The Three-Pillar Taxonomy: A Compositional Architecture for Data Manipulation Systems

Alexander Towell
Southern Illinois University
Department of Computer Science
Edwardsville, IL, USA
lex@metafunctor.com, atowell@siue.edu

***Abstract*—We present the *three-pillar taxonomy*, a novel organizational principle for data manipulation systems that decomposes operations into three orthogonal concerns: *Depth* (addressing and extraction), *Truth* (boolean logic and validation), and *Shape* (transformation and modification). Unlike existing approaches that conflate these concerns within monolithic languages, our taxonomy provides clean separation that enables predictable composition while maintaining formal algebraic properties. We formalize each pillar's mathematical foundations, showing how operations lift homomorphically from documents to collections. The taxonomy, implemented in dotsuite (an open-source Python ecosystem), offers both theoretical rigor and practical benefits: composable operations following the Unix philosophy, progressive learning paths from simple to sophisticated tools, and algebraic properties enabling optimization. We argue that this separation of concerns represents a fundamental design pattern for data manipulation systems, providing a blueprint that balances mathematical elegance with usability.**

***Index Terms*—data manipulation, software architecture, domain-specific languages, functional programming, compositionality**

I. Introduction

Data manipulation systems face a fundamental tension between expressive power and conceptual clarity. SQL provides relational algebra but struggles with nested structures. JSONPath offers path-based addressing but lacks compositional logic. Functional programming languages provide mathematical rigor but often at the cost of accessibility. We propose that this tension arises from the conflation of orthogonal concerns within monolithic systems.

This paper presents the *three-pillar taxonomy*, a novel architectural pattern that decomposes data manipulation into three fundamental concerns: **Depth** (where is the data?), **Truth** (is this assertion valid?), and **Shape** (how should data be transformed?). This taxonomy emerged from the development of dotsuite, a Python ecosystem for manipulating nested data structures, but represents a general organizing principle applicable to any data manipulation system.

The key insight is that by cleanly separating these concerns, we achieve several desirable properties simultaneously:

- **Compositionality**: Operations compose predictably through well-defined interfaces
- **Mathematical soundness**: Each pillar maintains algebraic properties that ensure correctness
- **Pedagogical clarity**: Users can master concepts progressively without cognitive overload
- **Implementation simplicity**: Each tool can be "stolen" – simple enough to copy rather than import

II. The Three-Pillar Architecture

A. The Depth Pillar: Addressing and Extraction

The Depth pillar answers the fundamental question: "Where is the data?" It provides mechanisms for navigating and extracting values from nested structures.

Formal Definition: Let $\mathcal{D}$ be the space of documents, $\mathcal{P}$ be the set of paths, and $\mathcal{V}$ be the universe of values. The Depth pillar implements:

Definition 1 (Path Algebra). *The path algebra $(\mathcal{P}, \cdot, \epsilon)$ forms a monoid where:*

- *$\cdot : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ is path concatenation*
- *ϵ is the empty path (identity element)*
- *$(p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$ (associativity)*

Definition 2 (Addressing Function). *The addressing function $addr : \mathcal{D} \times \mathcal{P} \to \mathcal{P}(\mathcal{V} \cup \{\bot\})$ maps document-path pairs to sets of values, where:*

- *Exact paths: $addr(d, p) = \{v\}$ or $\{\bot\}$*
- *Wildcards: $addr(d, p_1 \cdot * \cdot p_2) = \bigcup_{k \in keys(d')} addr(d'[k], p_2)$*
- *Predicates: $addr(d, p[?\phi]) = \{v \in addr(d, p) \mid \phi(v) = \top\}$*

Implementation Spectrum: The pillar provides tools ranging from simple (`dotget` for exact paths) to sophisticated (`dotpath` engine with JSONPath compatibility), demonstrating that complexity can be introduced gradually without invalidating simpler tools.

B. The Truth Pillar: Logic and Validation

The Truth pillar answers: "Is this assertion true?" It implements a boolean algebra that operates on documents and lifts homomorphically to collections, preserving logical structure across scales.

Mathematical Foundation: The pillar implements a boolean algebra $(B, \wedge, \vee, \neg, \top, \bot)$ where predicates form an abstract syntax tree. The crucial property is the homomorphic lifting:

$$filter(S, p_1 \wedge p_2) = filter(S, p_1) \cap filter(S, p_2) \quad (1)$$

This ensures that logical operations on predicates correspond directly to set operations on results, enabling both lazy evaluation and query optimization.

Compositional Logic: The `dotquery` tool provides a declarative DSL that compiles to the predicate algebra, supporting arbitrary boolean combinations of atomic predicates. The syntax deliberately mirrors natural language ("any equals role admin") to reduce cognitive load.

C. *The Shape Pillar: Transformation and Modification*

The Shape pillar answers: "How should the data be transformed?" It treats transformations as endofunctors on document spaces, with composition forming a monoid structure that ensures predictable chaining.

Mathematical Foundation: Transformations form the category $\mathcal{T}$ where:

- Objects are document spaces
- Morphisms are pure functions $f : D \to D'$
- Composition is function composition with identity

The monoid $(T, \circ, id)$ ensures that transformation pipelines can be optimized through algebraic rewriting while preserving semantics.

Immutability Principle: All transformations return modified copies, never mutating originals. This enables safe concurrent operations and predictable composition.

III. From Documents to Collections

The three-pillar taxonomy naturally lifts from single documents to collections through functorial mappings that preserve structure:

A. *Homomorphic Lifting*

Each pillar operation lifts to collections while preserving its essential properties:

- **Depth**: $map(extract_p, S)$ extracts path p from each document
- **Truth**: $filter(\phi, S)$ selects documents satisfying predicate ϕ
- **Shape**: $map(transform_f, S)$ applies transformation f to each document

The homomorphism ensures that operations can be freely reordered for optimization without changing semantics.

B. *Relational Algebra*

The `dotrelate` tool demonstrates how the taxonomy extends to multi-collection operations. Joins are expressed as:

$$\begin{aligned} join(S_1, S_2, k_1, k_2) = \{merge(d_1, d_2) \mid \\ d_1 \in S_1, d_2 \in S_2, get(d_1, k_1) = get(d_2, k_2)\} \end{aligned} \quad (2)$$

This leverages the Depth pillar for key extraction and the Shape pillar for document merging, showing how pillars compose to enable complex operations.

IV. The Unix Philosophy Applied to Data Structures

The three-pillar taxonomy embodies the Unix philosophy – "do one thing well" – applied to data manipulation:

A. *Composable Tools*

Each tool has a single, clear responsibility and composes through standard interfaces (JSON in, JSON out). This enables pipeline construction:

```
cat data.json | dotstar "users.*.email" | \
  dotquery "contains␣gmail" | \
  dotmod "set␣verified␣true"
```

B. *The "Steal This Code" Principle*

Many tools are deliberately simple enough to copy rather than import. For example, `dotget` is essentially:

```
def get(data, path, default=None):
    for segment in path.split('.'):
        data = data.get(segment, default)
        if data is default: return default
    return data
```

This philosophy acknowledges that sometimes a dependency is more costly than duplication, especially for foundational operations.

V. Pedagogical Value: Progressive Enhancement

The taxonomy supports a natural learning progression:

1) **Novice**: Master single-pillar tools (`dotget`, `dotexists`, `dotmod`)
2) **Intermediate**: Combine pillars (`dotstar` + `dotquery`)
3) **Advanced**: Use compositional engines (`dotpath`, `dotpipe`)
4) **Expert**: Extend the system with custom predicates and reducers

This progression ensures that users are never overwhelmed, while experts retain full power. Crucially, simple tools never become obsolete – a `dotget` call remains the right choice for known paths.

VI. Design Rationale and Expected Benefits

The three-pillar taxonomy is motivated by well-established software engineering principles. We discuss the expected benefits based on these foundations.

A. Compositionality Benefits

By decomposing operations into orthogonal concerns, the taxonomy enables:

- **Reusable primitives**: Each atomic operation (e.g., path extraction, predicate evaluation) can be composed into arbitrarily complex pipelines
- **Linear complexity growth**: Adding new capabilities extends a single pillar without affecting others, avoiding the combinatorial explosion seen in monolithic systems
- **Testability**: Each pillar can be tested in isolation with clear contracts

B. Cognitive Load Considerations

The Unix philosophy of "do one thing well" has proven effective for reducing cognitive load in command-line tools. We apply this principle to data manipulation:

- **Progressive disclosure**: Users learn simple tools first (`dotget`, `dotexists`) before advancing to compositional engines
- **Predictable interfaces**: Each tool has a single responsibility, making behavior easier to reason about
- **Natural mental model**: The three questions (Where? True? Transform?) map to intuitive problem-solving steps

C. Performance Characteristics

The compositional approach introduces some overhead but enables optimizations:

- **Lazy evaluation**: Collection operations can stream results without materializing intermediate collections
- **Algebraic optimization**: The homomorphic structure allows reordering and fusion of operations
- **Simplicity trade-off**: For simple operations, the overhead of composition is negligible; complex pipelines benefit from the algebraic structure

VII. Related Work

A. Path-Based Query Languages

JSONPath [3] pioneered path-based addressing for JSON but lacks compositional logic and transformation capabilities. Our Depth pillar extends JSONPath while maintaining compatibility through the `dotpath` engine.

XPath/XQuery [5] provides powerful XML manipulation with FLWOR expressions but tightly couples querying and transformation, making it difficult to reason about individual operations in isolation.

jq [2] offers a complete JSON manipulation language but combines all three concerns (addressing, logic, transformation) in a single DSL. While powerful, this monolithic approach creates a steep learning curve and limits reusability of sub-operations.

B. Relational and Functional Approaches

SQL [1] provides relational algebra with clear mathematical foundations but struggles with nested structures. The conflation of selection (WHERE) and projection (SELECT) demonstrates the same concern-mixing our taxonomy addresses.

LINQ [10] introduces composable query operators for .NET collections, sharing our compositional philosophy. However, LINQ requires a host language and doesn't cleanly separate our three concerns.

Apache Spark DataFrames [4] provides distributed data manipulation with lazy evaluation. While powerful for big data, the API combines transformation and selection without clear conceptual boundaries.

C. Theoretical Foundations

Functional Programming approaches in languages like Haskell provide strong compositionality through category theory [6]. Our approach achieves similar compositional properties through simpler algebraic structures, avoiding the complexity of advanced type systems.

Datalog [7] offers declarative logic programming for data queries. Our Truth pillar shares the logical foundation but extends it with homomorphic lifting to collections.

VIII. Limitations and Future Work

While the three-pillar taxonomy provides clear benefits, we acknowledge several limitations:

- **Overhead for Simple Tasks**: For trivial operations, the decomposition may introduce unnecessary complexity compared to direct manipulation.
- **Performance Trade-offs**: The compositional approach can prevent certain whole-program optimizations that monolithic systems achieve.
- **Domain Specificity**: The taxonomy is optimized for tree-structured data; graph databases and time-series data may require different organizational principles.
- **Learning Curve**: While individual tools are simple, understanding optimal composition patterns requires experience.

Future work includes:

- Extending the taxonomy to streaming and real-time data processing
- Developing formal verification tools for pillar compositions
- Investigating automatic optimization of pillar pipelines through program synthesis
- Applying the taxonomy to other domains (e.g., configuration management, schema evolution)

IX. Implementation and Availability

The three-pillar taxonomy is implemented in **dotsuite**, an open-source Python ecosystem available at:

https://github.com/queelius/dotsuite

The implementation includes 15 tools across four pillars:

- **Depth** (4 tools): `dotget`, `dotstar`, `dotselect`, `dotpath`
- **Truth** (5 tools): `dotexists`, `dotequals`, `dotany`, `dotall`, `dotquery`
- **Shape** (4 tools): `dotmod`, `dotbatch`, `dotpipe`, `dotpluck`
- **Collections** (2 tools): `dotfilter`, `dotrelate`

All tools are available both as Python APIs and command-line utilities, following the Unix philosophy of composable pipelines. The repository includes comprehensive documentation and a test suite covering all functionality.

X. Conclusion

The three-pillar taxonomy provides a principled decomposition of data manipulation concerns that balances mathematical rigor with practical usability. Our key contributions are:

1) **A novel organizational principle** that cleanly separates addressing (Depth), logic (Truth), and transformation (Shape) concerns in data manipulation systems.
2) **Formal mathematical foundations** showing how each pillar maintains algebraic properties that ensure compositional correctness.
3) **Design principles** grounded in the Unix philosophy and functional programming that enable composability, testability, and progressive learning.
4) **A concrete implementation** (dotsuite) demonstrating the taxonomy's practical viability as an open-source Python ecosystem with 15 composable tools.

The three-pillar taxonomy demonstrates that complex data manipulation need not require complex tools. By respecting the natural structure of the problem domain and maintaining clean separation of concerns, we can build systems that are simultaneously powerful, predictable, and pedagogical. We believe this represents a general design pattern applicable beyond our specific implementation, offering guidance for the next generation of data manipulation systems.

Acknowledgments

We thank the Unix philosophy pioneers for demonstrating that simplicity and composability are not at odds with power, and the functional programming community for showing that practical tools can have solid mathematical foundations.

References

[1] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[2] S. Dolan, "jq: Command-line JSON processor," [Online]. Available: https://stedolan.github.io/jq/. 2013.

[3] S. Goessner, "JSONPath - XPath for JSON," [Online]. Available: https://goessner.net/articles/JsonPath/. 2007.

[4] M. Armbrust et al., "Spark SQL: Relational Data Processing in Spark," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, 2015.

[5] A. Berglund et al., "XML Path Language (XPath) 2.0," W3C Recommendation, 2007.

[6] R. Bird and O. de Moor, *Algebra of Programming*. Upper Saddle River, NJ: Prentice Hall, 1997.

[7] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[8] B.W. Kernighan and R. Pike, *The Unix Programming Environment*. Englewood Cliffs, NJ: Prentice Hall, 1984.

[9] M.D. McIlroy, E.N. Pinson, and B.A. Tague, "Unix Time-Sharing System: Foreword," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1899–1904, 1978.

[10] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling Object, Relations and XML in the .NET Framework," *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 706–706, 2006.

[11] B.C. Pierce, *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.

[12] P. Wadler, "Theorems for free!" *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pp. 347–359, 1989.