

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

Alexander Aue-Johr  
Thünen-Institute  
Braunschweig, Germany  
alexander.aue@thuenen.de

## Abstract

Spreadsheets are widely used across domains, yet their human-oriented layouts (e.g., merged cells, hierarchical headers, and multiple tables) hinder automated extraction. We present the *Spreadsheet Data Extractor (SDE)*, a user-in-the-loop system that converts semi-structured spreadsheets into structured outputs without requiring programming. Users declare the structure they perceive; the engine deterministically propagates selections and renders results immediately.

Under the hood, SDE employs incremental loading, byte-level XML streaming that avoids DOM materialisation, and viewport-bounded rendering to support interactive exploration of large workbooks. Optimised for time-to-first-visual, SDE achieves approximately 70× faster opening of a selected worksheet from a large real-world workbook than Microsoft Excel. Under workload-equivalent conditions (single worksheet, full parse), SDE remains approximately 10× faster while preserving layout fidelity. These results demonstrate SDE’s ability to enable reliable and scalable extraction from diverse *Excel* spreadsheet layouts.

## 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.

## Keywords

Spreadsheets, Data cleaning, Relational Data, Excel, XML, Graphical user interfaces

## 1 Introduction

Spreadsheets underpin workflows across healthcare [7], nonprofit organizations [17, 22], finance, commerce, academia, and government [11]. Despite their ubiquity, reliably analysing and reusing their contents remains difficult for automated systems. In practice, many real-world spreadsheets are organized for human consumption [21] and therefore exhibit layouts with empty cells, merged regions, hierarchical headers, and multiple stacked tables. While such conventions aid human reading [16], they hinder machine readability and complicate extraction, integration, and downstream analytics. The reliance on spreadsheets as ad-hoc solutions poses several limitations:

- **Data Integrity and Consistency:** Spreadsheets are prone to errors, such as duplicate entries, inconsistent data formats, and inadvertent modifications, which can compromise data integrity. [1, 9]
- **Scalability Issues:** As datasets grow in size and complexity, spreadsheets become less efficient for data storage and retrieval, leading to performance bottlenecks. [15, 20]
- **Limited Query Capabilities:** Unlike databases, spreadsheets lack advanced querying and indexing features, restricting users from performing complex data analyses.

Transitioning from these ad-hoc spreadsheet solutions to standardized database systems offers numerous benefits:

- **Enhanced Data Integrity:** Databases enforce data validation rules and constraints, ensuring higher data quality and consistency.
- **Improved Scalability:** Databases are designed to handle large volumes of data efficiently, supporting complex queries and transactions without significant performance degradation.
- **Advanced Querying and Reporting:** Databases provide powerful querying languages like SQL, enabling sophisticated data analysis and reporting capabilities.
- **Seamless Integration:** Databases facilitate easier integration with various applications and services, promoting interoperability and data sharing across platforms.

Despite their ubiquity, reliable extraction into relational form remains challenging. This paper presents the *Spreadsheet Data Extractor (SDE)* [4], which enables selection-based, non-programmatic extraction from semi-structured spreadsheets and supports large, irregular workbooks with predictable, interactive performance. An earlier prototype [2] exposed limitations when operating on large workbooks; SDE addresses these limitations through a redesigned architecture and interaction model.

### 1.1 Contributions

- (1) **Unified interaction.** SDE integrates the selection hierarchy, worksheet view, and output preview into a single interface, reducing context switching and interaction count.
- (2) **DOM-free, byte-level worksheet parsing.** SDE implements a custom parser that operates directly on XLSX worksheet bytes, avoiding DOM construction and regular-expression passes over decoded strings; this substantially reduces memory footprint, parsing time, and improves robustness on large (“bloated”) sheets.

**Table 1: Spreadsheet Data Extractor counterparts**

Name	Technologies	Output	Accessibility	Frequency
DeExcelerator	heuristics	relational data	partially open source no access to GUI code	last publication 2015
FlashRelate	AI programming-by-example	relational data	proprietary, no access	last publication 2015
Senbazuru	AI	relational data	partially open source no access to GUI code	last commit 2015
XLindy	AI	relational data	no access	discontinued
TableSense	AI	diagrams	proprietary, no access	last commit 2021

- (3) **Incremental loading.** Worksheets are loaded and parsed on demand from the XLSX archive, enabling near-instant open times and interactive latency on large files (quantified in 7. Evaluation).
- (4) **Excel-faithful rendering.** Row heights, column widths, and merged regions are recovered from the XML to render worksheets faithfully to Excel, improving user orientation.
- (5) **Viewport-bounded rendering.** Only cells intersecting the viewport are rendered, keeping per-frame work proportional to visible content and enabling interactive frame rates on large worksheets.

## 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 as summarized in Table 1.

### 2.1 DeExcelerator

Eberius et al. [12] 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.

### 2.2 XLindy

Koci et al. [13] 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.

### 2.3 FLASHRELATE

Barowy et al. [6] 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.

### 2.4 Senbazuru

Chen et al. [8] 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.

Senzazuru’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 combining learning methods with an interactive repair workflow, Senzazuru improves extraction accuracy and consistency and produces relations suitable for integration into databases.

## 2.5 TableSense

Dong et al. [10] 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 **Senzazuru** 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.

## 2.6 Comparison and Positioning

A direct head-to-head comparison was not possible due to the lack of artifacts, because for the UI-oriented systems **FLASHRELATE**, **Senzazuru**, **XLIndy**, and **DeExcelerator**, we contacted the authors mentioned in the publications to obtain research artifacts (source code, UI prototypes). As of the submission deadline, we had

either not received any responses or that the project was discontinued; we could not find publicly accessible UI artifacts or runnable packages.

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 Design Philosophy

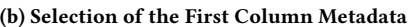
Spreadsheet tables are heterogeneous, noisy, and locally structured in ways that are hard to infer reliably with fully automatic extraction. Our goal is not to replace the analyst with an opaque model, but to *amplify* their judgment: the user points to the structure they already perceive (regions, labels, value columns), and the system guarantees a faithful, auditable transformation into a relational view. Instead of automatically extracting and then searching for mistakes, we invert the workflow: *select first, broadcast deterministically, render immediately*. This keeps discrepancies visible at the point of selection, where they can be corrected with minimal context switches.

We enforce three invariants: (i) **Provenance:** every emitted tuple is traceable to a set of visible source cells via an explicit mapping; (ii) **Stability:** small edits to selections induce bounded, predictable changes in the output (no global re-writes); (iii) **Viewport-bounded cost:** interactive operations run in time proportional to the number of cells intersecting the viewport, not the worksheet size. The parsing, indexing, and rendering subsystems are organized to uphold these invariants at scale.

*User-Centric Data Extraction.* The core interaction in the *SDE* lets users declare structure directly on the spreadsheet canvas and organize it into a hierarchy that drives a deterministic transformation into a relational view.

*Hierarchy Definition.* Users select individual cells or ranges (click, Shift-click for multi-selection). Each selection denotes either data (values) or metadata (labels/headers). Selections are arranged into a hierarchical tree: each node represents a data element; child nodes represent nested data or metadata. This hierarchy specifies how the *SDE* broadcasts selections into rows/columns in the output.

The interface supports flexible management: users drag-and-drop nodes to reparent or reorder the hierarchy when a different



organization is more appropriate. Users can also introduce *custom nodes* with user-defined text to encode metadata that is implicit in the spreadsheet but absent from cell contents.

**Reusability and Efficiency.** To reduce repetitive work on sheets with repeated layouts, users can duplicate an existing selection hierarchy and apply it to similar regions. When moving a hierarchy, some cells may need to remain fixed (e.g., vertically stacked tables where only the topmost table repeats header rows). While in “move and duplicate” mode, the SDE provides a *lock* function: users freeze specific cells while relocating the rest of the hierarchy. Locks can be toggled via the lock icon at the top-left corner of a cell or next to the corresponding selection in the hierarchy panel; locked cells remain stationary while other selections shift accordingly (see Figure 3 in 4. Example Workflow). Already-locked selections are visually indicated and can be unlocked at any time.

Consider a spreadsheet containing statistical forecasts of future nursing staff availability in Germany [18]. Figure 1a 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 worksheet for cell selection.

**Output Preview (Bottom):** Provides immediate feedback on the data extraction based on current selections.

The user adds a node to the hierarchy and selects the cell containing the metadata "Nursing Staff" (Figure 1b). This cell represents metadata that is common to all cells in this worksheet. Therefore, it should be selected first and should appear at the beginning of each row in the output CSV file.

Within this node, the user adds a child node and selects the cell *"Total"*, which serves as both a table header and a row label. This selection represents the table header of the first subtable. The user adds another child node and selects the range of cells containing row labels (e.g., *"Total"*, *"15-20"*, *"20-25"* and so forth) by clicking the first cell and shift-clicking the last cell.

A further child node is then placed under the row labels node, and the user selects the year "2024". Subsequently, an additional child node is created beneath the year node, and the user selects the corresponding data cells (e.g., "1673", "53", "154", etc.).

At this point, the hierarchy consists of five nodes, each—except the last one—containing an embedded child node. In the upper-right portion of the interface, the chosen cells are displayed in distinct colors corresponding to each node. The lower area shows a preview of the extracted output. For each child node, an additional column is appended to the output. When multiple cells are selected for a

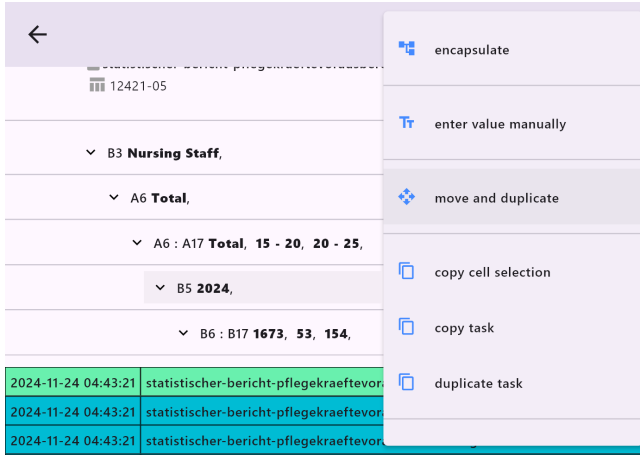


Figure 2: Invoking the “move and duplicate” feature on the 2024 column node.

given node, their values appear as entries in new rows of the output, reflecting the defined hierarchical structure.

## 4.2 Duplicating the Column Hierarchy

To avoid repetitive manual entry for additional years, the user duplicates the hierarchy for “2024” and adjusts the cell selections to include data for subsequent years (e.g., “2025,” “2026”) using the “Move and Duplicate” feature.

To do this, the user selects the node of the first column “2024” and right-clicks on it. A popup opens in which the action “move and duplicate” appears, which should then be clicked, as shown in Figure 2.

## 4.3 Duplicating the Table Hierarchy

Subsequently, a series of buttons opens in the app bar at the top right, allowing the user to move the cell selections of the node as well as all child nodes, as seen in Figure 3. By pressing the button to move the selection by one unit to the right, the next column is selected. However, this would also deselect the first column since the selection was moved. To preserve the first column, the “move and duplicate” checkbox can be activated. This creates the shifted selection in addition to the original selection. The changes are only applied when the “accept” button is clicked. The next columns could also be selected in the same way. But this can be done faster, because instead of moving the selection and duplicating it only once, the “repeat” input field can be filled with as many repetitions as there are columns. By entering the number 5, the selection of the first column is shifted 5 times by one unit to the right and duplicated at each step.

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. Only after the user has reviewed the shifted and duplicated selections in the worksheet and clicked the “accept” button are the nodes in the hierarchy created as desired. Figure 4 shows the resulting selection after the user approved the proposed changes.



[To the table of contents](#)

Supply of Nursing Staff (Trend Variant) in Germany up to 2049, in 1000

Age from ... to under ... Years	Nursing Staff					
	Year					
	2024	2029	2034	2039	2044	2049
Total	1673	1710	1738	1790	1839	1867
18 - 20	53	55	58	57	57	56
20 - 25	154	152	155	165	160	158

Figure 3: Moving the hierarchy one cell to the right while adjusting the number of repetitions to duplicate the column selection.

The same method that worked effectively for duplicating the columns can now 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, we can apply the selection of the “Total” subtable to the other subtables. This involves shifting the table downward by as many rows as necessary to overlap with the subtable below.

However, there is a minor issue: the child nodes of the “Total” node also include the column headers. If these column headers were repeated in the subtables below, shifting the selections downward would work without modification. Since these cells are not repeated in the subtables, we need to prevent the column headers cells from moving during the duplication process.

To achieve this, we can exclude individual nodes from being moved by locking their selection. This is done by clicking the padlock icon on the corresponding nodes, which freezes their cell selection and keeps them fixed at their original position, regardless of other cells being moved.

Therefore, we identify and select the nodes containing the column headers—specifically, the years 2024 to 2049—and lock their selection using the padlock button. By shifting the selection downward and duplicating it, we can easily move and duplicate the cell selections for the subtables below. By setting the number of repetitions to 2, all subtables are completely selected.

## 4.4 Path-wise Broadcasting

Figure 6 shows the selection hierarchy that users create on the spreadsheet. For readability, the example restricts itself to the first three columns and rows of the leftmost sub-table.

To produce a flat, relational table, the SDE performs *path-wise broadcasting*: for each root-to-leaf path  $P = (v_0, \dots, v_k)$  that ends in a list of  $N$  value cells  $(x_1, \dots, x_N)$ , the SDE materializes  $N$  output rows by repeating or aligning the labels found along the path.

Concretely:

- (1) **Broadcast singletons.** If a path node carries a single label (e.g., *Nursing Staff*, *Total*, or a single year), that label is replicated  $N$  times so each value cell inherits it.

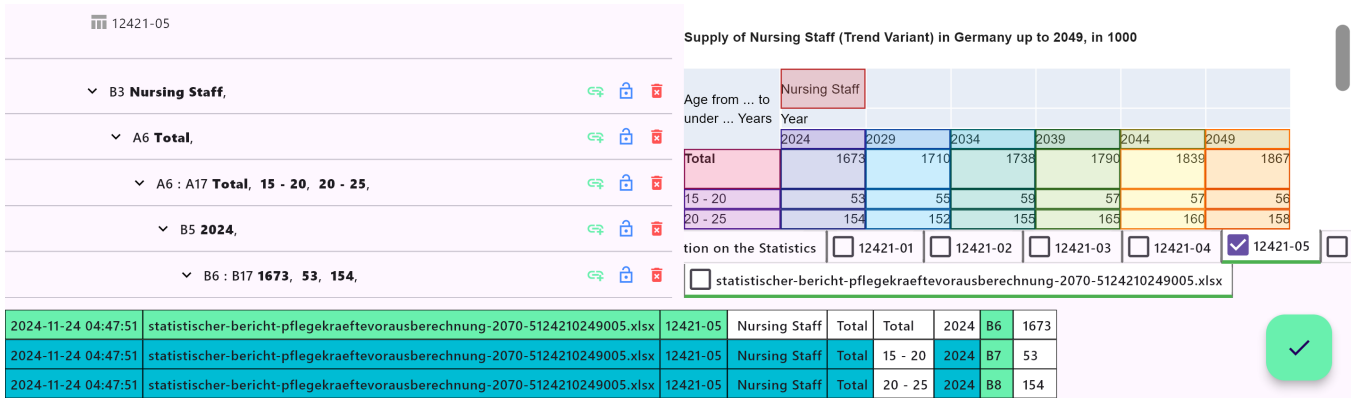


Figure 4: Resulting Hierarchy using the “Move and Duplicate” Function to Replicate the Column Selection Across Years



[To the table of contents](#)

Supply of Nursing Staff (Trend Variant) in Germany up to 2049, in 1000

Age from ... to under ... Years	Nursing Staff					
Year	2024	2029	2034	2039	2044	2049
Total	1673	1710	1738	1790	1839	1867
15 - 20	53	55	59	57	57	56
20 - 25	154	152	155	165	160	158
25 - 30	173	174	168	170	181	175
30 - 35	178	186	184	178	180	191
35 - 40	187	192	198	196	190	192
40 - 45	173	201	204	209	207	201
45 - 50	170	190	218	220	226	222
50 - 55	177	177	198	225	228	233
55 - 60	215	177	177	197	225	227
60 - 65	166	170	139	140	156	178
65 - 70	26	36	37	31	31	34
Male	284	304	321	339	356	368
15 - 20	12	13	14	13	13	13
20 - 25	32	32	32	34	33	33
25 - 30	39	40	39	39	41	40
30 - 35	39	43	43	42	42	45
35 - 40	38	42	46	46	44	45
40 - 45	27	35	38	42	42	41
45 - 50	24	26	35	38	41	41
50 - 55	26	25	27	36	39	42
55 - 60	25	25	25	27	35	39
60 - 65	18	18	18	18	19	25
65 - 70	/	/	/	/	/	/
Female	1390	1406	1416	1451	1484	1499

Figure 5: Selection of All Cells in the Subtables by Duplicating the Hierarchy of the First Table

- (2) **Align equal-length lists.** If a path node provides a list of  $N$  labels, those labels are paired index-wise with the  $N$  value cells.
- (3) **Emit one tuple per value cell.** For each  $j \in \{1, \dots, N\}$ , emit a row that contains the labels gathered from the path

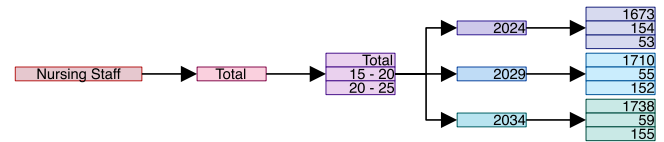


Figure 6: Selection hierarchy before path-wise broadcasting.

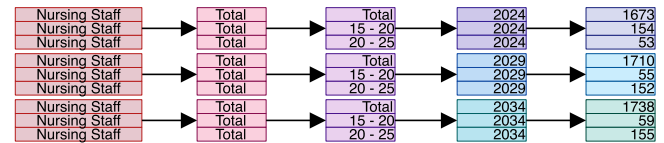


Figure 7: Selection hierarchy after path-wise broadcasting. Each value cell is paired with its repeated/aligned upstream context, producing one row per cell.

at position  $j$  (broadcast or aligned) together with the value  $x_j$ .

The resulting hierarchy after broadcasting is illustrated in Figure 7: upstream labels are repeated or aligned so that each numeric cell is paired with its full context, yielding one relational row per cell.

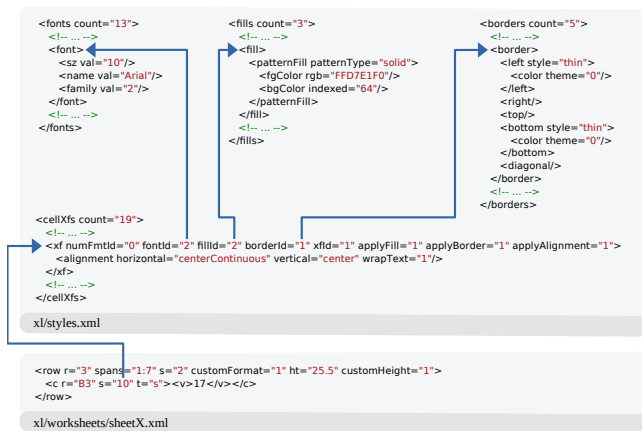
## 5 Interface Fidelity for Navigation

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.

**5.0.1 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 defined in units that do not directly correspond to pixels, requiring conversion for precise on-screen rendering. Moreover, different scaling factors are





**Figure 8: Illustration of the relationship between style definitions in `xl/styles.xml` (fonts, fills, borders, and `cellXfs`) and their application in a worksheet file (`xl/worksheet/s/sheetX.xml`).**

applied for columns and rows. Despite extensive research, we were unable to find official documentation that explains the rationale behind these specific scaling factors. Based on empirical testing, we derived the following scaling factors:

- **Column Widths:** Multiply the *width* attribute by 7.
- **Row Heights:** Multiply the *ht* attribute by  $\frac{4}{3}$ .

**5.0.2 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 as shown in Figure 8.

Each cell in the worksheet references a style index through the *s* attribute, which points to the corresponding `<xf>` 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.

## 6 Scalable Parsing, Indexing, and Rendering

This section describes how the SDE achieves interactive performance on large or bloated worksheets: (i) *incremental loading* of XLSX assets, (ii) a *byte-level worksheet parser* that avoids DOMs and regex, (iii) compact *indexes* for merged regions and column geometry, (iv) *on-demand* streaming of rows and cells, and (v) *viewport-bounded* rendering.

### 6.1 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 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 the `sharedStrings.xml` and the `styles.xml` files 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.

paratagging=false tag=Sect

---

**Algorithm 1:** Backward search for `</sheetData>`

---

**Input:**  $b$ : bytes;  $[lo, hi]$ : search window  
**Output:** `sheetDataCloseByte` or  $-1$   
**Function** `FindSheetDataEndBackward`( $b, lo, hi$ ):  
     $pat \leftarrow$  bytes of `</sheetData>`  
     $m \leftarrow |pat|$   
    **for**  $i \leftarrow hi - m$  **downto**  $lo$  **do**  
        **if**  $b[i] = pat[0]$  **and**  $b[i + m - 1] = pat[m - 1]$  **and**  
            `EqualAt`( $b, i, pat$ ) **then**  
                **return**  $i$   
    **return**  $-1$

---

paratag-  
ging=true

## 6.2 Parsing Worksheet XML with Byte-Level Pattern Matching

DOM-based parsers and regex-on-strings do not scale for very large worksheets: they require full decoding to UTF-16/UTF-8 and materialize enormous trees. In Dart, regular expressions cannot operate on byte arrays, so converting a gigabyte-scale `Uint8List` to a string alone can cost seconds. The SDE therefore parses *directly* on bytes, matching ASCII tag sentinels and reading attributes in place, decoding strings only on demand.

*Parsing roadmap.* Excel worksheets expose a stable top-level order: `<sheetFormatPr>`, `<cols>`, `<sheetData>`, `<mergeCells>`. We first anchor the end of `<sheetData>` by a backward byte search; then we parse metadata around it:

- `<mergeCells>` (after `</sheetData>`),
- `<sheetFormatPr>` (before `<sheetData>`) and
- `<cols>` (before `<sheetData>`)

and enter *sheet-data mode* only when rows or cells are actually needed.

*Anchoring & metadata.* We find the closing `</sheetData>` sentinel by scanning backward and validating the 12-byte pattern (Alg. 1). This yields byte indices that bound all subsequent searches and lets us enumerate `<mergeCell ... ref="A1:B3">` elements linearly, converting each A1 pair to  $(r, c)$  and inserting the span into a compact index with binary-search probes and a prefix-maximum (used later by `spanAt`, Alg. 4).

*Defaults and column bands.* We record the default row height  $H_d$  (attribute `defaultRowHeight`) and the default column width  $W_d$  (attribute `defaultColWidth`) in the `<sheetFormatPr ...>` node. From `<cols>` we parse each element of the form `<col min="i" max="j" width="w">`, which defines a *column band*  $[i:j]$  with width  $w$ . Bands are stored in ascending order of  $\min$ ; queries such as retrieving the width at column  $c$  or the column at a given horizontal offset are answered via a linear or  $O(\log B)$  search over these bands.

*Streaming rows and accumulating offsets.* Entering sheet-data mode, we stream `<row ...>` tags without decoding payloads. Within each opening tag we read  $r = \dots$  (row index) and  $ht = \dots$  (height) if present. For each discovered row we cache its byte interval and compute its top offset incrementally using explicit heights or the default  $H_d$ :

$$\text{off}_1 = (r_1 - 1) H_d, \quad \text{off}_i = \text{off}_{i-1} + (h_{i-1} \text{ or } H_d) + (r_i - r_{i-1} - 1) H_d.$$

---

**Algorithm 2:** Stream rows & compute offsets

---

**Input:**  $b$ : bytes; window  $[o, c]$  with  $o$  = index after `<sheetData...>`,  $c$  = index of `</sheetData>`; default height  $H_d$ ; pixel scale  $\rho$   
**Output:** sequence of  $\langle r, [s, e], h, \text{off} \rangle$   
**Function** `RowsWithOffsets`( $b, [o, c], H_d, \rho$ ):  
     $i \leftarrow o$ ;  $\text{prevR} \leftarrow \perp$ ;  $\text{prevH} \leftarrow \perp$ ;  $\text{prevOff} \leftarrow 0$ ;  $\text{prevS} \leftarrow \perp$   
    **while**  $i \leq c - 4$  **do**  
        **if**  $b[i..i+3] = \text{<row>}$  **then**  
             $s \leftarrow i$   
             $(r, h, j) \leftarrow \text{PARSEROWATTRS}(b, i, c)$  // advances to just after `>`  
            **if**  $\text{prevR} = \perp$  **then**  
                 $\text{off} \leftarrow \rho \cdot (r - 1) H_d$   
            **else**  
                 $g \leftarrow r - \text{prevR} - 1$   
                 $H_{\text{prev}} \leftarrow (\text{prevH} \text{ or } H_d)$   
                 $\text{off} \leftarrow \text{prevOff} + \rho \cdot H_{\text{prev}} + \rho \cdot g H_d$   
            **if**  $\text{prevR} \neq \perp$  **then** **emit**  $\langle \text{prevR}, [\text{prevS}, s], \text{prevH}, \text{prevOff} \rangle$   
             $\text{prevR} \leftarrow r$ ;  $\text{prevH} \leftarrow h$ ;  $\text{prevOff} \leftarrow \text{off}$ ;  $\text{prevS} \leftarrow s$   
             $i \leftarrow j$   
        **else**  
             $i \leftarrow i + 1$   
    **if**  $\text{prevR} \neq \perp$  **then** **emit**  $\langle \text{prevR}, [\text{prevS}, c], \text{prevH}, \text{prevOff} \rangle$

**Input:**  $b$ : bytes;  $i$ : index at `<row>`;  $c$ : close bound (sheet end)  
**Output:**  $(r, h, j)$ : row index  $r$  (or  $\perp$ ), optional height  $h$  (or  $\perp$ ), and  $j$  = first byte after `>`

**Function** `ParseRowAttrs`( $b, i, c$ ):  
     $r \leftarrow \perp$ ;  $h \leftarrow \perp$ ;  $j \leftarrow i + 4$   
    **while**  $j < c$  **do**  
        **if**  $b[j] = \text{>}$  **then**  
             $j \leftarrow j + 1$ ; **return**  $(r, h, j)$   
        **if**  $b[j] = r$  **and**  $b[j - 1]$  is space **then**  
             $k \leftarrow j + 1$   
             $(s, e) \leftarrow \text{GETINNERATTRINTERVAL}(k, c, b)$   
            **if**  $(s, e) \neq \perp$  **then**  $r \leftarrow \text{PARSEINTASCII}(b, s, e)$   
        **if**  $b[j..j+1] = \text{ht}$  **and**  $b[j - 1]$  is space **then**  
             $k \leftarrow j + 2$   
             $(s, e) \leftarrow \text{GETINNERATTRINTERVAL}(k, c, b)$   
            **if**  $(s, e) \neq \perp$  **then**  $h \leftarrow \text{PARSEDDOUBLEASCII}(b, s, e)$   
         $j \leftarrow j + 1$   
    **return**  $(r, h, j)$  // malformed tail safely falls through

**Input:**  $i$ : scan index after the attribute name;  $n$ : hard bound;  $b$ : bytes

**Output:**  $(s, e)$  inner half-open interval, or  $\perp$  if not well-formed

**Function** `GetInnerAttrInterval`( $i, n, b$ ):  
    **while**  $i < n$  **and**  $b[i]$  is space **do**  
         $i \leftarrow i + 1$   
    **if**  $i < n$  **and**  $b[i] = \text{=}$  **then**  
         $i \leftarrow i + 1$   
        **while**  $i < n$  **and**  $b[i]$  is space **do**  
             $i \leftarrow i + 1$   
        **if**  $i < n$  **and**  $(b[i] = \text{"} \text{ or } b[i] = \text{'})$  **then**  
             $q \leftarrow b[i]$ ;  $i \leftarrow i + 1$ ;  $s \leftarrow i$   
            **while**  $i < n$  **and**  $b[i] \neq q$  **do**  
                 $i \leftarrow i + 1$   
            **return**  $(s, i)$   
    **return**  $\perp$

---

paratag-  
ging=true

See Alg. 2.

paratagging=false tag=Sect



*Lazy cell parsing.* Given a row byte interval  $[s, e]$ , cells are parsed on demand. We scan for  $\langle c$ , read attributes ( $r = \text{"A123"}$ ,  $s = \text{"..."}\text{"}$ ,  $t = \text{"..."}\text{"}$ ), and bound the cell interval by the next  $\langle c$  or the row end. Values are extracted *within* this interval from  $\langle v \rangle \dots \langle /v \rangle$  or (for inline strings)  $\langle is \rangle \dots \langle /is \rangle$  (See Alg. 3). Because worksheet XML produced by Microsoft Excel uses strictly increasing row indices, a violation of this order is treated as malformed input and repaired on load. Consequently, during streaming we can safely terminate parsing once the requested row index is exceeded.

paratagging=false tag=Sect

---

### Algorithm 3: Resolve cell by streaming the row

---

**Input:**  $b$ : bytes; row interval  $[S, E]$ ; target column  $c$

**Output:** cell interval  $[s, e]$  or  $\perp$

```

i ← S
while i ≤ E - 2 do
  if b[i..i + 1] = <c then
    s ← i
    (c', j) ← PARSECELLCOL(b, i, E) // reads r="A123" and
    advances to just after >
    // end of this cell = next <c or E
    k ← j
    while k ≤ E - 2 and b[k..k + 1] ≠ <c do
      k ← k + 1
    e ← (k ≤ E - 2) ? k : E
    if c' = c then
      return [s, e]
    i ← e
  else
    i ← i + 1
return ⊥

```

---

paratag-  
ging=true

*Merged regions.* Merged areas are rectangles  $[r_1, r_2] \times [c_1, c_2]$ . We normalize and sort spans by  $(r_1, c_1)$ , store parallel arrays  $R_1, C_1, R_2, C_2$ , and build a prefix-maximum PM over  $R_2$ . A point query  $\text{spanAt}(r, c)$  uses (i) a binary search on  $R_1$  to cap candidates above  $r$ , (ii) a binary search over PM to drop candidates ending before  $r$ , (iii) a binary search on  $C_1$  to find those with  $c_1 \leq c$ , then a short local check; an origin map answers “is this cell the origin?” in  $O(1)$  (See Alg. 4).

*Scroll extents (without decoding payloads).* We obtain  $r_{\max}$  by scanning backward to the last  $\langle \text{row} \dots r = \text{"..."} \text{"} \rangle$ . A single forward pass accumulates  $\sum_{r \in E} \text{ht}(r)$  and counts  $|E|$ ; all other rows use  $H_d$ . In device units,

$$H_{\text{sheet}} = \sum_{r \in E} \text{ht}(r) + (r_{\max} - |E|) H_d.$$

Horizontally, letting  $c_{\max}$  be the largest covered column,

$$W_{\text{sheet}} = \text{colOff}(c_{\max}) + w(c_{\max}).$$

These extents parameterize the viewport and drive scrollbar sizing in the UI.

## 6.3 Two-Dimensional Grid Viewport Rendering

Given a (potentially very large) worksheet with variable row heights, banded widths, and merged regions, we lay out only tiles

---

### Algorithm 4: Viewport layout with merge-aware origin-first placement

---

**Input:**  $x_0, y_0$ ;  $W_{\text{vp}}, H_{\text{vp}}$ ; cache  $\Delta$ ; sheet accessors

$\text{rowIndexAt}, \text{colIndexAt}, \text{off}, \text{colOff}, h, w$ ; merge index  $\text{spanAt}, \text{isOrigin}$

**Output:** positioned tiles for current frame

// Visible indices

$r_t \leftarrow \lfloor \text{rowIndexAt}(y_0) \rfloor, r_u \leftarrow \lfloor \text{rowIndexAt}(y_0 + H_{\text{vp}} + \Delta) \rfloor$

$c_l \leftarrow \lfloor \text{colIndexAt}(x_0) \rfloor, c_u \leftarrow \lfloor \text{colIndexAt}(x_0 + W_{\text{vp}} + \Delta) \rfloor$

$B \leftarrow \emptyset$  // burned merged spans

// Pass 1: top border probes

**for**  $c \leftarrow c_l$  **to**  $c_u$  **do**

$S \leftarrow \text{spanAt}(r_t, c)$

**if**  $S \neq \perp$  **and**  $S \notin B$  **and**  $r_s < r_t$  **then**

        place tile for  $S$  at  $(\text{colOff}(c_s) - x_0, \text{off}(r_s) - y_0)$

$B \leftarrow B \cup \{S\}$

// Pass 2: left border probes

**for**  $r \leftarrow r_t$  **to**  $r_u$  **do**

$S \leftarrow \text{spanAt}(r, c_l)$

**if**  $S \neq \perp$  **and**  $S \notin B$  **and**  $c_s < c_l$  **then**

        place tile for  $S$  at  $(\text{colOff}(c_s) - x_0, \text{off}(r_s) - y_0)$

$B \leftarrow B \cup \{S\}$

// Pass 3: interior tiles

**for**  $c \leftarrow c_l$  **to**  $c_u$  **do**

$x \leftarrow \text{colOff}(c) - x_0$

**for**  $r \leftarrow r_t$  **to**  $r_u$  **do**

$y \leftarrow \text{off}(r) - y_0$

$S \leftarrow \text{spanAt}(r, c)$

**if**  $S \neq \perp$  **or**  $\text{isOrigin}(S, (r, c))$  **then**

            place tile at  $(x, y)$

//  $\text{ub}(A, x) = \min\{i \mid A[i] > x\}$ ,  $\text{lb}(A, x) = \min\{i \mid A[i] \geq x\}$ .

**Function**  $\text{spanAt}(r, c)$ :

$hi \leftarrow \text{ub}(R_1, r)$

**if**  $hi = 0$  **then**

        return  $\perp$

$lo \leftarrow \text{lb}(\text{PM}[0:hi], r)$

**if**  $lo \geq hi$  **then**

        return  $\perp$

$k \leftarrow \text{ub}(C_1[lo:hi], c) - 1$

**if**  $k < lo$  **then**

        return  $\perp$

**for**  $i \leftarrow k$  **downto**  $lo$  **do**

**if**  $C_1[i] > c$  **then**

            break

**if**  $r \leq R_2[i]$  **and**  $c \leq C_2[i]$  **then**

            return  $\text{span } i$

    return  $\perp$

---

paratag-  
ging=true

intersecting the current viewport. The renderer works in device pixels but derives all positions from byte-level parser. Our goal is *frame-local* work proportional to the of *visible* rows/columns, independent of sheet size. Algorithm 4 summarizes the procedure. Throughout, index intervals are half-open  $[a, b)$ .

paratagging=false tag=Sect

*Inputs.* Let  $x_0, y_0$  be the horizontal/vertical scroll offsets (device pixels) and  $W_{\text{vp}}, H_{\text{vp}}$  the viewport size. We use a small *cache extent*  $\Delta > 0$  to pre-build tiles that will imminently enter view, rendering over  $[x_0, x_0 + W_{\text{vp}} + \Delta) \times [y_0, y_0 + H_{\text{vp}} + \Delta)$ . Row geometry comes from the streaming parser: each row  $r$  has a top offset  $\text{off}(r)$  and a height  $h(r)$  (explicit  $\text{ht}$  if present, otherwise the default). Columns

are given as ordered bands; for column  $c$  we can query its width  $w(c)$  and cumulative offset  $\text{colOff}(c)$ . Merged regions are indexed by a structure that decides, in logarithmic time, whether a coordinate  $(r, c)$  is covered and, if so, by which span.

*Visible set.* We invert the cumulative-height/width functions:

$$r_\ell = \lfloor \text{rowIndexAt}(y_0) \rfloor, \quad r_u = \lceil \text{rowIndexAt}(y_0 + H_{vp} + \Delta) \rceil,$$

$$c_\ell = \lfloor \text{colIndexAt}(x_0) \rfloor, \quad c_u = \lceil \text{colIndexAt}(x_0 + W_{vp} + \Delta) \rceil.$$

Here  $\text{rowIndexAt}(y)$  and  $\text{colIndexAt}(x)$  are binary searches over cumulative extents built from parsed row heights and column bands, yielding  $O(\log R)$  and  $O(\log B)$  lookup time, respectively.

*Cell coordinates and merge spans.* A merged region is a closed rectangle  $[r_1, r_2] \times [c_1, c_2]$  with  $r_1 \leq r_2$  and  $c_1 \leq c_2$ . We sort spans by origin  $(r_1, c_1)$  and materialize parallel arrays  $R_1, C_1, R_2, C_2$  plus a prefix-maximum array  $\text{PM}[i] = \max_{0 \leq j \leq i} R_2[j]$ . An origin map supports  $O(1)$  checks that  $(r, c)$  is the top-left of a span. Membership “is  $(r, c)$  covered?” runs in three bounded steps:

- (1) **Row window (binary search).**  $hi = \text{ub}(R_1, r)$ ; candidates lie in  $[0, hi]$ . Then  $lo = \text{lb}(\text{PM}[0:hi], r)$  discards all indices with  $R_2 < r$ .
- (2) **Column narrowing (binary search).**  $k = \text{ub}(C_1[lo:hi], c) - 1$  is the last origin with  $c_1 \leq c$ .
- (3) **Local check (constant expected).** Scan left from  $k$  while  $C_1[i] \leq c$ ; accept if  $r \leq R_2[i]$  and  $c \leq C_2[i]$ .

We use  $\text{ub}(A, x) = \min\{i \mid A[i] > x\}$  and  $\text{lb}(A, x) = \min\{i \mid A[i] \geq x\}$ . This yields  $O(\log M + \alpha)$  time, where  $\alpha$  is a short local scan in practice.

*Origin-first placement for merged regions.* A naïve scan of  $[r_\ell : r_u] \times [c_\ell : c_u]$  would instantiate merged tiles multiple times. Instead, we *pre-place* only spans whose origins lie outside the leading edges but whose rectangles intersect the viewport: probe the top border  $(r_\ell, c_\ell : c_u)$  and the left border  $(r_\ell : r_u, c_\ell)$ , query  $\text{spanAt}(r, c)$ , and place the tile once at its origin  $(r_s, c_s)$ . We maintain a burned set  $B$  to avoid duplicates in the interior.

*Interior tiling.* Traverse the visible grid and place a tile at  $(r, c)$  only if (i) no span covers it, or (ii)  $(r, c)$  is the origin of its span (checked in  $O(1)$ ). Device positions are  $x = \text{colOff}(c) - x_0$ ,  $y = \text{off}(r) - y_0$ .

*Scroll extents.* We size the vertical scroll domain from row attributes without decoding payloads. Anchored at  $\langle \text{sheetData} \rangle$ , we scan backward to the last  $\langle \text{row} \dots \rangle$  to read  $r_{\max}$  from  $r = \dots$ . A single forward pass accumulates explicit heights  $\sum_{r \in E} \text{ht}(r)$  and counts them as  $|E|$ ; all other rows in  $1..r_{\max}$  use the default  $H_d$ . With device pixels,

$$H_{\text{sheet}} = \sum_{r \in E} \text{ht}(r) + (r_{\max} - |E|) H_d.$$

Horizontally, with  $c_{\max}$  the highest covered column,

$$W_{\text{sheet}} = \text{colOff}(c_{\max}) + w(c_{\max}).$$

These extents parameterize the viewport and drive scrollbar sizing/positioning (*two\_dimensional\_scrollables* [14]).

paratagging=false tag=Sect

---

#### Algorithm 5: Lazy row locator: driver and lookup

---

**Input:** hierarchy files  $\rightarrow$  tasks  $\rightarrow$  sheets  $\rightarrow$  cellTasks; caps  $B_{\max}, R_{\max}$   
**Output:** on demand: for global row  $r$ , return  $(\text{RowBlock}, \text{local})$  or  $\perp$ .  
**State:** arrays  $\text{blocks}$ ,  $\text{starts}$ ; queue  $Q$ ; indices  $fi, sti, si$ ;  $\text{curFile}, \text{curTask}$ ,  $\text{curSheets}$ ;  $(\text{activeWb}, \text{activeSheet})$ .

**Function**  $\text{EnsureBuiltThroughRow}(r)$ :

```

while builtRows ≤ r do
  rb ← NextBlock()
  if rb = ⊥ then
    break
  append rb to blocks; append rb.startRow to starts;
  PruneHeadIfNeeded()

```

**Function**  $\text{LocateRow}(r)$ :

```

EnsureBuiltThroughRow(r)
if blocks = ∅ then
  return ⊥
lo ← 0; hi ← |starts|
while lo < hi do
  mid ← (lo + hi) / 2
  if starts[mid] ≤ r then
    lo ← mid + 1
  else
    hi ← mid
idx ← lo - 1
if idx < 0 then
  return ⊥
b ← blocks[idx]; local ← r - b.startRow
if 0 ≤ local < b.len then
  return (b, local)
else
  return ⊥

```

---

paratag-  
ging=true

## 6.4 Lazy Output Flattening and Row Location

Large workbooks and selection hierarchies make fully materializing a relational view prohibitively expensive. Empirically, naive flattening leads to seconds or minutes of UI stalls on sheets with up to  $10^6$  rows (Excel’s limit) and many columns. Our goal is to *render only what the user is about to see*, without precomputing the entire output.

*Design overview.* We model the visible table as a single *global row space* obtained by concatenating many small, contiguous *row blocks*. Each block is produced by streaming the hierarchy files  $\rightarrow$  tasks  $\rightarrow$  sheets  $\rightarrow$  cellTasks in a deterministic order. The UI asks for a particular global row index  $r$  (e.g., the first row currently in the viewport); we then extend the stream just far enough so that  $r$  falls inside a cached block. This *on-demand flattening* avoids touching unrelated parts of the hierarchy.

*Data structures.* We maintain two parallel arrays: (i) *blocks* contains the realized row blocks (each stores *startRow*, *len*, *path*, *cells*, and the active workbook/sheet), and (ii) *starts* stores the corresponding *startRow* values. By construction, *starts* is strictly increasing. A FIFO queue  $Q$  holds pending (task, path) frames for a breadth-first streaming traversal. Traversal indices over files/sheet-tasks/sheets maintain the current context (*activeWb*, *activeSheet*). Optional caps  $B_{\max}$  and  $R_{\max}$  bound memory by pruning the oldest blocks.

*Driver and lookup.* When the UI requests row  $r$ ,  $\text{ENSUREBUILTTHROUGHROW}$  extends the stream until *builtRows*  $> r$  or the source

is exhausted (Alg. 5). We then locate the block via a binary search over *starts* (*upper bound* on *r* and step one back), yielding the block index and the local in-block offset (Alg. 5, LOCATEROW). Both steps are  $O(k)$  work to extend by *k* newly discovered rows plus  $O(\log |starts|)$  for the lookup.

paratagging=falsetag=Sect

---

#### Algorithm 6: Row block generation: NextBlock

---

```

Function NEXTBLOCK:
  if  $\neg \text{FillQueueIfEmpty}()$  then
     $\perp$  return  $\perp$ 
  while true do
    if  $Q = \emptyset$  then
      if  $\neg \text{FillQueueIfEmpty}()$  then
         $\perp$  return  $\perp$ 
      continue
    (t, path)  $\leftarrow$  pop-front Q
    if t.children =  $\emptyset$  then
      cells  $\leftarrow$  t.sortedSelectedCells
      if  $|cells| = 0$  then
         $\perp$  continue
      start  $\leftarrow$  builtRows; len  $\leftarrow$   $|cells|$ 
      return RowBlock(activeWb, activeSheet, path, cells, start, len)
    nextPath  $\leftarrow$  path  $\cup$  {t}
    foreach u  $\in$  t.children do
       $\perp$  push-back (u, nextPath) into Q

Function FILLQUEUEIFEMPTY:
  while  $Q = \emptyset$  do
    if curTask =  $\perp$  then
      if  $fi \geq |files|$  then
         $\perp$  return false
      curFile  $\leftarrow$  files[fi + +]; sti  $\leftarrow$  0
    if  $sti \geq |curFile.sheetTasks|$  then
       $\perp$  curTask  $\leftarrow$   $\perp$ ; continue
    curTask  $\leftarrow$  curFile.sheetTasks[sti + +]; curSheets  $\leftarrow$ 
    list(curTask.sheets); si  $\leftarrow$  0
    while si <  $|curSheets|$  and  $Q = \emptyset$  do
      (wb, sh)  $\leftarrow$  curSheets[si + +];
      roots  $\leftarrow$  curTask.importExcelCellsTasks
      if  $|roots| = 0$  then
         $\perp$  continue
      Q  $\leftarrow$  queue of (root,  $\emptyset$ ) for each root  $\in$  roots
      activeWb  $\leftarrow$  wb; activeSheet  $\leftarrow$  sh
    if  $Q \neq \emptyset$  then
       $\perp$  return true
  return true

```

paratag-  
ging=true

*Streaming next block.* NEXTBLOCK (Alg. 6) emits the next contiguous block. If *Q* is empty we call FILLQUEUEIFEMPTY to advance to the next *source segment* (file, sheet task, sheet pair) with non-empty roots. Otherwise we pop a frame. If the popped task has no children, its sortedSelectedCells form a new RowBlock (*start* = *builtRows*, *length* = number of selected cells) and we return it. If the task has children, we enqueue each child with the extended path. This BFS over the selection tree yields a deterministic, top-down order that matches the user’s mental model. FILLQUEUEIFEMPTY advances through the outer hierarchy: it steps files (*fi*), sheet tasks (*sti*), then sheet pairs (*si*). For each sheet pair (*wb*, *sh*) it materializes the roots (importExcelCellsTasks) into *Q* and sets the active context used to form RowBlocks. If a segment has no roots, it is

---

#### Algorithm 7: Cache pruning (part D): PruneHeadIfNeeded

---

```

Function PRUNEHEADIFNEEDED:
  rm  $\leftarrow$  0
  if  $B_{\max} \neq \perp$  and  $|blocks| > B_{\max}$  then
     $\perp$  rm  $\leftarrow$   $|blocks| - B_{\max}$ 
  if  $R_{\max} \neq \perp$  then
    while rm <  $|blocks|$  do
      cached  $\leftarrow$  blocks[ $|blocks| - 1$ ].startRow + blocks[ $|blocks| - 1$ ].len - blocks[0].startRow
      if cached  $\leq R_{\max}$  then
         $\perp$  break
       $\perp$  rm  $\leftarrow$  rm + 1
  if rm > 0 then
     $\perp$  drop first rm from blocks and starts

```

paratag-  
ging=true

skipped. When the last file is exhausted and *Q* remains empty, the function returns **false** and the stream is complete.

*Cache pruning.* PRUNEHEADIFNEEDED (Alg. 7) trims the front of the cache to respect either (i) a block count cap  $B_{\max}$ , or (ii) a total cached row cap  $R_{\max}$ . Trimming removes the oldest blocks and their starts while preserving the invariant that *starts* remains strictly increasing and aligned with blocks. In practice we choose small caps that comfortably cover one or two viewport heights plus prefetch.

paratagging=falsetag=Sect

*UI integration.* On each frame the viewport asks for the *leading* global row (top-left visible), calls LOCATEROW to obtain the block and local offset, and renders from there. We prefetch just beyond the viewport by requesting the row at  $y_0 + H_{vp} + \Delta$ , which amortizes the streaming cost without over-allocating memory. When the user scrolls back, previously cached blocks are reused; otherwise the binary search still runs in logarithmic time.

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.

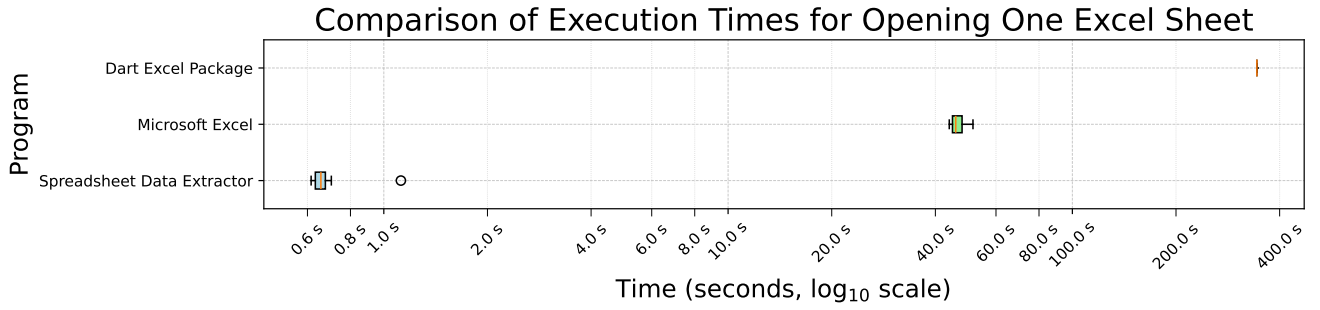
## 7 Evaluation

We evaluate the *Spreadsheet Data Extractor (SDE)* along two primary dimensions: *user experience* and *latency*. The evaluation focuses on interactive worksheet rendering, workflow integration, and responsiveness when operating on large and irregular *Excel* workbooks.

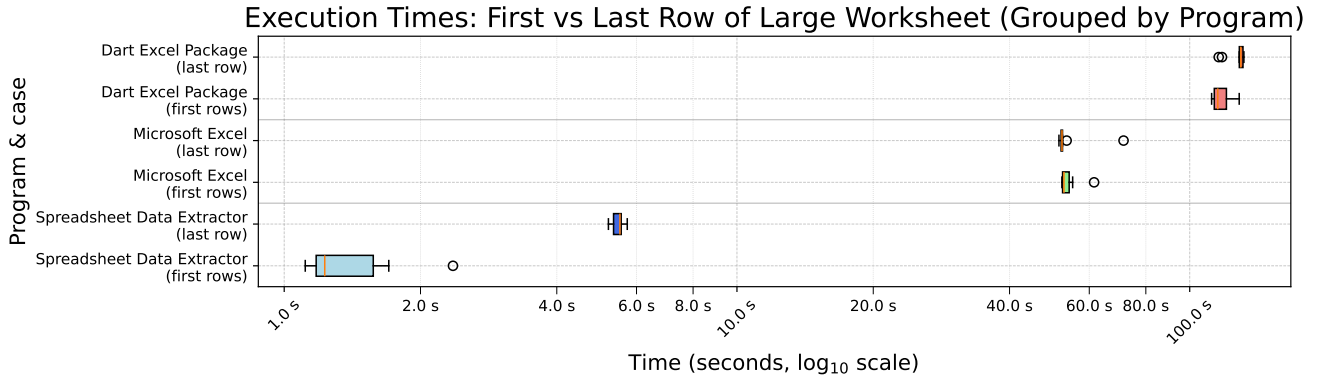
### 7.1 Baseline User Effort from Prior Work

To provide context for user effort, we report baseline measurements from earlier work that evaluated a predecessor system on large, real-world datasets [2]. In that study, student assistants extracted data from over 500 *Excel* files. Processing times were analysed for a subset of 331 workbooks comprising 3,093 worksheets, with pauses removed as outliers.

On average, users required 15 minutes per workbook, corresponding to 95 seconds per worksheet. The distribution ranged from 4 minutes at the 25% quantile to 18 minutes at the 75% quantile, with a median of 7 minutes per file. These results reflect manual



(a) Large multi-worksheet workbook: opening one selected sheet (user-centred latency).



(b) Single-worksheet benchmark (1,048,548 rows): first vs. last row per programme.

**Figure 9: Time-to-first-visual vs. full-parse costs across tools. (a) shows the practical benefit of initialising only the selected sheet in a multi-worksheet file; (b) isolates algorithmic differences under identical workload. Both plots aggregate 10 runs on a logarithmic time axis.**

interaction overheads, including repeated context switches between worksheet selection, hierarchy definition, and output inspection.

SDE eliminates these sources of interaction overhead by unifying selection, hierarchy construction, and output preview within a single interface. While a direct user study for SDE is left to future work, the redesigned workflow removes previously identified inefficiencies that contributed substantially to end-to-end extraction effort.

## 7.2 Acceleration When Opening Files

We quantify opening latency in two complementary scenarios: (i) a *user-centred* case that measures time-to-first-visual when opening a *selected worksheet* from a large, multi-worksheet workbook [5]; and (ii) a *workload-equivalent* case that uses a single worksheet with 1,048,548 rows to align the amount of work across programmes (first rows vs. last row) [3]. Throughout, we report medians over repeated runs and show distributions as box plots on a  $\log_{10}$  time axis.

*Setup.* We evaluate three programmes: the *SDE*, Microsoft Excel (automated via PowerShell/COM), and the Dart *excel* package used in an earlier prototype [2]. Each condition was repeated ten times *unless noted otherwise*.

*User-centred latency: opening a selected worksheet.* Opening a selected worksheet from a large real-world workbook [19], SDE initialises only the requested sheet and achieves a median of **0.657 s**, compared to **45.843 s** for Excel—a **69.8×** speedup (Fig. 9a). The Dart *excel* package completed only the first run at 343.568 s; runs 2–10 terminated with out-of-memory on our test machine ( $n=1$ ). Based on that single run, SDE is **522.9×** faster in this scenario.

*User-centred latency: opening a selected worksheet.* Opening a selected worksheet from a large real-world workbook [19], SDE initialises only the requested sheet and achieves a median of **0.657 s**, compared to **45.843 s** for Excel—a **69.8×** speedup (Fig. 9a). The Dart *excel* package completed only the first run at 343.568 s; runs 2–10 terminated with out-of-memory on our test machine ( $n=1$ ). This behaviour reflects the limitations of DOM-based XML parsing when processing large worksheets. Based on that single successful run, SDE is **522.9×** faster in this scenario.

*Controlling for workload comparability.* The multi-worksheet experiment reflects an important user scenario (time-to-first-visual for a selected sheet), but it is not a like-for-like comparison: SDE initialises only the requested worksheet, whereas the baselines initialise the entire workbook. To control for this confound and establish workload equivalence, we construct a *single-worksheet*

benchmark from the example in Section 3 (Design Philosophy) by pruning the workbook to one sheet and duplicating the table vertically until no additional table fits, yielding 1,048,548 rows of real data. The resulting file is  $\sim 35$  MB on disk (XLSX ZIP container) and  $\sim 282$  MB uncompressed. We then report two cases—*first rows* (time-to-first-visual) and *last row* (forces a full parse)—to align SDE’s measured cost with eager parsers. Under these identical I/O and parsing conditions, SDE achieves **1.229 s** vs. **52.756 s** for Excel on first rows (**42.9×**), and **5.515 s** vs. **52.228 s** on the last row (**9.47×**) (Fig. 9b).

*Benchmarking environment.* All tests were conducted on a machine with an AMD Ryzen 5 PRO 7535U with Radeon Graphics

(6-core CPU at 2.9 GHz), 31.3 GB RAM, running Microsoft Windows 11 Enterprise . The version of Microsoft Excel used was version 16.0.

## 8 Conclusion

We presented the *Spreadsheet Data Extractor (SDE)* [4], a system for extracting machine-readable data from semi-structured spreadsheets. SDE enables interactive exploration of large *Excel* workbooks through incremental loading and viewport-bounded rendering. By avoiding DOM-based worksheet parsing and instead processing only the data required for the current user interaction, SDE achieves orders-of-magnitude improvements in user-perceived latency on large real-world spreadsheets.