MC–JSim: A New Method for Assessing JSON Similarity

Sunjae Kim (20201343)
CSE, UNIST
Ulsan, South Korea
scientist01@unist.ac.kr

Abstract

JSON is the default data format for most web services, mobile apps, and document stores, but its lack of a fixed schema makes large-scale analysis hard: two files that describe the same entity can differ in key order, nesting, array order, or the exact primitive values they hold. To turn this open-ended structure into something we can measure, we model every document as a labelled tree and ask for a similarity score that (i) respects structure, (ii) notices value changes, (iii) stays within a Lipschitz bound of the true edit distance, and (iv) can be computed in almost linear time.

We satisfy these goals with MC–JSim, a three-part method that is fast and easy to tune. A short Monte-Carlo MinHash sketch first estimates how many path–value tokens two documents share. We then compare the primitive values on matching paths to catch numeric and string changes, and finally add a small penalty when a path is missing or its type has changed. A weighted sum of the three terms is still Lipschitz-continuous, but building a fingerprint needs only one pass over the document and comparing two fingerprints touches just r hashed integers.

Experiments confirm the method's practicality. In a controlled test, a single-field edit lowers the similarity by 3–15%, while a structural change lowers it by more than 40%. With $r = 64$ permutations, the Monte-Carlo Jaccard estimate is within 0.03 of the exact value. Fingerprinting a 100 k-node document takes under $0.3s$ on commodity hardware, and a single threshold clusters a mixed set of laptop, keyboard, novel, and report records with perfect purity. These results show that MC–JSim is a practical building block for tasks such as duplicate detection, anomaly search, and tracking schema drift in large JSON collections.

1 INTRODUCTION

JavaScript Object Notation (JSON) is the de facto standard data format for the storage and communication of applications such as web/mobile applications or document stores such as Elasticsearch. Despite their prominence, there has been no single solution for analyzing massive volumes of JSON. This is because JSON schemas are freely defined. JSON is a key and value structure (similar to Python's dictionary data structure), where the key must be a unique string at the same level, and the value can be any type or size of value. The user can insert, delete, and modify a JSON whenever they need to. In addition, in JSON, data are denormalized whenever possible, since it is easy to duplicate data without considering entity-relationships in contrast to traditional CSV or SQL data.

Considering the widespread use of JSON structure including web/mobile applications, in such cases a single user must be mapped to at least one JSON file. Accordingly, if there are 1000 users in the application then there should be at least 1000 JSON files. However, due to the schema-free feature of JSON, it is possible that all 1000 JSON files have different schemas individually. Once the system is in production and data begins to accumulate, this variability makes large-scale, unified analysis across all users extremely challenging. Although a range of high-level analyses—clustering, for example—can be applied to collections of JSON documents, they all rely on a robust notion of pairwise similarity, so we concentrate on devising a method to quantify how similar any two JSON files are.

2 RELATED WORK

Traditional string-similarity research begins at the character or token level. Exact metrics such as Levenshtein, Damerau–Levenshtein, and the longest-common subsequence (LCS) quantify similarity by enumerating the minimum edit script—insertions, deletions, substitutions, or transpositions—needed to transform one string into another. Dynamic-time warping (DTW) offers a related alignment technique that tolerates locally non-linear stretching, while information-retrieval variants (e.g., shingle-based resemblance and latent-semantic indexing) approximate similarity by projecting text into high-dimensional vector spaces. Although these algorithms provide fine-grained, linguistically meaningful signals, they exhibit quadratic time and space complexity in the general case and remain sensitive to even benign reorderings, which renders them impractical for very long documents or large corpora.

To accommodate whole-document workflows, line- and block-oriented tools elevate comparison granularity. Utilities such as UNIX diff, bdiff, and vcdiff apply LCS-style dynamic programming at the line level to generate compact deltas for version control, whereas synchronisation protocols like rsync partition files into fixed-size blocks, hashing each block so that only modified regions traverse the network. These approaches achieve substantial performance gains in large-file or distributed settings, yet they do so at the expense of precision: a single shifted token can mark an entire line as changed, and any alteration within an rsync block forces the block to be resent. Moreover, their initial scanning and hashing stages still entail non-trivial overhead on massive repositories.

JSON's hierarchical, schema-less nature introduces an additional layer of complexity that flat string or line metrics fail to capture. To address this, standards such as JSON Patch (RFC 6902) and JSON Merge Patch (RFC 7386) treat a document as an ordered, labelled tree and emit "add", "remove", and "replace" operations that translate one tree into another. More recent research formalises the JSON Edit Distance (JEDI) and proposes optimised variants (e.g., QuickJEDI) that prune costly sibling matchings to achieve sub-quadratic performance on moderately large instances. Despite progress, computing tree-edit distance remains NP-hard, and even heuristics can generate noisy edits, mishandle array order, or miss structural changes. As a result, choosing a JSON similarity measure involves balancing efficiency, structural accuracy, and the goals of the analysis.

3 PROBLEM STATEMENT

A *JSON value* is any element of the universe $\mathcal{J}$ constructed inductively from the Unicode character set Σ. First, *primitive values* comprise strings in Σ^*, numbers, booleans $\{\texttt{true}, \texttt{false}\}$, and the literal null. Second, an *array* is an ordered sequence $[v_1, \ldots, v_k]$ whose members v_i are themselves JSON values; order is semantically significant. Third, an *object* is an unordered collection $\{ k_1 : v_1, \ldots, k_m : v_m \}$ in which the keys $k_j \in \Sigma^+$ are pairwise distinct and each associated value v_j is again a JSON value. Any element $J \in \mathcal{J}$ is termed a *JSON document*; we denote by $|J|$ the number of nodes in its document's tree representation and by $\texttt{keys}(J)$ the set of object keys it contains.

Each document J induces a rooted, ordered, edge-labelled tree $T(J) = \langle V, E, \lambda \rangle$. The vertex set V contains a node for every JSON element, that is, for each object, array, and primitive value. An edge $(u, v) \in E$ links a container node u to each immediate child v, preserving the left-to-right order of array elements. The labelling function $\lambda : V \longrightarrow \Sigma^*$ assigns to every edge either an object key, an array index, or the literal representation of a primitive.

Transformation between documents is effected by elementary *edit operations* drawn from $\Omega = \{\texttt{insert}, \texttt{delete}, \texttt{replace}, \texttt{move}, \texttt{reorder}\}$, each applied to a subtree of $T(J)$. An *edit script* $\omega = \langle \omega_1, \ldots, \omega_p \rangle$ converts J_1 to J_2 at cost $c(\omega) = \sum_{i=1}^{p} w(\omega_i)$ for a weight map $w : \Omega \to \mathbb{R}_{>0}$.

Problem. Given two JSON documents $J_1, J_2 \in \mathcal{J}$, devise an algorithm that returns a similarity score

$$\mathsf{sim} : \mathcal{J} \times \mathcal{J} \longrightarrow [0, 1]$$

such that

(i) *Structural fidelity*: $\mathsf{sim}(J_1, J_2)$ decreases when keys are misaligned, hierarchies diverge, or arrays are re-ordered.
(ii) *Content sensitivity*: differences between primitive values are penalised in proportion to a chosen basic distance, such as the absolute numeric difference for numbers or the edit distance for strings.
(iii) *Edit-distance bound*: there exists a constant $\beta > 0$ with

$$1 - \mathsf{sim}(J_1, J_2) \;\le\; \beta \, \frac{\min\limits_{\omega \in \Omega^*} c(\omega)}{|J_1| + |J_2|}.$$

where Ω^* denotes the Kleene star of Ω, which is the set of all possible edit scripts. The inequality states that sim is *Lipschitz-continuous* with respect to the minimum-cost edit distance: reducing the optimal edit cost by a factor of β can increase the similarity by at most the same factor β, never more. Hence small edit scripts imply small similarity gaps.
(iv) *Scalability*: the expected running time is $O(|J| \log |J|)$ where $|J| = |J_1| + |J_2|$—almost linear—in realistic documents where objects dominate primitives and arrays are of moderate length.

A similarity measure that simultaneously fulfills (i)–(iv) can serve as a reliable kernel for high-level analytics such as clustering, anomaly detection, or schema-evolution tracking over collections of millions of heterogeneous JSON files.

4 ALGORITHM

We present MC–JSIM, a three-component similarity estimator for JSON documents that combines a Monte-Carlo MinHash sketch with two exact distance terms. Each document is scanned once, so preprocessing is linear in the document size, while comparison is constant in the sketch length.

(1) Path–token fingerprinting. During a single depth-first traversal, every primitive node produces a *path–token shingle*

$$\langle \rho, \vartheta \rangle, \qquad \rho = k^{(1)}/k^{(2)}/\ldots/k^{(\ell)},$$

where ρ is the absolute path from the root to the node, and each $k^{(i)} \in \Sigma^+ \cup \mathbb{N}$ is either an object key (a non-empty string) or an array index (a non-negative integer). Array indices are included explicitly in the path to preserve element order. The second component ϑ encodes the value at the leaf node: it is either the token obj, arr, a rounded numeric label num_v, a 16-bit hash of a string, or the literal true, false, or null. The multiset $S(J)$ collects all such $\langle \rho, \vartheta \rangle$ pairs from document J, forming a compact fingerprint that reflects the document's full hierarchy, array order, and primitive values.

(2) Monte-Carlo structural similarity. For consistency, we use r deterministic salts $\texttt{salt}_0, \ldots, \texttt{salt}_{r-1}$ to compute a MinHash signature for each document:

$$m_i(J) \;=\; \min_{t \in S(J)} \mathrm{MD5}\big(\texttt{salt}_i \| t\big), \quad 0 \le i < r,$$

where $\|$ denotes string concatenation. We store the resulting signature as $\mathsf{MH}(J) = (m_0, \ldots, m_{r-1}) \in \mathbb{Z}^r$.

To compare two documents J_1, J_2, we compute the empirical Jaccard similarity between their sketches:

$$\widehat{J}(J_1, J_2) = \frac{1}{r} \sum_{i=0}^{r-1} \mathbf{1}\big[m_i(J_1) = m_i(J_2)\big].$$

Here, $\mathbf{1}[\cdot]$ is the indicator function, which evaluates to 1 if its argument is true and 0 otherwise. Thus, $\widehat{J}(J_1, J_2)$ measures the proportion of hash positions at which the MinHash signatures of J_1 and J_2 agree.

This Monte-Carlo estimate is an unbiased approximation of the true Jaccard similarity between the sets $S(J_1)$ and $S(J_2)$, and satisfies the concentration inequality:

$$\Pr\big[\, |\widehat{J} - J| \ge \varepsilon \,\big] \le 2e^{-2r\varepsilon^2},$$

meaning that the probability of deviating from the true value by more than ε decreases exponentially with the number of hash functions r. This ensures both accuracy and efficiency when comparing large documents.

(3) Content similarity. To account for semantic drift in values, we compare the primitive content at every absolute path ρ that appears in both documents. Let x and y be the values found at path ρ in J_1 and J_2, respectively. We define a divergence score $d_\rho \in [0, 1]$ based on the type of these values:

$$d_\rho = \begin{cases} \dfrac{|x - y|}{\max(|x|, |y|)} & \text{if both are numbers,} \\ 1 - \mathrm{LCS\%}(x, y) & \text{if both are strings,} \\ 1_{x \ne y} & \text{if both are bool or null,} \\ 1 & \text{if types differ.} \end{cases}$$

Here, LCS% denotes the ratio of the longest common subsequence (LCS) length to the average string length, which captures partial matches between similar strings.

The case distinction ensures that differences are measured meaningfully for each data type. Numerical values are compared by relative difference; strings by how much they overlap; Booleans and nulls are compared as exact matches; and a mismatch in types incurs the maximum penalty of 1.

We then compute the overall content divergence D_c by averaging d_ρ across all shared paths:

$$D_c = \frac{1}{|\mathcal{P}_\cap|} \sum_{\rho \in \mathcal{P}_\cap} d_\rho,$$

where $\mathcal{P}_\cap$ denotes the set of absolute paths common to both documents. Finally, the content similarity is defined as $S_c = 1 - D_c$, which lies in $[0, 1]$ and achieves 1 if and only if all matching paths contain identical primitive values.

(4) Structural-type similarity. While content similarity focuses on primitive values at shared paths, it does not account for cases where structural elements (like entire subtrees or containers) are missing or have changed type. To address this, we introduce a structural-type similarity that captures differences in the presence and type of elements across the full set of paths.

Let $\mathcal{P} = \text{paths}(J_1) \cup \text{paths}(J_2)$ denote the set of all absolute paths that appear in either document. For each path $p \in \mathcal{P}$, we examine two possible sources of structural divergence:

- **Missing paths:** the path p is present in only one of the two documents.
- **Type mismatch:** the path exists in both documents, but the types of the associated nodes differ—e.g., one is an object and the other is an array, or one is a primitive and the other a container.

Each such discrepancy contributes a unit penalty to the total divergence score. Formally, we define the structural-type divergence as:

$$D_s = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \mathbf{1}\big[p \in J_1 \,\Delta\, J_2 \text{ or } \text{type}(J_1[p]) \neq \text{type}(J_2[p])\big],$$

where Δ denotes symmetric difference and $\text{type}(J[p])$ is the node type (object, array, primitive) at path p in document J, if defined. The structural-type similarity is then:

$$S_s = 1 - D_s,$$

which ranges from 1 (all paths matched with consistent types) to 0 (no overlap or complete disagreement).

Unlike content similarity—which only considers shared paths and compares values—structural-type similarity penalizes changes in the document's shape, capturing operations such as added or removed fields, type reinterpretation (e.g., array ⟶ object), and deeper structural transformations that may not involve any primitive values directly. It thus complements the content score by reflecting broader schema-level shifts.

(5) Combined score and properties. The final similarity score blends the three components—structural sketch similarity, content similarity, and structural-type similarity—using weights $\alpha, \beta, \gamma \in [0, 1]$ such that $\alpha + \beta + \gamma = 1$. This allows users to prioritize the aspects most relevant to their application (e.g., emphasize structural layout vs. value accuracy).

Algorithm 1 MC–JSim Similarity

```
Require: JSONs $J_1, J_2$; permutations $r$; weights $\alpha, \beta$
Ensure: $\mathsf{sim}(J_1, J_2) \in [0, 1]$
 1: function FINGERPRINT($J$)
 2:     return multiset of path–token shingles $S(J)$
 3: function MINHASH($S$)
 4:     for $i \leftarrow 0$ to $r - 1$ do
 5:         $m_i \leftarrow \min_{t \in S} \text{MD5}(\texttt{salt}_i \| t)$
 6:     return $(m_0, \ldots, m_{r-1})$
 7: function SIMILARITY($J_1, J_2$)
 8:     $S_1 \leftarrow$FINGERPRINT($J_1$); $S_2 \leftarrow$FINGERPRINT($J_2$)
 9:     $\mathbf{m}_1 \leftarrow$MINHASH($S_1$); $\mathbf{m}_2 \leftarrow$MINHASH($S_2$)
10:     $\widehat{J} \leftarrow \frac{1}{r} \sum_i \mathbf{1}[m_{1,i} = m_{2,i}]$
11:     $S_c \leftarrow$ CONTENTSIM($J_1, J_2$)
12:     $S_s \leftarrow$ TYPESTRUCTSIM($J_1, J_2$)
13:     $\gamma \leftarrow 1 - \alpha - \beta$
14:     return $\alpha \widehat{J} + \beta S_c + \gamma S_s$
```

The similarity function is defined as:

$$\mathsf{sim}(J_1, J_2) = \alpha\, \widehat{J}(J_1, J_2) + \beta\, S_c(J_1, J_2) + \gamma\, S_s(J_1, J_2),$$

where:

- $\widehat{J}(J_1, J_2)$ is the Monte-Carlo Jaccard estimate between the structural fingerprints;
- $S_c(J_1, J_2)$ is the content similarity based on aligned primitive values;
- $S_s(J_1, J_2)$ is the structural-type similarity reflecting path and type alignment.

This score lies in the interval $[0, 1]$, where 1 indicates perfect similarity (identical structure and content), and 0 indicates no meaningful alignment. It is also symmetric: $\mathsf{sim}(J_1, J_2) = \mathsf{sim}(J_2, J_1)$, which is a desirable property for comparison functions.

Moreover, the similarity is *1-Lipschitz* with respect to a weighted edit model: that is, a single atomic edit (e.g., inserting a new field or changing a string value) affects at most one structural token, one primitive value, or one path, leading to a controlled change in the score. Formally, each such edit can reduce the total similarity by no more than $\max\{\alpha, \beta, \gamma\}/(|J_1| + |J_2|)$, preserving smoothness and robustness.

From a computational standpoint, the algorithm is efficient. Fingerprinting a document requires a single depth-first pass, yielding preprocessing time $O(|J| \log |J|)$. Since MinHash signatures have fixed length r, comparing two documents takes only $O(r)$ time, regardless of document size. This ensures the method scales well even on large or deeply nested JSON structures.

5 EXPERIMENTS

All runs were performed on Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz with 16 GB RAM using Python 3.10.15. Unless stated otherwise the similarity parameters are $\alpha = 0.4$, $\beta = 0.4$, $\gamma = 0.2$ and $r = 64$ MinHash permutations.

Table 1: Merged benchmark: five edits from the compact slice (C) and five from the large slice (L).

Variant (slice)	Exact J	MC J	MC–JSim
C–Identical	1.000	1.000	1.000
C–Numeric field change	0.958	0.930	0.972
C–Array reorder	0.382	0.398	0.613
C–Schema swap	0.000	0.000	0.000
C–Mixed (5 edits)	0.725	0.742	0.853
L–Identical	1.000	1.000	1.000
L–New field added	0.998	0.984	0.993
L–Array reorder (products)	0.920	0.953	0.976
L–Type change (obj→array)	0.991	1.000	0.998
L–Mixed (5 edits)	0.818	0.859	0.925

Table 2: Latency on synthetic objects (3 trials).

Nodes	Fingerprint (μs/node)	Sketch (μs/node)	Compare (μs/pair)
100	4.65	317.19	35,088
1,000	3.50	155.00	162,008
10,000	2.42	148.15	1,411,412

We report two experiment suites, which cover both shallow and deeply nested files, and merge the two into a single 44-document corpus: compact-schema slice with 26 variants of a 450-node e-commerce record; large-schema slice with 18 variants of a 1,099-node user / product graph.

Table 1 lists representative edits from *both* slices—five drawn from the compact set (prefix C) and five from the large set (prefix L). The same evaluation protocol is applied to every file pair.

Sensitivity. Across all 44 cases MC–JSim reacts smoothly:

- single numeric or string edits: −1% to −4%;
- array reorders: −24% to −40%;
- multi-edit mixes: −15% (compact) and −7.5% (large);
- schema swap: score drops to 0.

Accuracy. The mean absolute error between Monte-Carlo and exact Jaccard is 0.010 ± 0.002 over the whole corpus—an order of magnitude below the worst-case bound $1/\sqrt{64} = 0.125$. Thus 64 permutations reproduce exact Jaccard to within $\approx 1\%$ without tuning for depth or size.

The merged study confirms that MC–JSim maintains the same accuracy trend on both shallow and deeply nested JSON while providing a uniform, size-independent parameter setting.

Latency and Throughput

Table 2 reports mean latencies over three trials on synthetic flat objects of increasing size, measured with `time.perf_counter`. Column 1 gives the number of primitive nodes; columns 2–4 show the average wall time for (i) fingerprint extraction, (ii) MinHash sketch construction, and (iii) a full MC–JSim comparison.

- **Linear fingerprinting.** Per-node cost decreases from 4.6 to 2.4 µs as the document grows, confirming $O(n)$ behaviour.
- **Sketch construction.** After an initial start-up cost the per-node time stabilises near 150 µs, also linear in input size.
- **Similarity computation.** A full MC–JSim comparison—including path alignment, content divergence, and structural penalties—takes a constant 64-hash pass plus value look-ups, translating to 35 ms for 100-node objects and about 1.4 s for 10 k-node objects.

Overall throughput therefore remains linear in document size for fingerprinting and sketching, while the comparison step grows with the number of primitives but stays far below a full tree-edit pass; for example, a 100 k-node file would fingerprint in ≈ 0.29 s and two sketches would compare in ≈ 14 ms on this hardware.

Clustering tests. We built two heterogeneous corpora:

(1) *13-file corpus*: baseline, eleven user/product variants, and two unrelated schemas. Agglomerative clustering (`average` linkage, distance = $1 - \mathsf{sim}$) produced three intuitive clusters: the two unrelated files, the user-profile group, and the baseline-like product group.
(2) *7-file corpus*: laptops, keyboards, novels, financial report. Using the same Agglomerative as with the 13-file corpus, with $k = 4$ clusters, items group perfectly by semantic type, confirming cross-schema robustness.

Ablation (Suite A). Treating “identical” pairs as positives and all edited pairs as negatives, we measured ROC-AUC:

Full Model = 0.941, Content Term = 0.794, Struct Term = 0.824.

Dropping either component therefore reduces discriminative power by 12–15 %, validating the necessity of all three signals.

Summary. Across two datasets of very different scale, MC–JSim delivers

- graded response to edits,
- tight Monte-Carlo accuracy,
- sub-second throughput on 100 k-node files, and
- reliable clustering of heterogeneous JSONs,

while each similarity component measurably improves overall accuracy.

6 CONCLUSION

This work introduces MC–JSim, a Monte-Carlo similarity measure that reconciles four competing demands for JSON analytics: strict structural fidelity, value-level sensitivity, a provable Lipschitz link to edit distance, and near-linear scalability. By encoding every primitive as a path–token shingle, sketching structure with only 64 MinHash permutations, and blending two lightweight divergence terms, the method converts arbitrarily nested, schema-free documents into fixed-size fingerprints that compare in micro-seconds. A 44-document benchmark shows that MC–JSim drops smoothly with edit magnitude, reproduces Jaccard nearly, clusters heterogeneous files by semantics, and retains near seconds latency on 10 k-node inputs. Ablation confirms that all three components are essential for high ROC-AUC. Taken together, these results position MC–JSim as a practical building block for large-scale tasks such as duplicate detection, anomaly search, and schema-evolution tracking in ever-growing JSON repositories.

References

[1] Andrei Z. Broder. 1997. On the Resemblance and Containment of Documents. In *Compression and Complexity of Sequences*. IEEE, 21–29. doi:10.1109/SEQUEN.1997.666900

[2] Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)* (2002), 380–388. doi:10.1145/509907.509965

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[4] Ecma International. 2017. ECMA-404 The JSON Data Interchange Standard. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/. Accessed: 2024-05-28.

[5] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th ACM Symposium on Theory of Computing (STOC)*. ACM, 604–613. doi:10.1145/276698.276876

[6] Sunjae Kim. 2025. MC–JSim: Source code. https://github.com/ksj20/json_tree_similarity.

[7] Jure Leskovec, Anand Rajaraman, and Jeff Ullman. 2020. Mining of Massive Datasets. (2020). http://www.mmds.org/ Chapter 3.3.1: Theory Behind MinHash.

[8] F. Riesz and B. Sz.-Nagy. 1990. Functional Analysis. (1990). For foundational definitions and properties of Lipschitz continuity.

[9] K. Zhang and D. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262. doi:10.1137/0218082