Practical Spreadsheet Parsing with SheetReader

Haralampos Gavriilidis
BIFOLD & TU Berlin
Berlin, Germany
gavriilidis@tu-berlin.de

Felix Henze
Access Microfinance Holding AG
Berlin, Germany
felix.henze@accessholding.com

Joel Ziegler
RSG Group GmbH
Berlin, Germany
joel.ziegler93@gmail.com

Jonas Benn
TU Berlin
Berlin, Germany
benn@campus.tu-berlin.de

Eleni Tzirita Zacharatou
HPI & Uni Potsdam
Potsdam, Germany
eleni.tziritazacharatou@hpi.de

Volker Markl
BIFOLD, TU Berlin & DFKI
Berlin, Germany
volker.markl@tu-berlin.de

Abstract

Spreadsheets remain a ubiquitous tool for data management and analysis. Since systems like Excel offer limited analytical capabilities, users routinely load spreadsheets into richer ecosystems such as Python, R, and DBMSes. However, existing spreadsheet loaders rely on general-purpose XML parsers that are ill-suited for the XLSX format, resulting in severe CPU and memory bottlenecks. In prior work, we introduced SheetReader, a specialized spreadsheet parser that leverages the structure of XLSX files and employs parallelism to significantly reduce ingestion costs, achieving up to an order of magnitude speedup and multi-gigabyte memory savings compared to state-of-the-art methods. This demonstration provides an interactive workbench where visitors can visualize XLSX internals, benchmark SheetReader against baseline parsers with live resource monitoring, and explore integrations for Python, R, PostgreSQL, and DuckDB, including running SQL directly over spreadsheets.

Keywords

Spreadsheet parser, XLSX data loading, Parsing parallelization

1 Introduction

Spreadsheets remain one of the most widely used data tools, with billions of users relying on them to store, share, and analyze data in business, science, government, and everyday tasks [1, 2, 8, 15]. Despite their popularity, spreadsheet systems offer only limited analytical capabilities. As a result, users routinely load spreadsheets into Python and R for machine learning and data wrangling, or into DBMSes to integrate them with relational data. All of these workflows begin by ingesting the spreadsheet file itself, a seemingly simple step that is, in practice, a major performance bottleneck on commodity hardware.

The Spreadsheet Ingestion Bottleneck. In our prior work [6], we found that loading a 172 MB spreadsheet in R required ∼30 s and up to 13 GB of memory with the fastest Excel parser, whereas the equivalent CSV required only 4 s and 1.1 GB. The most memory-efficient Excel parser still used around 5 GB and took roughly 160 s, 40× slower than CSV. We observed similarly severe overheads in other environments. These gaps stem from the XLSX format, which stores data as compressed XML that must be fully decompressed, tokenized, and materialized before use, resulting in ingestion costs that are unacceptable for interactive use on commodity hardware; business laptops may even become unresponsive during loading.

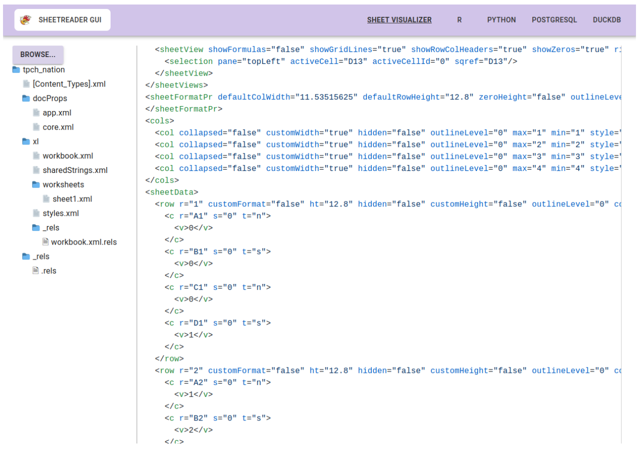
EDBT '26, Tampere (Finland)

Limitations of Current Spreadsheet Parsers. Despite the ubiquity of spreadsheets, efficient parsing has received far less attention than parsing other text-based formats. Recent work on CSV and JSON has produced highly optimized, structure-aware parsers [7, 10, 11, 16], yet these techniques do not transfer to spreadsheets. XLSX files package worksheets in a compressed ZIP archive containing several interdependent XML documents, including shared-string tables and cell-level markup. Accordingly, state-of-the-art spreadsheet loaders rely on standard DOM or SAX parsers. Although XML parsing has seen substantial research [9, 13], these techniques target general-purpose documents and cannot exploit spreadsheet-specific structure. As a result, existing loaders perform unnecessary tokenization and materialize large in-memory XML trees, which can reach several gigabytes even for moderately sized files [6]. Spreadsheet ingestion, therefore, remains inefficient in practice and requires a more targeted parsing approach.

Specialized Spreadsheet Parsing with SheetReader. To address these limitations, we introduced `SheetReader`, a structure-aware spreadsheet parser [6]. We leverage spreadsheet-specific characteristics such as shared-string tables and cell-level markup and combine parallelism at multiple levels with optimizations that reduce redundant operations. `SheetReader` offers two execution modes for different resource constraints: a *consecutive* mode that maximizes throughput by decompressing the XML document before parallel parsing, and an *interleaved* mode that lowers memory usage by parsing data as it is decompressed. These design choices reduce ingestion time by up to an order of magnitude and cut memory consumption by several gigabytes compared to state-of-the-art libraries [6]. Our prior work evaluated `SheetReader` in Python and R, and in this demo we extend its applicability with PostgreSQL and DuckDB integrations, showing that the same environment-agnostic parser can serve both data science and DBMS workloads. Its efficient, portable design and growing open-source adoption highlight practical value.[1]

Demonstration. Our demo provides a hands-on experience that illustrates the challenges of spreadsheet ingestion and how `SheetReader` improves performance across environments. We developed an application with a spreadsheet visualizer, embedded execution environments, and live resource monitoring. The visualizer exposes the structure of XLSX files to highlight the overheads faced by existing parsers. Visitors can then run baseline parsers alongside `SheetReader` in embedded Jupyter notebooks for Python, R, PostgreSQL, and DuckDB. Live monitoring dashboards display CPU and memory utilization during ingestion, allowing visitors to observe performance differences. Beyond

[1] Open source: https://github.com/polydbms/sheetreader-core.



```
tpch_nation
  [Content_Types].xml
  docProps
    app.xml
    core.xml
  xl
    workbook.xml
    sharedStrings.xml
    worksheets
      sheet1.xml
    styles.xml
    _rels
      workbook.xml.rels
  _rels
    .rels
```

```
  <sheetView showFormulas="false" showGridLines="true" showRowColHeaders="true" showZeros="true" ri
    <selection pane="topLeft" activeCell="D13" activeCellId="0" sqref="D13"/>
  </sheetView>
</sheetViews>
<sheetFormatPr defaultColWidth="11.53515625" defaultRowHeight="12.8" zeroHeight="false" outlineLeve
</sheetFormatPr>
<cols>
  <col collapsed="false" customWidth="true" hidden="false" outlineLevel="0" max="1" min="1" style='
  <col collapsed="false" customWidth="true" hidden="false" outlineLevel="0" max="2" min="2" style='
  <col collapsed="false" customWidth="true" hidden="false" outlineLevel="0" max="3" min="3" style='
  <col collapsed="false" customWidth="true" hidden="false" outlineLevel="0" max="4" min="4" style='
</cols>
<sheetData>
  <row r="1" customFormat="false" ht="12.8" hidden="false" customHeight="false" outlineLevel="0" co
    <c r="A1" s="0" t="n">
      <v>0</v>
    </c>
    <c r="B1" s="0" t="s">
      <v>0</v>
    </c>
    <c r="C1" s="0" t="n">
      <v>0</v>
    </c>
    <c r="D1" s="0" t="s">
      <v>1</v>
    </c>
  </row>
  <row r="2" customFormat="false" ht="12.8" hidden="false" customHeight="false" outlineLevel="0" co
    <c r="A2" s="0" t="n">
      <v>1</v>
    </c>
    <c r="B2" s="0" t="s">
      <v>2</v>
    </c>
```

Figure 1: The spreadsheet visualizer shows the XLSX archive layout and the XML of worksheets, highlighting how the markup-heavy structure becomes a bottleneck.

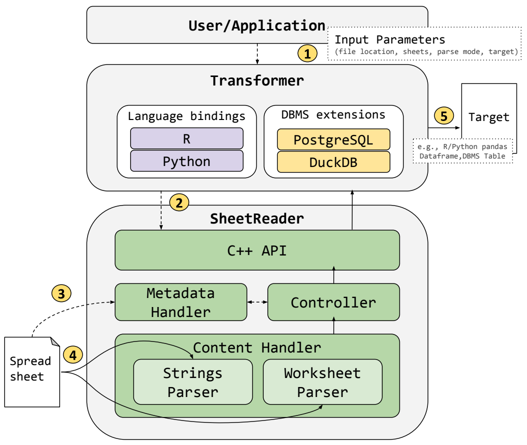


Figure 2: The core parser processes worksheet and string XML files in parallel and exposes a uniform interface to support multiple runtimes.

performance metrics, the demo illustrates how SheetReader supports practical workflows, from loading spreadsheets into Python and R for data science pipelines, to executing SQL directly over spreadsheets in PostgreSQL and DuckDB, and integrating spreadsheets with DBMS tables.

2 Background & Related Work

Since XLSX files are compressed XML archives, spreadsheet ingestion requires both decompression and parsing. We briefly outline the structure of XLSX files and summarize relevant work on XML, CSV, and JSON parsing to provide context for our main design choices.

2.1 Spreadsheet Structure

As illustrated by the spreadsheet visualizer included in our demo (Figure 1), an XLSX file is a ZIP container holding multiple interdependent XML documents. A workbook file lists the worksheets and their locations, individual worksheet files (e.g., `sheet1.xml`) store the cell data, and a separate `sharedStrings.xml` file holds the string table referenced by cells via indices. Additional relationship files define how these components link together. This organization improves storage efficiency but complicates ingestion, as parsers must navigate several XML files and follow indirections.

2.2 General-Purpose XML Parsing

Existing spreadsheet loaders typically rely on standard DOM or SAX parsers after decompressing the archive. Although prior work has explored parallel XML parsing and schema-specialized parsers [9, 13], these techniques target generic XML and do not exploit spreadsheet-specific structure such as shared-string tables or predictable cell markup. As a result, they perform unnecessary tokenization work and often materialize large XML trees, contributing to the high memory and runtime overhead commonly observed in practice.

2.3 Optimized Text-based Format Parsers

There has been significant progress on efficient parsing for CSV [7, 12] and JSON [10, 11, 14], using vectorized scanning, speculative parsing, and SIMD-based filtering to exploit the flat or lightly structured nature of those formats. However, these techniques do not directly transfer to spreadsheets, whose multi-file, XML-based layout introduces different constraints and dependencies. While some of these optimizations may complement a spreadsheet-specific parser, the core ingestion challenge requires a dedicated approach.

3 SheetReader

To make spreadsheet parsing practical on commodity machines, we introduced SheetReader [6], a structure-aware XLSX parser that exploits spreadsheet-specific properties, employs parallelism, and offers two modes for runtime or memory efficiency. We describe the components and optimizations in more detail in our prior work [6]; below, we briefly summarize the core ideas.

3.1 Overview

Figure 2 presents a high-level view of SheetReader. Users provide the input parameters, such as file location, selected sheets, and parsing mode, through the bindings (1). This information is forwarded to the C++ core API (2), where the Controller coordinates the parsing process. The Metadata Handler extracts workbook information such as sheet locations and names (3). The Content Handler then processes the spreadsheet data: the Strings Parser reads the shared-string table and the Worksheet Parser reads the cell data (4). Finally, the Transformer converts the intermediate columnar representation into the target structure (5), for example a Pandas DataFrame, an R data frame, or a DBMS table. This modular design enables the same parser to efficiently support Python, R, PostgreSQL, and DuckDB. SheetReader is implemented in C++ in a target-agnostic way to offload performance-critical computations to a native runtime [3], with language integrations provided through lightweight bindings.

3.2 Parsing Modes

SheetReader applies several spreadsheet-specific optimizations, such as specialized handling of shared strings, lightweight cell-level markup decoding, and preallocation strategies to avoid dynamic resizing of intermediate structures. In addition to these general optimizations, it offers two execution modes tailored to different resource constraints. The *consecutive* mode targets

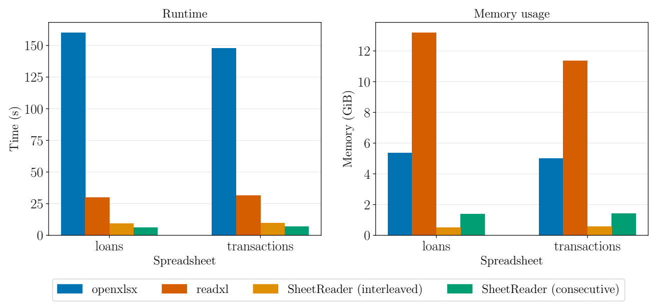


Figure 3: SheetReader vs. R baselines on real-world spreadsheets, for runtime (left) and memory usage (right).

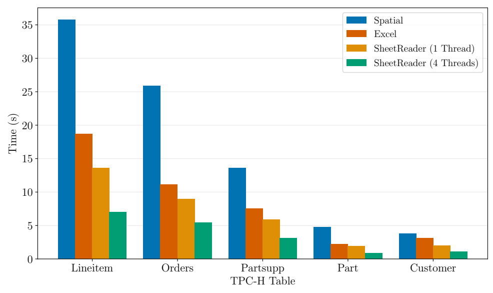


Figure 4: DuckDB spreadsheet loading runtime on TPC-H tables: core extensions (Excel, Spatial) vs. SheetReader.

maximum throughput: it fully decompresses the worksheet XML, identifies row boundaries, and parallelizes parsing the resulting segments to achieve high CPU utilization. The *interleaved* mode minimizes peak memory by interleaving decompression with parsing, streaming fixed-size buffers directly to the parser threads without fully materializing the XML tree. These complementary modes allow `SheetReader` to operate efficiently in practice on both memory-limited laptops and DBMS backends that require predictable resource usage.

Planned extensions include predicate pushdown, formula evaluation, richer multi-sheet and multi-table spreadsheet layouts, support for online spreadsheet sources and additional spreadsheet formats, and exposing the intermediate representation in popular interchange formats such as Apache Arrow.

3.3 Performance

Our prior work [6] provides an extensive evaluation across real and synthetic datasets, where `SheetReader` outperforms state-of-the-art spreadsheet loaders. Figure 3 highlights representative results, showing up to 17× faster ingestion than `openxlsx` and up to 3.2× faster than `readxl` in R, while reducing memory usage by up to 26×. Figure 4 reports spreadsheet loading performance in DuckDB 1.4.2, comparing our `SheetReader` extension with DuckDB's default core spreadsheet extensions (Spatial and Excel) on TPC-H tables (sf 1). `SheetReader` consistently outperforms the core extensions. It is generally faster in single-threaded mode, and with four threads it shows the largest improvements on larger sheets (e.g., Lineitem), with speedups of up to ~7× over the Spatial extension and about ~2.5× over the Excel extension. Additional results for Python and other environments, together with full experimental setup details, are provided in the full paper [6]. These gains directly address ingestion bottlenecks observed in heterogeneous data science and DBMS environments [5, 17].

4 Demonstration Walkthrough

The goal of our demonstration is threefold: (i) reveal the internal structure of XLSX files and show why general-purpose XML parsing leads to inefficiencies; (ii) let visitors observe, in real time, how SheetReader reduces CPU and memory consumption compared to state-of-the-art tools; and (iii) showcase SheetReader's portability across ecosystems, including Python, R, PostgreSQL, and DuckDB, where it enables efficient data loading, SQL analytics on spreadsheets, and integration with DBMS tables.

4.1 Interactive Spreadsheet Workbench

We developed an interactive application that brings together the main elements of our demo: a spreadsheet visualizer, environments for running SheetReader alongside baseline parsers, and live monitoring of resource usage. The workbench embeds Jupyter notebooks for Python, R, PostgreSQL, and DuckDB, so that users can run real workloads and directly compare different parsing approaches in their native environments. To make performance behavior visible, the interface includes lightweight Grafana dashboards that display CPU and memory utilization in real time, with metrics collected using Prometheus within our Docker setup.

4.2 Understanding Spreadsheet Internals

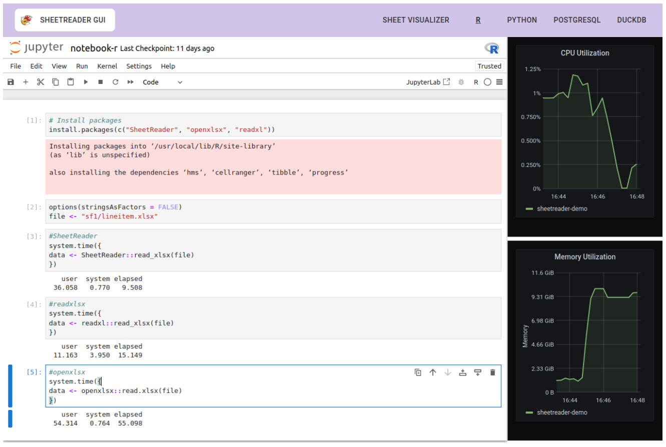
We include a spreadsheet visualizer (cf. Figure 1) that exposes the XLSX archive layout and the XML content of individual sheets. By inspecting the raw worksheet XML and estimating the size of its materialized tree, users can see why general-purpose XML parsers struggle: shared-string indirections and markup-heavy rows inflate the XML representation and lead to high memory overhead in practice.

4.3 SheetReader for Data Science Runtimes

To demonstrate SheetReader in data science contexts, our workbench provides notebooks for Python and R.

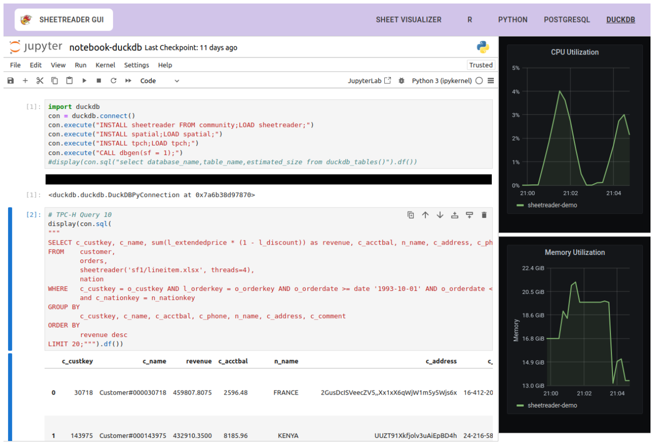
Libraries. To facilitate user interaction with SheetReader and demonstrate its integration within data science environments, our GUI offers two Jupyter notebooks with the Python and R kernels (cf. Figure 5). These notebooks contain predefined code snippets for loading spreadsheets into environment-specific data structures, specifically Python Pandas and R data frames, using SheetReader as well as state-of-the-art parsers, i.e., `openpyxl` and `calamine` for Python Pandas, and `openxlsx` and `readxl` for R. We provide real-world and synthetic spreadsheet files for the demonstration, and we also encourage the audience to experiment with their own files.

Performance. To showcase SheetReader's resource efficiency and improved runtime performance, the GUI includes two performance dashboards next to each notebook. These dashboards allow users to compare CPU and memory utilization across different parsing approaches. Specifically, users can observe the memory-intensive nature of state-of-the-art parsers, which materialize the entire XML tree before parsing. They can also compare SheetReader's two modes and observe how the *consecutive* mode makes spreadsheet parsing practical even on commodity laptops.



```
[1]: # Install packages
     install.packages(c("SheetReader", "openxlsx", "readxl"))

     Installing packages into '/usr/local/lib/R/site-library'
     (as 'lib' is unspecified)

     also installing the dependencies 'hms', 'cellranger', 'tibble', 'progress'

[2]: options(stringsAsFactors = FALSE)
     file <- "sf1/lineitem.xlsx"

[3]: #SheetReader
     system.time({
     data <- SheetReader::read_xlsx(file)
     })

        user  system elapsed
      36.058   0.770   9.508

[4]: #readxlsx
     system.time({
     data <- readxl::read_xlsx(file)
     })

        user  system elapsed
      11.163   3.950  15.149

[5]: #openxlsx
     system.time({
     data <- openxlsx::read.xlsx(file)
     })

        user  system elapsed
      54.314   0.764  55.098
```

Figure 5: UI showcasing SheetReader in an R Jupyter Notebook where users can load spreadsheets with different tools and compare live runtime/memory performance.



```
[1]: import duckdb
     con = duckdb.connect()
     con.execute("INSTALL sheetreader FROM community;LOAD sheetreader;")
     con.execute("INSTALL spatial;LOAD spatial;")
     con.execute("INSTALL tpch;LOAD tpch;")
     con.execute("CALL dbgen(sf = 1);")
     #display(con.sql("select database_name,table_name,estimated_size from duckdb_tables()").df())

[1]: <duckdb.duckdb.DuckDBPyConnection at 0x7a6b38d97870>

[2]: # TPC-H Query 10
     display(con.sql(
     """
     SELECT c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_ph
     FROM    customer,
             orders,
             sheetreader('sf1/lineitem.xlsx', threads=4),
             nation
     WHERE   c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate >= date '1993-10-01' AND o_orderdate <
             and c_nationkey = n_nationkey
     GROUP BY
             c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment
     ORDER BY
             revenue desc
     LIMIT 20;""").df())
```

	c_custkey	c_name	revenue	c_acctbal	n_name	c_address	c_
0	30718	Customer#000030718	459807.8075	2596.48	FRANCE	2GusDclSVeecZV5,,Xx1xX6qWjW1m5y5Wjs6x	16-412-20
1	143975	Customer#000143975	432910.3500	8185.96	KENYA	UUZT91Xkfjolv3uAiEpBD4h	24-216-58

Figure 6: UI showcasing SheetReader on DuckDB through a Notebook where users can write SQL on sheets and tables and compare live performance against baselines.

4.4 SheetReader for DBMSes

To illustrate SheetReader's use cases beyond data science, such as running SQL on spreadsheets and integrating them with DBMS tables, we provide notebooks for DuckDB and PostgreSQL.

SQL on Spreadsheets. The DBMS extensions allow running SQL queries over spreadsheets, a functionality not included out-of-the-box in modern spreadsheet systems. We demonstrate this functionality in our Jupyter Notebooks (cf. Figure 6), where users can efficiently execute SQL queries on spreadsheets with the in-process DuckDB DBMS. This capability is also available through our PostgreSQL extension, but requires a running server.

Data Integration. The DBMS extensions also enable users to integrate spreadsheets with tables stored in DBMSes. For example, with SheetReader's PostgreSQL (SQL/MED standard) extension, it is possible to register spreadsheets as foreign tables (cf. Figure 7), and even combine them with remote tables in cross-database settings when integrating data from different locations [5].

In this demonstration scenario, we guide the audience through PostgreSQL's SQL/MED and DuckDB's extension implementations, explain how SheetReader is exposed through these interfaces, and show how they enable users to interact with spreadsheets directly inside the DBMS environments. We demonstrate queries using the TPC-H dataset with both regular and foreign spreadsheet tables, and encourage visitors to bring their own spreadsheets to experiment with SheetReader interactively.

This demonstration scenario highlights the importance of fast spreadsheet loading in heterogeneous and cross-database analytics environments, where data from multiple systems must be integrated efficiently [4, 5].

```
      Table        |     Type      |                       fdw options
-------------------+---------------+------------------------------------------------------
 dbms_customer     | ordinary table |
 dbms_lineitem     | ordinary table |
 dbms_orders       | ordinary table |
 dbms_part         | ordinary table |
 dbms_partsupp     | ordinary table |
 dbms_supplier     | ordinary table |
 spreadsheet_nation | foreign table  | filepath=/spreadsheets/nation.xlsx,sheetname=nation
 spreadsheet_region | foreign table  | filepath=/spreadsheets/region.xlsx,sheetname=region
```

Figure 7: Integrating tables & spreadsheets in PostgreSQL.

Acknowledgments

We gratefully acknowledge funding from the German Federal Ministry of Education and Research under the grants BIFOLD25B and 01IS17052 (as part of the Software Campus project PolyDB).

References

[1] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin ChenChuan Chang, and Aditya Parameswaran. 2015. DataSpread: Unifying Databases and Spreadsheets. *PVLDB* 8, 12 (2015), 2000–2011.
[2] Sibei Chen, Yeye He, Weiwei Cui, Ju Fan, Song Ge, Haidong Zhang, Dongmei Zhang, and Surajit Chaudhuri. 2024. Auto-formula: Recommend formulas in spreadsheets using contrastive learning for table representations. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[3] Haralampos Gavriilidis. 2019. Computation offloading in jvm-based dataflow engines. In *BTW 2019–Workshopband*. Gesellschaft für Informatik, Bonn, 195–204.
[4] Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.
[5] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-situ cross-database query processing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2794–2807.
[6] Haralampos Gavriilidis, Felix Henze, Eleni Tzirita Zacharatou, and Volker Markl. 2023. SheetReader: Efficient specialized spreadsheet parsing. *Information Systems* 115 (2023), 102183.
[7] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *SIGMOD*. 883–899.
[8] Felix Henze, Haralampos Gavriilidis, Eleni Tzirita Zacharatou, and Volker Markl. 2022. Efficient Specialized Spreadsheet Parsing for Data Science. In *DOLAP (CEUR Workshop Proceedings, Vol. 3130)*. 41–50.
[9] Margaret G Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. 2006. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *WWW*. 93–102.
[10] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.
[11] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *PVLDB* 10, 10 (2017), 1118–1129.
[12] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant loading for main memory databases. *PVLDB* 6, 14 (2013), 1702–1713.
[13] Matthias Nicola and Jasmi John. 2003. XML parsing: a threat to database performance. In *CIKM*. 175–178.
[14] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter before you parse: Faster analytics on raw data with sparser. *PVLDB* 11, 11 (2018), 1576–1589.
[15] Sajjadur Rahman, Mangesh Bendre, Yuyang Liu, Shichu Zhu, Zhaoyuan Su, Karrie Karahalios, and Aditya G Parameswaran. 2021. NOAH: Interactive Spreadsheet Exploration with Dynamic Hierarchical Overviews. *PVLDB* 14, 6 (2021), 970–983.
[16] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *PVLDB* 13, 5 (2020), 616–628.
[17] Aske Wachs and Eleni Tzirita Zacharatou. 2024. Analysis of Geospatial Data Loading. In *DBTest*. 36–42.