SJT (Structured JSON Table): A High-Performance Tabular Format for Web APIs and Data Transmission

Yuki Akai

July 25, 2025

Abstract

We introduce **SJT (Structured JSON Table)**, a lightweight tabular data format that offers significant improvements in encoding/decoding speed and size efficiency over conventional JSON and MessagePack, especially in the context of structured tabular data transmitted via REST APIs or streaming. SJT is not a replacement for JSON but an augmentation: designed to fix the inefficiencies of `Array<Object>`-style JSON by flattening it into a columnar structure inspired by table representations, without departing from JSON compatibility. Benchmarks show that SJT not only reduces payload size but also outperforms native JSON in both parse and generation speed on modern JavaScript engines like V8. SJT can serve as a drop-in format for many data-intensive applications, providing better network performance, lower memory overhead, and faster deserialization time.

1 Introduction

JSON has long been the de facto standard for transmitting structured data across web applications. However, in scenarios where large tabular datasets are transferred (e.g., analytics APIs, dashboards, or log streams), traditional JSON introduces considerable overhead due to its verbose structure—especially the repeated field names within arrays of objects. This leads to larger payloads, slower parsing, and higher memory consumption.

This document proposes **Structured JSON Table (SJT)**, a lightweight, JSON-compatible format optimized for high-performance transmission of structured, column-aligned data.

2 Motivation

REST APIs and many data pipelines often communicate large amounts of tabular data in the form of `Array<Object>`, which is verbose and costly in size and performance. Formats like MessagePack aim to mitigate size but often do not improve speed. SJT leverages columnar layout (shared field names, aligned rows) to optimize both.

3 Format Specification (v1)

An SJT payload is represented as a 2-element array:

```
[
  [["id", "name", "age"]], // SjtHeader
  [[1, "Alice", 30], [2, "Bob", 25]] // SjtData
]
```

Supports arbitrarily nested structures via recursive headers.

This format consists of:

- **SjtHeader**: An array of field names (strings).
- **SjtData**: An array of arrays, where each inner array corresponds to a row, aligned with the header.

The format optionally supports a third element, reserved for metadata (e.g., versioning, schema hints):

```
[
  [["id", "name", "age"]],
  [[1, "Alice", 30], [2, "Bob", 25]],
  { "version": "1.0", "schema": "..." }
]
```

Note: Although the physical representation uses JSON arrays, SJT is *not* merely an array-of-arrays—it is a self-described tabular format with strict structure and semantics. Tools reading SJT should treat it accordingly.

4 Benchmark Results & Performance Analysis

To evaluate the efficiency of the SJT format compared to common serialization formats, we conducted performance benchmarks under realistic data transmission scenarios.

Test Conditions

- **Dataset:** A synthetic tabular dataset containing 50,000 records with mixed primitive fields, nested arrays, and nested objects (representative of large REST API payloads).
- **Runtime:** Node.js 20 (V8 engine)
- **Implementation:** JavaScript (via `sjt.js`)
- **Size (KB):** Uncompressed size in kilobytes (estimated for binary formats)
- **Encode / Decode (ms):** Average time in milliseconds to serialize/deserialize the entire dataset

Format	Size (KB)	Encode (ms)	Decode (ms)
JSON	3849.34	41.81	51.86
JSON + gzip	379.67	55.66	39.61
MessagePack	2858.83	51.66	74.53
SJT (JSON)	2433.38	36.76	42.13
SJT + gzip	359.00	69.59	46.82

Table 1: Serialization benchmark results

Key Observations

- **SJT (JSON)** reduced payload size by ∼37% compared to plain JSON and also demonstrated faster encoding/decoding performance due to minimized structural redundancy.
- When compressed (gzip), **SJT + gzip** achieves nearly identical size to **JSON + gzip**, but with lower decode overhead than MessagePack.
- **MessagePack** performs well in size but exhibits slower decoding, likely due to binary buffer parsing and lack of structural alignment for tabular data.
- **SJT** benefits from a consistent schema and monomorphic access patterns, which modern JavaScript engines like V8 optimize effectively.
- **Better GC behavior:** SJT arrays create less memory fragmentation.
- **Structural compactness:** minimized AST depth leads to faster traversal and serialization.

5 Why Is SJT Faster Than Regular JSON?

Although SJT is still serialized and parsed using `JSON.stringify` and `JSON.parse`, its internal structure enables significantly faster post-parse processing compared to standard `Array<Object>` JSON.

Columnar Layout Advantage

Instead of encoding each record as a standalone object with repeated keys, SJT separates field names (schema) from values:

```
// SJT format
[
  [["id", "name", "age"]], // Header (column names)
  [[1, "Alice", 25], [2, "Bob", 30]] // Data rows aligned by column
]

// Regular JSON
[
  { "id": 1, "name": "Alice", "age": 25 },
  { "id": 2, "name": "Bob", "age": 30 }
]
```

This columnar structure provides several low-level performance benefits:

- **Reduced key comparisons:** Engines avoid repeatedly parsing and matching string keys for each object.
- **Linear decoding:** Data can be reconstructed using tight loops without dynamic object allocation.
- **Improved CPU cache locality:** Arrays of homogeneous values are better optimized for memory access than scattered object fields.
- **Schema enforcement by design:** All rows are guaranteed to align with the declared header, eliminating the need for missing or extra field checks.

As a result, even though SJT passes through `JSON.parse` initially, it leads to faster downstream transformation, especially in performance-critical applications such as analytics pipelines or frontend data visualization.

6 Applications & Advantages in Storage and Querying

Structured JSON Table (SJT) is not only a compact data exchange format, but also a practical alternative to JSON for use cases involving data storage, retrieval, and query performance optimization. Its separation between structure (`header`) and content (`body`) enables several advantages, especially in read-heavy or resource-constrained environments.

6.1 Column-Like Access Pattern

Because each key is stored only once in the `header`, and values are aligned by their position in `body`, SJT mimics the behavior of column-oriented storage systems. Querying a specific field can be performed efficiently by:

- Locating the key's index in `header`
- Extracting the value at that index from each row in `body`

This allows for fast projection of selected fields without needing to fully deserialize each object, as required in traditional JSON.

6.2 Reduced Overhead in Repeated Structures

For datasets with repeated keys (e.g., rows of records with the same fields), SJT minimizes storage redundancy by avoiding the repetition of field names. This not only reduces file size but also simplifies parsing logic by allowing consumers to rely on a consistent schema during traversal.

6.3 Compatibility with Indexed Storage

Since `header` provides an explicit key ordering, it can be used to build lightweight indexing schemes or facilitate binary search over sorted data. This can be especially effective when combined with static or predictable schemas (e.g., logs, analytics, time series).

6.4 Use in Embedded or Constrained Environments

Due to its compactness and deterministic layout, SJT is well-suited for use in environments where performance or memory is limited, such as:

- Mobile applications
- Browser extensions (via fetch + lightweight decode)
- Embedded systems
- Serverless functions with cold start constraints

6.5 Lazy Decoding and Streaming Potential

The structure of SJT allows partial parsing or lazy decoding strategies—consumers may defer parsing of nested values or skip unknown fields without affecting overall data correctness. This opens the door to efficient streaming implementations or integration with columnar analytics pipelines.

6.6 High-Performance Data Transmission

Because of its reduced size and structural predictability, SJT is highly applicable for data **transmission over networks**, especially in scenarios where payload efficiency and fast parse time are critical:

- **Compression-friendly structure:** The non-repetitive key layout improves compression ratios (even with gzip or Brotli).
- **Minimal parse cost:** JSON parsers can decode data using index-based dispatch, avoiding costly key string matching.
- **Consistent schema:** Receivers can prepare parsers in advance, reducing runtime overhead.

Key transmission scenarios include:

- REST or GraphQL APIs returning large structured responses
- Edge computing and CDN-cached JSON payloads
- JavaScript/TypeScript-based analytics tools with frequent data sync
- Data pipelines bridging frontend $\leftrightarrow$ backend
- Low-memory embedded JSON readers in IoT or mobile contexts
- Columnar serialization for browser-side analytics and visualizations

6.7 Extensibility and Future Directions

SJT's tabular backbone and recursive support for nested structures position it as a hybrid between relational (SQL-style) and document-oriented (NoSQL) data representations. This opens the door to potential applications beyond mere data interchange:

- **Native indexing:** Header-based indexing could enable columnar access and selective queries, akin to column stores or vectorized engines.
- **Structured yet flexible storage:** SJT can represent deeply structured records without enforcing rigid schemas, offering an efficient middle ground between row- and document-based systems.
- **Potential database integration:** With further development, SJT could serve as an internal serialization or archival format for document databases, offering high compression without losing the ability to map fields precisely.

Although SJT is not a replacement for BSON, Avro, or Parquet, its compactness and schema-aware design suggest it may fulfill a distinct niche—particularly in systems that require fast transmission, selective field access, and minimal payload overhead.

If developed further, SJT could evolve into a foundational data format bridging the gap between lightweight interchange and high-performance storage. Its characteristics enable efficient transport-layer encoding while remaining extensible enough to support database-layer semantics such as projection, filtering, and indexing.

7 Limitations

While the Structured JSON Table (SJT) format provides significant improvements in encoding size, decoding speed, and tabular clarity, it also comes with inherent limitations:

- **Strict Schema Requirements:**
  SJT enforces uniqueness of keys within each header scope (including nested headers). It disallows complex dynamic structures like varying keys across rows, which are otherwise allowed in standard JSON.
- **No Native Support for Rich Types:**
  SJT does not represent advanced types such as `Date`, `BigInt`, or binary data. Applications must handle such conversions explicitly, typically by preprocessing the data before encoding.

8 Related Work

Several existing formats aim to improve over JSON:

- **MessagePack**: Efficient binary JSON; improves size, not structure.
- **Apache Arrow**: Columnar in-memory format, but non-JSON compatible.
- **JSON Lines**: Line-delimited objects; easier to stream but still redundant.
- **FlatBuffers, Cap'n Proto**: Highly performant, but binary and non-human-readable.

SJT distinguishes itself by maintaining **JSON compatibility**, offering structural compactness without requiring binary encoding or specialized tooling.

9 Compatibility & Future Work

- Fully JSON-compatible (parses with `JSON.parse`).
- The metadata section is reserved for schema validation, versioning, etc.
- Language-agnostic: works naturally in JS/TS, Python (via `zip`), Go, Rust.
- Fully supports `null`, nested arrays, and deep object trees — as long as each column in the data array aligns consistently with the defined header structure.

10 References

Since SJT is a novel format inspired by both JSON and CSV, this specification does not rely on prior published work. However, related technologies include:

1. JSON (ECMA-404 Standard)
2. CSV (RFC 4180)
3. `https://msgpack.org/`
4. `https://arrow.apache.org/`
5. `https://capnproto.org/`
6. `https://google.github.io/flatbuffers/`
7. `https://jsonlines.org/`

11 Implementation & Resources

- **Spec repo**: `https://github.com/SJTF/SJT`
- **JS Implementation**: `https://github.com/yukiakai212/SJT.js`
- **NPM Package**: `sjt.js`
- Experimental decoder benchmarks in Python and Rust.

12 License

MIT License