JSONL Algebra: A Relational Algebra Framework for Semi-Structured Data with Interactive Workspace

Alex Towell
PhD Student, Computer Science
Southern Illinois University Edwardsville
lex@metafunctor.com

December 3, 2025

Abstract

We present JSONL Algebra (ja), a command-line tool and interactive REPL that applies relational algebra operations to semi-structured JSONL (JSON Lines) data. Unlike traditional database systems that require rigid schemas, ja embraces the flexibility of JSON while providing the expressive power of relational operations including selection, projection, joins, and aggregations. We introduce a novel interactive workspace model that enables exploratory data analysis through named datasets with immediate execution, eliminating the cognitive overhead of pipeline construction while maintaining streaming efficiency. Our implementation demonstrates how classical database theory can be adapted to modern semi-structured data formats commonly found in web APIs, log files, and data pipelines. The tool has been successfully deployed in production environments for data transformation tasks ranging from simple filtering to complex multi-way joins and aggregations.

1 Introduction

The proliferation of semi-structured data formats, particularly JSON (JavaScript Object Notation), has created a gap between classical database tools and modern data processing needs. While relational databases excel at structured data with fixed schemas, and document databases handle hierarchical data, there exists a need for lightweight, command-line tools that can apply relational operations to JSON data streams.

JSONL (JSON Lines) [3] has emerged as a popular format for streaming and processing JSON data, where each line contains a complete, valid JSON object. This format is ubiquitous in:

- Web service logs and analytics
- Machine learning datasets
- API response streams
- Data pipeline intermediate formats
- Database export formats

However, existing tools for processing JSONL data fall into two categories: (1) low-level text processing tools (jq, sed, awk) that require complex command composition, or (2) full database systems that necessitate data loading and schema definition.

We present ja (JSONL Algebra), which bridges this gap by:

1. Providing relational algebra operations for JSONL data
2. Supporting nested/hierarchical data structures with dot notation
3. Maintaining streaming efficiency for large datasets

4. Offering an interactive REPL with named dataset management

5. Requiring no schema definition or data loading step

2 Background and Related Work

2.1 Relational Algebra

Relational algebra, formalized by Codd [1], provides a theoretical foundation for database operations. The fundamental operations include:

- **Selection** (σ): Filter rows based on predicates
- **Projection** (π): Select specific columns
- **Cartesian Product** ($\times$): Combine all rows from two relations
- **Union** ($\cup$): Combine rows from relations
- **Difference** ($-$): Remove rows present in another relation
- **Rename** (ρ): Rename attributes

Derived operations include joins (natural join, equijoin, theta-join) and set operations (intersection). These operations form a complete algebra for manipulating relations.

2.2 Semi-Structured Data

Semi-structured data [2] lacks the rigid schema of traditional databases but contains implicit structure through nesting, arrays, and key-value pairs. JSON has become the de facto standard for semi-structured data due to its:

- Human readability
- Native support in programming languages
- Flexibility for evolving schemas
- Hierarchical structure support

2.3 Existing Tools

`jq` [4] is a widely-used command-line JSON processor that provides a powerful domain-specific language for querying and transforming JSON. However, it requires learning a new syntax and does not directly support relational operations like joins.

JSONPath provides XPath-like querying for JSON but lacks the algebraic foundation and compositional properties of relational algebra.

SQL-on-JSON systems (e.g., PostgreSQL's JSONB, SQLite's JSON functions) require loading data into a database and impose SQL syntax, which may be heavyweight for simple transformations.

`mlr` (Miller) [6] operates on CSV and similar formats with SQL-like verbs but has limited JSON support.

Our work differs by providing native relational algebra operations on JSONL while maintaining a streaming model and offering an interactive workspace for exploratory analysis.

3 System Design

3.1 Data Model

We define a JSONL dataset as a sequence of JSON objects:

$$D = \{o_1, o_2, \ldots, o_n\} \text{ where } o_i \in \text{JSON} \quad (1)$$

Each object o_i may have arbitrary nesting and structure. To bridge this with relational algebra, we introduce *dot notation paths*:

$$\text{path} := \text{key} \mid \text{path} \,.\, \text{key} \quad (2)$$

For example, `user.address.city` navigates the nested structure:

```
{
  "user": {
    "address": {
      "city": "New York"
    }
  }
}
```

This allows applying relational operations to nested fields without flattening.

3.2 Relational Operations

3.2.1 Selection (σ)

Selection filters objects based on JMESPath [5] expressions:

$$\sigma_\phi(D) = \{o \in D : \phi(o) = \text{true}\} \tag{3}$$

Example: `select 'age > 30'`

3.2.2 Projection (π)

Projection extracts specific fields, supporting nested paths:

$$\pi_{f_1,\ldots,f_k}(D) = \{\{f_1 : o(f_1), \ldots, f_k : o(f_k)\} : o \in D\} \tag{4}$$

Example: `project name,user.email`

3.2.3 Join ($\bowtie$)

We implement equijoin on specified fields:

$$D_1 \bowtie_{f_1=f_2} D_2 = \{o_1 \cup o_2 : o_1 \in D_1, o_2 \in D_2, o_1(f_1) = o_2(f_2)\} \tag{5}$$

where $\cup$ denotes object merging (with right-side precedence for conflicts).

Example: `join orders --on user_id=id`

3.2.4 Aggregation

We extend classical relational algebra with grouping and aggregation:

$$\gamma_{g;\text{AGG}_1,\ldots,\text{AGG}_k}(D) \tag{6}$$

where g is the grouping key and AGG_i are aggregation functions (count, sum, avg, min, max, list).

Example: `groupby region --agg count,sum(amount)`

3.3 Interactive REPL Architecture

Traditional command-line data processing follows a pipeline model where users must construct complete transformation sequences before execution. This cognitive overhead hinders exploratory analysis. We introduce an interactive workspace model with three key components:

3.3.1 Named Dataset Registry

The REPL maintains a mapping $R : \text{Name} \to \text{Path}$ where:

$$R = \{(n_1, p_1), (n_2, p_2), \ldots, (n_k, p_k)\} \tag{7}$$

Each name n_i refers to a dataset stored at path p_i. Original files retain their physical locations; derived datasets are stored in temporary files.

3.3.2 Current Dataset Context

The REPL tracks a current dataset $D_c \in \text{dom}(R)$, which operations use by default. This eliminates repetitive file path specification while maintaining clarity through the `pwd` command.

3.3.3 Immediate Execution Model

Unlike pipeline-based REPLs that accumulate operations, each command executes immediately:

```
ja> load users.jsonl      # Executes: register dataset
ja> select 'age > 30' adults  # Executes: create temp file
Created: adults (current)
```

This provides instant feedback while preserving all intermediate results for inspection.

3.4 Safety and Non-Destruction

All operations create new datasets rather than modifying existing ones:

$$\forall f \in \text{Operations}, \forall D : f(D) \to D' \text{ where } D \neq D' \tag{8}$$

Additionally, name uniqueness is enforced:

$$\forall n \in \mathrm{dom}(R) : \mathrm{operation}(D, n) \text{ fails if } n \in \mathrm{dom}(R) \tag{9}$$

This prevents accidental data loss and enables experiment tracking.

4 Implementation

4.1 Architecture

`ja` is implemented in Python with a modular architecture:

- **Core module**: Implements relational operations as generators for streaming
- **CLI module**: Provides command-line interface using `argparse`
- **REPL module**: Interactive session management
- **Schema module**: JSON Schema inference and validation
- **Import/Export module**: Format conversion (CSV, JSON arrays)

4.2 Streaming Model

Operations are implemented as Python generators to maintain constant memory usage:

```
def select(data, expression):
    compiled = jmespath.compile(
        expression)
    for row in data:
        if compiled.search(row):
            yield row
```

This allows processing datasets larger than available memory.

4.3 REPL Implementation

The REPL session maintains state through a `ReplSession` class:

```
class ReplSession:
    datasets: Dict[str, str]  # name
        -> path
    current_dataset: Optional[str]
    settings: Dict[str, Any]
    temp_dir: str
```

Operations execute via subprocess calls to the `ja` CLI, capturing output to temporary files. This design:

- Ensures consistency between CLI and REPL behavior
- Simplifies implementation by reusing CLI code
- Maintains streaming properties

4.4 Expression Evaluation

We leverage JMESPath for expression evaluation, providing:

- Standardized query syntax
- Native JSON support
- Dot notation for nested access
- Rich function library

5 Evaluation

5.1 Performance

We evaluated `ja` on datasets ranging from 1MB to 1GB:

Operation	**1MB**	**100MB**	**1GB**
Select	0.1s	4.2s	42s
Project	0.1s	3.8s	38s
Join	0.3s	12.1s	125s
GroupBy	0.2s	8.5s	87s

Table 1: Operation performance on varying dataset sizes

Memory usage remained constant across dataset sizes due to streaming.

5.2 Usability Study

A survey of 12 data scientists showed:

- 92% preferred REPL over CLI for exploratory analysis
- 83% found dot notation intuitive for nested data
- Average learning time: 15 minutes to productivity

5.3 Real-World Usage

`ja` has been used in production for:

- Log analysis (filtering and aggregating web service logs)
- Data pipeline transformation (ETL intermediate processing)
- API response processing (extracting and joining data from multiple endpoints)
- ML dataset preparation (filtering, sampling, feature extraction)

6 Discussion

6.1 Design Trade-offs

Subprocess vs. Library Calls: The REPL uses subprocess calls rather than direct function invocation. While this incurs overhead, it ensures behavioral consistency and simplifies implementation.

Temporary Files vs. Memory: Intermediate results are stored in files rather than memory. This maintains streaming properties and enables inspection but increases I/O.

Name Uniqueness: Requiring unique names for all datasets prevents accidents but may feel restrictive. We chose safety over convenience.

6.2 Future Work

Potential extensions include:

- Parallel processing for operations on large datasets
- Incremental computation for interactive responses
- Query optimization and physical plan generation
- Integration with distributed processing frameworks
- Visual data flow representation
- Persistent workspace and history
- Model Context Protocol (MCP) server implementation to expose JSONL algebra operations as tools for LLMs with tool-use capabilities, enabling natural language data manipulation

7 Conclusion

We have presented JSONL Algebra, a tool that brings relational algebra to semi-structured JSONL data through both a CLI and interactive REPL. Our novel workspace model enables exploratory data analysis while maintaining streaming efficiency and safety through non-destructive operations. The system demonstrates that classical database theory remains relevant and powerful when adapted to modern data formats and usage patterns.

The tool is open source and available at `https://github.com/queelius/jsonl-algebra`.

References

[1] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 13(6):377-387, 1970.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1997.

[3] JSON Lines. `https://jsonlines.org/`

[4] jq - Command-line JSON processor. `https://jqlang.github.io/jq/`

[5] JMESPath - JSON Query Language. `https://jmespath.org/`

[6] Miller - Like awk, sed, cut, join, and sort for data formats such as CSV, TSV, JSON, JSON Lines, and positionally-indexed. `https://miller.readthedocs.io/`