Spreadsheet Data Extractor (SDE): A Performance-Optimized, User-Centric Tool for Transforming Semi-Structured Excel Spreadsheets into Relational Data

ANONYMOUS AUTHOR(S)
SUBMISSION ID:

Organizations across various sectors frequently struggle to analyze and utilize semi-structured data derived from spreadsheets due to the lack of defined structure. This paper introduces the Spreadsheet Data Extractor (SDE), an open-source tool designed to convert semi-structured spreadsheet data into structured formats without requiring programming knowledge. Building upon previous work, we have enhanced the SDE with incremental loading of worksheets, accurate rendering of cell dimensions by parsing XML data, and performance optimizations to handle large datasets efficiently. We compare our tool with existing solutions and demonstrate its effectiveness through performance evaluations, highlighting its potential to facilitate efficient and reliable data extraction from diverse spreadsheet formats.

CCS Concepts: • **Applied computing** → **Spreadsheets**; • **Information systems** → **Data cleaning**; • **Software and its engineering** → **Extensible Markup Language (XML)**; • **Human-centered computing** → *Graphical user interfaces*; • **Theory of computation** → Data compression.

ACM Reference Format:
Anonymous Author(s). 2025. Spreadsheet Data Extractor (SDE): A Performance-Optimized, User-Centric Tool for Transforming Semi-Structured Excel Spreadsheets into Relational Data. In *Proceedings of 2025 International Conference on Management of Data (ACM PODS '25).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/XXXXXXX.XXXXXXX

1 INTRODUCTION

Spreadsheets are ubiquitous tools used across various domains, including healthcare [3], nonprofit organizations [10, 13], finance, commerce, academia, and government [6]. Despite their widespread use, analyzing and utilizing data stored in spreadsheets poses significant challenges due to their semi-structured nature. Data in spreadsheets are often formatted for human readability, employing layouts with empty cells, merged cells, hierarchical headers, and multiple tables, which hinder machine readability and automated data processing. While these unstructured formats have advantages—such as providing an easily comprehensible hierarchy of metadata for humans—they complicate automated data extraction.

Previous work by Alexander Aue, Norbert Röder, and Andrea Ackermann introduced a tool that facilitates data extraction from Excel files [1]. While effective, their solution faced performance issues and inaccuracies in rendering cell dimensions, limiting its usability with large and complex datasets.

In this paper, we aim to transform semi-structured spreadsheet data into machine-readable formats by building upon their foundational work. We present the Spreadsheet Data Extractor

ACM PODS '25, June 22–27, 2025, Berlin, Germany

ACM ISBN …$15.00
https://doi.org/10.1145/XXXXXXX.XXXXXXX

(SDE), an enhanced tool that enables users to define data hierarchies through cell selection without any programming knowledge. Our enhancements address key limitations of the existing solution, making data extraction from complex spreadsheets more efficient and user-friendly.

1.1 Contributions

Our main contributions are as follows:

(1) We release the SDE under the open-source GNU General Public License v3.0, promoting community access and collaboration.
(2) We implement incremental loading of worksheets to enhance performance, allowing the tool to handle large Excel files efficiently.
(3) We accurately render row heights and column widths by parsing XML data, ensuring that the spreadsheet's visual representation closely matches that of Excel.
(4) We optimize the rendering engine to draw only the visible cells, significantly improving performance when dealing with large datasets.
(5) We integrate the selection hierarchy, worksheet view, and output preview into a unified interface, streamlining the user experience.

2 RELATED WORK

The extraction of relational data from semi-structured documents, particularly spreadsheets, has garnered significant attention due to their ubiquitous use across domains such as business, government, and scientific research. Several frameworks and tools have been developed to address the challenges of converting flexible spreadsheet formats into normalized relational forms suitable for data analysis and integration. Notable among these are **DeExcelerator**, **XLIndy**, **FLASHRELATE**, **Senbazuru**, **TableSense** und den Ansatz von Aue et al. auf dessen arbeit wir aufbauen.

2.1 Aue et al.'s Converter

Aue et al. [1] developed a tool aimed at facilitating data extraction from Excel spreadsheets by utilizing the Dart ‘excel‘ package to open ‘.xlsx‘ files. Users can select cells containing data and metadata to define the data hierarchy. However, this method encounters performance bottlenecks as the package requires loading the entire ‘.xlsx‘ file into memory before processing, leading to slow response times, especially with large files. Die Lösung nutzte das Dart-Package excel, um die .xlsx-Dateien zu öffnen. Dies war jedoch sehr langsam, da das Paket die gesamte .xlsx-Datei zunächst vollständig einliest. Wir haben daher eine eigene Funktionalität in Dart implementiert, die die Excel-Arbeitsblätter inkrementell lädt. Additionally, their solution calculates row heights and column widths based solely on cell content, disregarding the actual dimensions specified in the Excel file. This results in discrepancies between the tool's rendering and the original spreadsheet. The tool also renders all cells regardless of their visibility within the viewport, causing significant performance degradation when handling worksheets with numerous cells.

2.2 DeExcelerator

Eberius et al. [7] introduced **DeExcelerator**, a framework that transforms partially structured spreadsheets into first normal form relational tables using heuristic-based extraction phases. It addresses challenges such as table detection, metadata extraction, and layout normalization. While effective in automating normalization, its reliance on predefined heuristics limits adaptability to heterogeneous or unconventional spreadsheet formats, highlighting the need for more flexible approaches.

2.3 XLIndy

Koci et al. [8] developed **XLIndy**, an interactive Excel add-in with a Python-based machine learning backend. Unlike DeExcelerator's fully automated heuristic approach, XLIndy integrates machine learning techniques for layout inference and table recognition, enabling a more adaptable and accurate extraction process. XLIndy's interactive interface allows users to visually inspect extraction results, adjust configurations, and compare different extraction runs, facilitating iterative fine-tuning. Additionally, users can manually revise predicted layouts and tables, saving these revisions as annotations to improve classifier performance through (re-)training. This user-centric approach enhances the tool's flexibility, allowing it to accommodate diverse spreadsheet formats and user-specific requirements more effectively than purely heuristic-based systems.

2.4 FLASHRELATE

Barowy et al. [2] presented **FLASHRELATE**, an approach that empowers users to extract structured relational data from semi-structured spreadsheets without requiring programming expertise. FLASHRELATE introduces a domain-specific language, **FLARE**, which extends traditional regular expressions with spatial constraints to capture the geometric relationships inherent in spreadsheet layouts. Additionally, FLASHRELATE employs an algorithm that synthesizes FLARE programs from a small number of user-provided positive and negative examples, significantly simplifying the automated data extraction process.

FLASHRELATE distinguishes itself from both DeExcelerator and XLIndy by leveraging programming-by-example (PBE) techniques. While DeExcelerator relies on predefined heuristic rules and XLIndy incorporates machine learning models requiring user interaction for fine-tuning, FLASHRELATE allows non-expert users to define extraction patterns through intuitive examples. This approach lowers the barrier to entry for extracting relational data from complex spreadsheet encodings, making the tool accessible to a broader range of users.

2.5 Senbazuru

Chen et al. [4] introduced **Senbazuru**, a prototype Spreadsheet Database Management System (SSDBMS) designed to extract relational information from a large corpus of spreadsheets. Senbazuru addresses the critical issue of integrating data across multiple spreadsheets, which often lack explicit relational metadata, thereby hindering the use of traditional relational tools for data integration and analysis.

Senbazuru comprises three primary functional components:

(1) **Search**: Utilizing a textual search-and-rank interface, Senbazuru enables users to quickly locate relevant spreadsheets within a vast corpus. The search component indexes spreadsheets using Apache Lucene, allowing for efficient retrieval based on relevance to user queries.
(2) **Extract**: The extraction pipeline in Senbazuru consists of several stages:
    - **Frame Finder**: Identifies data frame structures within spreadsheets using Conditional Random Fields (CRFs) to assign semantic labels to non-empty rows, effectively detecting rectangular value regions and associated attribute regions.
    - **Hierarchy Extractor**: Recovers attribute hierarchies for both left and top attribute regions. This stage also incorporates a user-interactive repair interface, allowing users to manually correct extraction errors, which the system then generalizes to similar instances using probabilistic methods.
    - **Tuple Builder and Relation Constructor**: Generates relational tuples from the extracted data frames and assembles these tuples into coherent relational tables by

clustering attributes and recovering column labels using external schema repositories like Freebase and YAGO.

(3) **Query**: Supports basic relational operations such as selection and join on the extracted relational tables, enabling users to perform complex data analysis tasks without needing to write SQL queries.

Senbazuru's ability to handle hierarchical spreadsheets, where attributes may span multiple rows or columns without explicit labeling, sets it apart from earlier systems like DeExcelerator and XLIndy. By employing machine learning techniques and providing user-friendly repair interfaces, Senbazuru ensures high-quality extraction and facilitates the integration of spreadsheet data into relational databases.

2.6 TableSense

Dong et al. [5] developed **TableSense**, an end-to-end framework for spreadsheet table detection using Convolutional Neural Networks (CNNs). TableSense addresses the diversity of table structures and layouts by introducing a comprehensive cell featurization scheme, a Precise Bounding Box Regression (PBR) module for accurate boundary detection, and an active learning framework to efficiently build a robust training dataset.

While **DeExcelerator**, **XLIndy**, **FLASHRELATE**, and **Senbazuru** focus primarily on transforming spreadsheet data into relational forms through heuristic, machine learning, and programming-by-example approaches, **TableSense** specifically targets the accurate detection of table boundaries within spreadsheets using deep learning techniques. Unlike region-growth-based methods employed in commodity spreadsheet tools, which often fail on complex table layouts, TableSense achieves superior precision and recall by leveraging CNNs tailored for the unique characteristics of spreadsheet data. However, TableSense focuses on table detection and visualization, allowing users to generate diagrams from the detected tables but does not provide functionality for exporting the extracted data for further analysis.

2.7 Comparison and Positioning

While **DeExcelerator**, **XLIndy**, **FLASHRELATE**, **Senbazuru**, and **TableSense** each offer unique approaches to spreadsheet data extraction, they share certain limitations. Many of these tools are not readily accessible: **FLASHRELATE** and **TableSense** are proprietary, and **Senbazuru**, **XLIndy**, and **DeExcelerator** are discontinued projects with limited or no source code availability. In contrast, we contribute our spreadsheet data extractor under the GNU General Public License v3.0, allowing the community to access, use, and improve the tool freely.

Moreover, unlike the aforementioned tools that rely on heuristics, machine learning, or AI techniques—which can introduce errors requiring users to identify and correct—we adopt a user-centric approach that gives users full control over data selection and metadata hierarchy definition. While this requires more manual input, it eliminates the uncertainty and potential inaccuracies associated with automated methods. To streamline the process and enhance efficiency, our tool includes user-friendly features such as the ability to duplicate hierarchies of columns and tables, and to move them over similar structures for reuse, reducing the need for repetitive configurations.

By combining the strengths of manual control with enhanced user interface features and performance optimizations, our tool offers a robust and accessible solution for extracting relational data from complex and visually intricate spreadsheets. These enhancements not only improve performance and accuracy but also elevate the overall user experience, making our tool a valuable asset for efficient and reliable data extraction from diverse spreadsheet formats.

3 METHODOLOGY

In this section, we detail the design and implementation of the Spreadsheet Data Extractor (SDE), emphasizing its user-centric approach and performance optimizations. The SDE enables users to transform semi-structured spreadsheet data into structured, machine-readable formats without requiring programming expertise. We achieve this through an intuitive interface that allows for cell selection and hierarchy definition, incremental loading of worksheets, accurate rendering of cell dimensions, and optimized performance for handling large datasets by incrementally loading of worksheets and dadurch dass wir nur solche Zellen rendern, die in der aktuellen ansicht auch sichtbar sind.

3.1 User-Centric Data Extraction

The core functionality of the SDE revolves around allowing users to select cells containing data and metadata to define a data hierarchy. This process is facilitated through a graphical interface that displays the spreadsheet and allows for intuitive selection and manipulation of the selection hierarchy.

3.1.1 Hierarchy Definition. Users can select individual cells or ranges of cells by clicking and pressing using shift-click for multi-selection. These selections represent either data or metadata.

The selected cells are organized into a hierarchical tree structure, where each node represents a data element, and child nodes represent nested data or metadata. This hierarchy defines how the data will be transformed into a structured format.

3.1.2 Reusability and Efficiency. To optimize the extraction process and reduce repetitive tasks, the SDE allows users to duplicate previously defined hierarchies and apply them to similar regions within the spreadsheet. This feature is particularly useful for spreadsheets with repeating structures, such as multiple tables with the same format.

4 EXAMPLE WORKFLOW

Consider a spreadsheet containing statistical forecasts of future nursing staff availability in Germany [11]. Figure 1 shows the SDE interface, which consists of three main components:

- **Hierarchy Panel (Top Left)**: Displays the hierarchy of cell selections, initially empty.
- **Spreadsheet View (Top Right)**: Shows the currently opened Excel file and the selected worksheet for cell selection.
- **Output Preview (Bottom)**: Provides immediate feedback on the data extraction based on current selections.

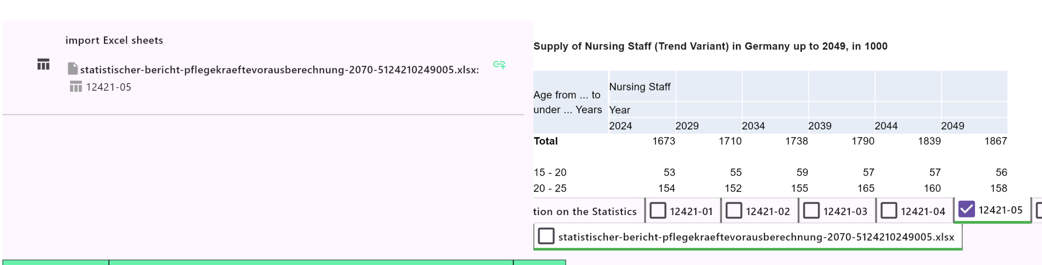


Fig. 1. The SDE interface overview, showing the hierarchy panel, spreadsheet view, and output preview.

4.1 Selection of the First Column

The user begins by adding a node to the hierarchy and selecting the cell containing the metadata "Nursing Staff" (Figure 2). This cell represents metadata common to all cells in the worksheet and should appear at the beginning of each row in the output CSV file.

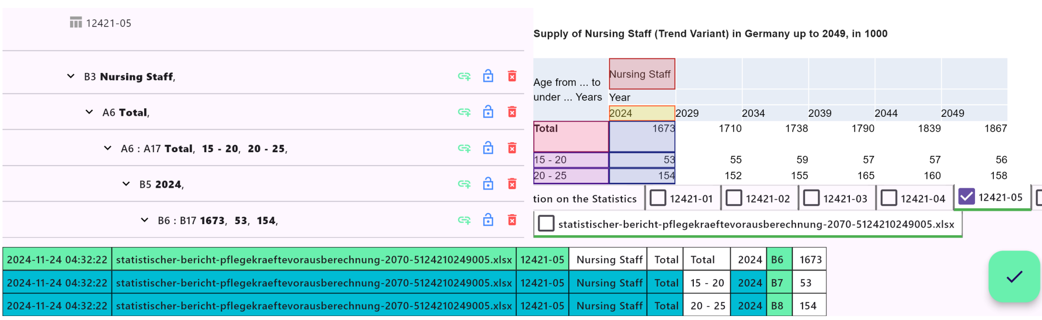


Fig. 2. Selecting the "Nursing Staff" cell to add as metadata in the hierarchy.

Within this node, the user adds a child node and selects the cell "Total", which serves both as a table header and a row label. This selection represents the header of the first subtable.

Next, the user adds another child node under "Total" and selects the range of cells containing row labels ("Total," "15-20," "20-25," etc.) by clicking the first cell and shift-clicking the last cell.

Under the row labels node, the user adds a child node for the year "2024" and selects the corresponding data cells (e.g., "1673," "53," "154," etc.). The hierarchy now contains five nodes, each nested within its parent node.

In the spreadsheet view, the selected cells are highlighted with different colors corresponding to their nodes in the hierarchy. The output preview displays the extracted data, with each node represented as a column. If a selection includes multiple cells, the values appear as rows in the output.

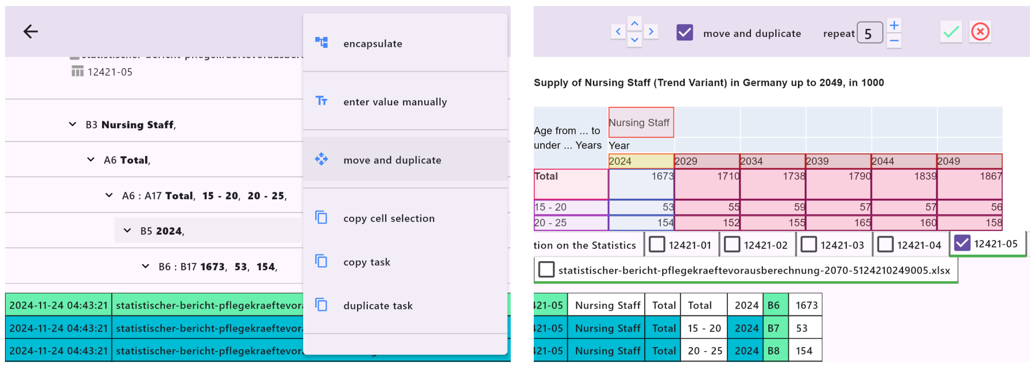
4.2 Duplication of the Column Hierarchy

To avoid repetitive manual entry for additional years, the user utilizes the "Move and Duplicate" feature to duplicate the hierarchy for the year "2024" and adjust the cell selections to include data for subsequent years (e.g., "2025," "2026").

The user selects the node corresponding to "2024" ,right-clicks it, and chooses "Move and Duplicate" from the context menu (Figure 3a). This feature allows shifting cell selections horizontally or vertically and specifying the number of repetitions, streamlining the process of capturing similar data structures.

After selecting "Move and Duplicate", a toolbar appears, allowing the user to shift the selection to the right by one unit. To retain the original selection while adding new ones, the user checks the "move and duplicate" option. By entering the number of repetitions—in this case, "5"—the user duplicates the selection of the "2024" column five times, shifting each new selection one column to the right.

The user reviews the selections in the spreadsheet view, where each selection is highlighted in a different color corresponding to its node in the hierarchy. After verifying the selections, the user clicks the "Accept" button to apply the changes. The updated hierarchy is shown in Figure 4.



(a) Accessing the "Move and Duplicate" feature for the "2024" node.

(b) Setting the number of repetitions to duplicate the selection across additional years.

Fig. 3. Duplicating the column hierarchy for additional years.

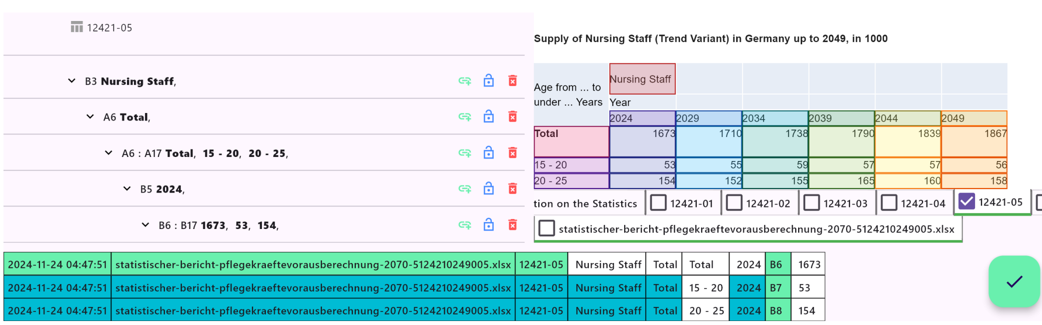


Fig. 4. The hierarchy after duplicating the column selections for additional years.

4.3 Duplicating the Table Hierarchy

The same method used for duplicating columns can be applied to the subtables, as shown in Figure 5. By selecting the node with the value "Total" and clicking the "Move and Duplicate" button, the user applies the selection of the "Total" subtable to the other subtables below by shifting the selection downward.

However, the child nodes of the "Total" node include the column headers, which are not repeated in the subtables. To prevent these header cells from moving during duplication, the user locks their selection by clicking the padlock icon on the corresponding nodes. This action freezes their cell selections, keeping them fixed during the duplication process.

With the header nodes locked as shown in Figure 5, the user sets the number of repetitions to "2" to duplicate the subtable selection for the remaining subtables. The final selection covers all relevant data in the worksheet.

4.4 Cross-Product Transformation

The hierarchy resulting from the selected nodes is represented as a graph shown in Figure 6. For simplicity, the example focuses on the first three columns and the first three rows of the first subtable.

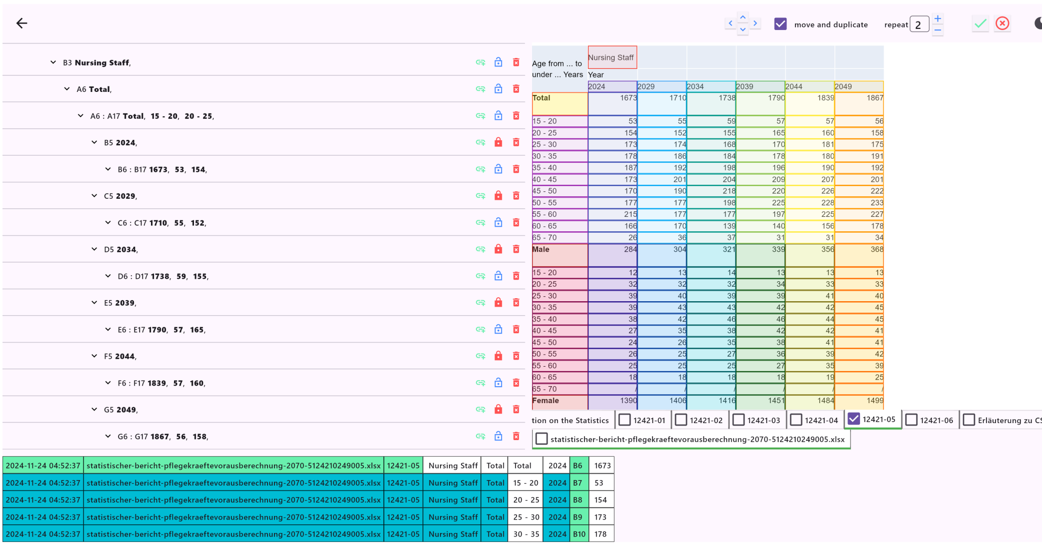


Fig. 5. Selection of all cells in the subtables after duplicating the hierarchy of the first table.

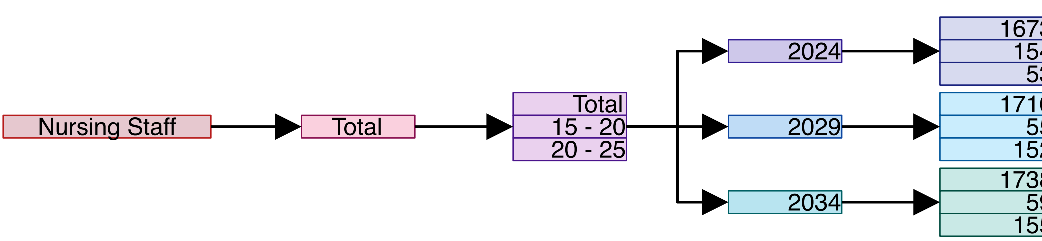


Fig. 6. The initial hierarchy graph before the cross-product transformation.

Once the hierarchy is defined, the SDE applies a cross-product transformation to generate a relational format from the selection graph. This transformation consists of two key steps:

(1) **Node Duplication**: Nodes with multiple incoming or outgoing edges (e.g., the row labels node with the values *Total*, *15-20*, *20-25*) are duplicated to ensure that each edge connects to a unique instance of the node. This adjustment replaces the original many-to-one relationships with one-to-one mappings for each edge.

(2) **Value Replication**: Nodes containing single values (e.g., the year *2024*) are replicated to align with the number of values associated with the connected node. This ensures a consistent structure in the relational output, maintaining alignment across all hierarchical levels.

The resulting graph after the cross-product transformation is shown in Figure 7. With this transformation, the SDE generates a structured output that reflects the original spreadsheet's hierarchical relationships, enabling users to analyze and integrate the data effectively.

4.5 Incremental Loading of Worksheets

Opening large Excel files traditionally involves loading the entire file and all its worksheets into memory before displaying any content. In files containing very large worksheets, this process

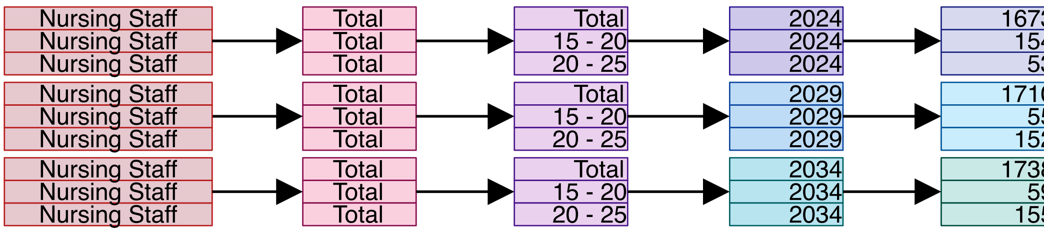


Fig. 7. The hierarchy graph after applying the cross-product transformation.

can take several seconds to minutes, causing significant delays for users who need to access data quickly.

To facilitate efficient data extraction from multiple Excel files, we implemented a mechanism for incremental loading of worksheets within the SDE.

Excel files (`.xlsx` format) are ZIP archives containing a collection of XML files that describe the worksheets, styles, and shared strings. Key components include:

- **`xl/sharedStrings.xml`**: Contains all the unique strings used across worksheets, reducing redundancy.
- **`xl/styles.xml`**: Defines the formatting styles for cells, including fonts, colors, and borders.
- **`xl/worksheets/sheetX.xml`**: Represents individual worksheets (`sheet1.xml`, `sheet2.xml`, etc.).

Our solution opens the Excel file as a ZIP archive and initially extracts only the essential metadata and shared resources required for the application to function. This initial extraction includes:

(1) **Metadata Extraction**:
We read the archive's directory to identify the contained files without decompressing them fully. This step is quick, taking only a few milliseconds, and provides information about the available worksheets and shared resources.

(2) **Selective Extraction**:
We immediately extract `sharedStrings.xml` and `styles.xml` because these files are small and contain information necessary for rendering cell content and styles across all worksheets. These files are parsed and stored in memory for quick access during rendering.

(3) **Deferred Worksheet Loading**:
The individual worksheet files (`sheetX.xml`) remain compressed and are loaded into memory in their binary unextracted form. They are not decompressed or parsed at this stage.

(4) **On-Demand Parsing**:
When a user accesses a specific worksheet—either by selecting it in the interface or when a unit test requires data from it—the corresponding `sheetX.xml` file is then decompressed and parsed. This parsing occurs in the background and is triggered only by direct user action or programmatic access to the worksheet's data.

(5) **Memory Release**:
After a worksheet has been decompressed and its XML parsed, we release the memory resources associated with the parsed data. This approach prevents excessive memory usage and ensures that the application remains responsive even when working with multiple large worksheets.

By adopting this incremental loading approach, users experience minimal wait times when opening an Excel file. The initial loading is nearly instantaneous, allowing users to begin interacting with the application without delay. This contrasts with traditional methods that require loading all worksheets upfront, leading to significant wait times for large files.

4.6 Rendering of Worksheets

To ensure that users can navigate worksheets without difficulty, we prioritize displaying the worksheets in a manner that closely resembles their appearance in Excel. This involves accurately rendering cell dimensions, formatting, and text behaviors.

4.6.1 Displaying Row Heights and Column Widths. Our solution extracts information about column widths and row heights directly from the Excel file's XML structure. Specifically, we retrieve the column widths from the *width* attribute of the `<col>` elements and the row heights from the *ht* attribute of the `<row>` elements in the `sheetX.xml` files.

In Excel, column widths and row heights are measured in units that do not directly correspond to pixels. Therefore, these values must be converted to pixel units for accurate on-screen rendering. We discovered that different scaling factors are needed for columns and rows:

- **Column Widths**: To convert the Excel column width to pixels, we multiply the width value by a factor of 7. This factor aligns the rendered column widths with their appearance in Excel.

$$\text{Column Width in Pixels} = \text{Excel Column Width} \times 7$$

- **Row Heights**: For row heights, we multiply the height value by a factor of $\frac{4}{3}$ to obtain the pixel height.

$$\text{Row Height in Pixels} = \text{Excel Row Height} \times \frac{4}{3}$$

We determined these scaling factors empirically through a series of tests. We created Excel files with cells of various sizes and examined the corresponding XML files to observe the recorded width and height values. By comparing the rendered sizes in Excel with the values in the XML, we derived the appropriate scaling factors. We then verified our calculations by adjusting the width and height values in the XML files and observing the effects in Excel.

Despite extensive research, we could not find official documentation explaining the rationale behind these specific scaling factors. This lack of documentation poses a challenge for accurately replicating Excel's rendering. Nevertheless, by applying these empirically derived factors, we can render the column widths and row heights to closely match Excel's display, enhancing the user's familiarity and ease of use.

4.6.2 Cell Formatting. Cell formatting plays a crucial role in accurately representing the appearance of worksheets. Formatting information is stored in the `styles.xml` file, where styles are defined and later referenced in the `sheetX.xml` files.

Each cell in the worksheet references a style index through the *s* attribute, which points to the corresponding `<xf>` (Extended Format) element within the `cellXfs` collection. These `<xf>` elements contain attributes such as *fontId*, *fillId*, and *borderId*, which reference specific font, fill (background), and border definitions located in the `fonts`, `fills`, and `borders` collections, respectively. By parsing these references, we can accurately apply the appropriate fonts, background colors, and border styles to each cell.

Through meticulous parsing and application of these formatting details, we ensure that the rendered worksheet closely mirrors the original Excel file, preserving the visual cues and aesthetics that users expect.

4.6.3 Handling Text Overflow. In Excel, when the content of a cell exceeds its width, the text may overflow into adjacent empty cells, provided those cells do not contain any data. If adjacent cells are occupied, Excel truncates the overflowing text at the cell boundary. Replicating this behavior is essential for accurate rendering and user familiarity.

We implemented text overflow handling by:

To implement this, we check if the adjacent cell to the right is empty before allowing text to overflow. If the adjacent cell is empty, we extend the text rendering. If the adjacent cell contains data, we truncate the text at the boundary of the original cell.

Figure 1 illustrates this behavior. The text "Supply of Nursing Staff ..." extends into the neighboring cell because it is empty. If not for this handling, the text would be truncated at the cell boundary, leading to incomplete data display as shown in Figure 8.

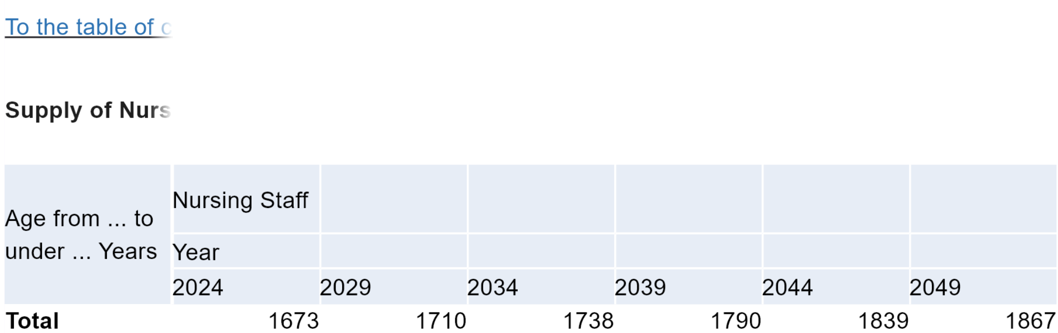


Fig. 8. Handling of text overflow in cells with adjacent empty or occupied cells

By accurately handling text overflow, we improve readability and maintain consistency with Excel's user interface, which is crucial for users transitioning between Excel and our tool.

4.7 Performance Optimization

To ensure high frame rates even with large worksheets, we optimized the Spreadsheet Data Extractor to render only the cells that are currently visible to the user. Rendering the entire worksheet, especially when it contains thousands of cells, can significantly degrade performance. By focusing on the visible cells within the viewport, we reduce computational overhead and improve responsiveness.

We achieve this optimization by utilizing the `two_dimensional_scrollables` package [9], which was first released on August 17, 2023. This package provides functionality for efficiently handling two-dimensional scrolling regions, making it suitable for rendering large grids like spreadsheets.

Since we extract all column widths and row heights from the XML files, we can calculate the exact dimensions and positions of each cell. By accumulating the widths and heights, we determine the coordinates of each cell within the worksheet grid. These coordinates are essential for identifying which cells fall within the current viewport and should be rendered.

We consider the current scroll offsets along the horizontal (x-axis) and vertical (y-axis) directions. The viewport is defined by the width and height of the panel displaying the Excel worksheet. To determine the visible cells, we perform the following steps:

- **Horizontal Visibility**:

  - Sum the column widths until the accumulated sum reaches the left edge of the viewport. All cells to the left are ignored.
  - Continue adding column widths until the sum exceeds the right edge of the viewport. Cells beyond this point are also ignored.
- **Vertical Visibility**:
  - Sum the row heights until the accumulated sum reaches the top edge of the viewport. All cells above are ignored.
  - Continue adding row heights until the sum surpasses the bottom edge of the viewport. Cells below this point are ignored.

By rendering only the cells within these boundaries, we significantly reduce the number of cells processed at any given time.

The `two_dimensional_scrollables` package provides interfaces where the logic for laying out the cells can be implemented. Parameters that describe the viewport, including the horizontal and vertical offsets as well as the viewport height and width, are supplied by these interfaces.

Algorithm 1 outlines the method for calculating and laying out the visible cells.

Algorithm 1 Layout of Visible Spreadsheet Cells

Initialize Indices
    leadingColumnIndex ← column index corresponding to *horizontalOffset*
    leadingRowIndex ← row index corresponding to *verticalOffset*
    trailingColumnIndex ← column index corresponding to *horizontalOffset* + *viewportWidth*
    trailingRowIndex ← row index corresponding to *verticalOffset* + *viewportHeight*
Calculate Initial Offsets
    leadingColumnOffset ← sum of widths from the first column up to *leadingColumnIndex*
    leadingRowOffset ← sum of heights from the first row up to *leadingRowIndex*
    horizontalLayoutOffset ← *leadingColumnOffset* − *horizontalOffset*
for each *columnIndex* from *leadingColumnIndex* to *trailingColumnIndex* **do**
        verticalLayoutOffset ← *leadingRowOffset* − *verticalOffset*
    for each *rowIndex* from *leadingRowIndex* to *trailingRowIndex* **do**
            cell ← build or retrieve the cell at (*columnIndex*, *rowIndex*)
            Layout *cell* at position (*horizontalLayoutOffset*, *verticalLayoutOffset*)
        if a custom height is defined for row *rowIndex* **then**
                verticalLayoutOffset ← *verticalLayoutOffset*+ height of row *rowIndex*
        else
                verticalLayoutOffset ← *verticalLayoutOffset* + *defaultRowHeight*
        end if
    end for
        columnWidth ← width of column *columnIndex*
        horizontalLayoutOffset ← *horizontalLayoutOffset* + *columnWidth*
end for

By applying this method, we render only the cells necessary for the current view, thereby optimizing performance and ensuring smooth user interactions even with large and complex worksheets.

5 EVALUATION

The fundamental approach of the Spreadsheet Data Extractor, based on the converter by Alexander Aue et al. [1], upon which we build, remains unchanged. The effectiveness of this approach has already been investigated. Aue et al. evaluated the extraction of data from over 500 Excel files. The time required for each file was determined from a sample of 331 processed Excel files comprising 3,093 worksheets. On average, student assistants needed 15 minutes per file and 95 seconds per worksheet.

Our focus is on improving the user experience and optimizing the performance of the Spreadsheet Data Extractor. We enhanced the user experience by displaying the Excel worksheets similarly to how they appear in Excel, and by reducing the number of required clicks through the integration of the selection hierarchy, worksheet view, and output preview into a single interface. Performance was improved by implementing incremental loading of Excel files and rendering only the visible cells.

5.1 Acceleration When Opening Files

To evaluate the performance improvements when opening Excel files, we conducted a series of tests using a large Excel file. We downloaded the entire collection of Excel files from the German Federal Statistical Office (Destatis)[1], which provides extensive statistical data across various domains. From this collection, we identified the largest file [12], which has a compressed size of 87 MB and an uncompressed size of 911 MB.

We performed three sets of tests to compare the performance of different methods for opening and reading data from this Excel file:

1. **VBA Script in Excel**: We wrote a VBA script that opens the Excel file and reads specific values from a worksheet. This script simulates how users might interact with the file using Excel's built-in capabilities. We ran this script 10 times, measuring the time required to open the file and read the values in each iteration.
2. **Spreadsheet Data Extractor**: We developed an equivalent unit test using the functions implemented in our Spreadsheet Data Extractor to open the same file and read the same cells. This test aimed to assess the performance of our tool under the same conditions. We repeated this test 10 times, recording the runtime for each execution.
3. **`excel` Package**: We tested the `excel` package used by Alexander Aue et al. [1], creating a unit test that attempts to read the same cells from the file. This test was intended to benchmark the performance of the prior solution. However, due to limitations with the package, we encountered an out-of-memory exception during the second run; therefore, only the time from the first run was recorded.

All tests were conducted on a machine with the following specifications: Intel Core i5-10210U quad-core CPU at 1.60 GHz, 16 GB RAM, and a solid-state drive (SSD), running Windows 10 Pro. The version of Microsoft Excel used was Microsoft Office LTSC Professional Plus 2021.

The results of these runtime measurements are presented in Figure 9.

The Spreadsheet Data Extractor opened the worksheet with a median time of 120milliseconds and an average time of 178milliseconds. The first run took 668 milliseconds, possibly because the file was not yet cached and had to be loaded from the disk.

In contrast, Excel opened the worksheet with a median time of 40.281 seconds and an average time of 41.138 seconds.

[1]https://www.destatis.de

The Dart `excel` package used in the previous work took 13minutes and 15seconds to open the worksheet in the first run. The other nine runs could not be completed because an out-of-memory exception was thrown during the second run.

These results demonstrate that the Spreadsheet Data Extractor opens worksheets over two orders of magnitude faster than Excel and nearly four orders of magnitude faster than the Dart `excel` package used in prior work.

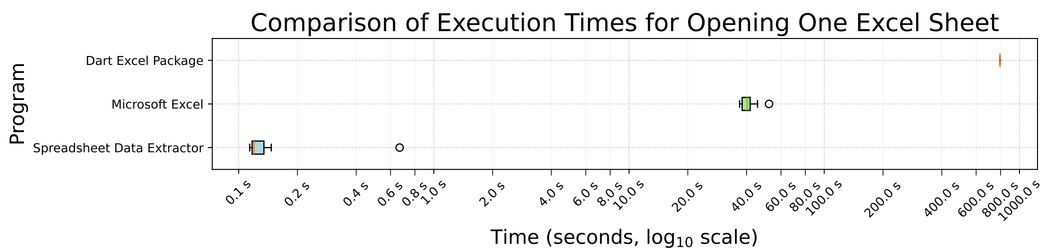


Fig. 9. Boxplot of worksheet opening times comparing the Spreadsheet Data Extractor, Excel, and the Dart `excel` package

6 FUTURE WORK

We plan to continue improving the Spreadsheet Data Extractor by implementing new features that enhance the user experience and address current limitations. One such improvement is the correct rendering of text aligned using Excel's "Center Across Selection" horizontal alignment.

In parallel, we intend to test the tool on additional datasets to further evaluate its effectiveness and efficiency.

6.1 Implementing "Center Across Selection"

Despite our efforts to replicate the display of Excel files as accurately as possible, there are still some differences between Excel's rendering and that of our Spreadsheet Data Extractor. These discrepancies are unintentional and can be resolved with further development. One such difference is illustrated in Figure 10. In Excel, the texts "Nursing Staff" and "Year" are centered across the entire column header, whereas in our tool, they are left-aligned.

Although we parse merged cells—which are commonly used in Excel to center text across multiple cells—the worksheet in question does not use merged cells for this visual representation. Instead, the first cell has a horizontal alignment that is set to `centerContinuous`. Our previous attempts to determine the cell coordinates over which this horizontal centering extends have been unsuccessful. Further testing and research are necessary to understand how such centered texts are stored in the Excel file's XML structure.

6.2 Evaluation on Real-World Data

To date, the base version of the tool has been tested on a dataset from the Agricultural Structure Survey on land use and livestock in Germany for 2020. It would be valuable to test the new version of the tool on data from before 2020 to assess whether the improvements we have made enhance the tool's effectiveness.

We plan to utilize the timestamps documented in the output CSV files to compare them with new timestamps obtained from re-testing. This comparison will help us determine whether the student assistants can work faster with the new version of the tool.

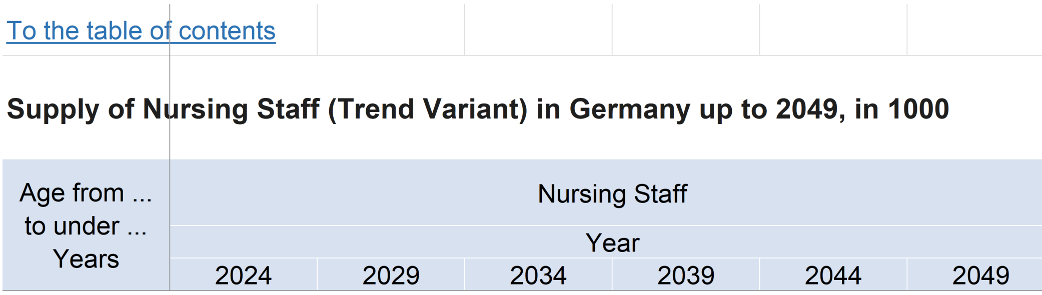


Fig. 10. Illustration of the text alignment "Center Across Selection" Excel

7 CONCLUSION

In this paper, we presented the Spreadsheet Data Extractor, an enhanced tool that builds upon the foundational work of Alexander Aue et al. [1]. By improving upon their existing approach, we addressed key limitations and introduced significant enhancements to both performance and user experience.

Our contributions include implementing incremental loading of worksheets, accurate rendering of cell dimensions by parsing XML data, and optimizing the rendering engine to draw only the visible cells. These improvements enable the tool to open large Excel files over two orders of magnitude faster than Microsoft Excel itself and nearly four orders of magnitude faster than the previous implementation by Aue et al. We also integrated the selection hierarchy, worksheet view, and output preview into a unified interface, streamlining the data extraction process.

By adopting a user-centric approach that gives users full control over data selection and metadata hierarchy definition without requiring programming knowledge, we provide a robust and accessible solution for data extraction. Our tool offers user-friendly features such as the ability to duplicate hierarchies of columns and tables and to move them over similar structures for reuse, reducing the need for repetitive configurations.

By contributing our improved Spreadsheet Data Extractor under the GNU General Public License v3.0, we enable the community to access, use, and enhance the tool freely, fostering collaboration and further innovation. By combining the strengths of the original approach with our enhancements in user interface and performance optimizations, our tool significantly improves the efficiency and reliability of data extraction from diverse and complex spreadsheet formats.

REFERENCES

[1] Andrea Ackermann Alexander Aue. 2024. Converting data organised for visual perception into machine-readable formats. In *44. GIL-Jahrestagung, Biodiversität fördern durch digitale Landwirtschaft*. Gesellschaft für Informatik eV, 179–184.

[2] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices* 50, 6 (2015), 218–228.

[3] D.J. Berndt, J.W. Fisher, A.R. Hevner, and J. Studnicki. 2001. Healthcare data warehousing and quality assurance. *Computer* 34, 12 (2001), 56–65. https://doi.org/10.1109/2.970578

[4] Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. 2013. Senbazuru: A prototype spreadsheet database management system. *Proceedings of the VLDB Endowment* 6, 12, 1202–1205.

[5] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. 2019. Tablesense: Spreadsheet table detection with convolutional neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 69–76.

[6] Angus Dunn. 2010. Spreadsheets-the Good, the Bad and the Downright Ugly. *arXiv preprint arXiv:1009.5705* (2010).

[7] Julian Eberius, Christoper Werner, Maik Thiele, Katrin Braunschweig, Lars Dannecker, and Wolfgang Lehner. 2013. DeExcelerator: a framework for extracting relational data from partially structured documents. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 2477–2480.

[8] Elvis Koci, Dana Kuban, Nico Luettig, Dominik Olwig, Maik Thiele, Julius Gonsior, Wolfgang Lehner, and Oscar Romero. 2019. Xlindy: Interactive recognition and information extraction in spreadsheets. In *Proceedings of the ACM Symposium on Document Engineering 2019*. 1–4.

[9] Kate Lovett. 2023. two_dimensional_scrollables package - Commit 4c16f3e. https://github.com/flutter/packages/commit/4c16f3ef40333aa0aebe8a1e46ef7b9fef9a1c1f Accessed: 2023-08-17.

[10] Gursharan Singh, Leah Findlater, Kentaro Toyama, Scott Helmer, Rikin Gandhi, and Ravin Balakrishnan. 2009. Numeric paper forms for NGOs. In *2009 International Conference on Information and Communication Technologies and Development (ICTD)*. IEEE, 406–416.

[11] Statistisches Bundesamt (Destatis). 2024. *Statistischer Bericht - Pflegekräftevorausberechnung - 2024 bis 2070*. Technical Report. Statistisches Bundesamt, Wiesbaden, Germany. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Bevoelkerungsvorausberechnung/Publikationen/Downloads-Vorausberechnung/statistischer-bericht-pflegekraeftevorausberechnung-2070-5124210249005.html Statistical Report - Projection of Nursing Staff - 2024 to 2070.

[12] Statistisches Bundesamt (Destatis). 2024. Statistischer Bericht: Rechnungsergebnis der Kernhaushalte der Gemeinden. https://www.destatis.de/DE/Themen/Staat/Oeffentliche-Finanzen/Ausgaben-Einnahmen/Publikationen/Downloads-Ausgaben-und-Einnahmen/statistischer-bericht-rechnungsergebnis-kernhaushalt-gemeinden-2140331217005.html Accessed: 2024-11-29.

[13] Hannah West and Gina Green. 2008. Because excel will mind me! the state of constituent data management in small nonprofit organizations. In *Proceedings of the Fourteenth Americas Conference on Information Systems*. Association for Information Systems, AIS Electronic Library (AISeL). https://aisel.aisnet.org/amcis2008/336

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009