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

Yuki Akai

2025-07-25

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": "..." }
]
```

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. Benchmarks Summary

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

5. Performance Notes

Despite requiring a conversion step (`JSON.parse  SJT.decode()`), SJT outperforms JSON in both speed and size. This is attributed to:

- **Reduced object allocation**: fewer nested objects, no repeated field names.
- **Better GC behavior**: SJT arrays create less memory fragmentation.
- **V8 optimization**: modern JS engines optimize monomorphic array access far more efficiently than dynamic object parsing.
- **Structural compactness**: minimized AST depth leads to faster traversal and serialization.

Note: Even though SJT still passes through `JSON.parse` before decoding, its internal structure allows significantly faster access and lower GC overhead than traditional JSON.

6. Applications

- REST/GraphQL APIs returning large datasets
- Edge computing and CDN-cached JSON payloads
- JavaScript/TypeScript-based analytics tools
- Data pipelines bridging frontend backend
- Low-memory embedded JSON readers
- Columnar serialization for browser/edge-side analytics

7. 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.

8. Compatibility & Future Work

- Fully JSON-compatible (parses with `JSON.parse`).
- A validator/spec tool is being developed for SJT strict mode.
- 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.

9. References

1. JSON (ECMA-404 Standard)
2. CSV (RFC 4180)
3. MessagePack
4. Apache Arrow
5. Cap'n Proto
6. FlatBuffers
7. JSON Lines

10. Implementation & Resources

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

11. License