

1 Spreadsheet Data Extractor (SDE): A Performance-Optimized, 2 User-Centric Tool for Transforming Semi-Structured Excel 3 Spreadsheets into Relational Data

4 Anonymous Author(s) 59
5 Submission Id: 60
6 61
7 62
8 63
9 64
10 65
11 66
12 67
13 68
14 69
15 70
16 71
17 72
18 73
19 74
20 75
21 76
22 77
23 78
24 79
25 80
26 81
27 82
28 83
29 84
30 85
31 86
32 87
33 88
34 89
35 90
36 91
37 92
38 93
39 94
40 95
41 96
42 97
43 98
44 99
45 100
46 101
47 102
48 103
49 104
50 105
51 106
52 107
53 108
54 109
55 110
56 111
57 112
58 113
59 114
60 115
61 116
62 117
63 118
64 119
65 120
66 121
67 122
68 123
69 124
70 125
71 126
72 127
73 128
74 129
75 130
76 131
77 132
78 133
79 134
80 135
81 136
82 137
83 138
84 139
85 140
86 141
87 142
88 143
89 144
90 145
91 146
92 147
93 148
94 149
95 150
96 151
97 152
98 153
99 154
100 155
101 156
102 157
103 158
104 159
105 160
106 161
107 162
108 163
109 164
110 165
111 166
112 167
113 168
114 169
115 170
116 171

Abstract

Spreadsheets are widely used across domains, yet their human-oriented layouts (merged cells, hierarchical headers, multiple tables) hinder automated extraction. We present the Spreadsheet Data Extractor (SDE), a user-in-the-loop system that converts semi-structured sheets into structured outputs without programming. Users declare the structure they perceive; the engine broadcasts selections deterministically and renders results immediately. Under the hood, SDE employs incremental loading, byte-level XML streaming (avoiding DOM materialisation), and viewport-bounded rendering.

Optimised for time-to-first-visual, SDE delivers about **70 \times** speedup over Microsoft Excel when opening a selected sheet in a large real-world workbook; under workload-equivalent conditions (single worksheet, full parse) it remains about **10 \times** faster, while preserving layout fidelity. These results indicate SDE's potential for reliable, scalable extraction across 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.

Keywords

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

ACM Reference Format:

Anonymous Author(s). 2026. Spreadsheet Data Extractor (SDE): A Performance-Optimized, User-Centric Tool for Transforming Semi-Structured Excel Spreadsheets into Relational Data. In *Proceedings of Conference on Human Factors in Computing Systems (ACM CHI '26)*. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/XXXXXXX.XXXXXXX>

50 Permission to make digital or hard copies of all or part of this work for personal or
51 classroom use is granted without fee provided that copies are not made or distributed
52 for profit or commercial advantage and that copies bear this notice and the full citation
53 on the first page. Copyrights for components of this work owned by others than the
54 author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or
55 republish, to post on servers or to redistribute to lists, requires prior specific permission
56 and/or a fee. Request permissions from permissions@acm.org.

57 *ACM CHI '26, Barcelona, Spain*

58 © 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

59 ACM ISBN

60 <https://doi.org/10.1145/XXXXXXX.XXXXXXX>

1 Introduction

Spreadsheets underpin workflows across healthcare [5], nonprofit organizations [16, 21], finance, commerce, academia, and government [10]. Despite their ubiquity, reliably analyzing and reusing the data they contain remains difficult for automated systems. In practice, many real-world spreadsheets are organized for human consumption [20] and therefore exhibit layouts with empty cells, merged regions, hierarchical headers, and multiple stacked tables. While such conventions aid human reading [15], 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, 7]
- **Scalability Issues:** As datasets grow in size and complexity, spreadsheets become less efficient for data storage and retrieval, leading to performance bottlenecks. [14, 19]
- **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.

Given the abundance of spreadsheet data and the clear advantages of database systems, there is a pressing need for tools that bridge these formats. Automated, accurate extraction from spreadsheets into relational representations is essential.

Prior work introduced a tool for extracting data from Excel files [3]. While effective, that system exhibited performance issues on large workbooks and imprecise rendering of cell geometry,

which limited usability for complex, real-world files. The user interface was also fragmented, requiring context switches between hierarchy definition and output preview.

This paper presents the *Spreadsheet Data Extractor (SDE)*, which transforms semi-structured spreadsheet content into machine-readable form. Users declare structure directly via cell selections—no programming required. The design addresses the above limitations and supports large, irregular spreadsheets with predictable, interactive performance.

1.1 Contributions

- (1) **Unified interaction.** We integrate the selection hierarchy, worksheet view, and output preview into a single interface, reducing context switches and lowering the number of required interactions.
- (2) **DOM-free, byte-level worksheet parsing.** We implement a custom parser that operates directly on the XLSX worksheet bytes, avoiding DOM construction and regex over decoded strings. This greatly reduces memory footprint and improves robustness on large/“bloated” sheets.
- (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).
- (4) **Excel-faithful rendering.** We recover row heights, column widths, and merges from XML to render worksheets closely to Excel, including text overflow behavior, which improves user orientation.
- (5) **Viewport-bounded rendering.** The renderer draws only cells intersecting the viewport; selection queries are pruned with cached axis-aligned bounding boxes. Together these keep per-frame work proportional to what is visible rather than to sheet size.

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 Aue et al.’s Converter

Aue et al. [3] developed a tool to facilitate data extraction from Excel spreadsheets by leveraging the Dart *excel* package [8] to process .xlsx files. This tool allows users to define data hierarchies by selecting relevant cells containing data and metadata. However, the approach faced significant performance bottlenecks due to the *excel* package’s requirement to load the entire .xlsx file into memory, resulting in slow response times, particularly for large files.

In addition to memory issues, the tool calculated row heights and column widths based solely on cell content, ignoring the dimensions specified in the original Excel file. This led to rendering discrepancies between the tool and the original spreadsheet. Furthermore, the tool rendered all cells, regardless of their visibility

within the viewport, significantly degrading performance when handling worksheets with large numbers of cells.

2.2 DeExcelerator

Eberius et al. [11] 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. [12] 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. [4] 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. [6] introduced **Senbazuru**, a prototype Spreadsheet Database Management System (SSDBMS) designed to extract relational information from a large corpus of spreadsheets. Senbazuru

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
Aue et al. (converter)	User-centric Selection-Workflow	relational data	no public repository	last publication 2024

Table 1: Spreadsheet Data Extractor counterparts

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. [9] 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

A direct head-to-head comparison was not possible due to the lack of artifacts, because for the UI-oriented systems **FLASHRELATE**, **Senbazuru**, **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

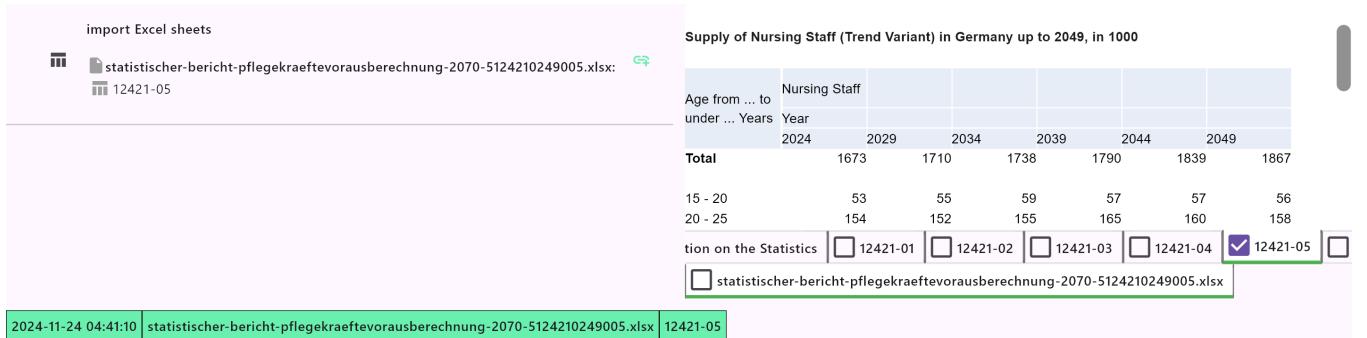


Figure 1: The SDE Interface Overview.

349 over similar structures for reuse, reducing the need for repetitive
 350 configurations.
 351

352 By combining the strengths of manual control with enhanced
 353 user interface features and performance optimizations, our tool
 354 offers a robust and accessible solution for extracting relational data
 355 from complex and visually intricate spreadsheets. These enhance-
 356 ments not only improve performance and accuracy but also elevate
 357 the overall user experience, making our tool a valuable asset for
 358 efficient and reliable data extraction from diverse spreadsheet for-
 359 mats.

3 Design Philosophy

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

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

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

402 *Hierarchy Definition.* Users select individual cells or ranges (click,
 403 Shift-click for multi-selection). Each selection denotes either data
 404 (values) or metadata (labels/headers). Selections are arranged into a
 405 hierarchical tree: each node represents a data element; child nodes

407 represent nested data or metadata. This hierarchy specifies how the
 408 SDE broadcasts selections into rows/columns in the output.
 409

410 The interface supports flexible management: users drag-and-
 411 drop nodes to reparent or reorder the hierarchy when a different
 412 organization is more appropriate. Users can also introduce *custom
 413 nodes* with user-defined text to encode metadata that is implicit in
 414 the spreadsheet but absent from cell contents.

415 *Reusability and Efficiency.* To reduce repetitive work on sheets
 416 with repeated layouts, users can duplicate an existing selection
 417 hierarchy and apply it to similar regions. When moving a hierarchy,
 418 some cells may need to remain fixed (e.g., vertically stacked tables
 419 where only the topmost table repeats header rows). In *DuplicateAndMove*
 420 mode, the SDE provides a *lock* function: users freeze specific cells
 421 while relocating the rest of the hierarchy. Locks can be toggled
 422 via the lock icon at the top-left corner of a cell or next to the cor-
 423 responding selection in the hierarchy panel; locked cells remain
 424 stationary while other selections shift accordingly (see Figure 4
 425 in 4). Already-locked selections are visually indicated and can be
 426 unlocked at any time.

4 Example Workflow

427 Consider a spreadsheet containing statistical forecasts of future
 428 nursing staff availability in Germany [17]. Figure 1 shows the SDE
 429 interface, which consists of three main components:

430 **Hierarchy Panel (Top Left):** Displays the hierarchy of cell
 431 selections, initially empty.

432 **Spreadsheet View (Top Right):** Shows the currently opened
 433 Excel file and the currently selected worksheet for cell selection.

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

4.1 Selection of the First Column

436 The user adds a node to the hierarchy and selects the cell contain-
 437 ing the metadata "Nursing Staff" (Figure 2). This cell represents
 438 metadata that is common to all cells in this worksheet. Therefore, it
 439 should be selected first and should appear at the beginning of each
 440 row in the output CSV file.

441 Within this node, the user adds a child node and selects the cell
 442 "Total", which serves as both a table header and a row label. This
 443 selection represents the table header of the first subtabel. The user
 444 adds another child node and selects the range of cells containing

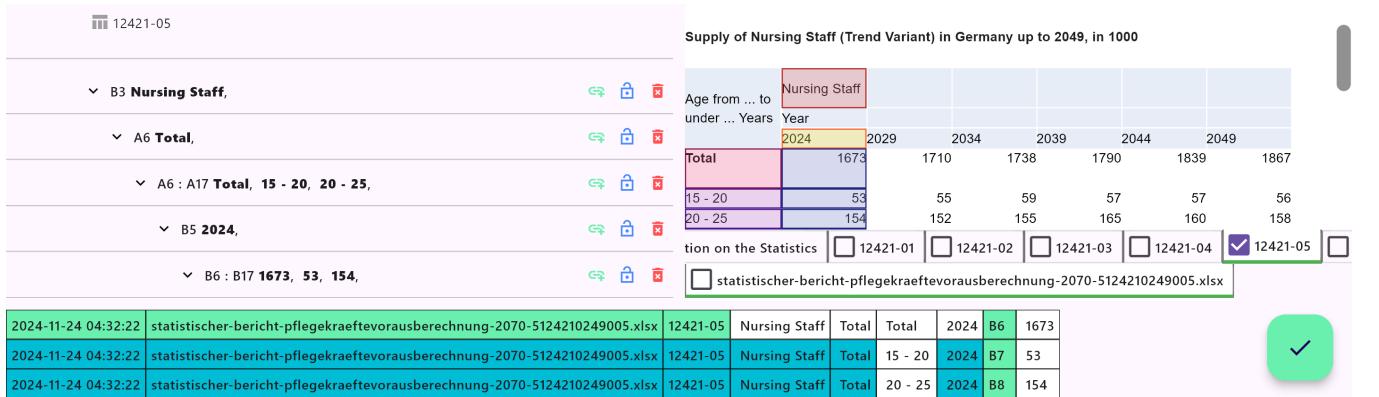


Figure 2: Selection of the First Column Metadata

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 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

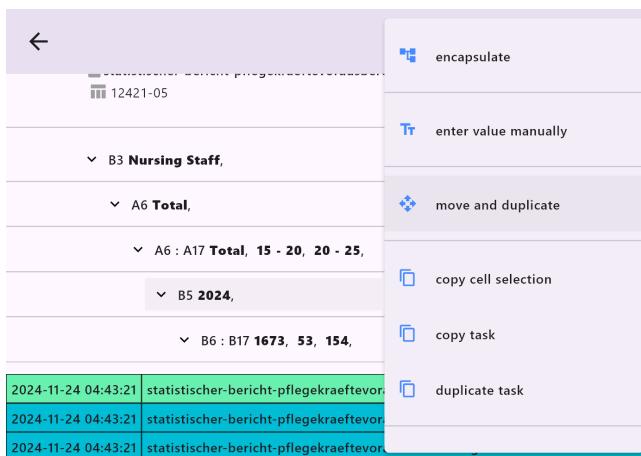


Figure 3: Invoking the "move and duplicate" feature on the 2024 column node.

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 3.

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 4. 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

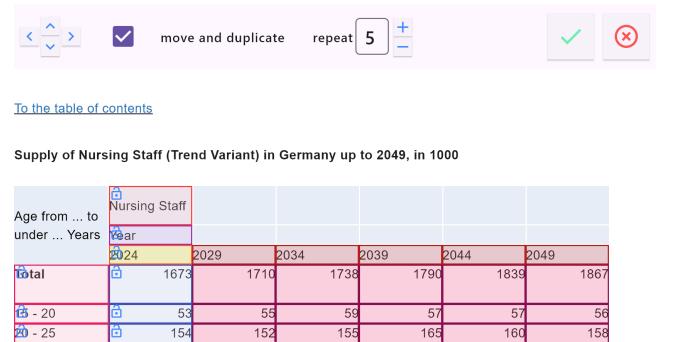


Figure 4: Moving the hierarchy one cell to the right while adjusting the number of repetitions to duplicate the column selection.

Figure 5: Resulting Hierarchy After Move and Accept

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 5 shows the resulting selection after the user approved the

The same method that worked effectively for duplicating the columns can now be applied to the subtables, as shown in Figure 6.

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 7 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.

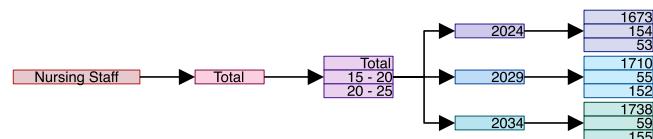


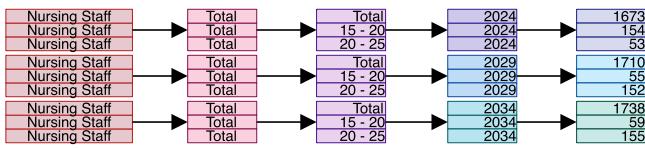
Figure 6: Selection of All Cells in the Subtables by Duplicating the Hierarchy of the First Table

697 To produce a flat, relational table, the **SDE** performs *path-wise*
 698 *broadcasting*: for each root-to-leaf path $P = (v_0, \dots, v_k)$ that ends
 699 in a list of N value cells (x_1, \dots, x_N) , the SDE materializes N output
 700 rows by repeating or aligning the labels found along the path.
 701

Concretely:

- 702 (1) **Broadcast singletons.** If a path node carries a single label
 703 (e.g., *Nursing Staff*, *Total*, or a single year), that label is
 704 replicated N times so each value cell inherits it.
- 705 (2) **Align equal-length lists.** If a path node provides a list
 706 of N labels, those labels are paired index-wise with the N
 707 value cells.
- 708 (3) **Emit one tuple per value cell.** For each $j \in \{1, \dots, N\}$,
 709 emit a row that contains the labels gathered from the path
 710 at position j (broadcast or aligned) together with the value
 711 x_j .

712 The resulting hierarchy after broadcasting is illustrated in Figure
 713 8: upstream labels are repeated or aligned so that each numeric
 714 cell is paired with its full context, yielding one relational row per
 715 cell.



716
 717
 718 **Figure 8: Selection hierarchy after path-wise broadcasting.**
 719 Each value cell is paired with its repeated/aligned upstream
 720 context, producing one row per cell.

721 5 Interface Fidelity for Navigation

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

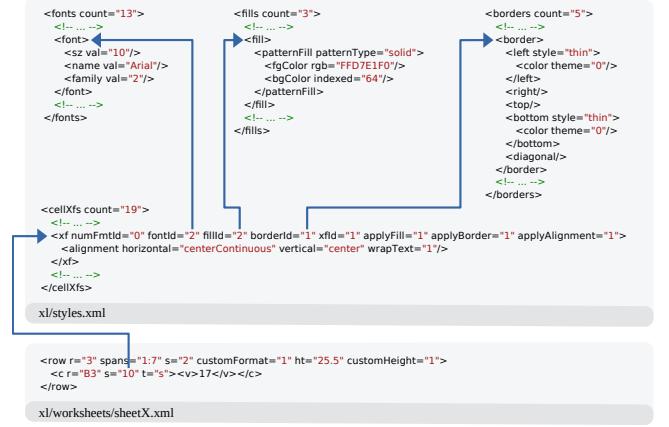
726
 727
 728
 729
 730 **5.0.1 Displaying Row Heights and Column Widths.** Our solution
 731 extracts information about column widths and row heights directly
 732 from the Excel file's XML structure. Specifically, we retrieve the
 733 column widths from the *width* attribute of the *<col>* elements and the
 734 row heights from the *ht* attribute of the *<row>* elements in the
 735 *sheetX.xml* files.

736 In Excel, column widths and row heights are defined in units
 737 that do not directly correspond to pixels, requiring conversion for
 738 precise on-screen rendering. Moreover, different scaling factors are
 739 applied for columns and rows. Despite extensive research, we were
 740 unable to find official documentation that explains the rationale
 741 behind these specific scaling factors. Based on empirical testing,
 742 we derived the following scaling factors:

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

745
 746
 747
 748
 749
 750
 751
 752
 753
 754 **5.0.2 Cell Formatting.** Cell formatting plays a crucial role in accurately representing the appearance of worksheets. Formatting

755 information is stored in the *styles.xml* file, where styles are defined and later referenced in the *sheetX.xml* files as shown in Figure 9.



756
 757
 758
 759
 760
 761
 762
 763
 764
 765
 766
 767
 768
 769
 770
 771
 772
 773
 774
 775
 776
 777
 778
 779
 780
 781
 782
 783
 784
 785
 786
 787
 788
 789
 790
 791
 792
 793
 794
 795
 796
 797
 798
 799
 800
 801
 802
 803
 804
 805
 806
 807
 808
 809
 810
 811
 812
Figure 9: Illustration of the relationship between style definitions in *xl/styles.xml* (fonts, fills, borders, and *cellXfs*) and their application in a worksheet file (*xl/worksheets/sheetX.xml*).

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

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

823
 824
 825
 826
 827
 828
 829
 830
 831
 832
 833
 834
 835
 836
 837
 838
 839
 840
 841
 842
 843
 844
 845
 846
 847
 848
 849
 850
 851
 852
 853
 854 **5.0.3 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.

855 We implemented text overflow handling by checking 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.

856 Figure 1 illustrates this behavior. The text "Supply of Nursing
 857 Staff ..." extends into the neighboring cell because it is empty. If not
 858 for this handling, the text would be truncated at the cell boundary,
 859 leading to incomplete data display as shown in Figure 10.

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

813	To the table of c						
814							
815	Supply of Nurs						
816							
817	Age from ... to	Nursing Staff					
818	under ... Years						
819	Year						
820	2024	2029	2034	2039	2044	2049	2049
	Total	1673	1710	1738	1790	1839	1867

Figure 10: Incorrect rendering without overflow for cells adjacent to empty cells.

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:

- **x1/sharedStrings.xml**: Contains all the unique strings used across worksheets, reducing redundancy.
- **x1/styles.xml**: Defines the formatting styles for cells, including fonts, colors, and borders.
- **x1/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.

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 (Alg. 1); 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 sentinel `</sheetData>` 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.

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  $-1$ 

```

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

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

See Alg. 2.

Lazy cell parsing. Given a row byte interval $[s, e]$, cells are parsed on demand. We scan for <c, read attributes ($r = "A123"$, $s = "..."$, $t = "..."$), and bound the cell interval by the next <c or the row end. Values are extracted *within* this interval from <v>...</v> or (for inline strings) <is>...</is>. See Alg. 3. Because Excel enforces increasing row indices, we can stop row streaming as soon as we pass the requested row.

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 <row ... r="...">. A single forward pass accumulates $\sum_{r \in E} ht(r)$ and counts $|E|$; all other rows use H_d . In device units,

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

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.

Complexity and memory. All searches are bounded by structural sentinels (>, next <row, next <c, or </sheetData>). Anchoring </sheetData> is $O(N)$ with tiny constants; row streaming is $O(R)$ with $O(1)$ per-row offset updates; a targeted cell within a row scans only that row’s interval; merge queries run in $O(\log M + \alpha)$ with a short local scan α . The representation is compact (byte slices + small numeric state), and strings are decoded only when needed.

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  $(r, [s, e], h, off)$ 
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
                 $off \leftarrow \rho \cdot (r - 1) H_d$ 
            else
                 $g \leftarrow r - \text{prevR} - 1$ 
                 $H_{\text{prev}} \leftarrow (\text{prevH or } H_d)$ 
                 $off \leftarrow \text{prevOff} + \rho \cdot H_{\text{prev}} + \rho \cdot g H_d$ 
            if  $\text{prevR} \neq \perp$  then emit  $(\text{prevR}, [\text{prevS}, s], \text{prevH}, \text{prevOff})$ 
             $\text{prevR} \leftarrow r$ ;  $\text{prevH} \leftarrow h$ ;  $\text{prevOff} \leftarrow off$ ;  $\text{prevS} \leftarrow s$ 
             $i \leftarrow j$ 
        else
             $i \leftarrow i + 1$ 
    if  $\text{prevR} \neq \perp$  then emit  $(\text{prevR}, [\text{prevS}, c], \text{prevH}, \text{prevOff})$ 

```

```

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  $\text{ParseRowAttrs}(b, i, c)$ :
     $r \leftarrow \perp$ ;  $h \leftarrow \perp$ ;  $j \leftarrow i + 4$ 
    while  $j < c$  do
        if  $b[j] = >$  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  $b[j:j + 1] = ht$  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  $h \leftarrow \text{PARSEDOUBLEASCII}(b, s, e)$ 
         $j \leftarrow j + 1$ 
    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  $\text{GetInnerAttrInterval}(i, n, b)$ :
    while  $i < n$  and  $b[i]$  is space do
         $i \leftarrow i + 1$ 
    if  $i < n$  and  $b[i] = =$  then
         $i \leftarrow i + 1$ 
        while  $i < n$  and  $b[i]$  is space do
             $i \leftarrow i + 1$ 
        if  $i < n$  and  $(b[i] = " or b[i] = ')$  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  $\perp$ 

```

6.3 Two-Dimensional Grid Viewport Rendering

Given a (potentially very large) worksheet with variable row heights, banded column widths, and merged regions, we lay out only tiles intersecting the current viewport. The renderer works in device

987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044

Algorithm 3: Resolve cell by streaming the row

```

1045 Input:  $b$ : bytes; row interval  $[S, E]$ ; target column  $c$ 
1046 Output: cell interval  $[s, e)$  or  $\perp$ 
1047  $i \leftarrow S$ 
1048 while  $i \leq E - 2$  do
1049   if  $b[i..i + 1] = <c$  then
1050      $s \leftarrow i$ 
1051      $(c', j) \leftarrow \text{PARSECELLCOL}(b, i, E)$            // reads r="A123" and
1052     advances to just after >
1053     // end of this cell = next <c or E
1054      $k \leftarrow j$ 
1055     while  $k \leq E - 2$  and  $b[k..k + 1] \neq <c$  do
1056        $k \leftarrow k + 1$ 
1057        $e \leftarrow (k \leq E - 2) ? k : E$ 
1058       if  $c' = c$  then
1059          $\perp$  return  $[s, e)$ 
1060        $i \leftarrow e$ 
1061     else
1062        $i \leftarrow i + 1$ 
1063   return  $\perp$ 
1064

```

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

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

$$r_l = \lceil \text{rowIndexAt}(y_0) \rceil, \quad r_u = \lceil \text{rowIndexAt}(y_0 + H_{vp} + \Delta) \rceil,$$

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

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 α is a short local scan in practice.

Origin-first placement for merged regions. A naïve scan of $[r_l : r_u] \times [c_l : c_u]$ 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 $(r_l, c_l : c_u)$ and the left border $(r_l : r_u, c_l)$, 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 $</sheetData>$, we scan backward to the last $<\text{row} \dots >$ 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* [13]).

Correctness and complexity. Edge probes ensure any merged region intersecting the viewport but originating above/left is instantiated *exactly once* at its origin. The interior pass either places unmerged cells or the unique origin cell of each merged region. Let $R_{\text{vis}} = r_u - r_l + 1$ and $C_{\text{vis}} = c_u - c_l + 1$. Passes 1-2 cost $O(R_{\text{vis}} + C_{\text{vis}})$; Pass 3 costs $O(R_{\text{vis}} C_{\text{vis}})$ and does no work outside the visible rectangle. The burned set is updated at most once per span per frame.

6.4 AABB-Indexed Selection Lookup

Each selection source (a task or composed group) maintains a lazily computed, cached *axis-aligned bounding box* (AABB) that is invalidated on updates and recomputed on demand. When many such selection sources (grids) intersect the viewport, a linear scan over their bounding boxes can dominate latency. We therefore maintain a lightweight index over axis-aligned bounding boxes (AABBS) to answer point queries (x, y) in $O(\log G + \alpha)$ time, where G is the number of visible grids and α is the size of a narrow candidate window.

Each grid i exposes a bounding rectangle $[L_i, R_i] \times [T_i, B_i]$ in grid coordinates and a point predicate $\text{findCell}_i(x, y)$ that returns the cell at (x, y) or \perp . We build parallel arrays L, R, T, B (left, right, top, bottom) and a stable original-order index O (used to break ties consistently with the UI). Let σ be the permutation that sorts grids by nondecreasing T (top edge). In that order we form a prefix-maximum array $\text{PM}[j] = \max_{0 \leq i \leq j} B_{\sigma(i)}$. For a query row y , the candidate window is the tightest interval $[\ell, h)$ such that all boxes whose top $\leq y$ and bottom $\geq y$ lie in that interval; we obtain

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

```

1161 Input:  $x_0, y_0; W_{vp}, H_{vp}$ ; cache  $\Delta$ ; sheet accessors
1162           rowIndexAt, colIndexAt, off, colOff,  $h, w$ ; merge index spanAt, isOrigin
1163 Output: positioned tiles for current frame
1164 // Visible indices
1165  $r_t \leftarrow [\text{rowIndexAt}(y_0)], r_u \leftarrow [\text{rowIndexAt}(y_0 + H_{vp} + \Delta)]$ 
1166  $c_t \leftarrow [\text{colIndexAt}(x_0)], c_u \leftarrow [\text{colIndexAt}(x_0 + W_{vp} + \Delta)]$ 
1167  $B \leftarrow \emptyset // burned merged spans$ 
1168
1169 // Pass 1: top border probes
1170 for  $c \leftarrow c_t$  to  $c_u$  do
1171    $S \leftarrow \text{spanAt}(r_t, c)$ 
1172   if  $S \neq \perp$  and  $S \notin B$  and  $r_S < r_t$  then
1173     place tile for  $S$  at  $(\text{colOff}(c_S) - x_0, \text{off}(r_S) - y_0)$ 
1174      $B \leftarrow B \cup \{S\}$ 
1175
1176 // Pass 2: left border probes
1177 for  $r \leftarrow r_t$  to  $r_u$  do
1178    $S \leftarrow \text{spanAt}(r, c_t)$ 
1179   if  $S \neq \perp$  and  $S \notin B$  and  $c_S < c_t$  then
1180     place tile for  $S$  at  $(\text{colOff}(c_S) - x_0, \text{off}(r_S) - y_0)$ 
1181      $B \leftarrow B \cup \{S\}$ 
1182
1183 // Pass 3: interior tiles
1184 for  $c \leftarrow c_t$  to  $c_u$  do
1185    $x \leftarrow \text{colOff}(c) - x_0$ 
1186   for  $r \leftarrow r_t$  to  $r_u$  do
1187      $y \leftarrow \text{off}(r) - y_0$ 
1188      $S \leftarrow \text{spanAt}(r, c)$ 
1189     if  $S = \perp$  or  $\text{isOrigin}(S, (r, c))$  then
1190       place tile at  $(x, y)$ 
1191
1192 //  $\text{ub}(A, x) = \min\{i \mid A[i] > x\}$ ,  $\text{lb}(A, x) = \min\{i \mid A[i] \geq x\}$ .
1193 Function  $\text{spanAt}(r, c):$ 
1194    $hi \leftarrow \text{ub}(R_1, r)$ 
1195   if  $hi = 0$  then
1196      $\perp$ 
1197    $lo \leftarrow \text{lb}(\text{PM}[0 : hi], r)$ 
1198   if  $lo \geq hi$  then
1199      $\perp$ 
1200    $k \leftarrow \text{ub}(C_1[lo : hi], c) - 1$ 
1201   if  $k < lo$  then
1202      $\perp$ 
1203   for  $i \leftarrow k$  downto  $lo$  do
1204     if  $C_1[i] > c$  then
1205        $\perp$ 
1206     if  $r \leq R_2[i]$  and  $c \leq C_2[i]$  then
1207        $\perp$ 
1208   return  $\perp$ 

```

$h = \text{ub}(T_\sigma, y)$ and $\ell = \text{lb}(\text{PM}[0 : h], y)$ by binary search (ub : strict upper bound; lb : lower bound). We then prune by the x -interval condition $L \leq x \leq R$. Among the surviving candidates we test findCell in increasing O (original) order and return the first hit; if none match, the answer is \perp .

The index builds in $O(G \log G)$ time and consists of six integer arrays plus the permutation π . It is invalidated on any structural change to the set of grids (move, resize, insert/delete) and rebuilt lazily on the next query. For small G (e.g., $G \leq 32$) we fall back to a linear scan; the cutover can be tuned empirically.

6.5 Lazy Output Flattening and Row Location

Large workbooks and selection hierarchies make fully materializing a relational view prohibitively expensive. Empirically, naive

Algorithm 5: AABB-indexed selection lookup over visible grids

```

1219 Input: point  $(x, y)$ ; arrays  $L, R, T, B$ ; permutation  $\sigma$  sorting by  $T$ ; prefix max
1220           PM over  $B_{\sigma(\cdot)}$ ; original-order  $O$ ; accessor  $\text{findCell}_i(x, y)$ 
1221 Output: either  $\langle i, \text{cell} \rangle$  or  $\perp$ 
1222 // Binary-search helpers:  $\text{ub}(A, z) = \min\{i \mid A[i] > z\}$ ,
1223            $\text{lb}(A, z) = \min\{i \mid A[i] \geq z\}$ .
1224
1225 // 1) Y-narrowing: window of candidates whose top  $\leq y$  and bottom
1226            $\geq y$ 
1227  $h \leftarrow \text{ub}(T_\sigma, y)$                                      // first index with  $T_{\sigma(h)} > y$ 
1228 if  $h = 0$  then
1229    $\perp$ 
1230    $\ell \leftarrow \text{lb}(\text{PM}[0 : h], y)$                          // first prefix with bottom  $\geq y$ 
1231   if  $\ell \geq h$  then
1232      $\perp$ 
1233
1234 // 2) X-pruning: keep only boxes covering  $x$ 
1235  $\mathcal{C} \leftarrow \emptyset$ 
1236 for  $j \leftarrow \ell$  to  $h - 1$  do
1237    $i \leftarrow \sigma(j)$ 
1238   if  $L_i \leq x \leq R_i$  then
1239     append  $i$  to  $\mathcal{C}$ 
1240
1241 if  $\mathcal{C} = \emptyset$  then
1242    $\perp$ 
1243
1244 // 3) Stable tie-break and confirmation
1245  $\mathcal{C} \leftarrow \text{sort } \mathcal{C} \text{ by increasing } O_i$                   // preserve UI order/colors
1246 foreach  $i \in \mathcal{C}$  do
1247    $ans \leftarrow \text{findCell}_i(x, y)$ 
1248   if  $ans \neq \perp$  then
1249      $\perp$ 
1250
1251 return  $\perp$ 

```

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/sheets/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 , *ENSUREBUILT-THROUGHRow* extends the stream until *builtRows* $> r$ or the source is exhausted (Alg. 6). 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. 6, LOCATEROW). Both steps are $O(k)$ work to extend by k newly discovered rows plus $O(\log |\text{starts}|)$ for the lookup.

Algorithm 6: Lazy row locator: driver and lookup (part A)

```

1277 Input: hierarchy files → tasks → sheets → cellTasks; caps  $B_{\max}$ ,  $R_{\max}$ 
1278 Output: on demand: for global row  $r$ , return (RowBlock, local) or  $\perp$ 
1279 State: arrays  $blocks$ ,  $starts$ ; queue  $Q$ , indices  $fi$ ,  $sti$ ,  $si$ ;  $curFile$ ,  $curTask$ ,
1280    $curSheets$ ; ( $activeWb$ ,  $activeSheet$ ).
1281 Function EnsureBuiltThroughRow( $r$ ):
1282   while  $builtRows \leq r$  do
1283      $rb \leftarrow NextBlock()$ 
1284     if  $rb = \perp$  then
1285       break
1286     append  $rb$  to  $blocks$ ; append  $rb.startRow$  to  $starts$ ;
1287     PruneHeadIfNeeded()
1288
1289 Function LocateRow( $r$ ):
1290   EnsureBuiltThroughRow( $r$ )
1291   if  $blocks = \emptyset$  then
1292     return  $\perp$ 
1293    $lo \leftarrow 0$ ;  $hi \leftarrow |starts|$ 
1294   while  $lo < hi$  do
1295      $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$ 
1296     if  $starts[mid] \leq r$  then
1297        $lo \leftarrow mid + 1$ 
1298     else
1299        $hi \leftarrow mid$ 
1300
1301      $idx \leftarrow lo - 1$ 
1302     if  $idx < 0$  then
1303       return  $\perp$ 
1304      $b \leftarrow blocks[idx]$ ;  $local \leftarrow r - b.startRow$ 
1305     if  $0 \leq local < b.len$  then
1306       return  $(b, local)$ 
1307     else
1308       return  $\perp$ 
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
```

Streaming next block. NEXTBLOCK (Alg. 7) 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.

Algorithm 7: Row block generation (part B): NextBlock

```

1314 Function NEXTBLOCK:
1315   if  $\neg FillQueueIsEmpty()$  then
1316     return  $\perp$ 
1317   while true do
1318     if  $Q = \emptyset$  then
1319       if  $\neg FillQueueIsEmpty()$  then
1320         return  $\perp$ 
1321       continue
1322      $(t, path) \leftarrow pop-front Q$ 
1323     if  $t.children = \emptyset$  then
1324        $cells \leftarrow t.sortedSelectedCells$ 
1325       if  $|cells| = 0$  then
1326         continue
1327        $start \leftarrow builtRows$ ;  $len \leftarrow |cells|$ 
1328       return RowBlock( $activeWb$ ,  $activeSheet$ ,  $path$ ,  $cells$ ,  $start$ ,  $len$ )
1329      $nextPath \leftarrow path \cup \{t\}$ 
1330     foreach  $u \in t.children$  do
1331       push-back  $(u, nextPath)$  into  $Q$ 
1332
1333
1334
```

Source advancement. FILLQUEUEIFEMPTY (Alg. 8) 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 skipped. When the last file is exhausted and Q remains empty, the function returns **false** and the stream is complete.

Algorithm 8: Traversal feeding (part C): FillQueueIsEmpty

```

1335 Function FILLQUEUEIFEMPTY:
1336   while  $Q = \emptyset$  do
1337     if  $curTask = \perp$  then
1338       if  $fi \geq |files|$  then
1339         return false
1340        $curFile \leftarrow files[fi + +]$ ;  $sti \leftarrow 0$ 
1341
1342       if  $sti \geq |curFile.sheetTasks|$  then
1343          $curTask \leftarrow \perp$ ; continue
1344
1345        $curTask \leftarrow curFile.sheetTasks[sti + +]$ ;  $curSheets \leftarrow$ 
1346       list( $curTask.sheets$ );  $si \leftarrow 0$ 
1347
1348       while  $si < |curSheets|$  and  $Q = \emptyset$  do
1349          $(wb, sh) \leftarrow curSheets[si + +]$ ;
1350          $roots \leftarrow curTask.importExcelCellsTasks$ 
1351
1352         if  $|roots| = 0$  then
1353           continue
1354          $Q \leftarrow$  queue of  $(root, \emptyset)$  for each  $root \in roots$ 
1355          $activeWb \leftarrow wb$ ;  $activeSheet \leftarrow sh$ 
1356
1357       if  $Q \neq \emptyset$  then
1358         return true
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
```

Cache pruning. PRUNEHEADIFNEEDED (Alg. 9) 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.

Algorithm 9: Cache pruning (part D): PruneHeadIfNeeded

```

1370 Function PRUNEHEADIFNEEDED:
1371    $rm \leftarrow 0$ 
1372   if  $B_{\max} \neq \perp$  and  $|blocks| > B_{\max}$  then
1373      $rm \leftarrow |blocks| - B_{\max}$ 
1374
1375   if  $R_{\max} \neq \perp$  then
1376     while  $rm < |blocks|$  do
1377        $cached \leftarrow blocks[|blocks| - 1].startRow + blocks[|blocks| -$ 
1378        $1].len - blocks[0].startRow$ 
1379       if  $cached \leq R_{\max}$  then
1380         break
1381        $rm \leftarrow rm + 1$ 
1382
1383     if  $rm > 0$  then
1384       drop first  $rm$  from  $blocks$  and  $starts$ 
1385
1386
1387
1388
1389
1390
1391
1392
```

Invariants and correctness. (1) **Monotone coverage:** every emitted block has $startRow = builtRows$ at creation time, and we update $builtRows \leftarrow builtRows + 1$; thus blocks are disjoint and cover $[0, builtRows]$ without gaps. (2) **Sorted index:** $starts$ is strictly increasing; binary search always returns the last block whose start $\leq r$. (3) **Determinism:** the traversal order is completely determined by the order of files/sheet-tasks/sheets and by the top-down BFS over the selection tree; repeated runs produce identical global row layouts. (4) **Progress:** each child is enqueued at most once and

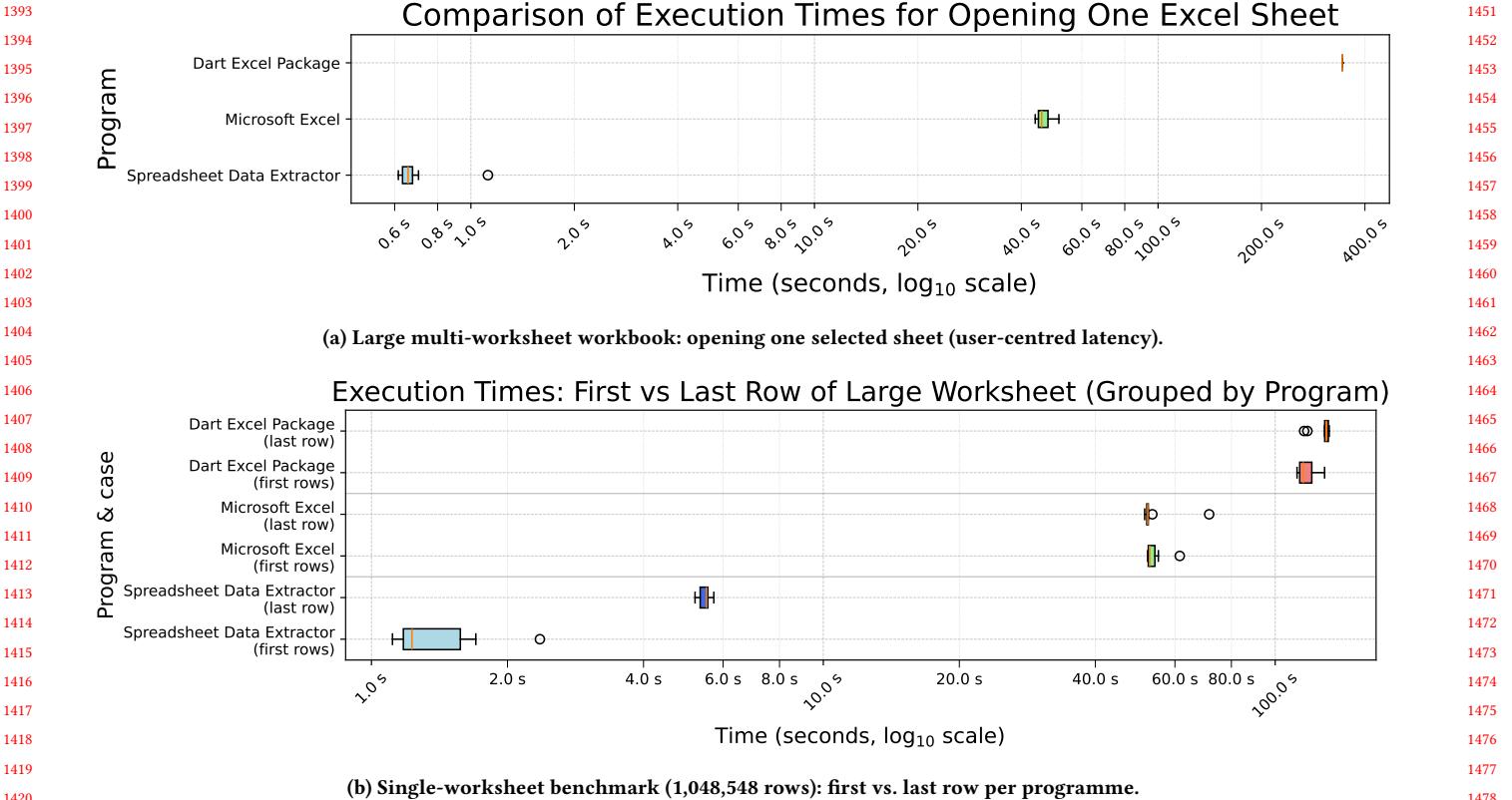


Figure 11: Time-to-first-visual vs. full-parse costs across tools. (a) shows the practical benefit of initialising only the selected sheet in a multi-worksheet file; (b) isolates algorithmic differences under identical workload. Both plots aggregate 10 runs on a logarithmic time axis.

each leaf produces at most one block; the stream terminates after a bounded number of steps.

Complexity. Let N be the number of leaf tasks that produce non-empty `sortedSelectedCells`, and let $M = \sum \text{len}$ be the total number of output rows. Streaming is linear in the explored work: each internal node is visited once and each leaf contributes $O(1)$ metadata plus $O(\text{len})$ to copy/select references to its cells. A lookup for row r costs $O(k + \log |\text{starts}|)$, where k is how many new rows must be streamed to cover r (often $k = 0$ for warm caches). Memory is $O(|\text{blocks}|)$, bounded by B_{\max} or by R_{\max} rows of history.

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 $y_0 + H_{\text{vp}} + \Delta$, 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.

Failure modes. Segments with empty selections yield no blocks but are skipped efficiently. If the hierarchy ends before covering the requested row, the driver returns \perp ; the UI interprets this as “no more data.” All pruning policies preserve correctness: they only remove *past* blocks and never create gaps inside the realized suffix.

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

The Spreadsheet Data Extractor (SDE) builds on the converter by Aue et al. [3], whose extraction effectiveness was evaluated on over 500 Excel files (331 files; 3,093 worksheets; mean 15 min per file, 95 s per worksheet). Our contribution focuses on *user experience* and *latency*: we render worksheets as they appear in Excel, reduce context switching by integrating selection hierarchy, worksheet view, and output preview, and engineer incremental loading, byte-level parsing and plus viewport-bounded rendering.

7.1 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; 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). Throughout, we report medians over repeated runs and show distributions as box plots on a \log_{10} time axis.



Figure 12: SDE performance profiling during scrolling. Timeline shows per-frame rendering costs (blue); average 18ms/frame and peak 54ms/frame.

Setup. We evaluate three programmes: the *SDE*, Microsoft Excel (automated via PowerShell/COM), and a Dart *excel* package. 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 [18], 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. 11a). The Dart *excel* package completed only the first run at 343.568 s; runs 2-to-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.

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 ~ 35 MB on disk (XLSX ZIP container) and ~ 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×**).

Results (overview). Initialising only the selected sheet yields much lower time-to-first-visual (Fig. 11a); under identical workload SDE remains ~ 10× faster even when a full parse is enforced (Fig. 11b).

7.2 Interactive scalability

While the underlying extraction model remained effective, the previous prototype did not stay interactive on very large sheets. Empirically, on workbooks with hundreds of thousands of rows—and in particular near Excel’s 10^6 -row limit—two failure modes dominated:

- (1) **Eager materialisation in views.** When the selection view or the output view attempted to realise large regions eagerly, per-frame times rose above 1 s, making panning and selection unusable.
- (2) **Large selection hierarchies.** With many nested selections, highlight updates and duplicate/move previews performed work proportional to the total number of selected cells, causing multi-second pauses per interaction and making operations on large datasets impractical.

These observations motivated the engineering in 6.2 Parsing Worksheet XML with Byte-Level Pattern Matching: (i) byte-level, on-demand parsing of worksheet XML; (ii) viewport-bounded rendering that lays out only $[r_l : r_u] \times [c_l : c_u]$ (Algorithm Scroll extents); (iii) AABB-based filtering of selection sources with cache invalidation (6.4 AABB-Indexed Selection Lookup); and (iv) on-demand flattening of the output table (6.5 Lazy Output Flattening and Row Location). Together these changes bound per-interaction work to the visible area rather than the sheet size.

On a sheet with 10^6 rows and a selection covering all cells, the UI remained smooth during scrolling, sustaining ≈ 2 ms per frame (Fig. 12), which corresponds to ≈ 500 FPS. When dragging the scrollbar—filling the entire viewport at once with new content—frame times rose to about 60-80 ms ($\approx 12\text{-}17$ FPS), which remained acceptable for rapid navigation at that scale.

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, and a hard disk drive (HDD), running Microsoft Windows 11 Enterprise (build 26100, 64-bit). The version of Microsoft Excel used was version 16.0.

8 Conclusion

In this paper, we introduced the Spreadsheet Data Extractor [2], an enhanced tool that builds upon the foundational work of Aue et al. [3]. By addressing key limitations of the existing solution, we implemented significant performance optimizations and usability enhancements. Specifically, SDE employs incremental loading of worksheets and optimizes rendering by processing only the visible cells, resulting in performance improvements that enable the tool to open large Excel files.

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 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] Rui Abreu, Jácome Cunha, Joao Paulo Fernandes, Pedro Martins, Alexandre Perez, and Joao Saraiva. 2014. Faultysheet detective: When smells meet fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 625–628.
- [2] Anonymous. [n. d.]. Spreadsheet Data Extractor (SDE). https://anonymous.4open.science/r/spreadsheet_data_extractor-13BD/README.md. Accessed: 2024-12-08.
- [3] Alexander Aue, Andrea Ackermann, and Norbert Röder. 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.
- [4] 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.
- [5] 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. doi:10.1109/2.970578
- [6] 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.
- [7] Jácome Cunha, Joao Paulo Fernandes, Pedro Martins, Jorge Mendes, and Joao Saraiva. 2012. Smellsheet detective: A tool for detecting bad smells in spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 243–244.
- [8] Kaval Desai. 2024. Excel Dart Package. <https://pub.dev/packages/excel>. Accessed: 2024-12-03.
- [9] 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.
- [10] Angus Dunn. 2010. Spreadsheets—the Good, the Bad and the Downright Ugly. *arXiv preprint arXiv:1009.5705* (2010).
- [11] 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.
- [12] 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.
- [13] Kate Lovett. 2023. two_dimensional_scrollables package - Commit 4c16f3e. <https://github.com/flutter/packages/commit/4c16f3ef40333aa0aebe8a1e46ef7b9fef9a1c1f> Accessed: 2023-08-17.
- [14] Kelly Mack, John Lee, Kevin Chang, Karrie Karahalios, and Aditya Parameswaran. 2018. Characterizing scalability issues in spreadsheet software using online forums. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–9.
- [15] Sajadur Rahman, Mangesh Bindre, Yuyang Liu, Shichu Zhu, Zhaoyuan Su, Karrie Karahalios, and Aditya G Parameswaran. 2021. NOAH: interactive spreadsheet exploration with dynamic hierarchical overviews. *Proceedings of the VLDB Endowment* 14, 6 (2021), 970–983.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] Alaaeddin Swidan, Felienne Hermans, and Ruben Koesemowidjojo. 2016. Improving the performance of a large scale spreadsheet: a case study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 673–677.
- [20] Dixin Tang, Fanchao Chen, Christopher De Leon, Tana Wattanawaroon, Jeaseok Yun, Srinivasan Seshadri, and Aditya G Parameswaran. 2023. Efficient and Compact Spreadsheet Formula Graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2634–2646.
- [21] 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>

1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740