SheetReader: Efficient Specialized Spreadsheet Parsing

Haralampos Gavriilidis [a,*], Felix Henze [b,1], Eleni Tzirita Zacharatou [c,1], Volker Markl [d]

[a] Technische Universität Berlin, Germany
[b] Access Microfinance Holding AG, Germany
[c] IT University of Copenhagen, Denmark
[d] Technische Universität Berlin and DFKI GmbH, Germany

Keywords:
Data loading
Spreadsheet parser
Parsing parallelization

a b s t r a c t

Spreadsheets are widely used for data exploration. Since spreadsheet systems have limited capabilities, users often need to load spreadsheets to other data science environments to perform advanced analytics. However, current approaches for spreadsheet loading suffer from either high runtime or memory usage, which hinders data exploration on commodity systems. To make spreadsheet loading practical on commodity systems, we introduce a novel parser that minimizes memory usage by tightly coupling decompression and parsing. Furthermore, to reduce the runtime, we introduce optimized spreadsheet-specific parsing routines and employ parallelism. To evaluate our approach, we implement prototypes for loading Excel spreadsheets into R and Python environments. Our evaluation shows that our novel approach is up to 3× faster while consuming up to 40× less memory than state-of-the-art approaches.

Artifact Availability: The source code is available at https://github.com/fhenz/SheetReader-r.

1. Introduction

Spreadsheets are widely used for data exploration and analysis due to their intuitive layout [1,2]. While modern spreadsheet systems provide some analysis tools, such as PivotTables and aggregation formulas, they do not support more advanced tasks, such as iterative analyses and model building. As a result, to perform their analyses, users turn to more specialized data science environments, such as R and Python, that provide ecosystems with a plethora of data science libraries. However, loading spreadsheets into different environments poses an important bottleneck with respect to both runtime performance and memory consumption.

Consider the following real-world example, which refers to a common use case in financial organizations. A data scientist needs to determine factors indicative of default risk from loan data for particular businesses. To predict the repayment capacity, she wants to run a logistic regression analysis. The relevant data is stored in spreadsheets and includes information about the business, such as sales, inventory, and years of activity. For data exploration and model validation on her laptop, she uses the R language and the corresponding libraries. Since the data is only available in spreadsheet files, before training the model, the first preprocessing step consists of loading the data. However, she notices that her data pipeline is slow. To illustrate the performance of such a pipeline, we show a runtime performance breakdown for different libraries in Fig. 1. Specifically, we compare two state-of-the-art Excel parsers for R (i.e., *openxlsx* and *readxl*) with the highly optimized CSV parser (*data.table*) when processing the same data in the appropriate format. We observe that when running this pipeline on CSV files, loading the dataset takes as long as training the model. On the other hand, when it comes to spreadsheets, the data loading step dominates the runtime completely when using state-of-the-art libraries. Of course, the CSV format is very different from the spreadsheet format; however, with our experiment, we aim to show that existing techniques for spreadsheet loading pose important bottlenecks in current data science pipelines. We provide the experimental setup and configuration in Section 8.

Fig. 2 further illustrates the inefficiency of existing solutions for the same real-world data. We observe that the fastest Excel parser takes around 30 s to load 172 MB of data while consuming up to 13 GB of memory. Compared to the CSV parser that only takes 4 s and consumes up to 1.1 GB of memory, this is an overhead of 7.5× for runtime and almost 12× for memory usage. In contrast, the most memory-efficient Excel parser consumes up to 5 GB memory, which represents an overhead of 4.5×, but is 40× slower, taking 160 s to parse the file. This performance gap is due to the fact that spreadsheet parsers are not specialized to exploit the spreadsheet file structure. Consequently, and given that many users work on commodity hardware (e.g., business

* Corresponding author.
E-mail addresses: gavriilidis@tu-berlin.de (H. Gavriilidis), felix.henze@accessholding.com (F. Henze), elza@itu.dk (E. Tzirita Zacharatou), volker.markl@tu-berlin.de (V. Markl).

[1] Part of the work done while the author was at Technische Universität Berlin.

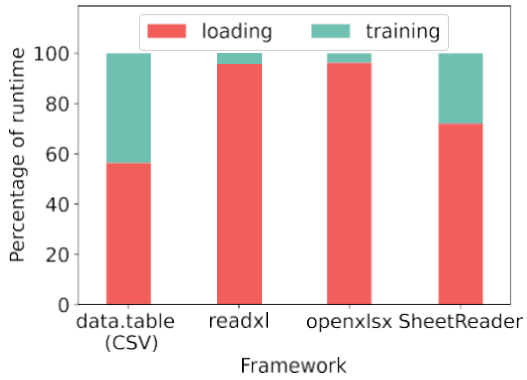


Fig. 1. Runtime performance breakdown for a data science pipeline that loads data from raw files. Prior to model training, raw data is loaded using state-of-the-art CSV and spreadsheet parsers. Loading data from spreadsheets dominates the runtime. Our approach alleviates the spreadsheet data loading bottleneck.

laptops, desktops), loading spreadsheets can easily become a significant bottleneck in data science applications.

To perform efficient analyses, users need tools that allow them to quickly load their spreadsheet data without consuming a large amount of resources. However, although spreadsheets are widely used among data scientists, there has been little work on interoperability with data science environments. Typical spreadsheet applications, such as *Microsoft Excel* and *LibreOffice Calc*, store data as a collection of individually compressed XML structured files. Existing tools for converting these data collections into an appropriate data format for the target environment rely on general methods for decompression and XML parsing (i.e., DOM and SAX) [3,4]. Specifically, state-of-the-art DOM-based parsers materialize the entire XML file in memory, thereby suffering from high memory usage. In contrast, SAX-based parsers expose a large number of parsing events through their event-based API, suffering from bad runtime performance.

We believe that generalized XML parsers are not well suited to achieve good loading performance in data science environments, as they suffer from high runtime performance and excessive memory usage. We argue that a specialized solution is needed to overcome the runtime and memory consumption bottlenecks when loading spreadsheets in data science environments. The solution should exploit the spreadsheet file structure and its characteristics and perform the parsing in the shortest time feasible while also minimizing memory consumption. We propose *SheetReader*, an efficient specialized spreadsheet parser. As Fig. 1 shows, SheetReader significantly reduces the data loading bottleneck for spreadsheet files in data science pipelines, achieving a better balance between data loading and model training. Furthermore, we introduce two parsing approaches for SheetReader with trade-offs between runtime performance and memory consumption. Our first approach, *consecutive* parsing, achieves very fast loading times by heavily utilizing parallelization. The second approach, *interleaved* parsing, while also employing parallelization, primarily aims to minimize memory consumption by tightly coupling decompression and parsing.

Contributions. Our contributions are summarized as follows:

- We introduce spreadsheet-specific optimizations and employ parallelism that requires minimal synchronization to reduce the runtime for spreadsheet parsing. Furthermore, we minimize memory utilization by tightly coupling decompression and parsing.
- We introduce two parsing approaches that allow users to choose between runtime and memory utilization based on their needs. The *consecutive* approach achieves fast loading times through massive parallelization, but its memory utilization is data-dependent. In contrast, the *interleaved* approach uses a configurable and constant amount of memory while still achieving low runtime.
- We provide a general solution for different data science environments, by storing the parsed data in an environment-agnostic intermediate data structure.
- We experimentally show that SheetReader outperforms the existing solutions by up to 3× and 40× in terms of runtime and memory utilization, respectively. We also show that parallelizing the decompression further reduces the runtime by around 35%.

This paper extends [5] with the following contributions:

- We present the challenges of spreadsheet parsing and analyze in great detail both theoretically and experimentally the inherent limitations and inefficiencies of state-of-the-art DOM-based and SAX-based parsers.
- We propose a new technique for parsing spreadsheets without dimension and location information, thereby generalizing *SheetReader* to different spreadsheet applications. In addition, we evaluate this new technique experimentally.
- Finally, we develop a new prototype for *SheetReader* implemented in Python. Unlike our previous work that only evaluated an R prototype of *SheetReader*, this paper also presents experimental results for our Python prototype.

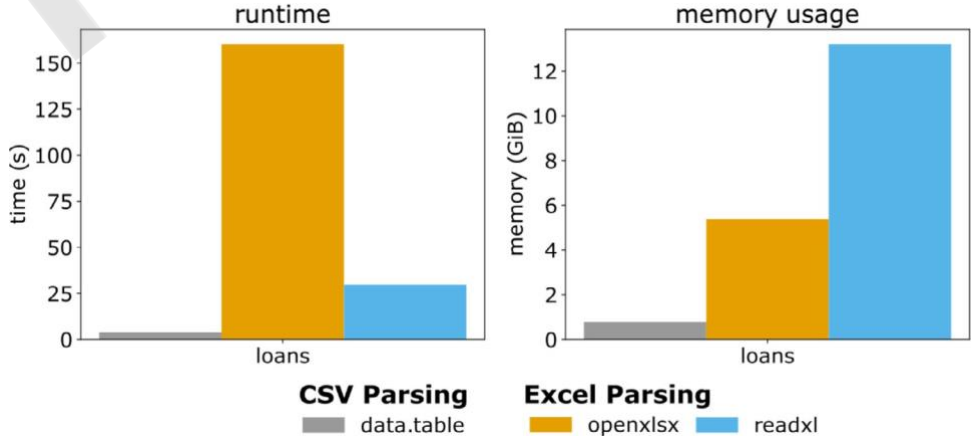


Fig. 2. Performance of existing R packages for parsing a real-world spreadsheet and the corresponding CSV file.

Fig. 3. Spreadsheet structure (simplified).

Outline. Next, after introducing the background in Section 2, we analyze the limitations of state-of-the-art spreadsheet parsers theoretically and further validate our theoretical observations experimentally in Section 3. Then, we give an overview of *SheetReader* and describe its parsing approaches in Sections 4 and 5. Furthermore, we describe additional parsing techniques and spreadsheet-specific parsing optimizations in Sections 6 and 7, respectively. Section 8 presents an experimental evaluation of both our R and Python prototypes against state-of-the-art solutions using real-world and synthetic datasets. We discuss the related work in Section 9 and conclude in Section 10.

2. Background

Spreadsheet standards. We describe the structure of a spreadsheet, focusing on the file format currently used by Excel, known as Office Open XML (OOXML) and standardized as ECMA-376 [6]. Part 2 of ECMA-376 specifies the Open Packaging Conventions (OPC) that describes the general structure of OOXML files. According to OPC, OOXML files are ZIP archives containing a collection of XML files, with some restrictions on file names and extensions for describing the types and relationships.

Fig. 3 provides a simplified overview of the spreadsheet structure. Excel documents consist of a workbook that can contain several worksheets. The workbook determines the names, IDs, and archive locations of all the spreadsheets. The worksheets, e.g., `sheet1.xml` in the figure, store the actual data. Additionally, Excel saves strings in a separate file from the actual worksheets, `sharedStrings.xml`, where they are referenced by index. The top level reserved files contain metadata that allows to identify files relevant for further processing and serve as an entry point for programs. Specifically, valid Excel files require the top level relationship file `/_rels/.rels` that specifies the locations of the workbook and the shared strings file inside the archive.

As stated in the OOXML specification, the XML files in the ZIP archive can be either uncompressed or compressed using the Deflate format. Deflate [7] is a block-based compression format with dynamic block sizes. It arranges the blocks in a stream and compresses them individually. Although it is possible to compress an entire document into a single large block, using smaller blocks typically improves the compression ratio. Within a block, Deflate uses duplicate string elimination, a technique where duplicate series of byte streams are replaced with back-references to the previous identical byte stream, which can in turn also be a back-reference. Back-references can point to previous blocks, as long as the distance does not exceed a sliding window of the last 32 KB of decompressed data. As a result, Deflate documents are challenging to decompress in parallel, because to decompress a given block, all previous blocks need to be decompressed first.

XML parsing. There exist two dominant approaches for XML parsing, DOM (Document Object Model) and SAX (Simple API for XML) [3,4]. The DOM approach maps the XML file contents to an in-memory tree and provides an interface that allows to In contrast, the SAX approach exposes an event handling interface. While traversing the XML document, the SAX parser fires events for the found XML entities (tags), which then trigger the previously registered handlers. DOM is well-suited for random access. However, a major disadvantage regarding resource consumption is that it needs to materialize the whole document in memory before parsing it. SAX parsers do not experience this bottleneck. However, the event handling interface makes it challenging to keep track of the entire document while parsing it, and leads to inefficient implementations.

Even though DOM and SAX approaches provide a solution for parsing XML documents, they are both very generic, i.e., *they are designed to support arbitrary XML structures*. We observe that for parsing spreadsheets it is not necessary to employ such generic approaches, as spreadsheet XML files have a very specific XML file structure that is defined by their specification. We argue that a *specialized parser for spreadsheets* can exploit their specific structure and find the sweet spot between DOM and SAX approaches, thereby offering reasonable memory consumption and fast runtime performance at the same time.

3. State-of-the-art spreadsheet parsers and their limitations

Overall, the goal of this work is to provide an efficient spreadsheet parser, which keeps memory consumption low and provides high runtime performance at the same time. In this section, we describe the general requirements for spreadsheet loading and analyze how existing XML parsing approaches can be employed for spreadsheets. In particular, we focus on their inherent limitations with regard to our optimization goals (low memory consumption, high runtime performance), and validate our hypotheses experimentally. Finally, we discuss our findings and conclude that to achieve our optimization goals, it is necessary to design a new specialized spreadsheet parser.

3.1. Spreadsheet parsing challenges

Designing a spreadsheet parser is a non-trivial task due to the complexity of the spreadsheet file format. Spreadsheets separate data into multiple individually compressed XML files and store strings in a dedicated file that is shared by all worksheets. Before discussing existing approaches for spreadsheet parsing, we examine the challenges that arise from the spreadsheet file format. First, the XML files comprising a spreadsheet need to be converted into a suitable data structure that can be returned to the user. To facilitate the transfer of data from different spreadsheet file types to different software environments, we examine the requirements for a general intermediate data structure. Then, we discuss how to handle string parsing and decompression.

Intermediate data structure. Overall, we assume that a spreadsheet parser requires to store data in an intermediate data structure, before this is mapped to a specific runtime environment data structure, e.g. R or Python Pandas dataframes. Spreadsheets contain a lot of metadata, which are not necessary when mapping them to tabular-like abstractions in data processing environments. We examine the data types found in spreadsheets and

Table 1
Mapping from spreadsheet (Excel in particular) types to native data types.

Spreadsheet type	Internal type	Size (bytes)
Boolean	Boolean	1
Date	Integer	8
Error		
Inline string	Integer	8
Number	Float	8
Shared string	Integer	8
Formula string	Integer	8

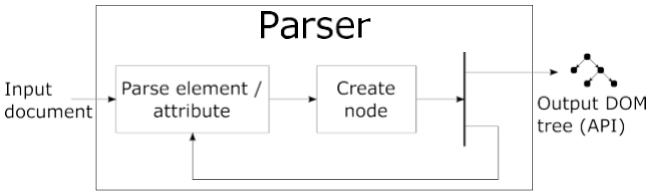


Fig. 4. Overview of the DOM parsing process.

how to store the data that is strictly necessary in an intermediate data structure. While keeping the memory footprint minimal is a major concern, we refrain from using techniques that would significantly affect runtime adversely, such as in-memory compression. A cell value can be one of seven simple types or blank. For an intermediate data structure, these types can be mapped to booleans, (unsigned) integers, and floats as shown in Table 1. This means that cell values can be stored in a type union of boolean, 8-byte integer, and 8-byte float, resulting in a total of 8 bytes of data plus 1 byte of type information per stored value. String tables, however, are not included in this calculation, as their memory usage depends on the number of unique strings and their lengths. For utmost memory efficiency, they are typically stored as null-terminated strings in contiguous memory blocks. However, considering the performance penalty of seeking through these blocks to locate wanted strings, it may be more efficient to store the strings in an array-like data structure using pointers. Inline strings and formula strings can be directly converted into shared strings by adding their values to the string table (or even a different one) and storing the table index. For cells of type Date (d), the cell value as stored in the file (presumably an ISO formatted date) is simply read as is and similarly stored as a string. Errors can be stored as any type as long as they are marked the appropriate type.

String handling. As discussed, Excel saves strings in a separate file from the actual worksheets and references them by index, allowing for simultaneous parsing of the string table and the worksheet. However, the exact strings that need to be parsed can only be determined during the parsing of the worksheet. As all worksheets share a single string table, there may be significantly more strings present than is actually needed. With regard to memory consumption, this means that the total consumption for parallel string and worksheet parsing would be the sum of both. With regard to runtime performance, an improvement can be achieved depending on the sizes of the shared string table and the worksheet. A potential bottleneck, however, arises with a large number of unnecessary strings.

On the other hand, parsing the strings after the worksheet would allow us to only parse strings that are strictly needed since we parsed all cells requiring strings beforehand. With regard to memory consumption, this means that we require only the maximum memory requirements of each of the tasks (worksheet and string parsing). With regard to runtime performance, this depends on the number of actually used strings.

Decompression. Spreadsheets are ZIP archives that contain individual compressed XML files. This means that to parse spreadsheets, one needs to first decompress them. In spreadsheets, individual files are compressed as *Deflate* streams. A Deflate stream consists of multiple blocks that are individually optimized for their respective contents. Each block possesses a short header signaling its compression method and if it is the last block of the stream. However, a plain Deflate stream does not contain any information about the stream itself, such as size (compressed and decompressed) or error-checking values. To compress individual files using the Deflate algorithm and maintain their metadata, one can use the well-established *gzip* file format. Generally, a ZIP archive (restricted according to the OPC standard) contains multiple files, in our case the individual XML files. Additionally, a ZIP archive contains a central directory with metadata, most importantly the uncompressed and compressed size, filename, and offset inside the archive. Most of this data is also present in the local file header, located at the offset for that respective file, immediately followed by the actual compressed data.

The particular file structure of spreadsheets, i.e., that individual XML files are stored as separate Deflate streams with known metadata, presents an opportunity for parallelizing the decompression process across multiple worksheets and string files. However, the Deflate format by default makes no accommodations for parallelizing the parsing of a single stream (i.e. file), as the back-references hinder attempts at fully parallel decompression. Any block may require an arbitrary number of previous blocks to be decompressed first to resolve these back-references. Because the spreadsheet XML contains a large portion of repeated strings (the repeating structural elements), even the last compressed block may require information from the first block.

Nevertheless, as spreadsheet loading comprises decompression and parsing tasks, there is an opportunity to parallelize the decompression and parsing process. For that, there exist two approaches: A sequential approach where the whole document is decompressed before parsing it, and an interleaved approach where the document is split into chunks which are decompressed and parsed in an interleaved manner. While the sequential approach is easier to handle, it may require a large amount of memory, as it needs all decompressed files in memory. Depending on the decompression method, it might also need to load the complete compressed data in memory. To avoid consuming such a large amount of memory, the interleaved approach requires only a constant amount of memory that is reused throughout decompression and parsing. In the first step, we decompress a chunk of the document, while in the second step we parse the decompressed content. Of course, those steps are repeated until the end of the document, and can also be executed in parallel for multiple chunks. However, to parallelize the process, we assume that the decompression and parsing steps can handle arbitrary chunks of XML documents, which is a challenging task. Overall, the sequential decompression/parsing approach is simple but requires holding the whole decompressed document in memory, while the interleaved is more complex but allows to recycle memory, and hence only requires a constant amount of memory.

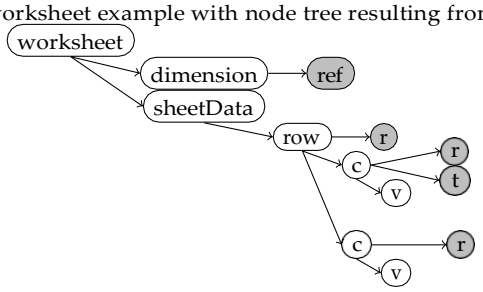
3.2. Parsing spreadsheets with DOM

The tree structure found in a spreadsheet's XML files naturally fits the Document Object Model (DOM). Therefore, one can parse the XML file into a DOM tree, as shown in Fig. 4.

DOM abstractly defines the capabilities that an XML document API should expose. In particular, it supports navigation within the tree, i.e. accessing parent, children, and sibling nodes from any node. Standard parsers do this by storing easily retrievable

Listing 1: Excel worksheet example with node tree resulting from DOM parsing

```
<worksheet>
    <dimension ref="A1:D2" />
    <sheetData>
        <row r="2">
            <c r="B2" t="s">
                <v>1</v>
            </c>
            <c r="D2">
                <v>1.23</v>
            </c>
        </row>
    </sheetData>
</worksheet>
```




```
<row r="2" x14ac:dyDescent="0.25" spans="1:5">
```

Listing 2: Excel worksheet XML extract: A row opening tag

references to these related elements inside each element. Indirect methods that involve traversing the tree to find such elements are also possible. Other capabilities such as namespaces, XPath navigation, and node event handlers are also specified [8], but not relevant to our work and thus not discussed further.

Regarding memory consumption, DOM parsing requires parsing the whole XML document into a tree data structure. We show an example of such a process in Listing 1. We estimate a lower bound for the memory required for storing a parsed spreadsheet XML document in a DOM tree structure. We achieve this by analyzing the relationships between the spreadsheet data (rows and cells) and the resulting XML file (XML elements and attributes). This allows us to calculate the total number of XML elements and attributes depending on the spreadsheet size (number of rows and cells). We then use a theoretical minimal implementation that provides a reasonable minimum of the capabilities specified by DOM (navigation) to determine how much memory is required for the elements and attributes stored as nodes in a tree data structure. We use the resulting estimate to decide whether DOM parsing is a viable approach for our use case.

Listing 1 shows the basic composition of the XML elements involved. We will only discuss the contents of the sub-tree originating from the *sheetData* element. According to the specification, the *sheetData* element contains only *row* elements. We further assume that each *row* element contains the same number of *c* (cell) elements and no others (the other possible elements are the so-called ''Extension Lists''). Additionally, every *c* element contains a single value element (either *v* or *is*, depending on the cell type), which contains the actual cell value as text. If the cell value is derived from a formula, the value element would have an additional *f* element as a sibling, but for simplification, we assume none of the cells contain formulas. Because XML allows the content of an element to be a mix of text and other XML elements, sections of text also need to be stored as nodes in the tree. We observe that every spreadsheet cell corresponds to two XML elements and one text node in parent–child relationships, and every spreadsheet row corresponds to one XML element.

Additionally, the elements can have a number of attributes, some of which are relevant to parsing the data but also some that are not, e.g., attributes that specify the row height. For this discussion, we assume that every *row* and every *cell* element has the respective *r* attribute containing the corresponding location information. If the cell type is not a number or that number should be interpreted differently (e.g. as a date) or the type is explicitly stated even for a number, another attribute is added (*t* for type or *s* for style). Excel in particular omits explicitly specifying the type for numbers, but also adds some attributes to *row* elements that are not relevant for parsing the data. This can be seen in Listing 2, namely the *x14ac:dyDescent* and *spans* attributes. In total, this results in three attributes for every spreadsheet row (of which two are irrelevant), one attribute for every cell, one additional attribute for every non-numeric cell, and finally one additional attribute for every styled cell. We analyzed the number of nodes and attributes retrieved from Excel worksheets of differing sizes by a representative DOM library (RapidXML) and show the results in Table 2.

Table 2
Number of XML nodes (element and text nodes) and attributes for worksheets with the given number of rows. The number of columns is 100, all cells have the number type, and there are no formula or inline string elements. Using the RapidXML DOM parsing library.

Rows	XML nodes	XML attributes
10 000	3 010 308	1 030 223
50 000	15 050 308	5 150 223
100 000	30 100 308	10 300 223
200 000	60 200 308	20 600 223
300 000	90 300 308	30 900 223
400 000	120 400 308	41 200 223
500 000	150 500 308	51 500 223
600 000	180 600 308	61 800 223

When storing element nodes in a tree data structure and providing access through a DOM API, how the relationships between the nodes are stored needs to be balanced between efficient storage and efficient access. Nodes could contain information about all adjacent nodes, meaning parent, siblings, and children, and thus offer direct and fast navigation of their immediate neighborhood. This requires storage that allows for referencing at least three nodes (parent, previous sibling, next sibling) and some dynamic list of child nodes. On the contrary, nodes can also contain only minimal relationship information, and navigation between nodes is done indirectly. At a minimum, nodes only need to store references to two nodes, for example, their next sibling and their first child node. All other navigation from some starting node would be accomplished by traversing the tree through a sequence of these references. Additional necessary information would be stored in the traversing procedure and returned upon encountering the starting node, e.g., for retrieving the previous sibling of the starting node, the most recently visited node while

traversing would be stored and returned upon arriving at the starting node. The attributes of elements can be stored similarly, directly as some list inside the element node data structure, or as a singly-linked list with only a reference to the first attribute.

Elements and attributes need to store more than just their relationships to other elements or attributes. All elements have a name, which is present in their respective opening and closing tag and describes their role in the document, e.g., the *row* element opening tag starts with its name `row`. Elements also have a value, consisting simply of the text between their opening and closing tags. Similarly, attributes also have a name and value, e.g., the *r* attribute seen in Listing 2 has the name `r` and the value `2`. We group methods for storing names and values into two approaches.

When keeping the original document in memory, i.e., some memory stays reserved for and filled by the entire document content, elements and attributes can simply reference certain document positions paired with lengths to store their names and values. Nodes would thus only need four further fields, two respectively for name and value. The length can be encoded either simply as the actual length of the targeted string or as a reference to its end. If the original document is not used for purposes other than as name and value storage, the character after a targeted string can be set to null, thus null-terminating the string and removing the need for storing its length separately (halving the number of fields required). This is possible because any content that will be modified has already been parsed and all names and values are followed by some character that is guaranteed to not be part of another name or value. Element names are followed by a whitespace, `>`, or `/` character, and element values are terminated by a closing tag (which will have been parsed to determine the end of the value) beginning with a `<` character, attribute names are followed by a whitespace or a structural equal sign and attribute values are ended by a structural quotation mark. While potentially costly memory allocations and copies are avoided for storing names and values, there are also disadvantages. It is effectively impossible to perform storage optimizations like string deduplication, and superfluous structural characters (`<`, `>`, `/`, etc.) are also kept in memory.

If it is not possible to keep the document in memory, or simply not desired, names and values will need to be copied and stored separately. Elements and attributes themselves are stored virtually identically to the previous approach, just that the references point to different locations in memory. The parser needs to allocate and manage this memory for storing the names and values. There are methods to reduce the required memory compared to naively copying every and all required strings individually. One basic technique is string deduplication. If some string is encountered that matches a string already present in the storage, the reference can be pointed to the existing string instead of storing the duplicate and pointing to that. Even if a document contains only unique elements, the strings for the opening tag and the closing tag of an element are the same, and deduplication would thus reduce memory usage considerably.

We estimate a lower bound for the total memory a DOM tree structure requires using a theoretical minimal implementation. This minimal implementation fulfills only part of the DOM API, arbitrary navigation from any node as a starting point. We disregard memory used for storing the actual data of dynamic content (element and attribute names and values) and make an estimate with only structural elements. This is done by counting the references for all nodes and multiplying that number by some factor that represents the storage required for a reference.

We arrive at a minimum of 5 references per element node, 3 references per text node, and 3 references per attribute. All elements and attributes have associated names and values, requiring at least two references each, assuming that the names and values are zero-terminated to determine their length. Previously we determined that at a minimum, element nodes need to reference two other nodes, for example their first child and next sibling. Leaf nodes are an exception (all text nodes are leaf nodes), which cannot have children and thus require only a reference to their next sibling. Additionally, element nodes (apart from text nodes) require a reference to their attributes. Attributes themselves only require a single reference to their next sibling attribute.

Table 3
Size of the DOM tree structure in memory with a minimal implementation (see Eq. (1)) compared to the uncompressed source document size. Tree node references have a size of 8 bytes. The number of columns is 100, all cells have the number type, and there are no formula or inline string elements.

Rows	Worksheet size (MB)	Calculated minimum tree size (MB) (relative increase)
10 000	43	129 (3.0)
50 000	219	643 (2.9)
100 000	439	1286 (2.9)
200 000	891	2573 (2.9)
300 000	1342	3859 (2.9)
400 000	1793	5146 (2.9)
500 000	2244	6432 (2.9)
600 000	2695	7718 (2.9)

The analysis of the relationship between spreadsheet data and XML elements has shown that every spreadsheet cell results in two element nodes and one text node, and every row in one element node. Disregarding irrelevant attributes and assuming every cell has neither their type specified (i.e. they are numeric) nor a style assigned, every cell and every row has one attribute each. Following Eq. (1), with n_{ref} being the total number of references, n_{cell} the number of cells containing a value, and n_{row} the number of rows containing at least one filled cell, this results in 16 references per spreadsheet cell and 8 references per spreadsheet row. This is also illustrated in Fig. 5.

$$\begin{aligned} n_{ref} &= 5 * 2 * n_{cell} + 3 * n_{cell} + 5 * n_{row} + 3 * n_{cell} + 3 * n_{row} \\ &= 16 * n_{cell} + 8 * n_{row} \end{aligned} \tag{1}$$

Assuming that references are implemented as pointers to memory addresses, their size depends on the system architecture. For 64-bit systems, the memory addresses are usually 64 bits wide as well, meaning references have a size of 8 bytes. Calculated estimates using 8 bytes per reference are shown in Table 3. Results for other architectures or implementations can be obtained by applying an appropriate factor to our estimates, e.g., 32-bit systems would require half the memory. It is evident that even a theoretical minimal DOM tree implementation leads to memory usage that is multiple times larger than that of the uncompressed source document. For 8-byte references, we can see that the DOM tree requires around three times the memory of the plain XML document, which is stable for all considered worksheet sizes. This factor is obviously influenced by the non-structural content of the document, i.e., the cell values.

Using Eq. (1) and 8 bytes per reference results in 128 bytes per cell, meaning the size of the cell content would need to approach or exceed this value to bring the factor below 1. This can theoretically be achieved with only numerical cells and thus without additional elements or attributes, but due to the numerical precision limit of 17 digits, this seems unrealistic. String values are also unsuitable since the strings themselves are lifted out of the worksheet and replaced by integer indices. A feasible method to achieve a higher ratio of content to structure is formulas that can grow to be of considerable size depending on their complexity. Formulas introduce an additional XML element and text node, thus raising the number of references for a cell from 16 to 21

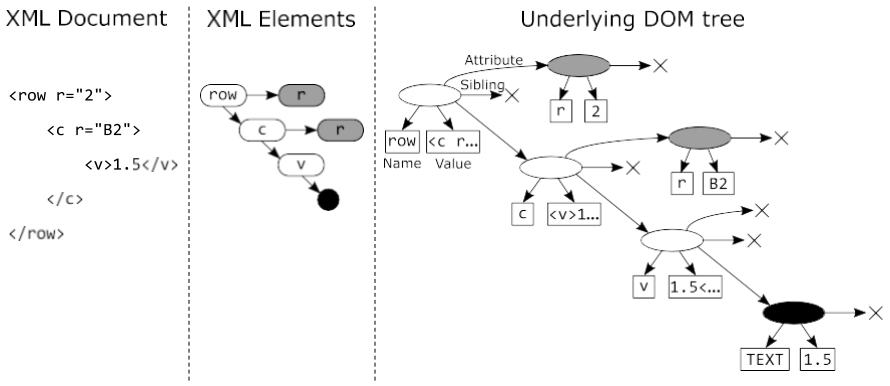


Fig. 5. The relationship between spreadsheet data and XML elements, and the underlying DOM tree structure.

resulting in 168 bytes per cell. Altogether we suppose that the minimum tree size exceeds the uncompressed source worksheet size in all but a few exceptional cases where the majority of cells contain formulas of considerable length.

The excessive tree size is exacerbated considerably when examining DOM tree implementations other than the theoretical minimum. State-of-the-art libraries for DOM parsing implement tree nodes (for elements and attributes) such that they additionally possess references to their parent and/or their previous sibling [9–12]. Some approaches using linked lists for element children and attributes instead of arrays also add references to the last child or attribute of an element node [9,11]. Additionally, there is often no differentiation between full nodes, leaf nodes, and text nodes, where unnecessary references to children and attributes could be avoided [9,11,12]. Consequently, the number of references per cell can easily reach 32, leading to double the required memory of the minimal tree explored previously.

In light of these findings, we only briefly discuss the additional storage required for the names and values of the elements and attributes. These considerations are not applicable to approaches that keep the source document in memory and use it to store the strings. Because there are only a few unique element and attribute names, the storage required for these is virtually constant and negligible next to all else. The only attribute values that change are for the locations of the rows and cells (the *r* attribute), and these are unique for every row and cell. Because they are completely dependent on the cell locations, the total storage required can be estimated simply from the worksheet dimensions (very accurately assuming that all cells in this range are serialized in the XML, that is, there are no blank cells). The length for the cell location string is determined by the column (col) and row (row) of the cell following $len = \lfloor \log_{26}(col - 1) \rfloor + 1 + \lfloor \log_{10}(row) \rfloor + 1$, since the column is alphabetically encoded (1 equals A and 27 equals AA) while the row simply increases for every power of 10. The cell values themselves (including formulas) are completely dependent on the dataset at hand, and thus their size cannot be reliably determined beforehand.

Ultimately we deem DOM parsing unsuitable for loading spreadsheets. Even with a theoretical minimal implementation, the tree structure itself is significantly larger than the initial document. This added cost makes building and keeping the complete tree in memory prohibitively expensive on current consumer machines, where this can feasibly lead to saturation of all available memory. Additionally, the main benefits provided by maintaining the DOM structure in memory, such as easier navigation of nodes and manipulation of the tree, are not needed for extracting the values from the worksheets. Once all information for a cell has been determined (location, type, value, etc.), access to it is no longer required and it can be discarded.

3.3. *Parsing spreadsheets with SAX*

Another prominent approach for XML parsing is to use the Simple API for XML (SAX). Contrary to DOM, SAX is not an explicit specification, but rather describes a different mechanism for parsing. While originally developed for XML, there exist parsers for other languages based on this concept. SAX describes an event-driven model, where elements are handled individually during parsing. Parsers based on this approach usually expose an API where developers supply their own methods for handling different components of the document, modeled as events. A SAX parser reads and processes the input until it obtains enough information to create and send an event to the handler method. After the event is handled, the parser resumes collecting information for the next event until it reaches the end of the document. The granularity of these events can vary depending on the parser, from individual events for opening tags, closing tags, and single attributes to, for example, a single event for a complete tag (including all attributes and/or the element value). We show an example with SAX-generated events in Listing 3. For each opening and closing tag found during the XML traversal (on the left side), the corresponding event is called (on the right side).

Fundamentally, SAX parsing does not retain any information about previously parsed document parts. Where with DOM parsers, developers have access to the generated DOM tree for potential validation and further processing, SAX offers no such convenience. Developers are responsible for any higher-level state-keeping, validation, and processing. While distinctly disadvantageous regarding complexity, this is also an advantage over DOM parsing. Developers have full control over any processing at a higher level than the basic parsing routines.

Instead of creating a full DOM tree from the XML document, SAX parsing allows for the extraction of specific parts with immediate deserialization. Because SAX offers full control over which and how the parsed XML is stored, parsing does not lead to the memory bottlenecks observed with the DOM approach. We briefly discuss the memory characteristics for SAX parsing, in particular, how much state the parser itself still needs to store but also which and how much state the developer needs to maintain.

We now discuss the memory requirements of SAX parsing, i.e., what state the developer needs to keep when processing a document using SAX, since SAX parsers delegate the state-keeping responsibility to the developer. The parser itself only needs to store minimal state information, through which the required memory is effectively constant and can be considered negligible. Any additional memory required depends on the processing by the developer.

To estimate the complexity of state-keeping for the examined spreadsheet formats, we analyze their structure. Spreadsheets

Listing 3: Excel worksheet example with events generated during SAX parsing

```
<worksheet>
    <dimension ref="A1:D2" />
    <sheetData>
        <row r="2">
            <c r="B2" t="s">
                <v>1</v>
            </c>
            <c r="D2">
                <v>1.23</v>
            </c>
        </row>
    </sheetData>
</worksheet>
```

```
worksheet
dimension/   ref
sheetData
row   r
c   r   t
v   content   /v
/c
c   r
v   content   /v
/c
/row
/sheetData
/worksheet
```

mainly consist of repeating elements, rows, and cells. These elements are completely independent of each other in a valid spreadsheet file. Even when assuming an invalid file, the only state that would need to be carried over from previous elements is the location information, e.g., multiple rows where the location attribute specifies the same row number. All other state information only needs to be propagated to child elements, e.g. the type information from the *c* element to its *v* child element, to deserialize and interpret its value appropriately. The majority of state can be reused for the next sibling element, e.g., the type of the previous cell is irrelevant for the current cell and can thus be overwritten, keeping the required memory constant and to a minimum. Additionally, because the elements are handled by developer-supplied methods, values and attributes can be deserialized early. This avoids the bookkeeping of names and values stored in the most general format, i.e., strings, as is the case with content-unaware DOM approaches. An additional benefit over DOM is that unnecessary elements and attributes such as those for formulas can be discarded immediately, preventing resources from being wasted.

The sequential nature of SAX parsing lends itself to only keeping the relevant part of the source in memory instead of storing the complete document for the entire parse duration. Many SAX parsers have the ability of receiving the source document in parts, maintaining enough state to bridge between parts even if all previous content is no longer available [10,11,13]. This necessarily incurs some overhead, since names and values need to be copied to intermediate storage if they have not yet been forwarded to the developer handlers and the currently available source part ends. While this overhead is negligible in terms of total memory usage, the repeated allocations and copies may negatively affect the runtime. Loading the document in parts also means the optimization used by some DOM parsers of referencing the original document to store strings is more difficult to implement.

While employing SAX parsing means greater complexity and effort for the developer compared to DOM parsing, the very low memory requirement independent of document size is extremely valuable. This can be further augmented by loading, or in our case decompressing, only the relevant part of the source document into memory.

3.4. Parser compilers

Parser compilers are based on the concept of automatically creating parsers that are specialized to the documents at hand. The compiler accepts a description of the document structure, e.g., an XML schema, to generate and optimize a parser according to that. This creates various advantages over general parsers both in terms of runtime and memory, for example by early deserialization of primitive data types (integers, floats, booleans, etc.) instead of processing them further as strings. Additionally, parser compilers can expose a more convenient interface to the parsed content by creating custom data structures that are fitted to the elements in the supplied structure description.

Previous work has discussed the approach of compiling XML parsers based on schemas [14,15]. While it is primarily observed that combining the validation step with the parsing step on a low level and optimizing the result achieves an advantage in terms of runtime, this also has benefits for memory efficiency. The authors mention that compiling the parser from the schema allows them to hardcode tag names into generated code, whereas parsers without or not fully utilizing schema information would need to store these in dynamic memory for subsequent access. XML Screamer [15] additionally integrates some deserialization into the low-level optimized routines. While again improving runtime performance, we suppose that deserializing as early as possible would be the biggest benefit for reducing memory consumption. Because compiler-based approaches rely on pre-written low-level parsing routines, deserialization of nonstandard types is again relegated to later processing steps. Unfortunately, the minority of the attributes in the examined spreadsheet formats are standard primitives, and as such many cannot take advantage of this early optimization. The most common attribute, the cell location, is stored in spreadsheet form (''A1'') and as such needs custom deserialization. Additionally, how the cell value is deserialized depends on the values of attributes of the parent cell element. Consequently, the optimization of early deserialization is of limited benefit for spreadsheets.

A rather unintuitive disadvantage of parser compilers can be seen with the *xlsx* R package. Here, the Apache XMLBeans Java framework[2] is used to compile the Excel schema into a specialized parser and custom Java classes to store the parsed elements. The Excel standard specified various optional attributes for rows and cells that are only rarely used, for example, the cell attribute *ph* which indicates whether the cell should display phonetic information about the cell content. In total the row element possesses twelve optional attributes, of which only one is relevant for extracting the data, while the cell element has six, of which three are relevant.

When creating the Java classes for the row and cell elements, Apache XMLBeans turns the attributes, including the optional ones, into class members. This drastically increases the storage required for the class compared to general methods of storing the attributes. A general method such as a linked list or array will generally only take as much space as there are actual attributes, without taking into account any optional but non-existent ones. Parser compilers offer significant advantages for documents with large complexity due to improving performance while only requiring the XML schema. But these compilers are still general solutions themselves, in that they cannot specialize for non-standard types or structures or may introduce features that are beneficial for some use cases but detrimental for others.

3.5. Benchmarking state-of-the-art XML parsers for spreadsheets

To validate our theoretical analysis for memory requirements and evaluate the overall performance of existing standard XML parsing libraries for spreadsheets, we perform an empirical evaluation. For that, we use two synthetic spreadsheet files containing only numeric values with 100 columns and a varying row count and measure the memory consumption and runtime

[2] https://xmlbeans.apache.org/.

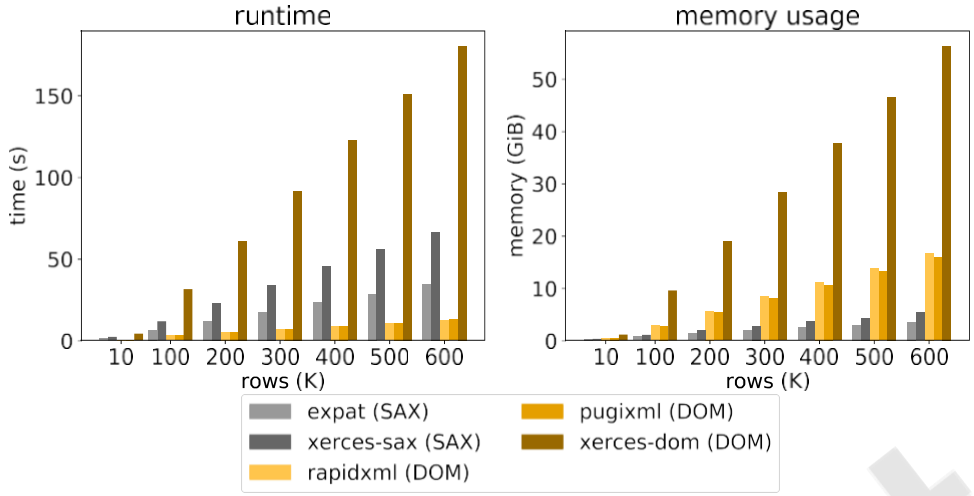


Fig. 6. Runtime performance (left) and memory usage (right) overview of state-of-the-art DOM and SAX libraries for spreadsheet parsing.

performance. The SAX libraries were tested with empty callbacks, i.e., the runtime is wholly determined by the parser itself without any further processing. We refer the reader to Section 8 for the detailed experimental setup.

The general consensus in literature seems to be that SAX parsing achieves superior runtime performance over DOM parsing [16–19]. Nevertheless, we observe the opposite in our experiment results (Fig. 6). While this may be specific to the libraries used in the experiments, we nonetheless discuss the factors that may cause SAX parsing to be slower than DOM parsing.

In particular, we can see that two of the DOM parsing libraries used in our experiments, *RapidXML* and *pugixml*, consistently achieve less than half the runtime of *Expat* or *Xerces*, the SAX libraries used. The DOM parser exposed by *Xerces* is built on-top of its SAX parser and thus cannot achieve better performance. We briefly expand on the consensus of SAX parsing being faster than DOM, followed by a more detailed discussion about the two libraries used in our experiments.

Previous work benchmarking XML parsing establishes SAX as superior to DOM both in terms of memory usage and runtime [16–19]. We postulate that this is because SAX parsing defers much of the processing to the developer, in that it only forwards the parsed elements to the callback functions. This facilitates specialization according to the developers' goals early in the parsing process. This is in contrast to DOM parsing, which is almost completely general and thus non-specialized during the whole parsing process and by creating a general DOM tree. We hypothesize that this specialization is the, or at least one of the main drivers for the runtime advantage of SAX over DOM that was found in previous work.

We attempt to explain why our experiments yield opposite results by examining two libraries in more detail. *RapidXML*, a DOM library used, describes itself as an *in-situ* parser. This means that the parser does not copy strings, but instead stores names and values as pointers to the source document. Consequently, the source document needs to be kept in memory even after parsing has concluded, since the generated DOM tree references it. Avoiding the costly operations associated with copying strings grants this type of parser a considerable runtime advantage.

It is our impression that memory operations, that is, allocation, copying, and freeing, are the main cause for the unexpected observed performance difference. In particular, *RapidXML* also uses self-managed memory pools for the generated DOM tree. This means that instead of allocating memory individually for the parsed elements, it preemptively allocates large blocks of memory that are then filled with the elements as they are parsed. This is generally better because it reduces system calls to the operating system for requesting memory and avoids memory fragmentation, which can help with cache locality.

Expat, one of the SAX libraries used, also utilizes memory pools, but for storing strings (i.e. names and values) instead of structural data. This is because storing structural information to the extent found in DOM parsing is not a concern with SAX. SAX parsers need to store the data for effectively only a single XML element. After parsing the element has concluded, the information is sent to the handler and afterward can be discarded, allowing the storage space to be reused for the next element.

Expat is unable to use the in-situ optimization from *RapidXML*, as it is designed to accept the input document in parts. It does not need to keep the complete source document in memory during the whole parsing process on the developer, so there is no guarantee that the whole name or value is available when invoking a callback function. It cannot simply supply the handler method with a pointer to the beginning of the element name or value in the source document, because it might be located in a previous block which may have since been freed. This means that *Expat* copies all name and value strings into intermediate storage, which is then exposed to the handler method.

We conclude that the observed runtime difference, i.e., DOM being faster than SAX parsing, stems from the in-situ parsing technique used by *RapidXML*. *Expat* has to copy the names and values of all elements and values it parses, while *RapidXML* simply creates pointers to the source document. The cost of copying seems to be the determining factor for the results of our runtime experiments. We broaden this conclusion to apply also to the other benchmarked libraries, in that guaranteed persistent access to the source document allows parsers to optimize how data is forwarded to any further processing steps without requiring potentially costly memory operations.

3.6. General findings and discussion

For our use case of parsing spreadsheets, we deem DOM parsing in any form unsuitable. We have shown that the memory required for even a minimal DOM tree implementation (without compression or compacting techniques) exceeds the size of the source document by a significant factor. Additionally, employing DOM parsing limits the number of potential approaches to the overall data loading process because creating the DOM tree is a prerequisite step for any further processing.

SAX parsing inherently possesses no such memory bottleneck since it does not store the document except for forwarding it to the developer. Unfortunately, the current SAX parsers that we tested do not yield satisfactory runtime performance compared to the other approaches. Contrary to expectations, the tested SAX parsers consistently take around double the runtime compared to the tested DOM parsers (excluding xerces-dom). The API imposed by SAX also limits lower-level optimizations, e.g., skipping irrelevant content for extracting the data we want.

Furthermore, we believe that parser compilers will not yield significant improvements over general parsers. The examined spreadsheet formats require custom deserialization for the majority of elements and attributes, reducing the effectiveness of the restricted low-level deserialization routines introduced by the compilers. Additionally, the subset of the spreadsheet XML structure relevant for extracting the desired data is of limited complexity, making manually created custom parsers feasible. Full control over the whole parsing process allows us to avoid the pitfalls of the more general approaches, while also granting full flexibility in terms of how parsing is integrated into the complete data loading process.

Overall, our findings show that each general approach has characteristics that do not allow us to reach our goal of memory efficiency and good runtime performance. Creating a custom parser specialized for the workload at hand offers most of the benefits of compiler-based approaches, while also removing any remaining restrictions imposed by the still somewhat general nature of parser compilers. Potential bottlenecks introduced by the compiler (Apache XMLBeans described earlier) can be avoided. While creating a custom parser manually can be rather complex, the relatively simple structure of the spreadsheet XML limits this to a manageable degree.

Similarly to compiler-based approaches, creating a specialized parser allows one to combine all steps of the parsing process, but even to the point of deserializing directly into the target data structure while characters are being read. This is much more similar to SAX than DOM, in that there is no intermediate data structure that stores the XML tree. Specialization eliminates many of the drawbacks of compiler-based and SAX parsing but also increases complexity due to the tight coupling between all parsing stages. All deserialization can be performed at the same step early in the parsing process, and the overhead of copying content for forwarding to event handlers is removed. The complexity is an argument for using parser compilers [14], in that handcrafting a low-level parser combining all steps gets unmanageable for highly complex schemas.

As shown previously, the schemas for worksheet XML files are relatively flat and simple. The structural relationships between rows, cells, and cell values remain consistent throughout the document. This is also demonstrated by the regular expression-based approach used by the *openxlsx* package. One exception is cells of the rarely used inline string type, where the cell value is in the *is* element rather than in the *v* element. The main complexity results from the simple type (*t*) and style (*s*) attributes of the cell element that determine how the cell value is to be deserialized.

In terms of memory efficiency, a custom parser only provides minor benefits compared to standard SAX parsing. On the other hand, given some assumptions about acceptable memory consumption, a custom parser allows fine-tuned management of the memory budget to allow other techniques or optimizations for improving other performance characteristics.

4. SheetReader

In the following, we give an overview of SheetReader's architecture. We show an overview of our approach in Fig. 7.

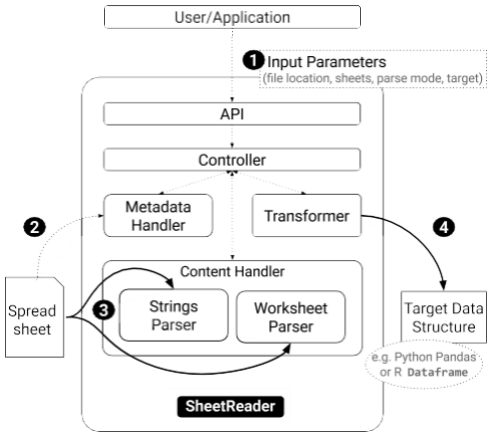


Fig. 7. SheetReader's architecture.

SheetReader expects as input parameters related to a spreadsheet file, and loads the worksheet contents into a data structure within the target environment. Users and applications submit parsing requests to SheetReader through its **API**, by providing I/O and parser configuration parameters (1). The **Controller** is the core component responsible for coordinating the overall loading routine. At first, the Controller fetches worksheet metadata (2), e.g. file location and sheet names, through the **Metadata Handler**. Then, the Controller initiates the loading routine by providing the sheet names and parse mode to the **Content Handler**, a component that decompresses input files and parses the spreadsheet content into an intermediate data structure (3). The Content Handler has two modules, the **Strings Parser**, which is responsible for parsing the shared strings XML file, and the **Worksheet Parser**, which is responsible for parsing the worksheets containing numeric data. These two parsers may operate in parallel, and have two different parsing modes that we describe in 5. To avoid costly reallocations for resizing the intermediate data structure during parsing, the Controller pre-allocates memory by relying on the available metadata, such as the file offset, archive size, and total strings number in the shared strings file. Our parsers assume valid spreadsheets as input, since spreadsheet systems, e.g. Excel, are unlikely to produce corrupt files.

When parsing is completed, the **Transformer** executes the final loading step, i.e., creating the target data structure (4). Contrary to the worksheet, SheetReader stores the cell data in column-wise data layout. This allows to transform intermediate data to column-based target data structures widely found in data science environments, e.g., R and Python Pandas Dataframes, without reconverting the layout. Additionally, SheetReader's internal intermediate data structure enables to reuse its core parsing routines in different runtimes by only implementing the Transformer interface and language bindings. This interface exposes methods for transforming the intermediate data structure into a target data structure, e.g., our prototype Transformer implementation in R converts intermediate data into a DataFrame.

SheetReader is implemented in C++ in a target-agnostic way, to offload performance-critical computations to a native runtime [20]. The parser does not use any target-specific data types (except for optimization with target defines) and only the interface that forwards user options and converts to the target data type is specific to the particular target environment. Besides the build process and compiler option, the only difference in C++ source files between our R and Python versions is the singular interface file. Thus it is possible to incorporate SheetReader in any language/environment that provides C/C++ bindings with capabilities for table- or list-like data structures.

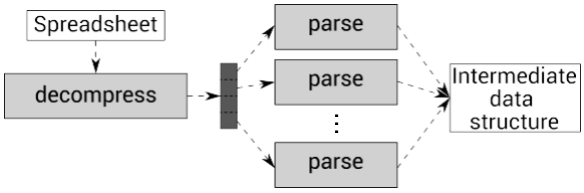


Fig. 8. Consecutive parsing.

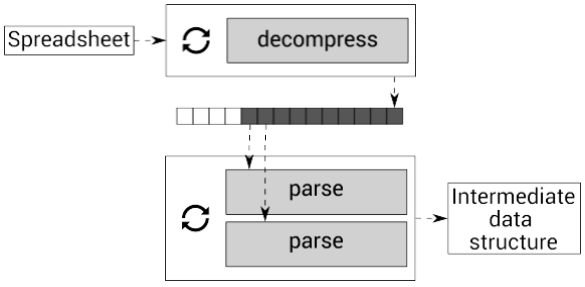


Fig. 9. Interleaved parsing.

5. Spreadsheet parsing

We introduce two parsing approaches that both the Strings and the Worksheet Parser components can use. These approaches express a trade-off between runtime performance and memory efficiency. The *consecutive* approach is optimized for runtime performance and the *interleaved* one for minimal memory usage. Users can choose their preferred approach based on their needs.

Both approaches rely on the same general parsing routine that we outline here. As our parser targets specialized XML documents, it operates by finding the opening and closing tags for specific XML elements. For example, as shown in Fig. 3, in Excel files, cell values `<v>val</v>` are enclosed in `<c></c>` tags, where the character sequence `<c` indicates the opening tag for a new cell. Inside this tag, there are attributes that contain cell metadata. Attributes are name-value pairs that are linked through the = character and are separated from other pairs by a whitespace. We parse this metadata as it defines the cell location and type, which we use to determine where to store the cell data in our intermediate data structure. The character > denotes the end of the cell opening tag. Inside the *c* element, we look for the `<v>` opening tag that contains the cell value. We parse the value until we encounter the closing tag `</v>` and insert it into our intermediate data structure.

5.1. Consecutive parsing

We optimize *consecutive* parsing for runtime performance. As shown in Fig. 8, *consecutive* parsing first decompresses the complete document into memory and then parses the content with multiple parallel parsing threads. Having the complete document in memory during parsing has several advantages. First, we do not need to use intermediate buffers to store values for later parsing, as the document itself serves as a buffer. This reduces costly memory operations such as allocations and copies. Additionally, since decompression is independent of parsing, the choice of decompression method is flexible, and we can use libraries that are optimized for full-buffer decompression. However, keeping the entire document in memory during parsing leads to inflated memory usage. In cases where the document cannot fit in memory, SheetReader uses *interleaved* parsing instead.

Once the entire document has been decompressed, we can parallelize the parsing by simply splitting the document into roughly equal-sized chunks and processing each chunk by a separate thread. However, splitting XML documents into multiple chunks that can be parsed in parallel is a challenging problem [21–23]. In particular, a parser that starts at an arbitrary point in the document lacks the context to determine how to process the encountered characters. To overcome this problem, we leverage the fact that spreadsheets have a predefined XML structure and determine the parsing state by identifying the type of the first XML element that we encounter in the chunk. Specifically, we scan the chunk for structural characters that denote the start or end of an XML tag (e.g., <). We then build additional context by determining the type of the corresponding XML element. For example, if we encounter the opening tag to a row element, we know that we are at the beginning of a new row, while if we encounter the closing tag to a cell element, we know that afterwards there is either another cell or the end of the row. The above approach is possible because the encoding of structural characters is different when these characters are not structurally significant, e.g., when they are part of an element or an attribute value. For example, while < denotes a structural character, the same character is encoded as `&lt;` inside an element or an attribute value.

Our parallel parsing approach also assumes that each cell has information about its location (i.e., the row and column number), so that individual threads can determine where to insert the parsed values in SheetReader's intermediate data structure. Although this information is optional according to the standard, the most widely used tool, Microsoft Excel, provides it. If there is no location information, we can employ an additional processing step, either before or after the parallel parsing. Before parsing, we can perform a reduced sequential scan over the document to count the rows and cells and calculate the offset for each chunk. This sequential scan can be implemented efficiently such that it does not significantly affect the runtime. An alternative approach is to let each thread insert the parsed values into its own separate intermediate data structure. In the last step, i.e., when converting to the target, we can then merge the partial data structures by sequentially retrieving the values.

Additionally, we determine the size of the worksheet, i.e., the number of rows and columns, from the dimension element in the spreadsheet document metadata. If the dimension element does not exist, and since we have the entire uncompressed document in memory, we can also determine the size by examining the row and column number of the last cell. Predetermining the worksheet size allows to pre-allocate the intermediate data structure and avoid costly resize operations. Furthermore, it enables multiple threads to insert values in the data structure without any write synchronization mechanism. Being unable to pre-allocate the intermediate data structure adds only minor complexity. When a column becomes full, we simply need to allocate a larger amount of space and copy over the existing values. During the resizing operation, we also need a synchronization mechanism (e.g., a lock) that blocks the insertion of new values.

Overall, each parsing thread of the *consecutive* approach takes as input the starting offset of its chunk and the end offset or the total chunk length. Then, it locates the first cell in the chunk as discussed previously and proceeds with parsing from there, skipping over all content before the first cell. This skipped content is actually relevant to the last cell of the previous chunk. Therefore, to ensure that all elements will be parsed, each thread finishes parsing the last cell of the chunk by extending its assigned parsing area over the beginning of the following chunk.

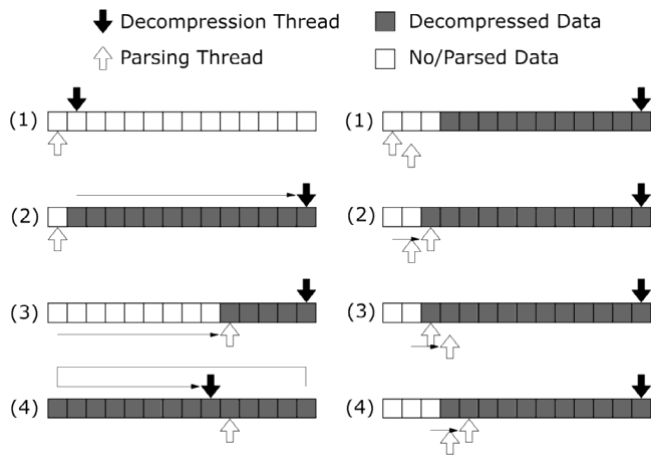


Fig. 10. Concurrency control.

5.2. Interleaved parsing

This approach aims to minimize memory usage. To that end, it continuously recycles a constant amount of memory so that the memory usage is independent of the input document, and interleaves decompression and parsing. Specifically, as depicted in Fig. 9, decompression and parsing occur repeatedly one after the other. First, the decompression stage decompresses part of the document. Then, the parsing stage processes this part and returns the control flow to the decompression stage, waiting for the next part to be decompressed. As a result, it is impossible to access arbitrary parts of the document at any time and the parser is unable to backtrack or look ahead very far in the document. This means that the parser needs to process any relevant content as soon as it encounters it, or store it immediately for later processing. Consequently, *interleaved* parsing imposes more restrictions on the employed decompression and parsing techniques compared to *consecutive* parsing.

To implement a single-threaded version of the interleaved approach, we only need a single-element shared memory buffer. The decompression stage fills the buffer with decompressed content, and the parsing stage parses it. However, to enable parallelization, we need a buffer with at least two elements. Using multiple elements allows to decouple the decompression and parsing stages and execute them in parallel by separate threads. The decompression thread writes the elements that are available for writing, while the parsing thread reads from the written elements and subsequently re-enables them for writing. Using a two element buffer, the threads can switch their elements only when they both finish processing. Since the decompression and parsing time are data-dependent, the time that a thread has to wait for the other thread to finish can fluctuate significantly. To reduce the total wait time and mitigate the resulting unpredictable runtime, our microbenchmarks showed that it is better to use larger buffers.

Fig. 10 (left) shows how the *interleaved* approach works with a *circular buffer* using a single parsing thread. The decompression and parsing threads iterate through the elements sequentially, i.e., the decompression thread writes its results into the first element, then the second one, and so on (step 2), while the parsing thread reads the elements in the same order (step 3). When a thread reaches the last element, it loops back to the first one (step 4).

To prevent the threads from using an invalid element, i.e., the decompression thread overwriting an element that is not parsed yet or the parsing thread parsing an element that is not written yet, we make the buffer *thread-safe*. Specifically, we use an index that indicates the element that each thread is currently operating on, and ensure that the parsing thread remains at least a single element behind the decompression thread. That is, if the decompression thread is currently writing into the element with index x, the parsing thread can process all elements with up to and excluding index x. If the parsing thread reaches this point, it simply blocks until the decompression index advances. For simplicity, the initial state satisfies this requirement by starting the decompression thread one element ahead of the parsing thread (step 1). To ensure that the parsing thread processes the last element, we increment the write index by one when the decompression thread finishes the current round. The decompression thread determines if it is allowed to write to an element by simply checking if that element is currently being parsed by the parsing thread. This can happen only when the decompression thread has filled all available elements and the parsing thread has not freed any element yet, as shown in step 4. We store the indexes as atomic integers, so that all threads see the same value when they access an index simultaneously.

In addition to parallelizing decompression and parsing, we also parallelize the parsing stage, i.e., use multiple parsing threads as shown in Fig. 10 (right). We explore this avenue since our preliminary benchmarks showed that decompression is typically faster than parsing. The *interleaved* approach can be easily extended to support parallelism in the parsing stage. Contrary to *consecutive* parsing, the parsing threads do not work with a large buffer containing the entire document, but with small buffer elements that contain only small parts of the document. As a result, the mechanism for distributing the elements among the parsing threads is slightly more complex. One solution would be to introduce a flag for each element that indicates its state, i.e., if the element is ready to be parsed, ready to be written, or currently being processed. The threads would then pick an element to process based on these flags. Instead, we decided to extend our existing index-based synchronization mechanism. Specifically, each individual parsing thread has a separate index and checks up to which element it is allowed to parse. The decompression thread simply checks if any of the parsing threads works on the element where it wants to move next.

One remaining issue is preventing the parsing threads from parsing an element multiple times, i.e., uniquely assigning elements to parsing threads. This is achieved by initializing their indexes in a staggered manner and advancing them by the number of parsing threads rather than singular increments as shown in Fig. 10 (right). For example, with three parsing threads, the first one starts at index 0, advancing to 3 and then 6. The second thread starts at index 1 and advances to 4 and then 7. The third one starts at index 2 and advances to 5 and then 8. This approach guarantees that all elements are fully processed exactly once.

Since we process the elements in sequential order, we also process the document sequentially, which enables using the parsing ''extension'' mechanism described for the *consecutive* approach. If a parsing thread reaches the end of its assigned element but the last cell has not been fully parsed yet, it simply advances into the next element to finish parsing. Afterwards, it readjusts its index to prevent overlap with the other threads. This is possible because the decompression thread only writes to elements up to the last parsing thread, i.e., every element in front of a parsing thread up to the decompression thread will always be valid.

Similarly to the *consecutive* approach, we exploit the predefined XML structure to deduct the parse states. However, dealing with the lack of location information is harder because the parsing positions are constantly changing. Each time a parsing thread advances, it skips over potentially multiple elements that contain the logical continuation of its acquired parse state. As a result, the

parsing threads are repeatedly placed in unknown and ambiguous parse states. We can adapt both solutions that we discussed for the *consecutive* approach here. Before the actual parsing, each parsing thread could perform a fast reduced scan over its assigned element to count the number of contained rows and cells, accounting also for the location information after blank cells. Then, all threads would need to share their results to determine the row and cell numbers at the beginning of all elements. Afterwards, the parsing threads would proceed with the actual processing. Alternatively, we could create an intermediate data structure for each element rather than for each thread. Finally, if we are unable to pre-allocate the intermediate data structure, we can apply the same solution of simply synchronizing the write and resize operations as in the *consecutive* approach.

6. Handling missing dimension and location information

Our previously presented parsing approaches assume existing dimension and location information, which is commonly present in the Excel format (see Section 2). In particular, we rely on dimension information to pre-allocate our intermediate data structure, and on location information to identify the location of the individual parsed cells across multiple threads without complex bookkeeping. Although Excel-generated files contain dimension and location information, this is optional. Thus, there is no guarantee that this information will be present in the output of other applications. The dimension information is generally not necessary and serves only as an optimization technique for pre-allocating the sheet data structure since without it one could simply use a container that dynamically grows as cells are parsed. While the standard does not explicitly specify how to proceed when cell location information is missing, Excel places these cells directly one after another, i.e. a newly parsed cell is placed in the next column after the previously parsed cell, and if it is the first cell in a row, it is placed in the first column. A complication is that NULL (blank) cells can be omitted from the XML, which means that all cells after a blank cell would be erroneously offset by 1 column. Therefore, cell location information is required to indicate the correct position of a cell (correction values) when one or more previous cells were NULL. To handle spreadsheets stored in this manner, we propose to adapt our interleaved approach, i.e., the intermediate data structure and how we populate it.

Instead of pre-allocating the intermediate data structure in a column-major fashion based on the information provided by the dimension elements, we allocate it on-the-fly in a row-major fashion. This way, we can still chunk the document and assign different threads to different chunks. For each thread, we maintain two lists: One list that stores the actual cell values and their corresponding types per chunk, and one list for bookkeeping, i.e. for storing location information. The bookkeeping list contains a chunk id, a cell index that corresponds to the index of a particular cell within a chunk, and the row and column identifiers. Each thread populates these two lists for their currently processed chunk. We add values to the bookkeeping list only when correction values are specified by the spreadsheet and they do not match the expected location.

Once parsing completes, we end up with two lists per thread, one containing the chunks with the actual cell values and one for the correction values. When converting the intermediate data structure into our target, e.g. R or Python dataframes, we process the chunks sequentially and place the values accordingly into the target data structure. While iterating over the parsed chunks, we keep track of the processed cells and perform lookups in the bookkeeping list. For example, the first cell corresponds to the first row and first column of the target structure, the second to the first row and second column, and so forth. If our lookup

```
<row r="2">
    <c r="B2" t="n" s="1">
        <v>43556.2</v>
    </c>
    <c r="D2" t="s">
        <v>4</v>
    </c>
</row>
```

Fig. 11. Excel worksheet XML extract. The highlighted sections are scanned for the opening tag of a cell element.

returns a correction value for a particular cell, we overwrite the current location. For example, after processing five cells, we would place the next cell in the sixth column. Now, for this particular cell, our lookup in the bookkeeping list returns the location of the first row and seventh column. Therefore, we place the current cell into the seventh column and leave the sixth column empty, indicating a NULL/blank value. We apply this method similarly to row correction values, where a special value indicates to just increment the row and move to the first column, instead of overwriting with a particular value.

The bookkeeping list can also help determining the number of rows when dimension information is missing. Therefore, we need to iterate through the bookkeeping list until we find an exact value, while keeping track of the encountered row increments.

7. Optimizations for spreadsheets

Aside from parallelization, we employ some further spreadsheet-specific optimizations to accelerate parsing, thereby further reducing the runtime. These optimizations aim to reduce the amount of work per input character. Ideally, when a character does not provide any relevant information, we do not want to perform any work for it. Additionally, we do not want to visit any given character, including potential copies of it, more than once. Our first optimization consists of parsing element names on-the-fly rather than copying characters into a new buffer and comparing against the complete string. We achieve this by checking if the scanned input characters match any of the predefined known element names. For example, for row elements, we add an integer field to the parsing state that checks if the current element name is `row`. At the start of parsing, we initialize this field to 0. If the parser is in the appropriate state and encounters an `r` character, we increment the field. If we encounter an `o` right after, we increment the field once more. We apply the same procedure for the `w` character. If at any point we encounter a different character than expected, the field is reset to 0. Upon encountering a whitespace character, we simply determine the currently parsed element by checking the integer fields of the relevant element names. If the field matches the length of the checked element name (e.g., 9 for `sheetData`, 3 for `row`, 1 for `c`), this means that we just encountered the corresponding element.

Our second optimization consists of skipping as much unneeded content as possible while also determining when to skip as quickly as possible. In other words, we aim to determine as early as possible the amount of required work for a character and then only perform this required work. We can identify opportunities to apply this optimization by examining the XML schema that is given by the specification. Using the Excel format as an example, we need to check for the opening tag of a cell element (`c`) only when we have encountered a row (`row`) opening tag

previously and have not encountered a row closing tag since then (cf. Fig. 11). This also applies for values (v) inside cells (c), rows (row) inside the sheet data (sheetData), and even for locating the sheet data element itself.

Furthermore, we avoid parsing and deserializing attributes that do not contain relevant data or metadata for creating the target data structure. For example, all row elements in Excel worksheets have an attribute that indicates the height of the row. Such irrelevant tags and their values should be skipped as early as possible. Given the XML format, we achieve this by skipping all content between the opening and closing quotation marks of the irrelevant attribute value. We note that we assume that the input document is a valid XML conforming to the specification. Otherwise, if the XML contains invalid values, e.g., if a quotation mark is missing, the parser might skip some relevant data.

To avoid visiting characters more than once, we try to perform parsing in-situ without any intermediate copies. This is particularly relevant for the *interleaved* parsing approach, where there are no guarantees regarding which part of the document is currently available in memory. For example, the value of the row number attribute might be split between buffer elements. Since it is impossible to access the first element once we advance to the second one, a naive solution would copy the relevant portion from each element into another intermediate buffer, so that the two parts can be combined and the complete value can be deserialized. We avoid such copies, and thus processing the same character twice, by deserializing characters as they arrive.

Deserializing integers in-situ is simple. We first initialize the integer value to 0 and then for every read character, we multiply the current value by 10 and add to it the deserialized character. We can use this approach to deserialize most of the required attribute and element values from the worksheet. We can, for example, use a virtually identical mechanism for spreadsheet form numbers where ''A'' corresponds to 1 and ''AA'' to 27. The only difference is that we need to multiply by 26 and adjust the deserialization of the characters to numbers. Other values such as booleans or cell types are also trivial to deserialize without copying. However, we cannot apply the above technique to deserialize floating point values in-situ, as it can potentially introduce rounding errors and thus produce erroneous results. Thus, for floating point values, we cannot avoid copy buffers.

Overall, our spreadsheet-specific optimizations improve the performance of the low-level parsing routine. As a result, in single-threaded execution, the optimizations directly translate into lower runtime. In the case of multiple threads, the optimizations accelerate the execution of each individual thread. Consequently, we can use fewer threads, thereby potentially reducing synchronization overheads.

8. Evaluation

In this section, we first describe the experimental setup and methodology and then present a thorough evaluation of *SheetReader* in terms of runtime and memory usage. To demonstrate *SheetReader*'s benefits, we first compare it with existing state-of-the-art solutions for spreadsheet parsing and then perform an in-depth analysis to study the trade-offs between our proposed parsing approaches. Lastly, we evaluate parallel decompression to determine its impact on the runtime.

8.1. Experimental setup & methodology

Hardware. The experiments were performed on a machine equipped with an AMD EPYC 7702P 64-Core CPU, 512 GB RAM, and a 512 GB SSD, running Ubuntu 20.04 (kernel version 5.4.0–90).

Benchmarks. We use various benchmarks to measure the runtime and memory usage of our approach and the competing ones. Following our prototype, the benchmarks involve loading an Excel spreadsheet file into R. As our prototype targets the xlsx format introduced in 2007, it is impossible to execute benchmarks designed for older format versions. We run every benchmark on a new R (or Python) instance to avoid potential residual objects in memory from influencing later measurements. While the instance is running, we periodically measure its memory usage. For the general comparison between the approaches, we use the maximum measured memory usage and the total runtime. Additionally, we insert logging messages that indicate the beginning and end of individual loading stages. This periodic data allows us to examine individual benchmarks in detail. That is, we associate the separate steps of each approach with particular messages and determine the impact of each step on the overall memory usage. We repeat each benchmark 5 times and report the average. We assume cold system caches, i.e., we clear OS caches before re-executing each benchmark. Unless stated otherwise, the experiments are run using our prototype in R.

Datasets. Most of our benchmarks use synthetically generated Excel spreadsheets according to specific desired parameters such as the percentage of numeric vs. text values or the percentage of blank cells. We generate Excel spreadsheet files for various row counts where larger spreadsheets are supersets of smaller ones. The compressed sizes range from 13.6 MB (10,000 rows), to 413 MB (300,000 rows), up to 827 MB (600,000 rows). Unless otherwise specified, each spreadsheet has 100 columns and contains only numeric values, without any blank cells. Furthermore, we use two real-world financial spreadsheet files from *AccessHolding* to study the performance of our parser in comparison with the state-of-the-art in a real setting. For data protection reasons, we anonymized the files before running our benchmarks. The first file, *loans*, has 280,973 rows, 110 columns, and a compressed size of 172 MB. The second file, *transactions*, has 447,241 rows, 84 columns, and a compressed size of 193 MB. Both files contain a mix of different data types, i.e., integers, dates, floats, booleans, and text. While the first file has only a few empty cells, the second one has significantly more (i.e. 20 columns are almost empty).

Baselines. We chose to implement our prototype in R and Python because of their popularity among data scientists. Hence, we experimentally compare *SheetReader* with existing R and Python packages for loading spreadsheets. After analyzing several packages for Excel parsing in R, we chose to include the openxlsx and readxl packages[3] as they showed the best performance for lowest memory usage and fastest runtime, respectively. Both packages work solely with Excel files and are written in C++. Openxlsx employs a hybrid DOM/SAX approach, and extracts cell values using regular expressions, while readxl first constructs a DOM tree from the spreadsheet XML using the XML DOM parsing library RapidXML and then processes the tree further to extract the cell values. For our experiments in Python, we use the openpyxl package, as it is the only method for loading spreadsheets that supports xlsx in the popular Pandas library.

Software configuration. We use the following versions of R, R packages, and libraries: R 4.0.3, data.table 1.13.2, openxlsx 4.2.3, readxl 1.3.1, miniz 2.1.0, libdeflate 1.7, and the following versions of Python and packages: Python 3.7, openpyxl 3.0.10. We implemented our prototype in C++ and compiled with gcc/g++ 9.3.0. By default, decompression uses 1 thread and parsing uses 8 and 2 threads for the *consecutive* and the *interleaved* approach, respectively. If applicable, shared strings are parsed in parallel using one additional thread. In the *consecutive* approach, we determine the buffer size for the decompressed

[3] https://github.com/ycphs/openxlsx, https://readxl.tidyverse.org/.

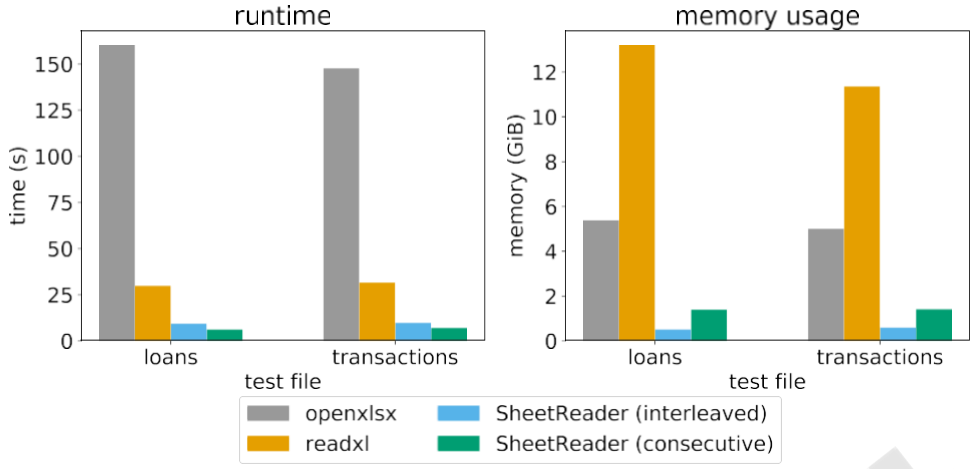


Fig. 12. Performance overview of *SheetReader* and existing R packages for parsing real-world spreadsheets.

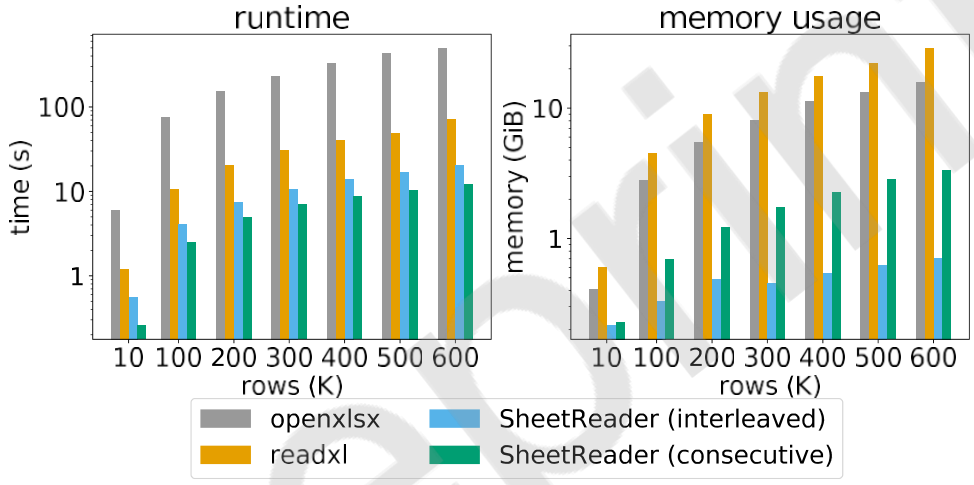


Fig. 13. Performance overview of *SheetReader* and existing R packages for parsing synthetic spreadsheets containing only numeric values.

content from the ZIP metadata. In the *interleaved* approach, we allocate a buffer with 1024 32 KB-sized elements after empirically evaluating several configurations.

8.2. Comparative analysis

Comparison using real data. Fig. 12 compares *SheetReader's* parsing approach (both *consecutive* and *interleaved*) with the state-of-the-art in terms of runtime and memory usage on the two real-world datasets. Furthermore, we parse the strings parallel to the worksheet in a single thread that performs both decompression and parsing. Using the interleaved approach, *SheetReader* is 3.2× faster than readxl, the fastest existing solution, while also consuming 26× and 20× less memory for loans and transactions, respectively. Compared to the most memory-efficient existing solution, openxlsx, *SheetReader* (interleaved) is 17× faster with 10.7× less memory consumption for the loans file and 15× faster with 8.6× less memory consumption for the transactions file. Overall, our results demonstrate that *SheetReader* provides runtime and memory-efficient spreadsheet parsing.

Scalability with spreadsheet size. Fig. 13 shows the runtime and memory usage as we increase the size of our synthetically generated spreadsheets. Comparing the runtime performance, openxlsx exhibits very long runtimes even for moderately sized files, taking more than 2 min for a spreadsheet with 200,000 rows. Readxl, which is the fastest existing solution for loading spreadsheets into R, reaches 65 s for the largest file. Our approach, *SheetReader*, outperforms both baselines by around 2.5 to 3 times across all tested worksheet sizes.

In terms of memory efficiency, *SheetReader* has a considerable lead over the other packages, consuming at most 728 MB for the largest file of 600,000 rows. Specifically, *SheetReader* consumes up to 40× and 20× less memory than readxl (29.5 GB) and openxlsx (16.3 GB), respectively.

The excessive memory usage of readxl is caused by its underlying XML DOM parsing approach. The generated DOM tree that is kept in memory for subsequent processing consumes large amounts of memory. As a result, the memory usage of readxl is consistently over 10 times more than the size of the uncompressed source worksheet, reaching almost 30 GB for 600,000 rows. For consumer machines, even worksheets with 200,000 or 300,000 rows can saturate all available memory (9 GB and 13.5 GB respectively in this benchmark), which would in turn also impact the runtime. That is, the runtime measurements will become significantly worse than the ones shown here if we use a machine that does not have a sufficient amount of memory.

The package openxlsx employs an approach that can be considered a mix between DOM and SAX parsing. Instead of extracting the whole document into a DOM tree, it extracts only the significant parts of the document into lists using regular expressions. However, it does not directly process the extracted values. Specifically, while the extraction of values from the worksheet is done in C++, the lists containing the values are returned

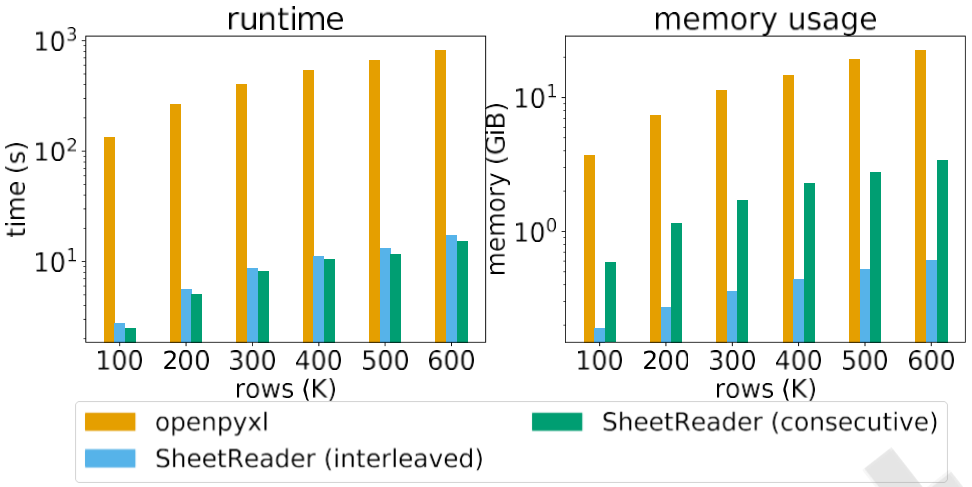


Fig. 14. Performance overview of *SheetReader* in Python compared to openpyxl.

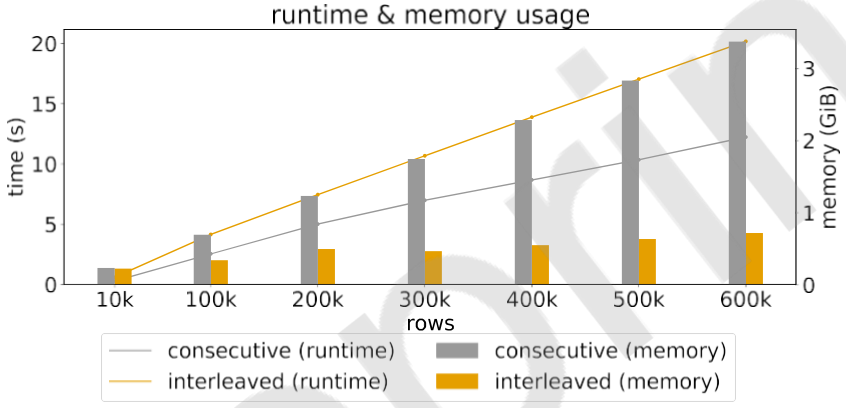


Fig. 15. Benchmarks of the *Consecutive* and the *Interleaved* parsing approaches.

to R. Then, the R wrapper function that wraps the extraction processes these values further to build the target Dataframe. Overall, while openxlsx uses considerably less memory than readxl, its memory usage is still excessive, i.e., around 8 GB for 300,000 rows and reaching 16 GB for 600,000 rows.

We repeat the same experiments with our prototype in Python. Fig. 14 shows the results (note the log scale on the y-axis). SheetReader is up to 47× faster in terms of runtime performance and consumes up to ∼ 40× less memory (for 600,000 rows, interleaved method). The performance improvement can be explained by the fact that openpyxl is internally relying on the expat library, a general XML parsing library using the SAX approach, and thus introduces the previously discussed performance overheads of generalized XML parsing.

8.3. SheetReader analysis

Parsing approaches comparison. We introduced two different parsing approaches for *SheetReader*; *consecutive* and *interleaved*. This benchmark studies the trade-offs between their runtime and memory usage. Particularly, it aims to determine the speedup of the *consecutive* over the *interleaved* approach, and to show how the *consecutive* approach achieves this speedup at the expense of an increased memory usage.

Fig. 15 shows the results when applying both approaches to the same synthetic spreadsheets and increasing the spreadsheet size. Both approaches exhibit a linear increase in runtime and memory usage that is proportional to the size, with the *consecutive* approach consistently having a better runtime but also substantially higher memory usage. In contrast, the increase in memory usage of the *interleaved* approach is negligible.

In the *consecutive* approach, the decompression step requires two buffers, one for the compressed and one for the decompressed content. Therefore, the maximum memory usage is effectively determined by the sum of the sizes of the compressed and the decompressed worksheet. The intermediate data structure is allocated only after the deallocation of the compressed document (i.e., after decompression), while it is generally considerably smaller than the worksheet. As such, it has no impact on the maximum memory usage. In contrast, in the *interleaved* approach, since the actual parsing process uses a constant amount of memory, any increase of the memory usage over different worksheet sizes is caused by the intermediate data structure, whose size depends on the input worksheet.

Furthermore, the benchmark shows that while the runtime rises linearly for both approaches, the increase for the *interleaved* approach is stronger than for the *consecutive* one, culminating in a difference of around 8 s for 600,000 rows.

Our benchmark confirms the advantages and disadvantages of both parsing approaches discussed in Section 4. Additionally, based on the experimental results, we propose using the *interleaved* approach as the ''safe default'' option because it loads the spreadsheet data in an acceptable amount of time while only rarely consuming more memory than the one that is already required by the target environment to store the same data. Users

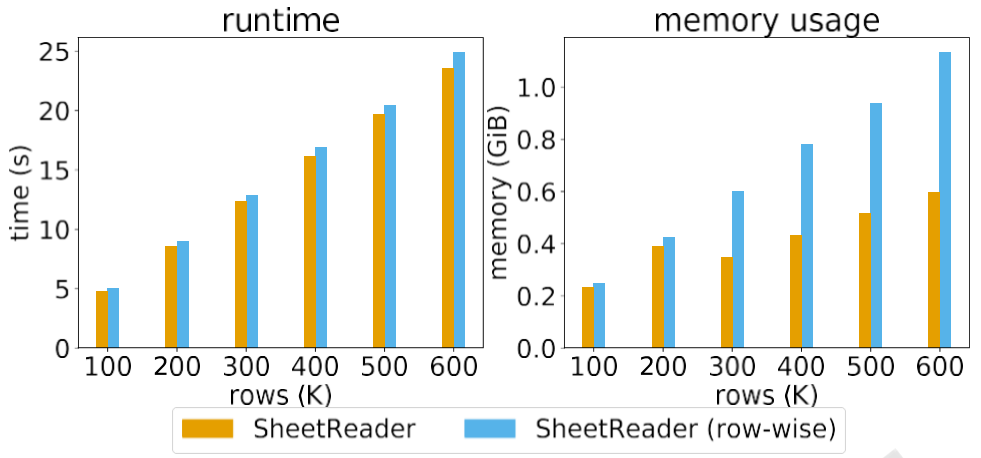


Fig. 16. Benchmarks for the *interleaved* approach adaption that handles missing dimension and location information.

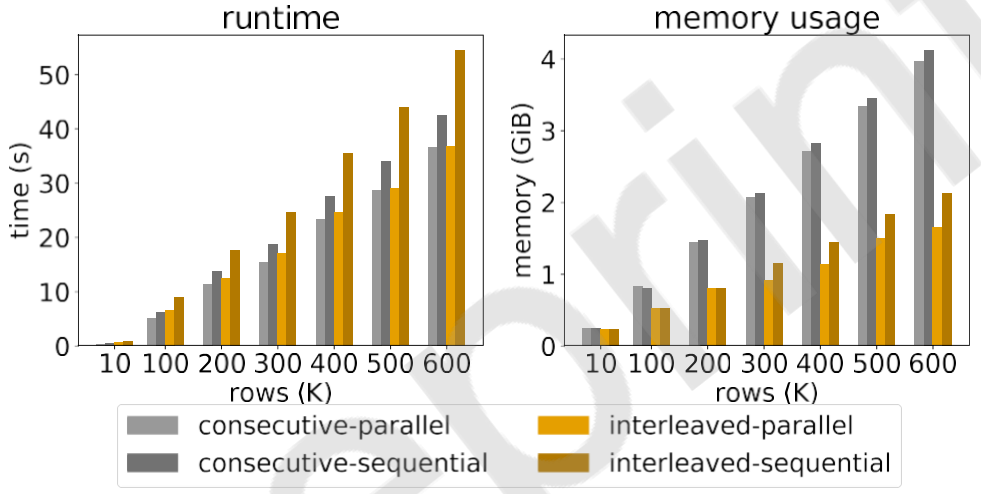


Fig. 17. Benchmarks of parsing the shared strings sequentially or parallel to the worksheet.

can choose the *consecutive* approach if they require faster loading times and have a machine with a sufficient amount of memory.

Missing dimension and location information. As discussed in Section 6, to deal with missing dimension and location information we need to adapt our parsing method. To evaluate the introduced overhead, we benchmark the two approaches using the numeric dataset. Fig. 16 shows that the adapted interleaved approach, indicated by *SheetReader (row-wise)* is on par with the default *interleaved* approach, with regard to runtime performance. However, with regard to memory, we observe that SheetReader (row-wise) performs up to 2× worse, occupying approximately more than 1 GB of memory. This is because during the construction of the target (R dataframe), we cannot free our columns in the intermediate data structure after their transformation to the target, as we need to iterate row by row through the row-based intermediate data structure.

Parallelizing worksheet and shared strings parsing. Apart from the parsing approach choice, in the case of spreadsheet systems that store strings separately from the worksheet, e.g. Excel, we can also choose between parsing sequentially or in parallel. To compare the performance of these two approaches, we generate synthetic spreadsheets for various row counts that contain a mix of different data types. Specifically, the synthetically generated mixed-type spreadsheets have 40 columns of floating point values, 30 columns of integer values, 20 columns of text with 25% unique values, and 10 columns of text with 75% unique values.

As expected, Fig. 17 shows that parsing the shared strings and the worksheet in parallel yields runtime improvements. The *interleaved* parsing approach benefits the most from this parallelization, reaching a runtime reduction of around 30%. However, contrary to our expectations, the parallel approach has a lower memory usage than the sequential one for almost all benchmarks. To determine the cause of this, next we examine the memory characteristics of the benchmarks in more detail.

Memory usage analysis. Fig. 18 shows a detailed memory profile of the sequential and parallel approaches when parsing the largest document (600,000 rows) from our previous experiment using the *consecutive* approach. To identify when and where the maximum memory usage occurred, we measure it periodically and associate different time spans with different steps in the parsing process. The green decompress and red parsing sections correspond to worksheet parsing, while the yellow shared_strings section combines both steps in the shared strings parsing process. In the parallel benchmark, the yellow section depicts the extra time taken by shared strings parsing. While worksheet parsing finishes at around 15 s, shared strings parsing takes over 10 extra seconds to finish, delaying the dataframe construction.

Both benchmarks show that parsing the shared strings table is two to three times slower than parsing the worksheet. The runtime difference can be explained by the inability to parallelize the parsing process for the shared strings table, while the worksheet is distributed among 8 threads. Furthermore, the memory

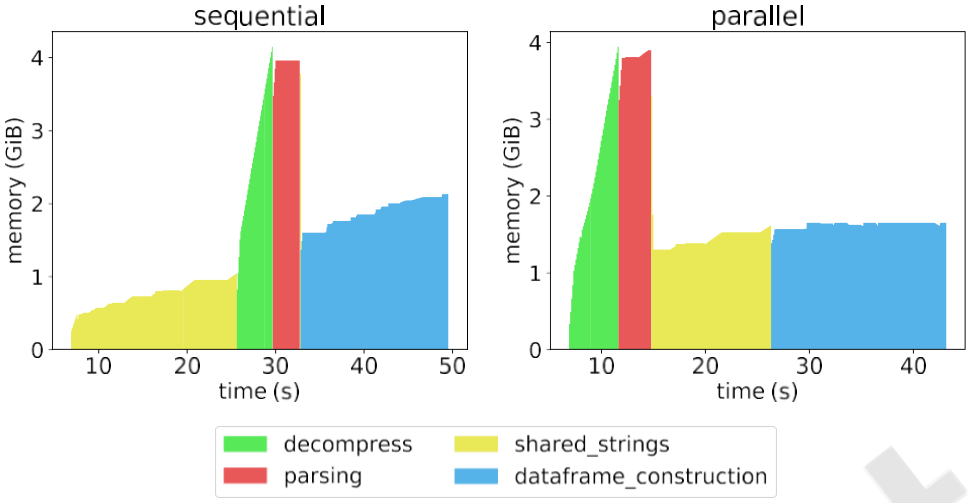


Fig. 18. Memory measurements of the sequential and the parallel approach (*consecutive* parsing). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

usage increases steadily during the parsing of the shared strings table. This increase in memory usage stems from allocating space to copy the strings out of the original document, so that we can return them to the user after the deallocation of the document.

The reason why parsing the worksheet and shared strings table in parallel has a lower memory usage than doing so sequentially is a combination of three factors: the dynamic string allocations, the long runtime of shared strings parsing compared to worksheet parsing, and the order of the two parsing steps in the sequential approach. The sequential approach processes all shared strings and allocates space for them before decompressing the worksheet, which represents a constant base memory usage for all subsequent steps, including any processing of the worksheet where shared strings are not required. As a result, the maximum memory usage is reached when the worksheet is decompressed, since the copied strings occupy additional memory on top of the decompressed content. In the parallel approach, since the shared strings parsing step is slow, by the time all strings are copied, the worksheet is fully parsed and the source document has been deallocated.

We conclude that for the sequential approach, the parsing of the shared strings table should occur after the worksheet parsing to reduce the maximum memory usage. Parsing the strings after the worksheet has the additional benefit of allowing to filter out unneeded strings, i.e., strings that do not occur in the specified sheet. Swapping the order of the parsing steps in our prototype is straightforward, as these steps are independent.

Impact of thread count. To evaluate the effectiveness of our parallelization efforts, we measure the impact of the number of used threads on the runtime for both the *consecutive* and the *interleaved* approach. Fig. 19 shows that the benefits decrease as we increase the thread count in both parsing approaches. Particularly for the *interleaved* approach, any noticeable runtime improvement (5 to 10%) stops at only two parsing threads, while the runtime actually increases with more than two threads. Further analysis when running the benchmarks reveals that the decompression thread becomes the limiting factor at this point, so that any additional parsing threads only introduce more synchronization overhead. Regardless of the number of parsing threads, the decompression is too slow and results in idle threads waiting for a new available buffer element. Thus, the only way to further reduce the runtime is accelerating the decompression.

The *consecutive* approach exhibits a more gradual runtime reduction, with the increase from 1 to 8 threads reducing the runtime by almost half (20 to 12 s for 600,000 rows), while the increase from 8 to 16 threads only has a marginal impact (12 to 10.5 s). We are again effectively limited by the speed of the decompression step. Since this approach performs parsing after the completion of the decompression, the slow decompression component imposes the lower limit for the runtime.

8.4. Parallel decompression

Since decompression is a runtime bottleneck, we performed an experiment to determine the advantage that we can get from parallelizing it. To that end, and since the current compression used by the OOXML and ODF formats does not support parallel decompression, we extracted the worksheet XML files from the Excel files and re-compressed them with a modified Deflate algorithm based on the MiGz library.[4] Furthermore, we established boundaries in the Deflate stream after which there are no back-references to previous blocks and stored the offsets of these boundaries in the file metadata. The result is a valid Deflate stream that can be decompressed with any existing library. A decompression algorithm can now start full decompression of the stream from any of the boundaries without requiring to first fully decompress the previous blocks in the stream. Therefore, we can parallelize the decompression of a single document. Specifically, in our implementation we assign separate threads to equally spaced boundaries. Each thread performs decompression and parsing in an interleaved manner (i.e., using our *interleaved* approach) until it reaches the next boundary.

Fig. 20 compares the *consecutive* approach without parallel decompression with the *interleaved* approach that parallelizes the decompression using our MiGz-derived algorithm when increasing the thread count. We see that the parallel decompression implementation outperforms the *consecutive* approach when using more than 2 threads, especially for larger files. In most cases, using only 4 threads, the parallel decompression implementation achieves the same runtime as the consecutive approach with 16 threads. In turn, 16 threads enable the MiGz-derived algorithm to lower the runtime by an additional 35%. Furthermore, increasing the number of threads has a larger effect on the runtime for the fully parallel implementation. Finally, we note that since the individual threads employ the *interleaved* parsing approach, the memory usage is significantly lower than the one of the *consecutive* approach. Therefore, parallel decompression allows us to further reduce the runtime while retaining low memory usage.

[4] https://github.com/linkedin/migz.

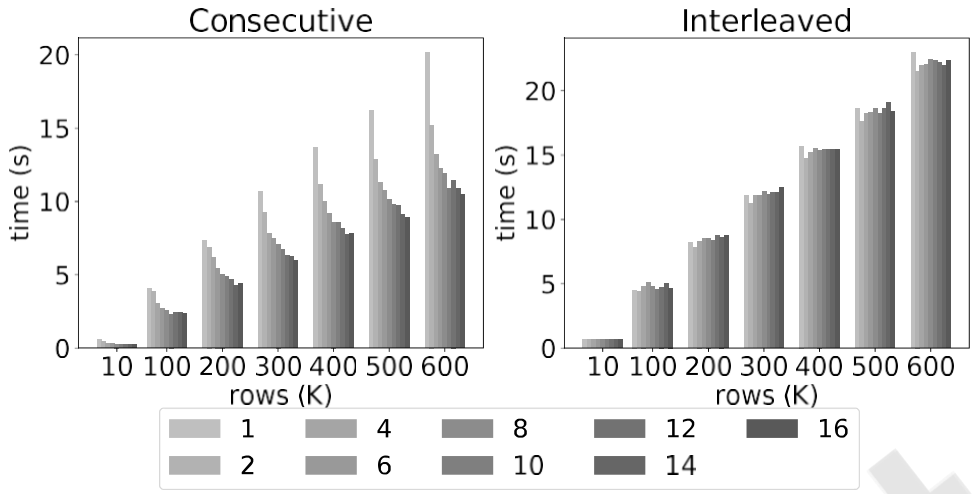


Fig. 19. Impact of the number of threads.

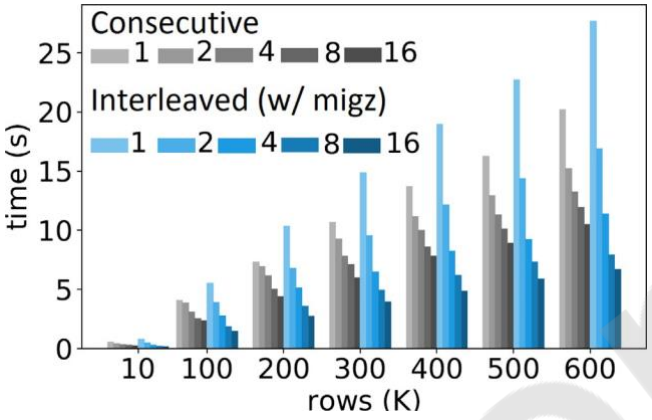


Fig. 20. Impact of parallel decompression.

8.5. Summary

Our experimental comparison with the existing solutions shows the efficiency of our proposed spreasheet parsing architecture. Overall, *SheetReader* with *interleaved* parsing loads spreadsheet files 2× to 3× faster than the fastest existing solution while consuming up to 20× less memory than the most memory-efficient existing solution. As such, our parser can process large spreadsheets on current consumer machines without requiring an excessive amount of resources and degrading the user experience.

SheetReader offers an alternative consecutive parsing approach which reduces the runtime by an additional 40% but also increases the memory usage by more than 4 times. Furthermore, parsing the shared strings and the worksheet in parallel reduces the runtime by 15% and 30% when using consecutive and interleaved parsing, respectively. Parallelizing the parsing process itself grants the *consecutive* parsing approach a 20% to 30% runtime reduction when using 4 threads, while having only negligible impact on the *interleaved* approach. Finally, we show that we could achieve significant additional performance improvements by parallelizing the decompression, which is unfortunately not supported by the current specification of OOXML and ODF.

9. Related work

While there is some work on extracting specific content from spreadsheets, e.g., tables [24], there is no related work directly focusing on spreadsheet parsing; current approaches rely on generalized XML parsing. Therefore, we review techniques proposed for efficient parsing of XML and other text-based formats.

XML parser parallelization. Parallelizing XML parsing is a non-trivial task [25]. As the XML format is self-describing, the difficulty lies in splitting an XML document into chunks that can be parsed in parallel. One line of work proposes a two-pass approach to build an XML skeleton structure, which allows to split the document before parsing and merge the individual results efficiently [21,22]. Follow-up work proposes to also parallelize the first pass by letting multiple threads create multiple skeletons for each chunk, and then merging them into one [23]. Another line of work shows that producing chunks with an arbitrary number of start and end XML tags and then merging partial results, offers better scalability than the two-step approach on multicore systems [26]. Furthermore, leveraging SIMD (single-instruction multiple-data) instructions of modern CPUs, allows to parallelize character scanning and to avoid cache misses, conditional branches and branch mispredictions, thereby further minimizing the parsing runtime [27]. Both lines of work are complementary to SheetReader, as they can be employed to better split the XML and parallelize character scanning at a lower level.

XML parser compilers. An approach to accelerate the parsing procedure is to specialize the parser to a given schema. Parser compilers generate parsers based on predefined schemata. XML Screamer [15] compiles specialized parsers that merge parsing and deserialization to avoid expensive data copying and transformation operations. Chiu and Lu [14] propose an intermediate representation with a generalized automata approach, through which they generate efficient parsers. Although schema-based specialization leads to better performance, it does not directly exploit spreadsheet-specific properties to further optimize parsing.

Parsing text-based formats. Parsing widespread text-based formats, e.g., CSV and JSON, is similarly challenging as parsing XML. There has been extensive work on improving the performance of CSV parsing, e.g., by employing speculative parsing techniques [28], by optimizing the parsing process for multicore CPUs [29], and by employing GPUs for parallelization [30,31]. In-situ data processing approaches also employ several optimizations, such as selective parsing and just-in-time compilation [32]. Furthermore, multi-hypothesis CSV parsing addresses the challenge of validating files with unknown schemata [33].

The JSON format shares more similarities with the XML format, as they are both self-describing. Several approaches have been proposed to improve the performance of JSON parsing. For example, Sparser [34] employs raw filtering through SIMD

instructions before parsing, while Mison [35] speculatively predicts the physical location of necessary fields through structural indices. Moreover, simdjson [36] proposes to limit the set of employed instructions to increase the parsing and validation performance of JSON documents on commodity CPUs. We see these lines of work as orthogonal to ours, as they can be applied in the context of SheetReader to further increase performance.

10. Conclusions

Spreadsheet systems are popular for accessible data analysis but have limited capabilities when it comes to data science applications. Existing solutions for loading spreadsheets into data science environments to perform advanced analytics exhibit critical performance problems in terms of either runtime or memory usage. To address these problems, this paper introduces SheetReader, a specialized spreadsheet parsing architecture that operates in two different parsing modes, *consecutive* and *interleaved*. To improve the runtime, SheetReader parallelizes the parsing by exploiting the flat and repeating structures inherently found in spreadsheet formats. It further uses task parallelism to process worksheets and strings of the spreadsheet concurrently. To reduce the memory utilization, SheetReader tightly couples decompression and parsing. To provide a general solution for different target environments, it stores the retrieved spreadsheet values in an environment-agnostic intermediate data structure. That way, it is easy to create bindings for different targets without the modifying the core parser.

Our evaluation showed that SheetReader is highly efficient in terms of both runtime and memory usage. The *consecutive* approach offers a significant improvement in runtime and a moderate reduction in memory usage, while the *interleaved* approach yields a more moderate runtime improvement but has very low memory consumption. Since decompression creates a bottleneck, we also introduced and evaluated a method for parallel decompression, showing that with fully data-parallel processing we can further reduce the runtime while keeping the memory usage low.

In future work, we plan to investigate the applicability of existing solutions that partially parallelize the decompression of general Deflate streams, such as pugz [37]. Furthermore, we plan to incorporate SheetReader as a DBMS spreadsheet wrapper, similar to SCANRAW [38], to allow querying spreadsheets in cross-database environments [39].

Acknowledgment

This work was funded by the German Ministry for Education and Research as BIFOLD22B, and supported by the Software Campus project PolyDB.

References

[1] S. Rahman, K. Mack, M. Bendre, R. Zhang, K. Karahalios, A. Parameswaran, Benchmarking spreadsheet systems, in: SIGMOD, 2020, pp. 1589–1599.
[2] S. Rahman, M. Bendre, Y. Liu, S. Zhu, Z. Su, K. Karahalios, A.G. Parameswaran, NOAH: interactive spreadsheet exploration with dynamic hierarchical overviews, PVLDB 14 (6) (2021) 970–983.
[3] C. Li, XML Parsing, SAX/DOM, Springer US, 2009, pp. 3598–3601,
[4] T.C. Lam, J.J. Ding, J.-C. Liu, XML document parsing: Operational and performance characteristics, Computer 41 (9) (2008) 30–37, http://dx.doi.org/10.1109/MC.2008.403.
[5] F. Henze, H. Gavriilidis, E. Tzirita Zacharatou, V. Markl, Efficient specialized spreadsheet parsing for data science, in: K. Stefanidis, L. Golab (Eds.), Proceedings of the International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP), Vol. 3130, 2022, pp. 41–50, URL http://ceur-ws.org/Vol-3130/paper5.pdf.
[6] Standard ECMA-376 office open XML file formats, 2021, Accessed on 2021-12-14 https://www.ecma-international.org/publications/standards/Ecma-376.htm.
[7] P. Deutsch, RFC1951: DEFLATE compressed data format specification version 1.3, 1996.
[8] Document object model (DOM), 2021, Accessed on 2021-07-05 https://dom.spec.whatwg.org/.
[9] M. Kalicinski, RapidXML, 2021, Accessed on 2021-05-21 http://rapidxml.sourceforge.net/.
[10] Apache Software Foundation, Xerces C++, 2021, Accessed on 2021-05-30 https://xerces.apache.org/xerces-c/.
[11] The GNOME Project, libxml2, 2021, Accessed on 2021-03-09 http://www.xmlsoft.org/.
[12] A. Kapoulkine, pugixml, 2021, Accessed on 2021-06-13 https://pugixml.org/.
[13] The Expat development team, Expat, 2021, Accessed on 2021-05-16 https://libexpat.github.io/.
[14] K. Chiu, W. Lu, A compiler-based approach to schema-specific XML parsing, in: 1st Int'L. Workshop on High Performance XML Processing, 2004.
[15] M.G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, M. Mercaldi, XML screamer: an integrated approach to high performance XML parsing, validation and deserialization, in: WWW, 2006, pp. 93–102.
[16] S.C. Haw, G.R.K. Rao, A comparative study and benchmarking on xml parsers, in: The 9th International Conference on Advanced Communication Technology, Vol. 1, IEEE, 2007, pp. 321–325.
[17] T.C. Lam, J.J. Ding, J.-C. Liu, XML document parsing: Operational and performance characteristics, Computer 41 (9) (2008) 30–37.
[18] E. Perkins, M. Kostoulas, A. Heifets, M. Matsa, N. Mendelsohn, Performance Analysis of XML APIs, Citeseer, 2005.
[19] S. Chilingaryan, The XMLbench project: Comparison of fast, multi-platform XML libraries, 2009, pp. 21–34, http://dx.doi.org/10.1007/978-3-642-04205-8_4.
[20] H. Gavriilidis, Computation offloading in JVM-based dataflow engines, BTW–Workshopband (2019).
[21] W. Lu, K. Chiu, Y. Pan, A parallel approach to XML parsing, in: GridCom, 2006, pp. 223–230.
[22] Y. Pan, W. Lu, Y. Zhang, K. Chiu, A static load-balancing scheme for parallel XML parsing on multicore CPUs, in: CCGrid, 2007, pp. 351–362.
[23] Y. Pan, Y. Zhang, K. Chiu, Simultaneous transducers for data-parallel XML parsing, in: IPDPS, 2008, pp. 1–12, http://dx.doi.org/10.1109/IPDPS.2008.4536240.
[24] E. Koci, M. Thiele, O. Romero, W. Lehner, Table identification and reconstruction in spreadsheets, in: CAiSE, 2017, pp. 527–541, http://dx.doi.org/10.1007/978-3-319-59536-8_33.
[25] M. Nicola, J. John, XML parsing: a threat to database performance, in: CIKM, 2003, pp. 175–178.
[26] B. Shah, P.R. Rao, B. Moon, M. Rajagopalan, A data parallel algorithm for XML DOM parsing, in: XSym, Vol. 5679, 2009, pp. 75–90.
[27] R.D. Cameron, K.S. Herdy, D. Lin, High performance XML parsing using parallel bit stream technology, in: CASCON, 2008, pp. 222–235.
[28] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli, D. Kossmann, Speculative distributed CSV data parsing for big data analytics, in: SIGMOD, 2019, pp. 883–899.
[29] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, T. Neumann, Instant loading for main memory databases, PVLDB 6 (14) (2013) 1702–1713.
[30] E. Stehle, H. Jacobsen, ParPaRaw: Massively parallel parsing of delimiter-separated raw data, PVLDB 13 (5) (2020) 616–628.
[31] A. Kumaigorodski, C. Lutz, V. Markl, Fast CSV loading using GPUs and RDMA for in-memory data processing, BTW (2021).
[32] M. Karpathiotakis, M. Branco, I. Alagiannis, A. Ailamaki, Adaptive query processing on RAW data, PVLDB 7 (12) (2014) 1119–1130.
[33] T. Döhmen, H. Mühleisen, P. Boncz, Multi-hypothesis CSV parsing, in: SSDBM, 2017, pp. 1–12.
[34] S. Palkar, F. Abuzaid, P. Bailis, M. Zaharia, Filter before you parse: Faster analytics on raw data with sparser, PVLDB 11 (11) (2018) 1576–1589.
[35] Y. Li, N.R. Katsipoulakis, B. Chandramouli, J. Goldstein, D. Kossmann, Mison: a fast JSON parser for data analytics, PVLDB 10 (10) (2017) 1118–1129.
[36] G. Langdale, D. Lemire, Parsing gigabytes of JSON per second, VLDB J. 28 (6) (2019) 941–960.
[37] M. Kerbiriou, R. Chikhi, Parallel decompression of gzip-compressed files and random access to DNA sequences, in: IPDPSW, 2019, pp. 209–217, http://dx.doi.org/10.1109/IPDPSW.2019.00042.
[38] Y. Cheng, F. Rusu, Parallel in-situ data processing with speculative loading, in: SIGMOD, 2014, pp. 1287–1298, http://dx.doi.org/10.1145/2588555.2593673, URL https://dl.acm.org/doi/10.1145/2588555.2593673.
[39] H. Gavriilidis, K. Beedkar, J.-A. Quiané-Ruiz, V. Markl, In-situ cross-database query processing, in: ICDE, 2023.