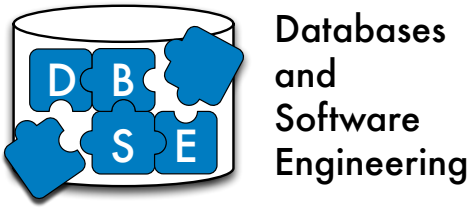
University of Magdeburg

School of Computer Science



Bachelor's Thesis

Fast Parsing of NDJSON Files in Main Memory Databases

Author:

Steven Schulze

June 24, 2020

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Department of Technical and Business Information Systems

Dr. David Broneske

Department of Technical and Business Information Systems

M.Sc. Marcus Pinnecke

Schulze, Steven:
Fast Parsing of NDJSON Files in Main Memory Databases
Bachelor's Thesis, University of Magdeburg, 2020.

Abstract

The Karbonit project aims to create an main memory database that is able to process data efficiently. Main memory databases tend to be faster than databases that use disc storage because the access to information in the main memory is way faster. But although the time to access is lower it is important that every part of the Karbonit database is as efficient and fast as possible. To process data, the data must first be inserted into the database. This process of inserting data is focus of this thesis.
A great way to insert a large amount of data at once is to retrieve this data from an input file like a JSON file, which is already possible in the Karbonit project. This is done by parsing the JSON input. Additionally to the option of parsing JSON the Karbonit JSON parser also has to be able to parse NDJSON files. A data format that is derived from JSON and enables the parser to process chunks of information one by one.
To improve the Karbonit JSON parser by enabling the parse of NDJSON files and in general some more strategies like parallelization or statistics are discussed and evaluated.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Over the past years computer science has developed to handle large amounts of data in less and less time. It's a continuous process driven by many companies and individuals, that are interested in the research of new technologies or the optimization of existing ones. An example of this process is the way processors were improved over the last five decades, like described by Moore's Law [M+65]. Moore predicted, that every two years the amount of transistors build into an IC (integrated circuit) will be doubled, which increases the computation speed of the CPU, which is basically an IC. The general rule is the faster the better.
To suit this trend this bachelor thesis aims to find ways to optimize a part of the Karbonit project [PCZ+19].

Goal of the Karbonit project is to create an document store, that stores and handles data in a way, that enables an efficient analysis and exploration of the stored data [PCZ+19]. To reach this goal, a new variant of the widely used compact JavaScript Object Notation (JSON) Format [Bra17] was developed, the Columnar Binary JSON (Carbon) file format [P+20]. It is an binary data format that stores the data as key-value-pairs in a columnar structure. Because of this columnar structure it is easier to perform compression or query the data, as if it would be stored as rows.
One way to create a Carbon file is to create an empty Carbon record, fill this record step by step with information and save it afterwards as a file. This approach is very inefficient if you already have a large amount of data, that has to be stored as a Carbon file. For this reason it is also possible to convert from JSON to a Carbon record, which can be saved as a file. It is way faster to convert, than to manually fill the record. But as seen with the processors there may be potential to improve this conversion too.

Goal of this Thesis

The conversion from JSON to Carbon currently consists of two parts:

- The Karbonit JSON Parser, which creates an spanning tree structure containing all the information provided within the JSON file.
- A function that creates a Carbon record out of the spanning tree build up by the parser. It may also optimizes the way the information are stored, so that less memory is needed to save the record.

Any improvement to one of those two parts is an improvement for the conversion itself. Focus of this bachelor thesis is the Karbonit JSON parser. As an improvement counts anything, that achieves any of the following:

- lower the memory consumption while parsing a JSON file
- parse a JSON file in less time compared to the initial situation
- parse larger amounts of data than initially possible

Additionally to this, the parser has to be able to parse Newline Delimited JSON (NDJSON) files. Because of their structure those files allow the parser to parse way larger amounts of data step by step, which stretches the memory consumption over a certain period of time.

Structure of the Thesis

To guide step by step through the work and results this thesis is based on, the next chapters will contain the following:

- Chapter 2 Background
  gives all important background information about the Karbonit project, JSON and NDJSON, JSON parser, Microsoft Academic Graph files and Multithreading.
- Chapter 3 Requirement Analysis
  explaines the initial situation, defines the scope of this thesis more specific and states the functional and non-functional requirements the implementation result has to fulfill.
- Chapter 4 Implementation
  contains all the implementations made to reach the goal of this thesis.
  - Section 4.1 Parse Line by Line
    enables the parser to parse NDJSON files. Instead of parsing the entire input as a whole, every line is parsed separately. To do this, some changes have to be made to the current parse procedure.

  - Section 4.2 Multiple Threads describes how parallelization can be used to speed up the parser based on the implementations made in the previous chapter.
  - Section 4.3 Get Tokens and Interpret Them Immediately combines the two main components of the JSON parser. It describes what those components are, why it is more effective to combine them and how it is implemented in the resulting parser.
  - Section 4.4 Check for escaped Quotes fixes the problem that escaped quotes are not ignored under certain circumstances which ends a string early and causes errors.
  - Section 4.5 Build up Statistics explains why in case of the Karbonit JSON parser a statistic can help to parse larger input files. It is also described how exactly the statistic is implemented.
  - Section 4.6 Parser Configuration describes all options the new Karbonit JSON parser has to configure the way files are parsed.

- Chapter 5 Evaluation separately shows the results of the implementations described in chapter 4, 5, 6 and 7, discusses which of the implementations are useful and compares the resulting Karbonit JSON parser with the original parser.

- Chapter 6 Related Work mentions other (ND-)JSON parser and publications in connection with these parsers.

- Chapter 7 Conclusion and Future Work sums up the results presented in Chapter 5, discusses if all requirements are fulfilled and what may follows in the future.

2. Background

This Chapter gives important background information about JSON, NDJSON, the Karbonit project, Parallelization and the MAG files.

2.1 JSON and variants

Like stated in Chapter 1 Introduction one goal of this thesis is to enable the Karbonit JSON parser to parse NDJSON files. NDJSON is based on JSON and to fully understand the structure of NDJSON files some information about the JSON structure and NDJSON itself is required.

2.1.1 JSON

The JavaScript Object Notation (JSON) data format is regulated by two standards. The ECMA standard ECMA-404 [INT20] and the RFC standard RFC 8259 [Bra17]. They describe the JSON structure in detail, but not the entire structure is relevant for this thesis.

The part of the JSON structure relevant to this thesis:

```
1  JSON         = element
2  element      = whitespace value whitespace
3  value        = object | array | string | number |
4                 "true" | "false" | "null"
5  object       = '{' whitespace '}' | '{' members '}'
6  members      = member | member ',' members
7  member       = whitespace string whitespace ':' element
8  array        = '[' whitespace ']' | '[' elements ']'
9  elements     = element | element ',' elements
10 element      = whitespace value whitespace
11 string       = '"' characters '"'
```

Listing 2.1: JSON Structure

2.1.2 NDJSON

Newline Delimited JSON (NDJSON) is a JSON variant that can be used to store or stream valid instances of JSON text. The NDJSON specification [HDPW20] states the following:

- A line of a NDJSON file contains a valid JSON text.
- Every line must end with the newline character '\n'.
- Newline characters are not allowed inside the JSON texts.

Every newline character indicates the end of a JSON text and the line end. The JSON texts contained in a NDJSON file are independent of each other because of this structure. That allows to parse them one by one, unlike a JSON file, that has to be processed as a whole.

To show the difference between JSON and NDJSON the following three valid JSON objects should be transferred:

- {"title":"t1"}
- {"title":"t2"}
- {"title":"t3"}

To transfer those objects as a JSON file they have to be stored in a JSON array or object, so that the receiver is able to interpret the JSON file. JSON objects require a key for every value, because of this the best way is to generate an array, that contains the three objects.

```
1 [{"title":"t1"},{"title":"t2"},{"title":"t3"}]
```

Listing 2.2: JSON Example

To get a valid NDJSON file the three JSON objects can be stored one by one/line by line, while a newline character is added to the end of every line.

```
1 {"title":"t1"}
2 {"title":"t2"}
3 {"title":"t3"}
```

Listing 2.3: NDJSON Example

While the three objects are included into an Array if saved in a JSON file, they are independent when saved in a NDJSON file. If the receiver wants to get the information of the n-th object he has to look at the whole JSON file or just the n-th line of the NDJSON file.
The same example, only this time the memory consumption is considered:
(1 Char = 1 Byte)

File Type	File Size	Basic Memory Consumption
JSON	46 Byte	46 Byte + x
NDJSON	45 Byte	15 Byte + x

Table 2.1: Comparing memory consumption while processing JSON and NDJSON file to retrieve information about a single object

2.2 Threads and Parallelization

Parallelization is a powerful technique to speed up computations. Normally instructions are processed in sequence, but if instructions are independent of each other they can be processed at the same time.

For example the two instructions Ins1 and Ins2. The following applies to these instructions:

- They are independent.
- Ins1 runs 5s.
- Ins2 runs 4s.

An Algorithm that runs first Ins1 and afterwards Ins2 would need 9s to finish.
Both instructions are independent, so they can run at the same time, if the algorithm and the hardware, on which the algorithm runs, allow it. The Algorithm that runs Ins1 and Ins2 parallel would only need the time, the slowest instruction needs. In this case 5s, which is nearly half of the time needed for the sequential run.

The basic process of parallelizing an algorithm and how the resulting tasks are run, described by Thomas Rauber and Gudula Rünger:
"The design starts with the decomposition of the computations of an application into several parts, called tasks, which can be computed in parallel on the cores or processors of the parallel hardware. [...] The tasks of an application are coded in a parallel programming language or environment and are assigned to processes or threads, which are then assigned to physical computation units for execution." [RR13]

2.3 Karbonit Project

The Karbonit project was initially called "Protobase". It is a project by the Working Group Database and Software Engineering of the University of Magdeburg.
The source code is developed with C as the programming language.
Some functions, that are already implemented, will be used in this bachelor thesis as reference for a similar implementation or directly called to work with the result.

2.3.1 The Carbon-Tool

The Carbon-Tool is an executable, that enables the user to do one of the following actions:

- checkjs - Checks if the input is a valid JSON file and if it can be converted to Carbon
- convert - Converts a valid JSON file to a Carbon file
- view - Prints a Carbon file in human readable form
- inspect - Displays information about a Carbon file
- to_json - Converts a Carbon file to a JSON file
- list - Lists properties and configurations for Carbon-Tool

The descriptions are taken from the ”help” documentation of the Carbon-Tool.

More important than these actions is the way those actions are implemented.
This implementation can be used as a reference to implement the different options to configure the resulting parser.

2.3.2 Hash Functions

Hash functions are used to distribute given input values across a specified value range. Christof Paar and Jan Pelzl stated that ”For a particular message, the message digest, or hash value, can be seen as the fingerprint of a message, i.e., a unique representation of a message.” [PP10].

The Karbonit project already contains some implementations of hash functions that can be used anywhere in the source code. Those functions are actually implemented as macros, which are more flexible than functions.
Some examples:

- HASH_ADDITIVE
  For every char of the input string add the value representing the current char to the result.
- HASH_BERNSTEIN
  For every char the current result is multiplied by 33 and the value of the char is added.

How the hash value is computed differs between those hash functions, so with the same input they will return different values.

2.3.3 Vector

The programming language C does not contain a vector-like datatype, whose size is dynamically adjusted during runtime. It only supports fixed size arrays with whom you could implement a vector-like structure, by reallocating the memory every time the array would be overfilled. But that is only one way to do it.

In addition to the hash functions, the Karbonit project contains a vector implementation too. The most important methods implemented for the vector are:

- vec_create - Creates a new vector
- vec_at - Returns the element stored at the specified position
- vec_push - Adds a new element at the end of the vector
- vec_is_empty - Checks if the vector contains any element
- vec_pop - Removes the last element stored in the vector
- vec_drop - Drops the entire vector and frees the memory

2.3.4 Threadpool

There is one more important structure that is already implemented in the Karbonit project, the threadpool.
Important functions for this thesis:

- thread_pool_create - Creates a threadpool with the specified amount of threads
- thread_pool_enqueue_task - Adds a new task to the queue for waiting tasks
- thread_pool_enqueue_tasks_wait - Adds multiple new tasks to the queue and waits until all tasks are finished
- thread_pool_wait_for_all - Waits until every task, added to the threadpool, is finished
- thread_pool_free - Drops the threadpool and frees the memory

2.4 JSON Parser

Grune et al. describe that parsing is a process to structure data according to a given grammar. They also state, that the result of this process supports further processing of the data.[GJ08]

Those statements are pretty abstract, so that they can be interpreted depending on the situation. In the case of a JSON parser the data that has to be structured is the input JSON text. The given grammar is the JSON structure. In the case of the Karbonit JSON parser, the resulting spanning tree structure helps to optimize the data and to create a Carbon file.

2.5 Microsoft Academic Graph Files

The Microsoft Academic Graph files (MAG files) as part of the Open Academic Graph [AMi20] serve as a basis for later evaluations. Their structure represents a valid NDJSON file, so they contain a JSON text in every line followed by a newline character.

Every JSON text in a MAG file is a JSON object, that describes a paper available at https://academic.microsoft.com/home. The object can contain up to 22 different keys. Some common example keys:

- title
- authors
- year
- keywords
- abstract
- ...

3. Requirement Analysis

A requirement analysis is important to understand what exactly needs to be achieved as the result of a project. The requirements are divided into two different groups, functional and non-functional requirements. While functional requirements describe how something has to work, non-functional requirements describe what else must be met, for example specific performance requirements. Also, a requirement analysis helps to understand how everything works before the project and what exactly is in and out of scope.
To give all this information the initial situation is explained, followed by the scope, functional and then non-functional requirements.

3.1 Initial Situation

Now that the setting and the background of this thesis is clear, it is important to understand how the Karbonit JSON parser is structured and how it works.
The parser follows a straight forward approach to parse the data based on the JSON data structure. It operates single threaded by tokenizing the content of the JSON file and builds up the spanning tree structure afterwards.

The whole procedure shown in Figure 3.1 on the following page[1] can be split into four steps:

- Check for errors in input JSON (Condition ”Empty buffer?”)
- Get all tokens from input JSON (Process ”Tokenize”)
- Go through every token and build up the tree step by step (Process ”Interpret stored tokens”)
- Return the tree

[1] Data, i.e. the input, is available throughout the entire procedure, it is only mentioned if needed by a process.

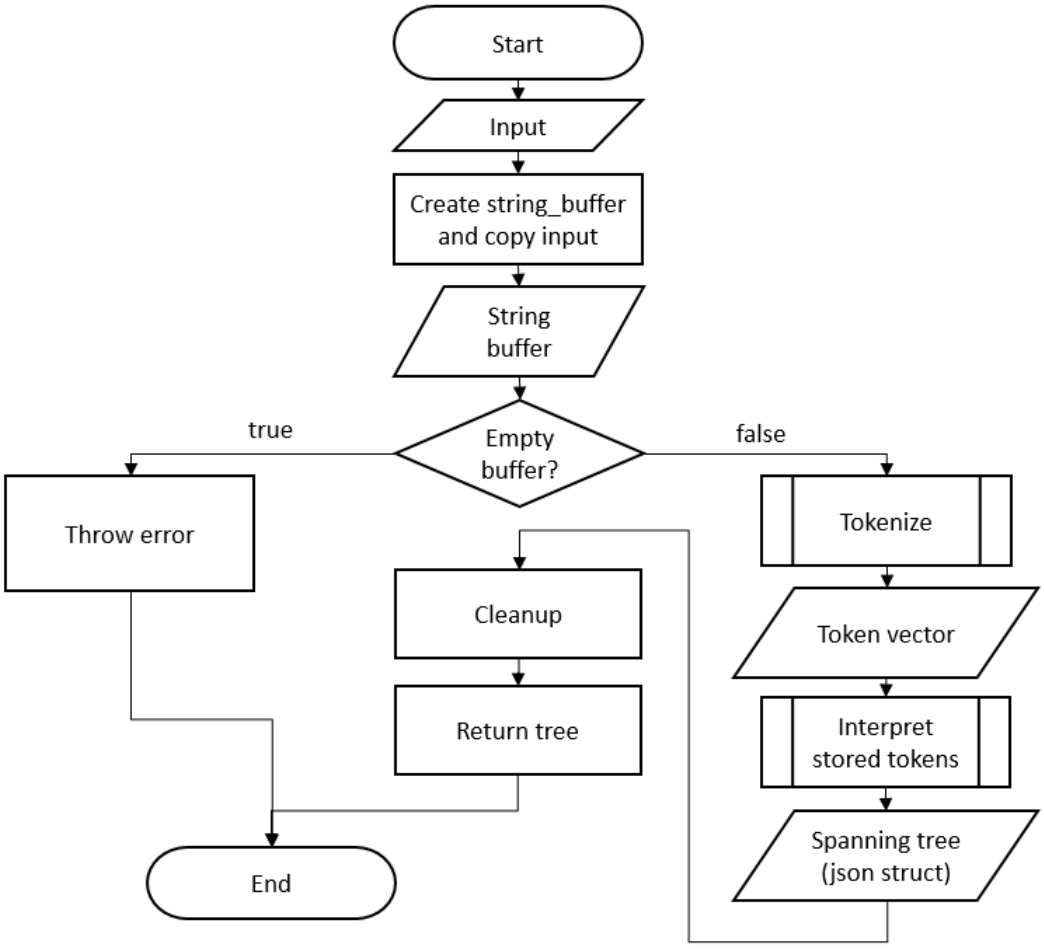


Figure 3.1: A flowchart describing the procedure to parse JSON input

The following sections describe those steps more detailed and show important parts of the source code that will be target or at least important for adjustments to the parser. The processes ”Tokenize” and ”Interpret stored tokens” are defined in Section 3.1.4 Tokenizer and Section 3.1.5 Interpreter.

3.1.1 Structs

There are some structs defined, that are important for the parser to function. They store information about the input JSON, the current position in the input and the resulting tree structure.

3.1.1.1 JSON Token Types

```
1  typedef enum json_token_type {
2          OBJECT_OPEN,
3          OBJECT_CLOSE,
4          LITERAL_STRING,
5          LITERAL_INT,
6          LITERAL_FLOAT,
7          LITERAL_TRUE,
8          LITERAL_FALSE,
9          LITERAL_NULL,
10         COMMA,
11         ASSIGN,
12         ARRAY_OPEN,
13         ARRAY_CLOSE,
14         JSON_UNKNOWN
15 } json_token_e;
```

Listing 3.1: Enum json_token_type

The enum from Listing 3.1 defines what exactly is a token in this context. Every token type, except the JSON_UNKNOWN, has a string representation, or at least a pattern to determine of what type the current token is. The OBJECT_OPEN token is represented by ’{’ in the input JSON text, and a LITERAL_STRING token matches the rule string = ”” characters ”” introduced in Chapter 2 Background.

3.1.1.2 JSON Token

```
1  typedef struct json_token {
2          json_token_e type;
3          const char *string;
4          unsigned line;
5          unsigned column;
6          unsigned length;
7  } json_token;
```

Listing 3.2: Struct json_token

The struct json_token from Listing 3.2 stores the information of one token. In addition to the token type, it stores also the string this token represents and the exact position in the input JSON text. The position is specified by:

- line - current line in the input
- column - position of first char in the current line
- length - amount of chars that belong to the token

3.1.1.3 JSON Tokenizer

```
1 typedef struct json_tokenizer {
2         const char *cursor;
3         json_token token;
4 } json_tokenizer;
```

Listing 3.3: Struct json_tokenizer

The struct json_tokenizer (Listing 3.3) stores a pointer to the current position in the input JSON text and the current token. During tokenization, the pointer is used to iterate through the input.

3.1.1.4 JSON Parser

```
1 typedef struct json_parser {
2         json_tokenizer tokenizer;
3 } json_parser;
```

Listing 3.4: Struct json_parser

The struct json_parser (Listing 3.4) only stores the tokenizer and is one of the most top-level structs currently implemented. This struct can be altered, if more information have to be stored that are needed throughout the entire parse process. It is accessible for the most parts of the parser.

3.1.1.5 JSON

```
1 typedef struct json {
2         json_element *element;
3 } json;
```

Listing 3.5: Struct json

The struct from Listing 3.5 is used to build up the result and to return it. There are way more structs implemented that define the tree structure, like the struct json_element of which an instance is stored in the json struct. The tree structure is described in Section 3.1.6 Tree Structure.

3.1.2 Parse Function

The parse function (see Listing 3.6) is the main function called, if a JSON text has to be analyzed and restructured for the further use of the information it contains.

```
1 bool json_parse(json *json,
2                 json_err *error_desc,
3                 json_parser *parser,
4                 const char *input)
```

Listing 3.6: Function json_parse

Json_parse needs some input:

- json - Pointer, used to store the parse result
- error_desc - Used to return information if an error occured
- parser - Pointer to the parser instance that will be used
- input - Pointer to the input JSON text, that has to be parsed

3.1.3 Error Check

To check if the input is not empty is the first thing, that happens after calling json_parse.

```
1  str_buf str;
2  str_buf_create(&str);
3  str_buf_add(&str, input);
4  str_buf_trim(&str);
5  if (str_buf_is_empty(&str)) {
6      set_error(error_desc, NULL, "input str_buf is empty");
7      str_buf_drop(&str);
8      return false;
9  }
10 str_buf_drop(&str);
```

Listing 3.7: Error Check

The error check from Listing 3.7 performs the following steps:

- Create a string buffer (lines 1-2)
- Copy input to buffer (line 3)
- Remove leading and trailing whitespaces (line 4)
- Check if buffer is empty (lines 5-9)
- return error or drop buffer and continue (line 10)

3.1.4 Tokenizer

The Tokenizer collects all the tokens the input JSON text contains (see the flowchart in Figure 3.2 on the next page).

To get the next token (step "Get next token from input if available"), the tokenizer checks what char is at the current position in the input. Depending on that char different actions are performed.
If the char equals:

- '\0' - return NULL, which indicates, that the end is reached
- '\n' or '\r' or whitespace - resume with next token
- '{' or '}' - return OBJECT_OPEN or OBJECT_CLOSE
- '[' or ']' - return ARRAY_OPEN or ARRAY_CLOSE
- ':' - return ASSIGN
- ',' - return COMMA
- '"' - call function parse_string_token

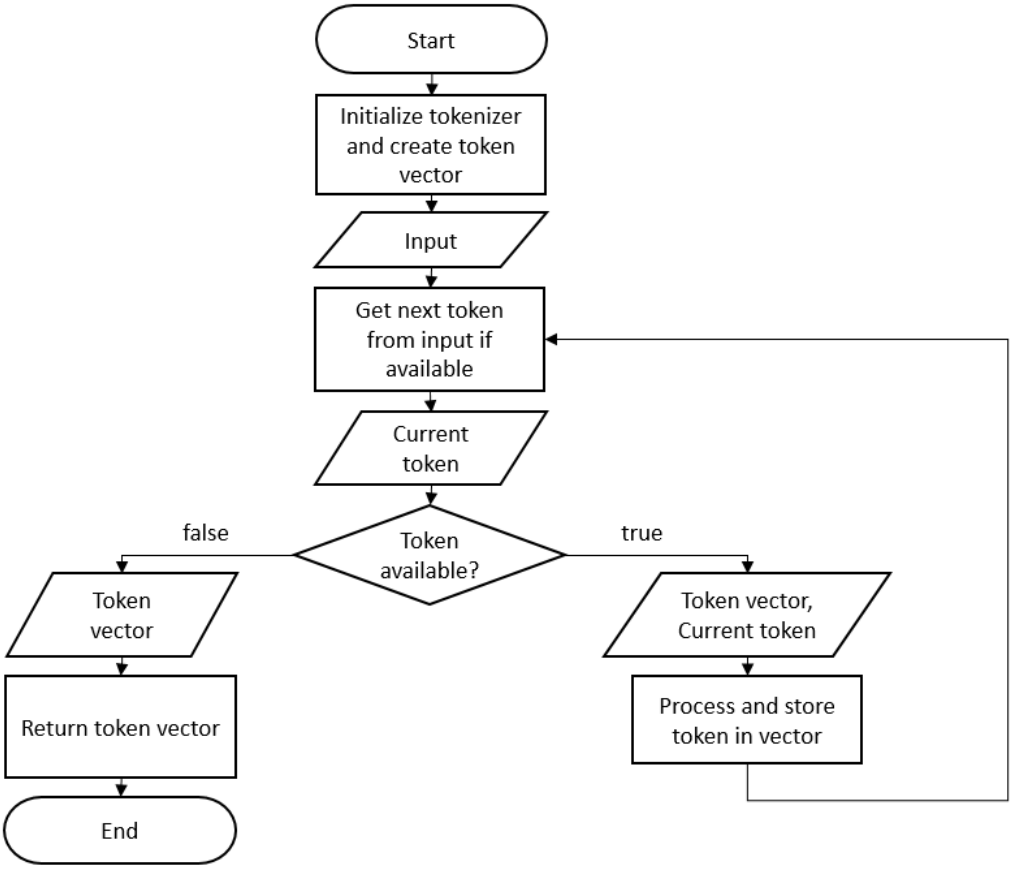


Figure 3.2: A flowchart describing the procedure to tokenize JSON input

- 't' - check if LITERAL_TRUE and return, else JSON_UNKNOWN
- 'f' - check if LITERAL_FALSE and return, else JSON_UNKNOWN
- 'n' - check if LITERAL_NULL and return, else JSON_UNKNOWN
- '-' or digit - check if LITERAL_INT or LITERAL_FLOAT and return, else JSON_UNKNOWN

Only in the case char equals "" a function is called. This function not only has to search for the end of the string, it also has to handle escaped chars. To do this the function parse_string_token checks the next chars until it discovers a "", that is not escaped by one or more leading '\'. A brief example: \" or \\\" are escaped quotes, while " or \\" are not and will end the current string. So also the amount of '\' has to be considered.
The parse_string_token function currently is able to save the last four chars to check whether a quote is escaped or not.

Every token is stored in a vector named token_stream that is used later by the interpreter.

3.1.5 Interpreter

The next step after all tokens are determined is to interpret those tokens and build up the tree structure (see the flowchart in Figure 3.3 on the following page).

To do this it is necessary to go through every token stored in the vector token_stream and add a new node to the tree, that represents the token type.
Simplified it works as follows:

1) Get next token
2) Check for token type
3) Depending on token type do:
   - For strings, floats, true, false and null store the corresponding value as the value of the current element
   - For arrays and objects add new child elements to the current element and repeat the whole process with every child

The tree structure is build up recursively from top to bottom.
After finishing this process all parent nodes have a connection to their child nodes. The interpreter now adds the connection from the child nodes to their parents and returns the result afterwards.

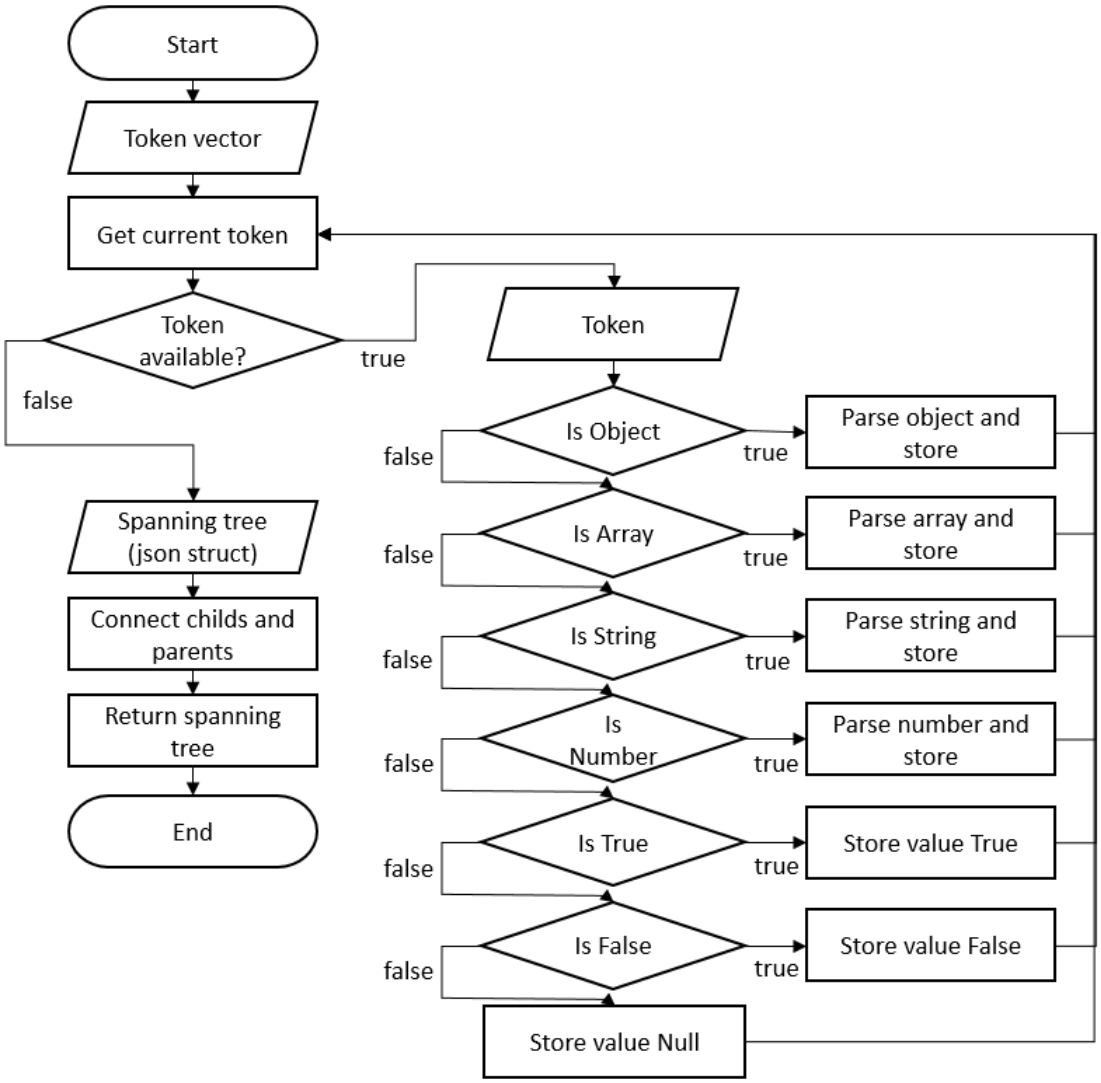


Figure 3.3: A flowchart describing the procedure to interpret the collected tokens

3.1.6 Tree Structure

To understand what is the result of this parsing process it is also important to know, how the nodes itself are structured and what information they store.
The struct that contains the result was already mentioned in Section 3.1.1.5 JSON. It stores a pointer to a json_element, which is the head node of the tree structure.

Every json_element stores the following information:

- Of what type is this element
- Information about the parent
- Information about the value (which is a json_node_value)

A json_node_value stores:

- Information about the parent
- Of what type it is
- The value of this node

The json_node_value contains a pointer to a json_array, json_object, json_string, json_number and a void pointer, no matter of what type the node_value is. To get the actual value the pointer corresponding to the type has to be dereferenced.

A json_string or json_number stores:

- Information about the parent
- The value it represents

A json_arrays or json_object stores:

- Information about the parent
- A vector that contains all elements/members that the array/object contains

Members are similar to elements, but they store a key-value-pair, not only a value.

A simplified example helps to understand the result better. The JSON input "[1, 2, 3]" is used to generate the result from Figure 3.4 on the next page.

The structure shown in Figure 3.4 on the following page is simplified to show the important information. Pointer to addresses are represented by the letters assigned to every element of this structure.
At the top is an instance of the json struct, which points to a json_element. The json_node_value this json_element contains, specifies in this case, that the input contains an array and points to the json_array. The json_array points to a json_elements instance, that stores all elements of the input array in a vector of type json_element.

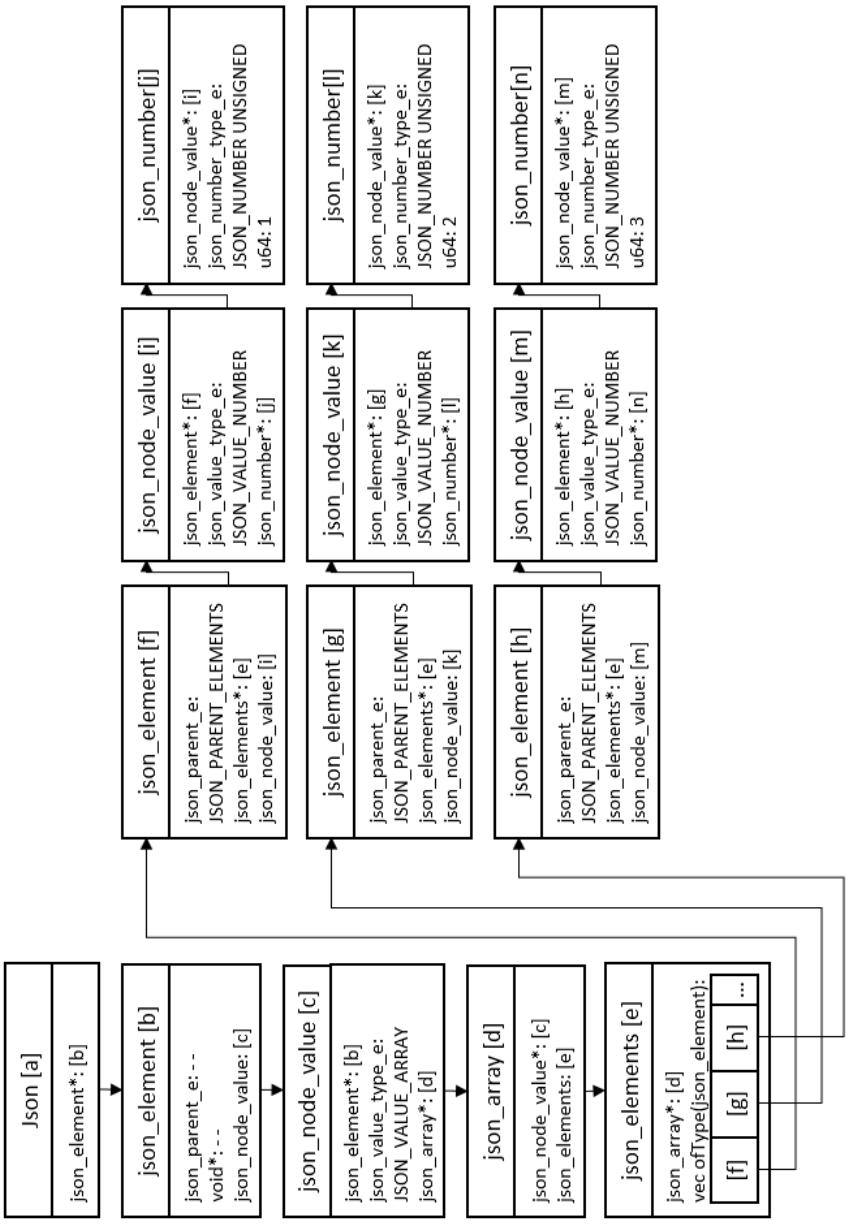


Figure 3.4: An example that shows the tree structure

3.2 Scope

Like stated in Chapter 1 Introduction the main focus of this bachelor thesis is the Karbonit JSON parser. The whole process of parsing JSON starts with calling the json_parse function and ends if the function returns an error or the resulting tree structure.
Everything not included in this process is out of scope, for example the further usage of parsed results. Except, other parts of the Karbonit project are affected by some changes. In this case the affected source code will be altered, so that everything runs as expected.
Additionally for this bachelor thesis it will be irrelevant what happens with the results after parsing is finished. Every result can be dropped the moment it was returned and does not have to be saved.

3.3 Functional Requirements

In addition to the functional requirement to enable the parser to parse NDJSON files (mentioned in Chapter 1), the user also has to be able to configure the parser. Both requirements are further explained in the following sections.

3.3.1 Parse NDJSON files

NDJSON files have some advantages over regular JSON files. They may are not as common, but parsing them is a real use case. The Karbonit JSON parser has to be able to parse them.
Depending on the situation it may be useful to parse line by line in sequence or randomly. Both variants should be possible.

3.3.2 Enable Parser Configuration

Depending on the situation, it may be useful to be able to configure the way the parser handles the input. If the input is a JSON file the parser should not try to parse line by line. If the input is a NDJSON file it can make sense to parse multiple lines parallel.
Every new functionality added to the parser has to be optional for parsing a file.

3.4 Non-functional Requirements

Unlike functional requirements, non-functional requirements do not expand the functionality of the parser. In this case they are performance goals. What exactly those goals are is described in the next section.

3.4.1 Performance

As stated in Chapter 1 Introduction there are some indicators that can be used to determine if the parser was improved or not. These include:

- A lower memory consumption while parsing

- Less time needed to parse a JSON file
- Able to parse larger amounts of data

Due to the second functional requirement Enable Parser Configuration the performance of the parser depends on the configuration chosen by the user. The non-functional requirements are met, if all of following statements are fulfilled by some possible configurations of the parser. It is not necessary to fulfill all at the same time.

- In the worst case, the maximum amount of space required throughout the entire parse process is equally as high, as that of the original JSON parser.
- The throughput of the resulting parser is higher than the throughput of the original parser.
- The resulting JSON parser is able to parse a larger amount of data at once, than the original JSON parser.

3.5 Summary

This chapter gave important information about the Karbonit JSON parser. How it works and how it structures data that was retrieved from the input JSON file. The implementations made in the following chapters are based on this knowledge.
Additionally the scope of this thesis, and functional as well as non-functional requirements were defined, which are later used in Chapter 5 to evaluate the results of following implementations.

4. Implementation

This chapter contains all implementations made in this thesis. It starts with enabling the parse of NDJSON files and parallelization, fixes an issue within the existing Karbonit JSON parser and it is described how a statistic helps to lower the memory consumption.

4.1 Parse Line by Line

The section Parse Line by Line describes the changes, that are made to enable the parse of NDJSON files. First of all, it is described how the parser currently behaves and what problems have to be solved. After that, some possible ways to solve these problems are listed and compared. The best solution is selected and the implementation is documented step by step.

Before making changes to the original parser to enable it to parse NDJSON files, it is important to understand, what is the main problem, that has to be solved.
What will happen if the Karbonit JSON parser has to parse a NDJSON file?
It will start to parse the input char by char/token by token, as it is intended to do. But if it reaches the end of the first line it also has reached the end of the first valid JSON text, that means from now on it only expects whitespace characters as defined by the JSON structure (see Section 2.1.1 JSON).
A line of an NDJSON file would end with an '\n' and the next line again can contain a valid JSON text. If the parser reads the '\n' it interprets it as a regular whitespace character and the next JSON text causes an error, because the parser strictly follows the rules that define the JSON structure. At this point it assumes, that the input is invalid.

Like mentioned in Section 2.1.2 it would be possible to create an JSON array that contains every JSON text of the NDJSON input file. The Karbonit JSON parser can parse this array, but the advantages a NDJSON file has over a regular JSON file would be lost. The parser would see this array as a whole, not every element in this array as a separate part.

At this point, there are multiple options, to enable the Karbonit JSON parser to parse NDJSON files line by line and use the advantages of these files.

- Before starting to parse, all '\n' can be replaced with '\0', so that the parser will stop at the end of every line. After the parser finished the current line the pointer to the input has to be set to the start of the next line. Repeat this until the last line was parsed.
- Get the current line and write it to a temporary file, string buffer or anything else, that can store a string. Start to parse the temporary file or string buffer, clear it and repeat it with the next line, until the last line was parsed.
- Alter the current parser implementation, so that the length of the input (l - Length of current line) can be specified. The parser will stop if l chars were read and parsed. Set the pointer to the input to the start of the current line, get the length of the current line (l) and start to parse only this line. Repeat it until the last line was parsed.

To choose the best option the advantages and disadvantages of every option have to be considered.
The first option requires to alter the input. If the changes were not undone, the file content would be lost, except the first line. To use a copy of the input would be possible, but this would double the memory used only to store the input. Apart from that, this option would only require to set the pointer to the input to specific positions in the input.
The second option uses files or string buffers to temporarily store the lines, which then just have to be parsed. But there will be costs to create or clear files or the buffer.
The third option is the only option, that requires to alter the parser directly. Some minor changes have to be done to the way the parser works. Again the pointer to the input has to be set to different positions multiple times.

Because the third option just stores the length of the current line, not a duplicate of the entire line, like in the second option, the third option would use less additional memory. The first option will not be useful, if the input is lost after parsing it. Due to this the third option will be implemented.

The procedure will be implemented as shown in Figure 4.1 on the next page. A pointer (line_start) stores the position of the current line and a counter (line_length) is used to get the length of the current line. Every char in the input is read until either a '\n' or '\0' is reached, which indicates the end of the current line. If one of those chars is reached, the current line is parsed and the pointer to the line position is set to the first position after the end of the current line. The counter to get the length of the line is set to 0 to start with the next line.
Currently the Karbonit JSON parser is unable to parse only a given count of chars, so some changes to the parser itself are necessary.

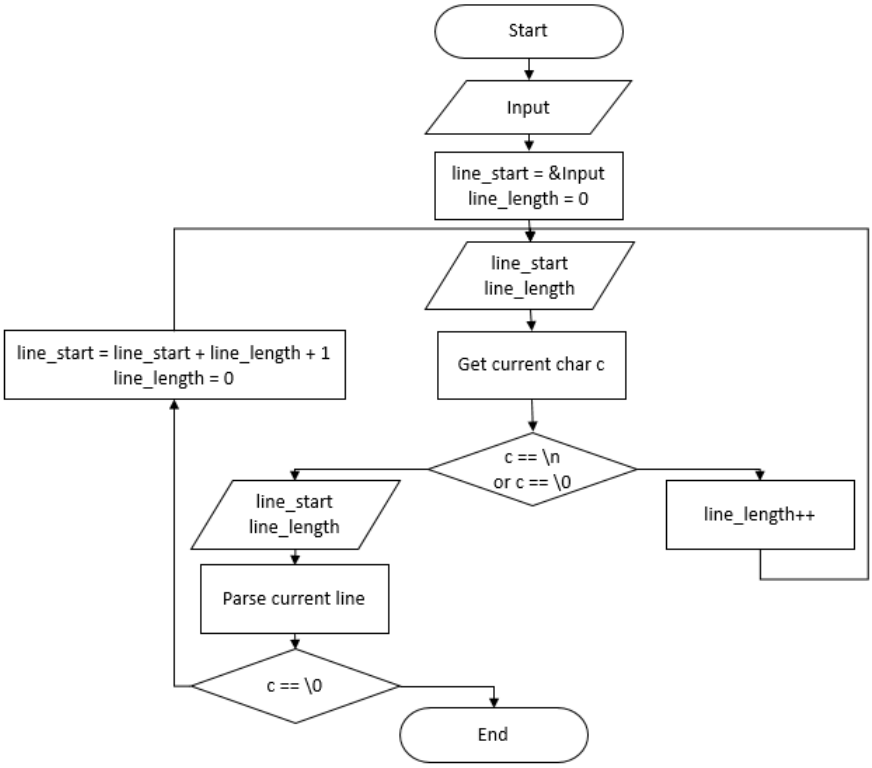


Figure 4.1: A flowchart describing the procedure to parse line by line

The following steps are needed to implement the third option:

- Store the input length
- Use input length to determine whether to continue or stop
- Write a function that splits the input file in separate lines and parses them

Store the input length

First thing to check is, which functions use the input length to replace the strlen-call with a given input length. Only functions to tokenize the input are relevant, because only the stored tokens are used after tokenizing, not the entire input itself.
To get the input length the current parser calls the function strlen and hands over a pointer to the current position in the input. This returns the amount of chars between this position and '\0', which indicates the end of the input.
The tokenizer uses at least three different functions to process the input. However, only the function json_tokenizer_next (see Listing 4.1) uses the input length.

```
1 const json_token
2         *json_tokenizer_next(json_tokenizer *tokenizer)
```

Listing 4.1: Function json_tokenizer_next

The specified input length can be handed over to this function by simply creating a new parameter, that stores this information. Another way is to store this information

inside the json_tokenizer struct. In this case the struct is used. It already stores a pointer to the input, so any information about this input should be stored the same way.
The following changes are made:

1) Add a new attribute (charcount) to the struct json_tokenizer.
2) Initialize the new attribute when the json_tokenizer is initialized.
   - Use strlen, if the input length is not specified.
   - Use the given input length, if it is handed over. To do this a new function is created based on the function that currently initializes the json_tokenizer.
3) The function json_parse_limited is created based on json_parse.
   json_parse is called to parse an input if no input length is specified, otherwise json_parse_limited is used.

To avoid source code redundancy in step three, two more functions were created. The function json_parse_check_input checks, if the input is not empty and json_parse_input that contains any instruction after initializing the json_tokenizer. So json_parse and json_parse_limited call json_parse_check_input, the respective function to initialize the tokenizer and return the result of json_parse_input as shown in Listing 4.2.

```
1  bool json_parse(json *json, json_err *error_desc, json_parser *
       parser, const char *input)
2  {
3          if(!json_parse_check_input(error_desc, input, 0))
4          {
5              return false;
6          }
7
8          json_tokenizer_init(&parser->tokenizer, input);
9
10         return json_parse_input(json, error_desc, parser);
11 }
12
13 bool json_parse_limited(json *json, json_err *error_desc,
       json_parser *parser, const char *input, size_t charcount)
14 {
15     if(!json_parse_check_input(error_desc, input, charcount))
16     {
17         return false;
18     }
19
20     json_tokenizer_init_limited(&parser->tokenizer, input,
           charcount);
21
22     return json_parse_input(json, error_desc, parser);
23 }
```

Listing 4.2: Functions json_parse and json_parse_limited after greater redundancy of the source code is prevented

Use input length to determine whether to continue or stop

Now that the input length is stored and available in the json_tokenizer, every call of strlen to get the input length has to be replaced with tokenizer->charcount. To keep track of the amount of chars that still have to be processed the charcount attribute is decreased every time the pointer to the input (tokenizer->cursor) is set to a new position. Every tokenizer->cursor += n is now followed by
tokenizer->charcount -= n.
Currently to end the tokenizer the char '\0' has to be reached. This condition is replaced so that the tokenizer now stops, if tokenizer->charcount reaches 0.

Write a function that splits the input file in separate lines and parses them

```
1  bool json_parse_split(const char *input, size_t size_input)
2  {
3      size_t i = 0;
4      size_t lastPart = 0;
5      int l = 0;
6      const char* currentPart;
7      bool end_parse = false;
8      while (!end_parse)
9      {
10         if((input[i] == '\n') || (input[i] == '\0'))
11         {
12             if((input[i] == '\0') || (i == size_input))
13             {
14                 end_parse = true;
15             }
16
17             //set pointer to the beginning of current part
18             currentPart = input + lastPart;
19
20             //rec doc;
21             l++;
22
23             struct json data;
24             json_err err;
25             json_parser parser;
26             if(json_parse_limited(&data, &err, &parser, currentPart
                   , i-lastPart)) {
27                 json_drop(&data);
28             }
29             lastPart = i+1;
30         }
31         i++;
32     }
33     return true;
34 }
```

Listing 4.3: Function json_parse_split

The function json_parse_split shown in Listing 4.3 requires a pointer to the input and the size of the whole input. The procedure behind this function is already described in Figure 4.1 on page 25.

It is now possible to parse NDJSON files with the Karbonit JSON parser. Like described in Section 3.2 Scope, for this thesis it is not important to store the results or use them further, so they are dropped immediately after json_parse_limited returns them. To use the results, some logic has to be added before dropping them or they have to be stored in some way.

4.2 Multiple Threads

Based on the implementation made in Section 4.1 Parse Line by Line, this section aims to speed up the whole process to parse every single line of an NDJSON file. The following paragraphs describe, why it is possible to parse multiple lines parallel even if the results have to be used further, the concept of how to parse parallel and how it is implemented within the given Karbonit JSON parser.

Why it is possible to parse parallel

In general, the JSON texts a NDJSON file contains are independent of each other. Every line can be parsed separately. So the sequence in which the lines are parsed does not matter. As long as every line is parsed it is okay to start the parse anywhere in the file, to skip lines to return later or to jump to random lines.
But even if the results have to be stored or processed further it is possible to parse parallel. The sequence in which the lines occur is represented by the line number. The line number can be used as a key in the Karbonit database that gets the parse result of the current line assigned. By sorting the results by their keys or searching for a special line number will return the expected result.

The concept of how to parse lines parallel

To explain and understand the concept it is best to use an example. The following NDJSON file (Listing 4.4) will contain six lines of valid JSON text.

```
1 {"title":"t1"}
2 {"title":"t2"}
3 {"title":"t3"}
4 {"title":"t4"}
5 {"title":"t5"}
6 {"title":"t6"}
```

Listing 4.4: NDJSON Example to parse parallel

Assuming the parser has a threadpool that manages n threads ($n \geq 2$). Every thread can handle one task at a time. Supposing $n = 2$, then the parser has two threads (thread1 and thread2) and can handle two tasks at the same time.
Possible methods to parse:

1) Parse the next available line
   In this case the parser starts by parsing lines 1 and 2. Each one is parsed in a separate thread. If one of the lines is finished, the next available line

can be parsed by the idle thread. This would be line 3 and after that line 4. This means that every task has to parse exactly one line, so six tasks are needed (named t1 - t6). Which task is processed by which thread depends on what task is finished first. So thread1 may processes t1 and t4, while thread2 processes t2, t3, t5 and t6.

2) Split lines in n groups and parse groups
Another possible method is to split the six lines in two groups. Group1 contains the lines 1 to 3 and group2 contains 4 to 6. A task is to parse a whole group, so only two tasks are needed in this case (t1 and t2). Thread1 will process t1 and thread2 will process t2.

The first method has an advantage, if the length of every line varies widely. The next line is parsed as soon as possible and no thread has to wait for new tasks, except there are no more lines to parse. A task would use the function json_parse_limited, because it only has to parse one line.
The second method is easier to handle, if the NDJSON file contains many lines. Even if the file contains a million lines, only two tasks are generated, while the first method generates a million tasks. To many tasks can cause problems for example if they need to much memory.
There is no best option. The second is slower than the first and the first may cause some problems. To avoid those disadvantages both methods are combined, so the user can decide, what is the best in his case.

The user has to decide how many threads should be used and how many tasks should be created. The number of threads still is called n and the number of tasks is called k. Behaviour depending on the parameters n and k: As seen in Table 4.1

ratio n : k	Behaviour of the parser
x : 1	No use of parallelization, only calls json_parse_split
1 : x	No use of parallelization, only calls json_parse_split
n = k	Is similar to the second method
n > k	n-k threads are unused, will be handled like n = k
n < k	Threads process more than one task (if new task is available)

Table 4.1: Behaviour depending on the amount of threads and tasks

there are some special cases. If either $n = 1$ or $k = 1$ applies it makes no sense to create a thread pool, threads and tasks ("x" can be replaced with any number). To create all this would just be overhead and perhaps it would run slower than just parsing one line at a time. So the parser will just parse the NDJSON file normally. If $n \geq k$ applies the parser will create k threads and splits the lines in as many groups, as threads are created, like described by method 2.
The last case $n < k$ is the best case to use parallelization. It can be assumed, that not all threads will finish at the same time. So while other threads still run, threads that finished their task can process the next task until no more tasks are available. In most cases the throughput will be better than if $n = k$.

The implementation

To group the lines and parse them parallel some functions can be used, that already exist in the Karbonit project. The function that will parse the groups is json_parse_split, that was implemented in Section 4.1. The thread pool, that manages all the threads and tasks already exists, like described in Section 2.3.4.
First to do now is to implement a function that manages everything. This function is called json_parse_split_parallel (see Listing 4.5), because it works like an extension to the json_parse_split function. This new function requires information about the input (pointer to the input and the input length) and the number of threads and tasks, that have to be created.

```
1  bool json_parse_split_parallel(const char *input, size_t size_input
       , size_t num_threads, size_t num_parts)
2  {
3      if(num_threads > num_parts)
4      {
5          num_threads = num_parts;
6      }
7
8      size_t size_part = 0;
9      size_t current_char_pos = 0;
10     size_t current_size = 0;
11     size_t i = 0;
12     const char* start_of_part = input;
13
14     thread_pool *pool = thread_pool_create(num_threads, 0);
15
16     //divide size by num_parts
17     size_part = size_input / num_parts;
18     size_part++;
19
20     //create Array of num_parts tasks
21     //task_handle hndl[num_parts];
22     thread_task tasks[num_parts];
23     parser_task_args task_args[num_parts];
24
25     while (i < num_parts)
26     {
27         //if out of bounds
28         if((current_size + size_part) > size_input)
29         {
30             current_char_pos = size_input - current_size;
31         }
32         //else skip size_part chars
33         else
34         {
35             current_char_pos = size_part;
36
37             //search for next EOL or End of String
38             for (; start_of_part[current_char_pos] != '\n' &&
                   LIKELY(start_of_part[current_char_pos] != '\0');
                   current_char_pos++) {}
39         }
40
41         //start_of_part - pointer to start of part
```

```
42          //end_of_part - position of end of part
43          task_args[i].start = start_of_part;
44          task_args[i].size = current_char_pos;
45          task_args[i].count = i+1;
46          current_size += current_char_pos;
47
48          tasks[i].args = (void *) &task_args[i];
49          tasks[i].routine = task_routine_lbl;
50          //thread_pool_enqueue_task(&tasks[i], pool, &hndl[i]);
51
52          start_of_part = start_of_part + (current_char_pos + 1);
53
54          i++;
55      }
56
57      thread_pool_enqueue_tasks_wait(tasks, pool, num_parts);
58      //thread_pool_wait_for_all(pool);
59      thread_pool_free(pool);
60      return true;
61  }
```

Listing 4.5: Function json_parse_split_parallel

The parameter num_parts equals the amount of groups and tasks.
Simplified the json_parse_split_parallel from Listing 4.5 on the preceding page has the following structure:

1) Check if a special case has to be handled based on num_threads and num_parts (lines 3 to 6)
2) Create the thread pool with num_thread threads (line 14)
3) Compute the expected size of every group of lines (lines 17 to 18)
4) Create an array to store num_parts tasks (line 22)
5) Create an array to store num_parts task arguments (line 23)
6) Use expected group size to build groups and create the tasks and arguments (lines 25 to 55)
7) Add all tasks to the thread pool and wait until the last task is finished (line 57)

To be more specific json_parse_split_parallel first checks num_threads and num_parts to determine what to do. The case that num_threads or num_parts is 1 is actually handled in a later section (Section 4.6) so that the right function for this case can be called immediately. However, should num_threads be larger than num_parts, num_threads will be set to the value of num_parts. All other cases do not need any actions.
The thread pool simply is created by calling the function thread_pool_create with num_threads as the first argument. The option to monitor the thread pool is turned off by handing over '0' as the second argument.

In step 3) the length of the entire input is divided by num_parts, to get the average group size.
Step 4) and 5) create arrays to handle the tasks and their arguments. The tasks are instances of the struct thread_task. The thread pool requires tasks to be thread_tasks. The task arguments however are unique in this case. To store all the information needed to process a task a new struct is created.

```
1 typedef struct parser_task_args{
2         const char* start;
3         size_t size;
4 } parser_task_args;
```

Listing 4.6: Struct parser_task_args

An instance of parser_task_args shown in Listing 4.6 stores a pointer to the start of the first line of a group and the amount of chars this group of lines contains.
To group the lines, a pointer to the start of the current group is stored in a temporary variable. From this position, with an offset of the average group size, the next '\n' or '\0' is searched and for every char that does not match, the size of this group is increased by 1. If the char matches, start and length of this group are stored in the corresponding instance of parser_task_args. Additionally the task is created, but to do this a task routine is required that is undefined by now. The task itself only has to call the function json_parse_split. The routine implemented to do this is named task_routine_lbl. It simply dereferences the pointer to the arguments and hands the start and length of the group it has to parse over to the parse function.

After all preparations are done all tasks can be added to the thread pool, so that they can be processed (step 7). The function thread_pool_enqueue_tasks_wait not only adds the tasks, it also waits until all tasks are finished.

4.3 Get Tokens and Interpret Them Immediately

Like described in Chapter 3 the Karbonit JSON parser parses the input in two steps. The first step is to tokenize the input and to store the tokens, the second one is to interpret every token and to build up the tree.
Because in step 2 every token is interpreted in sequence and it is not necessary to know information about the next token at any point, it is possible to combine those two steps. The following paragraphs describe the concept of how to combine those two steps, and how it is implemented.

The concept of how to combine both steps

The function that is mainly responsible for parsing is json_parse_input (see Listing 4.7). It was implemented in Section 4.1 Parse Line by Line, to prevent redundancy and is called by json_parse_limited and json_parse.

```
1 static bool json_parse_input(json *json, json_err *error_desc,
     json_parser *parser)
2 {
3     ...
```

```
4     while ((token = json_tokenizer_next(&parser->tokenizer))) {
5         if (LIKELY(
6                 (status = process_token(error_desc, token, &
                      brackets, &token_mem)) == true)) {
7             json_token *newToken = VEC_NEW_AND_GET(&token_stream,
                  json_token);
8             json_token_dup(newToken, token);
9         } else {
10            goto cleanup;
11        }
12    }
13    ...
14    if (!parse_token_stream(&retval, &token_stream)) {
15        status = false;
16        goto cleanup;
17    }
18    ...
19 }
```

Listing 4.7: Function json_parse_input

The first part of json_parse_input tokenizes the input (Listing 4.7 lines 4-12) and the second interprets the collected tokens (Listing 4.7 lines 14-17). Both parts are clearly separated from each other.
To combine the tokenizer and the interpreter, the entire parsing method must be changed. Instead of storing every token it is interpreted immediately.

The first step is to get the first token to decide whether a array or object node has to be created. In both cases a vector is created to store the elements/members and a function is called to tokenize and interpret further. For an array the function is called parse_array and for an object parse_object. The while loop from Listing 4.7 lines 4 to 12 is now located in parse_array and parse_object with the change that the body of this loop now interprets the tokens.
A ARRAY_OPEN or OBJECT_OPEN token causes the creation of an array or object node and a recursive call to the corresponding function. For every other value token a node is created and stored as an element or member at the current position in the array or object node vector.

The implementation

To combine the tokenizer and the interpreter is a major change to the parser. Because of that the relevant functions will be copied and only those copies are altered. The original parser with a separate tokenizer and interpreter can still be used afterwards. Every function used in this section that is copied or created ends with the suffix "_exp" which stands for "experimental" in this case.

First function to modify is json_parse_input.

```
1 static bool json_parse_input_exp(json *json, json_err *error_desc,
      json_parser *parser)
2 {
```

```
3       struct json retval;
4       ZERO_MEMORY(&retval, sizeof(json))
5       retval.element = MALLOC(sizeof(json_element));
6       const json_token *token;
7       int status;
8
9       struct token_memory token_mem = {.init = true, .type =
            JSON_UNKNOWN};
10
11      token = json_tokenizer_next_exp(&parser->tokenizer);
12      if (LIKELY((status = process_token_exp(error_desc, token, &
            token_mem)) == true)) {
13          switch(token->type){
14              case OBJECT_OPEN:
15                  retval.element->value.value_type =
                        JSON_VALUE_OBJECT;
16                  retval.element->value.value.object = MALLOC(sizeof(
                        json_object));
17
18                  parse_object_exp(parser, retval.element->value.
                        value.object, error_desc, &token_mem);
19                  break;
20              case ARRAY_OPEN:
21                  retval.element->value.value_type = JSON_VALUE_ARRAY
                        ;
22                  retval.element->value.value.array = MALLOC(sizeof(
                        json_array));
23
24                  parse_array_exp(parser, retval.element->value.value
                        .array, error_desc, &token_mem);
25                  break;
26              default:
27                  goto cleanup;
28          }
29      } else {
30          goto cleanup;
31      }
32
33      OPTIONAL_SET_OR_ELSE(json, retval, json_drop(json));
34      status = true;
35
36      cleanup:
37      return status;
38  }
```

Listing 4.8: Function json_parse_input_exp

The new function json_parse_input_exp from Listing 4.8 now only gets the first token (line 11) and then decides whether to create a JSON_VALUE_OBJECT node (lines 14-19) or a JSON_VALUE_ARRAY node (lines 20-25).
In case of an object the function parse_object_exp (see Listing 4.9) allocates space for an instance of json_members and calls parse_members_exp to parse the rest of the input.

```
1 static bool parse_object_exp(json_parser *parser, json_object *
      object, json_err *error_desc, struct token_memory *token_mem)
2 {
3     object->value = MALLOC(sizeof(json_members));
4 
5     return parse_members_exp(object->value, parser, error_desc,
          token_mem);
6 }
```

Listing 4.9: Function parse_object_exp

```
1  bool parse_members_exp(json_members *members, json_parser *parser,
       json_err *error_desc, struct token_memory *token_mem)
2  {
3      vec_create(&members->members, sizeof(json_prop), 20);
4      const json_token* delimiter_token;
5
6      do {
7          json_prop *member = VEC_NEW_AND_GET(&members->members,
               json_prop);
8          json_token keyNameToken = *json_tokenizer_next_exp(&parser
               ->tokenizer);
9
10         member->key.value = MALLOC(keyNameToken.length + 1);
11         strncpy(member->key.value, keyNameToken.string,
               keyNameToken.length);
12         member->key.value[keyNameToken.length] = '\0';
13
14         json_token assignment_token = *json_tokenizer_next_exp(&
               parser->tokenizer);
15         if (assignment_token.type != ASSIGN) {
16             return false;
17         }
18
19         json_token valueToken = *json_tokenizer_next_exp(&parser->
               tokenizer);
20         size_t pred_count = 0;
21         switch (valueToken.type) {
22             case OBJECT_OPEN:
23                 ...
24             case ARRAY_OPEN:
25                 ...
26             case LITERAL_STRING:
27                 ...
28             case LITERAL_INT:
29             case LITERAL_FLOAT:
30                 ...
31             case LITERAL_TRUE:
32                 ...
33             case LITERAL_FALSE:
34                 ...
35             case LITERAL_NULL:
36                 ...
37             default:
38                 return ERROR(ERR_PARSETYPE, NULL);
39         }
40
41         delimiter_token = json_tokenizer_next_exp(&parser->
               tokenizer);
42     } while (delimiter_token != NULL && delimiter_token->type ==
           COMMA);
43
44     return true;
45 }
```

Listing 4.10: Function parse_members_exp

parse_members_exp (Listing 4.10) creates a vector with a default capacity of 20 to store the key-value-pairs the JSON object contains. Then it parses the key-value-pairs in a while loop. It gets the token (lines 8-12), the assignment token (lines 14-17) and then the value (lines 19-39). Depending of the value type a new child node is created, or just the value is stored.

In case of an array the function parse_array_exp (see Listing 4.11) creates a vector and calls parse_elements_exp.

```
1 static bool parse_array_exp(json_parser *parser, json_array *array,
      json_err *error_desc, struct token_memory *token_mem)
2 {
3     vec_create(&array->elements.elements, sizeof(json_element),
          250);
4     return parse_elements_exp(&array->elements, parser, error_desc,
          token_mem);
5 }
```

Listing 4.11: Function parse_array_exp

```
1  static bool parse_elements_exp(json_elements *elements, json_parser
       *parser, json_err *error_desc, struct token_memory *token_mem)
2  {
3      const json_token* delimiter;
4      do {
5          json_token current = *json_tokenizer_next_exp(&parser->
               tokenizer);
6          if (current.type != ARRAY_CLOSE && current.type !=
               OBJECT_CLOSE) {
7              if (!parse_element_exp(parser, VEC_NEW_AND_GET(&
                   elements->elements, json_element), current,
                   error_desc, token_mem)) {
8                  return false;
9              }
10         }
11         delimiter = json_tokenizer_next_exp(&parser->tokenizer);
12     } while (delimiter != NULL && delimiter->type == COMMA);
13
14     return true;
15 }
```

Listing 4.12: Function parse_elements_exp

The function parse_elements_exp (Listing 4.12) basically works in the same way as parse_members_exp. The switch statement from Listing 4.10 in lines 21 to 39 is evaluated by the function parse_element_exp. Further actions are also initiated by this function.

4.4 Check for escaped Quotes

In Section 3.1.4 Tokenizer a function named parse_string_token is mentioned. Because of its implementation there is a good chance, that errors occur while parsing a valid JSON text. This section explains what exactly causes the problem, why this problem is relevant for this thesis and how the problem is solved.

The problem

To parse an input JSON text the tokenizer first splits the input into the several tokens it contains. Some of those token types are simple to determine, for example OBJECT_OPEN, which is indicated by the single char '{'. Unlike this token type, a token of type LITERAL_STRING can contain a variable amount of chars. A LITERAL_STRING token starts with a quote (char "") and ends with it, but it is also possible that the token itself contains an escaped quote. That this quote is escaped means that it does not affect the structure in any way, it is just part of the text.
Some examples:

- "This is a valid string."
- "\"Hello\" Still a valid string."
- "\\\"Hello\" Still a valid string."
- "\\"Hello\" Not a valid string."

Like stated in the strings itself, the first three examples are valid and only the last one causes some trouble.
All strings start and end with a quote, so this criteria is matched. The first example contains no quotes, so there can not be any problem with the structure. The next examples all contain two quotes, that should be part of the text, but that is only the case in the second and third example. The amount of leading backslashes determines whether a quote is escaped or not. A backslash escapes the next char and an escaped backslash is just part of the text and has no effect on the following char.

More examples that show what exactly is escaped:

- \" - The quote is escaped.
- \\" - The second backslash is escaped.
- \\\" - The second backslash, as well as the quote is escaped.
- \\\\" - Second and fourth backslash are escaped.
- \\\\\" - Second and fourth backslash, as well as the quote is escaped.

Every time a backslash is added in front of the quote it either switches the quote from escaped to not escaped or vice versa. A simple rule can be derived from this pattern: If the amount of leading backslashes is odd, the char is escaped. Because there is no maximum amount of leading backslashes, it is not possible to decide if the current quote is escaped, by simply looking at the last four chars. The current implementation of parse_string_token does exactly this. parse_string_token decides that every quote that follows four or more backslashes ends the current string. If the string still goes on after this quote the tokenizer will throw an error and the input can not be parsed.

Why is this problem relevant?

Regardless of how large the input is, if it contains a quote, that is escaped by an odd number of leading backslashes greater than or equal to 5 ,the current Karbonit JSON parser is unable to parse it. The time needed to parse this input can be seen as infinite. The only way to parse the input is to alter the input and decrease any odd amount of consecutive backslashes to 1 or 3. After solving the problem the time needed to parse the input will be decreased from infinite to a finite number, which is an improvement.

Solution of this problem

To solve this problem the pattern earlier described can be used. Either the amount of consecutive backslashes is tracked, or even simpler, if a backslash is read, just skip the next char. In this case the second option is implemented because it is easier to implement, no values have to be stored and it even skips chars, which speeds up the loop a bit.

```
1  static void
2  parse_string_token(json_tokenizer *tokenizer, char c, char
       delimiter, char delimiter2, char delimiter3,
3                     bool include_start, bool include_end)
4  {
5          size_t step = 0;
6
7          tokenizer->token.type = LITERAL_STRING;
8          if (!include_start) {
9              tokenizer->token.string++;
10         }
11         tokenizer->token.column++;
12         c = *(++tokenizer->cursor);
13         tokenizer->charcount--;
14
15         while ((c != delimiter && c != delimiter2 && c !=
               delimiter3) && c != '\r' && c != '\n') {
16             if(c == '\\')
17             {
18                 //skip next char
19                 tokenizer->token.length++;
20                 c = *(++tokenizer->cursor);
21                 tokenizer->charcount--;
22             }
23             tokenizer->token.length++;
24             c = *(++tokenizer->cursor);
25             tokenizer->charcount--;
26         }
27
28         if (include_end) {
29             tokenizer->token.length++;
30         } else {
31             tokenizer->cursor++;
32             tokenizer->charcount--;
33         }
34
35         step = (c == '\r' || c == '\n') ? 1 : 0;
```

```
36          tokenizer->cursor += step;
37          tokenizer->charcount -= step;
38  }
```

Listing 4.13: Function parse_string_token

The function parse_string_token (Listing 4.13) now does the following:

- Get the first char after the quote that started the LITERAL_STRING token.
- While the current char does not match the char that is searched for (in this case "") do:
  - If the current char is a backslash, get the next char.
  - Get the next char.

4.5 Build up Statistics

This section aims to improve the interpreter. There are some unnecessary costs connected to the creation of an array or object node. The following paragraphs explain why there are unnecessary costs, how those costs may be prevented by analyzing the structure of the input and how it is implemented in this thesis.

Reasons for unnecessary costs

If an ARRAY_OPEN or OBJECT_OPEN token is read the interpreter creates a new node that contains an vector to store all contents of the current array or object. A vector has to be initialised with a certain capacity. If the capacity does not match the amount of elements/members there are unnecessary costs either to create a vector with to much capacity or to resize the vector.

In case an array node has to be created the function parse_array is called.

```
1  static bool parse_array(json_array *array,
2                          vec ofType(json_token) *token_stream,
3                          size_t *token_idx)
4  {
5          ...
6          vec_create(&array->elements.elements,
7                     sizeof(json_element),
8                     250);
9          if (!parse_elements(&array->elements,
10                             token_stream,
11                             token_idx)) {
12                 return false;
13         }
14         ...
15         return true;
16 }
```

Listing 4.14: Function parse_array

And in the case an object node has to be created the function parse_object is called.

```
1  static bool parse_object(json_object *object,
2                           vec ofType(json_token) *token_stream,
3                           size_t *token_idx)
4  {
5          ...
6          /** test whether this is an empty object */
7          json_token token = get_token(token_stream,
8                                       *token_idx);
9
10         if (token.type != OBJECT_CLOSE) {
11                 if (!parse_members(object->value,
12                                    token_stream,
13                                    token_idx)) {
14                         return false;
15                 }
16         } else {
17                 vec_create(&object->value->members,
18                            sizeof(json_prop),
19                            20);
20         }
21         ...
22         return true;
23 }
```

Listing 4.15: Function parse_object

The vector created in the function parse_array always has a initial capacity of 250 vector elements of type json_element (see lines 6-8 in Listing 4.14). Other than the parse_array function the current parse_object function first checks if the object is empty (Listing 4.15 lines 6-10). If the object is not empty the function parse_members is called which then creates a vector with a capacity of 20 vector elements of type json_prop (Listing 4.16 lines 5-7).

```
1  bool parse_members(json_members *members,
2                     vec ofType(json_token) *token_stream,
3                     size_t *token_idx)
4  {
5          vec_create(&members->members,
6                     sizeof(json_prop),
7                     20);
8          ...
9  }
```

Listing 4.16: Function parse_members

But even if the object is empty a vector with the same capacity is created (Listing 4.15 lines 17-19). The capacity of this vector can be decreased to a maximum of 1, which should also be done, if an array is empty.
Empty arrays and objects are part of the problem that has to be solved in this section. They can be assigned to the first case in which less elements have to be stored than the vector capacity allows.

Now that it is described how the vectors are created and what their capacity is based on the node type, the effect of underfilled arrays and objects can be explained.

The Table 4.2 shows that the default vector of an array node has a size of 10000

node type	element type	element size	default vector capacity	vector size
array	json_element	40 Byte	250	10000 Byte
object	json_prop	64 Byte	20	1280 Byte

Table 4.2: Used space per element or prop vector

Byte and the vector of an object node has a size of 1280 Byte.
Besides this table, a modified MAG file will be used to explain more details. A MAG file normally has a NDJSON structure, but it only requires little changes to create a valid JSON file. Add square brackets at the start and end of the file and a comma at the end of every line, except the last line. So it will work for now.
Some facts about a single MAG file:

- contains 1000000 lines
- average filesize of approximately 2GB
- every line contains a valid JSON text (+ ',' because of modification)
- every JSON text is an object
- 22 possible keys for top level object
- good chance that for example the values for keys author, keywords, fos, references and url are arrays

There are 22 possible keys, so the default capacity of the object node vector does not deviate to much. Looking at the array node vectors belonging to the keys author and fos (field of study), it is pretty unlikely, that they have to store 250 elements. Lets take as a high average amount 20 elements for author and 50 for fos.

key	used capacity	unused capacity	unused allocated space
author	20/250	230/250	9200 Byte
fos	50/250	200/250	8000 Byte
			17200 Byte

Table 4.3: Array node vector - unused allocated space

Table 4.3 shows that in every line approximately 17200 Byte or 17 KB[1] of unused space is allocated. The example MAG file has 1000000 lines, so if the MAG file is parsed as a JSON file there the unused space is $17.200 Byte * 1.000.000 = 17.200.000.000 Byte = 17 GB$. Just to convert the 2GB JSON input would cause a overhead of more or less 17 GB unused memory. That does not include the actual information so the size of the parse result easily exceeds 17 GB.

[1] conversion factor 1000 instead of 1024: 1000 Byte = 1 KB

Resizing the vector because more elements need to be stored than the capacity of the vector allows may supports this problem. The grow factor of an vector is set to 1.7 by default (see Listing 4.17 line 4).

```
1 bool vec_create(vec *out, size_t elem_size, size_t cap_elems)
2 {
3         ...
4         out->grow_factor = 1.7f;
5         return true;
6 }
```

Listing 4.17: Function vec_create

If the vector has to store 251 elements, but only has a default capacity of 250, it is resized to a capacity of 425 ($250 * 1.7 = 425$). So there are 174 unused elements allocated only because one elements of the 251 did not fit into the vector.

Prevent those costs

The default vector size of 250 for array nodes and 20 for object nodes are estimated values that may have worked best with the test inputs used when the parser was developed. Since all inputs differ from each other a more flexible approach would be better.
A way to be more flexible is to build up a statistic, that represents the current input. The best case would be if the statistic can give the exact amount of elements for every vector. This would require a dynamic structure to store all the information, because JSON objects and arrays can be nested. The structure that will be implemented for the Karbonit JSON parser is not as dynamic, but uses way less memory.

An example JSON text to explain what information the statistic collects and how it handles the information:

```
1 [{"key1":"value1",
2   "key2":["value2.1", "value2.2"],
3   "key3":{"text":"hello", "number":[1234]}},
4  ["element1",
5   ["element2.1", "element2.2", "element2.3", "element2.4"],
6   {"text":"world", "number":[11, 75, 103]}]]
```

Listing 4.18: Example JSON for statistic

This example from Listing 4.18 contains an array with two elements. The first element is an object with three key-value-pairs. The second element is an array that contains a string, an array and an object.
Information about the arrays depending on their position:

- The top level array contains two elements.
  "[1, 2]"
- The first array on the second level contains three elements.
  "[..., [1, 2, 3]]"

- The first array on the third level contains two elements.
  "[{..., ...:[1, 2], ...}, ...]"
- The second array on the third level contains four elements.
  "[..., [..., [1, 2, 3, 4], ...]]"
- The first array on the fourth level contains one element.
  "[{..., ..., {..., ...:[1]}}, ...]"
- The second array on the fourth level contains three elements.
  "[..., [..., ..., {..., ...:[1, 2, 3]}]]"

The same can be done for objects based on their position. This method may work, if every input has the exact same structure. If for example the first array on the third level is replaced with a string, there is no way for the statistic/parser to know that the now first array on the third level will have four instead of two elements. This method is not suitable to solve the problem.
What can be used instead are the key-value-pairs of the objects. If the input structure is nearly the same every time and the same keys are used to describe an object, information mapped to the keys can be used, even if an array or object is replaced by another value.

- "key2": Array contains two elements
- "key3": Object contains two member
- "number": Array contains either one or three elements
  (average element count 2)

This method can not cover the whole input, but it is reliable even if the input varies. It is only necessary, that the keys keep their meaning.
What needs to be stored:

- The key
- The total amount of elements of an array/object assigned to this key
- The sample size

For the key "number" in Listing 4.18 would be stored:

- Key: "number"
- elem_count: 4 (1 element + 3 elements)
- sample_size: 2

If the next input contains the key ”number” and the value is an array or object, the count is increased by the amount of elements the new array/object contains and the sample size is increased by one.

Before implementing the statistic, there is one more option for this method to consider. It is possible to not only map the key, but to also map the type (array/object), which would be more precise, if those two types occur equally often and the amount of elements is very different. In case of the MAG files it is not useful to differentiate between both types, so the type will not be stored.

Implement the statistic

First step is to implement a basic structure to store all information mapped to the corresponding keys (key-value-pairs). All information have to be quickly accessible, so to store the key-value-pairs in a vector or list may not be the best option. A vector or list would require to check every element until the key matches. A better way to store the information is to use a hash table. It allows to quickly find the required information. How exactly the hash table works in the case of the Karbonit JSON parser is now explained step by step.

To manage the hash table the struct parseStats (Listing 4.19) is created.

```
1 typedef struct parseStats {
2     statsElement* arr[50];
3     size_t size;
4     size_t count;
5 } parseStats;
```

Listing 4.19: Struct parseStats

It stores an array of pointer to statsElements that is used as the hash table, the current size of the array and how many key-value-pairs are already stored.

```
1 typedef struct statsElement {
2     const char* key;
3     size_t count;
4     size_t sample_size;
5     size_t max_count;
6 } statsElement;
```

Listing 4.20: Struct statsElement

A statsElement (Listing 4.20) stores information for a single key. It stores the key, the total count of elements and the sample size, like described earlier. Additionally it stores the maximum amount of elements a single array or object associated with the stored key had. The max_count can be used to weight the prediction if needed. Now that the basic structs are implemented, the hash table has to be initialized and functions to add or get information have to be created.

```
1  static void init_parseStats(parseStats* stats)
2  {
3      stats->size = 50;
4      stats->count = 0;
5
6      size_t i = 0;
7      while (i < stats->size)
8      {
9          stats->arr[i] = NULL;
10         i++;
11     }
12 }
```

Listing 4.21: Function init_parseStats

The function init_parseStats (Listing 4.21) initializes a instance of the parseStats struct. The size is set to 50 in this case, because the hash table can store 50 values at the moment. The size/capacity is just a default value for now. Although default values primarily caused the problem, an instance of parseStats with a capacity of 50 just uses 416 Byte of space, which is reasonable. There is also no need for multiple instances of parseStats, so the 416 Bytes are fixed.

```
1  static void insert_statsElement(parseStats* stats, char* key,
       size_t count)
2  {
3      ...
4
5      size_t key_hash = HASH_ADDITIVE(strlen(key), key);
6      size_t pos = key_hash % stats->size;
7
8      while (stats->arr[pos] != NULL)
9      {
10         pos++;
11         pos %= stats->size;
12     }
13
14     statsElement* item = malloc(sizeof(statsElement));
15     item->key = key;
16     item->count = count;
17     item->sample_size = 1;
18     item->max_count = count;
19
20     stats->arr[pos] = item;
21     stats->count++;
22 }
```

Listing 4.22: Function insert_statsElement

The function insert_statsElement (Listing 4.22) inserts a new key-value-pair into the hash table. To get the position in the hash table, the key is hashed with HASH_ADDITIVE and the result of $hashvalue\%thesize$ is calculated. Now the next free position is searched starting from the calculated position. If the position is determined, a statsElement is instantiated and stored.

```
1  static statsElement* get_statsElement(parseStats* stats, char* key)
2  {
3      size_t key_hash = HASH_ADDITIVE(strlen(key), key);
4      size_t pos = key_hash % stats->size;
5
6      if(stats->count != stats->size)
7      {
8          while((stats->arr[pos] != NULL) && (strcmp((stats->arr[pos
               ])->key, key) != 0))
9          {
10             pos++;
11             pos %= stats->size;
12         }
13     }
14     else
15     {
16         size_t orig_pos = pos;
17
18         while((stats->arr[pos] != NULL) && (strcmp((stats->arr[pos
               ])->key, key) != 0))
19         {
20             pos++;
21             pos %= stats->size;
22
23             if(pos == orig_pos)
24             {
25                 return NULL;
26             }
27         }
28     }
29
30     return stats->arr[pos];
31 }
```

Listing 4.23: Function get_statsElement

The fuction get_statsElement (Listing 4.23) searches the key-value-pair with a matching key. It uses the same method to calculate the expected position as insert_statsElement and starts to check every pair from this position. To prevent endless loops the function has to check if it reached the starting position of the search, if the hash table is full (see Listing 4.23 lines 14-28).

```
1  static void update_statsElement(parseStats* stats, char* key,
       statsElement* elem, size_t count)
2  {
3      if(elem == NULL)
4      {
5          elem = get_statsElement(stats, key);
6      }
7
8      elem->count += count;
9      elem->sample_size++;
10
11     if(elem->max_count < count)
12     {
```

```
13          elem->max_count = count;
14      }
15  }
```

Listing 4.24: Function update_statsElement

The function update_statsElement (Listing 4.24) either updates a specified key-value-pair if the parameter elem is set or it searches the element with a matching key to increase the count and sample size.

```
1  static void update_or_insert_statsElement(parseStats* stats, char*
       key, size_t count)
2  {
3      statsElement* elem = get_statsElement(stats, key);
4
5      if(elem == NULL)
6      {
7          insert_statsElement(stats, key, count);
8      }
9      else
10     {
11         update_statsElement(stats, key, elem, count);
12     }
13 }
```

Listing 4.25: Function update_or_insert_statsElement

update_or_insert_statsElement (Listing 4.25) is the last function that manipulates the hash table. It tries to get an element with an matching key. If an element is found, the element is updated otherwise a new key-value-pair is inserted into the table.
Information can be inserted, updated and searched for, the next step is to get a prediction based on these information.

```
1  static size_t stats_get_prediction(parseStats* stats, char* key)
2  {
3      statsElement* elem = get_statsElement(stats, key);
4
5      if(elem == NULL)
6      {
7          return 20;
8      }
9
10     return ((elem->count/elem->sample_size) + elem->max_count)/2;
11 }
```

Listing 4.26: Function stats_get_prediction

A prediction is calculated by the function stats_get_prediction (Listing 4.26). It tries to get the element with a matching key. If no element is found, it simply return the default value 20, otherwise it returns a calculated value based on the amount of elements, the sample size and/or the maximum amount of elements.

Line 10 in Listing 4.26 can be altered to use different calculations.
Some examples:

- A maximum weighted average
  $((elem- > count/elem- > sample\_size) + elem- > max\_count)/2$
- The average amount
  $elem- > count/elem- > sample\_size$
- Or just the maximum amount
  $elem- > max\_count$

Although different calculations are possible, currently it is hard-coded, so the user can not influence it.

Now the hash table can be filled with information to use it. All information about keys and element counts have to be retrieved from the input. This is done by calling the function build_parseStats.

```
1  static void build_parseStats(parseStats* stats, const char* input,
       size_t input_size)
2  {
3      bool escaped = false;
4      bool in_string = false;
5
6      size_t i = 0;
7      size_t count = 0;
8
9      char* key;
10
11     char c;
12
13     while(i < input_size)
14     {
15         c = input[i];
16         i++;
17
18
19         if(isspace(c))
20         {
21             continue;
22         }
23
24         //toggle in string and out of string, if not escaped
25         if((c == '"') && (!escaped))
26         {
27             in_string = !in_string;
28         }
29
30         if((c == ':') && (!in_string) && (!escaped))
31         {
32             key = build_parseStats_get_key(input, i-1);
33             if(key != NULL)
```

```
34              {
35                  count = build_parseStats_get_count(input, i - 1,
                        input_size);
36
37                  if (count != 0) {
38
39                      update_or_insert_statsElement(stats, key, count
                            -1);
40
41                      count = 0;
42                  }
43              }
44          }
45
46
47          if(c == '\\')
48          {
49              escaped = !escaped;
50          }
51          else
52          {
53              escaped = false;
54          }
55      }
56  }
```

Listing 4.27: Function build_parseStats

The function shown in Listing 4.27 checks the entire input for assignments (':') that indicate key-value-pairs. If a key-value-pair is found, the function build_parseStats_get_key is called, which reads the input backwards to get the key. If a key is returned the function build_parseStats_get_count reads forward and counts the elements if the value is an array or object, otherwise it returns 0. The collected information is then stored by calling update_or_insert_statsElement.

The last step is to include the statistic into the current parse procedure. The instance of parseStats has to be accessible if an object node is created, which requires some changes to structs and functions.
Following changes are made to the structs json_parser (Listing 4.28) and parser_task_args (Listing 4.29):

```
1  typedef struct json_parser {
2          ...
3          parseStats* stats;
4  } json_parser;
```

Listing 4.28: Struct json_parser with statistic

```
1  typedef struct parser_task_args{
2          ...
3          parseStats* stats;
4  } parser_task_args;
```

Listing 4.29: Struct parser_task_args with statistic

The function json_parse_split_exp shown in Listing 4.30 now accepts an additional parseStats parameter to add the statistic to the json_parser it creates.

```
1  bool json_parse_split_exp(..., parseStats* stats)
2  {
3      ...
4      while (!end_parse)
5      {
6          if((input[i] == '\n') || (input[i] == '\0'))
7          {
8              ...
9              if(destdir == NULL)
10             {
11                 struct json data;
12                 json_err err;
13                 json_parser parser;
14                 parser.stats = stats;
15                 ...
16             }
17             else
18             {
19                 ...
20                 struct json data;
21                 json_err err;
22                 json_parser parser;
23                 parser.stats = stats;
24                 ...
25             }
26             lastPart = i+1;
27         }
28         i++;
29     }
30
31     return true;
32 }
```

Listing 4.30: Function json_parse_split_exp with statistic

Which then is used in parse_members_exp (Listing 4.31) to get the prediction.

```
1  bool parse_members_exp(..., json_parser *parser, ...)
2  {
3      ...
4
5      do {
6          ...
7          switch (valueToken.type) {
8              case OBJECT_OPEN:
9                  member->value.value.value_type = JSON_VALUE_OBJECT;
10                 member->value.value.value.object = MALLOC(sizeof(
                       json_object));
11                 pred_count = stats_get_prediction(parser->stats,
                       member->key.value);
12                 parse_object_exp(parser, member->value.value.value.
                       object, error_desc, token_mem, pred_count);
13                 break;
```

```
14              case ARRAY_OPEN:
15                  member->value.value.value_type = JSON_VALUE_ARRAY;
16                  member->value.value.value.array = MALLOC(sizeof(
                        json_array));
17                  pred_count = stats_get_prediction(parser->stats,
                        member->key.value);
18                  parse_array_exp(parser, member->value.value.value.
                        array, error_desc, token_mem, pred_count);
19                  break;
20              ...
21          }
22          ...
23      } while (...);
24
25      return true;
26  }
```

Listing 4.31: Function parse_members_exp with statistic

Only the call of build_parseStats is still missing. Because of the changes made up to now, the statistic will mainly be used to parse line by line and parallel, so there are two options:

- The statistic is build up parallel with other tasks.
- The statistic is build up before the input is parsed.

The first option is expected to be faster than the second option, because there is way less delay until the parse tasks are processed.

To use the statistic combined with parallelization a new task routine is implemented as shown in Listing 4.32.

```
1  static void task_routine_stats(void *args)
2  {
3      parser_task_args* task_args = (parser_task_args*) args;
4
5      build_parseStats(task_args->stats, task_args->start, task_args
           ->size);
6  }
```

Listing 4.32: Function task_routine_stats

Either in json_parse_split_parallel the function build parse stats is called before even the thread pool or tasks are created or a task with the routine task_routine_stats is added to the thread pool.

4.6 Parser Configuration

The user should be able to configure the parser to decide how the input will be parsed. To enable configuration most new implementations and changes are bundled into modules. This section explains what options the Karbonit JSON parser

executable accepts and what effects those options have. In this section the executable is referred to as "exec".

To use the parser only one argument is required, the path to the input file. The input file can either be a JSON or NDJSON file. An example:
./exec "/home/user/input.json"

Additionally the following options are available:

- -l
  The content will be parsed line by line. This option is required to parse a NDJSON file.
- -t
  Enables parallelization and sets the amount of threads that can be used to parse the input. This option only works, if the option -l is also set.
- -p
  Requires options -l and -t to be set. Sets the amount of parts the input is split into. If -t is set, but not -p, the amount of parts is set to the amount of threads.
- -e
  The parser will use the functions that combine tokenizer and interpreter. If this option is not set the parser will use the procedure with separate tokenizer and interpreter.

Some examples:

- Parse a JSON file.
  ./exec "/home/user/input.json"
- Parse a NDJSON file.
  ./exec -l "/home/user/input.ndjson"
- Parse a NDJSON file with 5 threads and parts.
  ./exec -l -t 5 -p 5 "/home/user/input.ndjson"

4.7 Summary

This chapter described every change to the original Karbonit JSON parser. It is now not only possible to parse JSON files, it is also possible to parse NDJSON files. Depending on the parser configuration that is chosen the parser can use parallelization or statistics to parse more efficient. The exact results will be evaluated in the following Chapter 5 Evaluation.

5. Evaluation

In this chapter all implementations are evaluated based on the resulting parsing times and the throughput when parsing a MAG file. To evaluate all results properly, a method is defined to measure the parsing time and reference data was generated. The results are evaluated in Section 5.3.

5.1 Methods of Measurement

To get meaningful measurements it is important that every measurement follows the same procedure. This is guaranteed by using a python script (see Listing 5.1) that performs the following steps:

- Loop 5 times
  - get current timestamp (start of measurement)
  - run the parser with a certain configuration
  - get current timestamp (end of measurement)
  - calculate difference between start and end and save the result in a csv file
- calculate the average time needed and display the result

```
1  import subprocess
2  import os
3  import time
4
5  #set path to input, executable and output
6
7  inputPath = "[path]/[filename]"
8
9  execPath = "[path]/[executable]"
10
11 resultPath = "[path]/[filename]"
12
```

```
13 #open file to store the results
14 resultFile = open(resultPath, "w")
15
16 i = 0
17 n = 5
18 t = 0
19
20 while i < n:
21         start = time.time()
22         process = subprocess.run([execPath, "-l", inputPath],
               stdout = subprocess.DEVNULL)
23         if process.returncode != 0:
24                 print("Error")
25         end = time.time()
26         diff = end - start
27         t += diff
28         resLine = str(i) + ";" + str(diff) + "\n"
29         resultFile.write(resLine)
30         i += 1
31
32 resultFile.close()
33
34 t /= n
35 print("Avg:", t)
```

Listing 5.1: Python script template

The script shown in Listing 5.1 is a template, that is edited depending on the parser configuration. To use the script the placeholder in line 7, 9 and 11 have to be replaced. The variable inputPath contains the input file path, execPath contains the executable path and resultPath contains the path to the output file, where all measurements are stored. Usually the output file is a structured file. In our case it is a CSV file.
The subprocess.run function call in line 22 starts the parse of the input file. The first parameter of this function is a list, that specifies the executable path at the first position and every other entry is an option or argument that is handed over to the executable. In this case subprocess.run runs the following command: ”executable -l input_file”, which causes the parser to parse the input file line by line. To use other parser configurations the line 22 has to be edited.

A MAG file is used as the input. Normally the chosen MAG file contains 1000000 lines and has a file size of 1836 MB. To evaluate if the input size affects the time to parse, three other files with different sizes are generated based on this MAG file. The first generated file contains 250.000 lines with a total size of 459 MB, the second one contains 500.000 lines with a size of 918 MB and the third contains 750.000 lines and has a size of 1377 MB. Due to the uniform structure of all MAG files and that all MAG files contain 1000000 lines with a file size of nearly 2 GB, the chosen file also represents all other MAG files. To use another MAG file will return similar results.

All deviations from this procedure are documented in the following sections.

5.2 Performance Baseline

The performance of the original Karbonit JSON parser is used as a reference to assess the impact of all implementations made in this thesis. To get reference values a MAG file has to be parsed with the original parser, which requires some adjustments to the python script shown in Listing 5.1.

The python script used to get reference values is shown in Listing 5.2:

```
1  import subprocess
2  import os
3  import time
4
5  #set path to input, executable and output
6
7  inputPath = "[path]/[filename]"
8
9  execPath = "[path]/[executable]"
10
11 resultPath = "[path]/[filename]"
12
13 temppath = "[path]/[filename]"
14
15 #open file to store the results
16 resultFile = open(resultPath, "w")
17
18 i = 0
19 n = 5
20 t = 0
21
22
23 while i < n:
24         errorcount = 0
25         inputFile = open(inputPath, "r")
26         start = time.time()
27         for line in inputFile:
28                 temp = open(temppath, "w")
29                 temp.write(line)
30                 temp.close()
31
32                 process = subprocess.run([execPath, temppath],
                       stdout = subprocess.DEVNULL)
33                 if process.returncode != 0:
34                         errorcount += 1
35         end = time.time()
36         diff = end - start
37         t += diff
38         resLine = str(i) + ";" + str(diff) + "\n"
39         resultFile.write(resLine)
40         print("Fehleranzahl:", errorcount)
41         inputFile.close()
42         i += 1
43
44
45
46 resultFile.close()
```

```
47 os.remove(temppath)
48
49 t /= n
50 print("Avg:", t)
```

Listing 5.2: Python script to get reference values

The only change made to python script shown in Listing 5.1 was to replace the subprocess.run call in lines 22-24 by a loop that does the following for every line(see Listing 5.2):

- open a temporary file (line 28)
- write current line into file (line 29)
- close the file (line 30)
- parse the temporary file (lines 32-34)

It is necessary to split the NDJSON input file in separate lines, because the original JSON parser is unable to parse the NDJSON file as a whole. What is not apparent from the Listing 5.2 is that the variable execPath contains the path to an executable build on the source code from the revision with which this project was initialized. Implementations made in this project do not affect the reference values.

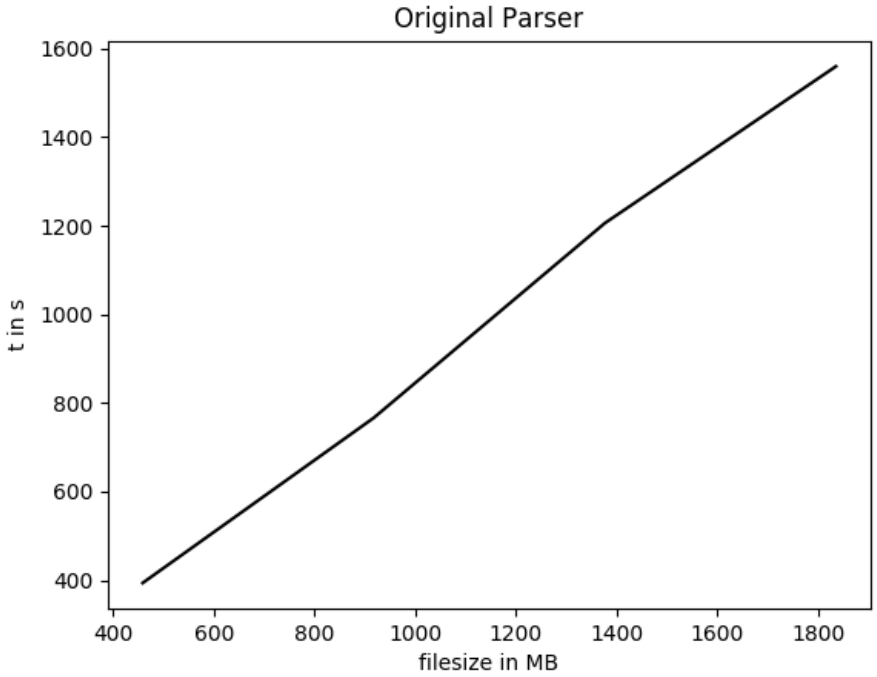


Figure 5.1: A chart displaying the parse results of the original Karbonit JSON parser

Figure 5.1 shows the time needed to parse depending of the file size of a MAG file. The file size in MB is plotted on the x-axis and the time for parsing the input in

seconds on the y-axis. Four data points are used to create the graph. Coordinates of the four other data points (x - filesize, y - avg time):

- (459, 393)
- (918, 766)
- (1377, 1206)
- (1836, 1559)

The graph formed by those data points is nearly linear, so time to parse is proportional to the file size. The average throughput[1] based on those data points is 0.009 GBit/s.
The python script shown in Listing 5.2 also counts how often the parse command failed. The command failed two times for each of the input files with a file size of 459 MB, 918 MB and 1377 MB. The input with a file size of 1836 MB caused three parse commands to fail. A failed parse command means that the current line could not be parsed. In case of the 459 MB file with approximately 250000 lines only 0.0008% of the lines could not be parsed. This percentage is even lower for the 918 MB file and 1377 MB file. Both files contain more lines than the 459 MB file, but the amount of errors stayed the same. The percentage of lines that could not be parsed for the 918 MB file is 0.0004% and for the 1377 MB file it is 0.00026%. In case of the 1836 MB file with 1000000 lines and 3 errors the percentage is 0.0003%. These errors lead to a somewhat inaccurate parse result because information has been lost. However, the impact on the measured parsing times is small enough that the measurement is still valid.
The parse failed because of escaped quotes, that the parse mistook for unescaped. More about this problem is described in Section 4.4.

5.3 Results

Now that all implementations were made, the effects of every single change has to be evaluated. The following sections describe what exactly has to be evaluated, how the measurements were done, if they differ from the standard definded in Section 5.1 and how the results have to be interpreted. Last part of every section is to state which functional and non-functional requirements are fulfilled.

5.3.1 Check for escaped Quotes

This section is based on the implementations of Section 4.4 Check for escaped Quotes that fixed a problem in the function parse_string_token. The new implementation of this function is shown in Listing 4.13 and is a fixed component of the parser, regardless of the configuration used.
To evaluate how the new implementation of parse_string_token affects the parser performance, the original JSON parser is used. The old implementation of parse_string_token is replaced by the new one. Otherwise changes made in other sections of this thesis

[1] throughput in GBit/s= (file size in MB / parsing time in s) * 0.008

would affect the results, for example the changes made to the function json_tokenizer_next in Section 4.1 Parse Line by Line.

The python script (Listing 5.2) used in Section 5.2 is used again to measure the parsing time.

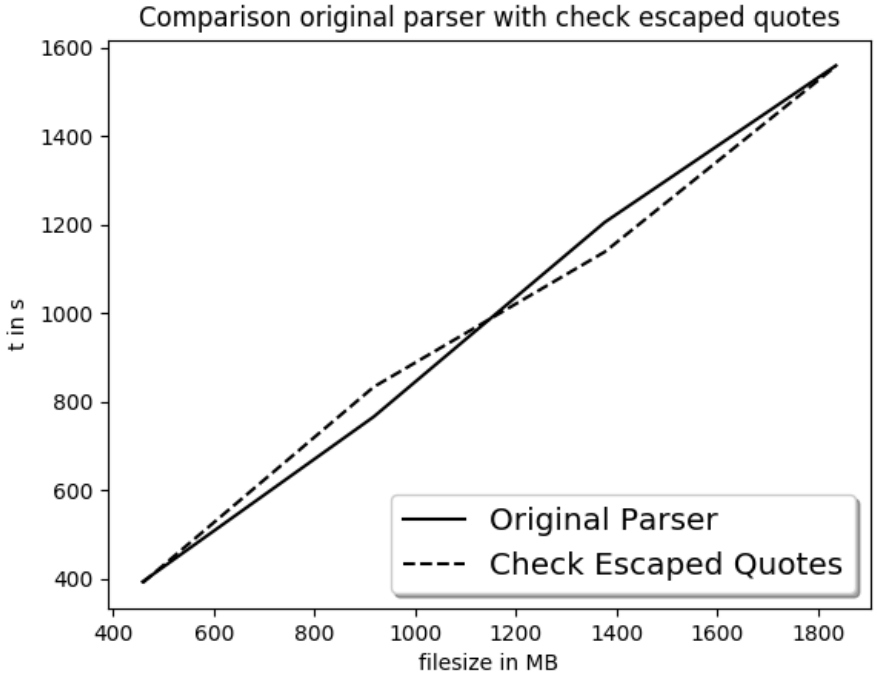


Figure 5.2: A chart displaying the parse results of the original Karbonit JSON parser with the new implementation of parse_string_token

Figure 5.2 extends the chart in Figure 5.1 on page 58 and adds a second graph "Check Escaped Quotes". The new graph represents the results of the parser with the new implementation of parse_string_token. The following data points form the graph:

- (459, 391)
- (918, 833)
- (1377, 1139)
- (1836, 1558)

Like the first graph "Original Parser" the new graph also is nearly linear. In general both versions of the Karbonit JSON parser will need approximately the same time to parse an input. The throughput in case of the graph "Check Escaped Quote" is also 0.009 GBit/s.
Both versions need the same time to parse, but the input can be parsed without any errors with the new implementation of parse_string_token. No failed parse commands

were recognized.
Based on the current evaluation the implementation made in Section 4.4 does not fulfill any functional or non-functional requirement, it only fixes a problem.

5.3.2 Parse Line by Line

This section is based on Section 4.1. The results of parsing NDJSON files line by line, like implemented in the current Karbonit JSON parser, will be compared to the results shown in Section 5.2 and Section 5.3.1. To measure the parsing time the python script template from Listing 5.1 is used. The placeholder for input, executable and output paths are replaced and the parser configuration with option "-l" enabled causes the parser to parse the input line by line.

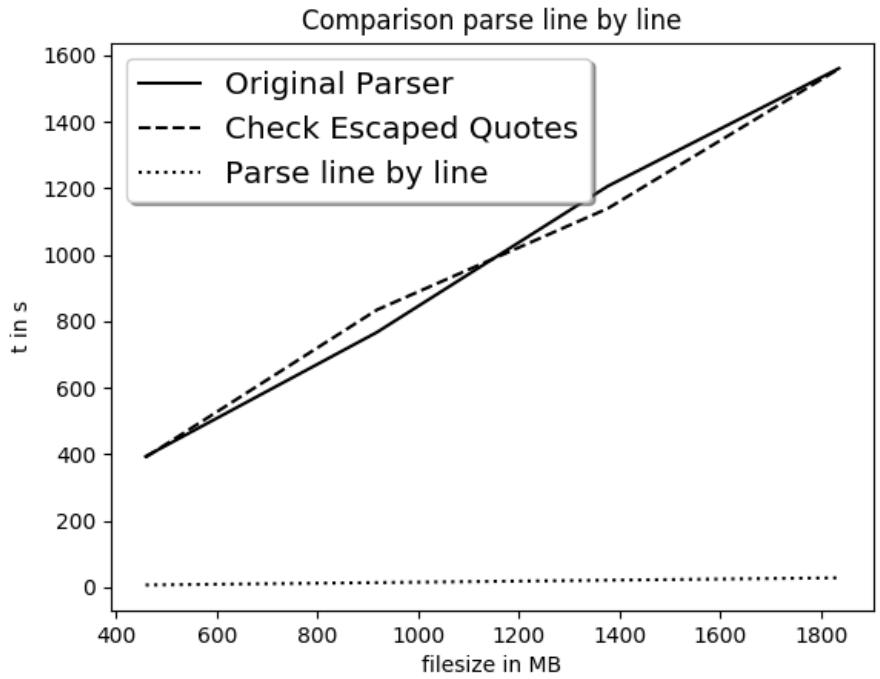


Figure 5.3: A chart to compare parsing line by line with the the previous approach to parse with the original parser

Figure 5.3 again shows the parsing time in relation to the input file size of an MAG file. The parsing time needed increases slower depending on the file size in the graph "Parse line by line", than in "Original Parser" or "Check Escaped Quotes". To inspect the graph "Parse line by line" more detailed, it is shown separately in Figure 5.4 on the following page.

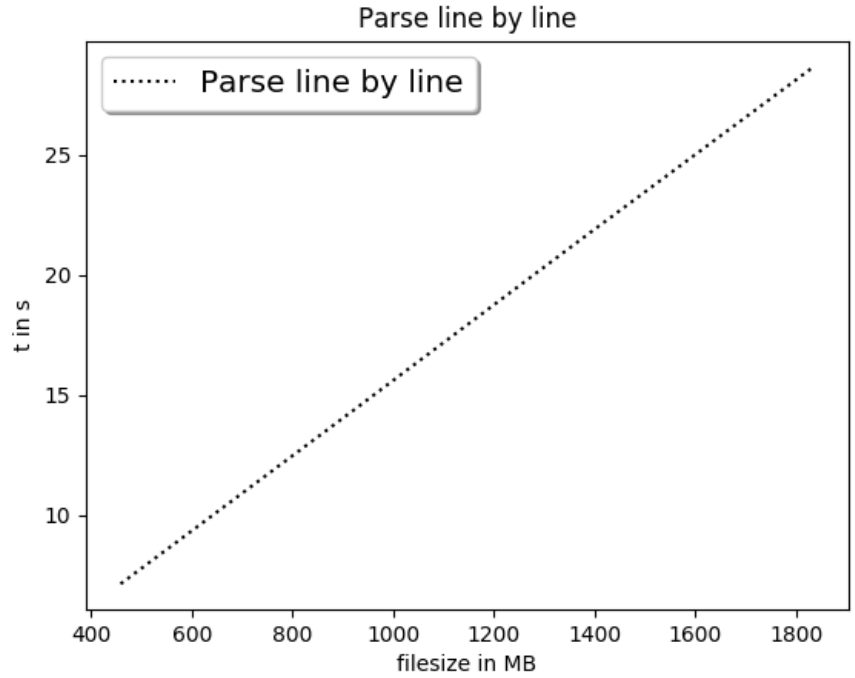


Figure 5.4: A chart showing only the results of parsing line by line

The data points that form the graph ”Parse line by line”:

- (459, 7)
- (918, 14)
- (1377, 21)
- (1836, 28)

The average throughput based on those data points is 0.525 GBit/s.

The big difference between the reference values and the result of parsing line by line directly, is that generating files, writing content and closing the files is extremely expensive compared to the way the current parser parses line by line. The original parser was unable to accept specific start and end positions that is why the NDJSON file had to be split into single lines before the parser could take action.

The implementations made in Section 4.1 fulfill the functional requirement to enable parsing of NDJSON files and it also fulfills the non-functional performance requirement to increase the throughput of the Karbonit JSON parser. The original parser had a throughput of 0.009 GBit/s which is significantly lower than the 0.525 GBit/s reached by ”Parse line by line”.

5.3.3 Multiple Threads

This section is based on the implementations made in Section 4.2, which enables the parser to parse multiple lines in parallel. To evaluate the effect of parsing with multiple threads, the method to measure parsing times described in Section 5.1 is adjusted slightly. Instead of parsing four different MAG files only the largest one with 1000000 lines will be parsed. Also the python script from Listing 5.1 is changed to parse the file repeatedly, but to increase the amount of threads and parts used step by step.

```
1  import subprocess
2  import os
3  import time
4
5  #set path to input, executable and output
6
7  inputPath = "[path]/[file name]"
8
9  execPath = "[path]/[executable]"
10
11 resultPath = "[path]/[file name]"
12
13 resultFile = open(resultPath, "w")
14 resultFile.write("threads;parts;i;time\n")
15
16 n = 5
17
18
19 threads = 2
20 parts = 2
21 max_threads = 24
22
23 while threads <= max_threads:
24         parts = threads
25         while parts < (threads + 4):
26                 t = 0
27                 i = 0
28                 while i < n:
29                         start = time.time()
30                         process = subprocess.run([execPath, "-l", "
                               -p", str(parts), "-t", str(threads),
                               inputPath], stdout = subprocess.DEVNULL)
31                         if process.returncode != 0:
32                                 print("Error")
33                         end = time.time()
34                         diff = end - start
35                         t += diff
36                         resLine = str(threads) + ";" + str(parts) +
                                ";" + str(i) + ";" + str(diff) + "\n"
37                         resultFile.write(resLine)
38                         i += 1
39                 t /= n
40                 print("Threads:", threads, "Parts:", parts, "
                       Durchschnittlich:", t)
41                 parts += 1
42         threads += 1
43
```

```
44
45 resultFile.close()
```

Listing 5.3: Python script to measure parsing times depending on the amount of threads and parts

The python script shown in Listing 5.3 contains two additional loops. The outer loop (line 23) increases the amount of threads by one every step and the loop in line 25 increases the amount of parts. When the amount of threads is increased, the amount of parts is set the the same amount. The amount of parts in the loop in line 25 will not exceed the current amount of threads increased by four. Also the maximum amount of threads will not exceed 24.
The maximum amount of threads is chosen because of the hardware used for measurements. A CPU with four cores and a total of eight threads is installed. The maximum amount of threads to measure is three times as high as the amount of CPU threads. However this limit is chosen and not absolute.
Why the maximum amount of parts is set to the amount of threads + 4 will be explained after showing the effect of parallel parsing.

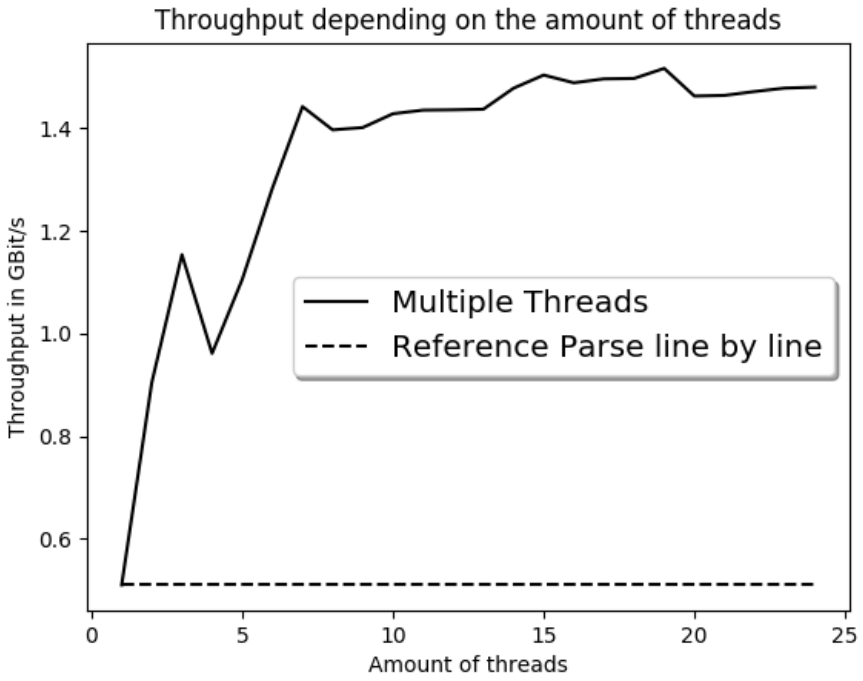


Figure 5.5: A chart showing the effect of parsing parallel

Unlike the previous charts, the chart in Figure 5.5 shows the throughput in GBit/s depending on the amount of threads used. The amount of parts equals the amount of threads for every data point of the graph "Multiple Threads". The first data point (1, 0.52) represents the results discussed in Section 5.3.2. All further points are measured by the python script from Listing 5.3. The graph "Reference Parse line by line" is used as a reference to show the difference in the throughput for every amount of threads compared to the throughput reached by just parsing line by line.

The graph "Multiple Threads" shown in Figure 5.5 on the facing page is similar to the graph of the function $f(x) = \sqrt{x}$. The lowest throughput is at the point (1, 0.52), where no parallelization was used. The highest throughput was reached with 15 to 19 threads, with approximately 1.51 GBit/s. 1.53 GBit/s were reached with 18 threads, which is the maximum of the measured values.

The amount of parts was fixed for the chart in Figure 5.5 on the preceding page. How the amount of parts in relation to the amount of threads affects the throughput is shown in the next chart in Figure 5.6.

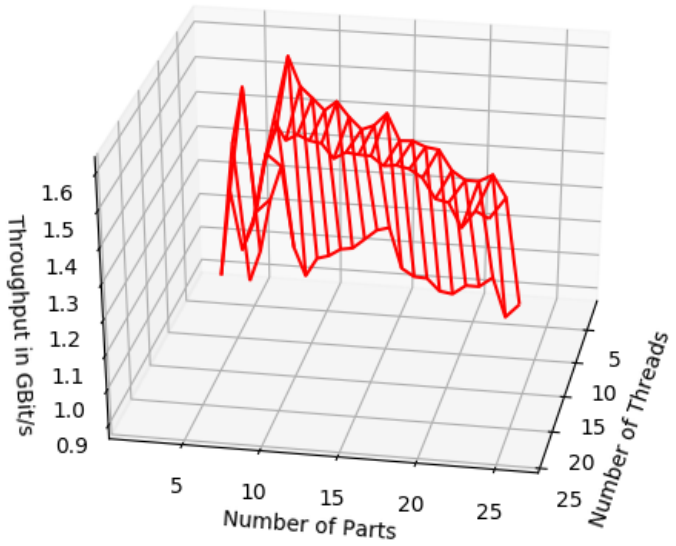


Figure 5.6: A chart showing how the amount of parts affects the throughput

Figure 5.6 shows a 3d plot (x - Amount of threads, y - Amount of parts, z - Throughput in GBit/s) that contains three data points per thread count. For example it contains (2,2,z), (2,3,z) and (2,4,z). Every 3-tupel is represented by a V-shape in this 3d plot. Those V-shapes are connected by three long lines that connect the data points like the following:

(2,2,z)—(3,3,z)—(4,4,z)—...

(2,3,z)—(3,4,z)—(4,5,z)—...

(2,4,z)—(3,5,z)—(4,6,z)—...

The line in the middle seems to have higher z-values than the other two lines. This the line in which the amount of parts equals the amount of threads + 1.

The plot in Figure 5.7 on the following page shows the part of the plot in Figure 5.6, where the amount of threads is 16. As previously assumed, the data point with 17

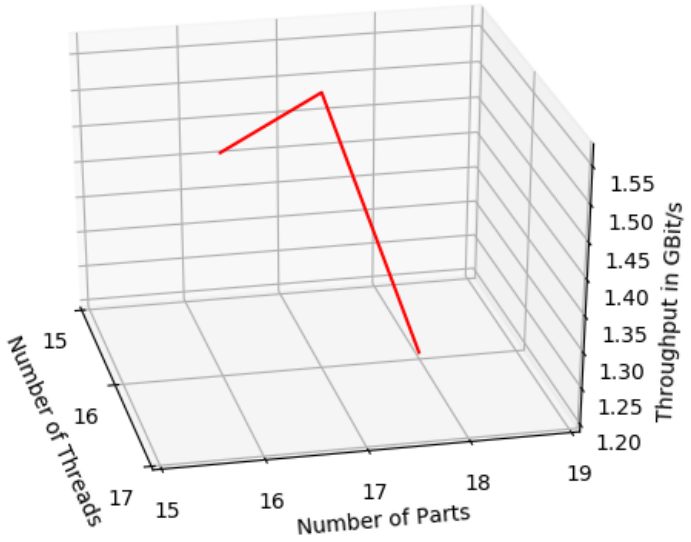


Figure 5.7: A chart showing how the amount of parts affects the throughput

parts has the highest throughput of all datapoints with 16 threads.
The highest throughput of 1.627 GBit/s reached the configuration with 7 threads and 8 parts.

Parallel parsing speeds up the entire parsing process. By parsing line by line a throughput of 0.525 GBit/s was reached, with 1.627 GBit/s by using parallelization the throughput is three times as high. So the non-functional performance requirement of increasing the throughput was met.

5.3.4 Get Tokens and Interpret Them Immediately

This section is based on the implementations made in Section 4.3 to combine the tokenizer and the interpreter. Those implementations aim to increase the throughput by leaving out intermediate steps. An effect should be seen both in line by line parsing, as well when parallel parsing.

The chart in Figure 5.8 on the facing page shows the parsing time in seconds depending on the input size in MB. The ”Parse line by line” graph introduced in Section 5.3.2 is used as a reference to compare with the results achieved by combining the tokenizer and interpreter. In case of this chart, the lower the y-value, the better.
Every data point of the graph ”Combined tokenizer and interpreter” has a lower parsing time than the corresponding datapoint in the graph ”Parse line by line”. This is also reflected in the throughput. ”Parse line by line” has a throughput of 0.525 GBit/s, while ”Combined tokenizer and interpreter” has a throughput of 0.65.

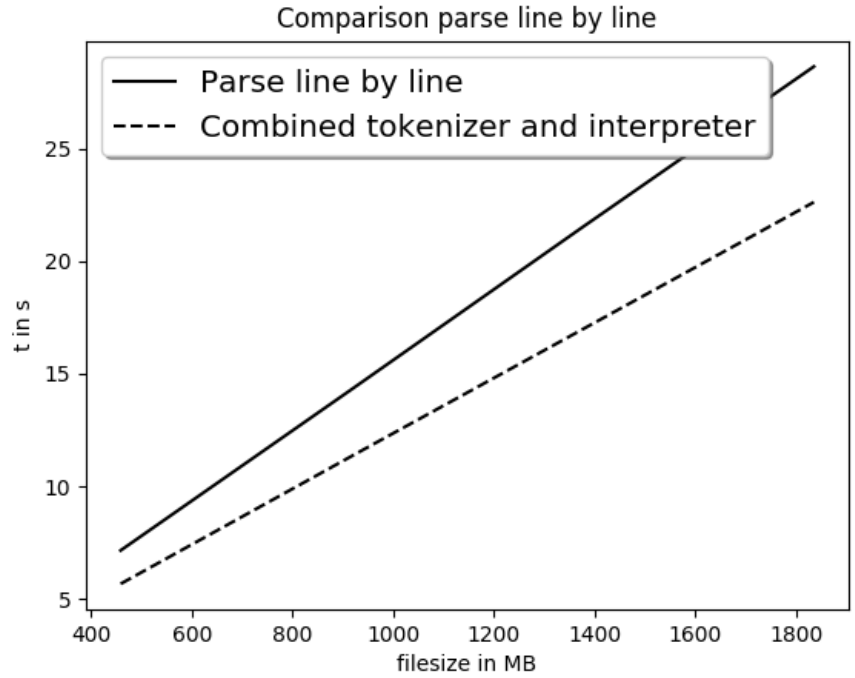


Figure 5.8: A chart showing the effect of combining tokenizer and interpreter when parsing line by line

The same effect should be visible when comparing the results of Section 5.3.3 with the results of parallel parsing with combined tokenizer and interpreter.

As expected the chart in Figure 5.9 on the next page shows, that the throughput (y-value) of ”Combined tokenizer and interpreter” is higher than the throughput of ”Parallel parsing”, no matter how many threads are used (x-value - Amount of threads). The highest throughput achieved is now 1.826 GBit/s at 19 threads and parts.

By combining the tokenizer and interpreter the throughput was increased from 1.627 GBit/s to 1.826 GBit/s, which fulfills the non-functional performance requirement of increasing the throughput.

5.3.5 Build up Statistsics

This section is based on the implementations made in Section 4.5, to decrease the memory consumption while parsing. Because of the current implementation, the statistic can only be used when parsing parallel. As mentioned in Section 4.5 there are two possible ways to build up the statistic. The first one is to build up the statistic before creating the thread pool and the tasks. The second one is to create a task that builds up the statistic and runs parallel to the parser tasks. Both options will be evaluated in this section.

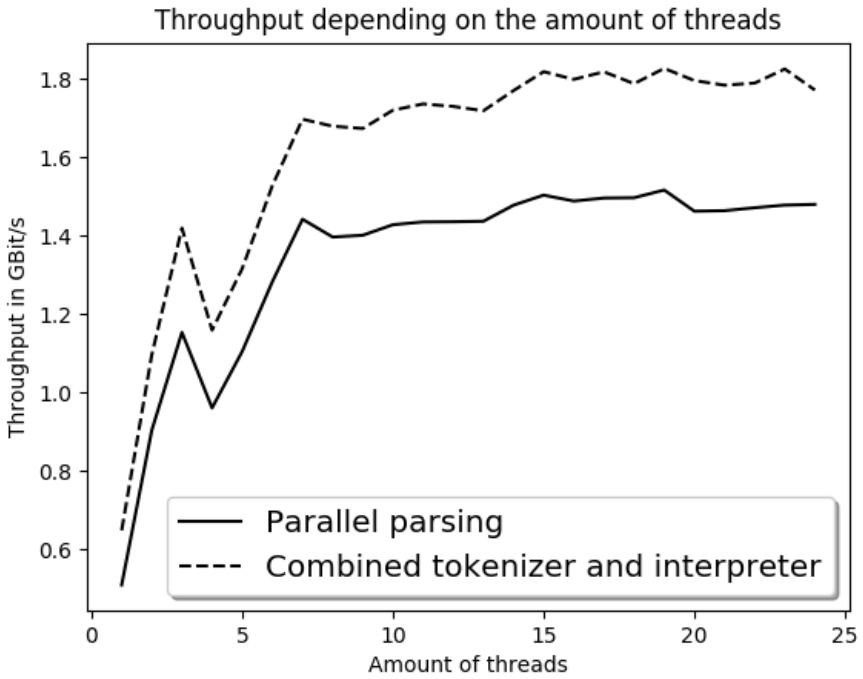


Figure 5.9: A chart showing the effect of combining tokenizer and interpreter when parsing parallel

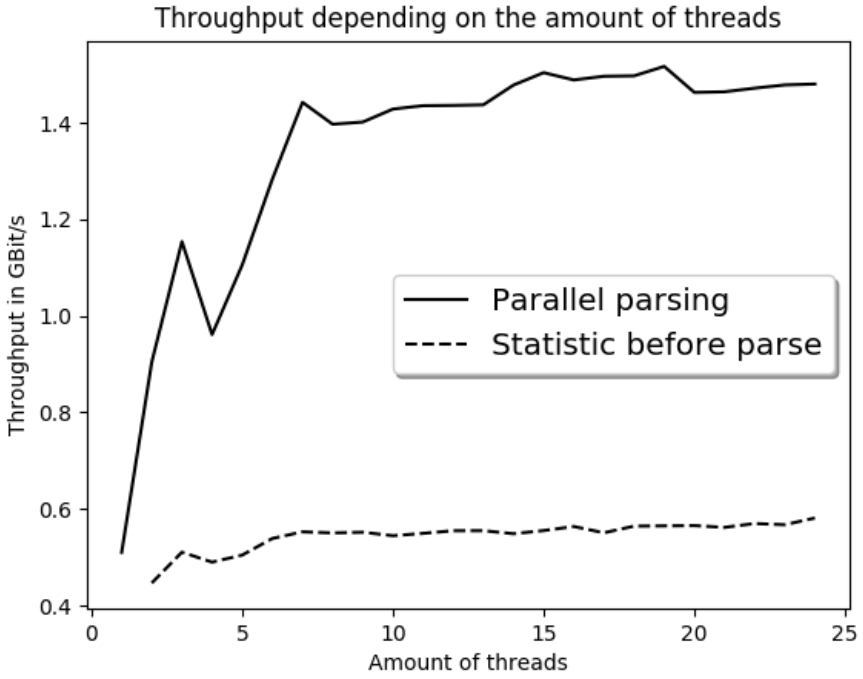


Figure 5.10: A chart showing the effect of building the statistic before starting to parse

Build statistic and parse afterwards

The chart in Figure 5.10 on the facing page shows the throughput depending on the amount of threads used to parse. The graph "Parallel Parsing" (Figure 5.5 on page 64) is used as a reference to compare the results.
Building the statistic first before parsing the input will increase the parsing time by a constant amount, independent of the amount of threads used to parse the input. The graph "Statistic before parse" differs way to much from the reference graph. The slowdown seems to increase the more threads are used. Because the time needed to build up the statistic is constant, there have to be other dependencies that influence the parsing. The most likely cause is that the entire input has to be read to build the statistic, which then causes trouble when reading the input again.

Use task to build the statistic

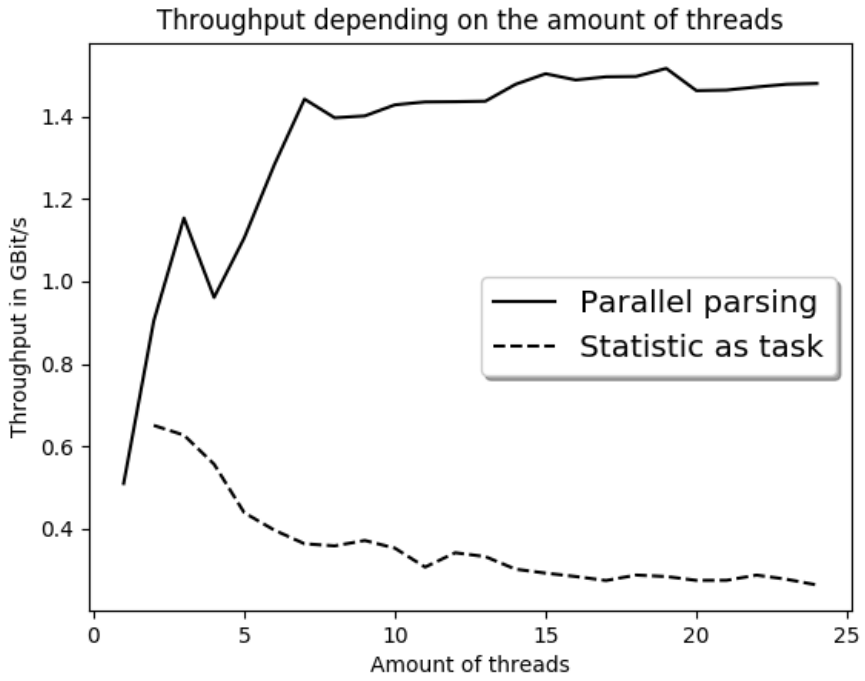


Figure 5.11: A chart showing the effect of building the statistic while parsing the input

The effect of building the statistic as a task parallel to parsing is even worse, like the graph "Statistic as task" in Figure 5.11 shows.

No matter how the statistic is build up, it affects the parallel parsing negatively. If only the parsing time is an aspect, then it would be better to discard the statistic, like it is implemented currently. But the main reason to implement the statistic was to decrease the amount of unused vector capacity.

After building the statistic, the hashtable contained the following entries:

- Key: references
  - count: 1542648
  - sample_size: 71617
  - max_count: 1198
- Key: authors
  - count: 866801
  - sample_size: 249999
  - max_count: 5758
- Key: fos
  - count: 813741
  - sample_size: 165603
  - max_count: 53
- Key: keywords
  - count: 1519416
  - sample_size: 142308
  - max_count: 759
- Key: url
  - count: 1279536
  - sample_size: 241947
  - max_count: 657

The average element count [2] can be calculated based on that entries. For ”references” the average count is 21, for ”authors” it is 3, for ”fos” 5, for ”keywords” 11 and for ”url” 5. The sample size indicates how often the input contains the key and based on the structure of MAG files each occurrence of a key means, that an array node has to be created. Like described in Section 4.5 every array node contains a vector with a capacity of 250 elements and every element has a size of 40 Byte.

As the Table 5.1 on the next page shows, the statistic prevented more than 8 GB unused space over the entire parse of the input NDJSON file. If the file would not be parsed line by line, the 8 GB space would have been allocated, until the tree structure is dropped. So the parsing time got worse, but the maximum amount of space required throughout the entire parse was decreased, which met one of the non-functional performance requirements. Also, if the statistic is used without parsing line by line the parser would be able to parse larger inputs than the original parser. A problem with the large inputs is, that the main memory has not enough capacity to store all the data generated by the parser. The 8 GB of prevented unused space could be used to store other data.

[2] average element count = count / sample_size

key	unused elements	unused allocated space in Byte	total in GB
references	229	9160	0.656
author	247	9880	2.469
fos	245	9800	1.622
keywords	239	9560	1.360
url	245	9800	2.371
			8.478

Table 5.1: Amount of unused memory prevented by the statistic

5.3.6 Parser Configuration

This section is based on Section 4.6, which simply describes the possible options and how to configure the parser. Next to the original parser, which can be used simply by running the parser executable and handing over the input, four options were introduced.

- ”-l” - Parse line by line
- ”-t” - The amount of threads to use
- ”-p” - The amount of parts to use
- ”-e” - Use the parser version with combined tokenizer and interpreter

Those options were implemented to enable the user to configure the parser, which fulfills the second functional requirement ”Enable Parser Configuration” (see Section 3.3.2).

5.3.7 Summary

This chapter defined a uniform way to measure all results of implementations made in the previous chapters. To evaluate if an improvement was achieved, the results of each implementation were compared with reference measurements of the original Karbonit JSON parser (Section 5.2 Performance Baseline) or with results of preceding implementations. Some implementations like Section 4.4 Check for escaped Quotes do not serve the purpose to fulfill the requirements directly, while others like Section 4.1 Parse Line by Line and Section 4.2 Multiple Threads improved the achievable throughput step by step depending on the chosen parser configuration.

6. Related Work

This chapter mentions some related work based on parallel JSON and NDJSON parsing.

Probably one of the fastest state of the art C++ JSON parser is the simdjson parser [sim], which can reach a throughput of up to 2GB and even more depending on the input. Like the name simdjson implicates, this parser uses SIMD ("Single instruction, multiple data") instructions to process more data simultaneously.
The simdjson parser is subject of the paper "Parsing gigabytes of JSON per second" [LL19], which describes the method the parser uses to parse JSON input. One of the recently implemented features also is to parse NDJSON files and related formats multithreaded.

Another C++ JSON parser is Mison. This parser also uses SIMD instructions to process more data with a single instruction. Mison maps the positions of all structural elements of the input into bit vectors, which then are used to find values in the JSON input. So the main method is to only parse the parts needed to find a search result in the input. Nevertheless, Mison is also able to parse the entire input at once. How exactly Mison works is described in the paper "Mison: A Fast JSON Parser for Data Analytics" [LKC+17]

7. Conclusion and Future Work

To complete this thesis, this chapter summarizes what has been achieved in this work, which requirements are met by which approaches and mentions some possible future work.

Part of this project was to analyze the original Karbonit JSON parser and to find approaches to enable NDJSON parsing, parallelization and to improve the performance of the parser in general. The steps taken to achieve this were to parse the input line by line Section 4.1, to parallelize this procedure Section 4.2, to skip intermediate steps Section 4.3 and to build up a statistic Section 4.5. In addition to this an error has been fixed Section 4.4 and the parser can now be configured by the user Section 4.6.
What exactly has to be achieved, was defined by the requirements in Chapter 3 and in Chapter 5 was discussed which of the implementations made in Chapter 4 met which requirements. Every requirement was met at least by one of those implementations.

Section	Parse NDJSON files	Parser Configuration
Section 4.1	X	
Section 4.2		
Section 4.3		
Section 4.4		
Section 4.5		
Section 4.6		X

Table 7.1: A table that shows which functional requirements were fulfilled in which sections

As shown in Table 7.1 the functional requirement to enable the parser to parse NDJSON files was fulfilled by Section 4.1 Parse Line by Line and Section 4.6 Parser Configuration fulfilled the second functional requirement to enable the user to configure the parser.

Section	Memory consumption	Increased throughput	Parse larger amounts
Section 4.1		X	
Section 4.2		X	
Section 4.3		X	
Section 4.4			
Section 4.5	X		X
Section 4.6			

Table 7.2: A table that shows which non-functional requirements were fulfilled in which sections

The Table 7.2 shows that the non-functional performance requirement to improve the throughput of the parser was fulfilled by Section 4.1, Section 4.2 and Section 4.3, while the last two non-functional requirements to lower the memory consumption while parsing and to parse larger input were met by Section 4.5.
However, not all results were positive. Although the statistic helps to decrease the memory consumption which leads to be able to parse larger files, to build the statistic will slow down the parser a lot. Depending on the method to build up the statistic, it is possible to slow down the parser increasingly the more threads are used. The parsing is still faster than with the original parser, but the slowdown is not desirable.

Still we argue that most of our reasonably chosen optimizations led to a significant improvement of the parser.

Future Work

Besides the approaches discussed in this thesis, there are many more possible ways to improve the parser.

One option to make the parser even more efficient is to make the tree structure less expensive in general. For example by removing creating more specific node structs, that only store information required for the node type.
Another option to lower the memory consumption is to count the elements of an array or object based on the tokens that are stored in a vector, if tokenizer and interpreter are not combined. The result of the tokenizer in this case is a vector that contains all tokens the input JSON contains. This vector can be used to count the elements without reading the input a second time. Only nested arrays and objects have to be considered. There would be no unused capacity, no reallocation would be needed and the memory consumption would be as minimal as possible.

A. Appendix

This chapter contains all measured data used to evaluate the results in Chapter 5.

A.1 Performance Baseline

459 MB file

i;time
0;391.07500314712524
1;394.2227520942688
2;393.7881455421448
3;394.30970883369446
4;394.56094455718994

918 MB file

i;time
0;764.7094397544861
1;770.2000632286072
2;788.1516652107239
3;760.5739235877991
4;748.5503733158112

1377 MB file

i;time
0;1120.1396741867065
1;1145.3100640773773
2;1117.545338153839
3;1123.4885578155518
4;1131.4883267879486

1836 MB file

i;time
0;1555.8608973026276
1;1563.9708065986633
2;1552.5876972675323
3;1555.5466767436763
4;1557.6802314253690

A.2 Check for escaped Quotes

459 MB file

i;time
0;390.0969669818878
1;392.1064422130585
2;389.6890923846293
3;393.9038472365422
4;391.4627839023747

918 MB file

i;time
0,821.8788149356842
1,830.108934879303
2,845.7418029308319
3,837.233983056201
4,836.672048362947

1377 MB file

i;time
0;1126.6639831066132
1;1125.563288450241
2;1155.8383815288544
3;1149.1600861549377
4;1138.1024436950684

1836 MB file

i;time
0;1558.6173732280731
1;1560.6782349482934
2;1561.1829217111344
3;1557.1274389478912
4;1558.4385729802344

A.3 Parse Line by Line

459 MB file

i;time
0;7.1410722732543945
1;7.160266876220703
2;7.133079767227173
3;7.168677568435669
4;7.162585020065308

918 MB file

i;time
0;14.306886434555054
1;14.310778856277466
2;14.302895545959473
3;14.351329803466797
4;14.301592826843262

1377 MB file

i;time
0;21.471574544906616
1;21.729344844818115
2;21.450453996658325
3;21.49856472015381
4;21.48119592666626

1836 MB file

i;time
0;28.3014657497406
1;28.2936954498291
2;28.323665857315063
3;28.315242052078247
4;28.287986993789673

A.4 Multiple Threads

459 MB file

threads; parts; i; time
2; 2; 0; 3.675694227218628
2; 2; 1; 3.644070625305176
2; 2; 2; 3.745394706726074
2; 2; 3; 4.232731103897095
2; 2; 4; 4.238935232162476
3; 3; 0; 3.191840171813965

3; 3; 1; 3.1929500102996826
3; 3; 2; 3.201521396636963
3; 3; 3; 3.184314727783203
3; 3; 4; 3.1911461353302
4; 4; 0; 4.03334641456604
4; 4; 1; 3.645172595977783
4; 4; 2; 4.026708364486694
4; 4; 3; 4.0157434940338135
4; 4; 4; 3.6471352577209473
5; 5; 0; 3.3421308994293213
5; 5; 1; 3.3371198177337646
5; 5; 2; 3.3509681224823
5; 5; 3; 3.3096489906311035
5; 5; 4; 3.301035165786743
6; 6; 0; 2.911708354949951
6; 6; 1; 2.866940498352051
6; 6; 2; 2.8701303005218506
6; 6; 3; 2.875973701477051
6; 6; 4; 2.8530893325805664
7; 7; 0; 2.579263925552368
7; 7; 1; 2.5597541332244873
7; 7; 2; 2.549781084060669
7; 7; 3; 2.5564982891082764
7; 7; 4; 2.5464770793914795
8; 8; 0; 2.655320882797241
8; 8; 1; 2.7286953926086426
8; 8; 2; 2.674896478652954
8; 8; 3; 2.736262321472168
8; 8; 4; 2.748507499694824
9; 9; 0; 2.7726340293884277
9; 9; 1; 2.683147430419922
9; 9; 2; 2.682614326477051
9; 9; 3; 2.7080159187316895
9; 9; 4; 2.6502978801727295
10; 10; 0; 2.539699077606201
10; 10; 1; 2.582001209259033
10; 10; 2; 2.6376471519470215
10; 10; 3; 2.6760857105255127
10; 10; 4; 2.721660852432251
11; 11; 0; 2.5439090728759766
11; 11; 1; 2.735471248626709
11; 11; 2; 2.5239408016204834
11; 11; 3; 2.661468982696533
11; 11; 4; 2.6567180156707764
12; 12; 0; 2.6223015785217285
12; 12; 1; 2.556380271911621
12; 12; 2; 2.56278657913208

12; 12; 3; 2.580399513244629
12; 12; 4; 2.5477702617645264
13; 13; 0; 2.5333573818206787
13; 13; 1; 2.6779255867004395
13; 13; 2; 2.6917355060577393
13; 13; 3; 2.661738157272339
13; 13; 4; 2.717153549194336
14; 14; 0; 2.484339475631714
14; 14; 1; 2.5248560905456543
14; 14; 2; 2.5565664768218994
14; 14; 3; 2.537529706954956
14; 14; 4; 2.532092332839966
15; 15; 0; 2.4043021202087402
15; 15; 1; 2.4968817234039307
15; 15; 2; 2.5923855304718018
15; 15; 3; 2.6499204635620117
15; 15; 4; 2.575989007949829
16; 16; 0; 2.5534117221832275
16; 16; 1; 2.4515786170959473
16; 16; 2; 2.463137626647949
16; 16; 3; 2.4509060382843018
16; 16; 4; 2.5053656101226807
17; 17; 0; 2.516324281692505
17; 17; 1; 2.4466233253479004
17; 17; 2; 2.5430705547332764
17; 17; 3; 2.4791228771209717
17; 17; 4; 2.521158218383789
18; 18; 0; 2.491100788116455
18; 18; 1; 2.4183897972106934
18; 18; 2; 2.5308380126953125
18; 18; 3; 2.520677328109741
18; 18; 4; 2.440828800201416
19; 19; 0; 2.4450652599334717
19; 19; 1; 2.535633087158203
19; 19; 2; 2.4114620685577393
19; 19; 3; 2.4388720989227295
19; 19; 4; 2.508894443511963
20; 20; 0; 2.5389297008514404
20; 20; 1; 2.4295294284820557
20; 20; 2; 2.4422104358673096
20; 20; 3; 2.5807292461395264
20; 20; 4; 2.8323495388031006
21; 21; 0; 2.6853244304656982
21; 21; 1; 2.5299782752990723
21; 21; 2; 2.4909937381744385
21; 21; 3; 2.4534761905670166
21; 21; 4; 2.5972139835357666

22; 22; 0; 2.5940046310424805
22; 22; 1; 2.4196083545684814
22; 22; 2; 2.5404484272003174
22; 22; 3; 2.586940288543701
22; 22; 4; 2.4886221885681152
23; 23; 0; 2.8159029483795166
23; 23; 1; 2.377570867538452
23; 23; 2; 2.973078727722168
23; 23; 3; 2.4616687297821045
23; 23; 4; 2.501861810684204
24; 24; 0; 2.389784812927246
24; 24; 1; 2.4098289012908936
24; 24; 2; 2.6346547603607178
24; 24; 3; 2.9561104774475098
24; 24; 4; 2.475982427597046

918 MB file

threads; parts; i; time
2; 2; 0; 7.425968885421753
2; 2; 1; 7.982012987136841
2; 2; 2; 8.446012496948242
2; 2; 3; 8.448160648345947
2; 2; 4; 8.454441547393799
3; 3; 0; 6.391037940979004
3; 3; 1; 6.364300727844238
3; 3; 2; 6.376290798187256
3; 3; 3; 6.58435320854187
3; 3; 4; 6.346737623214722
4; 4; 0; 8.055561065673828
4; 4; 1; 8.251227378845215
4; 4; 2; 6.9848034381866455
4; 4; 3; 7.645891427993774
4; 4; 4; 8.032302379608154
5; 5; 0; 6.586134195327759
5; 5; 1; 6.667384147644043
5; 5; 2; 6.592597007751465
5; 5; 3; 6.669434547424316
5; 5; 4; 6.626957893371582
6; 6; 0; 5.74951171875
6; 6; 1; 5.843610048294067
6; 6; 2; 5.701301097869873
6; 6; 3; 5.713399171829224
6; 6; 4; 5.733128309249878
7; 7; 0; 5.111572265625
7; 7; 1; 5.025888204574585
7; 7; 2; 5.144285440444946
7; 7; 3; 5.114412307739258

7; 7; 4; 5.062015533447266
8; 8; 0; 5.399593353271484
8; 8; 1; 5.222348213195801
8; 8; 2; 5.401500463485718
8; 8; 3; 5.109241962432861
8; 8; 4; 5.058269262313843
9; 9; 0; 5.321466445922852
9; 9; 1; 5.476903915405273
9; 9; 2; 5.305761814117432
9; 9; 3; 5.262674808502197
9; 9; 4; 5.115492582321167
10; 10; 0; 5.193623065948486
10; 10; 1; 5.306705713272095
10; 10; 2; 5.088681936264038
10; 10; 3; 5.002310276031494
10; 10; 4; 5.146385669708252
11; 11; 0; 5.179155111312866
11; 11; 1; 5.189513206481934
11; 11; 2; 5.108834981918335
11; 11; 3; 4.9691832065582275
11; 11; 4; 5.283656120300293
12; 12; 0; 5.570156812667847
12; 12; 1; 5.075011968612671
12; 12; 2; 4.964383125305176
12; 12; 3; 5.1023406982421875
12; 12; 4; 4.967207431793213
13; 13; 0; 5.065645694732666
13; 13; 1; 5.135455846786499
13; 13; 2; 5.106322765350342
13; 13; 3; 4.992714881896973
13; 13; 4; 5.1063032150268555
14; 14; 0; 4.94265079498291
14; 14; 1; 4.900098562240601
14; 14; 2; 4.881211757659912
14; 14; 3; 5.015653848648071
14; 14; 4; 4.9629199504852295
15; 15; 0; 4.817902326583862
15; 15; 1; 4.792454957962036
15; 15; 2; 4.801695823669434
15; 15; 3; 4.726691484451294
15; 15; 4; 4.80176043510437
16; 16; 0; 4.897876501083374
16; 16; 1; 4.946598052978516
16; 16; 2; 4.910421848297119
16; 16; 3; 5.117289304733276
16; 16; 4; 5.229649543762207
17; 17; 0; 4.99357795715332

17; 17; 1; 4.917179584503174
17; 17; 2; 4.8100152015686035
17; 17; 3; 4.827451944351196
17; 17; 4; 5.363219499588013
18; 18; 0; 4.851883888244629
18; 18; 1; 4.862940788269043
18; 18; 2; 4.818382024765015
18; 18; 3; 5.068448305130005
18; 18; 4; 4.8424882888793945
19; 19; 0; 4.877956390380859
19; 19; 1; 4.817676544189453
19; 19; 2; 4.773924112319946
19; 19; 3; 4.806714296340942
19; 19; 4; 4.772234201431274
20; 20; 0; 4.779540777206421
20; 20; 1; 4.989758253097534
20; 20; 2; 6.005253314971924
20; 20; 3; 4.924513578414917
20; 20; 4; 4.863050699234009
21; 21; 0; 5.409443616867065
21; 21; 1; 5.522742033004761
21; 21; 2; 5.00721287727356
21; 21; 3; 5.091361999511719
21; 21; 4; 4.989336967468262
22; 22; 0; 4.795754432678223
22; 22; 1; 5.239341497421265
22; 22; 2; 4.822510004043579
22; 22; 3; 4.8308186531066895
22; 22; 4; 5.14035177230835
23; 23; 0; 4.875995635986328
23; 23; 1; 4.810211896896362
23; 23; 2; 5.312031030654907
23; 23; 3; 4.836001873016357
23; 23; 4; 4.916743993759155
24; 24; 0; 5.141464948654175
24; 24; 1; 4.895897150039673
24; 24; 2; 4.8106420040130615
24; 24; 3; 4.860838174819946
24; 24; 4; 4.9943695068359375

1377 MB file

threads; parts; i; time
2; 2; 0; 11.169979810714722
2; 2; 1; 12.649433135986328
2; 2; 2; 12.64449143409729
2; 2; 3; 12.657259464263916
2; 2; 4; 12.693336963653564

3; 3; 0; 9.518513917922974
3; 3; 1; 9.571817874908447
3; 3; 2; 9.539587020874023
3; 3; 3; 9.503049612045288
3; 3; 4; 9.549484968185425
4; 4; 0; 12.07082724571228
4; 4; 1; 12.081619501113892
4; 4; 2; 10.1875581741333
4; 4; 3; 11.876362085342407
4; 4; 4; 11.946631908416748
5; 5; 0; 10.019387483596802
5; 5; 1; 9.93368411064148
5; 5; 2; 10.02534794807434
5; 5; 3; 9.973225355148315
5; 5; 4; 9.839662551879883
6; 6; 0; 8.523430824279785
6; 6; 1; 8.519059896469116
6; 6; 2; 8.698403120040894
6; 6; 3; 8.555286169052124
6; 6; 4; 8.526690244674683
7; 7; 0; 7.6980578899383545
7; 7; 1; 7.532260179519653
7; 7; 2; 7.610573053359985
7; 7; 3; 7.6609790325164795
7; 7; 4; 7.630373001098633
8; 8; 0; 7.588699817657471
8; 8; 1; 8.124820232391357
8; 8; 2; 7.826935529708862
8; 8; 3; 7.676201820373535
8; 8; 4; 7.560831785202026
9; 9; 0; 7.712303876876831
9; 9; 1; 7.609032154083252
9; 9; 2; 7.532341003417969
9; 9; 3; 7.811867713928223
9; 9; 4; 8.30909252166748
10; 10; 0; 7.606166362762451
10; 10; 1; 7.8871543407440186
10; 10; 2; 7.502520322799683
10; 10; 3; 7.43290114402771
10; 10; 4; 7.757253408432007
11; 11; 0; 7.6144280433654785
11; 11; 1; 7.88922119140625
11; 11; 2; 7.41341757774353
11; 11; 3; 7.47533392906189
11; 11; 4; 7.489175319671631
12; 12; 0; 7.696140289306641
12; 12; 1; 7.825692653656006

12; 12; 2; 7.494961977005005
12; 12; 3; 7.251938819885254
12; 12; 4; 7.756883382797241
13; 13; 0; 7.458520174026489
13; 13; 1; 7.991739511489868
13; 13; 2; 7.588012933731079
13; 13; 3; 7.532933235168457
13; 13; 4; 7.560602426528931
14; 14; 0; 7.428539276123047
14; 14; 1; 7.437840461730957
14; 14; 2; 7.329972505569458
14; 14; 3; 7.49554181098938
14; 14; 4; 7.460112571716309
15; 15; 0; 7.208484888076782
15; 15; 1; 7.183613538742065
15; 15; 2; 7.095059633255005
15; 15; 3; 7.307342052459717
15; 15; 4; 7.181903600692749
16; 16; 0; 7.305602073669434
16; 16; 1; 7.358995676040649
16; 16; 2; 7.262423038482666
16; 16; 3; 7.165042161941528
16; 16; 4; 7.2798309326171875
17; 17; 0; 7.300049066543579
17; 17; 1; 7.353281021118164
17; 17; 2; 7.200004577636719
17; 17; 3; 7.209147691726685
17; 17; 4; 7.135981798171997
18; 18; 0; 8.022638082504272
18; 18; 1; 7.258249282836914
18; 18; 2; 7.325464963912964
18; 18; 3; 7.525938034057617
18; 18; 4; 7.232431173324585
19; 19; 0; 7.2758026123046875
19; 19; 1; 7.058107852935791
19; 19; 2; 7.14412522315979
19; 19; 3; 7.391486167907715
19; 19; 4; 7.273197174072266
20; 20; 0; 7.58905291557312
20; 20; 1; 7.33687949180603
20; 20; 2; 7.43955397605896
20; 20; 3; 7.351610422134399
20; 20; 4; 7.22730278968811
21; 21; 0; 7.409696817398071
21; 21; 1; 7.331563949584961
21; 21; 2; 7.356695652008057
21; 21; 3; 7.207636117935181

21; 21; 4; 7.342055559158325
22; 22; 0; 7.25664758682251
22; 22; 1; 7.363528728485107
22; 22; 2; 7.52045750617981
22; 22; 3; 7.415174961090088
22; 22; 4; 7.216646909713745
23; 23; 0; 7.307894706726074
23; 23; 1; 7.412282943725586
23; 23; 2; 7.279107332229614
23; 23; 3; 7.0359416007995605
23; 23; 4; 7.480842113494873
24; 24; 0; 7.251620054244995
24; 24; 1; 7.1050941944122314
24; 24; 2; 7.722868204116821
24; 24; 3; 7.736133575439453
24; 24; 4; 7.287385702133179

1836 MB file

threads; parts; i; time
2; 2; 0; 15.333295345306396
2; 2; 1; 16.846818208694458
2; 2; 2; 16.89298367500305
2; 2; 3; 16.814071655273438
2; 2; 4; 16.94606304168701
3; 3; 0; 12.613967895507812
3; 3; 1; 12.62196135520935
3; 3; 2; 12.622909307479858
3; 3; 3; 12.585439682006836
3; 3; 4; 12.62214970588684
4; 4; 0; 13.463301420211792
4; 4; 1; 15.865137577056885
4; 4; 2; 15.98202919960022
4; 4; 3; 13.874597311019897
4; 4; 4; 13.66763186454773
5; 5; 0; 13.302692890167236
5; 5; 1; 13.142772436141968
5; 5; 2; 13.321333885192871
5; 5; 3; 13.354174137115479
5; 5; 4; 13.371187925338745
6; 6; 0; 11.348493099212646
6; 6; 1; 11.399630069732666
6; 6; 2; 11.364320039749146
6; 6; 3; 11.39313006401062
6; 6; 4; 11.334156036376953
7; 7; 0; 10.194431781768799
7; 7; 1; 10.0536789894104
7; 7; 2; 10.243787050247192

7; 7; 3; 10.026723384857178
7; 7; 4; 10.229522228240967
8; 8; 0; 10.546931982040405
8; 8; 1; 10.574367761611938
8; 8; 2; 10.12274980545044
8; 8; 3; 10.528894186019897
8; 8; 4; 10.318014860153198
9; 9; 0; 10.280654668807983
9; 9; 1; 10.210265398025513
9; 9; 2; 10.177366018295288
9; 9; 3; 10.08040738105774
9; 9; 4; 10.045994758605957
10; 10; 0; 10.23366904258728
10; 10; 1; 10.256286382675171
10; 10; 2; 10.151060342788696
10; 10; 3; 10.02030324935913
10; 10; 4; 10.025511980056763
11; 11; 0; 10.045711040496826
11; 11; 1; 9.935594320297241
11; 11; 2; 9.869433403015137
11; 11; 3; 10.295708656311035
11; 11; 4; 10.11257290840149
12; 12; 0; 10.377053022384644
12; 12; 1; 10.240468740463257
12; 12; 2; 10.094614267349243
12; 12; 3; 10.152620792388916
12; 12; 4; 10.174321174621582
13; 13; 0; 9.911816358566284
13; 13; 1; 9.949748516082764
13; 13; 2; 9.982895374298096
13; 13; 3; 10.001947402954102
13; 13; 4; 9.893924951553345
14; 14; 0; 9.868454694747925
14; 14; 1; 9.69982099533081
14; 14; 2; 9.938599824905396
14; 14; 3; 9.881905555725098
14; 14; 4; 9.891145467758179
15; 15; 0; 10.055797100067139
15; 15; 1; 10.076286315917969
15; 15; 2; 9.534838914871216
15; 15; 3; 9.529908180236816
15; 15; 4; 9.540908813476562
16; 16; 0; 9.730239629745483
16; 16; 1; 9.637408971786499
16; 16; 2; 9.729732275009155
16; 16; 3; 10.356102466583252
16; 16; 4; 9.513161420822144

17; 17; 0; 9.64376163482666
17; 17; 1; 9.703303575515747
17; 17; 2; 9.625109672546387
17; 17; 3; 9.687269926071167
17; 17; 4; 9.60399079322815
18; 18; 0; 9.555036783218384
18; 18; 1; 9.617037773132324
18; 18; 2; 9.614023208618164
18; 18; 3; 9.575489521026611
18; 18; 4; 9.578495264053345
19; 19; 0; 10.392997980117798
19; 19; 1; 9.517624378204346
19; 19; 2; 9.380934953689575
19; 19; 3; 9.352191925048828
19; 19; 4; 9.408839702606201
20; 20; 0; 9.826740503311157
20; 20; 1; 9.924376726150513
20; 20; 2; 9.608911514282227
20; 20; 3; 9.95989179611206
20; 20; 4; 9.891538619995117
21; 21; 0; 9.781179189682007
21; 21; 1; 9.728749990463257
21; 21; 2; 9.83387017250061
21; 21; 3; 9.678431510925293
21; 21; 4; 9.836724281311035
22; 22; 0; 9.832699060440063
22; 22; 1; 10.318568468093872
22; 22; 2; 10.5569748878479
22; 22; 3; 9.686455726623535
22; 22; 4; 10.040098905563354
23; 23; 0; 9.757434129714966
23; 23; 1; 9.314077615737915
23; 23; 2; 9.532939195632935
23; 23; 3; 10.019240617752075
23; 23; 4; 9.59271240234375
24; 24; 0; 9.414386749267578
24; 24; 1; 9.482690811157227
24; 24; 2; 9.484129190444946
24; 24; 3; 9.791040420532227
24; 24; 4; 10.058252334594727
2; 3; 0; 10.225850820541382
2; 3; 1; 11.718693733215332
2; 3; 2; 11.774205207824707
2; 3; 3; 11.726470232009888
2; 3; 4; 11.736743927001953
2; 4; 0; 16.751455545425415
2; 4; 1; 16.776192903518677

2; 4; 2; 16.736833095550537
2; 4; 3; 16.76661777496338
2; 4; 4; 16.79950523376465
2; 5; 0; 14.039953231811523
2; 5; 1; 14.02452826499939
2; 5; 2; 14.042206764221191
2; 5; 3; 14.053919076919556
2; 5; 4; 14.117568016052246
3; 4; 0; 9.99531102180481
3; 4; 1; 10.031470537185669
3; 4; 2; 10.021962881088257
3; 4; 3; 10.008198261260986
3; 4; 4; 10.043594121932983
3; 5; 0; 14.901890516281128
3; 5; 1; 14.871399641036987
3; 5; 2; 14.855791807174683
3; 5; 3; 14.875925302505493
3; 5; 4; 14.876579761505127
3; 6; 0; 12.55096173286438
3; 6; 1; 12.523515224456787
3; 6; 2; 12.481531143188477
3; 6; 3; 12.541426420211792
3; 6; 4; 12.528174877166748
4; 5; 0; 13.15460467338562
4; 5; 1; 13.130901575088501
4; 5; 2; 13.036494970321655
4; 5; 3; 13.01246166229248
4; 5; 4; 13.02140736579895
4; 6; 0; 12.525734186172485
4; 6; 1; 12.665049076080322
4; 6; 2; 12.509825468063354
4; 6; 3; 12.683187246322632
4; 6; 4; 12.671388149261475
4; 7; 0; 11.20969295501709
4; 7; 1; 11.184772968292236
4; 7; 2; 11.116456508636475
4; 7; 3; 11.25132441520691
4; 7; 4; 11.257601261138916
5; 6; 0; 11.12259292602539
5; 6; 1; 11.201432943344116
5; 6; 2; 11.1497483253479
5; 6; 3; 11.214973211288452
5; 6; 4; 11.160795450210571
5; 7; 0; 12.055468797683716
5; 7; 1; 11.272855997085571
5; 7; 2; 11.336492538452148
5; 7; 3; 11.347594976425171

5; 7; 4; 11.299616813659668
5; 8; 0; 14.824941873550415
5; 8; 1; 12.822134494781494
5; 8; 2; 14.356011390686035
5; 8; 3; 13.988942861557007
5; 8; 4; 16.246515035629272
6; 7; 0; 10.087706089019775
6; 7; 1; 11.356496095657349
6; 7; 2; 10.175281047821045
6; 7; 3; 10.177193403244019
6; 7; 4; 9.880218744277954
6; 8; 0; 11.211910724639893
6; 8; 1; 16.550249099731445
6; 8; 2; 14.727752923965454
6; 8; 3; 16.42145013809204
6; 8; 4; 10.048686742782593
6; 9; 0; 14.988988876342773
6; 9; 1; 14.356834888458252
6; 9; 2; 12.128447532653809
6; 9; 3; 14.687348127365112
6; 9; 4; 13.902979373931885
7; 8; 0; 9.063411712646484
7; 8; 1; 9.028158903121948
7; 8; 2; 9.027875185012817
7; 8; 3; 9.031661987304688
7; 8; 4; 8.968091487884521
7; 9; 0; 14.690387964248657
7; 9; 1; 14.682060241699219
7; 9; 2; 14.669357538223267
7; 9; 3; 14.747083187103271
7; 9; 4; 14.515413761138916
7; 10; 0; 13.459108352661133
7; 10; 1; 13.514287948608398
7; 10; 2; 13.523886442184448
7; 10; 3; 13.660500526428223
7; 10; 4; 13.56286096572876
8; 9; 0; 9.25555968284607
8; 9; 1; 9.338200092315674
8; 9; 2; 9.326081991195679
8; 9; 3; 9.58026933670044
8; 9; 4; 9.46570873260498
8; 10; 0; 13.89117693901062
8; 10; 1; 13.033979415893555
8; 10; 2; 13.515982151031494
8; 10; 3; 13.865566968917847
8; 10; 4; 13.995025873184204
8; 11; 0; 12.566312313079834

8; 11; 1; 12.710740566253662
8; 11; 2; 12.394574880599976
8; 11; 3; 12.663302183151245
8; 11; 4; 12.599889278411865
9; 10; 0; 9.678433418273926
9; 10; 1; 9.424230098724365
9; 10; 2; 9.580726861953735
9; 10; 3; 9.422023296356201
9; 10; 4; 9.265196561813354
9; 11; 0; 13.030354738235474
9; 11; 1; 13.133134841918945
9; 11; 2; 13.1613929271698
9; 11; 3; 14.014001846313477
9; 11; 4; 13.1542387008667
9; 12; 0; 12.675937175750732
9; 12; 1; 12.745989561080933
9; 12; 2; 12.435895681381226
9; 12; 3; 12.665143489837646
9; 12; 4; 12.358785152435303
10; 11; 0; 9.401549816131592
10; 11; 1; 10.034760236740112
10; 11; 2; 9.33821702003479
10; 11; 3; 9.3426194190979
10; 11; 4; 9.705013751983643
10; 12; 0; 12.780786275863647
10; 12; 1; 12.987531661987305
10; 12; 2; 12.741139888763428
10; 12; 3; 12.558784008026123
10; 12; 4; 13.056843042373657
10; 13; 0; 12.051939249038696
10; 13; 1; 12.255029678344727
10; 13; 2; 11.78379774093628
10; 13; 3; 12.18848443031311
10; 13; 4; 11.908034563064575
11; 12; 0; 9.54860520362854
11; 12; 1; 9.205434083938599
11; 12; 2; 9.205846071243286
11; 12; 3; 9.407979726791382
11; 12; 4; 9.1297926902771
11; 13; 0; 12.926320552825928
11; 13; 1; 12.498820781707764
11; 13; 2; 12.658674716949463
11; 13; 3; 12.775068283081055
11; 13; 4; 11.917809963226318
11; 14; 0; 12.174001932144165
11; 14; 1; 11.905399560928345
11; 14; 2; 12.253548383712769

11; 14; 3; 12.173987627029419
11; 14; 4; 12.13727855682373
12; 13; 0; 9.213051080703735
12; 13; 1; 9.791273355484009
12; 13; 2; 9.348400115966797
12; 13; 3; 9.584836721420288
12; 13; 4; 9.251200437545776
12; 14; 0; 12.540553331375122
12; 14; 1; 11.897972822189331
12; 14; 2; 11.867610454559326
12; 14; 3; 12.178441762924194
12; 14; 4; 11.84830117225647
12; 15; 0; 11.74101710319519
12; 15; 1; 11.902405500411987
12; 15; 2; 11.801936149597168
12; 15; 3; 11.741109132766724
12; 15; 4; 11.57973599433899
13; 14; 0; 9.651309967041016
13; 14; 1; 9.625476121902466
13; 14; 2; 9.481859683990479
13; 14; 3; 9.627171993255615
13; 14; 4; 9.245818138122559
13; 15; 0; 11.439289808273315
13; 15; 1; 11.839872121810913
13; 15; 2; 10.97079849243164
13; 15; 3; 11.945432662963867
13; 15; 4; 11.879683256149292
13; 16; 0; 11.837560415267944
13; 16; 1; 11.5397047996521
13; 16; 2; 11.967454433441162
13; 16; 3; 11.847322225570679
13; 16; 4; 11.843432903289795
14; 15; 0; 9.418494701385498
14; 15; 1; 9.286189556121826
14; 15; 2; 9.498473644256592
14; 15; 3; 9.128997325897217
14; 15; 4; 9.356186628341675
14; 16; 0; 11.359632968902588
14; 16; 1; 12.027220487594604
14; 16; 2; 11.561415672302246
14; 16; 3; 11.131266593933105
14; 16; 4; 10.740788698196411
14; 17; 0; 11.54667043685913
14; 17; 1; 12.020808696746826
14; 17; 2; 11.01969599723816
14; 17; 3; 11.386135578155518
14; 17; 4; 11.693111181259155

15; 16; 0; 9.060620069503784
15; 16; 1; 9.124414205551147
15; 16; 2; 9.069246053695679
15; 16; 3; 8.955002546310425
15; 16; 4; 8.985918521881104
15; 17; 0; 12.591375827789307
15; 17; 1; 11.589674472808838
15; 17; 2; 11.846633195877075
15; 17; 3; 12.71322226524353
15; 17; 4; 12.500209331512451
15; 18; 0; 12.40169620513916
15; 18; 1; 12.40330719947815
15; 18; 2; 12.206022500991821
15; 18; 3; 12.451056241989136
15; 18; 4; 11.409948587417603
16; 17; 0; 9.140667915344238
16; 17; 1; 9.16381311416626
16; 17; 2; 9.507192850112915
16; 17; 3; 9.568517446517944
16; 17; 4; 9.370420932769775
16; 18; 0; 12.383825540542603
16; 18; 1; 12.64214038848877
16; 18; 2; 11.515597581863403
16; 18; 3; 12.274936437606812
16; 18; 4; 12.525648355484009
16; 19; 0; 12.091243743896484
16; 19; 1; 12.130249261856079
16; 19; 2; 11.265078783035278
16; 19; 3; 12.540275812149048
16; 19; 4; 12.087217092514038
17; 18; 0; 9.276318073272705
17; 18; 1; 9.242934226989746
17; 18; 2; 9.180132389068604
17; 18; 3; 9.21198558807373
17; 18; 4; 9.266850233078003
17; 19; 0; 12.377530097961426
17; 19; 1; 11.29396390914917
17; 19; 2; 12.14478611946106
17; 19; 3; 12.184097528457642
17; 19; 4; 12.460594654083252
17; 20; 0; 12.436039209365845
17; 20; 1; 12.059046030044556
17; 20; 2; 12.273751258850098
17; 20; 3; 11.528265476226807
17; 20; 4; 12.327784061431885
18; 19; 0; 9.327099323272705
18; 19; 1; 9.251083850860596

18; 19; 2; 9.254140138626099
18; 19; 3; 9.135546445846558
18; 19; 4; 9.1076340675354
18; 20; 0; 12.137340307235718
18; 20; 1; 12.2872896194458
18; 20; 2; 12.16214895248413
18; 20; 3; 12.50695252418518
18; 20; 4; 12.077784061431885
18; 21; 0; 11.741506576538086
18; 21; 1; 11.816670417785645
18; 21; 2; 11.916003942489624
18; 21; 3; 11.8755464553833
18; 21; 4; 11.765974521636963
19; 20; 0; 9.122866153717041
19; 20; 1; 9.100477457046509
19; 20; 2; 9.04523754119873
19; 20; 3; 9.379606008529663
19; 20; 4; 9.064191341400146
19; 21; 0; 11.779472827911377
19; 21; 1; 12.04414677619934
19; 21; 2; 11.984267234802246
19; 21; 3; 12.478188514709473
19; 21; 4; 12.175537824630737
19; 22; 0; 11.59219241142273
19; 22; 1; 11.676460266113281
19; 22; 2; 11.846041679382324
19; 22; 3; 11.330451250076294
19; 22; 4; 11.218959093093872
20; 21; 0; 9.394661903381348
20; 21; 1; 9.210243225097656
20; 21; 2; 9.289798021316528
20; 21; 3; 9.539265155792236
20; 21; 4; 9.306569814682007
20; 22; 0; 11.496504783630371
20; 22; 1; 11.832014799118042
20; 22; 2; 11.41121530532837
20; 22; 3; 11.839616537094116
20; 22; 4; 11.683916091918945
20; 23; 0; 11.005003690719604
20; 23; 1; 11.450972080230713
20; 23; 2; 11.656380891799927
20; 23; 3; 11.40780234336853
20; 23; 4; 11.108836650848389
21; 22; 0; 9.35202407836914
21; 22; 1; 9.254137516021729
21; 22; 2; 9.476943492889404
21; 22; 3; 9.266134262084961

21; 22; 4; 9.545651197433472
21; 23; 0; 11.235914468765259
21; 23; 1; 11.74433422088623
21; 23; 2; 11.75151801109314
21; 23; 3; 11.574331283569336
21; 23; 4; 11.074695348739624
21; 24; 0; 11.205450773239136
21; 24; 1; 11.267396926879883
21; 24; 2; 10.822227001190186
21; 24; 3; 11.515425205230713
21; 24; 4; 11.300536870956421
22; 23; 0; 9.211407899856567
22; 23; 1; 9.248399257659912
22; 23; 2; 9.284655809402466
22; 23; 3; 9.308408975601196
22; 23; 4; 9.326447486877441
22; 24; 0; 11.189809322357178
22; 24; 1; 10.662423133850098
22; 24; 2; 11.548677206039429
22; 24; 3; 11.213705778121948
22; 24; 4; 10.729117155075073
22; 25; 0; 11.333751916885376
22; 25; 1; 11.207059860229492
22; 25; 2; 11.038039684295654
22; 25; 3; 10.739969253540039
22; 25; 4; 11.142162561416626
23; 24; 0; 8.912641763687134
23; 24; 1; 9.170289278030396
23; 24; 2; 8.982739210128784
23; 24; 3; 8.962525367736816
23; 24; 4; 9.275045156478882
23; 25; 0; 11.609531879425049
23; 25; 1; 11.362231254577637
23; 25; 2; 12.665630340576172
23; 25; 3; 11.70033073425293
23; 25; 4; 11.890239477157593
23; 26; 0; 10.740344762802124
23; 26; 1; 11.118008136749268
23; 26; 2; 11.279337882995605
23; 26; 3; 11.198941707611084
23; 26; 4; 11.520267248153687
24; 25; 0; 9.128893375396729
24; 25; 1; 9.352426052093506
24; 25; 2; 9.157090902328491
24; 25; 3; 9.234499454498291
24; 25; 4; 9.567725419998169
24; 26; 0; 11.330049276351929

24; 26; 1; 11.328798294067383
24; 26; 2; 11.332713603973389
24; 26; 3; 11.37369966506958
24; 26; 4; 11.241392850875854
24; 27; 0; 10.9593346118927
24; 27; 1; 11.109243154525757
24; 27; 2; 10.901697635650635
24; 27; 3; 10.967724323272705
24; 27; 4; 11.279257535934448

A.5 Get Tokens and Interpret them Immediately

459 MB file

0; 5.684483051300049
1; 5.697651147842407
2; 5.6687023639678955
3; 5.65347695350647
4; 5.635324239730835

918 MB file

0; 11.3733491897583
1; 11.396219491958618
2; 11.300142526626587
3; 11.335394620895386
4; 11.326725006103516

1377 MB file

0; 16.97883367538452
1; 16.927834510803223
2; 16.938838481903076
3; 17.058173656463623
4; 17.020673990249634

1836 MB file

threads; parts; i; time
2; 2; 0; 12.788670539855957
2; 2; 1; 13.556326389312744
2; 2; 2; 13.496263265609741
2; 2; 3; 13.529265403747559
2; 2; 4; 13.571284532546997
3; 3; 0; 10.192696571350098
3; 3; 1; 10.200490474700928
3; 3; 2; 10.380575895309448
3; 3; 3; 10.571521043777466

3; 3; 4; 10.384373664855957
4; 4; 0; 13.138819217681885
4; 4; 1; 13.525922060012817
4; 4; 2; 13.46910834312439
4; 4; 3; 11.8429434299469
4; 4; 4; 11.358203887939453
5; 5; 0; 11.192975282669067
5; 5; 1; 11.216751098632812
5; 5; 2; 11.127444505691528
5; 5; 3; 11.160232067108154
5; 5; 4; 11.083632230758667
6; 6; 0; 9.662487030029297
6; 6; 1; 9.58927869796753
6; 6; 2; 9.554301261901855
6; 6; 3; 9.55219578742981
6; 6; 4; 9.702538013458252
7; 7; 0; 8.668906927108765
7; 7; 1; 8.646862745285034
7; 7; 2; 8.677087545394897
7; 7; 3; 8.66260838508606
7; 7; 4; 8.611644983291626
8; 8; 0; 8.900697231292725
8; 8; 1; 8.622090339660645
8; 8; 2; 9.030452251434326
8; 8; 3; 8.577407121658325
8; 8; 4; 8.598071813583374
9; 9; 0; 9.032681941986084
9; 9; 1; 8.835063695907593
9; 9; 2; 8.71563196182251
9; 9; 3; 8.627982378005981
9; 9; 4; 8.666740894317627
10; 10; 0; 8.44123101234436
10; 10; 1; 8.823607921600342
10; 10; 2; 8.464938640594482
10; 10; 3; 8.515480995178223
10; 10; 4; 8.443562507629395
11; 11; 0; 8.434444427490234
11; 11; 1; 8.423215627670288
11; 11; 2; 8.638120412826538
11; 11; 3; 8.356096506118774
11; 11; 4; 8.446194171905518
12; 12; 0; 8.473012447357178
12; 12; 1; 8.4103262424469
12; 12; 2; 8.730968236923218
12; 12; 3; 8.279907703399658
12; 12; 4; 8.560417652130127
13; 13; 0; 8.563105344772339

13; 13; 1; 8.580832958221436
13; 13; 2; 8.370094537734985
13; 13; 3; 8.681121826171875
13; 13; 4; 8.533451557159424
14; 14; 0; 8.270603656768799
14; 14; 1; 8.218002796173096
14; 14; 2; 8.438312292098999
14; 14; 3; 8.262165307998657
14; 14; 4; 8.296711683273315
15; 15; 0; 8.07123851776123
15; 15; 1; 8.039655447006226
15; 15; 2; 8.102413654327393
15; 15; 3; 7.986111402511597
15; 15; 4; 8.186776399612427
16; 16; 0; 8.04846978187561
16; 16; 1; 8.054174900054932
16; 16; 2; 8.347478151321411
16; 16; 3; 8.312690734863281
16; 16; 4; 8.063177585601807
17; 17; 0; 8.034317016601562
17; 17; 1; 7.9979493618011475
17; 17; 2; 8.131180047988892
17; 17; 3; 8.04263186454773
17; 17; 4; 8.18734860420227
18; 18; 0; 8.635046482086182
18; 18; 1; 8.035287857055664
18; 18; 2; 7.983266353607178
18; 18; 3; 8.221981763839722
18; 18; 4; 8.202171564102173
19; 19; 0; 8.000172853469849
19; 19; 1; 7.938717603683472
19; 19; 2; 8.017958164215088
19; 19; 3; 8.327906370162964
19; 19; 4; 7.9155449867248535
20; 20; 0; 8.179020404815674
20; 20; 1; 8.22004747390747
20; 20; 2; 8.203316688537598
20; 20; 3; 8.171312808990479
20; 20; 4; 8.121801853179932
21; 21; 0; 8.141727924346924
21; 21; 1; 8.224933385848999
21; 21; 2; 8.357909440994263
21; 21; 3; 8.168915748596191
21; 21; 4; 8.270437002182007
22; 22; 0; 8.099329233169556
22; 22; 1; 8.165600299835205
22; 22; 2; 8.349928617477417

22; 22; 3; 8.10308837890625
22; 22; 4; 8.32566237449646
23; 23; 0; 7.821270704269409
23; 23; 1; 8.204171419143677
23; 23; 2; 8.033974409103394
23; 23; 3; 8.122055292129517
23; 23; 4; 8.053260087966919
24; 24; 0; 8.341841697692871
24; 24; 1; 8.100516557693481
24; 24; 2; 8.23569655418396
24; 24; 3; 8.465953350067139
24; 24; 4; 8.309066772460938

A.6 Build up Statistics

A.6.1 Statistic before Tasks

threads; parts; i; time
2; 2; 0; 31.840819358825684
2; 2; 1; 33.92733573913574
3; 3; 0; 29.16814637184143
3; 3; 1; 28.37638831138611
4; 4; 0; 30.140077590942383
4; 4; 1; 29.827077627182007
5; 5; 0; 28.949597358703613
5; 5; 1; 29.281116724014282
6; 6; 0; 27.238547325134277
6; 6; 1; 27.277579307556152
7; 7; 0; 26.607458114624023
7; 7; 1; 26.545225620269775
8; 8; 0; 26.9787495136261
8; 8; 1; 26.398918867111206
9; 9; 0; 26.93246603012085
9; 9; 1; 26.306195735931396
10; 10; 0; 26.950472354888916
10; 10; 1; 26.992431640625
11; 11; 0; 26.50733733177185
11; 11; 1; 26.95758557319641
12; 12; 0; 26.429866552352905
12; 12; 1; 26.506593465805054
13; 13; 0; 26.31982970237732
13; 13; 1; 26.607473611831665
14; 14; 0; 25.99662137031555
14; 14; 1; 27.539554834365845
15; 15; 0; 26.32862901687622
15; 15; 1; 26.57988476753235
16; 16; 0; 25.90336298942566
16; 16; 1; 26.21211290359497

17; 17; 0; 26.660654306411743
17; 17; 1; 26.687901258468628
18; 18; 0; 25.856456995010376
18; 18; 1; 26.187333345413208
19; 19; 0; 25.909522533416748
19; 19; 1; 26.084969520568848
20; 20; 0; 26.242486000061035
20; 20; 1; 25.69759464263916
21; 21; 0; 25.74015974998474
21; 21; 1; 26.548271894454956
22; 22; 0; 25.733455419540405
22; 22; 1; 25.8167781829834
23; 23; 0; 25.936447381973267
23; 23; 1; 25.83998942375183
24; 24; 0; 25.093294620513916
24; 24; 1; 25.45053005218506

A.6.2 Statistic as Task

threads; parts; i; time
2; 2; 0; 23.166685342788696
2; 2; 1; 22.141868829727173
3; 3; 0; 22.863210439682007
3; 3; 1; 22.796582460403442
4; 4; 0; 23.534923315048218
4; 4; 1; 28.702109336853027
5; 5; 0; 35.184582233428955
5; 5; 1; 35.49239540100098
6; 6; 0; 34.68421649932861
6; 6; 1; 38.8381142616272
7; 7; 0; 40.33648753166199
7; 7; 1; 40.643123626708984
8; 8; 0; 40.67575025558472
8; 8; 1; 41.34840774536133
9; 9; 0; 41.90084266662598
9; 9; 1; 41.73187470436096
10; 10; 0; 42.935622692108154
10; 10; 1; 44.05286264419556
11; 11; 0; 43.71732997894287
11; 11; 1; 41.82308602333069
12; 12; 0; 43.60588598251343
12; 12; 1; 42.568440198898315
13; 13; 0; 42.66556167602539
13; 13; 1; 45.49223804473877
14; 14; 0; 48.15483617782593
14; 14; 1; 47.505239486694336
15; 15; 0; 51.19782900810242

15; 15; 1; 52.09050440788269
16; 16; 0; 52.917410373687744
16; 16; 1; 52.78210091590881
17; 17; 0; 55.999412298202515
17; 17; 1; 55.06817579269409
18; 18; 0; 50.99715042114258
18; 18; 1; 52.882848024368286
19; 19; 0; 51.94090437889099
19; 19; 1; 54.72609043121338
20; 20; 0; 45.44247126579285
20; 20; 1; 55.47311735153198
21; 21; 0; 51.07176399230957
21; 21; 1; 53.79232478141785
22; 22; 0; 54.34403371810913
22; 22; 1; 52.949633836746216
23; 23; 0; 54.956303358078
23; 23; 1; 58.64877939224243
24; 24; 0; 51.19544315338135
24; 24; 1; 54.34854865074158

Bibliography

[AMi20] AMiner. Open academic graph. Website, May 2020. Available online at https://www.aminer.org/open-academic-graph; visited on May 15th, 2020.; Licensed under ODC-By url-https://opendatacommons.org/licenses/by/1.0/. (cited on Page 10)

[Bra17] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, RFC Editor, December 2017. (cited on Page 1 and 5)

[GJ08] Dick Grune and Ceriel J. H. Jacobs. *Introduction*, pages 1–4. Springer New York, New York, NY, 2008. (cited on Page 10)

[HDPW20] Thorsten Hoeger, Chris Dew, Finn Pauls, and Jim Wilson. NDJSON-spec. Website, April 2020. Available online at https://github.com/ndjson/ndjson-spec; visited on April 8th, 2020. (cited on Page 6)

[INT20] ECMA INTERNATIONAL. The json data interchange syntax. Website, May 2020. Available online at https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf; visited on May 15th, 2020. (cited on Page 5)

[LKC+17] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast json parser for data analytics, 2017. Available online at http://www.vldb.org/pvldb/vol10/p1118-li.pdf; visited on May 27th, 2020. (cited on Page 73)

[LL19] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second, 2019. Available online at https://arxiv.org/abs/1902.08318; visited on May 27th, 2020. (cited on Page 73)

[M+65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965. Available online at https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf; visited on May 26th, 2020. (cited on Page 1)

[P+20] Marcus Pinnecke et al. Carbonspec. Website, April 2020. Available online at http://www.carbonspec.org/carbon.html; visited on April 7th, 2020. (cited on Page 1)

[PCZ+19] Marcus Pinnecke, Gabriel Campero, Roman Zoun, David Broneske, and Gunter Saake. Protobase: It's About Time for Backend/Database Co-Design. In Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke, editors, *BTW 2019 - Workshopband*, volume P-289 of *Lecture Notes in Informatics (LNI)*, pages 515–518. Gesellschaft für Informatik, mar 2019. (cited on Page 1)

[PP10] Christof Paar and Jan Pelzl. *Hash Functions*, pages 293–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. (cited on Page 8)

[RR13] Thomas Rauber and Gudula Rünger. *Introduction*, pages 1–7. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. (cited on Page 7)

[sim] simdjson.org. The simdjson library. Available online at https://simdjson.org/; visited on May 27th, 2020. (cited on Page 73)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 24.06.2020