JEDI: These aren’t the JSON documents you’re looking for...

Thomas Hütter  
Nikolaus Augsten  
thomas.huetter@plus.ac.at  
nikolaus.augsten@plus.ac.at  
University of Salzburg  
Austria

Christoph M. Kirsch  
ck@cs.uni-salzburg.at  
University of Salzburg  
Austria

Michael J. Carey  
chenli@ics.uci.edu  
University of California, Irvine  
USA

ABSTRACT
The JavaScript Object Notation (JSON) is a popular data format used in document stores to natively support semi-structured data. In this paper, we address the problem of JSON similarity lookup queries: given a query document and a distance threshold \( \tau \), retrieve all documents that are within \( \tau \) from the query document. Different from other hierarchical formats such as XML, JSON supports both ordered and unordered sibling collections within a single document which poses a new challenge to the tree model and distance computation. We propose JSON tree, a lossless tree representation of JSON documents, and define the JSON Edit Distance (JEDI), the first edit-based distance measure for JSON. We develop QuickJEDI, an algorithm that computes JEDI by leveraging a new technique to prune expensive sibling matchings. It outperforms a baseline algorithm by an order of magnitude in runtime. To boost the performance of JSON similarity queries, we introduce an index called JSIM and an effective upper bound based on tree sorting. Our upper bound algorithm runs in \( O(\pi \tau) \) time and \( O(n + \pi \log n) \) space, which substantially improves the previous best bound of \( O(n^2) \) time and \( O(n \log n) \) space (where \( n \) is the tree size). Our experimental evaluation shows that our solution scales to databases with millions of documents and JSON trees with tens of thousands of nodes.

CCS CONCEPTS
• Information systems → Semi-structured data; Similarity measures; Database query processing; Proximity search.

KEYWORDS
JSON edit distance, similarity lookup queries, document stores

ACM Reference Format:

1 INTRODUCTION
The JavaScript Object Notation (JSON) has evolved into one of the most prominent data formats. It is used in a large variety of scenarios, e.g., to publish datasets [20, 54] or as an open-standard data interchange format in mobile and web applications [19]. JSON-like formats are also used in document stores to natively support semi-structured data [1, 6, 39].

As JSON does not enforce a schema, it increases the variety in which a given piece of information can be represented. Consider the following scenario: a web crawler collects movies in JSON format from multiple sources and saves them in a document store. In order to avoid duplicate entries, the crawler queries the database for the existence of the movie to be inserted. However, a query for exact duplicates is not effective since key names or the structure will typically vary. Consider the document in Figure 1a, which was discovered by the crawler. The document in Figure 1b already resides in the database. Both documents represent the same movie, but due to their different representations a query for exact duplicates will fail. Detecting near-duplicate entries remains a challenge.

Figure 1: Two JSON representations of the same movie.

In this paper, we study the following problem of similarity queries for JSON: given a query document \( T_q \), retrieve all documents \( T_i \) from a database of JSON documents that are within a given distance threshold \( \tau \) from \( T_q \). To answer such queries, a distance function that assesses the similarity of two JSON documents is needed.

JSON similarity queries are poorly supported in many existing systems. Stand-alone tools for JSON differences are either line-based hence ignore the hierarchical information [15, 25], or do not provide any guarantees on the quality of the result [11]. Similarity functions related to JSON in database systems are limited to basic values, e.g., strings and sets, and cannot be used to compute the distance between JSON documents [17, 33, 35, 39, 42]. For other data formats, similarity queries and the related distance measures are well studied, e.g., for XML [16, 21, 32, 37] data, which – like JSON – is a hierarchical data format. Common approaches for XML are based on the well-known tree edit distance [41], which is the minimal difference between two documents respecting both their hierarchical structure and data values. For JSON, assessing a minimal, edit-based difference is still an unsolved problem.

Assessing the edit-based difference between JSON documents is challenging. JSON differs from other hierarchical data formats,
We now introduce a new distance measure that assesses the minimal \( \tau \) operations and a novel tree representation of JSON data. To our knowledge, this measure is the first distance for JSON that respects its nested structure and provides quality guarantees, i.e., the difference defining the distance is guaranteed to be minimal.

The JSON Data Format. We recap the definition of the JSON data format (cf. RFC8259 [8]). A JSON document is recursively composed of values, arrays, and objects: (1) A value is either a literal (string, number, boolean, or null), an object, or an array. (2) An array is an ordered, possibly empty list of values enclosed by brackets. (3) An object is an unordered, possibly empty collection of key-value pairs enclosed by curly braces. The keys (called "names" in [8]) are string literals that are unique within an object.

Example 1. The JSON document in Figure 1a is an object of three key-value pairs. The keys are "title", "running time", and "cast". The value of "cast" is an object, and the other values are string and number literals.

2.1 JSON Tree Representation

Due to its recursive definition, JSON is hierarchically structured and naturally represented as a tree. The specifics of transforming a JSON document into a tree, however, are not obvious. Previous attempts to model JSON as trees are unsuitable for distances based on a minimal number of node edit operations because either (1) the object and array information is not modelled [34, 46], e.g., `['A', []]` and `['A']` are transformed to identical trees such that the structural information is lost; or (2) arrays are modeled as objects with the array order as a key [7, 47], which generates an error of \( O(n) \) when a single element in an array of size \( n \) is missing. Consider 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. Tree models for JSON documents are not suitable since XML is ordered by definition; although XML has been modeled as unordered trees to capture the semantics of data-centric XML [2], these models do not support a mix of ordered and unordered siblings.

JSON Tree. We introduce the new concept of a JSON tree. The constraints that we impose on JSON trees model all aspects of JSON data and allow for 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 \( \Psi \) assigns a type to each node \( v \in N, \Psi(v) \in \{\text{object}, \text{array}, \text{key}, \text{literal}\} \). 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 \( y <_S x \), if 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 \( x \) itself) and an ancestor \( y' \) of \( y \) (including \( y \)) such that \( x' <_S y' \).

In the second condition, \( x <_S y \) iff \( x' <_S y' \). Intuitively, the order among the children of an array node imposes an order on the subtrees rooted in these children; all other nodes are incomparable. The children of an object node (i.e., key nodes) must have unique labels among their siblings.
null that satisfies these requirements is the edit distance, which has the minimality of the similarity value. A well-known approach is to aim for a similarity measure that captures fine-grained differences.

2.2 JSON Edit Distance (JEDI)

The edit distance is defined as the cost of the edit mapping with the minimum number of edit operations required to transform one tree to the other. Allowable operations include: delete node \( v \) and connect its children to the parent of \( v \); insert a new node \( w \) between an existing node \( v \) and a possibly empty subset of \( v \)’s children; and rename the label of node \( v \).

**JSON Edit Mapping.** Following previous works, we formally define the JSON edit distance using the concept of an edit mapping. The edit mapping aligns the nodes of the input trees, \( T_1 \) and \( T_2 \), and must respect some constraints to be valid. The interpretation is as follows: nodes in \( T_1 \) that are not mapped are deleted, nodes in \( T_2 \) 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; they are discussed in detail below.

**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 \( (v, w) \), \((v', w')\) \( \in M \):

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. \( \text{type}(v) = \text{type}(w) \) [type].
4. \( v <_S v' \) and \( w \) is comparable to \( w' \) in \( <_S \), then \( w \prec_S w' \) [array-order].
5. \( \text{lca}(v, v') \) is a proper ancestor of \( v'' \) iff \( \text{lca}(w, w') \) is a proper ancestor of \( w'' \) [document-preserving].

A mapping \( M' \subseteq M \) between two subtrees \( T_1[v] \) and \( T_2[w] \) is an edit mapping iff \( M' \) is an edit mapping from \( T_1[v] \) to \( T_2[w] \).

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, \( \gamma(M) \), is the total cost of all edit operations. The edit distance is defined as the cost of the edit mapping with the lowest cost.

**Example 3.** Figure 2 shows a JSON edit mapping between two JSON trees. The cost of the mapping is 5: delete nodes “Han”, “Le1a”, and ( ) from the left tree; insert ( ) into the right tree; rename “t i 1 e” to “name”. There is no mapping with a lower cost, thus JEDI is 5.

Constraints (1) and (2) of the edit mapping ensure that the node mapping can be interpreted as a set of edit operations. Constraint (3) ensures that labels can only be renamed between nodes of the same type. This prevents that nodes with identical labels but different types (e.g., a key and a literal value may have identical labels) are mapped at zero cost, thus ignoring their difference. Note that it is still possible to substitute (delete and insert) a node of one type by a node of another type, but the cost is higher than for rename. The array-order constraint (4) uses the partial order \( <_S \) defined on JSON trees to enforce the order imposed by array nodes; children of object nodes are not restricted and can be arbitrarily mapped.
Document-Preserving Constraint. The recursive definition of JSON gives rise to its nested document structure. A nested document (e.g., representing the cast of a movie) often is meaningful only in the context of the enclosing document (in the example, the movie the cast belongs to). Constraint (5), the document-preserving constraint, forces the edit mappings to respect the nested document structure of JSON and leads to more intuitive mappings. In particular, shortcuts that delete the root nodes of subtrees (thus disassembling the documents they root), rearrange their nested subtrees, and recompute the nested subtrees into new documents by inserting new subtree roots are prevented. We illustrate the effect of this constraint in Example 4.

![Figure 3: Edit mappings with and without the document-preserving constraint.](image)

**Example 4.** Consider the schematic illustration of the two JSON trees \(T_1\) and \(T_2\) in Figure 3. Each tree consists of two subtrees \(T_1[v_1]\), \(T_1[v_2]\) resp. \(T_2[w_1]\), \(T_2[w_2]\), which in turn are composed of an object node and two smaller subtrees \(T_1[x_i]\) resp. \(T_2[y_j]\) each, \(i,j\leq 4\). The subtree pairs \(T_1[x_i]\) and \(T_2[y_j]\), \(i = j\), are identical (same color in the figure) and are all of size \(n\). Any two subtrees \(T_1[v_1]\) and \(T_2[w_1]\), \(i \neq j\), are different with an edit mapping of cost \(O(n)\).

The minimum-cost edit mapping (with document-preserving constraint) will delete \(T_1[x_2]\) and \(T_1[x_3]\), and insert their identical counterparts \(T_2[y_2]\) and \(T_2[y_3]\) since they belong to different documents in \(T_2\). An edit mapping that does not respect the document-preserving constraint, however, has only cost 8: delete nodes \(o_1, o_2, o_1, o_2\), insert \(o_3\) as parent of \(y_1, y_2\), \(o_4\) as parent of \(y_2, y_4\); \(w_1\) as parent of \(o_3\); and \(w_2\) as parent of \(o_4\). Without the document-preserving constraint, rearranged subtrees form new documents, which is not desired for JSON trees.

As a pleasant side effect, the document-preserving constraint substantially reduces the search space for the minimal cost mapping and allows for faster algorithms. In fact, we show that finding a minimum JSON edit mapping that ignores the document-preserving constraint is an NP-hard problem [30]. The proof is by reducing the problem of exact cover by 3-sets (X3C).

**Theorem 1.** Without the document-preserving constraint, the problem of computing the JSON edit distance between two JSON trees is NP-hard.

3 AN EFFICIENT ALGORITHM FOR JEDI

Next, we introduce QuickJEDI, an efficient algorithm for computing the JSON edit distance. We first discuss a baseline solution, analyze its performance bottlenecks, and finally propose effective techniques to address these bottlenecks.

3.1 A Baseline Algorithm

None of the previous algorithms that computes the minimum edit distances between trees is applicable in our scenario due to the type and the array-order constraints in the JSON edit mapping (cf. Definition 1). Our baseline extends two algorithms for the so-called constrained tree edit distance. These algorithms compute minimal edits under the document-preserving constraint (constraint (5) in the JSON edit mapping) for ordered [59] resp. unordered trees [60]. Since a single JSON tree may include both ordered and unordered siblings, neither of the two algorithms is applicable; also, these algorithms deal with generic trees and do not consider node types.

We recap the solutions for the constrained tree edit distance and show how they can be extended to compute the JSON edit distance. Both algorithms are based on a recursive solution that is implemented using dynamic programming.

![Figure 4: Recursive decomposition of two trees; pairs of subtrees resp. subforests of the same color form the subproblems required to compute the distance btw. \(T[v]\) and \(T[w]\).](image)

**Recursive Solution.** The recursive solution decomposes two trees \(T_1\) and \(T_2\) with root nodes \(v\) in \(T_1\) and \(w\) in \(T_2\) into subtrees and subforests as illustrated in Figure 4. The distance between \(T_1\) and \(T_2\) is computed from the distances between the subproblems resulting from their decomposition. With \(d(f, v)\) we denote the tree distance between subtrees \(T_1[v]\) and \(T_2[w]\), and \(df(v, w)\) denotes the forest distance between subforests \(T_1[e]\) and \(T_2[w]\). Then, the recursive solution is defined as follows:

\[
\begin{align*}
df(e, e) &= 0; \quad df(v, e) = 0 \\
df(v, e) &= \sum_{c \in chd(v)} dt(c, e); \quad dt(v, e) = df(v, e) + \gamma(v, e) \quad (1) \\
df(e, w) &= \sum_{c \in chd(w)} dt(e, c); \quad dt(e, w) = df(e, w) + \gamma(e, w) \\
df(v, w) &= \min \left\{ \begin{array}{l}
df(v, e) + \min_{c \in chd(v)} (dt(c, w) - dt(c, e)) \\
\text{Min-cost-matching (chd(v), chd(w))} \\
\text{Min-cost-matching (chd(v), chd(w))} \\
df(w, v) + \gamma(v, w) \\
\end{array} \right. \quad (2a)
\end{align*}
\]

The tree distance, \(dt(v, w)\), is the minimum cost of three scenarios (cf. Figure 4 and Eq. 3), each of which represents an edit operation: (3a) \(w\) is inserted, hence the nodes in subtree \(T_1[v]\) are mapped to the nodes of one of \(w\)’s children \(T_2[c]\) (green), (3b) \(v\) is deleted, hence the nodes of subtree \(T_2[w]\) are mapped to the nodes of one of \(v\)’s children \(T_1[c]\) (red), and (3c) \(v\) is mapped to \(w\) with rename cost \(\gamma(v, w)\), hence also the subtrees of their children are mapped (blue); \(\gamma(v, w)\) and \(\gamma(e, w)\) denote the cost of deleting resp. inserting a node. The cost of matching the children of nodes \(v\) and \(w\) in scenario (3c)
is equivalent to their forest distance $df(v, w)$. The base cases of the recursion are shown in Eq. 1. The forest distance, $df(v, w)$ (cf. Eq. 2) is computed analogously for insertion (2a) and deletion (2b). In the third scenario (2c), a minimum-cost matching between the subtrees rooted in $chd(v)$ and $chd(w)$ is established.

The minimum-cost matching $M$ is one-to-one and models the subtrees rooted in $chd(v)$ and $chd(w)$ as nodes of a bipartite graph (cf. Figure 5); the cost of an edge between two subtrees rooted in $c_i\in chd(v)$ and $c'_j\in chd(w)$ is their tree distance, $dt(c_i, c'_j)$. In the unordered case [60], the minimum-cost bipartite graph matching, $MBPM(v, w)$, with cost $γ(M_{MBPM(v, w)}) = BPM(v, w)$ must be computed (e.g., using a min-cost max-flow algorithm [50]). In the ordered case [59], the subsequence edit distance matching, $M_{SED(v, w)}$, with cost $γ(M_{SED(v, w)}) = SED(v, w)$ must be computed.

**Adaptation to JSON.** In Lemma 1, we show how previous solutions can be extended to compute JEDI between two JSON trees.

**Lemma 1.** Given two JSON trees $T_1$ and $T_2$, the recursive formulas (1), (2), and (3), compute the JSON edit distance between $T_1$ and $T_2$, $JEDI(T_1, T_2) = dt(root(T_1), root(T_2))$ with the following extensions.

1. The minimum-cost matching $M \subseteq chd(v) \times chd(w)$ observes the node type:

   $$ M = \begin{cases} M_{SED(v, w)} & \text{if type}(v) = \text{type}(w) = \text{array} \\ M_{BPM(v, w)} & \text{otherwise} \end{cases} $$

(4)

2. The rename cost must be redefined as follows:

   $$ γ'(v, w) = \begin{cases} γ(v, w) & \text{if type}(v) = \text{type}(w) \\ γ(v, e) + γ(e, w) & \text{otherwise} \end{cases} $$

(5)

**Dynamic Programming Implementation.** Algorithm 1 implements the recursive solution of Lemma 1. The results for subproblems are stored in two matrices, $dt$ and $df$, each of size $(|T_1| + 1) \times (|T_2| + 1)$. The distance between subtrees $T_1[v]$ and $T_2[w]$ is stored in row $v$ and column $w$, and we refer to the value as $dt(v, w)$; similarly, $df(v, w)$ stores the distance between subforests $F_1[v]$ and $F_2[w]$. Table 1 shows examples of a forest and a tree distance matrix.

**Initialization:** The first row and column of each matrix are initialized in lines 1-8. Mapping two empty trees has cost 0; for all other nodes, the cost results from summing up the deletion resp. insertion costs of their child subtrees, e.g., $dt(\epsilon, e) = 6$ (cf. Table 1) is the cost of deleting the node of subtree $\epsilon$ in Figure 2.

**Distance Computation:** The algorithm processes the tree nodes bottom-up in postorder and the distance matrices are filled row by row. We label the three cases in Eq. 2 (forest distance) with insF (2a), delF (2b), and renF (2c); the cases in Eq. 3 (tree distance) are labeled insT (3a), delT (3b), renT (3c). Due to the postorder traversal, all values required to compute $dt(\epsilon, w)$ and $df(v, w)$ are available in the distance matrices. To compute renF, a min-cost matching $M$ among the children must be established. If both $v$ and $w$ are array nodes (ordered case), the edit distance between ordered sequences of siblings establishes the min-cost matching (line 16), in all other cases a bipartite graph matching must be computed (line 18). The distance between $T_1$ and $T_2$ results in the lower right corner of the tree distance matrix, e.g., $dt(root(T_1), root(T_2)) = 5$ in Table 1.

**Complexity:** The space complexity is dominated by the distance matrices of size $O(|T_1||T_2|)$. The runtime is dominated by the bipartite graph matching, which for a node pair $v, w$ with degrees $d_v = deg(v)$ and $d_w = deg(w)$ is computed in time $O(d_v \times d_w \times (d_v + d_w) \times \log(d_v + d_w))$ using a min-cost max-flow algorithm [50]. For the overall algorithm (cf. Algorithm 1), the runtime complexity is $O(|T_1||T_2| \times (deg(T_1) + deg(T_2)) \times \log(deg(T_1) + deg(T_2)))$ [60].

<table>
<thead>
<tr>
<th>$df$</th>
<th>$\epsilon$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_9$</th>
<th>$dt$</th>
<th>$\epsilon$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>$d_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>$d_9$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>$d_9$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>$d_{10}$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$d_{10}$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$d_{11}$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>$d_{11}$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$d_{13}$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>$d_{13}$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Forest and tree distance matrices $df$ and $dt$ for the JSON trees in Figure 2.

### 3.2 Avoiding the Expensive Min-Cost Matching

JEDI must compute the min-cost matching between the child subtrees of each node pair of the input trees. This step is expensive and dominates the overall runtime. In this section, we show that the expensive min-cost computation can be avoided in many cases, thus substantially improving the runtime of the distance computation.

The key idea is that the min-cost matching in Eq. (2) is the minimum of three values. Two of them are efficient to compute, one is the expensive matching. If we can show that the cost of the matching is higher than one of the other two values, the exact cost of the matching is irrelevant and the computation can be skipped.

We are the first to follow this approach. The challenge is to identify a lower bound on the min-cost matching that is both effective and can be computed efficiently. 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.

Figure 5 illustrates the bipartite graph for the nodes $chd(v)$ and $chd(w)$. The edge cost $cost(c_i, c'_j)$ between two nodes $c_i \in chd(v)$ and $c'_j \in chd(w)$ is the tree distance between the subtrees rooted in these nodes, $dt(c_i, c'_j)$. To simplify the presentation, we assume $l = deg(v) < deg(w) = m$, i.e., $k = m - l$ subtrees will be matched to the empty tree. We denote the cost of the bipartite matching between the children of two nodes $v, w$ with $BPM(v, w)$.

**Aggregate Size Bound.** To establish a lower bound on the bipartite graph matching cost, we leverage the specific characteristics of the edge costs in our scenario. Since the edge costs are given by the respective subtree distances, we can bound the cost...
by the size difference of the subtrees, \( \text{cost}^*(c_i, c'_j) = |(|T[c_i]| - |T[c'_j]|)| \leq \text{cost}(c_i, c'_j) \). A minimal matching \( \text{BPM}^*(v, w) \) that uses \( \text{cost}^*(c_i, c'_j) \) cannot be more expensive than the original matching, \( \text{BPM}^*(v, w) \leq \text{BPM}(v, w) \). We leverage this fact to derive a novel lower bound based on subtree sizes. We define the sorted aggregate size between start \( s \) and end \( e \) in a subforest \( F[v] \) as

\[
\text{SAS}(s, v, e) = \sum_{i=s}^{e} |T[c_i]|, \quad c_i \in \text{chd}(v),
\]

where \( c_i \) is the \( i \)-th smallest subtree in \( F[v] \) (ties broken arbitrarily).

The intuition of our bound is as follows: There exists a matching with cost \( \text{BPM}(v, w) \), \( \text{deg}(v) < \text{deg}(w) \), that matches the \( k = \text{deg}(w) - \text{deg}(v) \) smallest subtrees to the empty tree, inducing cost \( \text{SAS}(s, 1, k) \). The matching cost between the remaining subtrees is no longer than the difference of their aggregate subtree sizes.

**Theorem 2 (Aggregate Size Bound).** Given two JSON tree nodes \( v \in T_1, w \in T_2 \). Let \( d_v = \text{deg}(v), d_w = \text{deg}(w), k = d_w - d_v \), and \( d_v \leq d_w \), then:

\[
\text{BPM}(v, w) \geq |\text{SAS}(v, 1, d_v) - |\text{SAS}(w, k + 1, d_w)|| + |\text{SAS}(w, 1, k)|.
\]

**Example 5.** For the root nodes of the JSON trees in Figure 2, \( \text{BPM}((1), (1)) = 5 \) and the aggregate size bound is 2 (\( \text{SAS}((1), 1, k) = 0 \) since both nodes have the same degree, i.e., \( k = 0 \)). In Figure 6, \( k = 1 \) and the aggregate subtree bound is 9: \( \text{SAS}(w, 1, k) = 5 \) and \( |\text{SAS}(v, 1, d_v) - |\text{SAS}(w, k + 1, d_w)|| = 4 \). Note that our aggregate size bound performs much better than a simple subtree size difference bound, which is \( |\text{SAS}(v, 1, d_v) - |\text{SAS}(w, 1, d_w)|| = 1 \) in this example.

**Efficient Computation of Aggregate Size Bound.** The aggregate size bound requires us to compute sums of subtree sizes. Since the bound is computed \( O(|T_1||T_2|) \) times (for all pairs of parent nodes), computing these sums is too expensive. We precompute an array \( \text{SAS}_v \) of size \( \text{deg}(v) \) for each node \( v \in T_1 \) with \( \text{SAS}_v[1] = \text{SAS}(v, 1, 1) \) (cf. Eq. (6)); analogously \( \text{SAS}_w \) for all \( w \in T_2 \) is computed. Thanks to the SAS arrays we can compute the bound in constant time:

\[
|\text{SAS}(v, 1, d_v) - |\text{SAS}(w, k + 1, d_w)|| + |\text{SAS}(w, 1, k)| = |\text{SAS}_v[d_v] - \text{SAS}_w[d_w] + \text{SAS}_w[k]| + |\text{SAS}_w[k]|.
\]

**Example 6.** \( \text{SAS}_{w_0} = [2, 4, 8] \) for root node \( w_0 \) in Figure 2. Figure 6 shows the SAS arrays for the root nodes \( v \) and \( w \) of the example trees.

**Local Greedy Lower Bound.** The local greedy lower bound on \( \text{BPM}(v, w) \) matches each node by following the lowest cost edge. The result may violate the one-to-one requirement and therefore may not be a valid matching. Similar bounds have been used before (e.g., [45]). Since this bound is as expensive as the sequence edit distance (quadratic in the node degrees as all edge costs must be checked), it is only useful for the bipartite graph matching.

**Lemma 2 (Local Greedy Lower Bound).** Let \( T_1, T_2 \) be JSON trees, \( v \in T_1, w \in T_2 \). Let \( \text{GM}_v \subseteq \text{chd}(v) \times \text{chd}(w) \) map \( c_j \in \text{chd}(v) \) to some \( c_j \in \text{chd}(w) \) such that \( \text{cost}(c_i, c_j) \) is minimal; \( \text{GM}_w \subseteq \text{chd}(w) \times \text{chd}(w) \) is defined analogously:

\[
\text{BPM}(v, w) \geq \max \{|\gamma(\text{GM}_v), \gamma(\text{GM}_w)|\}.
\]

We show how to compute \( \text{GM}_v \) and \( \text{GM}_w \) with low overhead: While we build the bipartite graph and retrieve all edge costs between the children of two nodes \( v, w \), we maintain the minimum cost edge for each node \( c_i \in \text{chd}(v) \) and \( c'_j \in \text{chd}(w) \). In a single pass over the nodes, we get \( \text{GM}_v \) and \( \text{GM}_w \) with linear overhead.

An interesting opportunity arises when \( \text{GM}_v \) or \( \text{GM}_w \) is one-to-one: In this case, we can skip the bipartite graph matching since \( \text{BPM}(v, w) \geq \max \{|\gamma(\text{GM}_v), \gamma(\text{GM}_w)|\} \) and we know the exact costs.

### 3.3 The QuickJEDI Algorithm

We present QuickJEDI, our efficient algorithm for computing the JSON edit distance. QuickJEDI extends JEDI-baseline (Algorithm 1) with the results in Section 3.2. While the baseline must compute the expensive min-cost matching between the children of each node pair \((v, w)\), QuickJEDI checks the aggregate size bound (cf. Th. 2) to assess whether the matching is required. The aggregate size bound
is a lower bound for both types of min-cost matchings: the sequence edit distance, \( SED(v, w) \), for pairs of array nodes, and the bipartite graph matching, \( BPM(v, w) \), which is applied otherwise. Only if the lower bound is smaller than both \( insF \) and \( delF \) (line 2), the min-cost matching must be computed. Before computing \( BPM(v, w) \), we also check the local greedy lower bound (cf. Lemma 2).

We further avoid the min-cost matching for two special cases (omitted in Algorithm 2 for brevity): if both \( v \) and \( w \) are key nodes, they have only one child each (\( c_v \) resp. \( c_w \)), and \( renF = dt(c_v, c_w) \). If both \( v \) and \( w \) are literal values, they are leaves, and \( renF = 0 \).

<table>
<thead>
<tr>
<th>Algorithm 2: QuickJEDI(T1, T2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> JSON trees ( T_1 ) and ( T_2 ).</td>
</tr>
<tr>
<td><strong>Result:</strong> JSON edit Distance: ( \text{JEDI}(T_1, T_2) ).</td>
</tr>
<tr>
<td>/* Lines 1-14 from Algorithm 1 */</td>
</tr>
<tr>
<td>1 ( \text{AggSize} =</td>
</tr>
<tr>
<td>2 if ( \text{AggSize} &lt; \min(\text{insF}, \text{delF}) ) then</td>
</tr>
<tr>
<td>3 ( \text{if type}(v) = \text{type}(w) = \text{array} ) then</td>
</tr>
<tr>
<td>4 ( \text{renF} = SED(v, w) )</td>
</tr>
<tr>
<td>5 else</td>
</tr>
<tr>
<td>6 ( \text{LocalGreedy} = \max{\gamma(GM_v), \gamma(GM_w)} )</td>
</tr>
<tr>
<td>7 ( \text{if LocalGreedy} &lt; \min(\text{insF}, \text{delF}) ) then</td>
</tr>
<tr>
<td>8 ( \text{renF} = BPM(v, w) )</td>
</tr>
<tr>
<td>/* Lines 15-21 from Algorithm 1 */</td>
</tr>
<tr>
<td>9 return ( dt(\text{root}(T_1), \text{root}(T_2)) )</td>
</tr>
</tbody>
</table>

4. **THE JEDIORDER FILTER**

In this section, we propose *JediOrder*, a highly effective upper bound filter on the JSON edit distance. In a JSON similarity query, the upper bound is evaluated before JEDI: if the upper bound is within the similarity threshold \( \tau \), the expensive JEDI needs not be computed.

We discuss Wang’s algorithm [55], the fastest known algorithm that (with some adaptions to JSON trees) computes JediOrder. Wang’s algorithm is faster than JEDI (quadratic vs. cubic) and requires less space. It turns out, however, that Wang’s algorithm is still too slow compared to JSON trees. Thanks to the order, JediOrder does not need to compute a bipartite graph matching, \( BPM(v, w) \), between the children of two nodes \( v \) and \( w \); instead, the cheaper sequence edit distance, \( SED(v, w) \), is evaluated (cf. Section 3.1). Formally, JediOrder is defined as the cost of the min-cost mapping that satisfies Definition 2.

**DEFINITION 2 (ORDERED JSON EDIT MAPPING).** A JSON edit mapping \( M \) is ordered if for any node pairs \((v, w)\), \((v', w')\) \( \in M \):

- \( v \) is to the left\(^1 \) of \( v' \) iff \( w \) is to the left of \( w' \) [order].

**EXAMPLE 7.** Considering the (ordered) JSON edit mappings in Figures 2 and 7, \( \text{JEDI}(T_1, T_2) = 5 \) vs. \( \text{JediOrder}(T_1, T_2) = 8 \). Due to the lexicographical order of the key nodes in Figure 7, the node pairs \((v_9, w_5)\) and \((v_{10}, w_6)\) violate the order constraint and are not in the minimum-cost ordered JSON edit mapping.

The order constraint in Definition 2 subsumes the array-order in Definition 1, thus JediOrder provides an upper bound for JEDI.

**THEOREM 3 (JEDIORDER UPPER BOUND).** Given JSON trees \( T_1, T_2 \), then \( \text{JediOrder}(T_1, T_2) \geq \text{JEDI}(T_1, T_2) \).

4.2 **JediOrder Baseline: Wang’s Algorithm**

JediOrder is based on sorted, hence, ordered trees. As a baseline algorithm for JediOrder, we adopt the state-of-the-art constraint tree edit distance algorithm by Wang and Zhang [55], which runs in \( O(T_1 | T_2|) \) time and \( O(|T_2| \log |T_1|) \) space, to JSON trees.

**Recursive Solution:** The recursive solution discussed in Section 3.1 (cf. Eq. 1-3 and Lemma 1) also holds for JediOrder. Due to the total order among siblings, the minimum-cost matching in Eq. 2c is always computed by the sequence edit distance (rather than the more expensive bipartite graph matching). Zhang [59] shows the correctness of the recursion.

**Memory Efficient Implementation:** Similar to Algorithm 1, Wang’s algorithm uses dynamic programming and a nested loop over all node pairs of the input trees \( T_1 \) and \( T_2 \). To reduce the memory complexity, Wang implements two key ideas: (1) The deletion and rename costs of a node \( v \) in \( T_1 \) (\( delF/delF \) and \( renF/renF \) in Algorithm 1) w.r.t. all nodes \( w \in T_2 \) (inner loop) are computed incrementally while the children of \( v \) are processed (in the outer loop).

---

1. **\( v \) is to the left of \( v' \) if \( v \) is not a descendant of \( v' \) and precedes \( v' \) in postorder.**
The required cost arrays of size $|T_2|$ are maintained with each node $v$; they are allocated when the first child of $v$ is processed and are released after processing $v$. (2) The nodes of $T_1$ (outer loop) are processed in favorable child order, a postorder traversal that visits the so-called favorable child (defined as the child with the largest subtree) first and all other children in the usual left-to-right order. This traversal guarantees that only $\log |T_1|$ nodes $v \in T_1$ maintain their cost arrays concurrently, thus reducing the memory complexity from quadratic to $O(|T_2| \log |T_1|)$. In Figure 7, the favorable children of $T_1$ are marked with an orange bullet $\star$.

We will reuse these concepts and in addition leverage the similarity threshold to evaluate the JediOrder filter in linear time.

4.3 Leveraging the Distance Threshold

In the similarity lookup scenario, we are only interested in assessing whether JediOrder is within the similarity threshold $\tau$. Hence, we do not need to consider mappings $M_{JO}$ with a cost larger than $\tau$.

On top of the two optimizations of Wang’s algorithm (cf. Section 4.2), we add a third key idea: (3) leverage the user-defined similarity threshold $\tau$ in combination with the postorder lower bound (cf. Lemma 3) to reduce the number of relevant node pairs. (3)

Lemma 3 (Postorder Lower Bound [31]). Given an ordered JSON edit mapping $M_{JO}$ with cost $\gamma(M_{JO})$, for every node pair $(v, w) \in M_{JO}$ the following holds: $|\text{post}(v) - \text{post}(w)| \leq \gamma(M_{JO})$.

In similarity queries, the distance is bounded by the threshold $\tau$. Therefore, Lemma 3 implies that there are only $2\tau + 1$ eligible mapping partners $w \in T_2$ for a given node $v \in T_1$ such that the cost of the overall ordered JSON edit mapping is within $\tau$. We refer to the eligible nodes $w \in T_2$ as the $\tau$-range of a node $v \in T_1$.

Example 8. Consider the JSON trees in Figure 7 and a threshold $\tau = 2$. Any ordered JSON edit mapping that maps $v_0$ to a node in $T_2$ and has a cost of at most $\tau$ must map node $v_9$ to a node in its $\tau$-range, i.e., $w_4$, $w_5$, $w_6$, $w_7$, or $w_8$.

Our goal is to apply the $\tau$-range in Wang’s algorithm to avoid the nested loop over all node pairs. In particular, we strive to replace the inner loop over all nodes of $T_2$ by a constant $\tau$-range of $2\tau + 1$ nodes. This has an impact on the computation of the tree, the forest, and the sequence edit distance (SED) matrices.

In the tree and forest distance matrix, at most $2\tau + 1$ cells are filled per row. The other cells are guaranteed to exceed the threshold due to the $\tau$-range and do not need to be computed. Whenever these cells appear in a minimum computation, their value is considered to be infinite. If the overall mapping cost is within the threshold, the matrices store the correct JediOrder values. The correctness proof for the tree and the forest distance matrix is similar to the proof for the SED matrix, which we discuss in detail below.

We leverage the $\tau$-range also for SED, which is used to compute the minimum-cost matching between the ordered children of two nodes. A sequence edit matching must satisfy Definition 3.

Definition 3 (Sequence Edit Matching). Matching $M_{SED}(m,n) \subseteq \text{chd}(m) \times \text{chd}(n)$, $m \in T_1$ and $n \in T_2$, is a sequence edit matching iff for any pairs $(v, w), (v', w') \in M_{SED}(m,n)$ the following holds:

- $v = v'$ iff $w = w'$ [one-to-one].
- $v$ is to the left of $v'$ iff $w$ is to the left of $w'$ [order].

Restricting SED to the $\tau$-range results in $\tau$-restricted SED matchings and the corresponding $\tau$-sequence edit distance ($\tau$SED).

Definition 4 ($\tau$-restricted). Let $M_{SED}(m,n)$, $m \in T_1$ and $n \in T_2$, be a sequence edit matching. $M_{SED}(m,n)$ is $\tau$-restricted iff for any pair $(v, w) \in M_{SED}(m,n)$ the following holds:

- $|\text{post}(v) - \text{post}(w)| \leq \gamma(M_{JO})$

The cost of a minimal SED matching $\gamma(M_{SED}(m,n))$ is identical to the cost of a minimal $\tau$-restricted SED matching $\gamma(M_{SED}(m,n))$ whenever the overall JediOrder value is within the threshold $\tau$ (cf. Theorem 4). Otherwise, $\gamma(M_{SED}(m,n))$ provides an upper bound on $\gamma(M_{SED}(m,n))$ and hence an upper bound on $\gamma(M_{JO})$ is computed. However, only tree pairs with $\gamma(M_{JO}) \leq \tau$ have to be considered in a similarity lookup.

Theorem 4 (Exact $\tau$SED). If the minimal ordered JSON edit mapping $M_{JO}$ between $T_1$ and $T_2$ has a cost of $\gamma(M_{JO}) \leq \tau$, then $\gamma(M_{SED}(m,n)) = \gamma(M_{SED}(m,n))$ for any node pair $(m, n) \in M_{JO}$.

Note that $\tau$SED is superior to a simple approach that uses a threshold on the string edit distance [44]. While $\tau$SED prunes based on the postorder positions in the tree, the latter approach prunes based on the position in the string/sequence. Hence, for subtrees of size larger than one, $\tau$SED provides better pruning power than the simple approach, and the same pruning power otherwise.

Example 9. Table 2 shows the SED matrix for the root node $v_{11}$ and $w_9$ of the trees in Figure 7. Consider node $v_9$ (at sequence position 2 and postorder 8) and a threshold $\tau = 2$. The unrestricted SED must compute all cells of the matrix. The simple threshold-based approach for the string edit distance must compute all cells for nodes with sequence positions $2 \pm 2$, i.e., all nodes $w_6 \in \text{chd}(w_9)$ must be considered. $\tau$SED, however, only computes the cells in the $\tau$-range of the postorder positions (highlighted in green), e.g., for node $v_9$ only nodes with postorder positions $8 \pm 2$ ($w_6$ and $w_8$) need to be considered.

<table>
<thead>
<tr>
<th>$\text{chd}(v_{11})$</th>
<th>$w_4$</th>
<th>$w_5$</th>
<th>$w_6$</th>
<th>$w_7$</th>
<th>$w_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$v_9$</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$w_6$</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$w_8$</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: $\text{SED}(v_{11}, w_9)$ matrix of the root nodes in Figure 7. For threshold $\tau = 2$, $\tau$SED only computes the cells highlighted in green.

4.4 Challenges of Applying the $\tau$-Range

For the loop variables $v \in T_1$ and $w \in T_2$, Wang computes row $v$ of the SED($p(v), w$) matrix (cf. Algorithm 3). This matrix has a row for each child of $p(v)$ and a column for each child of $w$.

To apply the $\tau$-range in Wang’s algorithm, (1) the inner loop over all node pairs must be restricted to the nodes in the $\tau$-range and (2) the SED must be $\tau$-restricted. Unfortunately, extending Wang’s algorithm with the $\tau$-range (highlighted in line 2, Algorithm 3) will lead to incorrect results. Consider the matrix of the SED($v_{11}, w_{18}$) computation between the two identical JSON trees in Figure 8 with
We now present a novel algorithm, called \textit{JOFilter},\footnote{For a detailed discussion of \textit{JOFilter}, see Section 4.5.} which is based on the incremental matching of trees \cite{Wang}. Incremental matching is essential in many large-scale applications where additional input is expected to be processed incrementally, which includes our tree similarity index. The algorithm is able to leverage all key ideas of the space-efficient algorithm by Wang (incremental cost computation and favorable child order, cf. Section 4.2) and the $\tau$-range introduced in Section 4.3. In the following, we discuss the key challenges that must be addressed and show that \textit{JOFilter} runs in $O(n\tau)$ time and $O(n + \tau \log n)$ space.

\begin{algorithm}[H]
\caption{Wang($T_1, T_2, \tau$)}
\begin{algorithmic}[1]
\Function{Wang}{$T_1, T_2, \tau$}
\State /* Outline of SED computation in Wang’s algorithm. */
\For{$v$ in $T_1$}
\For{$w$ in $T_2$ with $\text{post}(v) - \text{post}(w) \leq \tau$}
\For{$c$ in $\text{child}(w)$}
\State Compute cell $(v,c)$ of the SED($p(v), w$) matrix.
\EndFor
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[H]
\caption{JOFilter($T_1, T_2, \tau$)}
\begin{algorithmic}[1]
\Function{JOFilter}{$T_1, T_2, \tau$}
\State /* Outline of SED computation in the JOFilter. */
\For{$v$ in $T_1$}
\For{$w$ in $T_2$ with $\text{post}(v) - \text{post}(w) \leq \tau$}
\State Compute cell $(v,w)$ of the SED($p(v), p(w)$) matrix.
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

Cost arrays of size $\tau$. Similar to Wang’s algorithm, we split the auxiliary matrices into rows and store each row with the relevant nodes $v \in T_1$. A node $v$ maintains the following data: row $v$ of (1) the tree and (2) the forest distance matrix, denoted $v.dt$ and $v.df$, respectively; (3) the tree distance matrix row of $v$’s favorable child, $v.df_c$; finally, (4) two rows of the SED($v, w$) matrices for all $w \in T_2$, denoted $v.sed_{w, \tau}$, and $v.sed_{w, \tau}$, which are sufficient to compute SED \cite{Wang}. Due to the $\tau$-range, the size of these cost arrays (i.e., matrix rows) can be reduced from $O(|T_2|)$ in Wang’s algorithm to $O(\tau)$ in \textit{JOFilter}. Summarizing, a node $v \in T_1$ stores auxiliary data of size $O(\tau)$. Moreover, the insertion (resp. deletion) costs of node $v$ in the forest, tree, and SED matrices, denoted $v.df_c$, $v.df_e$, and $v.sed_{w, \tau}$, are stored in global arrays of size $|T_2|$.

Logarithmic number of active nodes. A node is called active while the node and its auxiliary data are held in main memory. A node $v$ becomes active when its favorable child is processed and inactive after it was processed. The favorable child order guarantees that at most $O(\log |T_1|)$ nodes are active at any time \cite{Wang}.

Applying the $\tau$-range. We apply the $\tau$-range by replacing the inner loop over all nodes $w \in T_2$ by a constant range of $2\tau + 1$ nodes. As shown in Section 4.4, applying the $\tau$-range in Wang’s algorithm leads to incorrect results. We therefore adapt the computation order of the values in the SED computation as shown in Algorithm 4: only a single cell $(v, w)$ of the SED($p(v), p(w)$) matrix is filled in the inner loop rather than an entire matrix row. Since $v$ and $w$ are the loop variables, we guarantee that all node pairs in the $\tau$-range are considered in the SED computation.

\begin{algorithm}[H]
\caption{Wang($T_1, T_2, \tau$)}
\begin{algorithmic}[1]
\Function{Wang}{$T_1, T_2, \tau$}
\State /* Outline of SED computation in the JOFilter. */
\For{$v$ in $T_1$}
\For{$w$ in $T_2$ with $\text{post}(v) - \text{post}(w) \leq \tau$}
\State Compute cell $(v,w)$ of the SED($p(v), p(w)$) matrix.
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

Our JSIM index leverages a novel lower bound for JSON trees, called \textit{JSIM 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 $T_i \in T'$ that pass all filters. Moreover, we introduce a technique that decreases the search threshold level by level during the index lookup. This allows us to aggressively prune index branches at deeper index levels.
Algorithm 5: JOFilt(T₁, T₂, τ)

Input: JSON trees T₁ and T₂, and threshold τ. 
Result: True if JedOrder(T₁, T₂) ≤ τ, False otherwise.

for v in T₁ do /* Favorable child order */
    p = p(v)
    for w with |post(v) − post(w)| ≤ τ do /* Postorder */
        if w is favorable child then
            p.delT(w) = v.dt(v) − v.dt(w)
            p.sed(w) = p.sed(w) + p.dtT(w)
        else
            p.delT(w) = min(p.delT(w), p.dtT(w) − v.dt(w))
            p.sed(w) = p.sed(w) + v.dt(w)
        if v is left-most child then
            v.sed(w) = p.sed(w) + v.dt(w)
            v.sed(w) = v.sed(w) + p.dtT(w)
        else
            if v is not favorable child then
                v.sed(w) = p.sed(w) + v.dt(w)
                v.sed(w) = v.sed(w) + p.dtT(w)
            else
                p.sed(w) = p.sed(w) + p.dtT(w)
        for w with |post(v) − post(w)| ≤ τ do
            p.sed(w) = p.sed(w) + p.dtT(w)
            p.sed(w) = p.sed(w) + p.dtT(w)
        if w is left sibling of favorable child cf then
            p.sed(w) = p.sed(w) + p.dtT(w)
        return root(T₁).dt(root(T₂)) ≤ τ

5.1 Leveraging Node Position and Labels

We now present the JSON region bound that is based on the ancestor constraint of the JSON edit mapping in Definition 1. Assume that the node pair (v, w) in Figure 9 is mapped; then, anc(v) must be mapped to anc(w) (red), desc(v) to desc(w) (green), and lr(v) to lr(w) (blue). The left-right nodes lr(v) of node v are all nodes in Tq different from v, desc(v), and anc(v). Intuitively, the size difference of the individual regions imposes a lower bound on the respective mapping cost. For example, the cost of mapping the ancestors in Figure 9 is at least one.

Lemma 4 (JSON Region Bound). Let T₁, T₂ be JSON trees, M a JSON edit mapping from T₁ to T₂. For a given similarity threshold τ, if the cost of the mapping is γ(M) ≤ τ, then for each (v, w) ∈ M:

|desc(v) − desc(w)| + |anc(v) − anc(w)| + |lr(v) − lr(w)| ≤ τ.

Figure 9: Due to the ancestor constraint, mapping node v to w splits the JSON tree into three regions.

5.2 Index Structure and Lookup

We discuss the structure of the JISIM index and our lookup technique that leverages the filters discussed in Section 5.1.

Building the Index. JISIM is a tree with four levels that store (1) node labels, (2) descendant counts, (3) ancestor counts, and (4) left-right node counts, respectively. Each index node is a sorted list of entries that either points to a child node (non-leaf entry) or to a list of indexed trees (leaf entries).

A new tree Tᵢ is inserted [Tᵢ] times into the index, once for each node. Each node adds a constant number of (at most 5) index entries. Therefore, the overall index size is proportional to the aggregated number of nodes of the indexed JSON trees. The insert path for a node v ∈ Tᵢ is determined by its label, its number of descendants, ancestors, and left-right nodes. New values are inserted into the respective index node, for existing values the child pointer is followed. The process of inserting node v₀ ∈ T₁ from Figure 2 into the index is highlighted in Figure 10a (green). Tree T₁ is inserted with label = "r. time", |desc(v₀)| = 1, |anc(v₀)| = 1, and |lr(v₀)| = 8.

Index Lookup. The lookup for query tree Tq processes τ + 1 nodes v ∈ Tq, and for each node proceeds in two steps: (1) Label lookup: Follow the branch for the label of v in the index root node. The index lookup is limited to only τ + 1 nodes since any tree Tᵢ that has more than τ + 1 mismatching labels with Tq cannot be within edit distance τ [31, 38]. (2) Region traversal: We leverage Lemma 4 to traverse the remaining levels. At each node, we follow all keys k (i.e., region counts) that fall into the range given by Lemma 4, e.g., d = ||desc(v)| − k| ≤ τ at the descendant count level. Note that each of the three size differences (which are all positive) must be within the threshold τ. At the lower index levels, we leverage the size difference that we know from previous levels, e.g., the threshold for the ancestor level can be decreased to τₐ = τ − d and the index verifies all keys kₐ with a = ||anc(v)| − kₐ| ≤ τₐ. The process for the fourth level is similar, we verify all keys k₄ with |lr(v)| − k₄| ≤ τ₄ against an even further reduced threshold τ₄ = τₐ − a. All trees
We experimentally evaluate our solution for JSON similarity lookups on a real-world dataset provided by the state-of-the-art JediOrder algorithm [JOFilter] and our JediOrder filter (Algorithm 5).

Dataset: The evaluation is performed on a collection of 22 real-world JSON datasets. We summarize their most important characteristics: Collection sizes of up to 8.76 million JSON trees; JSON tree sizes of up to 48k nodes; a type distribution within a JSON tree of up to 20% objects, 10% arrays, 49% keys, and 49% literals; the degree of object nodes is typically less than 20 with the exception of one dataset (104); the degree of array nodes is up to 1603 values; one dataset provides a depth of 50 (less than 14 for all other datasets).

We briefly describe the datasets used for the experiments in Figures 11 and 12. (1) FENF [22]: FDA enforcement actions, ~14k documents with an average of 49 nodes per document and a depth of 3. (2) Reddit [43]: 25 Reddit articles with an average of 265 nodes per document. This dataset provides the highest object degree of 104 children. (3) Cards [26]: ~20k Magic cards with an average number of 132 nodes per document. (4) StanDev [53]: question-answering dataset, 48 documents with up to ~18k nodes and an average of 5,379 nodes per document. (5) Movies [40]: TV and movie ratings, ~8.7 million documents with an average of 23 nodes per document. (6) NBA [14]: ~31k NBA games with an average of 977 nodes per document. (7) Device [22]: ~150k FDA enforcement actions with up to 3,264 nodes per document. (8) arXiv [52]: 1.8 million research publications with an average of 53 nodes per publication. (9) Twitter [51]: ~19k tweets with an average of 195 nodes per document. (10) DENF [22]: ~7k FDA enforcement actions with an average of 59 nodes per document. (11) Schema [5]: 81k JSON schemas with up to 48k nodes per schema document. (12) SMSen [13]: ~55k SMS messages with an average of 81 nodes per document.

Experimental Setup: For each dataset, we perform JSON similarity lookup queries for three different query trees and four different thresholds. Since the runtime of the distance algorithms depends on the tree sizes, we pick the query trees that are closest to the 25%, 50%, and 75% quantiles of the tree sizes for each dataset (denoted $T_{25\%}$, $T_{50\%}$, and $T_{75\%}$). The goal of similarity lookup queries is to return documents that are similar to the query document, hence useful thresholds depend on the size of the query tree. We experiment with thresholds that are 5%, 10%, 20%, and 30% of the respective query tree size. The timeout for computing the results for all thresholds for a given algorithm and dataset is 24 hours.

Evaluation: We analyze the overall runtime and the effectiveness of the introduced bounds. Each plot in Figure 11 and 12 shows the results of a single experiment, i.e., a given dataset and query tree for varying thresholds on the x-axis. For example, Figure 11a shows the results for dataset FENF and the 50% quantile query tree $T_{50\%}$.

Figure 11 shows the overall lookup runtimes in milliseconds for various algorithm combinations. Figure 12 evaluates the number of trees pruned by the individual filters as well as the number of required verifications. The total height of a bar is the number of documents (i.e., trees) in the dataset, the colors distinguish the tree pairs that are pruned by the JSIM index (orange), the label intersection (red), the upper bound (purple), and the number of verifications (blue). The runtime and effectiveness plots are aligned, e.g., Figures 11a and 12a result from the same experiment.

The overall experiment includes 66 dataset/query combinations. Due to space restrictions, we provide a representative selection that covers the most relevant phenomena (cf. Figures 11 and 12).

6.2 Results

**JSIM Index vs. Dataset Scan.** We measure the effectiveness of the index by the number of returned candidates (cf. Figure 12).

 Especially for small thresholds, the returned candidates are orders of magnitude smaller than the collection size (e.g., Figure 12b and 12c).

Due to the smaller number of candidates, the index outperforms the scan in each experiment, e.g., the index is up to five orders of magnitude faster in Figure 11f. For larger datasets (e.g., Figure 11e), the index is needed to answer the query within the timeout. In some scenarios, however, applying an index without further optimizations is not enough: The Reddit dataset used for the experiment...
Wang vs. JOFilter. Next, we compare the state-of-the-art JediOrder algorithm (Wang) with our optimized algorithm (JOFilter). The experimental results show the behaviour expected based on the runtime complexities of the algorithms. The complexity of JOFilter depends on the threshold. Even for larger thresholds, JOFilter is superior to Wang due to the quadratic complexity of the latter. We compare the runtimes of Wang (purple) and JOFilter (blue) in Figure 11. In Figure 11a, no candidates must be verified except for threshold 15; hence the runtime improves from JOFilter alone. We observe the largest improvements of JOFilter in Figures 11k and 11l, where Wang is up to an order of magnitude slower.

In many scenarios (cf. Figures 12), the upper bound identifies most of the result set and only few trees must be verified (blue bar). However, the upper bound is applied to each candidate and introduces additional overhead which may increase the runtime in cases where the upper bound is not effective (cf. Figure 11f). These results show that an efficient verification algorithm is indispensable.

Baseline Verification vs. QuickJEDI. We also evaluated the effect of the optimized verification algorithm QuickJEDI over the baseline without applying the upper bound (red stars vs. orange pentagons in Figure 11). The complexities of both algorithms heavily depend on the degrees of the trees. QuickJEDI aims at skipping the expensive min-cost matching computation, which substantially reduces the runtime. Consider the measurements for threshold 55 in Figures 11b and 12b: even though only 16 trees have to be verified, the runtime difference between the baseline and QuickJEDI is almost two orders of magnitude. This results from the characteristics of the Reddit dataset, where some documents feature up to 104 unordered key-value pairs per object. Moreover, in 4 out of the 22 datasets the lookup terminated within the timeout only in configurations that include QuickJEDI (e.g., Figures 11f and 11i).
7 RELATED WORK

JSON Tree Representations. There exist multiple tree representations of JSON documents. Bourhis et al. [7] represent keys and the array order as edges and values as leaf nodes; the approach by Shukla et al. [46] is similar, but keys and the array order are inner nodes instead of edges. Similar to our approach, Klettke et al. [34] introduce three different types of nodes (object, array, property) in addition to the label. Spoth et al. [47] use a tree containing atomic values at the leaves and complex values in the inner nodes. These representations either discard the object and the array information or encode the information in the edges of a tree; both choices are unsuitable for node edit operations. Tree representations of XML data (e.g., by Augsten et al. [2]) cannot be applied in the context of JSON since XML siblings are considered to be ordered.

JSON Similarity: To the best of our knowledge, there is only one scientific work on JSON diffs. Cao et al. [11] present an algorithm that computes a JSON patch based on the edit operations defined in RFC6902 [9]. In an experimental study, a comparison to four open source solutions was performed. However, the runtime and space complexity of the presented algorithm was not discussed. Further, the resulting patch is not minimal and therefore unsuitable for similarity queries. Yahia et al. [56] proposed a YAML-based language for describing change-detection strategies on JSON data.

Diff algorithms do exist for other hierarchical data formats. Chawathe et al. [12] present an algorithm that computes minimal diffs for XML and HTML documents. The following edit operations are considered: insert and delete leaf nodes, update the value of any node, and subtree moves. The XML diffs by Cobena et al. [16] consider insertions and deletions of subtrees, value updates of any node, and moves of a node or a part of a subtree. Both approaches operate on ordered trees and are therefore unsuitable for JSON.

JSON Schema. Most of the scientific work related to JSON deal with schema extraction. Schemas are used as dataset descriptions or to enable optimization techniques in database systems. Durner et al. [18] present a solution to extract multiple local schemas for a single dataset. The schemas are grouped based on the label sets of the keys in a document. Bazzizi et al. [4] introduce a parametric and parallel schema inference algorithm. Klettke et al. [34] present a schema extraction algorithm to identify structural outliers based on structure identification graphs. While the goal of schema extraction is different from that of similarity queries, JEDI could be used to identify schemas for similar documents.

Tree Edit Distance. A well-known edit distance for hierarchical data is the tree edit distance (TED). The current best algorithm for ordered trees by Pawlik and Augsten [41] computes TED in cubic time using quadratic memory. Computing TED for unordered trees is NP-hard [61]. Further, TED was applied for different query types, e.g., similarity joins [31] and top-k similarity joins [36]. However, these techniques are not applicable for JSON since JSON trees consist of ordered as well as unordered children. In fact, we showed that a TED adaption for JSON results in an NP-hard problem.

Zhang introduced a constraint TED version which can be computed in time $O(n^2)$ for ordered [59] and $O(|T_1| \cdot |T_2| \cdot (\deg(T_1) + \deg(T_2)) \cdot \log_2(\deg(T_1) + \deg(T_2)))$ for unordered trees [60]. Similar to TED, both algorithms are designed for either ordered or unordered trees. We combined both approaches to construct the baseline JEDI algorithm. As shown in our experimental evaluation, we introduce heuristics that decrease the runtime of JEDI often by orders of magnitude. The ordered constraint TED algorithm by Wang et al. [55] using $O(n \log n)$ memory was used as a baseline algorithm for JediOrder. We introduced a novel JediOrder algorithm that improves the complexity to be linear in time and space.

Heuristics for the Unordered Tree Edit Distance. Due to the computational complexity of the unordered TED, a number of heuristics have been presented. Augsten et al. [2] introduced an approximation based on tree decomposition, called windowed pq-graums, that splits a tree into a set of smaller elements which are then compared to the decomposition of another tree. They experimentally showed that windowed pq-graums outperform other tree decomposition algorithms (binary branches [37], path shingles [10], and valid subtrees [24]). Rather than introducing approximations, we defined an exact and minimal JEDI distance.

8 CONCLUSION AND FUTURE WORK

In this paper, we addressed the problem of JSON similarity lookup queries: Given a query document $T_q$ and a distance threshold $\tau$, retrieve all documents from a JSON database $T$ that are within distance $\tau$ from the query. We proposed (a) a lossless tree representation for JSON, (b) JEDI, the first edit-based distance for JSON documents, (c) the efficient QuickJEDI algorithm for JEDI, (d) the JSIM index to efficiently retrieve candidate trees for JSON similarity queries, and (e) JediOrder, an effective upper bound on JEDI. In our experiments, we scaled JSON similarity lookup queries to databases with millions of documents and JSON trees with thousands of nodes.

In an ongoing effort, our solution is being integrated into Apache AsterixDB, an open-source big data management system that uses partitioned-parallel query processing and a JSON-like data format.

ACKNOWLEDGMENTS

We thank Wail Alkowaileet, Daniel Kocher, Mateusz Pawlik, and Zhilui Yang for valuable discussions. This work was supported by the Austrian Marshall Plan Foundation, the Austrian Science Fund (FWF): P 29859 and P 34962, the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program, under grant agreement No. 695412, and the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421.