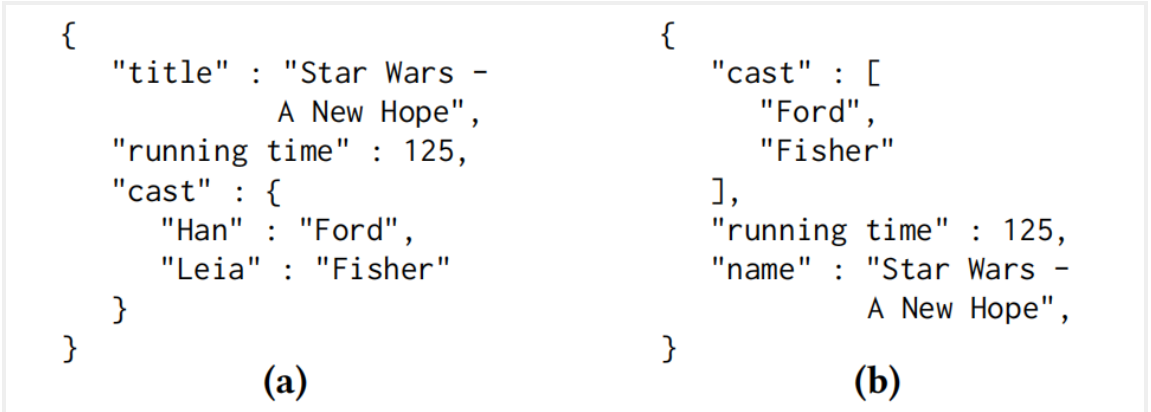
JEDI: These aren't the JSON documents you're looking for…

This research article examines the issues of JSON similarity queries and proposes unique ways to effectively retrieve comparable JSON documents, such as the JEDI distance measure, the QuickJEDI algorithm, and the JSIM index. It emphasizes the significance of these contributions in effectively managing JSON data in a variety of application domains.

Intro:

JSON, a popular data format, is used for a variety of purposes, including publishing datasets and simplifying data interaction in mobile and web apps. Its versatility, due to the lack of a defined schema, allows for a wide range of data representations. Consider a case in which a web crawler receives JSON movie data from several sources. Before inserting data, the crawler examines the databases if they are similar inorder to avoid duplicates. It is useless to have near-duplicate queries which occur due to mere changes in key names and structure of DBs.

The absence of a strict schema in JSON data poses challenges in identifying duplicate information, as it allows for various representations of the same data. For instance, when gathering movie details from multiple sources and saving them in a database, the lack of standardized formats results in different JSON structures for the same movie. Consequently, querying for exact duplicates becomes inefficient. The provided example illustrates this issue with two JSON representations of the same movie in Figure 1a and 1b, showcasing how the varying key names and structures hinder the detection of duplicates through conventional methods. This highlights the need for specialized techniques to effectively manage and identify near-duplicate entries in such scenarios.



```
{
   "title" : "Star Wars -
            A New Hope",
   "running time" : 125,
   "cast" : {
      "Han" : "Ford",
      "Leia" : "Fisher"
   }
}
```

(a)

```
{
   "cast" : [
      "Ford",
      "Fisher"
   ],
   "running time" : 125,
   "name" : "Star Wars -
            A New Hope",
}
```

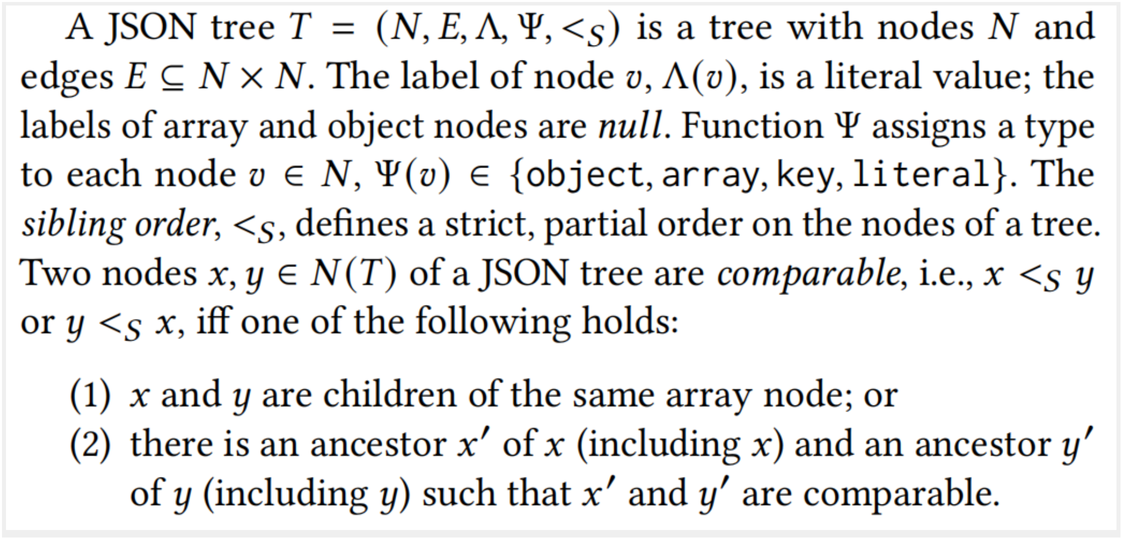
(b)

Figure 1: Two JSON representations of the same movie.

Edit Based Distance for JSON Trees

For the given two arrays ['A', 'B', 'C', 'D'] and ['B', 'C', 'D'], the array index keys of all identical elements differ due to element 'A' that is not present in the second array. Thereby, generating an error of $O(n)$ when a single element in an array of size n is missing

New concept of a JSON tree allowing a lossless transformation between JSON documents and JSON trees.



A JSON tree $T = (N, E, \Lambda, \Psi, <_S)$ is a tree with nodes N and edges $E \subseteq N \times N$. The label of node v, $\Lambda(v)$, is a literal value; the labels of array and object nodes are *null*. Function Ψ assigns a type to each node $v \in N$, $\Psi(v) \in \{$`object`, `array`, `key`, `literal`$\}$. The *sibling order*, $<_S$, defines a strict, partial order on the nodes of a tree. Two nodes $x, y \in N(T)$ of a JSON tree are *comparable*, i.e., $x <_S y$ or $y <_S x$, iff one of the following holds:

(1) x and y are children of the same array node; or
(2) there is an ancestor x' of x (including x) and an ancestor y' of y (including y) such that x' and y' are comparable.

Json Tree Transformation

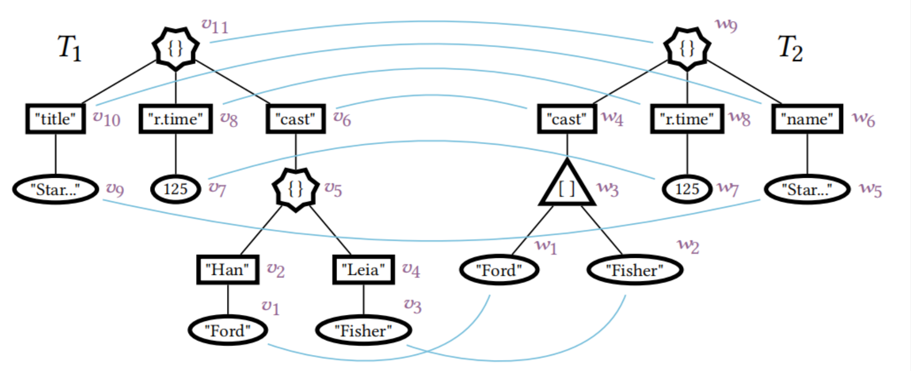


Figure 2: JSON trees of the documents in Figure 1. Object nodes are visualized as stars with symbol { }, array nodes as triangles with symbol [ ], keys as rectangles, and literals as ellipses with their original labels, respectively. Blue lines – depict the JSON edit mapping, and v_i, w_j are node identifiers.

<u>JEDI</u>

It is the minimum number of edit operations required to transform one tree to the other.

JSON Edit Mapping
The edit mapping aligns the nodes of the input trees, $T1$ and $T2$, and must respect some constraints to be valid. The interpretation is as
follows: nodes in $T1$ that are not mapped are deleted, nodes in $T2$ that are not mapped are inserted, and nodes that are mapped are renamed.
The constraints imposed on the mapping control which edit operations are allowable depending on the tree context

Definition 1 (JSON Edit Mapping). *A mapping* $M \subseteq N(T_1) \times N(T_2)$ *is a* JSON edit mapping *from* T_1 *to* T_2 *iff the following constraints hold for any node pairs* $(v, w), (v', w'), (v'', w'') \in M$*:*

1. $v = v'$ *iff* $w = w'$ [one-to-one],
2. v *is an ancestor of* v' *iff* w *is an ancestor of* w' [ancestor],
3. $type(v) = type(w)$ [type],
4. *if* $v <_S v'$ *and* w *is comparable to* w' *in* $<_S$, *then* $w <_S w'$ [array-order],
5. $lca(v, v')$ *is a proper ancestor of* v'' *iff* $lca(w, w')$ *is a proper ancestor of* w'' [document-preserving].

The cost of all edit operations is one except for rename: if the labels of the mapped nodes are identical, then the cost is zero.
The cost of an edit mapping, γ (M), is the total cost of all edit operations.
<u>The edit distance is defined as the cost of the edit mapping with the lowest cost.</u>

We develop a lossless tree representation of JSON and the function guarantees that the difference is minimal and the document nesting is respected.

Baseline Algorithm (A Recursive Solution)

$$df(\epsilon, \epsilon) = 0; \; dt(\epsilon, \epsilon) = 0$$

$$df(v, \epsilon) = \sum_{c \in chd(v)} dt(c, \epsilon); \; dt(v, \epsilon) = df(v, \epsilon) + \gamma(v, \epsilon) \tag{1}$$

$$df(\epsilon, w) = \sum_{c' \in chd(w)} dt(\epsilon, c'); \; dt(\epsilon, w) = df(\epsilon, w) + \gamma(\epsilon, w)$$

$$df(v, w) = \min \begin{cases} df(\epsilon, w) + \min_{c' \in chd(w)} \{df(v,c') - df(\epsilon,c')\} & (2a) \\ df(v, \epsilon) + \min_{c \in chd(v)} \{df(c, w) - df(c, \epsilon)\} & (2b) \\ \text{Min-cost-matching } (chd(v), chd(w)) & (2c) \end{cases}$$

$$dt(v, w) = \min \begin{cases} dt(\epsilon, w) + \min_{c' \in chd(w)} \{dt(v, c') - dt(\epsilon, c')\} & (3a) \\ dt(v, \epsilon) + \min_{c \in chd(v)} \{dt(c, w) - dt(c, \epsilon)\} & (3b) \\ df(v, w) + \gamma(v, w) & (3c) \end{cases}$$

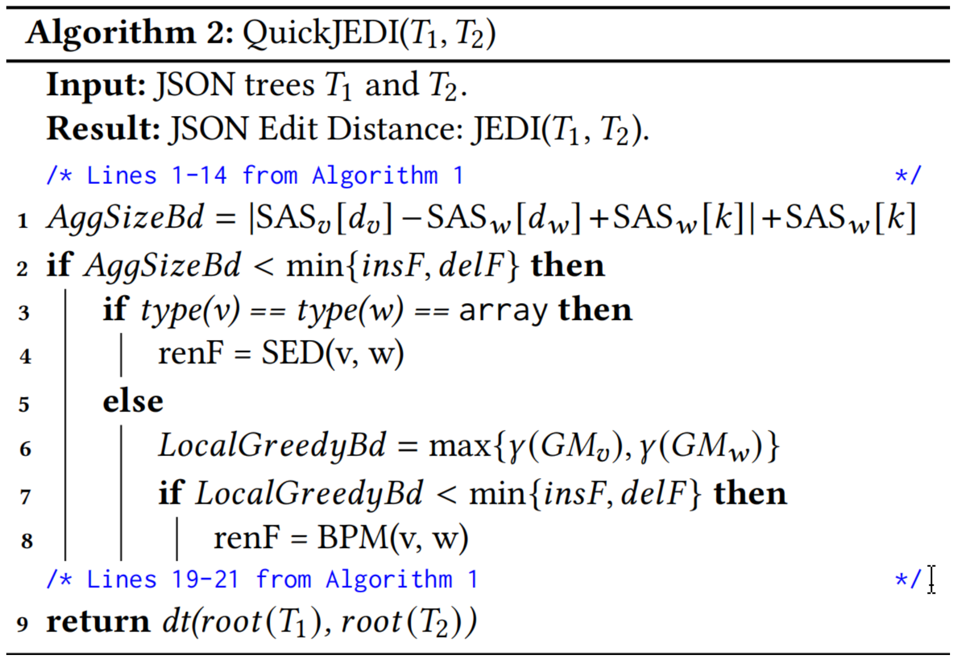
dt(v,w) = Tree distance between subtrees T[v] and T[w]
df(v,w) = Forest distance between subforest F[v] and F[w]
$\gamma\ (v, \epsilon)$ = cost of Deleting a node
$\gamma\ (\epsilon, w)$ = cost of Inserting a node
$\gamma\ (v, w)$ = rename cost

Avoiding the Expensive Min-Cost Matching

It's not necessary to know the exact values for the three evaluations, it's sufficient to know the relative order in order of values to determine the evaluation which leads us to the minimum value.
Efficiency is crucial since the lower bound filter will be evaluated in addition to the min-cost matching.
Whenever the filter cannot avoid the matching computation. The min-cost matching is a bipartite graph matching in the unordered case and a sequence edit distance computation in the ordered case.
Since the sequence edit distance cannot be smaller than the bipartite graph matching cost, we focus on the bipartite graph matching.

QuickJEDI (Recursive Solution)

This algorithm leverages a novel technique, the aggregate size bound, to prune the expensive min-cost matching between sibling sets in each recursive step



Algorithm 2: QuickJEDI(T_1, T_2)

```
Input: JSON trees $T_1$ and $T_2$.
Result: JSON Edit Distance: JEDI($T_1$, $T_2$).
/* Lines 1-14 from Algorithm 1 */
$AggSizeBd = |SAS_v[d_v] - SAS_w[d_w] + SAS_w[k]| + SAS_w[k]$
if $AggSizeBd < \min\{insF, delF\}$ then
    if $type(v) == type(w) ==$ array then
        renF = SED(v, w)
    else
        $LocalGreedyBd = \max\{\gamma(GM_v), \gamma(GM_w)\}$
        if $LocalGreedyBd < \min\{insF, delF\}$ then
            renF = BPM(v, w)
/* Lines 19-21 from Algorithm 1 */
return $dt(root(T_1), root(T_2))$
```

JSIM (Performance Booster for JSON Similarity)

only searches the τ-range around the query document instead of scanning all documents. JSIM is a 4-level tree and each level routes the search into one or more branches. A new technique allows us to reduce the τ-range at each level, thus reducing the total number of explored branches.

```
Algorithm 5: JOFilter($T_1, T_2, \tau$)
Input: JSON trees $T_1$ and $T_2$, and threshold $\tau$.
Result: True if JediOrder($T_1$, $T_2$) $\leq \tau$, False otherwise.
for $v$ in $T_1$ do                                   /* Favorable child order */
    p = p(v)
    for $w$ with $|post(v) - post(w)| \leq \tau$ do    /* Postorder */
        /* Cost for inserting node $w$. */
        insF = w.df$_\epsilon$ + min$_{c \in chd(w)}$\{v.df(c) - c.df$_\epsilon$\}
        insT = w.dt$_\epsilon$ + min$_{c \in chd(w)}$\{v.dt(c) - c.dt$_\epsilon$\}
        /* Costs for deleting and renaming already computed. */
        renF = v.sed$_{L_0}(w_t)$        /* $w$'s rightmost child $w_t$. */
        v.df(w) = min{insF, v.delF(w), renF}
        renT = v.df(w) + min{$\gamma(v, w)$, $\gamma(v, \lambda) + \gamma(\lambda, w)$}
        v.dt(w) = min{insT, v.delT(w), renT}
        /* Compute deletion and rename costs for parent. */
        if v is favorable child then
            p.dt$_{fc}$(w) = v.dt(w)
            p.delF(w) = v.df$_\epsilon$ + v.df(w) - v.df$_\epsilon$
            p.delT(w) = v.dt$_\epsilon$ + v.dt(w) - v.dt$_\epsilon$
        else
            p.delF(w) = min{p.delF(w), p.df(0)+v.df(w)-v.df(0)}
            p.delT(w)=min{p.delT(w), p.dt(0)+v.dt(w)-v.dt(0)}
        if v is left-most child then
            p.sed$_{L_1}$(0) = p.sed$_{L_0}$(0) + v.dt$_\epsilon$
            p.sed$_{L_1}$(w) = min{p.sed$_{L_1}$(ls(w)) + w.dt$_\epsilon$, w.sed$_\epsilon$ +
              v.dt$_\epsilon$, ls(w).sed$_\epsilon$ + v.dt(w)}
        else if v is not favorable child then
            p.sed$_{L_1}$(0) = p.sed$_{L_0}$(0) + v.dt$_\epsilon$
            p.sed$_{L_1}$(w) = min{p.sed$_{L_1}$(ls(w)) + w.dt$_\epsilon$,
              p.sed$_{L_0}$(w) + v.dt$_\epsilon$, p.sed$_{L_0}$(ls(w)) + v.dt(w)}
    if v is left sibling of favorable child $c_f$ then
        p.sed$_{L_0}$(0) = p.sed$_{L_1}$(0) + p.dt$_{fc}$(0)
        for $w$ with $|post(v) - post(w)| \leq \tau$ do
            p.sed$_{L_0}$(w) = min{p.sed$_{L_0}$(ls(w)) + w.dt$_\epsilon$,
              p.sed$_{L_0}$(w) + p.dt$_{fc}$(0), p.sed$_{L_0}$(ls(w)) + p.dt$_{fc}$(w)}
    else
        p.sed$_{L_0}$ = p.sed$_{L_1}$
return root($T_1$).dt(root($T_2$)) $\leq \tau$
```

JSIM index leverages a novel lower bound for JSON trees,called JSON region bound, that is based on the position of a node in the tree. Based on this lower bound and a node label filter, we build an effective multi-level index that only returns trees $Ti \in$ T that pass all filters. Thereby, allowing us to aggressively prune index branches at deeper index levels.

<u>Results</u>

Measuring the effectiveness of the index by the number of returned candidates, The returned candidates are orders of magnitude smaller than the collection size. Due to which the index outperforms scan in each experiment.

-> (Scan, QuickJEDI, JOFilter) outperforms the
index-based solution with baseline verification (JSIM, Baseline)

-> comparing the state-of-the-art JediOrder algorithm (Wang) with the optimized algorithm (JOFilter) : Even for larger thresholds, JOFilter is superior to Wang due to the quadratic complexity of the latter.

In Summary,the best performance results are achieved by
combining the JSIM index, JOFilter, and QuickJEDI. This configuration provides the lowest runtimes for 90% cases.