programming language a lot of its features are common in other functional languages such as Haskell and Lisp.

First of all, Erlang programs are organized in modules. Each module contains definitions of functions and a subset of these functions can be exported to be used by other modules. For instance, to declare a module containing the two functions *foo* and *bar*, we can write this code to a file:

```
-module(my_module)

-export([
  foo/0
])

foo() ->
  Bar = bar(),
  Bar.

bar() ->
  my_atom.
```

In the code above, only the function *foo* is exported and can be used outside the module *my_module*, the */0* suffix indicates the arity, i.e. the number of arguments, of the function foo (as overloads are possible).

In Erlang, variable names start with upper case characters and cannot be reassigned. This is a feature of Erlang which tries to guarantee that each function does not produce any side effect for the application. The line *Bar = bar()* is a variable assignment in which the returning value of the function *bar* is assigned to the variable *Bar*. Also, Erlang is a dynamically typed language, therefore, the data type of the variable *Bar* will be decided in runtime.

The function definitions can be a sequence of any number of statements, each one of them separated by a comma. Additionally the function's returning value is the value returned by its last statement. For instance, the function *bar* is composed of only one

statement that returns the atom *my_atom* and the function *foo* is composed by two statements, the first being the assignment explained above and the second statement just returning the variable *Bar*'s value.

As any functional programming languages, the variables are bound to values through pattern matching. This means that the pattern in the left hand side of an assignment is compared to its right hand side term, if the pattern matches, then the variables in the pattern are bound to their corresponding values in the right hand side term. For example, in the expression below:

```
{X, Y} = {1, "My String"}
```

The pattern in the left hand side *{X, Y}* looks for a tuple of two values in the right hand side term *{1, "My String"}*. As they both match, the variable X is bound to the first term of the RHS tuple while the variable Y is bound to its second term, resulting in X = *1* and Y = *"My String"*.

Functional languages heavily rely on pattern matching to make its code more concise and readable. Besides the use case presented above, pattern match could be used to specify multiple function clauses, each one of them responsible to match a specific pattern. As an example, in the code below:

```
foo(string) ->
  "String";
foo(boolean) ->
  "Boolean".
```

If the parameter of this new version of the foo function matches the atom *string* it outputs the string *"String"*, but if it matches the atom *boolean* then it outputs the string *"Boolean"*. If the parameter does not match any of these atoms, a runtime error is thrown.

Also, another functional programming feature that erlang supports is the use of high order functions. High order functions are functions that receive or return other functions. Using the fun syntax in erlang, it is possible to declare lambda functions, which could be passed to other functions or outputted by a function. To declare a fun, we can use the syntax:

```
DoubleOf = fun (X) -> X*2 end.
```

This creates a lambda function which takes one argument X and multiplies its value by two, and assigns a reference to this function to a variable called *DoubleOf*. It is now possible to call the function referenced by *DoubleOf* using:

```
DoubleOf(3)
```

Which will output 6. It is now possible to pass the function referenced by *DoubleOf* to other functions, that will be able to call then as they need. High order functions are useful to let some of the behavior of a function up to the caller, creating a new layer of flexibility.

A.3 OTP

OTP is a set of Erlang libraries and design principles that provide a middleware to develop distributed, fault-tolerant systems. In OTP, not only it is possible to find modules implementing distributed databases and distributed applications but more importantly, it is also defined the standard principles of how an Erlang application should be built.

The core abstraction in OTP is that of a Behaviour. In OTP, behaviours are a way to separate generic logic of a module from its specific use. For instance, a functionality such as a server in a client-server relationship shares some pieces of functions and abstractions from every application like obtaining the requests, sending the replies and maintaining a server state, in other words, these functions are the generic part of a server. What to do with the received request, how the response should be generated and how the state should be altered is almost always a task that varies across different applications, therefore, these functions are the specific part of the server.

That said, in Erlang/OTP, a behaviour is the generic part of the functionality (the receiving/replying functions in the server example) and defines a set of functions that should be implemented by a callback module that implements that behaviour in order to provide the specific part of the functionality (the processing of the request and the response generation in the example). This way, it is possible to separate a well-behavioured and well-tested common code from the more error prone application logic.

OTP already provides some useful and well known behaviour modules like servers, finite state machines and event handlers, but it also provides some Erlang specific behaviours that help the development of applications in a standard manner, such as supervisors and applications.

As lightweight processes are core concepts to Erlang as a way to develop concurrent applications, a process structuring model is necessary in order to create a standardized

way of developing said applications. The abstraction used by OTP to organize Erlang applications is called the supervision tree. This model is based in the concepts of workers and supervisors and aims to build fault-tolerant concurrent applications.

As the name denotes, the supervision tree has a tree structure where each leaf is a worker process and the internal nodes, i.e the nodes that are not leaves, are supervisors, as shown in Figure A.1. Each type of node has a specific role to to play in an Erlang application:

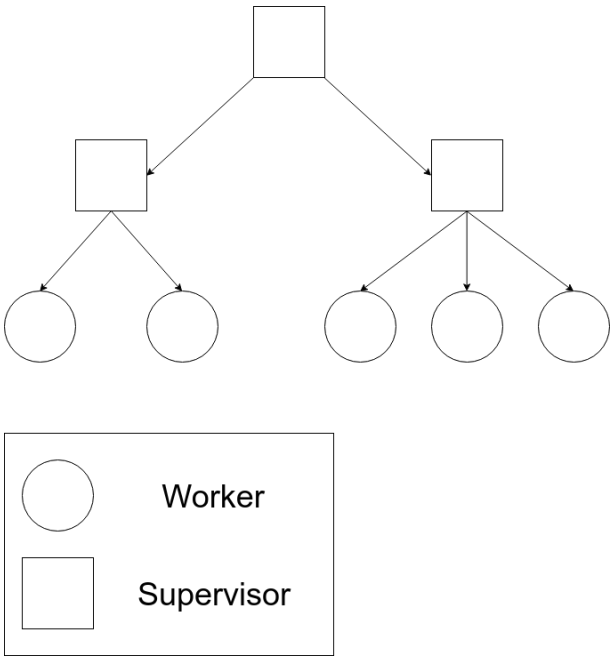

Figure A.1: Example of a supervision tree.

- Supervisors are processes that have the purpose of monitoring its child processes, checking if they have died (stopped working) and restarting them in case they did. A supervisor can have multiple children and it is possible to choose along some restarting strategies.
- Workers are the processes that actually handle the application logic. There are some default worker processes like servers, (modules that implement the *gen_server* behaviour) or finite state machines (the ones that implement the *gen_statem*).

As hinted above, these process types can be used by implementing behaviours provided by OTP. This way, it is possible to write worker and supervisor processes without the need to delve into the supervision tree logic. For instance, it is possible to implement a supervisor behaviour by writing the following callback module:

```
-module(ch_sup).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link(ch_sup, []).

init(_Args) ->
    SupFlags = #{strategy => one_for_one, intensity => 5, period => 10},
    ChildSpecs = [#{id => ch3,
                    start => {ch3, start_link, []},
                    restart => permanent,
                    shutdown => brutal_kill,
                    type => worker,
                    modules => [cg3]}],
    {ok, {SupFlags, ChildSpecs}}.
```

The supervisor behaviour just needs to know which child processes it must instantiate and how it should treat its children. In the above code, the callbacks *start_link* and *init* are needed to implement the supervisor. The *start_link* callback provides a way to instantiate this supervisor, it should define properties of the process like its name and which arguments should be passed to the *init* function. The *init* callback is where the children of the supervisor are specified and where the supervisor behaviour will be configured.

SupFlags is a variable that specifies which supervising restarting strategy the supervisor will use and how many restarting attempts it should try before deciding to kill itself. In this case, the most common restarting strategy, *one_for_one*, will be used, meaning that every time a child dies, it will be restarted without interfering with other children. Some other restarting strategies are:

- *one_for_all*: if one child dies, all other children are terminated and restarted.
- *rest_for_one*: every child in the children list that are after the dead child is restarted.
- *simple_one_for_one*: it is a simplified version of *one_for_one* where every child is of the same type and is added dynamically

Moreover, if 5 (the intensity property) child restarts occur in a time interval of 10 seconds (the period property), then this supervisor should give up and kill itself, letting its parent process deal with the consequences or killing the application if its the root process.

Next, a list of children specifications are assigned to the variable *ChildSpecs*. Each specification defines how to deal with a specific child. In the example above, there is only one child which has its properties set as shown below:

- id: an identifier for this child.
- start: specifies the function to be called in order to instantiate this child process. It is a tuple in the format *{ module, function, args }*, in this case, the module is called *ch3*, the function is called *start_link* and receives no arguments, as it is passed an empty list
- restart: can be the atoms *permanent*, which means the process should always be restarted, *temporary*, meaning that it should never be restarted, or *transient*, in the case it should be restarted only if it terminated in an error.
- shutdown: specifies how the child shutdown should be handled, it is deadline on how long the process should take to terminate itself gracefully. If the value *brutal_kill*, is passed, then the child is terminated immediately
- type: if the child is a worker or a supervisor
- modules: a list of only one item that contains the callback module of the child behaviour.

And finally, these parameters are returned from the *init* function to the supervisor behavior module. This example code shows how behaviours are useful. Just by specifying a set of properties to a generic module, it is possible to use its capabilities with little effort in writing code.

Finally, another major behaviour that OTP provides is the application behaviour. This behaviour provides an abstraction for starting, managing, shutting down, updating, handling dependencies and configuration of Erlang apps. In essence it is one more way to standardize Erlang apps structure. For instance, the application behaviour has a standard way of starting a supervision tree, also providing mechanisms for safe control of Erlang apps. The feature of configuration in the application behaviour is of utmost importance to Erlang apps since it provides an easy way of changing application behaviour without changes to Erlang code. Finally, structuring Erlang apps around application behaviours enable the usage of numerous tools that expects the application structure to be followed.