Artifacts Available acm V1.1
Artifacts Evaluated acm Reusable V1.1
Results Reproduced acm V1.1

GIO: Generating Efficient Matrix and Frame Readers for Custom Data Formats by Example

SAEED FATHOLLAHZADEH, Graz University of Technology & Know-Center GmbH, Austria
MATTHIAS BOEHM*, Technische Universität Berlin, Germany

Data Scientists deal with a wide variety of file data formats and data representations. Probably the most difficult to handle are custom data formats that liberally define their own particular flat or nested structure with multiple custom delimiters, multi-line records, or undocumented semantics of attribute sequences, co-appearances, and repetitions. As a prerequisite for exploratory ML model training, data scientists need to map these data representations into regular frames or matrices. Unfortunately, existing tools and frameworks provide only limited support for aiding this process, which causes redundant manual efforts and unnecessary data quality issues. In this paper, we initiate work on automatic matrix and frame reader generation by example. A user provides a sample of raw text data and its mapped matrix or frame representation. Our GIO framework then first identifies the mapping rules from raw to structured data, and subsequently generates source code of an efficient, multi-threaded reader for reading full raw datasets of this format. In order to facilitate manual improvements, both the mapping rules, and generated reader can be modified as needed. Our experiments show that GIO is able to correctly identify the mapping rules for basic text formats like CSV, LibSVM, MatrixMarket; custom text formats from publishing, automotive, and health care; as well as various nested formats such as JSON and XML. Additionally, the automatically generated readers yield competitive performance compared to hand-coded readers and tuned libraries like RapidJSON.

CCS Concepts: • **Information systems → Database management system engines**; **Data scans**; **Record and block layout**; • **Theory of computation → Database query processing and optimization (theory)**.

Additional Key Words and Phrases: Raw Data; Custom Data Format; Efficient Readers; Data Loading

ACM Reference Format:
Saeed Fathollahzadeh and Matthias Boehm. 2023. GIO: Generating Efficient Matrix and Frame Readers for Custom Data Formats by Example. *Proc. ACM Manag. Data* 1, 2, Article 120 (June 2023), 26 pages. https://doi.org/10.1145/3589265

1 INTRODUCTION

The typical data science lifecycle is exploratory, where data scientists formulate hypotheses, integrate and clean the necessary data in order to build and evaluate predictive models [31, 36]. Data sourcing and integration often deals with files in open data formats, stored in cloud object storage or distributed file systems [79]. Common formats include text formats such as CSV, fixed-width text, JSON, and XML; as well as binary formats such as Parquet, HDF5, and Protobuf. Apart from these domain-agnostic, syntactic data formats, there are also a number of domain-specific, semantic

[*] This work was partially done at Graz University of Technology, Austria.

Authors' addresses: Saeed Fathollahzadeh, Graz University of Technology & Know-Center GmbH, Austria; Matthias Boehm, Technische Universität Berlin, Germany.

2836-6573/2023/6-ART120 $15.00
https://doi.org/10.1145/3589265

formats with similar structure but specific schemas. Examples are MDF (automotive measurements), HL7 (health care), and SWIFT (finance), as well as many custom formats.

Problem of Custom Data Formats: In many applications, custom data formats originate from systems and machines, whose data representation (e.g., logs) was not designed for data exchange and interoperability. Challenging characteristics of such data formats include flat or nested structure, optional key or positional attributes, multiple custom delimiters or prefixes, potentially multi-line records, undocumented semantics of attribute sequences, co-appearances, and repeating groups of attributes. Examples from our industrial partners—where custom data formats are a major hurdle for integrated data analysis—include semiconductor manufacturing, smart grid data management, paper production, and mixed, non-hazardous waste recycling. A common process of dealing with such data formats is taking a sample and writing custom extractors (for reading, parsing, ingestion, and mapping) to capture all relevant data in matrices or frames, which causes redundant manual efforts for extractor implementations and data quality issues due to brittle, insufficiently-tested extractors. Unfortunately, writing such extractors or readers—especially for complex custom formats—is still a painful process that is poorly supported in existing ML systems, libraries, and data-parallel computation frameworks.

Existing Raw Data Processing: Data and query processing of raw input files is an important yet challenging problem. The inspiring and highly-influential NoDB work (and RAW Labs) [9, 47] enables SQL query processing on CSV and JSON files, while providing high-performance via techniques such as positional maps for selective parsing, caching, and partitioning. In this context, Proteus [51] further enabled efficient JSON data extraction and query processing via source code generation. State-of-the-art in practice include custom hand-coded extractors, often using data-parallel frameworks such as Apache Spark [78], as well as SQL processing via SparkSQL [15] and JSON processing via RumbleDB [63], augmented with techniques for selective parsing [65]. However, reading and processing complex custom formats still requires custom, manually-created extractors, or even parsers generated via ANTLR/JavaCC and hand-crafted grammars. Existing work on raw data primarily focus on reading and query processing of simple, existing data formats, while automatic mapping identification and complex data handling remain open problems.

GIO Overview: In order to address the missing support for custom data formats, we initiate work on *automatic matrix/frame reader generation by example* and present our GIO framework. Given a sample of raw text data and a corresponding example frame or matrix that cover subsets of raw data values, we generate efficient readers for processing the full dataset (and future datasets of the same format). The conceptual basis of our framework are position (row/column) and value mapping functions, assembled into mapping rules. In a first identification step, GIO mines these mapping rules from the user-provided examples of raw and extracted data. In a second step, GIO generates source code for an efficient reader that implements the mapping rules. In contrast to schema matching [26] and mapping [44, 77]—which deal with schema-level correspondences extracted via schema- or instance-based matchers, and the generation of mapping programs—GIO handles raw, text-based data formats and their ingestion into matrices and frames for ML applications.

Contributions: Our primary contribution is the practical GIO[1] framework for matrix/frame reader generation by example of custom, text-based (flat or nested) data formats. Besides conceptualizing this new problem, we also provide novel techniques for efficient mapping identification by example, and efficiently generating efficient readers. Our technical contributions are:

[1] GIO is short for generated I/O primitives and honors Gio Wiederhold's contributions to compilers, data acquisition, information integration, and knowledge management. All code and experiments are available open source in Apache SystemDS (https://github.com/apache/systemds) and our reproducibility repository (https://github.com/damslab/reproducibility).

- *Problem Formulation:* As conceptual basis for this and future work, we first formulate the overall problem, and survey how existing data formats align with it in Section 2.
- *Mapping Identification:* We then introduce novel algorithms for identifying mapping rules from flat and nested raw data to matrices/frames in Section 3. This identification leverages prefix trees for efficient candidate evaluation.
- *Reader Generation:* We further describe a template-based code-generation approach for generating efficient matrix or frame readers in Section 4. These readers utilize multi-threading and selective parsing of projected attributes.
- *Experimental Evaluation:* Finally, we report on extensive experiments in Section 5. We use multiple basic and custom data formats, and compare with existing parallel readers.

2 BACKGROUND AND OBJECTIVES

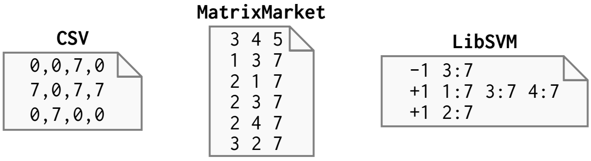
We first describe the background of custom data formats, and then formulate the problem of generating readers from user-provided examples, including identifying the mapping rules.

2.1 Custom Data Format Characteristics

Files in custom data formats are widely used inputs to ML pipelines and applications. For the sake of a clear presentation, we first describe the characteristics of commonly used ML data formats, show examples of custom data formats, and finally, summarize related challenges that need to be handled by generated I/O primitives.

ML Data Formats: ML Applications often consume their input data from consolidated, self-contained files. The most commonly used data formats are general-purpose formats such as CSV (comma-separated values) [69], JSON (javascript object notation) [72], XML [59], and Protobuf. Additionally, specialized matrix formats are used to capture specific characteristics such as sparsity and labels. Examples are MatrixMarket (coordinate or array format with different value types like real/integer, and different symmetry types) [2] and LibSVM (sparse row representation with labels) [34], but also scientific formats such as NetCDF and HDF5, where the latter uses B-trees for the efficient extraction of chunks. For large-scale data processing formats like Parquet, ORC, and Arrow are common as well. However, open text-based formats like CSV, MatrixMarket, LibSVM, JSON, and XML are largely preferred in practice in order to ensure interoperability (human readable, independent of existing systems and tools, and stable over time). Many ML systems already natively support these formats, but their structural concepts are reappearing in custom data formats as well.

Custom Data Formats: In contrast to general-purpose formats, we use the term custom data formats for specialized domain-specific formats (e.g., HL7 and SWIFT messages), tailor-made application-specific formats (e.g., vendor-specific machine logs), as well as specific schemas in general-purpose JSON or XML representations. Such formats are often designed without interoperability in mind or provide advantages in terms of flexibility and simplicity in their domain. Although there is a spectrum of formats without clear demarcation of custom data formats, we generally refer to specialized formats unsupported by most ML systems. In contrast, CSV, MatrixMarket, and LibSVM are widely-used, text-based formats with often-reused key structural elements. Figure 1


CSV

```
0,0,7,0
7,0,7,7
0,7,0,0
```

MatrixMarket

```
3 4 5
1 3 7
2 1 7
2 3 7
2 4 7
3 2 7
```

LibSVM

```
-1 3:7
+1 1:7 3:7 4:7
+1 2:7
```

Fig. 1. A 3 × 4 Matrix with 5 Non-Zero Values.

shows a 3-by-4 matrix with 5 non-zero values (constant value 7) in CSV, MatrixMarket, and LibSVM, where only LibSVM additionally contains labels (binary -1/+1 labels).

Key concepts are (1) one or multiple, single-/multi-character delimiters, (2) positional or prefix encodings of cell positions, and (3) mixed data and metadata. For example, the first row in MatrixMarket represents the metadata of numbers of rows, columns, and number of non-zero values; followed by 5 triples of (row column value) for these non-zero values. Many custom data formats have additional structure. Figure 2 shows three AMiner publications [73] as an example. Here, we have additionally (4) custom prefixes (e.g., #* for paper title, #t for year, and #c for venue), (5) lists of authors and references, and (6) multi-line records. Currently, such custom data formats are only poorly supported in existing systems, often requiring reading data as plain text and then parsing records via custom user-defined functions.

```
#index 2015101    ← beginning of record #1
#* NoDB: efficient query execution on raw data files
#@ Ioannis Alagiannis; Renata Borovica; Anastasia Ailamaki
#o EPFL Switzerland; EPFL Switzerland; EPFL Switzerland
#t 2015
#c VLDB Conference
#% ... unlimited set of references
#! As data collections become larger and larger, data loading evolves to a major bottleneck. ...
#index 2019102    ← beginning of record #2
#* Pangea: Monolithic Distributed Storage for Data Analytics
#@ Jia Zou; Arun Iyengar; Chris Jermaine
#o Rice University; Rice University; Rice University
#t 2019
#c VLDB Conference
#% ... unlimited set of references
#! Storage and memory systems for modern data analytics are heavily layered, managing shared ...
#index 2018103    ← beginning of record #3
#* Filter Before You Parse: Faster Analytics on Raw Data
#@ Shoumik Palkar; Firas Abuzaid; Matei Zaharia
#o Stanford InfoLab; Stanford InfoLab; Databricks Inc
#t 2018
#c VLDB Endowment
#% ... unlimited set of references
#! Exploratory big data applications often run on raw unstructured or semi-structured data ...
```

Fig. 2. AMiner Publication Dataset: Three Records.

Additional Challenges: Although some metadata is included, other properties have to be inferred from the given example datasets. First, the data might include extra values that are not taken over into the target matrix or frame representation. Other attributes are optional and need to be treated as such during reading. Second, readers might have to deal with variable-length and alternative textual representations. For example, "70", "70.0", and "7e1" should all be recognized as the same value. Third, attribute values with keys or explicit position encoding might appear in the sample out-of-order and at different hierarchy levels, which requires arbitrary row- and column-oriented mappings as well as handling path expressions in nested representations. Fourth, readers need to infer the dimensions (number of rows and columns) as well as sparsity for output pre-allocation. These characteristics render the generation of readers for custom formats a very challenging problem, not handled by existing JSON/XML parsers alone.

2.2 Reader Generation Problem

Given a custom text-based dataset D, and user-provided examples of raw and target data, our goal is to generate a matrix or frame reader for reading the full dataset and other data of this format. The user-provided examples have the following structure:

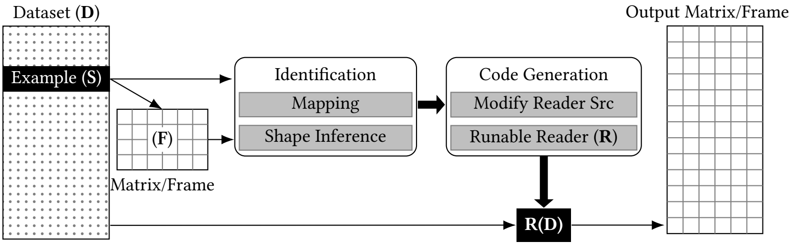


Fig. 3. Reader Generation Workflow in GIO.

- *Sample Raw (S) Input:* Let $\mathbf{S} = \{S_1, S_2, \ldots, S_l\}$ be a list of input strings (i.e., selected rows of the input dataset D), and be stored as a newline-separated text file. We denote S_i as a vector of characters of size $r_i = |S_i|$.
- *Sample Matrix/Frame (F) Input:* Let $\mathbf{F} = \{F_1, F_2, \ldots, F_n\}$ be a sample matrix or frame, corresponding to S with n records and the following schema: `F(valueType`$_1$`, valueType`$_2$`, . . . , valueType`$_m$`)`, where `valueType`$_i$ $\in$ {`I32, I64, Bool, String, FP32, FP64`}. Thus, $\mathbf{F}$ has dimensions $n \times m$ and $\mathbf{F}_{ij}$ denotes the cell value of the ith row and jth cell.

Problem Formulation: Based on the provided sample raw and matrix/frame inputs S and **F**, we aim to generate a reader for the entire dataset D. For the sake of flexibility in practice, we split this problem into two sub-problems with well-defined intermediates as shown in Figure 3. First, given S and **F**, we aim to identify a valid set of mapping rules $\mathcal{M}$ that capture the mapping of values from S to **F**. The mapping rules $\mathcal{M}$ are *valid* if and only if $\mathrm{S} \xrightarrow{\mathcal{M}} \mathbf{F}$, that is, $\mathcal{M}$ logically applied to S yields exactly **F**. Second, given $\mathcal{M}$, we aim to generate a reader that can read S into **F** and be applied to any dataset of the same format like S. The user API of GIO accordingly comprises three functions:

- `M = gio_identify(S,F)` (identification),
- `R = gio_codegen(M)` (reader generation), and
- `F2 = R.read(D)` (reader usage on full or different datasets).

This separation into sub-problems simplifies testing or debugging, and it enables optionally handcrafting or fine-tuning the mapping rules $\mathcal{M}$ as input to the reader generation process.

2.3 Mapping Rules

There are two types of essential mapping rules. First, *shape inference functions* allow to infer the dimensions $n \times m$ (and optionally sparsity) of **F** from D, so a generated reader can pre-allocate the matrix or frame accordingly. Second, *cell value mapping functions* encode how the values are extracted from S or D and placed in $\mathbf{F}_{ij}$. Both types of functions can be parameterized with values such as the delimiters, keys, path expressions, and offsets. Additional types of mapping rules is an interesting direction for future work.

Shape Inference Functions: A set of shape inference functions allows to determine the number of rows and columns, and optionally estimate the sparsity, for new datasets D of the given custom data format. The types and parameters of these functions need to be discovered from the sampled raw S and target matrix/frame **F**. In detail, we currently support the following functions:

- *Identity:* Infers a dimension by the number of data items produced by applying a specific delimiter (e.g., $n = |D|$ with del=\n for CSV or `"#index"` for AMiner; or $m = |D_1|$).
- *Constants:* Infers a dimension by a fixed constant or via existing metadata at a fixed location in D (e.g., MatrixMarket).
- *Max:* Infers a dimension as the maximum of explicitly given cell indexes at specific locations.

- *Stack:* Infers a dimension by sequential counting of entries at a certain hierarchy level via a stack of begin and end delimiters (e.g., in XML or other nested representations).
- *Special:* As general fallback for complex shapes, we provide special arithmetic functions for deriving dimensions. For example, consider a single-line $\mathbf{S} = \{1, 2, 3, 3, 4, 4, 5, 6, 7, 8, 9\}$ and the following different **F** targets:

$$F_1 = \begin{bmatrix} 1\ 2\ 3 \\ 4\ 5\ 6 \\ 7\ 8\ 9 \end{bmatrix} \quad F_2 = \begin{bmatrix} 1\ 4\ 7 \\ 2\ 5\ 8 \\ 3\ 6\ 9 \end{bmatrix} \quad F_3 = \begin{bmatrix} 1\ 2\ 3\ 4\ 5 \\ 6\ 7\ 8\ 9\ 0 \end{bmatrix} \quad F_4 = \begin{bmatrix} 1\ 3\ 5\ 7\ 9 \\ 2\ 4\ 6\ 8\ 0 \end{bmatrix}$$

  The corresponding shape functions are $n = m = \sqrt{|\mathbf{S}|}$ for $\mathbf{F}_1$ and $\mathbf{F}_2$, as well as $n = \lceil |\mathbf{S}|/5 \rceil$ and $m = 5$ for $\mathbf{F}_3$ and $\mathbf{F}_4$ (both with row- and column-major value mappings).

Cell Value Mapping Functions: Similar to the shape inference functions, mapping rules of values in S and F are encoded with coarse-grained, parameterized mapping functions. Conceptually, we need to map all values in **F** onto S, and infer rules that cover all values (i.e., no value violates the rules). With duplicate values many valid mappings are possible, but based on the principle of minimality [27] one should choose the simplest rules. The values of **F** might be fully existing in S (e.g., CSV), partially existing in S (e.g., MatrixMarket-symmetric where values in S are replicated once in **F**), or generated from constants (e.g., defaults). In detail, we support the following functions:

- *Identity:* F_{ij} values appear grouped and sorted in S. This means, the positions are identical to the position in S after splitting by appropriate delimiters.
- *Exist:* The indexes of values in **F** are explicitly included in S before or after **F**'s cell values.
- *Scattered-Sequential:* Values of a record in **F** are collected from single-/multi-line strings in S.

Note that there are separate functions for row and column mappings. For example, CSV has row and column Identity mappings, whereas LibSVM has row Identity and column Exist mapping rules. In the following, we describe in detail how GIO identifies these mapping rules and then generates efficient readers based on these rules.

3 MAPPING IDENTIFICATION

We obtain the mapping rules for a given S and **F** , by first collecting detailed mapping information for all cells values in **F** and then synthesizing the coarse-grained rules $\mathcal{M}$. In this section, we describe key algorithms for cell value mapping, efficient pattern matching, and the overall identification algorithm along with illustrative examples for conveying the underlying intuitions.

3.1 Cell Value Mapping

As a basis for the overall identification algorithm, we aim to extract a detailed cell value mapping. This mapping indicates indexes where each value in **F** can be found in S.

Mapping Algorithm. The target matrix or frame **F** explicitly contains all cell values $\mathbf{F}_{ij}$ as well as their locations in terms of row and column indexes i and j. Algorithm 1 maps these values to strings in S by considering valid mapping alternatives, and returns the most likely mapping. In Lines 3-6 of Algorithm 1, we construct indexes to group values of each line in S into numeric, boolean, and string values via our indexing Algorithm 2. Here, we assume that records in S (single or multiple rows) can be separated by a single- or multi-character delimiter. To obtain the most likely mapping, we iterate over all mapping orders from **F** to S (see Line 7), compute each mapping (see Lines 8-19), and finally score the mappings via selectTrustedMapping to return the best mapping. The Algorithm selectTrustedMapping (not shown due to its mechanical complexity) follows the mentioned principle of minimality [27] by assigning scores according to the simplicity of mapping functions and consistency of row/column cell mappings. Our indexing Algorithm 2, constructs

Algorithm 1 MAP(S, F)

```
Input: Sample Raw S , Sample Frame F
Output: Mapping $\mathcal{M}^*$
n ← nrow(F);   m ← ncol(F);   $\mathcal{M} \leftarrow \emptyset$
$S'_{\text{numeric}} \leftarrow \emptyset$;   $S'_{\text{bool}} \leftarrow \emptyset$                // multimap
for $i$ in 1 : nline(S) do
    (N, B) ← INDEX($S_i$, $i$)
    $S'_{numeric} \leftarrow S'_{numeric} \cup N$
    $S'_{bool} \leftarrow S'_{bool} \cup B$
for $r$ in |S × F| do                                   // any order of F's values on S
    M = $[null]_{n \times m}$                          // $M_{n \times m}$ matrix of (x,y) pairs
    for $i$ in 1 : $n$ do
        for $j$ in 1 : $m$ do
            if $F_{ij} \notin \{null, 0\}$ then
                if $valueType_j \in$ {I32, I64, FP32, FP64} then
                    $(p, l) = F_{ij} \in S'_{numeric}$
                else if $valueType_j \in$ {Bool} then
                    $(p, l) = F_{ij} \in S'_{bool}$
                else
                    $s_l^{(p, p+\alpha)} \leftarrow F_{ij} \in S$
                $M_{ij} \leftarrow (p, l)$                   // save $(p, l)$ pair at $M_{ij}$
    $\mathcal{M} \leftarrow \{\mathcal{M}\} \cup M$
return SELECTTRUSTEDMAPPING($\mathcal{M}$)
```

Algorithm 2 INDEX(Raw, L)

```
Input: A String Raw, Line Number L
Output: Numeric Multimap $\mathcal{N}$, Boolean Multimap $\mathcal{B}$
 1: $\mathcal{N} \leftarrow \emptyset$;   $\mathcal{B} \leftarrow \emptyset$;   l ← $len$(Raw)
 2: bn ← $Bitset$(l);   bb ← $Bitset$(l)                 // Empty bitsets with size l
 3: for $i$ in 1 : l do
 4:     if GETSTRINGCHAR(Raw, $i$) ∈ {0-9,+,-,.,',',E} then
 5:         bn[$i$] ← 1
 6:     if GETSTRINGCHAR(Raw, $i$) ∈ {0,1,T,F} then
 7:         bb[$i$] ← 1
 8: [(ns, np)] ← GETALLSEQUENCESETSUBSTRINGS(Raw, bn)
 9: [(bs, bp)] ← GETALLSEQUENCESETSUBSTRINGS(Raw, bb)
10: $\mathcal{N}$ ← [PARSESTRINGTONUMBER(ns), (np, L)]
11: $\mathcal{B}$ ← [PARSESTRINGTOBOOLEAN(bs), (bp, L)]
12: return $\mathcal{N}, \mathcal{B}$
```

indexes by 15 characters {0-9, '+', '-', '.', ',', 'E'} (see Line 4) that might appear in numeric values, 6 keywords {'0', '1', 'T', 'F', 'True', 'False'} (see Line 6) that might represent boolean values, and any characters can appear in string values. Our cell value mapping algorithm does not yet support special values like NaN, -Inf, +Inf, which is possible but requires additional dictionaries for common variants and framework extensions for NaN-awareness (e.g., on value comparisons). After indexing the raw string by two bitmaps in Lines 3-7 of Algorithm 2, getAllSequenceSetSubstrings extracts the actual (string of numeric or boolean) values from the raw string (see Lines 8-9). We parse extracted strings of actual values according to their inferred types, and use a hash-map with bag semantics (duplicate keys allowed) as the underlying data structure. Actual values are the keys, mapping to individual (char offset, line number) pairs.

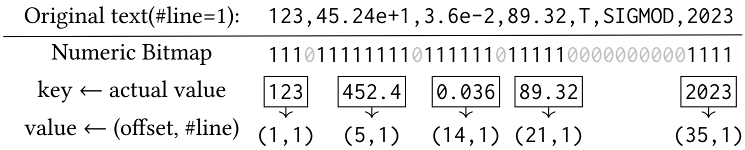

Original text(#line=1): 123,45.24e+1,3.6e-2,89.32,T,SIGMOD,2023
Numeric Bitmap 111011111111011111101111100000000001111
key ← actual value
123
452.4
0.036
89.32
2023
value ← (offset, #line)
(1,1)
(5,1)
(14,1)
(21,1)
(35,1)

Fig. 4. Example Numeric String Index.

Example 1 (Index Construction). *Figure 4 shows an example of the index construction for a raw string record (top). Here, the bitmap (second row) indicates which characters might be part of numeric values, encoded as runs of set bits. Additionally, the last two rows show the resulting value to position mapping. Furthermore, assume the following* $\mathbf{S}$ *(in CSV) and* $\mathbf{F}$ *along with four alternative mappings* M *(each containing (char-offset, row-index) pairs for all values in* $\mathbf{F}$*).*

$$\mathbf{S}=\begin{bmatrix}1,2,3,3\\4,4,5,6\\7,8,9,0\end{bmatrix}\quad \mathbf{F}=\begin{bmatrix}1&3\\4&5\\7&9\end{bmatrix}\quad \mathbf{M}=\Bigg\{\underbrace{\begin{bmatrix}(1,1)&(1,5)\\(2,1)&(2,5)\\(3,1)&(3,5)\end{bmatrix}}_{M_1},\underbrace{\begin{bmatrix}(1,1)&(1,7)\\(2,1)&(2,5)\\(3,1)&(3,5)\end{bmatrix}}_{M_2},\underbrace{\begin{bmatrix}(1,1)&(1,5)\\(2,3)&(2,5)\\(3,1)&(3,5)\end{bmatrix}}_{M_3},\underbrace{\begin{bmatrix}(1,1)&(1,7)\\(2,3)&(2,5)\\(3,1)&(3,5)\end{bmatrix}}_{M_4}\Bigg\}$$

The matrix $\mathbf{F}$ *projects columns 1 and 3 from the four column* $\mathbf{S}$ *and the values 3 and 4 appear each twice in* $\mathbf{S}$*. Accordingly,* $\mathbf{M}$ *contains four different mappings, and we select* $\mathbf{M}^* = M_1$ *as the best because of consistent character and column alignment.*

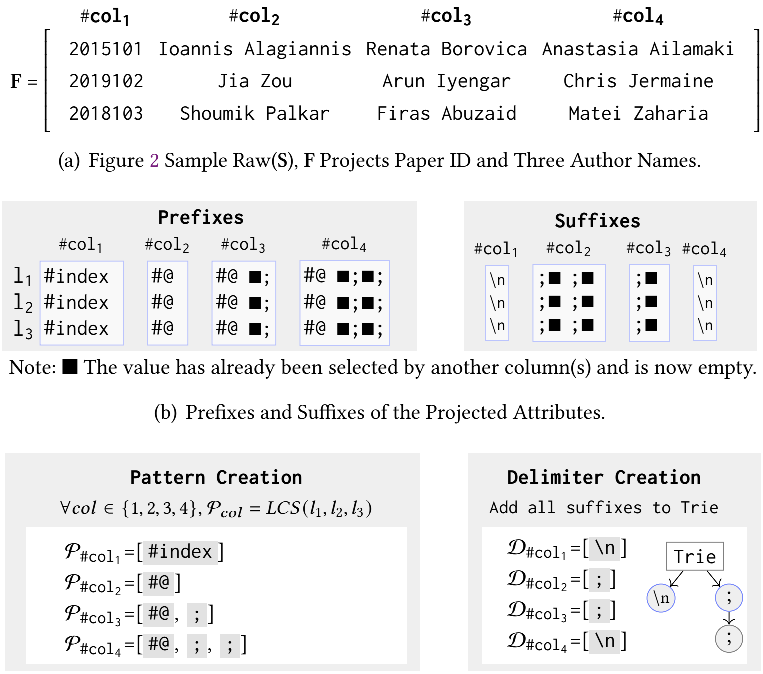
3.2 Pattern Matching

Pattern creation and matching during identification are essential for extracting delimiters and cell values. We leverage prefix trees (tries) [29, 33, 57, 62] for efficient matching, which reduced the identification time from super-linear to linear in the sample size.

Algorithm 3 pattern($\mathbf{P}, \mathbf{S}$)

```
Input: List of Prefix Strings $\mathbf{P}$, List of Suffix Strings $\mathbf{S}$
Output: List of Strings $\mathcal{P}$, Set of Delimiters $\mathcal{D}$
$C \leftarrow$ getAllCommonSubsequences($\mathbf{P}$)          // array of list
$trie \leftarrow$ Initial Prefix Tree                            // prepare prefix tree for delimiters
for $c \in C$ do                                                 // iterate over common sub-strings
    if ($\forall \mathbf{P}$, c reach the end of string then
        $\mathcal{P} \leftarrow c$;   break
for $s \in \mathbf{S}$ do                                        // insert all suffixes into prefix tree
    $trie.insert(s)$
$\mathcal{D} \leftarrow$ getRootNodeChildren($trie$)
return $\mathcal{P}, \mathcal{D}$
```

Pattern Creation: Algorithm 3 takes a list of strings and finds patterns of common sub-sequences as candidates of delimiters. This algorithm is invoked multiple times in the overall identification


#col1
#col2
#col3
#col4
F =
2015101 Ioannis Alagiannis Renata Borovica Anastasia Ailamaki
2019102 Jia Zou Arun Iyengar Chris Jermaine
2018103 Shoumik Palkar Firas Abuzaid Matei Zaharia
Prefixes
l1 #index #@ #@ ■; #@ ■;■;
l2 #index #@ #@ ■; #@ ■;■;
l3 #index #@ #@ ■; #@ ■;■;
Suffixes
\n ;■ ;■ ;■ \n
\n ;■ ;■ ;■ \n
\n ;■ ;■ ;■ \n
Pattern Creation
∀col ∈ {1, 2, 3, 4}, P_col = LCS(l1, l2, l3)
P#col1 =[#index]
P#col2 =[#@]
P#col3 =[#@ , ;]
P#col4 =[#@ , ; , ;]
Delimiter Creation
Add all suffixes to Trie
D#col1 =[\n]
D#col2 =[;]
D#col3 =[;]
D#col4 =[\n]
Trie
\n
;
;

(a) Figure 2 Sample Raw(S), **F** Projects Paper ID and Three Author Names.

Note: ■ The value has already been selected by another column(s) and is now empty.

(b) Prefixes and Suffixes of the Projected Attributes.

(c) Pattern and Delimiter Creation for Projected Attributes.

Fig. 5. Example of Pattern Creation for Publication.

algorithm, with different prefixes and suffixes. At its core, we compare all input strings (as lists of characters) and compute the "intersection" in terms sub-sequences of characters that appear in all strings (see Line 1). This intersection function needs to consider the order of characters, and thus, it is a modified version of the Longest Common Subsequence (LCS) [25] that keeps all common sub-sequences. We take lists of string prefixes as input, try to find a pattern in the prefix list (see Lines 3-5), and if no common sub-sequence is found, invoke the PATTERN algorithm multiple times with different prefixes and suffixes. The order of built pattern is kept available in the additional meta data (whose explanation we otherwise skip for sake of presentation simplicity). We are also storing the indicators of the end of values by a list of single characters or more complex patterns. Subsequently, we initialize a prefix-tree with an empty string (Line 2), and insert the suffixes (Lines 6-7). Finally, we select and return all children of the root node of the prefix tree as delimiters, which represents the common sub-sequence at the start of prefixes and end of suffixes (Line 8).

Delimiters in Values: Ideally, non-projected values would not contain any delimiters, but these are valid data characteristics we aim to support. There are issues if such values contain a full or partial delimiter sequence. To overcome this issue, we enumerate all topological sub-sequences of strings and select the correct of them. For example, assume we want to find the intersection of `a,b,”c,d”,e` and `f,g,”h”,i`. Here, $\mathcal{P}_1$ = [`,`, `,”`, `,`] and $\mathcal{P}_2$ = [`,`, `,”`, `",`] are two topological sub-sequences. Our correction algorithm then selects patterns among $\mathcal{P}_1$ and $\mathcal{P}_2$ by testing these patterns on the source strings. Pattern $\mathcal{P}_1$ cannot reach the end of strings but $\mathcal{P}_2$ can reach both of them. The only drawback of this technique is that all prefix/suffix strings contain delimiters as values for non-projected columns, which later affects the pattern matching in generated source code. However, that way we can handle things like CSV quoting but in a very generic manner (without the usual stack-based CSV parsing for quoted delimiters and quotes).

Example 2 (Pattern Creation). *An example of pattern and delimiter construction for the AMiner publication dataset is shown in Figure 5. Assume the example from Figure 2 is a sample raw* $\mathbf{S}$ *, and* $\mathbf{F}$ *—shown in Figure 5(a))—comprises four columns of the paper ID (#col*$_1$*) and the three first authors (#col*$_2$*-#col*$_4$*) of each paper. In Figure 5(b), we first map* $\mathbf{F}_{ij}$ *values on* $\mathbf{S}$ *and then find the prefixes and suffixes of each column individually. We left some strings blank in the prefixes and suffixes because they are already selected in other columns. With Algorithm 3, we obtain the LCS patterns for each column as well as value delimiters in a prefix tree (trie) as shown in Figure 5(c).*

Syntactic Characters: We use the notion of syntactic characters as natural, but superfluous separators such as the fixed characters `{`, `}`, `[`, `]`, . . . in JSON. Our mined delimiters would include these syntactic characters as well, which is not just unnecessary but may even create incorrect patterns. For the sake of generality, we are not explicitly excluding fixed characters but different attribute orders help eliminate most of these from the rule sets.

3.3 Overall Identification Algorithm

Putting it all together, we describe the overall identification algorithm and its remaining limitations.

Identification Algorithm: Our overall identification algorithm is shown in Algorithm 4, which takes $\mathbf{S}$ and $\mathbf{F}$ as input and returns the coarse-grained, parameterized mapping rules $\mathcal{M}$ as union of row, column, and value mapping functions. $\mathcal{M}$ also contains shape inference functions which are, however, not explicitly shown here. In detail, the algorithm comprises five steps. First, we obtain the detailed mappings for cell values in $\mathbf{F}$—but also row and column indexes—with Algorithm 1 from Section 3.1. In this context, we also apply basic pre-processing—for checking properties of $\mathbf{F}$ like Skew, Symmetric, Skew-Symmetric, as well as some simple patterns—whose overhead was negligible in our experiments. Second, we evaluate the detailed mappings and derive mapping functions (as defined in Section 2.3) and their basic parameters. Valid alternatives are scored by ranked simplicity. Third and fourth, we enumerate prefix and suffix strings—in a mapping-function-specific manner—for rows and columns respectively, and create patterns for the extraction of delimiters and cell values with Algorithm 3 from Section 3.2. Fifth and finally, we obtain the concrete patterns as parameters of the mapping rules and return these rules accordingly.

Limitations: Despite good generalization for different structural properties of custom data formats, GIO has several limitations, which also directly characterize the class of supported formats. The remaining high-level limitations are:

- *No Unseen Patterns:* GIO uses prefixes and suffixes to build patterns for begin and end positions of values, and then extracts these values. This extraction is heavily dependent on the input samples and does not support unseen patterns.
- *No Mixed Data Formats:* The mapping identification currently extracts homogeneous mapping functions for the entire dataset. Thus, we do not support data comprised of a mix of different formats (e.g., sections in CSV and JSON).
- *No Query Processing:* Apart from index mapping and projections, GIO does not support query processing such as selections (row filtering) or aggregations to derive the matrix/frame representations from the raw input data.

Addressing these remaining limitations is interesting future work.

4 READER GENERATION

After having successfully identified the mapping rules for a custom data format (and potential manual refinements), GIO efficiently generates source code for efficiently reading datasets encoded

Algorithm 4 IDENTIFICATION(S, F)

```
Input: Sample Raw S , Sample Frame F
Output: $\mathcal{R}$: a triple (structure, pattern, and delimiter) for row index, $\mathcal{C}$: col index structure and a
    list of (pattern, delimiter) pairs, $\mathcal{V}$: col value supplier and a list of (pattern, delimiter) pairs
$\mathrm{n} \leftarrow \mathrm{nrow}(\mathbf{F})$;    $\mathrm{m} \leftarrow \mathrm{ncol}(\mathbf{F})$
// a) map value of F, row index RI, and col index CI on S
$\mathrm{M}^* \leftarrow \textsc{map}(\mathrm{S}, \mathrm{F})$
$\mathrm{M}^{r*} \leftarrow \textsc{map}(\mathrm{S}, \mathbf{RI})$        // 2d-Array integers $(n \times m)$ where $\mathbf{RI_{ij}} = \mathbf{i}$
$\mathrm{M}^{c*} \leftarrow \textsc{map}(\mathrm{S}, \mathbf{CI})$        // 2d-Array integers $(n \times m)$ where $\mathbf{CI_{ij}} = \mathbf{j}$
// b) check data supplier and index structures
$\mathcal{V}^{\text{supplier}} \leftarrow \mathcal{D}(M^*)$
$\mathcal{R}^{\text{structure}} \leftarrow \mathcal{I}_{row}(\mathrm{M}^*, \mathrm{M}^{r*})$
$\mathcal{C}^{\text{structure}} \leftarrow \mathcal{I}_{col}(\mathrm{M}^*, \mathrm{M}^{c*})$
prefix $\leftarrow \emptyset$; suffix $\leftarrow \emptyset$        // 2d-Array strings $(n \times m)$ for $M^*$
prefix$^r \leftarrow \emptyset$; suffix$^r \leftarrow \emptyset$        // 2d-Array strings $(nm \times 1)$ for $M^{r*}$
prefix$^c \leftarrow \emptyset$; suffix$^c \leftarrow \emptyset$        // 2d-Array strings $(n \times m)$ for $M^{c*}$
// c) create pattern for row index counter
if $\mathcal{R}^{\text{structure}} \in$ {Constants, Max} then
    [prefix$^r$, suffix$^r$] $\leftarrow$ GETPREFIXANDSUFFIX(S, $\mathrm{M}^{r*}$)
    $\mathcal{R}^{(\text{pattern, delimiter})} \leftarrow$ PATTERN(prefix$^r$, suffix$^r$)
else if $\mathcal{R}^{\text{structure}} \in$ {Stack} then
    $y \leftarrow \mathcal{V}^{\text{rows}}$        // 2d-Array ints $(n \times m)$, lines on S are for record
    for $i$ in n do
        $[\min_i, \max_i] \leftarrow \textsc{minmax}(y[i,])$
        $\text{prefix}^r_i \leftarrow \bigcup_{l=\max_{i-1}}^{\min_{i+1}} \mathrm{S}_l$
    $\mathcal{R}^{(\text{pattern, delimiter})} \leftarrow$ PATTERN(prefix$^r$, $\emptyset$)
// d) create pattern for col index
if $\mathcal{C}^{\text{structure}} \in$ {Constants, Max} then
    [prefix$^c$, suffix$^c$] $\leftarrow$ GETPREFIXANDSUFFIX(S, $\mathrm{M}^{c*}$)
    for $j$ in m do        // build pattern for each column index
        pc $\leftarrow$ prefix$^c[, j]$
        sc $\leftarrow$ suffix$^c[, j]$
        $\mathcal{C}^{(\text{pattern, delimiter})} \leftarrow$ PATTERN(pc, sc)
// e) create pattern for cell value extraction
if $\mathcal{V}^{\text{supplier}} \in$ {Identity, Scattered-Sequential} then
    [prefix, suffix] $\leftarrow$ GETPREFIXANDSUFFIX(S, $\mathrm{M}^*$)
    for $j$ in m do        // pattern for each column of matrix/frame
        pc $\leftarrow$ prefix$[, j]$
        sc $\leftarrow$ suffix$[, j]$
        $\mathcal{V}^{(\text{pattern, delimiter})} \leftarrow$ PATTERN(pc, sc)
return $\mathcal{R}^{(\text{structure, pattern, delimiter})}$,
       $\mathcal{C}^{(\text{structure,}[(\text{pattern, delimiter})]_{1\times m})}$,
       $\mathcal{V}^{(\text{supplier,}[(\text{pattern, delimiter})]_{1\times m})}$
```

in this format. In this section, we describe the template-based code generation, key techniques for efficient pattern matching, and example templates and generated code.

4.1 Template-based Code Generation

Our code generation approach relies on templates for the reader skeleton, parsing primitives, conditions, path expressions, and value indexing. These templates are instantiated and hierarchically composed according to the passed mapping rules $\mathcal{M}$.

Code Templates: Figure 6(a) shows the main skeleton template for a single-threaded frame reader, while multi-threaded readers parallelize over blocks of rows. This template has three main parts. First, a pre-pass can be instantiated for obtaining additional metadata from the data (e.g., dimensions or row block offsets). Second, we infer the dimensions, estimate sparsity, and allocate an in-memory frame block. Third, we iterate over records of the raw dataset, parse and insert the data into the frame. We separately generate the code for row index parsing, column index and value parsing, and subsequently instantiate them into the main template. Finally, we materialize both the Java source code (for manual fine-tuning) as well as the compiled class files. Different backends for generating code in other programming languages and exploiting SIMD instruction level parallelism are interesting future work.

Row Indexing Code: The readers can determine the number of rows during the pre-pass and thus, pre-allocate the matrix or frame upfront. Figure 6(b) shows the row indexing template and related code generation, which instantiates code according to the mapping functions. These functions also carry necessary metadata such as the delimiters and suffix patterns, which allows iterating over rows and incrementing or parsing the row indexes. For Exist functions (in Lines 6-11), we have to find the target row index before extracting the cell values. We do so by scanning a sequence of patterns on the record, allowing to distinguish the next values. For scattered-sequential mapping functions—which include the handling of multi-row records—we generate a more complex finite state machine (FSM) into the pre-pass for keeping track of passed records and keys, and increment the row indexes in the inner loop accordingly.

Column Index and Value Code: The generated cell code extracts column indexes and values for projected attributes from each record. During identification, dedicated key patterns can be created for the individual columns, allowing attributes to appear in different orderings while utilizing local extraction prefixes and suffixes. We instantiate the related code templates according to the types of mapping functions, and with the goal of efficient pattern matching and attribute projections.

4.2 Pattern Matching Approaches

For efficient pattern matching, we introduce three dedicated code generation approaches that utilize (1) nested conditions derived from a prefix tree, (2) a regular loop with sequential string matching, as well as (3) compiled regular expressions.

Cell Value by Nested Conditions: Our first approach is to build a prefix tree for all patterns and generate code with nested conditions for a tailored pipeline of parsing, extraction, and type handling. For both flat and nested data, common sub-paths (part of multiple patterns) are extracted only once, with group aborts if a sub-path does not exist.

Example 3 (Pattern Matching via Nested Conditions). *Figure 7 shows an example of the cell value extraction via compiled nested conditions. For the sake of presentation, we generated code for the AMiner publication dataset from which we projected four columns of type $INT64$, and String as shown in Figure 5. The overlapping patterns (see Figure 5(c)) are compactly represented in the prefix tree. By utilizing the prefixes and suffixes, individual values can be locally extracted without scanning the entire record. Figure 7 shows the built prefix tree (on the right), comprising 3 levels and 4 column patterns at levels 1 to 3—for which the reader extracts values—as well as the generated code with nested conditions (on the left). This code is tailored for dense access, and skips non-projected attributes without parsing (which is an expensive iterative procedure for floating point values).*

```
GenericTemplate(ℛ, 𝒞, 𝒱)
 [srcInitRow, srcRow] = GenerateRowCode(ℛ);
 [srcInitCol, srcCol] = GenerateColCode(𝒱, 𝒯);
 src = " InitFile(); " +
       " FrameBlock ReadBlock( BufferReader br ) {
       " pro = Estimation(br, ℛ, 𝒞, 𝒱); " +
       " FrameBlock fb = new FrameBlock(pro); " +
            srcInitRow + // row pre-source
            srcInitCol + // col pre-source
        " for each (record r in br) { " +
            srcRow + // row index counter
            srcCol + // extract and parse cell value
        " } "
        " return fb; } ";
 return src;
```

(a) Skeleton Code Template

```
GenerateRowCode(ℛ)
srcInit = "   ";   src = "   ";
if (ℛ_structure == Identity)
  srcInit = " if (ℛ_pattern ∈ r) "+
            "   rowIndex++; ";
else if (ℛ_structure == Constraint || ℛ_structure == Max)
  srcInit = " _index = 0; " +
            " for (key k ∈ ℛ_pattern) " +
            "   _index = r.IndexOf(k, _index); " +
            " _end = r.IndexOf(ℛ_delimiter, _index); " +
            " _text = r.Substring(_index, _end); " +
            " rowIndex = ParseInt(_text); ";
else if (ℛ_structure == Stack)
  srcInit = " _bList = [];   _eList = []; " +
            " for (record r ∈ br) { " +
            "  _index = 0;   _end = 0; " +
            "  while (_index != 0) { " +
            "   _index = r.IndexOf(ℛ_pattern[0], _index); " +
            "   if(_index != 0)  " +
            "    _blis.append(Pair(r.index, _index)); } " +
            "  while (_end != 0) { " +
            "   _end = r.IndexOf(ℛ_pattern[1], _end); " +
            "   if(_end != 0)  " +
            "    _eList.append(Pair(r.index, _end));  }} " +
            " _rIndexes = [];   _stack = Stack(); " +
            " for (int i=0, j=0; i < min(len(_bList), len(_eList));) { " +
            "  if(_bListi] < _eList[j])   _stack.push(_bList[i++]); " +
            "  else { " +
            "   _index = _stack.pop(); " +
            "   if(_stack.empty()) " +
            "    _rIndexes.append(Pair(_index, _eList[j++])); } } " +
  src = " if(_rIndexes[rowIndex].key <= r.index && " +
        "   r.index <= _rIndexes[rowIndex].val) { " +
        "  _rStr += getRecordStr(r, r.index, _rIndexes[rowIndex]); " +
        "  continue; } " +
        " else  { r = _rStr;   _rStr = "";   rowIndex++; } "
return srcInit, src;
```

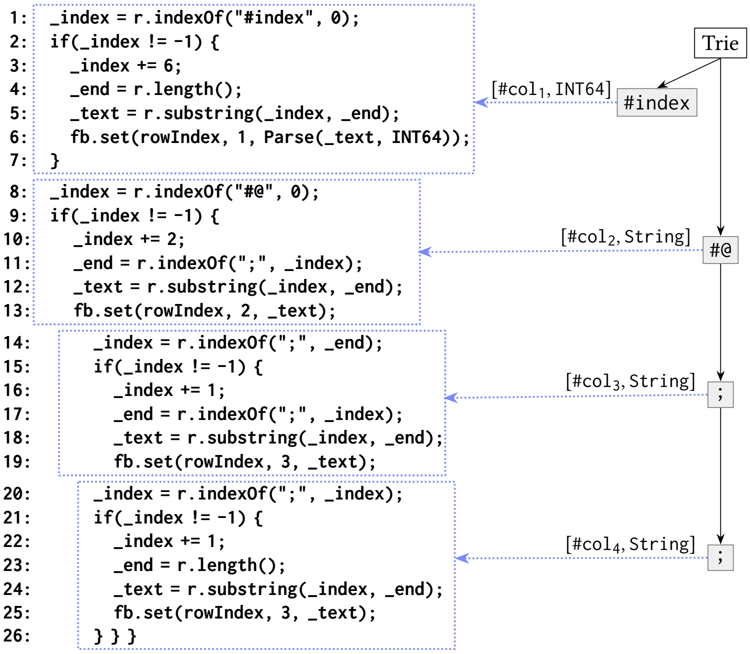
(b) Row Index Code Template

```
GenerateColCode( 𝒱, 𝒯)
srcInit = "   ";   src = "   ";
trie = InitTrie( Root );
for ( column v ∈ 𝒱)
  Node node = new Node( 𝒱_key, 𝒱_colIndex, 𝒱_valueType );
  trie.insert( node ); // node's key is a list of strings
if 𝒯> |trie.Root.GetChild()|
 if trie.GetHeight() == |trie.GetNodes()|
   src = GenerateCodeRegular( trie );
 else
   src = GenerateCodeTrie( trie.Root, src, " 0 " );
else
 regexes = EmptySet();   map = EmptyMap();
 for ( column v ∈ 𝒱)
   regex = BuildRegex( v_key );   regexes.Add( regex );
   map.Put( v_key, v_colIndex );
 srcInit = MapToString( map ); // convert map to string src
 src = GenerateCodeRegex( regexes );
return srcInit, src;

GenerateCodeTrie( Node node, String pos )
if ( node ∈ EndOfColPattern )
 src += " _end = FindEndPos(" + node_delimiter+ ");" +
        " _text = r.Substring(" + pos + " , _end"); " +
        " _val = Parse(_text, " + node.valType + " ); " +
        " fb.set(rowIndex, " + node.cellIndex + " ", _val); "
if ( |node.GetChild()| > 0 )
 for ( child in node.GetChild() )
   src += " index = r.IndexOf(" + child_key +" , "+ pos+ " ); " +
          " if ( index != 0 ) { " +
          "  _newPos = _index+len( "+child_key+" ); "
   GenerateCodeTrie(child, newPos);
return src + " } ";

GenerateCodeRegular( PrefixTree trie )
src += " Node[] nodes = "+ trie.GetAllNodes()+" ; "+
       " _index = 0;   _end = 0; +
       " for ( node ∈ nodes ) { " +
       "  _index = r.IndexOf( node_key, _index); +
       "  _end = FindEndPos(_index, node_delimiter);" +
       "  _text = r.Substring(_index, _end"); " +
       "  _val = Parse(_text, node.valType ); " +
return src;

GenerateCodeRegex( Regexes regexes )
for (Regex reg ∈ regexes)
  src += " matcher= Pattern.Compile( " + reg + " .Matcher(r)) " +
         " while(matcher.Find()) { " +
         "  _key = matcher.Group(1); " +
         "  colIndex = map.Get(_key); " +
         "  if ( colIndex != Null ) { " +
         "   _end = FindEndPos(map.GetSuffix(colIndex)); " +
         "   _text = r.Substring(matcher.End(), _end); " +
         "   _val = Parse(_text, map.GetValType(cellIndex)); " +
         "   fb.Set(rowIndex, cellIndex, _val)"); } }  "
return src;
```

(c) Cell Value Code Template

Fig. 6. Overview of GIO Code Templates and Code Generation.

Cell Value by Sequential String Matching: The main drawbacks of extracting values via nested conditions are the assumption of a fixed record structure (e.g., number of columns) and a very large code size for datasets with many columns and few shared paths. For example, sparse matrix representations (e.g., LibSVM) with millions of columns would require millions of generated conditions and runtime checks. For such cases, our second approach compiles a regular loop (over



```
_index = r.indexOf("#index", 0);
if(_index != -1) {
  _index += 6;
  _end = r.length();
  _text = r.substring(_index, _end);
  fb.set(rowIndex, 1, Parse(_text, INT64));
}
_index = r.indexOf("#@", 0);
if(_index != -1) {
  _index += 2;
  _end = r.indexOf(";", _index);
  _text = r.substring(_index, _end);
  fb.set(rowIndex, 2, _text);
    _index = r.indexOf(";", _end);
    if(_index != -1) {
      _index += 1;
      _end = r.indexOf(";", _index);
      _text = r.substring(_index, _end);
      fb.set(rowIndex, 3, _text);
    _index = r.indexOf(";", _index);
    if(_index != -1) {
      _index += 1;
      _end = r.length();
      _text = r.substring(_index, _end);
      fb.set(rowIndex, 3, _text);
    } } }
```

Fig. 7. Example Reader Code with Nested Conditions.

projected attributes) with generic string matching and cell value extraction. This approach applies to formats where column indexes are prefixes of cell values, and allows efficient projections without scanning all columns. We use this approach for simple patterns and many columns.

Cell Value by Regular Expressions: Finally, our third approach uses compiled regular expressions for matching more complex patterns. During code generation (see Figure 6(c), Line 6), we check the prefix tree structure against a threshold $\mathcal{T}$ (number of projected, non-overlap attributes; default 100) in order to select the appropriate code generation approach. As shown in Lines 41-51, we generate a regular expression for each column (with simple "s+" and "d+" for spaces and numbers) and add them to a set of regular expressions that are processed in a generic loop, similar to the regular string matching approach. In addition to the regular expressions, we are also keeping a map of column-index, value-type, and suffixes. This approach yields very small code size, and the auxiliary data structures are of moderate size as well.

5 EXPERIMENTS

We study our GIO framework on a variety of real-world datasets with different data characteristics as well as both existing and custom data formats. The primary insights are:

- *Identification and Generation:* GIO identifies and generates correct mapping rules and readers for various formats with small overhead that is linear in the sample size.
- *Reader Performance:* GIO yields readers with competitive runtime performance—similar to hand-coded multi-threaded readers when reading the entire datasets—and significant improvements if only few attributes are extracted.

Table 1. Datasets (n Rows, m Columns, o Nested Objects).

Dataset	n (nrow)	m (ncol)	o (objects)	Size [GB]
AMiner-Author (JSON)	1,712,432	Nested	1	0.62
AMiner-Paper (JSON)	2,092,355	Nested	2	3.7
Yelp (JSON)	8,635,403	Nested	7	19
AMiner-Author (Custom)	1,712,432	N/A	N/A	0.5
AMiner-Paper (Custom)	2,092,355	N/A	N/A	2.1
HL7 (Custom)	10,240,000	100	N/A	7.5
Yelp-Review (CSV)	8,635,403	9	Flat	6.5
Mnist8m (LibSVM)	8,100,000	784	Flat	12
Susy (LibSVM)	5,000,000	18	Flat	2.4
Higgs (CSV)	11,000,000	28	Flat	7.5
Queen (MM)	4,147,110	4,147,110	Flat	4.5
ReWaste F (CSV)	1,953,434	313	Flat	1.2
ADF (XML)	10,000,000	146	20	41

5.1 Experimental Setting

HW/SW Environment: We ran all experiments on a server node with an AMD EPYC 7302 CPU @ 3.0-3.3 GHz (16 physical/32 virtual cores) with 512KB, 8MB and 128MB L1, L2 and L3 caches, 128 GB DDR4 RAM (peak performance is 768 GFLOP/s, 183.2 GB/s), two 480 GB SATA SSDs (system/home), and twelve 2 TB SATA HDDs (data). All reader experiments utilize a single SSD. The software stack comprises Ubuntu 20.04.1, OpenJDK 11 with 120 GB max and initial JVM heap sizes for GIO, as well as Python 3.8 and clang++10 for other baseline readers.

Implementation Details. The entire GIO framework is implemented in Java and has been integrated into the open-source ML system Apache SystemDS. In detail, SystemDS compiles hybrid runtime plans of local, in-memory operations and distributed operations on Apache Spark. For a seamless integration with Spark and HDFS file system implementations (e.g., local, HDFS, S3, Azure, FTP), SystemDS is primarily implemented in Java expect for performance-critical kernels in C++ and CUDA, which are accessed through JNI. For compiling the generated readers, we use the fast in-memory Janino Java compiler [74] (as used for whole-stage code generation in Apache Spark as well as code generation for operator fusion in Apache SystemDS [32]). Byte-code compilation is negligible and the JVM just-in-time (JIT) compiler runs asynchronously a multi-tier compilation into native code to yield very good performance. Generating LLVM and vectorized SIMD code as well as distributed readers for Spark are interesting directions for future work.

Datasets. Table 1 shows the real-world datasets and characteristics used for both micro benchmarks of identification and reader generation as well as reader runtime experiments. These datasets comprise existing and custom formats as well as flat and nested representations.

- *AMiner:* The AMiner publications dataset (extraction and mining of academic social networks dataset) [73] contains information about papers, citations, authors, affiliations, and author collaborations. The experiments use the AMiner's original, text-based custom data format as well as another JSON format containing nested authors and publications.
- *Yelp:* The Yelp dataset is in JSON format containing multiple hierarchy levels. Additionally, we extracted the Yelp reviews into CSV format and saved them as two Yelp-Review files.
- *Flat Datasets:* We further use the real-world datasets Higgs (UCI) and ReWaste F in dense CSV format, as well as Mnist8m, Susy, and Queen (all UCI) in sparse LibSVM and MatrixMarket. These datasets represent a good mix of characteristics (e.g., dimensions and sparsity).

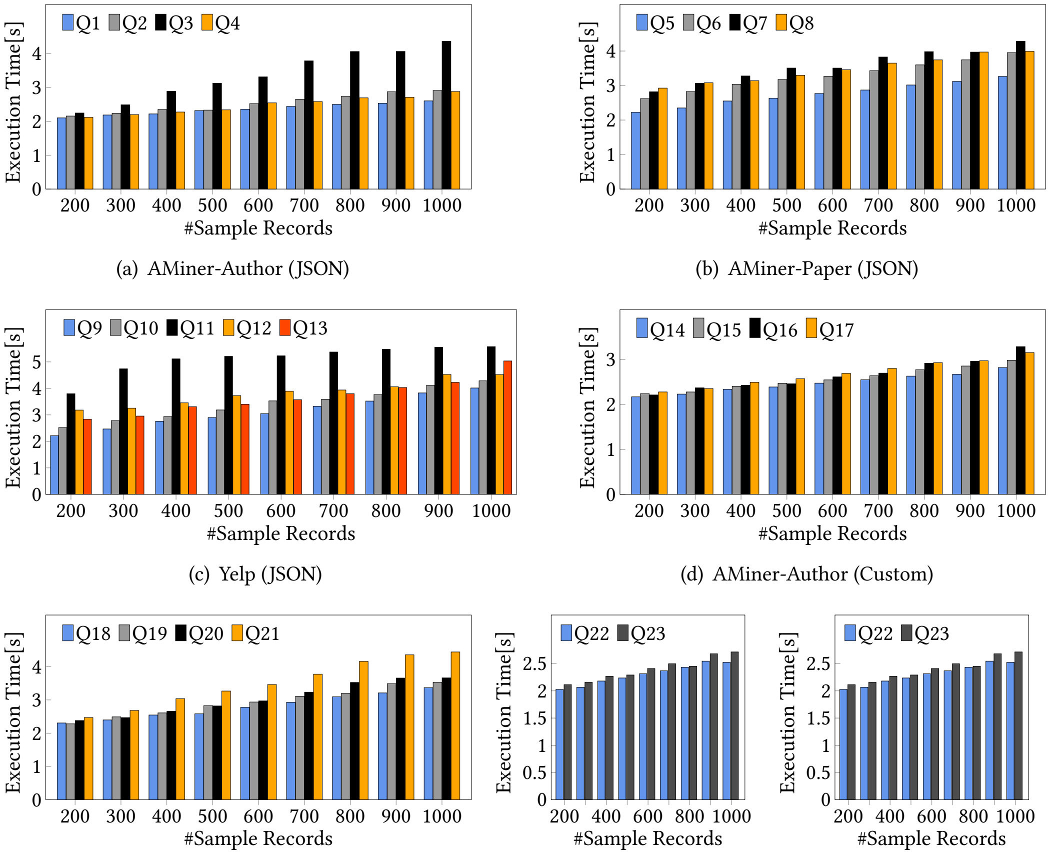
Table 2. Micro-Benchmark Use Cases with Various Data/Query Characteristics.

Q#	Dataset	Format	(Projection) Query	Nesting & Array
Q1	AMiner-Author	JSON	index	L1
Q2	AMiner-Author	JSON	name, paper_count	L1
Q3	AMiner-Author	JSON	index, name, paper_count, citation_number, hIndex	L1
Q4	AMiner-Author	JSON	name, affiliations[1, 2, 3, 4]	L1, L1 Array
Q5	AMiner-Paper	JSON	index	L1
Q6	AMiner-Paper	JSON	title, year	L1
Q7	AMiner-Paper	JSON	index, title, year, publication_venue, abstract	L1
Q8	AMiner-Paper	JSON	index, references[1, 2, 3, 4]	L1, L1 Array
Q9	Yelp	JSON	id	L1
Q10	Yelp	JSON	id, text	L1
Q11	Yelp	JSON	id, text, business.id, user.id, business.postal_code	L1, L2
Q12	Yelp	JSON	id, text, business.id, user.id, business.checkin.date, business.attribute.wifi	L1, L2, L3
Q13	Yelp	JSON	business.checkin.date, business.hours.monday, business.attribute.HhashTV	L3
Q14	AMiner-Author	Custom	index	N/A
Q15	AMiner-Author	Custom	name, paper_count	N/A
Q16	AMiner-Author	Custom	index, name, paper_count, citation_number, hIndex	N/A
Q17	AMiner-Author	Custom	name, affiliations[1, 2, 3, 4]	N/A
Q18	AMiner-Paper	Custom	index	N/A
Q19	AMiner-Paper	Custom	title, year	N/A
Q20	AMiner-Paper	Custom	index, title, year, publication_venue, abstract	N/A
Q21	AMiner-Paper	Custom	index, references[1, 2, 3, 4]	N/A
Q22	Yelp-Review	CSV	id	FLAT
Q23	Yelp-Review	CSV	id, text, stars	FLAT
Q24	HL7	Custom	evn_code, datetime, reason_code, operator_id	N/A
Q25	HL7	Custom	patient_name, birth_day, address, phone_number, account_number	N/A

- *HL7:* Health-Level 7 is a health-care communication protocol and message format, which is the de-facto standard for data exchange among different clinical/health information systems and medical devices. HL7 version 2.x is a custom text-based format, while version 3.x is an XML-based format. To create a large dataset, we manually generated 1024 of these messages and duplicated them 10,000 times.
- *ADF:* The Auto-Lead Data Format (ADF) [1] is an XML-based format, designed for communicating automotive dealership purchase requests including equipment variants and financing. Many vendors of customer management systems serving the automotive industry support ADF, which was developed by thirteen major organizations in the automotive ecosystem. We created 10 million instances with their relationships and stored them as ADF XML files.

Sample Raw and Frame/Matrix Inputs: As a prerequisite for evaluating GIO, we need to construct the input sample-raw and target frame/matrix representations. For existing data formats, we automatically construct these samples (of different sizes), whereas for custom formats—without existing readers—we constructed these samples semi-manually. In both cases, we ensure that each column contains at least two values to facilitate the identification of valid mappings.

Baseline Comparisons: We aim to compare our GIO framework with state-of-the-art libraries and systems. For the nested datasets, we compare GIO with four best-of-breed JSON parsers: (1) Jackson [5] as a well-known standard JSON library for Java as used in Apache Spark and Apache Drill, (2) JSON4J, a Java JSON library from IBM, (3) Gson [3], another Java JSON parser from Google, (4) HAPI-HL7 [4] object-oriented HL7 2.x parser for Java, and (5) RapidJSON [6] as a C++ JSON parser and generator. Additionally, we use Python libraries and Apache SystemDS for comparing reading dense and sparse matrices from data formats such as CSV, MatrixMarket, and LibSVM; as well as hand-coded readers for custom data formats not supported in existing systems.



(a) AMiner-Author (JSON)

(b) AMiner-Paper (JSON)

(c) Yelp (JSON)

(d) AMiner-Author (Custom)

(e) AMiner-Paper (Custom)

(f) Yelp (CSV)

(g) HL7 (Custom)

Fig. 8. Q1-Q25 Execution Time for Identifying Mapping Rules.

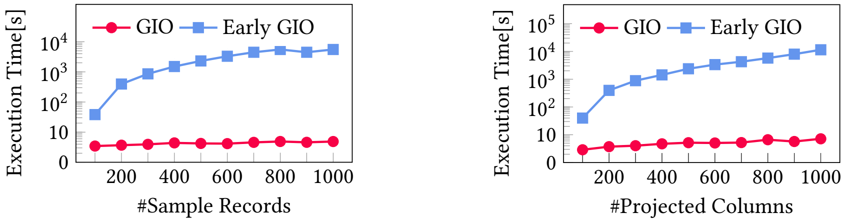
5.2 Identification and Reader Generation

In order to study the GIO framework components of identifying mapping rules and generating readers, we composed 25 diverse use cases, with different datasets and characteristics, in existing and custom formats, and with different projected attributes. These micro-benchmark use cases resemble a mix of different characteristics—with different nesting levels, projections, and array access—which are loosely inspired by real custom data formats we worked with in the ecosystems of automotive vehicles, recycling economy, process industry, and health-care as well as related use cases. Table 2 summarizes these uses cases and queries, as well as their key properties, where we utilize a subset of the datasets from Table 1.

Identification with Varying Sample Size: In a first series of experiments, we compare the runtime overhead of identifying mapping rules for various data characteristics (e.g., flat and nested, out-of-order attributes) and increasing sample size |S|. Conceptually, increasing sample sizes are challenging due to increasing number of values, where each such value needs to get mapped to an increasing size of the sample raw strings. Figure 8 shows the runtime for identifying mapping rules when varying the sample size from 200 to 1,000 rows. For single-line datasets, prefix trees and suffix patterns show excellent linear scaling. However, for multi-line datasets, GIO still has to find a row delimiter, which is very challenging when the number of records in the raw sample increases.

Table 3. Execution Time for Identifying Mapping Rules, from 1k to 10k Sample Records (times are sec.)

#Rows	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Q23	Q24	Q25
1K	2.5	2.9	4.1	2.7	3.3	3.8	4.2	3.9	4.1	4.3	4.9	4.6	4.7	2.8	3.1	3.4	3.2	3.5	3.7	4.1	4.5	2.7	2.7	7.2	8.8
2K	3.1	3.4	4.3	4	4.3	5.3	5.3	5.1	4.7	5.4	5.8	5.7	5.4	3.8	3.9	4.7	4.5	4.7	4.9	5.2	6.8	3.3	3.5	10.6	9.6
3K	3.6	4	4.4	4.4	5.1	5.7	6.2	5.6	5.3	5.5	6	6.1	6.5	5.2	5.7	6.4	5.7	6.4	7.1	7.5	9.4	3.8	4	18.4	17.1
4K	3.9	4.6	4.6	4.7	5.6	5.8	6.3	5.8	5.3	6.5	6.8	6.4	6.8	7.7	8.2	9.1	7.7	9.7	10	11.5	10.8	4.5	4.5	20.5	20
5K	4.2	4.9	5.1	4.9	6	6.2	6.4	6.7	5.6	7.3	7	6.6	6.7	11.1	12.4	14.3	12.3	16	16.3	19.3	19.2	4.6	4.8	21.3	22.5
6K	4.8	5.5	5.5	5.9	6.3	6.3	6.5	7.3	5.7	7.4	7.5	7.1	6.8	18.4	19.2	22.8	20.4	22.9	25.4	30.3	29.1	4.9	5.2	30.9	31
7K	5	5.6	5.8	6	6.4	6.4	6.8	7.4	6.1	7.5	7.7	7.2	7.3	26.7	28.2	31	28.4	33.7	35.3	38.9	38.6	5	5.4	38	39
8K	5.2	5.7	6.2	6.1	6.5	6.5	7.6	7.5	6.5	7.7	8.1	7.6	7.6	37.2	39.6	42.2	39.4	48.4	51.3	57.1	55.2	5.1	5.4	44.9	45
9K	5.3	6.3	6.3	6.3	6.6	6.6	8	7.6	6.9	7.8	8.3	7.8	7.8	50.8	54.7	54.8	53.5	64.9	67.9	74.2	71.7	5.2	6.2	55.1	56
10K	5.5	6.4	6.4	6.6	6.7	6.8	8.1	7.8	7.1	7.9	8.4	8.2	8.3	67.8	72.4	72.8	70.1	86.7	92.1	97.3	94.3	5.5	6.2	61.2	63



(a) #Columns = 200

(b) #Sample Records = 200

Fig. 9. Identification Overhead of Different GIO Versions.

In order to provide evidence for safer conclusions, Table 3 scales the number of input samples for all queries from 1,000 to 10,000 samples and thus, well beyond the typical scale of user-provided mappings. The absolute overhead of identification is moderate in the order of few seconds for most use cases and the scaling to 10,000 samples is well-behaved. The AMiner and HL7 use cases with more complex custom data formats are exceptions with overhead up to 10 seconds for 1K samples), but even that is still very reasonable for offline use in practice. Generally speaking, identifying rules for deep nesting levels and projected array fields (e.g., Q4, Q8, Q11, and Q12 projected arrays and fields from L1-L3 levels) is more expensive than identifying rules for flat datasets.

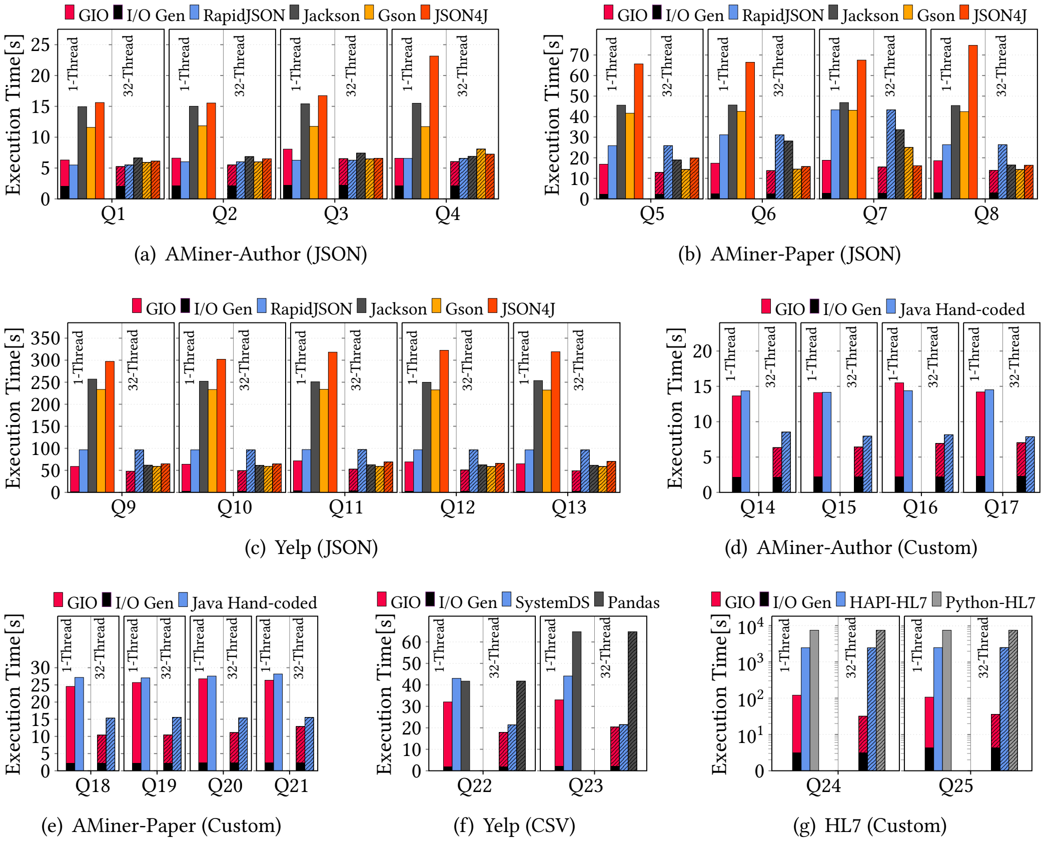
Prefix Matching: Experiments with early versions of GIO showed very high overhead and super-linear scaling. Accordingly, Figure 9 shows an ablation study of comparing GIO (using prefix/suffix matching) with an earlier version of GIO (using brute-force matching) on data in MatrixMarket format. In detail, Figure 9(a) fixed the number of columns at 200 and varied the number of sample records from 100 to 1,000, whereas Figure 9(b) fixed 200 sample records and varied the number of projected columns from 100 to 1,000. In both settings, we observe up to three orders of magnitude runtime improvements by prefix/suffix matching.

Reader Code Generation: Furthermore, Figure 10 shows the runtime of reading all 25 datasets, while also indicating the fraction of time (I/O-Gen in black) spent for both identification (with |S| = 200) and reader generation. We observe that together, identification and reader generation account only for a small constant fraction of the end-to-end runtime (independent of dataset size). The actual reader generation (source code generation and compilation) is almost negligible in the tens of milliseconds because we avoid generating a large number of nested conditions.

Correctness: These use cases also serve as a verification of the correctness of identification and reader generation. Compared with other baselines, GIO yields equivalent results for all use cases.

5.3 Reader Runtime Performance

We now can turn our attention to the actual reader performance in comparison with various baselines, as well as the impact of multi-threading and the number of projected attributes.



(a) AMiner-Author (JSON)

(b) AMiner-Paper (JSON)

(c) Yelp (JSON)

(d) AMiner-Author (Custom)

(e) AMiner-Paper (Custom)

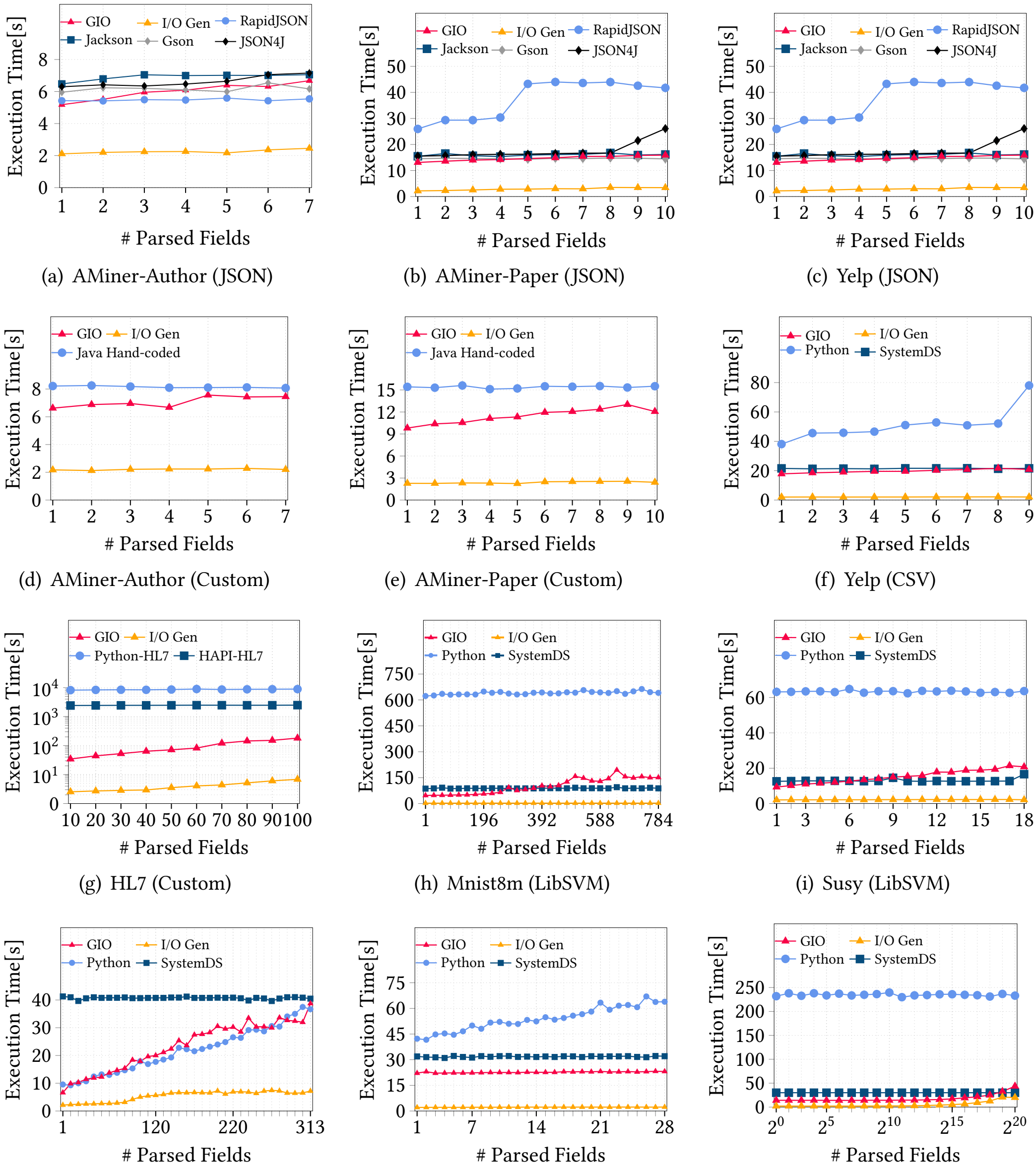
(f) Yelp (CSV)

(g) HL7 (Custom)

Fig. 10. Q1-Q25 Reader Runtime Comparison, including Identification and Reader Generation (|**S**| = 200).

Single-threaded Comparison: Figure 10 (left half of every use case) shows the single-threaded reader performance of GIO (including generation) as well as the four baselines RapidJSON, Jackson, Gson, and JSON4J. Here, I/O Gen further shows the time GIO spends on mapping identification and reader generation, which is important because generated readers can be used many times, potentially amortizing these identification and reader generation costs. The use cases from Table 2 are kept unchanged which includes a mix of attribute projections. GIO efficiently skips non-projected attributes (no value parsing), whereas several baseline systems do not support this projection push-down. Overall, GIO shows competitive performance, close to or better than RapidJSON as the fastest baseline. The other baselines (Jackson, Gson, and JSON4J) show substantial overheads between 2.5x and 7x. For the Yelp CSV dataset, we also compare with SystemDS and Pandas, where GIO shows again moderate runtime improvements despite reader generation. Finally, for the AMiner and HL7 custom formats, we compare with a hand-crafted baseline, which shows up to 2x runtime overhead due to less efficient pattern matching.

Multi-threaded Comparison: Figure 10 (right half of every use case) also shows the multi-threaded reader performance of GIO and the different baselines. We make several interesting observations. First, RapidJSON and Pandas do not exploit multi-threading and thus, become less competitive in such configurations. Second, there are several use cases where GIO multi-threading improves performance by more than an order of magnitude (e.g., Q9, Q10, Q12, Q13, Q24, Q25). However, there are also use cases (Q1-Q4) where the improvements are very small. Third, with

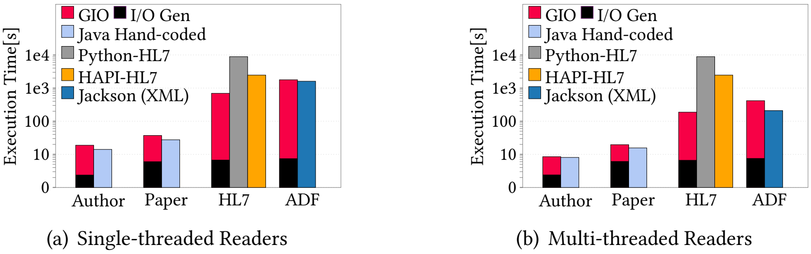


(a) AMiner-Author (JSON)
(b) AMiner-Paper (JSON)
(c) Yelp (JSON)
(d) AMiner-Author (Custom)
(e) AMiner-Paper (Custom)
(f) Yelp (CSV)
(g) HL7 (Custom)
(h) Mnist8m (LibSVM)
(i) Susy (LibSVM)
(j) ReWaste F (CSV)
(k) Higgs (CSV)
(l) Queen (MM)

Fig. 11. Reader Runtime Comparison with Varying Number of Attributes.

multi-threaded reading, the identification and reader generation overhead becomes substantial on some use cases (e.g., 50% on Q18-Q21). The reason is that GIO identification and reader generation are not fully parallelized yet, which is an interesting direction for future work.

Varying Number of Projected Attributes: The use cases covered a mix of workloads with projections, which have large impact on performance. We now study this impact by varying the number of projected attributes (from one to a fixed number of attributes, or all in case of few columns) for all datasets from Table 1. Figure 11 shows the results with multi-threaded readers.



(a) Single-threaded Readers (b) Multi-threaded Readers

Fig. 12. Reader Performance on Full Custom Datasets.

For JSON datasets, we see that RapidJSON is the slowest because of single-threaded parsing, where the peculiar transition from 4 to 5 attributes is likely due to the use of SIMD instructions. Furthermore, for flat matrix and frame formats (CSV, LibSVM, MatrixMarket), SystemDS shows fairly good performance but does not exploit projection push-down. For that reason, GIO yields performance improvements for small and moderate number of projected attributes but SystemDS often outperforms GIO slightly when reading the entire dataset. Python readers show mixed results: when reading from dense CSV, performance is good due to projection push-down, but when reading from sparse formats (LibSVM and MatrixMarket) performance is non-competitive. Finally, GIO shows very robust performance with mostly linear scaling—except Yelp (CSV)—even when reading sparse matrices with millions of columns (e.g., the Queen dataset).

5.4 Full Data of Custom Data Formats

Until this point, many of our experiments with complex custom data formats extracted a moderate number of attributes. As additional end-to-end experiments (for truly custom data formats unsupported by ML systems), we use the AMiner-Author, AMiner-Paper, HL7, and ADF datasets in their custom text representations, and generate readers for all contained attributes as a stress test.

Runtime Performance: Figure 12 shows the runtime of reading the full datasets, including mapping rules identification and reader generation, for both single-threaded and multi-threaded readers. We make again multiple interesting observations. First, multi-threading yields again rather small speedups on the AMiner and ADF datasets but a good speedup of 5x on HL7. Second, the identification overhead remains moderate for AMiner-Author, HL7, and ADF. In contrast, on AMiner-Paper, GIO shows substantial identification overhead, both in single- and multi-threaded readers because of many attributes, longer strings, and deeper structure. For this reason, the hand-crafted baseline also outperforms GIO on this specific AMiner-Paper dataset. Third, GIO yields—even for these stress tests—good runtime performance, close to the hand-coded and Jackson baselines and is almost two orders of magnitude faster than the single-threaded Python HL7 baseline (and one order of magnitude faster than the Java-based HAPI-HL7 baseline).

Summary: Overall, the experiments have shown that GIO correctly identifies mapping rules and generates code for efficient readers. These readers scale linearly with the number of attributes and datasize, and are very competitive to available baseline systems. The identification and reader generation overhead is usually very small, but there are edge cases where this overhead can be substantial. However, identification and reader generation are typically conducted offline (just once) and amortized when reading multiple datasets of the same custom format or very large datasets. Finally, GIO has shown robust performance, which makes it a practical tool for custom formats that otherwise require hand-crafted extractors.

6 RELATED WORK

Identifying mapping rules and generating efficient readers for custom data formats in our GIO framework has connections to the broad areas of schema inference, schema matching and mapping, query processing on raw data, as well as efficient readers for text-based and open formats. In the following, we survey these areas individually and discuss how GIO differs in detail.

Schema Inference: Discovering metadata and schema information is a broad area and comprises a variety of techniques. Basic techniques include inferring attribute types via regular expressions (e.g., via `option("inferSchema",true)` in Spark [78]) as well as data profiling [8, 37] to identify domain characteristics and find key candidates by discovering uniqueness constraints, inclusion dependencies, and functional dependencies. More recent techniques also discover semantic types [46, 80] (e.g., date, currency, location) and feature types [70] (e.g., numeric, ordinal, categorical) with classifiers trained on large corpora of schemas. Furthermore, schema inference for semi-structured JSON data has received considerable attention in the literature [22], and is supported by several systems such as SparkSQL [15], Jaql [28], and Schema Guru [7]. Besides JSON schema, many systems introduce tailor-made, simpler schema/type languages [23, 28], some of which also handle different levels of abstraction [20, 21]. A common approach of schema inference/extraction is counting the occurrences of attributes at certain paths [20, 53]. Schema inference for semi-structured data is an old problem though, where early work focused on the Object Exchange Model (OEM) and related schema discovery algorithms [64, 75]. Recent work on JSON Schema introduced a witness generation algorithm [16, 17] for addressing the problems of schema satisfiability, inclusion, and equivalence. GIO is related to some of these works by also relying on sampling and guiding example instances, but GIO differs in its goal of identifying mapping rules from raw, text-based data formats to a structured frame or matrix representation, which does not need full schema inference.

Schema Matching and Mapping: Schema matching [26, 60] aims to find correspondences (mapping rules) between two or more schemas, whereas schema mapping [61] generates data transformation programs for created correspondences. These fields are naturally related to GIO's mapping rules identification and reader generation. Schema matching techniques are broadly classified into schema-based, instance-based, and hybrid [68]. COMA++ [18, 38]—and its evolution to COMA 3.0—is an example of a feature-rich schema matching tool. In contrast, schema mapping tools like Clio [44, 45] generate—given the high-level correspondences—correct and efficient SQL or XSLT transformation programs. In database theory, schema mappings were formalized via declarative mappings called tuple-generating dependencies (tgds) and extensively studied [40, 54]. Multiple lines of work also leverage examples. Sample-driven schema mappings [67] requires only user-provided target example records, which reduce the effort for specifying mappings. Other work like Clio [77], IREINE [11, 12, 14], and Muse [13] leverage examples to aid the design, understanding and refinement of mappings. Examples are also used to learn extractors from hierarchical JSON and XML data to relational tables [76]. In contrast to schema matching and mapping, GIO identifies mapping rules for raw data in custom data formats and generates efficient readers.

Query Processing on Raw Data: Our GIO framework was partially inspired by the highly-influential NoDB work [9, 10, 47] that enables SQL query processing on raw CSV and JSON files. Efficiency for exploratory data analysis is achieved by avoiding data loading, selective tokenization and parsing (via positional maps), as well as horizontal/vertical partitioning and caching [9]. RAW [52] and Proteus [51] later introduced code generation for data extraction and query processing on heterogeneous formats and JSON data. Other work then handled in-situ raw data access and query processing for scientific formats like HDF5 and NetCDF [30, 48], JSONiq query processing [63, 66], and integrated these techniques into systems with continuous scans and speculative loading [35]. Typically, these systems assume known syntactic and semantic properties of the raw data formats, which does not apply to custom data formats. In contrast, our GIO framework does not require any metadata on schemas, delimiters, or other structure but identifies such properties from the examples, which makes code generation more important and challenging.

Efficient Readers: Related to query processing on raw data, recent work—in the context of accessing open data formats [79] and avoiding data loading in exploratory analysis [9]—focus on the implementation and generation of efficient readers for various formats. Major lines of work are selective and speculative parsing (e.g., via raw string filtering, or parsing of projected columns) [35, 41, 58, 65], the exploitation of SIMD instruction-level parallelism [42, 49, 55, 56] and HW accelerators [71], dedicated cost estimation techniques [19, 39, 65], as well as the generation of specialized code [43, 51, 55]. These lines of work share the concepts of eliminating unnecessary overheads and leveraging modern hardware. Similar, to query processing on raw data, these efficient readers typically assume known format properties. Although our GIO framework already shows competitive performance (similar to existing libraries and systems for known formats), generating high-performance readers for custom data formats is interesting future work.

7 CONCLUSIONS

To summarize, we introduced the GIO (generated I/O) reader framework for custom text data formats. Given a sample of raw data and its mapping to matrices or frames as user-provided examples, GIO automatically identifies position/value mapping rules, and efficiently generates code for efficient, multi-threaded readers for datasets in this format. Our experiments show that GIO is capable of correctly identifying—even on samples of very moderate size—the mapping rules for basic text formats, custom flat text formats, and nested data formats. At the same time, the generated readers yield competitive performance. In conclusion, GIO simplifies exploratory data analysis and predictive modeling with custom data formats by reducing manual effort and potential data quality issues. Users can additionally perform manual fine-tuning of mapping rules and generated readers. Interesting future work includes the generation of data-parallel readers for distributed computation, the generation of more efficient readers, the handling of custom binary data formats, a richer set of shape inference and position/value mapping functions, as well as the integration with query processing on raw data [9, 47, 51] and federated learning on raw data [24, 50].

ACKNOWLEDGMENTS

Partial funding for this work was provided by the COMET project Recycling and Recovery of Waste for Future (acronym ReWaste F, contract 882512) in the COMET—Competence Centers for Excellent Technologies—program, which is financially supported by BMK, BMDW and the federal state of Styria, and managed by the FFG. We also thank in particular the ReWaste F partners Siemens and REDWAVE, as well as Infineon for inspiration and use cases related to custom data formats.

REFERENCES

[1] 2000. Auto-lead Data Format / ADF: An Industry Standard Data Format for the Export and Import of Automotive Customer Leads using XML. https://adfxml.info/adf_spec.pdf
[2] 2013. *Matrix Market Exchange Formats.* Technical Report. Math, Statistics, and Computational Science. https://math.nist.gov/MatrixMarket/formats.html
[3] 2022. Gson. https://github.com/google/gson/
[4] 2022. HAPI object-oriented HL7 2.x parser for Java. https://hapifhir.github.io/hapi-hl7v2/
[5] 2022. Jackson. https://github.com/FasterXML/jackson/
[6] 2022. RapidJSON. http://rapidjson.org/
[7] 2022. Schema Guru. https://github.com/snowplow/schema-guru
[8] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data Profiling: A Tutorial. In *SIGMOD*. 1747–1751. https://doi.org/10.1145/3035918.3054772

[9] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*. 241–252. https://doi.org/10.1145/2213836.2213864

[10] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB in Action: Adaptive Query Processing on Raw Data. *PVLDB* 5, 12 (2012), 1942–1945. https://doi.org/10.14778/2367502.2367543

[11] Bogdan Alexe, Balder TEN Cate, Phokion G Kolaitis, and Wang-Chiew Tan. 2011. Characterizing schema mappings via data examples. *TODS* 36, 4 (2011), 1–48. https://doi.org/10.1145/2043652.2043656

[12] Bogdan Alexe, Balder ten Cate, Phokion G Kolaitis, and Wang-Chiew Tan. 2011. EIRENE: Interactive design and refinement of schema mappings via data examples. *PVLDB* 4, 12 (2011), 1414–1417. http://www.vldb.org/pvldb/vol4/p1414-alexe.pdf

[13] Bogdan Alexe, Laura Chiticariu, Renée J Miller, and Wang-Chiew Tan. 2008. Muse: Mapping understanding and design by example. In *ICDE*. 10–19. https://doi.org/10.1109/ICDE.2008.4497409

[14] Bogdan Alexe, Balder Ten Cate, Phokion G Kolaitis, and Wang-Chiew Tan. 2011. Designing and refining schema mappings via data examples. In *SIGMOD*. 133–144. https://doi.org/10.1145/1989323.1989338

[15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394. https://doi.org/10.1145/2723372.2742797

[16] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, and Stefanie Scherzinger. 2021. A Tool for JSON Schema Witness Generation. In *EDBT*. 694–697. https://doi.org/10.5441/002/edbt.2021.86

[17] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness Generation for JSON Schema. *PVLDB* 15, 13 (2022), 4002–4014. https://www.vldb.org/pvldb/vol15/p4002-sartiani.pdf

[18] David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. 2005. Schema and ontology matching with COMA++. In *SIGMOD*. 906–908. https://doi.org/10.1145/1066157.1066283

[19] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data. *PVLDB* 11, 3 (2017), 324–337. https://doi.org/10.14778/3157794.3157801

[20] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting types for massive JSON datasets. In *DBPL@VLDB Workshop*. 1–12. https://doi.org/10.1145/3122831.3122837

[21] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *VLDB J.* 28, 4 (2019), 497–521. https://doi.org/10.1007/s00778-018-0532-7

[22] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. In *SIGMOD*. 2060–2063. https://doi.org/10.1145/3299869.3314032

[23] Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema inference for massive JSON datasets. In *EDBT*. https://doi.org/10.5441/002/edbt.2017.21

[24] Sebastian Baunsgaard, Matthias Boehm, Ankit Chaudhary, Behrouz Derakhshan, Stefan Geißelsöder, Philipp M. Grulich, Michael Hildebrand, Kevin Innerebner, Volker Markl, Claus Neubauer, Sarah Osterburg, Olga Ovcharenko, Sergey Redyuk, Tobias Rieger, Alireza Rezaei Mahdiraji, Sebastian Benjamin Wrede, and Steffen Zeuch. 2021. ExDRa: Exploratory Data Science on Federated Raw Data. In *SIGMOD*. 2450–2463. https://doi.org/10.1145/3448016.3457549

[25] Lasse Bergroth, Harri Hakonen, and Timo Raita. 2000. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. IEEE, 39–48. https://doi.org/10.1109/SPIRE.2000.878178

[26] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *PVLDB* 4, 11 (2011), 695–701. http://www.vldb.org/pvldb/vol4/p695-bernstein_madhavan_rahm.pdf

[27] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. 2013. On the relative trust between inconsistent data and inaccurate constraints. In *ICDE*. 541–552. https://doi.org/10.1109/ICDE.2013.6544854

[28] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. 2011. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB* 4, 12 (2011), 1272–1283. http://www.vldb.org/pvldb/vol4/p1272-beyer.pdf

[29] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*. 521–534. https://doi.org/10.1145/3183713.3196896

[30] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. 2014. Parallel data analysis directly on scientific file formats. In *SIGMOD*. 385–396. https://doi.org/10.1145/2588555.2612185

[31] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[32] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018), 1755–1768. https://doi.org/10.14778/3229863.3229865

[33] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*. 227–246. https://dl.gi.de/20.500.12116/19581

[34] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* 2, 3 (2011), 27:1–27:27. https://doi.org/10.1145/1961189.1961199

[35] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *SIGMOD*. 1287–1298. https://doi.org/10.1145/2588555.2593673

[36] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD Skills: New Analysis Practices for Big Data. *PVLDB* 2, 2 (2009), 1481–1492. https://doi.org/10.14778/1687553.1687576

[37] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *CIDR*. http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf

[38] Hong Hai Do and Erhard Rahm. 2002. COMA - A System for Flexible Combination of Schema Matching Approaches. In *VLDB*. 610–621. https://doi.org/10.1016/B978-155860869-6/50060-3

[39] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *SIGMOD*. 445–458. https://doi.org/10.1145/3448016.3452809

[40] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336, 1 (2005), 89–124. https://doi.org/10.1016/j.tcs.2004.10.033

[41] 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. https://doi.org/10.1145/3299869.3319898

[42] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*. 883–899. https://doi.org/10.1145/3299869.3319898

[43] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient stream processing through adaptive query compilation. In *SIGMOD*. 2487–2503. https://doi.org/10.1145/3318464.3389739

[44] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. 2005. Clio grows up: from research prototype to industrial tool. In *SIGMOD*. 805–810. https://doi.org/10.1145/1066157.1066252

[45] Mauricio A. Hernández, Renée J. Miller, and Laura M. Haas. 2001. Clio: A Semi-Automatic Tool For Schema Mapping. In *SIGMOD*. 607. https://doi.org/10.1145/375663.375767

[46] Madelon Hulsebos, Kevin Zeng Hu, Michiel A. Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César A. Hidalgo. 2019. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *SIGKDD*. 1500–1508. https://doi.org/10.1145/3292500.3330993

[47] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. 2011. Here are my Data Files. Here are my Queries. Where are my Results?. In *CIDR*. 57–68. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper7.pdf

[48] Milena Ivanova, Yagiz Kargin, Martin L. Kersten, Stefan Manegold, Ying Zhang, Mihai Datcu, and Daniela Espinoza-Molina. 2013. Data vaults: a database welcome to scientific file repositories. In *SSDBM*. 48:1–48:4. https://doi.org/10.1145/2484838.2484876

[49] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. 2020. Scalable Structural Index Construction for JSON Analytics. *PVLDB* 14, 4 (2020). https://doi.org/10.14778/3436905.3436926

[50] Peter Kairouz, Brendan McMahan, and Virginia Smith. 2020. Federated Learning Tutorial. In *NeurIPS*. https://slideslive.com/38935813/federated-learning-tutorial

[51] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB* 9, 12 (2016), 972–983. https://doi.org/10.14778/2994509.2994516

[52] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *PVLDB* 7, 12 (2014), 1119–1130. https://doi.org/10.14778/2732977.2732986

[53] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *BTW*. 425–444. https://dl.gi.de/20.500.12116/2420

[54] Phokion G Kolaitis. 2005. Schema mappings, data exchange, and metadata management. In *PODS*. 61–75. https://doi.org/10.1145/1065167.1065176

[55] Marcel Kornacker et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

[56] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *VLDB J.* 28, 6 (2019), 941–960. https://doi.org/10.1007/s00778-019-00578-5

[57] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[58] 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. https://doi.org/10.14778/3115404.3115416

[59] Ericsson M. Garcia-Martin, G. Camarillo. 2008. *Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists.* RFC 5364. RFC Editor. https://datatracker.ietf.org/doc/html/rfc5364

[60] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. Generic Schema Matching with Cupid. In *VLDB*. 49–58. http://www.vldb.org/conf/2001/P049.pdf

[61] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. 2000. Schema Mapping as Query Discovery. In *VLDB*. 77–88. http://www.vldb.org/conf/2000/P077.pdf

[62] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534. https://doi.org/10.1145/321479.321481

[63] Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. 2020. Rumble: Data Independence for Large Messy Data Sets. *PVLDB* 14, 4 (2020), 498–506. https://doi.org/10.14778/3436905.3436910

[64] Svetlozar Nestorov, Jeffrey Ullman, Janet Wiener, and Sudarashan Chawathe. 1997. Representative objects: Concise representations of semistructured, hierarchical data. In *ICDE*. 79–90.

[65] 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). https://doi.org/10.14778/3236187.3236207

[66] Christina Pavlopoulou, E Preston Carman Jr, Till Westmann, Michael J Carey, and Vassilis J Tsotras. 2018. A Parallel and Scalable Processor for JSON Data. In *EDBT*. 576–587. https://doi.org/10.5441/002/edbt.2018.68

[67] Li Qian, Michael J Cafarella, and HV Jagadish. 2012. Sample-driven schema mapping. In *SIGMOD*. 73–84. https://doi.org/10.1145/2213836.2213846

[68] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB J.* 10, 4 (2001), 334–350. https://doi.org/10.1007/s007780100057

[69] Y. Shafranovich. 2005. *Common Format and MIME Type for Comma-Separated Values (CSV) Files.* RFC 4180. RFC Editor. https://www.rfc-editor.org/rfc/rfc4180

[70] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. Towards Benchmarking Feature Type Inference for AutoML Platforms. In *SIGMOD*. 1584–1596. https://doi.org/10.1145/3448016.3457274

[71] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *PVLDB* 13, 5 (2020). https://doi.org/10.14778/3377369.3377372

[72] Ed. T. Bray. 2017. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259. RFC Editor. https://datatracker.ietf.org/doc/html/rfc8259

[73] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. Arnetminer: extraction and mining of academic social networks. In *SIGKDD*. 990–998. https://doi.org/10.1145/1401890.1402008

[74] Arno Unkrieg. 2014. Janino: A super-small, super-fast Java Compiler. https://janino-compiler.github.io/janino/2014-02-18_SWM-JAK.pdf

[75] Qiu Yue Wang, Jeffrey Xu Yu, and Kam-Fai Wong. 2000. Approximate graph schema extraction for semi-structured data. In *EDBT*. 302–316. https://doi.org/10.1007/3-540-46439-5_21

[76] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *PVLDB* 11, 5 (2018), 580–593. https://doi.org/10.1145/3187009.3177735

[77] Ling-Ling Yan, Renée J. Miller, Laura M. Haas, and Ronald Fagin. 2001. Data-Driven Understanding and Refinement of Schema Mappings. In *SIGMOD*. 485–496. https://doi.org/10.1145/375663.375729

[78] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[79] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[80] Dan Zhang, Yoshihiko Suhara, Jinfeng Li, Madelon Hulsebos, Çagatay Demiralp, and Wang-Chiew Tan. 2020. Sato: Contextual Semantic Type Detection in Tables. *PVLDB* 13, 11 (2020), 1835–1848. http://www.vldb.org/pvldb/vol13/p1835-zhang.pdf

Received October 2022; revised January 2023; accepted February 2023