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

Design and Implementation of a MapReduce Architecture for AngraDB

Fábio Costa Farias Marques

Assignment presented as parcial requirement
for Computer Engineering course conclusion

Supervisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

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

Design and Implementation of a MapReduce Architecture for AngraDB

Fábio Costa Farias Marques

Assignment presented as parcial requirement
for Computer Engineering course conclusion

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

Prof. Dr. Eduardo Alchieri
CIC/UnB

Edward Ribeiro
Senado Federal/PRODASEN

Prof. José Edil Guimarães de Medeiros
Coordinator of Computer Engineering course

Brasília, January 30th, 2019

Dedication

This assignment is dedicated to my mother, father and sister, for their confidence on me, helping me on every battle I had, always fighting with me as one. Our love united us and nothing will ever break it. You are my everything.

It is also dedicated to my friends Déborah Araújo, Fernando Nunes, Bruno Rodrigues, Ismael Medeiros and Bruno Bergamaschi which knew each hard time I had, helping me on every possible manner, trying to make me happier, move forward and solve all my problems. I will never forget you, we have been bonded by heart, so for me you are family too.

Acknowledgements

I would like to offer my special thanks to my family, who always supported me in all projects I had, investing on me all that time and money, trusting on my capability having not a single doubt on it. I could not ask for a better family, the strong love and confidence we have for each other makes me fell confident to do everything. I am sorry for the long noisy and working night, but it is all paid off right now.

I would like to express my deep gratitude to Prof. Dr. Rodrigo Bonifácio, he was always pro-active, inviting me for new projects, wanting for me to develop new skills and offering all kind of possibilities on the scientific scenario. There would not exist a better mentor than you, so I appreciate all the confidence deposited on me and my potential. I hope we keep maintaining this productive partnership on.

My gratitude is also for all teachers I had in my life, specially on Universidade de Brasília, for their attention and excellence on the art of transmiting knowledge. Without you the human kind would not reach anywhere but the extinction.

I wish to acknowledge the support provided by all my friends and those who care for me, knowing that you wishes the best for, even from far away, gave me the strength I needed to reach this victories in life I'm having. I know I can vanish sometimes, not staying in touch in a while, but I acknowledge each one of you that always cheered for my success.

I would like to thank the existence of the Stack Overflow, without the contribution of the hundreds of thousands questions successfully answered this work would not be even close to exist.

I also want to extend my special thanks to the AngraDB group, which are real friends for me. We worked together side by side, investing out time and assets on this project, I wish that my victory be yours too, because we did it! Specially I would like to express my gratitude to Ismael Medeiros, the person who implemented the distribution feature for AngraDB. Without his work, the MapReduce feature would not be possible.

"*If I have seen further it is by standing on the shoulders of Giants.*", *Isaac Newton*

Abstract

The AngraDB is a NoSQL document based database developed by Universidade de Brasília students, whose main initial goal was to explore the Erlang programming language features. The database was developed to support a wide set of documents but there aren't any mechanisms for processing large scale data sets, which is a desirable functionality for those kinds of databases. That is why a feature based on Google's MapReduce is being developed to fulfill this demand. The AngraDB MapReduce uses the distribution facilities of Erlang to process the documents in a given database using user provided functions. Taking advantage of the available concurrency processing it allows the users to extract useful and valuable information from the data stored in a flexible way using the Erlang language. The actual architecture of the feature is distributed and was tested using virtual machines and local networks, processing documents in time that it is faster than the traditional sequential processing, showing the success of the technique.

Keywords: MapReduce, distribution, data science, Erlang, database

Contents

List of Figures

List of Tables

Acronyms

CAP	Consistency, Availability, Partition Tolerance.

Chapter 1

Introduction

AngraDB[1] is a document based database developed by Universidade de Brasília written in Erlang programming language, a project that aims increasing the domain of this functional language. The database was designed to support huge amounts of data, but no custom procedures were developed to be applied on the stored documents. The MapReduce is a technique developed by Google engineers create to process data sets in parallel, processing each document with custom user functions, extracting information in reasonable time.

Since its debut, it has been extensively studied and applied with different implementations, such as Prof. Dr. Ralf Lammel's which was developed in Haskell [2]. In his article, he details a MapReduce model explaining the main concepts used and the differences from the Google's model. Lammel is a reference in functional language community and his model showed how MapReduce programming model is compatible with functional languages.

Another reference implementation that uses MapReduce is Hadoop[2], which is a framework for processing large amounts of data. Ghazi and Gangodkar document in a deeper way the framework in their work [3] and how important MapReduce is for those kind of processing being a powerful tool in this Big Data scenario.

Real-world applications when dealing with massive amounts of data cannot rely on traditional data processing techniques, such as sequential processing. Most of the time, the whole data to be processed can't be even loaded into memory, that's why techniques that takes advantage on distribution to use parallel processing, such as MapReduce [1], are being used extensively to fulfill the real-world demands.

Looking to evolve AngraDB for the Big Data context, in order to allow and be able to process large data sets, a feature based on Google's MapReduce but adapted to the application context was developed and will be documented in this work. The feature was

[1]See https://github.com/Angra-DB

[2]Hadoop, see https://hadoop.apache.org/

developed in Erlang and rely on the modules for distribution that are available on the latest version of AngraDB, using all the nodes to execute the operations needed in the data extraction process.

This first version will handle smaller and simpler tasks and will store its results in memory, focusing on understanding the MapReduce concept and principles.

1.1 Objectives

The main goal of this work is to design and implement a MapReduce architecture on top of AngraDB. More specifically, to achieve this goal we:

- Develop an AngraDB feature that creates a sorted array with the key and location of its document for the databases, following the modularity and flexibility principles.
- Design and implement an early version of the MapReduce feature, capable of work on several environments and databases and focusing on obtaining the expected results of a MapReduce tasks.
- Evaluate the proposed solution with different number of MapReduce nodes and several databases with different number of documents, evaluating the results obtained and the performance.

1.2 Methodology

Initially a MapReduce literature review was made, looking for successful models [4][3][1][2] [5] and there was a conception about of what aspects the MapReduce feature needed to be implemented on AngraDB. The first one, as defined previously, was develop a tool for creating and maintaining a file which holds an array of references for the stored documents, allowing a faster document retrieval when using the document indexes.

The architecture of the feature had its interfaces and execution steps defined following mostly [1]. The implementation was made using the distribution in the current AngraDB version and the Erlang/OTP distribution, in order use the facilities available on the standard Erlang library behaviors.

In order to check the results and evaluate the performance of the architecture, tests were conducted using several environments and datasets, looking for recreating real-world workloads and configurations. Virtual machines with the same hardware configuration were used in the tests. The results in different conditions were compared and the the conclusion enforced on [1], that use a distributed environment for processing a task improves the performance when compared to sequential processing even with the procedure cost.

1.3 Dissertation organization

The next chapter describes the AngraDB project in detail, highlighting some of its architectural decisions. Then, the Chapter 3 presents a review of the Erlang language and distribution concepts. Next, the implementation of the MapReduce feature is explained, showing the decisions made, its architecture, the tests performed and the results achieved. The last chapters expose some of the development issues faced, the future works and the conclusion.

Chapter 2

AngraDB Project

The Angra group was created by Professor Rodrigo Bonifácio and Universidade de Brasilia students to study functional languages and current technologies that are not taught on undergraduate subjects so deeply. The word "Angra" is from the Tupi-Guarani culture, a brazilian indian dialect, which means their fire goddess. The name was chosen to honor the brazilian culture, place where the project started.

After studying world-wide demands for functional programming skills, the group had decided to adopt the Erlang language for all its benefits. In order to get used to the language, the Angra Database project was started. With some guidance of LightBase[1] company and analyzing the growing of the NoSQL, the group found out that the development of a document based database could improve the group skills and would make the members stay in touch with the latest NoSQL technologies and tendencies, using the Erlang language to support along the project.

2.1 NoSQL

The terminology "NoSQL database" in this context means that the database is not relational, because it does not necessarily supports the SQL ("*Structured Query Language*") which is a language developed in 1974 to create an interface to the relational databases using relational algebra concepts, allowing the user to execute operations on the database, and does not organize the data in tables that are composed by columns, each one having it own type, which must be numbers, booleans, strings, even a whole object, such as the relational databases. The group of columns in a table defines the object that will be stored.

The columns in a table defines the structure of the data that can be stored, that's why the data stored in relational databases are often called as "structured". Objects stored

[1]See http://www.lightbase.com.br/

defines its keys to identify and relate objects. They usually have a property called “Id”, a unique value in the table used to identify it. The relations between objects are also represented with keys, that are columns used to store a reference to an object in other tables.

An operation in a relational database is called transaction and it is often composed by several SQL commands. Relational databases uses “ACID” transactions. The acronym means:

- **Atomicity:** demands that the whole transaction is successful, if any operation fails, all operations are rolled back even successful ones
- **Consistency:** all transactions leave the database in a consistent state, following the defined rules an constraints
- **Isolation:** each transaction is independent and executes as if there was not any other transaction running. Isolation guarantees that their executing will result in the same result that would have if they are executed sequentially
- **Durability:** all changes must persist if the transaction is successful

The SQL has fulfilled most of the demands since the 70’s, but with the growth of the amount of data that must be stored, the different ways that the data can be related and the need of performance increase for certain applications generated the need of new ways of persisting data. In this context, the non-relational databases, mostly called NoSQL, had their development started. Different from the SQL, they follow the “BASE” concepts, which are:

- **Basically Avaiable:** guarantees availability within the CAP theorem (CAP theorem will be explained in the next chapter)
- **Soft State:** the database state can change even without the execution of any operation
- **Eventual consistency:** data will converge to a consistent state after some time

Those concepts shows that a NoSQL database data and state is not constrained as a SQL one, aspect that the database developers use to create different mechanisms of searching and indexing, as example. One example is a document based database, which is created to store documents. NoSQL databases work with a key-value mechanism, which identifies each value to be stored with an unique key, that can be used for retrieval, update and delete operations. It also allows the developers to create custom native operations.

For AngraDB, as example, an inverted indexer was created to provide statistics of the words in the documents stored.

Executing a query in the everyday data generated by Google or Facebook would take much more time if the data were stored on SQL databases, that's why they are also looking for alternative systems to deal with their Big Data problems, such as *Hive*[2], or even developing theirs, such as *Google's BigTable*[3]. For all these applications and scenarios, NoSQL is an important matter nowadays.

2.2 AngraDB structure

This section explains the structure of the AngraDB before the distribution mode, in order to show how it works. AngraDB in its higher state, is an Erlang application. All module names usually start with the prefix "*adb*", which means "AngraDB", and the final part briefly explain the files objective, for example "adbtree" is the module that creates the B+Tree persistence or "adb_server_sup" which is the server supervisor file.

The AngraDB module that handles the requests for the database is the "adb_server", it is responsible for handling a connection with one client. Every time a client connects to an "adb_server" instance, the node automatically instanciates other "adb_server" node that will be available for a new client connection. The nodes that are handling client demands will simply die after the connection shutdown. This design decision was made to improve availability, avoiding refusing connections even when responding to other clients.

Some clients were developed to interact with the AngraDB, the most important are the TCP client and the rest client. As their names explain, the TCP client allows the clients to interact with the AngraDB using the TCP protocol and the rest client permits interaction. There are not any authentication module providing access control to the databases and the available operations that can be executed on AngraDB are:

- ***create_db*:** creates a database with the name received as argument
- ***delete_db*:** deletes a database with the name received as argument
- ***connect*:** connects to a database with the name received as argument
- ***save*:** saves a document received as argument into the database, returning the id
- ***save_key*:** the database will have consistency after a time having no operations executed

[2]Hive, see https://hive.apache.org/

[3]BigTable, see https://cloud.google.com/bigtable

- ***lookup*:** retrieves the document from the database with the given id received as argument
- ***update*:** updates the document related to the received id
- ***delete*:** removes from the database the document related to the received id

The first 3 operations can be executed at the moment that a connection to the AngraDB is made, but the last 5 can only be executed after successfully connecting to an existing database.

AngraDB has also the principle of modularity, which means that each module for the project was created to work as a black box with a set of interfaces, making it easy to interact with other modules and to develop new ones. Following this principle, AngraDB is highly configurable and adaptable and easy to fill the needs of it users. The supervisor files follow the Erlang configuration, which has a large documentation, turning them easy to configure.

Other facility of AngraDB is the persistence interface. Any storage engine in order to interact with Angra must simply implement this interface to be one of the persistence methods. There are 3 current persistence methods, as can be seen on Figure 2.1 already implemented: ETS tables (documents are stored in Erlang ETS tables), HanoiDB persistence[4] and AngraDBTree (or ADBTree).

AngraDBTree is one persistence method completely develop by the Angra group with its principles based on CouchDB[6]. This solution is a B+Tree developed in Erlang to hard persist in the file system the data and it was one of the first projects of the Angra group, being tested and used for almost 3 years. Since it was developed in Erlang the interface and feature development for this module is easier.

AngraDB is flexible and can store a wide set of documents. Distribution mode is being developed and other features as well, but there isn't any massive processing that can be applied to the documents in the database, so the MapReduce comes as an important addition to the project, making it even more ready to the real-world workloads.

[4]HanoiDB, see https://github.com/krestenkrab/hanoidb

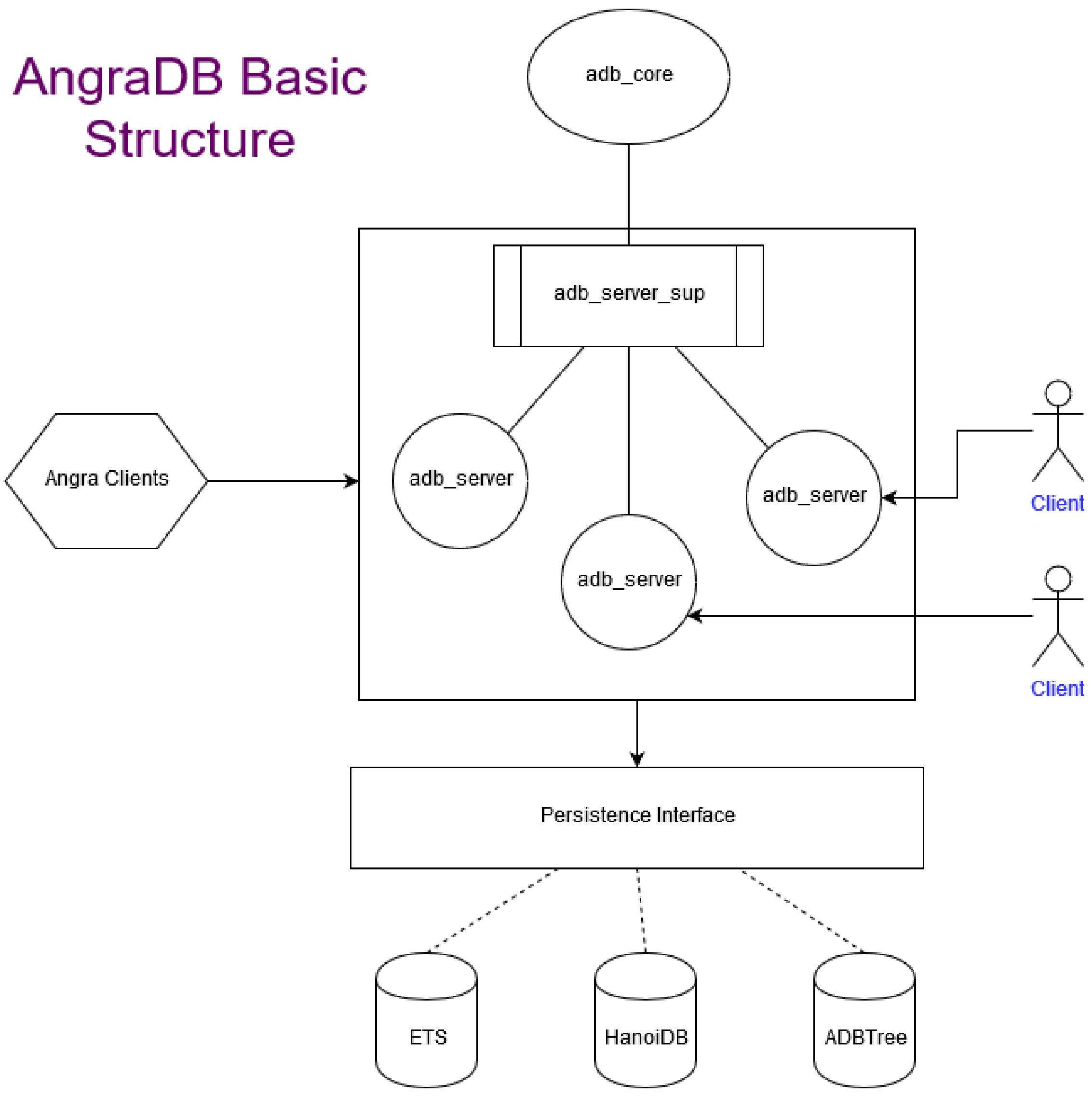


Figure 2.1: Basic structure of AngraDB and some of the persistence methods..

Chapter 3

Erlang and Distribution concepts

3.1 Erlang

Erlang is a resourceful and useful language, but its concepts and structures may not be the easiest to understand. This chapter is meant to explain the constructions and some of the key features of the language and, given our need for a distributed environment and concurrent programming, why the language was chosen for being the main tool in the AngraDB project.

3.1.1 Erlang history

Erlang history started in Ericsson, which is a swedish company founded in 1876 by Lars Magnus Ericsson[1]. The invention of the telephone and its arrival in Sweden were the game-changing factors to Ericsson history, because after Lars started repairing the new tool, he realized that this invention could be improved to connect the world in way never seen before, so he started to create his own and innovative versions of the telephone. The results were so impressive, that the high quality of its products would establish what is called "Swedish pattern" for telephones and turning Ericsson into a world class company in a few years.

After years producing telephones, the company started to invest in infrastructure and information technology development, becoming a leading technology and innovation company in the 20th century. Today Ericsson has 40% of world's mobile traffic on its network and is a key actor on 5G mobile network, IoT and cloud solutions development.

Back to the 80's, Ericsson started a project to create a new programming language adapted to the next generation of telecommunication products. In order to create a language with productivity gains several studies with more than 20 different programming

[1]See https://www.ericsson.com/en/about-us/history/shaping-history

languages were made and after that they reached the conclusion that using a very high level symbolism, for example Lisp[2], Prolog[3] and Parlog[4]; would result in those gains. After studying those languages, the main features observed were the concurrency and error recovery primitives, the lack of existence of back-tracing and the increase of the granularity of the processes. Along concurrency and error recovery, those desirable features were chosen to compound the new language called Erlang. Different from other languages, Erlang requirements were given by the need of the market.

The language was first Ericsson's property language[5], developed and used just for internal projects. The year of 1993 was very important for Erlang, because an update which allowed Erlang homogeneous installation on heterogeneous hardware made the distributed mode not a possibility, but a reality for the language. Beyond this, that year was when Ericsson decided to sell implementations externally and to make a separate organization in charge of maintaining, developing and supporting the language.

Today most of the 3G phones software, most of 40% the mobile network infrastructure and some databases as *CouchDB*[6] and *Riak*[7] are written in Erlang, Facebook uses the language to control servers and even in the game development industry the language is used the build services, such as WhatsApp[8] and RabbitMQ[9], facts that show the consolidation and the success of Ericsson's project.

One of the most used Erlang frameworks is OTP, that was developed and released in 1996. It provides a wide variety of powerful Erlang tools and libraries to facilitate developing systems. It's so common to use OTP features, that is a good and recommended practice to install the Erlang/OTP distribution.

3.1.2 General features

Erlang is a declarative language, which means that statements try to describe the situations to be computed. In the given example below, it is shown that the function to be executed will depend on the input, that have to match one of the defined conditions or an error will occur, showing by now the extensive use of pattern-matching in the language. This technique can be used with different types, including to match byte sequencies with defined length, making the interaction with telecom protocols easier for example. Erlang has its functions as first order objects class, meaning that functions can be bounded to

[2]See https://lisp-lang.org/
[3]See http://www.swi-prolog.org/
[4]See http://www.parlog.com/en/index.html
[5]See https://www.ericsson.com/en/about-us/history/shaping-history
[6]See http://couchdb.apache.org/
[7]See http://basho.com/products/
[8]See https://www.whatsapp.com/
[9]See https://www.rabbitmq.com/

variables, stored in lists and received or returned by functions (high-order functions). The example below was taken from [7], p. 38, Example 2-1, to exemplify functions overcharge and pattern-matching.

```
area({square, Side}) ->
  Side * Side ;
area({circle, Radius}) ->
  math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
  S = (A + B + C)/2,
  math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
  {error, invalid_object}.
```

Erlang has a fully concurrent environment, since threads are not provided and memory is not shared by processes, each process has its own memory, heap and stack, and other processes cannot make a direct interference. Processes still can communicate to each other using Erlang messages, which can be used to asynchronously send any kind of object and type to other process. The message must be collected from the mailbox in order to be read.

Erlang processes when created do not spawn an OS thread for each one, instead the Erlang Virtual Machine ("VM") create, handle and schedule them within its environment, turning those processes "lightweight". This fact shows that the processes creation, which takes microseconds to complete, does not depends on the OS or hardware, demonstrating the scalability and flexibility of the language: once installed, any kind of program can be executed.

The language has the concept of variables, but they a special property: the "*single-assignment*", which means Erlang allows any amount of them in a program, but they cannot have their values changed once some value have been attributed the that variable, resulting in an error when tried. This kind of variable assignment was chosen for guarantee that the program will always have the same behavior when executed, with no side-effects. The recursion is also a key-strategy on Erlang applications architectures, the language was designed to have a huge support for it, allowing even the use of pattern-matching when calling functions, i.e, a function clause will only be executed if the execution request meets the exactly function signature, resulting in an error if no meeting function signatures were found.

Location transparency is a feature that makes Erlang node communication much easier, because the network and machine PID are not concerns of the application, the Erlang VM, when configured to support this feature, uses Erlang PIDs in order to communicate to other nodes, providing those ids to client use.

3.1.3 Behaviours

Behaviours are Erlang modules with functions and definitions that can be implemented by other modules. Behaviours can also have interfaces, which are method signatures that require implementation by the module which is using the behaviour.

One of the most used behaviors is the “gen_server”, which is a module that has all functionalities to create a server. After Ericsson programmers realize that a good amount of code was repeated on creating servers, they decided to group all the common code into a behavior and created some interfaces the allow who uses the “gen_server” to control how clients interact with the server, what to do on server start, shutdown and on code update.

3.1.4 Supervisors

Erlang has several design principles and one of them, which was used in this project, is the Supervisor. This principle is an Erlang behavior, it means an interface provided by the language with many functionalities. The processes created by code that implemented the Supervisor behavior have just one job: supervise its child nodes, checking if they are alive, restarting them when itś needed or killing them, among other duties.

All Supervisors can be configured using supervisor flags, used to set its restart strategy (the available strategies are defined on Erlang documentation), “one_for_one” and “one_for_all” are some examples, the first one works when a child node dies, making the supervisor restart only the dead node, therefore in the second strategy the supervisor kills all child nodes when a node dies and restart them all, showing that different strategies can be set depending on the pretended system. To avoid infinite loops the supervisor terminates itself and all its children if more restarts occur than the number of restarts set in a chosen amount of seconds.

The Supervisor behavior also demand a children configuration which is a list of children, defining each id, the function to be called to start the children (requesting the module, the start function and it arguments), when the child must be restarted, the function called on child termination, the type of child (supervisors may have a supervisor child) and a module identification for release module replacement.

In this project all “adb_mr” nodes have a supervisor looking for them spawned by AngraDB. The configuration used was the default for the supervisor flags, using “one_for_one” for restart strategy and making the supervisor terminate if the child nodes restart more than once in a 5 seconds interval. The only notable aspect in child is the shutdown configuration, since “brutal_kill” was chosen, the node is shut down unconditionally.

3.1.5 Generic Server or “gen_server”

The AngraDB MapReduce module was designed to be distributed, flexible, to execute MapReduce tasks, receive and respond task requests and maintain a way to retrieve results. The design made the module have similarities to the Client-Server principles, that’s why an Erlang behavior available in standard Erlang standard distribution was created to facilitate the creation of client-server applications and was chosen to be used in the implementation: the “gen_server”.

The “gen_server” is an Erlang behavior, as said before, that works as a server, maintaining a state and available to receive calls from clients. It requires some functions to be implemented:

- **“init”:** the function that is called to start the server, set initial configurations, execute one time functions before the server start and set the initial state.
- **“handle_call”:** deals with synchronous calls, receiving the request, the sender and the current state. It must return 1 among the 8 tuples defined in the documentation, making the server reply or not, or even stopping the server. It’s possible to set a timeout for the server which is dealt in the “handle_info” call or make the server hibernate. It is always needed to set the new state in the return, even if does not change (return the old state).
- **“handle_cast”:** deals with asynchronous messages, receiving only the message and the current state. The return type is similar to the previous function, but this one must never reply, just stop the server or send a tuple with a “noreply” atom.
- **“handle_info”:** deals with another kind of received messages, such as Erlang messages. Every time a timeout occurs a call to this function is made, so it’s one of this function duties treat the timeout event. It receives just a message and the current state and have the same return of the "“handle_cast” function (“noreply” or “stop”).
- **“terminate”:** this function is called whenever the previous 3 functions returns the stop type or when the parent process dies. It is last function called before server shutdown.
- **“code_change”:** a function used to upgrade code, which is not used for this project. As cited before, all “gen_server” applications have to maintain a state, which can be of any type. It is often used to store information of the server and fast access data frequently accessed when messages are received.

3.1.6 Rebar3

One more tool used in the AngraDB project is Rebar3[10]. It is a tool for creating, building and managing dependencies for Erlang systems or applications, suitable for using when building a project needs the same procedure or command. It is also an Erlang script, so its only dependency is the language support.

Rebar3 also helps with the distribution in its projects, because it provides a wide number of plugins that modify the tasks and behaviors provided by OTP and the standard libraries. Every AngraDB instance is compiled and executed using Rebar3.

3.1.7 Advantages

Erlang has a native support for parallelization, which makes it easy to work in a distribution environment. The standard library provided all the necessary code used to build the AngraDB MapReduce feature, even the servers used for handling any kinds of calls (synchronous and asynchronous) and state management. Even the Map, Reduce and Merge functions were written with Erlang language. All the features explained made possible this first version of the MapReduce feature and its documentation, which architecture will be described in the following sections.

3.2 Distribution

This chapter summarizes some key points of the distribution and it's main concepts. AngraDB as a NoSQL document based database created from scratch following the principles, documentation and experience of the current greatests databases, looking for providing a good and secure experience for users the distribution was added to the project and documented in [8]. Thus, the existence of distribution for AngraDB is one of the points that make the addition of the MapReduce feature reasonable.

3.2.1 Main Concepts

According to Ozsu[9], the concept of a distributed database is:

> "a collection of multiple, logically interrelated databases distributed over a computer network".

For the AngraDB context it means a set of instances of the database working together. Having an outstanding performance from a distributed system is not an easy job, that's why the computer scientist Eric Brewer defined a theorem[10] after studying the design

[10]Rebar3, see https://github.com/erlang/rebar3

choices of developing those kind of systems: the CAP theorem. CAP is an acronym for the following properties:

- Consistency
- Availability
- Partition Tolerance

The theorem defines that a distributed system can only guarantee 2 of the properties above, so the distributed database are usually designed selecting the ones more in line with the project purposes. Consistency is a property that guarantees that the database will try to not provide older information to the user, it will always try to provide the recent data stored, resulting in errors when its data is not latest. Availability is the guarantee of always providing a response when there are healthy nodes of the database in the network, even if the information is not the newest. This aspect can be conflicted with consistency, forcing the users often choose between one of this two properties. It's well-known that Partition Tolerance is a must have guarantee because it's the ability of handling network partition, providing strategies of recovery if some parts of the database can't be reached or retrieved. Figure 3.1 shows graphically the relationship of the properties and some examples of choices. The major influences for the AngraDB distribution mode are the CouchDB, Riak and MongoDB, all of them are NoSQL.

3.2.2 Implementation, Configuration and Relation to MapReduce

The distribution for the AngraDB was also implemented in Erlang/OTP, inheriting the facilities that comes along with the language and framework, such as encapsulation of process communication even the network's hard work, i.e, dealing directly with network protocols, parsing the responses and identifying communication errors from then network. The communication is made using Erlang messages in the highest abstraction and the TCP protocol to communicate over the network. The whole feature was splitted in 6 different modules:

- **adb_gossip_***: synchronizes and disseminates the information about the distribution management over the cluster using the *Gossip Protocol*
- **adb_server_***: client request parsing and processing into a valid database operation in a distributed way
- **adb_dist_***: request pre-processing and choosing the right node according to the sharding strategy, using the "**_partition*" modules to execute these operations

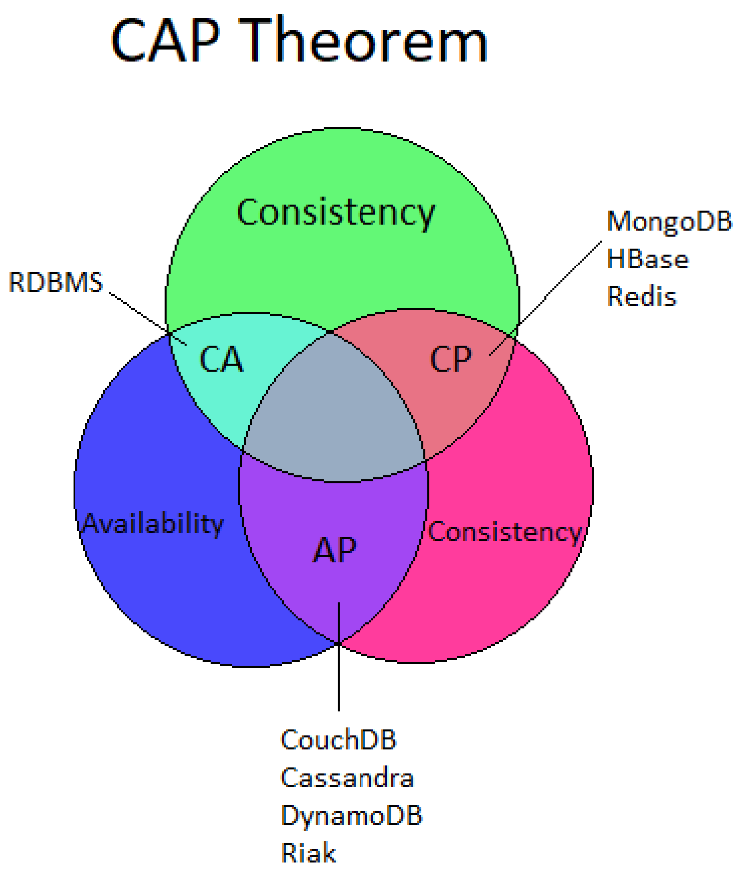


Figure 3.1: The CAP theorem's relationship and the choises of well-knows databases..

- ***__partition**: implements one of the two possible sharding strategies: Full partition (every node has all data in the database) or Consistent partition (data replicated to the nearest neighbors)
- **adb__vnode__***: forwards the request to the received virtual node and processes the responses to validate the operation
- ***__persistence**: responsible for choosing the data persistence strategy. The default option is AngraDBTree, the storage developed for the database according to the project needs

The default implementation of AngraDB defines a Sharding Strategy (divide the whole data into fragments or shards, looking for scalability improvement) which means that the stored data is split among the nodes making a node do not store the whole data in the database, but it knows where to find the data. A Ring Architecture is also used, making the nodes organized as a ring, each one having two neighbors. The data of one node is always replicated to its two nearest neighbors as a fault tolerance strategy.

The distribution is the basis of the MapReduce feature, because without it wouldn't exist any other nodes to execute operations and it would be more complex and costly to send the data to other nodes instances. Following the AngraDB philosophy, the distribution is also highly configurable and for the MapReduce it was configured to use the Full partition in order to increase the availability and to decrease the complexity of the splitting documents algorithm. There will not have nodes processing the same documents, the full partition is used to allow any instance to be able to process any documents without the need of fetching it.

Chapter 4

The MapReduce model

In 2004, Google engineers Jeffrey Dean and Sanjay Ghemawat published an article [1] documenting a way to process large data sets: the MapReduce. This chapter exposes the key points of the article *"MapReduce: Simplified Data Processing on Large Clusters"*, the most fundamental article for this graduation work, which documents the MapReduce concepts and how it is applied on Google's servers.

The MapReduce model, inspired by the map and reduce primitives present on *Lisp* and other functional languages, is quite simple and consist on 2 major steps: the map and the reduce phases. The process receives as input and also produces a set of key/value pairs, usually of different types. The map is a function written by the user and receives as input a pair and outputs an intermediate set of key/value pairs. There is a step after mapping that groups all values with the same key. The reduce function is also created by the user and receives the result of the grouping phase, merging the values of each key into a possibly smaller group and usually has zero or one output. A representation of the model can be seen in Figure 4.1.

Since Google maintains one of the most used search engines in the world, processing petabytes of data everyday along thousands of computers, the amount of data generated is so big that the usual techniques of data processing could not process the whole input in reasonable time. The need of distribution came along, because processing all the data in one machine was not possible anymore. In this context, MapReduce was emerged with a huge influence of the divide-and-conquer paradigm, which says that a huge problem can be decomposed into smaller problems easier to solve. When solving all the smaller problems, in the end, the huge problem will be solved. In this case, the data to be processed is divided instead of being treated as a huge input.

The design of MapReduce makes it flexible and allows a task to be processed in a wide set of computers, because the map process of a certain key/value pair or the reduce of the intermediate results does not need any information related to whole data, each

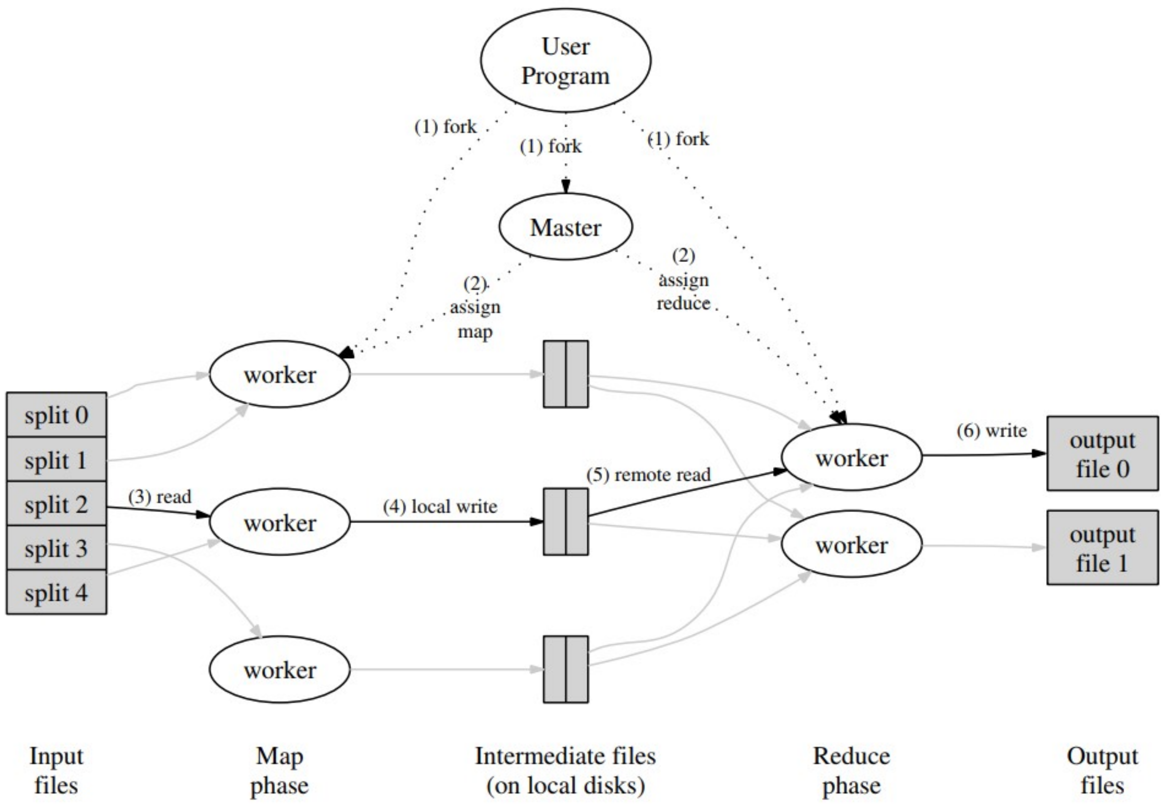


Figure 4.1: Figure from [1], representing the general MapReduce structure.

operation is atomic that's why the task can be spread to any machine that can process data. The scientists who projected MapReduce were aware of the need of distributing a task, because some tasks could not be processed in one machine due to its input size or due to an unreasonable execution time. Because of this fact a gigantic task can be parallelized among MapReduce workers in order to reduce execution time. Google documented the implementation of a MapReduce architecture into a cluster composed by hundreds or thousands machines with 2-4 GB of memory each, connected by Ethernet link up to 1 gigabit/second.

Some of MapReduce examples of application are creating inverted indexes, by mapping each word in a document into <*word, documentId*> and the reduce function merges all the occurrences of a word into a list and returns <*word, list of document occurrences*>; count the url access frequency, mapping the url into <*URL, 1*> and the merge results in <*URL, total of occurrences*>. Other advanced applications are reverse web-link graph, distributed grep, term-vector per host and distributed sort, all explained on [1].

4.1 Structures and Execution

Google's MapReduce framework works as a library for an user program, receiving all parameters needed from it in order to work properly. Users when creating the program to be executed can define as parameters the number of parts that the input will be split (M) and the number of files the whole task will output (R), number that should be greater than the number of workers, in order to increase granularity and turn task recovery easier and faster. The load balance is made dynamically, so if a worker fails, it work can be easily split to other machines.

In the MapReduce execution documentation [1] thousands of machines were used. Each machine is able to execute the programs needed in the process. As described before the user must provide a MapReduce program to be spread among all the machines, but there a two kinds of program: "master" and "worker". The worker program has only the job of executing the map or reduce functions, one of them, and storing the results on the file system. All task has a master program defined, which controls one task managing the map and reduce workers status (*idle*, *in-progress* or *completed*), input path, output path and responding back to the user when the task finishes or fails.

Dean and Ghemawat[1] also details the steps executed when a task is being processed:

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the *MapReduce* call in the user program returns back to the user code.

4.2 Fault tolerance and Refinements

When systems have to manage a huge amount of data, a fault tolerance policy is important to avoid losing the progress achieved. The MapReduce uses hundreds of machines, making real the possibility of occurring an error, that's why the fault tolerance has a key role in this model. In order to verify if the worker is alive, the master pings from time to time. When a machine that completed a map task has a failure, the task is re-executed, because the results are stored in local file system. Reduce nodes are notified when a map node fails, to read its input data from the new node elected to execute the task of the failed machine. The fault tolerance for workers can be noticed by the fact that the task is always completed, even with machine failures, because the master re-schedules the tasks to working machines until the whole task is completed.

The failure of a master is a critical case, but there are some strategies to deal with it such as configuring the master to write temporary checkpoint files in the global file system reporting the managing information, making it possible that a brand new master could continue the task from the point reported at the last checkpoint. Even with this strategy, it was chosen to abort a MapReduce computation on a master failure, making the clients retry the operation if needed.

An important feature present in the MapReduce model are the backup tasks. Not always the machines can finish the computation in expected time, delays may occur for different reasons, such as bad disks making the read and write operation time take longer or congested network decreasing data transfer speed. Those cases result in a whole task completion time increase, so in order to avoid it the backup tasks mechanism was created. This mechanism works when the operation is almost finished, at this point the master re-schedules the remaining in-progress tasks, making them available to other machines in the cluster execute them. The task is marked as completed if the primary or the backup execution reports it completion. There is a trade-off of execution time and computational resources, but as noticed in the experiment reported in [1], the gain achieved by this feature is huge, mainly in large operations.

Other main refinements documented are the ordering guarantees for sorted output files, allowing the user to provide an intermediate combining function to perform a partial merge of the data before sending it in the network, support for different input and output types of data, a skipping bad records mechanism, the maintenance of a http server to the user verify the status of the operation and a counter object provided by the library to allow the user count the occurrences of the wanted events. The MapReduce model, could work without this refinements, but having a decreasing performance.

4.3 Google's MapReduce Tests

The MapReduce library was written in C++. The tests performed and documented [1] on Google's MapReduce implementation had the following environment:

> *All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond*

.

And the whole operation was executed when most of the machines were in idle state. Two programs were used in the test, a grep program and a sort program, both hard-coded.

> *One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.*

The input were 10^{10} 100-bytes records, about of 1 terabyte of data. "*The entire computation takes approximately 150 seconds from start to finish*" and the sort took 891 seconds to finish. Tests were made for the sort but having no backup tasks used, which resulted in a increase of the completion time of 44%, showing how important are the backup tasks for the performance. The fault tolerance was also tested by killing 200 machines when the tasks was already stared/ The re-schedule worked well and the completion time was 933 seconds for the sort, 5% over the normal execution time.

4.4 Real-World Application

MapReduce is used in Google for a wide variety of problems, such as:

- *large-scale machine learning problems*
- *clustering problems for the Google News and Froogle products*
- *extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist)*
- *extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search)*
- *large-scale graph computations*

But the most important use of this model is the complete rewrite of the Google's program that generates the data structures used for the web search service, showing how important and well performed this computation model is. The model is successful because it's easy to use even in distributed environments, since its details are hidden; also the fault tolerant, other refinements and the improvements over use and time make the model very robust. The other reason for the success is that a gigantic amount of problems can be modeled to be computed by a MapReduce operation. The user, being able to write the Map and Reduce functions, can use the facilities of the model to benefit the task execution and if executed in an powerful environment can retrieve the results quickly.

For the presented reasons, a MapReduce feature written in Erlang language, taking advantage of the parallelization present in the language, based on Google's model was developed for the AngraDB project in order execute complex processing over the documents stored in the databases.

Chapter 5

The proposed solution

The MapReduce feature for the AngraDB was developed in the Erlang programming language and had as structural and conceptual reference the Google model described by Dean and Ghemawat in [1]. The MapReduce implementation can be checked in its GitHub repository[1]. The AngraDB was developed to handle huge sets of documents, but, with the exception of an early staged inverted index, still there isn't any kind of processing to obtain information about the data stored. The MapReduce comes as a tool to allow the users retrieve information about their databases, even in huge databases. This version of the feature was created to handle small tasks, focusing on understanding how MapReduce and distribution works, processing just a MapReduce task at a time.

5.1 Implementation

The feature was created as an Erlang module and its main module name is "adb_mr", a shorter identifier for AngraDB MapReduce module. It was developed to the distributed AngraDB version, based on Google's MapReduce model but with several adjustments in order to simplify the implementation and adapt the computational model to the AngraDB environment. AngraDB distribution was configured to replicate all information to each node, which means that all the nodes has every data ever stored in the database.

5.1.1 Supervisor module

The supervisor of the MapReduce node is very simple and follows almost all default settings. The restart strategy is "one_for_one", since it has no relation to others modules in an AngraDB instance, so if the node has an abnormal termination, it will be restarted. The intensity set is 1 and the period is 5, i.e the supervisor module will kill itself if the

[1]AngraDB MapReduce, see https://github.com/Angra-DB/core/tree/mapreduce

worker it is supervising needs a restart more than once in a 5 seconds period. Every supervisor child has a id, so the one chosen for its only child was the name of the MapReduce module itself ("adb_mr") since there will not exist any other child instantiated. No arguments are needed for the initialization function, nodes are always restarted when they terminate abnormally and the shutdown method chosen was default, wait for 5000 milliseconds before unconditionally shutdown, were the last supervisor configurations set.

In order to integrate the MapReduce tool, its supervisor module was added to the "*adb_dist_sup*" worker list. The name of this module means that it holds the AngraDB ("*adb*") distribution ("*dist*") supervisor ("*sup*"). Setting this configuration means that when the AngraDB is started, the supervisor that watches the distribution modules will spawn the MapReduce supervisor, which will start the MapReduce module next.

5.1.2 Main differences from Google model

The MapReduce feature had to be modified in order to adapt the model for the AngraDB. The database in distribution mode has more than one node running AngraDB making them the only machines available for job processing in opposition to Google's MapReduce model that was applied on a cluster of hundreds or thousands of machines. The concept of having one master, also called as manager, that manages the operation and a set of workers was preserved. Depending of the situation, the AngraDB distributed mode instance may be running with limited resources, such as a small number of nodes. This MapReduce implementation allows its execution every time that there are at least 2 nodes running: one node acting as the master and the other as worker.

For this feature, in order to simplify how the worker does it's job, every worker applies both operations in sequence, the Map and the Reduce. As explained before, this version of the feature was developed to handle tasks which all results are stored in memory and sent back to the master. In addition to Map and Reduce functions, there is one more function on the AngraDB feature: the Merge. The Merge function is executed by the manager when all workers finishes their tasks and it's applied to the list of reduce results, resulting in the output expected by the user.

For Google's MapReduce, clients must create a program using Google's library that was developed in C++ in order to request a MapReduce operation that will be replicated in the cluster to execute the tasks. For the AngraDB feature, clients must create an Erlang module containing implemented *map/1*, *map/reduce* and *merge/1* functions, because these are the only mandatory functions required from user, without them it is not possible the execute an operation.

Another difference is that Google uses distributed file system not relying in the Internet for file transmission and message exchange. This version of the MapReduce feature was designed to work with small amount of data, using the memory and Erlang ETS tables to store the results of the operations, increasing the performance but not being able to deal with huge data sets, aspect that must be improved in future versions, keeping the well performed operations with incredibly large amounts of data.

5.1.3 General Structure

The two main factors that defined the architecture were the flexibility and modularity of AngraDB and the decision of maintaining a manager node for each MapReduce task. The decision of defining a task manager makes it easier to control the whole task execution because this node knows all the workers executing the task and can distribute the documents among them in a simple and safe way to avoid reprocessing documents. The manager is also known by all the workers and receives the results from each one, making it possible to execute a merge function.

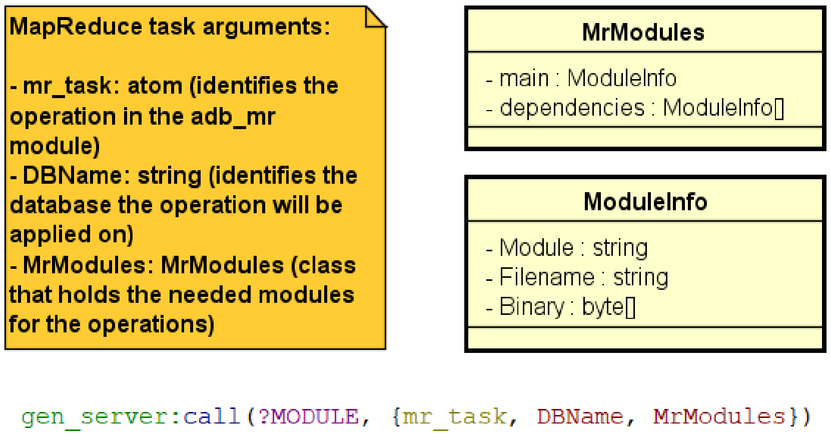
The Erlang language does not provide the concept of a class such as C++ or C#, but allows creating custom structures called "records". Each record receives a name and can have several named properties, that can be used for an easier understanding of the record's purpose and for pattern-matching. Some records were used in order to organize the data interaction, for example, the "gen_server" MapReduce state is stored using the "*nodeInfo*" record, that holds the status, the database name if any task is being executed, the management task id if the node is a manager or a workerTask id if it is a worker, and the last MapReduce whole task result, used for user consulting. The code below shows the record declaration.

```
-record(nodeInfo, {status, database, managementTask, workerTask, lastResult}).
```

All nodes have a status in their state indicating if it can execute a task or not. The "idle" state is self explained, means that the node is not executing any task and the "busy" status means that the node is executing a task. Following the AngraDB philosophy and some of the Google's MapReduce structure, the MapReduce node implementation gave flexibility a key role that is why there are two kinds of task that a node can execute, a management task (as explained before), node that will be called manager, and a worker task, node that will be called worker.

A node becomes a manager when receives a MapReduce task from a user in the "idle" status, changing its state to store all information needed to manage the task and its execution. The node becomes a worker when it receives a work task from a manager node in the "idle" status. An AngraDB instance can only process just one task at a time. This decision was made to simplify the execution workflow and the task management, avoiding dealing with multiple MapReduce operations, which would overcharge the managers.

Ease of use was not a concern in this work, that is why starting a MapReduce task is quite simple and can be done just by executing a synchronous "gen_server" call to a MapReduce node, sending as arguments the user created module (containing the *map/1*, *map/reduce* and *merge/1* functions) and all the dependencies modules. The Figure 5.1 illustrates and explains the arguments needed for requesting a MapReduce operation and gives an example of a real request. A function can use the shortcut "*?MODULE*" to reference its own module, in this example the MapReduce task request is being executed from the "*adb_mr*" module.

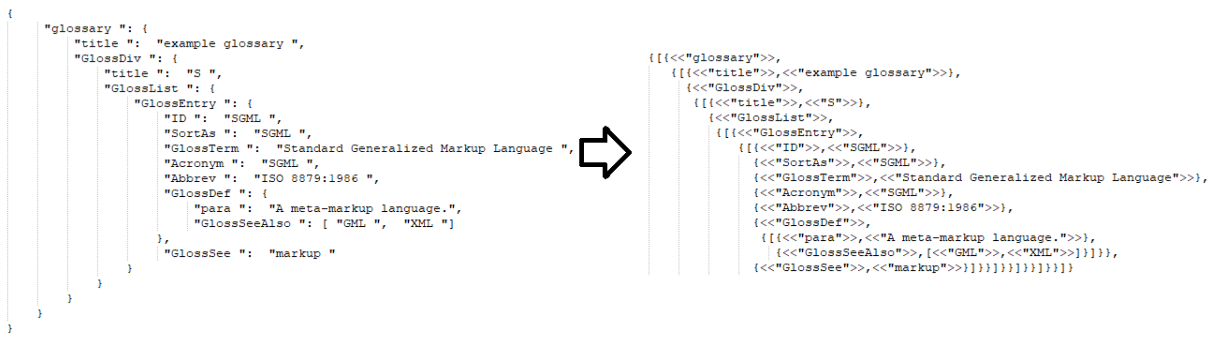

MapReduce task arguments:
- mr_task: atom (identifies the operation in the adb_mr module)
- DBName: string (identifies the database the operation will be applied on)
- MrModules: MrModules (class that holds the needed modules for the operations)
MrModules
- main : ModuleInfo
- dependencies : ModuleInfo[]
ModuleInfo
- Module : string
- Filename : string
- Binary : byte[]

```
gen_server:call(?MODULE, {mr_task, DBName, MrModules})
```

Figure 5.1: Visual explanation and representation of task request call arguments, with a call example.

The decision of making this a synchronous call is to provide a response for the task creator, even if the task could not be accepted, since the Erlang asynchronous calls may not send data on it responses. As can be seem in the previous code example, the arguments of the call are name of the MapReduce module ("adb_mr"), the atom "mr_task", which is the identifier of the function to be called, the name of the existent distributed database whose documents will be processed, and a record called "taskModules", that holds all the modules that need loading in order to allow the execution of Map, Reduce and Merge functions. There is a property in the record called "main" that must hold the module that contains the Map, Reduce and Merge functions. The lack of implementation of one of these functions will result in an error, as explained previously.

The Map function must receive one argument whose type is a parsed JSON document into a list of properties 5.2, with every field of the JSON turned into a tuple composed by the header and its contents. The Reduce function must receive one argument, but the argument must be a list of Map functions result type elements. Following the logic of the latter, the Merge functions must receive just one argument too, a list of Reduce function result type elements. The *map/1*, *map/reduce* and *merge/1* functions used for testing can be found in the attachments section.



```
{
    "glossary ": {
        "title ":  "example glossary ",
        "GlossDiv ": {
            "title ":  "S ",
            "GlossList ": {
                "GlossEntry ": {
                    "ID ":  "SGML ",
                    "SortAs ":  "SGML ",
                    "GlossTerm ":  "Standard Generalized Markup Language ",
                    "Acronym ":  "SGML ",
                    "Abbrev ":  "ISO 8879:1986 ",
                    "GlossDef ": {
                        "para ":  "A meta-markup language.",
                        "GlossSeeAlso ": [ "GML ",  "XML "]
                    },
                    "GlossSee ":  "markup "
                }
            }
        }
    }
}
```

```
{[{<<"glossary">>,
   {[{<<"title">>,<<"example glossary">>},
     {<<"GlossDiv">>,
      {[{<<"title">>,<<"S">>},
        {<<"GlossList">>,
         {[{<<"GlossEntry">>,
            {[{<<"ID">>,<<"SGML">>},
              {<<"SortAs">>,<<"SGML">>},
              {<<"GlossTerm">>,<<"Standard Generalized Markup Language">>},
              {<<"Acronym">>,<<"SGML">>},
              {<<"Abbrev">>,<<"ISO 8879:1986">>},
              {<<"GlossDef">>,
               {[{<<"para">>,<<"A meta-markup language.">>},
                 {<<"GlossSeeAlso">>,[<<"GML">>,<<"XML">>]}]}},
              {<<"GlossSee">>,<<"markup">>}]}}]}}]}}]}}]}
```

Figure 5.2: Representation of the original JSON and the result of the parser.

As explained before, Erlang allows the use of high order functions, but the functions passed as arguments will work only if all modules required by the function to work, including the function module itself, were loaded by the Erlang VM that is executing the function. That is why one of the design decisions made on the user task submission was to require from the user all modules needed to load in order to execute the map, reduce and merge functions provided. All tasks are received by a "gen_server" call, so this decision allows users to provide complex map, reduce and merge functions, but a large number of modules may increase the execution time when all modules sent are loading, since every node, worker or manager will have to load those modules.

Every task, when accepted, receives an id that will be used for further identification. This id is used as index in the ETS tables in order to store task information. Each node uses3 different ETS tables, the first for storing worker received tasks and the two remaining store the manager and worker task configuration. The tasks information are retrieved from the tables when required by the module using the previously created id.

5.1.4 Document position list

A feature that was being requested for AngraDB was developed along the MapReduce, the document position list file. Placed in the "*adb_doc_list*" module, the functions presented in it allow the generation of a file that stores the absolute position of a B+Tree leaf inside the index file, which can be used for a quicker retrieval of all data position inside the B+Tree document storage file. An auxiliary function that returns a sorted by document key array, each element holding the document access key and its exact position inside the document storage file, was implemented in "*adbtree*" module.

Other important function used in MapReduce is the document count, which returns the number of documents stored in a certain database. It's the only way to obtain this information and it is used on document delegation process. This module was not integrated with "*adbtree*", so when there is a save operation, for example, the list is not updated, all the use process of these functions is manual.

5.1.5 Worker task

The worker task was named as this to follow Google's model and because it simply receives the resources and the documents needed to execute a task, executes it and returns the result to the task manager. In order to be able to execute a task, the node must be in "idle" state when a task is received from a manager node. A worker task is received by an asynchronous call ("gen_server" cast), decision made to decrease the task distribution time, making the manager not wait for the worker response. The arguments received for the task are the id, the name/identifier of the manager, the document index list to process and the modules that must to be loaded.

This task was separated in two parts: the loading and the execution parts. The loading phase is in charge of loading the modules that will be used in the map and reduce functions and inserting in the ETS table the worker task information, but no start date is set. A design decision was taken on this point in order to validate the status check: the task must not be executed before the "gen_server" node state update. That is why this first phase ends updating the state and setting a timeout time of 5 ms, returning then the new state value (the "gen_server" state only updates with the return of the function). When a timeout happens in a "gen_server", a "handle_info/2" function is called with the atom "timeout" as first argument. In this case, there is a check to verify if the worker task has a start date or not. The case of not having a starting date means that the node has not started to execute the job, so the execution phase starts.

An asynchronous call is made to the node itself to start the map and reduce operation phase. When executing the map each document is retrieved using its index, using the list of document indexes the node must process received from its manager, and uses the sorted array of document positions, created by the functions implemented in "*adb_doc_list*" and "*adbtree*" modules, to retrieve the documents to be processed reading it from the exact position it is in the database storage file.

Each document is processed sequentially, applying the received map function in each of them, returning a list of documents processed. Then, the Reduce phase is executed by processing the list of Map results, resulting in the expected outcome of the worker. After that, a call is made to the manager sending the output of the task,

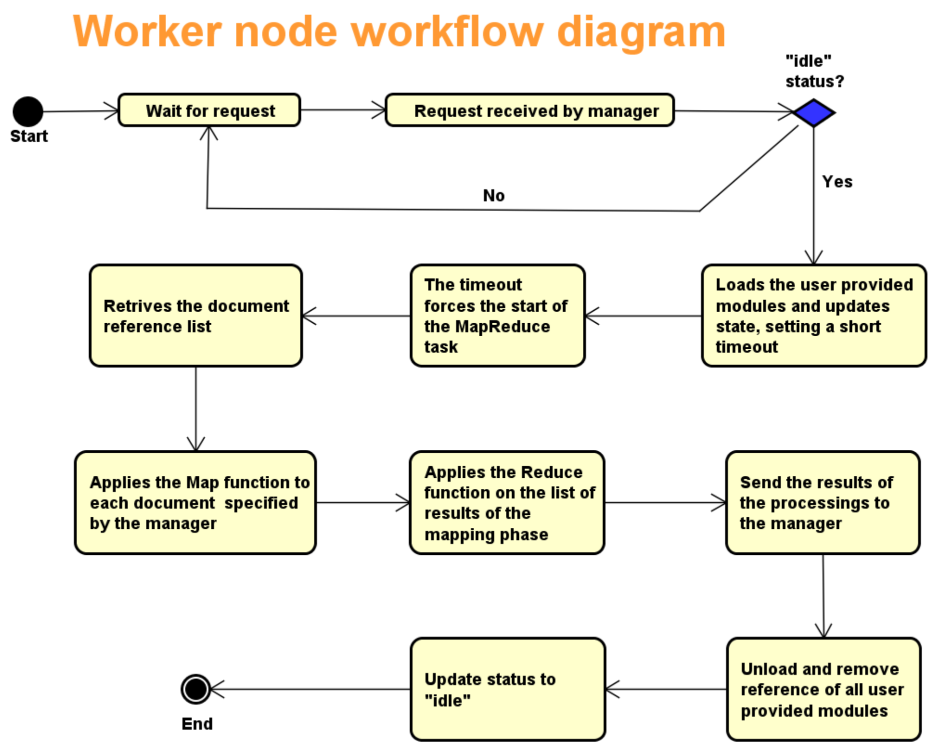

Worker node workflow diagram
Start
Wait for request
Request received by manager
"idle" status?
No
Yes
Loads the user provided modules and updates state, setting a short timeout
The timeout forces the start of the MapReduce task
Retrives the document reference list
Applies the Map function to each document specified by the manager
Applies the Reduce function on the list of results of the mapping phase
Send the results of the processings to the manager
Unload and remove reference of all user provided modules
Update status to "idle"
End

Figure 5.3: Full worker node workflow UML activity diagram.

A clening up phase starts, so the the user provided main module and its dependencies are unloaded and removed from the task information table to reduce the size of the stored information of a finished job. The state of the node is updated and it returns to the "idle" status. An UML activity diagram of this kind of node can be seen on figure 5.3.

5.1.6 Management task

Another important feature present on Erlang standard library is the "*code*" module. Only because of it, the existence of a flexibility aspect given for users was possible, allowing them creating their Map, Reduce and Merge functions in Erlang language anyway they want, using any Erlang dependencies as possible. What makes it possible are the functions "*get_object_code/1*" and "*load_binary/3*", the former retrieves the binary version of a loaded module and the latter loads a module binary into the Erlang context. This way, each client only needs to extract the binary from the created module and send as one of the MapReduce task request module. The users start a MapReduce task executing a call to a MapReduce node with all the modules needed to execute the Map, Reduce and Merge functions as binaries. The example below shows the functions use and its successful responses.

```
{_Module, Binary, Filename} = code:get_object_code(Module),
{module, Module} = code:load_binary(Module, Filename, Binary)
```

The management task means that the node received a task from some client and has the job of managing the whole task, whose workflow can be seen in Figure 5.4 represented in an UML activity diagram. The manager has to reset the tables used to store the tasks completion information and load all the modules received using the "code" Erlang library and the "load_binary/3" function. It must load those modules, because the merge function that will be executed when all workers end their jobs was provided in the module provided by the user in a module not previously loaded in the Erlang VM. Then, it creates an identification key also called as id using the same algorithm that generates the keys for the AngraDB document storage system, using the Hashids library[2].

To send the tasks to the workers, the manager must first find other "idle" MapReduce nodes. For that, it uses the function "nodes/0" provided by the distribution environment that will return a list of Erlang identification names of the acknowledged nodes. Those names are needed by the Erlang communication system to reach the workers in the calls. To verify the status of the worker nodes, the manager executes a ping call to all acknowledged ones, checking if they are "idle" or not, considering for the task only the ones that are not "busy".

After choosing all the nodes that will become workers, the manager has to coordinate which documents a node will process. The work split function was hard coded using a very simple logic, dividing the number of documents by the number of worker nodes but getting the floor integer result of the division. This logic, when the number of documents is not a multiple of the number of nodes, makes all the workers process the same amount of data, except by the last node, which will operate on the remaining documents. This decision was made to not overcharge the last node, splitting the work to the others instead.

The tasks are finally sent to the workers, now with all information needed to execute the operations. Tasks are sent in asynchronous calls in order to avoid blocking execution. The status of the manager node is finally updated to "busy" and the manager will keep waiting for the results of the workers.

Every time a worker ends its job, it sends the results to the manager by an asynchronous call. The manager then stores the result in its ETS table and verify if all workers delivered their results. If not, the manager will keep waiting for results, but if all were delivered, the list of worker results is processed by the merge function resulting in the output expected by the client. All modules provided by the user are then unloaded and had their references removed from the operation ETS table, the manager update it status to "idle" and the result of the whole process become available for lookup.

[2]Hashids, see https://hashids.org/erlang/

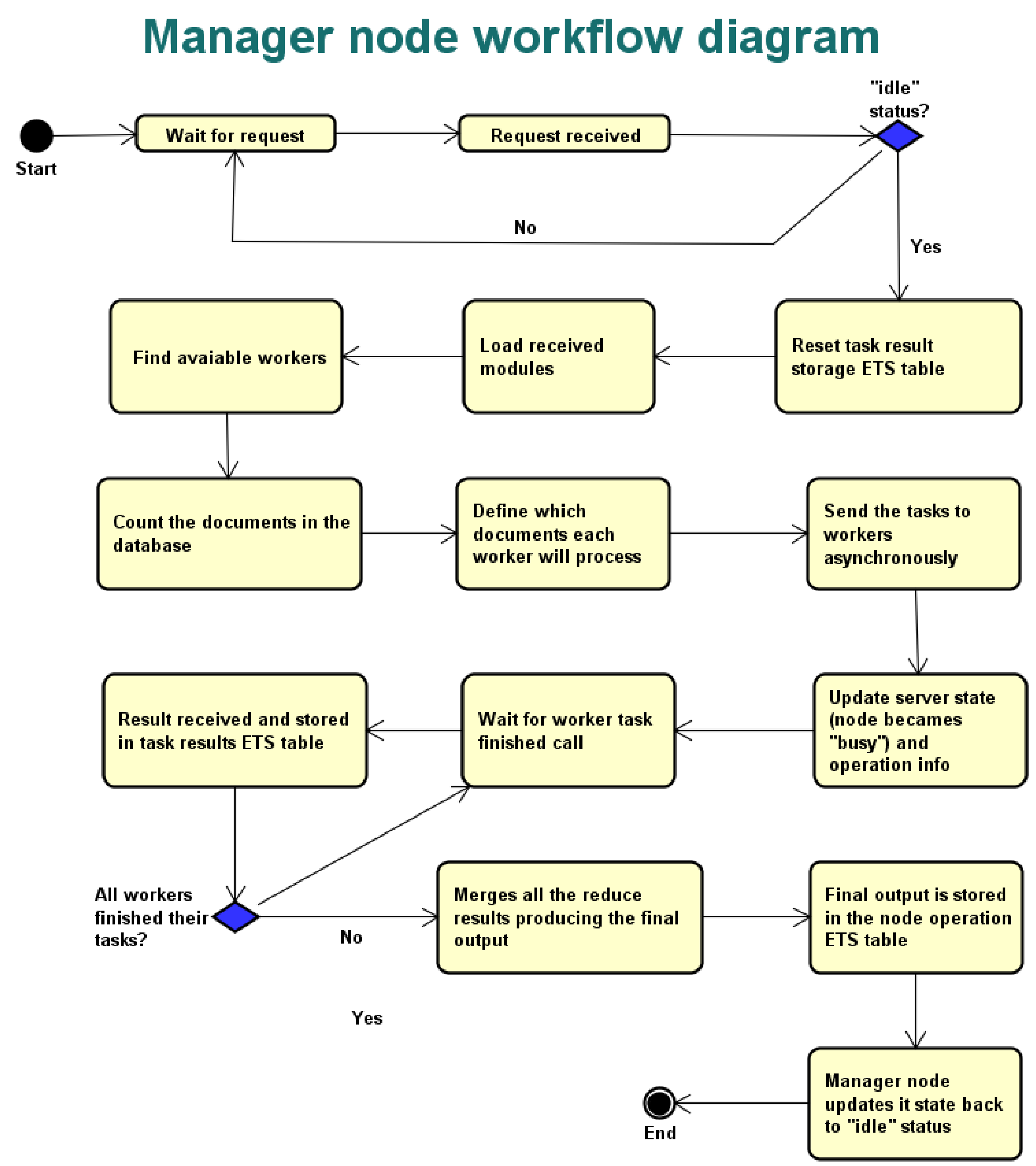

Manager node workflow diagram
Start
Wait for request
Request received
"idle" status?
No
Yes
Reset task result storage ETS table
Load received modules
Find avaiable workers
Count the documents in the database
Define which documents each worker will process
Send the tasks to workers asynchronously
Update server state (node becames "busy") and operation info
Wait for worker task finished call
Result received and stored in task results ETS table
All workers finished their tasks?
No
Yes
Merges all the reduce results producing the final output
Final output is stored in the node operation ETS table
Manager node updates it state back to "idle" status
End

Figure 5.4: Full manager workflow UML activity diagram.

5.1.7 Core programming

The current MapReduce version, as explained previously, is not very user-friendly, so some of the steps needed to use the feature require some experience with Erlang/OTP and functional programming, that is why this subsection has the goal of explaining the steps needed to create a function that is able to execute a MapReduce task. The Erlang VM command line will be used to compile and execute the functions presented.

The first requirement for request a MapReduce task is creating a module that implements a "*map/1*", "*reduce/1*" and "*merge/1*". The notation used to express the function means: "*<name of the function>/<number of arguments the function receives>*". The implementation of a module that seeks to count all words in the documents in a database can be seen in the attachments section, the module called "*adb_mr_tests*" will be used as example and will be referenced as "functional module". The module which will be called as "*adb_mr_helper*" contains the function that starts the MapReduce task, called "create_task/1" receiving as its only argument a string with the database name to be used.

In order to use the referenced modules, they must be compiled in the command line, using the commands inside an Erlang VM environment following the sequence:

```
c(adb_mr_tests).
c(adb_mr_helper).
```

Creating the request

The first step of the task creation is obtaining the functional module information and binary, using the Erlang "*code*" library. The following code extracts the module information from the "*adb_mr_tests*" module, receiving its name as an atom argument and splitting the response into 3 variables: Module, which holds the module name as an atom; Binary, it is the byte sequence representation of the module; and Filename, a string referring to the full filename that holds the module.

```
{Module, Binary, Filename} = code:get_object_code(adb_mr_tests)
```

Then, it is needed to create the record that will hold the modules needed for the MapReduce execution. The record is called "*taskModules*" and only contains two properties: "*main*" and "*dependencies*". The former must contain a tuple with the functional module information, but it's important to notice that the order of the arguments it's different from the received from "*code:get_object_code/1*", being the module's name, the filename and then the binary sequence. The latter is a list of Erlang modules information, because the flexibility of the architecture allows the user creating complex functional modules, even using other Erlang modules. So if the functional module has any other Erlang dependency, their binary and information must be extracted using the "*code:get_object_code/1*" and added to the "*dependencies*" property. The following code shows how to create the record with no needed dependency.

```
MrModules =
    #taskModules{main = {Module, Filename, Binary}, dependencies = []}
```

The last step is executing the call to the MapReduce node. For that, the function "*gen_server:call/2*" is used. It receives 2 arguments, the first is the destination of the call, which must be a running MapReduce node that can be identified by its module name (the atom "*adb_mr*") and Erlang runtime id. The second is the argument of the call, which must be a tuple containing the following arguments: the atom "mr_task", the database name and the record containing the modules. An example call is shown below.

```
Res = gen_server:call({adb_mr, NodeId }, {mr_task, DBName, MrModules})
```

When the server answers back, the variable "Res" will contain a tuple with the atom "ok", the task id and the string "Task stated." if everything worked or an error if any exception was thrown.

To start the whole MapReduce task request, in the command line, after compiling the modules, the following command must be executed, having as argument the database name.

```
adb_mr_helper:create_task(DBName).
```

The full module implementation for the helper module can be checked below.

```
-module(adb_mr_helper).
-record(taskModules, {main, dependencies}).
-compile(export_all).

create_task(DBName) ->
    Module = adb_mr_tests,
    {Module, Binary, Filename} =
        code:get_object_code(Module),
    MrModules =
        #taskModules{main = {Module, Filename, Binary}, dependencies = []},
    Res =
        gen_server:call(?MODULE, {mr_task, DBName, MrModules}),
    {ok, Res}.
```

Retrieving the results

In order to obtain the result from the MapReduce task, a ping call must be made to the node that the MapReduce task was requested which can be made in command line. The first argument of the call is the destination, a tuple composed by the MapReduce module atom "adb_mr" and the node Erlang runtime id and the second is a tuple only containing the atom “mr_ping”. The successful response of the call, returns a tuple with an “ok” atom, the status of the node, the name of database of the last task and the last task result, which can be seen in the following code.

```
{ok, Status, Database, LastResult} =
    gen_server:call({adb_mr, NodeId }, {mr_ping}).
```

These are the only operations possible at the moment.

5.2 Evaluation

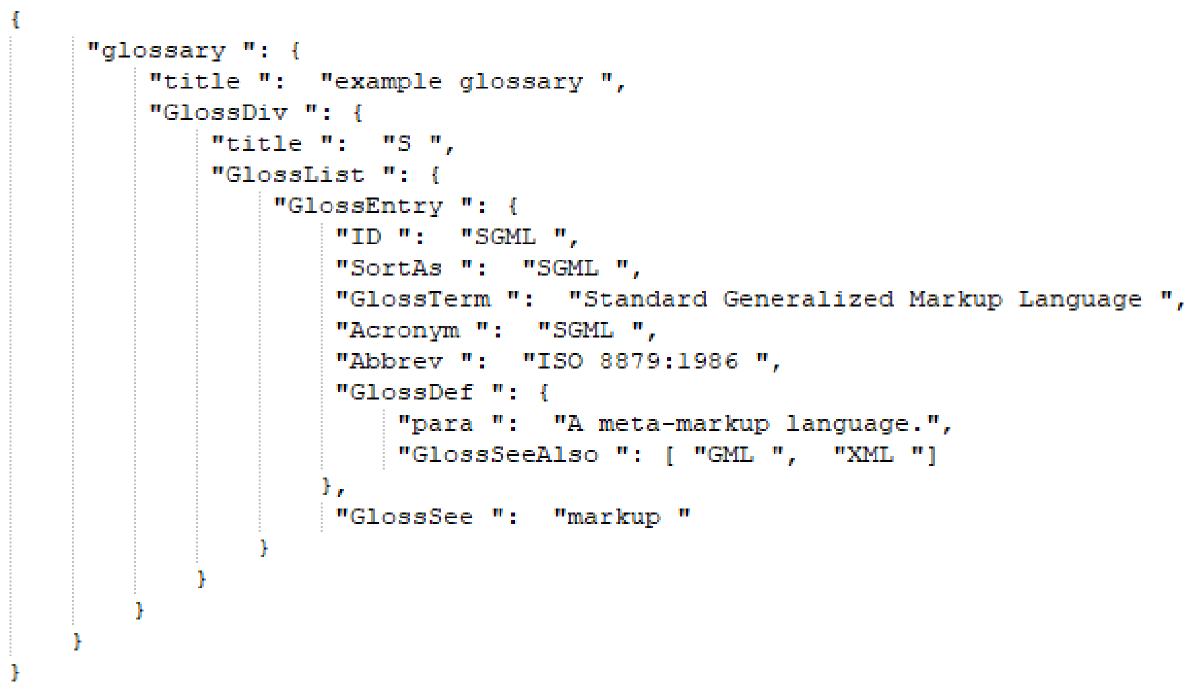
The MapReduce feature was tested using the distributed version of the AngraDB, where all data stored is replicated to all nodes, with a counting word application and Azure virtual machines with the following configuration:

- 4 GB of VRAM
- 8 GB of local SSD
- 2 CPU cores (Intel® Broadwell E5-2673 v4 2.3 GHz or Intel® Haswell 2.4 GHz E5-2673 v3)
- Ubuntu 18.04 (LTS)
- Virtual subnet hosted in Azure connecting all the machines

In total, 10 machines were created and placed on east side of EUA to be used in the tests, reaching the case of 1 manager and 9 workers. The tests were made using different number of workers and different number of documents stored, in order to verify the effect on performance in each case. Just one MapReduce test was developed, a simple counting word process. This decision was made to facilitate the result check even when applied on huge data sets.

5.2.1 Tests Performed

The tests were made considering 1, 3, 6 and 9 different workers, each one in its own virtual machine, and the following number of documents: 100 (28 KB), 1000 (274 KB), 3000 (820 KB), 8000 (2186 KB), 15000 (4099 KB), 45000 (12296 KB), 100000 (19297 KB). The Figures 5.5, 5.7 and 5.8 shows the JSON documents and their word count that were used in the tests, always having the same amount of each kind of document in every test case. Table 5.1 shows the time results on each case.



```
{
    "glossary ": {
        "title ":  "example glossary ",
        "GlossDiv ": {
            "title ":  "S ",
            "GlossList ": {
                "GlossEntry ": {
                    "ID ":  "SGML ",
                    "SortAs ":  "SGML ",
                    "GlossTerm ":  "Standard Generalized Markup Language ",
                    "Acronym ":  "SGML ",
                    "Abbrev ":  "ISO 8879:1986 ",
                    "GlossDef ": {
                        "para ":  "A meta-markup language.",
                        "GlossSeeAlso ": [ "GML ",  "XML "]
                    },
                    "GlossSee ":  "markup "
                }
            }
        }
    }
}
```

Figure 5.5: Document 1 - 33 words. Document used in test data sets..

Table 5.1: Time that a task took to complete in ms.

	Number of processed documents						
Worker number	**100**	**1000**	**3000**	**8000**	**15000**	**45000**	**100000**
1	35	257	678	2100	4588	22347	83678
3	32	109	252	903	1995	10304	41650
6	22	64	158	518	1035	6919	23041
9	48	55	134	342	675	3730	15925

It took more than 10 hours to insert document by document in the databases used on the tests, specially the 100000 documents database. All documents were saved using the AngraDB TCP Client, which generates a random key for each document stored. It makes the nodes, when receiving its data set to be processed, do not operate in just 1 kind of document.

As expected, the amount of time that a MapReduce task takes to complete is inversely proportional to the number of nodes used for operations execution, fact that is better noticed for the larger data sets. When comparing the execution with 1 node, which is the same situation when there is a traditional sequential processing, to the case of using 9 nodes for the 45000 documents set, the execution time decreases more than 83%, showing a huge performance increase.

Other noticed aspect is the feature behavior when working with a 100 documents database. Different from the other databases, the completion time did not decrease when

Table 5.2: Average time to process 1 document in ms.

	Number of processed documents						
Worker number	**100**	**1000**	**3000**	**8000**	**15000**	**45000**	**100000**
1	0.35	0.257	0.226	0.263	0.306	0.497	0.837
3	0.32	0.109	0.084	0.113	0.133	0.229	0.417
6	0.22	0.064	0.053	0.065	0.069	0.154	0.230
9	0.48	0.055	0.045	0.043	0.045	0.083	0.159

increasing the worker number, even all tasks taking less than 50 ms to be completed. It shows that when dealing with small amount of data the "processual" cost that comes with operations not directly related to MapReduce process, such as network data transmission and ETS table operations, have a direct influence to the whole task execution time.

Table 5.2 also shows the average time in ms took to one document be processed in each case, information that allows to conclude that this average time does not have a direct linear relation to the number documents stored, because the smallest times in the test were found when processing the 3000 and 8000 documents database. Although, the relation between average time to process one document is directly proportional to the number of workers used, when dealing with databases composed by 1000 or more documents, preserving and enforcing the premise of increasing performance when having more workers available.

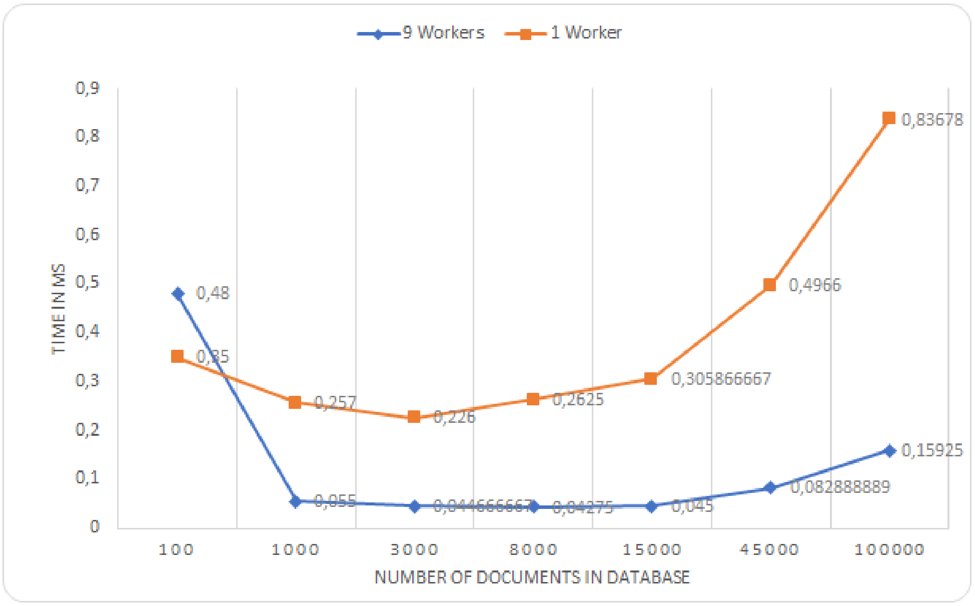
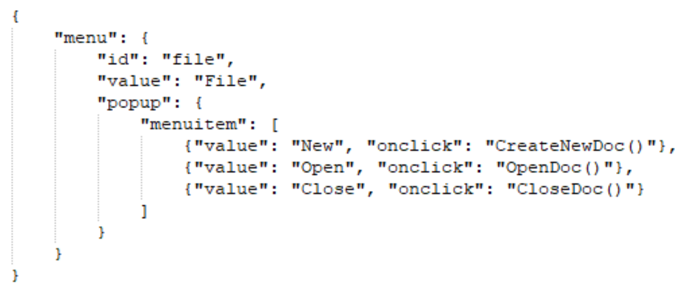


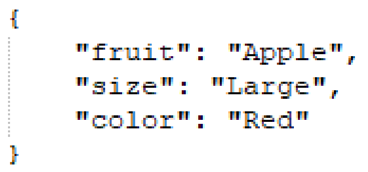
Figure 5.6: Document used since the most preliminar stages of the feature..

The chart on Figure 5.6 compares the average time in ms took to one document be processed using 1 worker and 9 workers. It's clear that the case with 9 workers out per-



```
{
    "menu": {
        "id": "file",
        "value": "File",
        "popup": {
            "menuitem": [
                {"value": "New", "onclick": "CreateNewDoc()"},
                {"value": "Open", "onclick": "OpenDoc()"},
                {"value": "Close", "onclick": "CloseDoc()"}
            ]
        }
    }
}
```

(a) Document 2 - 19 words



```
{
    "fruit": "Apple",
    "size": "Large",
    "color": "Red"
}
```

(b) Document 3 - 6 words

Figure 5.7: Documents stored in AngraDB to test MapReduce.

forms the other case in all data sets, what another fact to notice is the non-linear behavior of the chart. The cases with less average time were operating on 3000 and 8000 data sets, although the times are very similar to the 1000 data set. Observing the processing time along the growth of data sets, it's possible to verify that the "operational" cost of the operation is not constant and increases along the number of processed documents.

A deeper analysis can be made using the results obtained by the following cases:

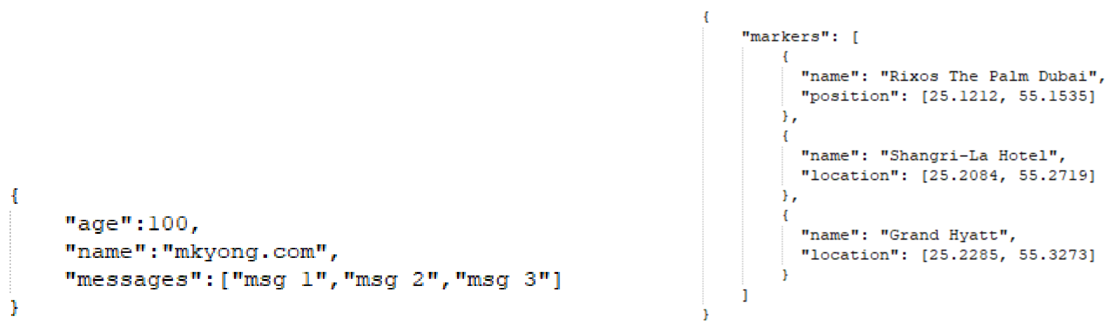
- 15000 dataset processed by 3 workers: 1995 ms
- 45000 dataset processed by 9 workers 3730 ms

Considering the function used to split documents for processing, on both cases each node will process exact 5000 documents. In an ideal scenario, not considering the "operational" and network costs, both tasks would take the same amount of time to be completed, since each machine is processing in parallel the same amount of documents, but it did not happen in the results. The conclusion that can be taken from it, is that the process of receiving results from more nodes and merging a wider set of results have a direct influence in the performance, showing that the manager is the bottleneck of the feature.

The original implementation of MapReduce in Google's article does not specify a way to merge the reduce phase results, which is an original addition to AngraDB. In order to not execute any processing in this phase the following function must be used, forwarding the reduce results making them the final one, creating no merge execution costs.

```
merge(List) ->
    List.
```

A similar analysis can be made comparing the databases containing 1000 and 100000 documents:



```
{
    "age":100,
    "name":"mkyong.com",
    "messages":["msg 1","msg 2","msg 3"]
}
```

(a) Document 4 - 11 words

```
{
    "markers": [
        {
          "name": "Rixos The Palm Dubai",
          "position": [25.1212, 55.1535]
        },
        {
          "name": "Shangri-La Hotel",
          "location": [25.2084, 55.2719]
        },
        {
          "name": "Grand Hyatt",
          "location": [25.2285, 55.3273]
        }
    ]
}
```

(b) Document 5 - 21 words

Figure 5.8: More documents stored in AngraDB to test MapReduce.

- 1000 doc. database processed by 1 worker: 257 ms; 100000 doc. database processed by 1 worker: 83678 ms;
- 1000 doc. database processed by 3 workers: 109 ms; 100000 doc. database processed by 3 workers: 41650 ms;
- 1000 doc. database processed by 6 workers: 64 ms; 100000 doc. database processed by 6 workers: 23041 ms;
- 1000 doc. database processed by 9 workers: 55 ms; 100000 doc. database processed by 9 workers: 15925 ms;

The time to process the wider document set wasn't in any case 100 times greater that the smaller one, showing one more time that there is no linear relation between the number of documents operated on and the processing time, demonstrating that the "operational" cost is directly proportional to the number of documents.

Chapter 6

Conclusion and Future works

AngraDB, the project started in Universidade de Brasilia, aimed for its members to learn Erlang language main concepts while creating a well-performed document based database, suitable for not any kind of application. But analyzing the current scenario, databases cannot offer support for only the basic operations, i.e insert, retrieve, update and delete documents, using traditional computational techniques such as sequential processing if they want to be suited for real-world applications.

6.1 Conclusion

In [1], Google produces an incredibly amount of data has to be processed, whose size make it impossible to use memory in order to execute the whole processing, resulting in an unreasonable execution time. In [5], a massive genome database also need to be processed, a task described as "*difficult for even computationally sophisticated individuals*". Both cases from different contexts had the same solution for its problem: a MapReduce model. The MapReduce has emerged as a powerful tool for processing huge amounts of data concurrently in a reasonable time, that is why a MapReduce implementation was created for AngraDB.

The MapReduce model developed was integrated with AngraDB, using its distribution system to replicate the nodes. Tests were made using different number of nodes and a variety of datasets with different sizes. The model was successful in all its tests, being able to execute the required tasks even with several configurations, decreasing the whole task time each time the number of workers increased, fact which was expected.

A lot of challenges were faced during the development, most of them related to the aspects of Erlang language and virtual machines and the rest were related to distribution. Halfway in the implementation of the planned architecture, one discovered problem was related to how Erlang high order function works when they act as arguments, only a reference to the implementation of the function is used, not a binary containing everything needed to execute it. This decision also obligates the Erlang VM that is using a function received as argument to load the binary of the functions module. In a single machine and Erlang context there wouldn't exist any problem related to this issue, but since the MapReduce allows the users to provide their own functions, a solution to this issue had to be developed.

The standard Erlang library called "code" was used to solve this problem, because it provides the following functions: "*get_object_code/1*", "*load_binary/3*", "purge/1" and "delete/1". The former is used to extract from a loaded module its binary simply providing its name. This function allows the user develop their own Erlang modules using other Erlang modules as dependencies and obtain its binaries to use it in the MapReduce process, sending them in the task request call. The "*load_binary/3*" is the opposite, using the result of the "*get_object_code/1*" function, it can load a module to the current Erlang context, making available for use all its functions. The last two functions are used to remove modules that were loaded in the Erlang VM.

The decision of using the "*code*" standard Erlang library made the MapReduce feature in this version only usable by code implemented in this language. This is a huge constraint, but it counterpoint is that all Map, Reduce and Merge functions can use the full potential of Erlang, since the feature were implemented in the same language, decreasing the chances of compatibility errors.

It is important to take note that the implementation had no problems with distribution or node communication. Every single test finished successfully with different number of nodes used in the task, showing that even in limited resources environments, i.e with a restricted number of workers, the MapReduce implementation will be able to work properly.

Even with all problems and adaptations documented, the objective of the project which was deliver a working MapReduce feature for AngraDB was completed. The tests made enforced the importance of this technique for current Big Data problems, showing that parallelization can considerably decrease the execution time of data processing operations. Adding this feature to AngraDB prepares it to the real world demands, showing that the group is aware of the successful existent technologies and is capable of implementing them, even with the ones with higher complexity.

6.2 Future works

It took a really long time to insert the documents in the database, a aspect that must be improved in AngraDB future versions. It was noticed that the time to store documents grows as the number of documents that are currently stored in the database. The lack of improvement of the insertion mechanism will make this operation the struggle of the system, making the insertion time of a huge number of documents unreasonable.

A better interface was still not developed for the MapReduce feature, an issue that became much harder with the constraint of using the "*code*" library for user interaction and module load and transmission. Erlang is a language that works very well with byte arrays, it was designed to deal with telecommunication demands, that's why the support for module binaries is easy to use, but will require another techniques to provide interface to other protocols, such as treating the binaries as base 64 strings allowing the request processing by most protocols.

In Google's article all the result and intermediate files are stored in file systems, because the clusters share those systems. In the future, AngraDB MapReduce will start to use files or even the database itself to store the results instead of using memory. This change will allow the feature process huge amount of data without having the memory limit. This first version had this decision applied in order to increase operations speed and to simplify the development.

The current feature version uses a "*gen_server*" synchronous call to user retrieve the results of the demanded task. As explained before, no special attention was given to user interface, just the minimum to allow the tests developed.

Google's MapReduce implemented an improvement that increase significantly the performance of its model, the backup tasks, which was not developed for AngraDB. In the former architecture, each document is treated as unit when assigned to a machine, making it easy to identify a demand for backup task and it assignment. Any time each document is processed, the manager is notified of that. In the developed MapReduce, each worker node receives a set of documents to be processed and the manager node only is updated about the documents processed when the node sends its results. In a context that there won't be a huge number of machines to execute operations that won't be necessarily in local networks, relying in the Internet for communication, the decision of delegating a set of documents to be processed was made in order to decrease network traffic and the amount of dependency on the net, making the nodes execute the calls just in appropriated moments.

The only distribution partition method that works with MapReduce feature is the "*full*", where all documents are replicated to all nodes. This method increases the dependency on the network because of the nodes synchronization, aspect that has much less effect on "*consistent hash*" partitions. It's a future improvement the refactoring the document delegation function in order to work with the other partition strategy.

In order to follow the highly configurable AngraDB principle, it'll be a great addition to the project adapt the workers to have its behavior configured to an environment with large amount of machines connected by high speed and reliable networks, making each node execute just one kind of task, Map or Reduce, just the way it was documented in Dean and Ghemawat work[1].

As explained in the previous chapters another contribution to AngraDB that MapReduce uses constantly is the "*adb_doc_list*", which creates and maintains a file that allows the retrieval of the list of documents absolute position in the file system, making the document retrieval instantaneous. It is still used to count how many documents are stored in the database. All those functions are currently working, but this module was not still connected to "*adbtree*" module, in order to make the operations on the document list automatic, at each database operation.

The greatest issue of the project that must be improved as soon as possible is the error and exception handling. Not all exceptions are being treated as they must and errors in the process are not being logged in a way that is easy to trace what ended up in a problem. Still on the error handling aspect, a way to trace the user provided functions it would desirable, because there were cases, even when the MapReduce was being tested, that errors happened on Map, Reduce or Merge functions, showing that those kind of mistake can be very common.

References

[1] Dean, Jeffrey and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters.* In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004. `https://storage.googleapis.com/pub-tools-public-publication-data/pdf/16cb30b4b92fd4989b8619a61752a2387c6dd474.pdf`. viii, 1, 2, 18, 19, 20, 21, 22, 24, 40, 43

[2] Lämmel, Ralf: *Google's MapReduce programming model — Revisited.* Science of Computer Programming, 70(1):1–30, jan 2008, ISSN 0167-6423. `https://www.sciencedirect.com/science/article/pii/S0167642307001281`. 1, 2

[3] Ghazi, Mohd Rehan and Durgaprasad Gangodkar: *Hadoop, MapReduce and HDFS: A Developers Perspective.* Procedia Computer Science, 48:45–50, jan 2015, ISSN 1877-0509. `https://www.sciencedirect.com/science/article/pii/S1877050915006171`. 1, 2

[4] Marozzo, Fabrizio, Domenico Talia, and Paolo Trunfio: *P2P-MapReduce: Parallel data processing in dynamic Cloud environments.* Journal of Computer and System Sciences, 78(5):1382–1402, sep 2012, ISSN 0022-0000. `https://www.sciencedirect.com/science/article/pii/S0022000011001668`. 2

[5] McKenna, Aaron, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A DePristo: *The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data.* Genome research, 20(9):1297—1303, September 2010, ISSN 1088-9051. `http://europepmc.org/articles/PMC2928508`. 2, 40

[6] *Couchdb documentation.* `http://http://docs.couchdb.org/en/stable/index.html`, visited on 2019-01-20. 7

[7] Cesarini, Francesco, Simon Thompson, Beijing • Cambridge, • Farnham, • Köln, • Sebastopol, • Taipei, and • Tokyo: *Erlang Programming.* O'Reilly Media, 2009, ISBN 0596518188. `www.it-ebooks.info`. 11

[8] Medeiros, Ismael C.: *Implementing a Distributed Architecture onAngraDB.* pages 1–41, jan 2019. 14

[9] Özsu, M. Tamer: *Distributed databases.* In Bidgoli, Hossein (editor): *Encyclopedia of Information Systems*, pages 673 – 682. Elsevier, New York, 2003,

ISBN 978-0-12-227240-0. `http://www.sciencedirect.com/science/article/pii/B0122272404000460`. 14

[10] Brewer, Eric A: *Towards robust distributed systems (abstract).* Proceedings of the nineteenth annual ACM symposium on principles of distributed computing, 2000. 14

Attachment I

Map, Reduce and Merge functions used to test the feature.

```
map(Doc) ->
    word_count(Doc).

reduce([]) ->
    0;
reduce([Head | Tail]) ->
    Head + reduce(Tail).

merge(List) ->
    reduce(List).

word_count(Element) when is_atom(Element) ->
    1;
word_count(Element) when is_integer(Element) ->
    1;
word_count(Element) when is_float(Element) ->
    1;
word_count(<<Element/binary>>) ->
    ElementList = string:tokens(binary_to_list(Element), " "),
    length(ElementList);

word_count([]) ->
    0;
word_count([Head | Tail]) when is_atom(Head) ->

word_count([Head | Tail]) when is_integer(Head) ->
```

```
    1 + word_count(Tail);
word_count([Head | Tail]) when is_float(Head) ->
    1 + word_count(Tail);
    word_count([<<Head/binary>> | Tail]) ->
    HeadList = string:tokens(binary_to_list(Head), " "),
    length(HeadList) + word_count(Tail);

word_count([{<<Header/binary>>, Content} | Tail]) ->
    HeadList = string:tokens(binary_to_list(Header), " "),
    HeadLength = length(HeadList),
    ContentLength = word_count(Content),
    HeadLength + ContentLength + word_count(Tail);

word_count([Head | Tail]) when is_list(Head) ->
    HeadCount = word_count(Head),
    HeadCount + word_count(Tail);

word_count(_) ->
    throw(invalid_json).
```