JED: JSON Edit Distance

Marshall Plan Scholarship

FINAL RESEARCH PAPER

by

Dipl.-Ing. Thomas Hütter

Supervisor:	Univ.-Prof. Nikolaus Augsten, PhD (University of Salzburg)
Host-Supervisor:	Prof. Michael J Carey, PhD (University of California, Irvine)

Salzburg, September 16, 2020

Department of Computer Sciences • Faculty of Natural Sciences
Paris Lodron University of Salzburg
Jakob-Haringer-Straße 2 • A-5020 Salzburg

Abstract

In the last decade, JSON became one of the most popular data formats in computer sciences, especially since the rise of document stores. Commonly, these database systems natively store data in a JSON-like format to address the need for schemaless data. While these systems evolved into full-fledged applications, an unaddressed problem is the similarity of their data items, hence JSON documents. Existing databases only support similarity queries for individual values, e.g., strings or sets, rather than for the entire data record.

To this end, we define an intuitive edit-based distance measure for JSON documents, called JSON edit distance (JED). Unfortunately, computing JED is NP-complete as proven in this paper. Therefore, we present JED approximations with lower and upper bound guarantees that can be computed in polynomial time. In an experimental evaluation on several real-world datasets, we compare and analyze the runtime and tightness of these bounds. Finally, we present a prototype system, called AsterixDB, and discuss two approaches of how to integrate JED.

Contents

CHAPTER 1

Introduction

Nowadays, one of the most prominent data formats is the JavaScript Object Notation (JSON). It is used in a large variety of scenarios, e.g, a vast amount of real-world datasets are provided in JSON format or, being native to Javascript, JSON is the standard data interchange format in mobile and web applications. Another application of interest are non-relational database systems, in particular document stores, which use JSON-like formats to natively store semi-structured data.

Due to the schema-less approach of document stores, individual data records may not have similar structures or attributes. Consider a product database of an online shop. A certain product may be offered by multiple vendors and is stored several times in the database with minor differences, e.g., only some products have a description or provide compatible operating systems (cf. Figure 1.1). To avoid duplicates in the product catalog, similar records must be identified and similarity queries that allow differences in the data records are required.

```
{                              {
  "Trackpad" : {,                "Trackpad" : {,
    "model" : "X",                 "model" : "X",
    "size" : {                     "size" : {
      "Weight" : 8.6,                "Weight" : 8.6,
      "Height" : 7,                  "Height" : 7,
      "Depth" : 1.1                 }
     }                             "Layout" : "DE",
    "Layout" : "EN",               "Description" : "Input
    "OS" : ["OSX", "Win"]                         device."
  }                              }
}                              }
```

Figure 1.1: Two example JSON documents of products in an online shop.

Currently, most database systems provide only limited or no support for similarity queries. In order to compute the similarity of data records, stored as JSON documents, existing solutions divide the data into smaller junks of information, e.g., sets of tokens or strings of certain values. However, the overall structure and the hierarchically encoded information of JSON documents is neglected. While distance measures for other data formats (e.g., XML [14, 24, 22]) are well studied, the similarity of JSON documents is poorly addressed in literature and by database system vendors.

To the best of our knowledge we are the first to address edit-based distance measures for JSON documents. The main contributions described in this paper are:

- We introduce JED, an edit-based distance for JSON documents based on an intuitive tree representation.
- We give a detailed NP-completeness proof to show the computational complexity of JED.
- We present four JED approximation algorithms with lower and upper bound guarantees.
- We gathered a collection of 20 JSON datasets and analyzed their characteristics.
- In an experimental evaluation, we use the collected datasets to verify the quality and runtime of the proposed approximations.
- We analyze a prototype system, called AsterixDB, and discuss JED integration methods.

The remainder of the paper is structured as follows: Section 2 discusses important preliminary informations on JSON and similarity measures for trees. An edit distance for JSON documents (JED) and its NP-completeness proof is shown in Section 3. Next, we present efficient and effective approximation algorithms for JED in Section 4. The related work is discussed in Section 5. In Section 6, we analyze the characteristics of 20 real-world JSON datasets and experimentally evaluate our approximation algorithms. Methods on how to integrate JED within a big data management system can be found in Section 7. In Section 8, we summarize and give an outlook on future JED projects.

CHAPTER 2

Preliminaries

Before we head on to the definition of a novel JSON edit distance, we summarize essential information on the JSON data format, JSON trees, and edit-based tree distances.

2.1 JSON Format

JSON, as described in RFC8259 [9] is composed of the following three basic components:

- An *object* is an unordered list of key-value pairs that are surrounded by curly braces. Keys are string literals that are unique within a certain object. The definition of values can be found below.
- An *array* is an ordered list of JSON values surrounded by brackets.
- A *value* is either a literal (string, number, boolean, or null), an object, or an array. The recursive definition of values results in the hierarchical structure of JSON documents.

Two examples of JSON documents are given in Figure 1.1. Consider the document on the left hand side. The root node is an object that contains a key-value pair with key `"Trackpad"` and a nested object as a value; the value of key `"OS"` is a nested array that contains a list of two objects. In this example, only string and number literals are used.

2.2 JSON tree representation

Based on their recursive definition, JSON documents are naturally represented as trees. A *JSON tree* T is a directed, acyclic, connected graph with a set of nodes

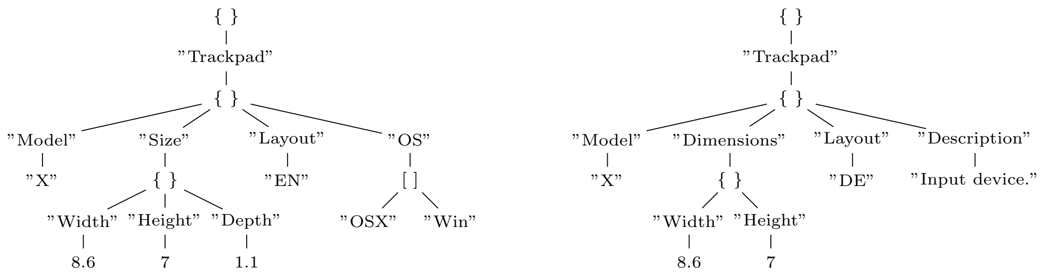

Figure 2.1: According JSON tree representations of the JSON documents shown in Figure 1.1.

$N(T)$, a set of edges $E(T) \subseteq N(T) \times N(T)$, and a relation $<_S$ that defines the sibling order.

Each node $v \in N(T)$ has a type, $type(v)$, and a label, $lbl(v)$. The type of a node represents a certain JSON component, i.e., an object, an array, a key, or a literal. Labels carry the data of a node, but are not necessarily unique. Note, that object and array nodes have an empty label. The size $|T|$ of tree T is the number of its nodes $|N(T)|$.

In an edge $(v, w) \in E(T)$, v is the parent and w the child, $p(w) = v$. Children of the same node are siblings. Each node v has at most one incoming edge. The node without a parent is the root node, a node without children is a leaf. The relationship between parent v and child w is limited by the following type conditions: (1) $type(v) = \text{object} \implies type(w) \in \{\text{key}\}$, (2) $type(v) = \text{array} \implies type(w) \in \{\text{object}, \text{array}, \text{literal}\}$, and (3) $type(v) = \text{key} \implies type(w) \in \{\text{object}, \text{array}, \text{literal}\}$. Since literals cannot have children, there is no edge $(v, w) \in E(T)$ with $type(v) = \text{literal}$. Node x is an ancestor of node v, $x \in anc(v)$, iff $x = p(v)$ or x is an ancestor of $p(v)$. x is a descendant of v, $x \in desc(v)$, iff v is an ancestor of x.

JSON documents are usually a hybrid consisting of ordered (values of arrays) and unordered siblings (key-value pairs of objects). The *sibling order* is defined by the pair $(N(T), <_S)$, a strict partially ordered set, with the following properties: (1) the order is total among the children of an array node, and (2) for any distinct nodes $x, y, x', y' \in V$, if $x \in desc(x')$, $y \in desc(y')$ and $x' < y'$, then $x < y$.

An additional constraint, introduced by the definition of JSON, is that the labels among all children of an object node, namely keys, must be unique.

Figure 2.1 shows the tree representations of the JSON documents in Figure 1.1. In this visualization, object nodes are depicted as { }, array nodes as *[ ]*, and key and literal nodes with their original values.

2.3 Tree Edit Distance

After defining a tree representation of JSON documents, we will discuss a well established distance measure for tree-structured data. The so-called *tree edit distance*, $\delta(T, T')$, between two trees T and T', which is defined as the minimum number of node edit operations that transform T into T'. Allowable node edit operations are: *delete* node v and connect all its children to the parent of v maintaining the sibling order; *insert* a new node w between an existing node v and a consecutive subset or subsequence of v's children, *rename* the label of node v.

The state-of-the-art algorithm by Pawlik and Augsten [26] computes the tree edit distance between two ordered trees in $O(n^3)$ while the problem was shown to be MaxSNP-hard for unordered trees by Zhang [38]. JSON trees may include ordered (array) and unordered (object) siblings.

2.4 Edit Mapping

An *edit mapping*, M, between the nodes of two trees T and T', represents the edit operations that transform T into T'.

Node pairs that are mapped represent rename operations, unmapped nodes in T are deleted, and unmapped nodes in T' are inserted. Considering the unit-cost model, a mapped node pair with identical labels is renamed at zero cost, all other operations have cost 1. The cost of an edit mapping, $cost(M)$, is the cost sum of all its operations.

The edit mapping is defined based on restrictions on the node mappings. Note that Definition 2.1 lists the conditions for ordered trees. The edit mapping between unordered trees neglects the order condition (cf. Definition 2.2).

Definition 2.1 (Edit mapping for ordered trees).
Given two trees T and T' and a mapping $M \subseteq N(T) \times N(T')$, M is an edit mapping from T to T' if and only if the following conditions hold for any two node pairs $(v, w), (v', w') \in M$:

- $v = v'$ iff $w = w'$ (one-to-one),
- v is an ancestor of v' iff w is an ancestor of w' (ancestor),
- v is to the left of v' iff w is to the left of w' (order).

Definition 2.2 (Edit mapping for unordered trees).
Given two trees T and T' and a mapping $M \subseteq N(T) \times N(T')$, M is an edit mapping from T to T' if and only if the following conditions hold for any two node pairs $(v, w), (v', w') \in M$:

- $v = v'$ iff $w = w'$ (one-to-one),
- v is an ancestor of v' iff w is an ancestor of w' (ancestor).

In the following section, we will similarly define an edit mapping for JSON trees.

CHAPTER 3

JSON Edit Distance (JED)

We present a novel edit-based distance measure for JSON documents, called JSON Edit Distance (JED). JED operates on the tree representations of JSON documents as described in Section 2.2. The distance, denoted $JED(T_J, T'_J)$, is defined as the minimum-cost sequence of node edit operations that transform JSON tree T_J into T'_J.

Conceptually, JED is similarly defined as the widely adopted and accepted string and tree edit distance. The core of the distance are three edit operations on the nodes of a JSON tree: delete a node, insert a node , and rename the label of a node. Note that only the label of a node can be renamed, not its type. Hence, objects and arrays are not affected by rename operations. In Definition 3.1, we define the edit mapping for JED.

Definition 3.1 (JSON Edit Mapping).
Given two JSON trees T and T' and a mapping $M \subseteq N(T) \times N(T')$, M is a JSON edit mapping from T to T' if and only if the following conditions hold for any node pairs $(v, w), (v', w') \in M$:

- $v = v'$ iff $w = w'$ (one-to-one),
- v is an ancestor of v' iff w is an ancestor of w' (ancestor),
- $v <_S v'$ and w is comparable to w' in $<_S$ then $w <_S w'$ (array-order),
- $type(v) = type(w)$ (type).

The JSON edit mapping extends the edit mapping of unordered trees by two conditions, array-order and type. First, JSON trees are a hybrid containing unordered children for objects and ordered children for arrays. We leverage the partial order $<_S$ to express the order that is imposed by arrays. Second, only nodes of the same type are allowed to be mapped, to introduce more intuitive mappings.

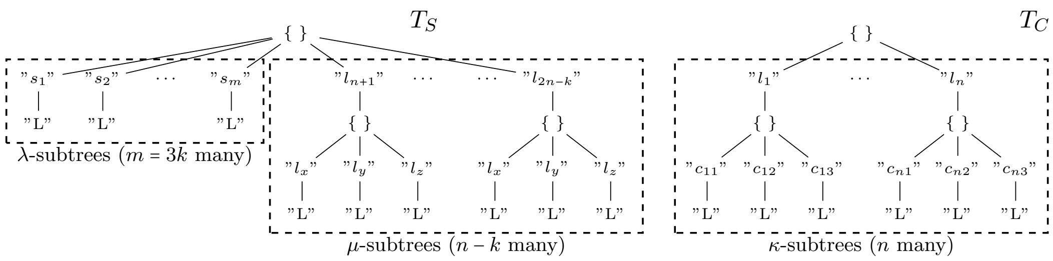
T_S
T_C
{ }
"s_1" "s_2" ... "s_m"
"L"
λ-subtrees ($m = 3k$ many)
"l_{n+1}" ... ... "l_{2n-k}"
"l_x" "l_y" "l_z"
μ-subtrees ($n - k$ many)
"l_1" ... "l_n"
"c_{11}" "c_{12}" "c_{13}" "c_{n1}" "c_{n2}" "c_{n3}"
κ-subtrees (n many)

Figure 3.1: Polynomial time transformation of an exact cover by 3-sets (X3C) instance into JSON trees T_S and T_C.

The JSON edit mapping for the two JSON trees in Figure 2.1 is as follows: the two object nodes and key `"Trackpad"` starting at the root of the tree are mapped, the subtrees of key `"Model"` are mapped, the subtrees of key `"Layout"` are mapped with cost 1, `"Size"` is renamed to `"Dimensions"`, the key-value pair `"Depth" : 1.1` is deleted. In the remaining subtree `"OS"` is renamed to `"Description"`, `"OSX` is renamed to `"Input device."`, and the array node and `"Win"` is deleted.

3.1 NP-completeness of the JSON Edit Distance

In order to show that the computation of the JSON edit distance is NP-complete, we consider the following decision problem called JED:

Definition 3.2 (JED).
Instance: Two JSON trees T_S and T_C and a positive integer i.
Question: Is the minimal JSON edit distance $JED(T_S, T_C) \le i$?

Similar to the unordered tree edit distance [35], the proof is by reducing the *exact cover by 3-sets (X3C)* problem, c.f. Definition 3.3, to *JED*. The NP-completeness proof of *X3C* was published by Garey and Johnson [18].

Definition 3.3 (Exact cover by 3-sets (X3C)).
Instance: A finite set $S = \{s_1, s_2, \ldots, s_m\}$, where $m = 3k$, and a collection $C = \{C_1, C_2, \ldots, C_n\}$, where $C_i = \{c_{i_1}, c_{i_2}, c_{i_3}\}$ and $c_{i_j} \in S$.
Question: Is there a subcollection $C' \subseteq C$ such that every element of S occurs in exactly one member of C'?

Given an instance of *X3C*, let $l_1, \ldots, l_{2n-k}, l_x, l_y, l_z, L \notin S$ be strings that are not elements of set S and, w.l.o.g, each element $s_i \in S$ occurs only once in S. The polynomial time transformation of an *X3C* instance into a *JED* instance (i.e., JSON trees T_S, T_C, and integer i) is shown in Figure 3.1. Tree T_S is built based on set S; for each element $s_i \in S$, a λ-subtree is inserted ($3k$ many), and additionally $n - k$ μ-subtrees are inserted. Tree T_C is built based on collection C; for each element $C_i \in C$, a κ-subtree is inserted (n many). Further, let $i = 4n - 2k$.

Given an instance of *X3C* with JSON tree transformations T_C and T_S, we show that the following holds: $JED(T_S, T_C) \leq 4n - 2k$ iff there is an exact cover by 3-sets.

Lemma 3.4.
Given an instance of *X3C* with JSON tree transformations T_S and T_C, let M_{JED} be the minimal JSON edit mapping between JSON trees T_S and T_C, then $JED(T_S, T_C) \geq 4n - 2k$.

Proof.
As a result of the described transformation process, the sizes of the trees $|T_C|$, $|T_S|$ and the mapping size $|M_{JED}|$ are as follows:

- $|T_S| = 8(n-k) + 2(3k) + 1$: $n-k$ μ-subtrees of size 8 are inserted, a λ-subtree with 2 nodes is inserted (key and literal) for each $s_i \in S$ with $|S| = 3k$, and one node for the root.
- $|T_C| = 8n+1$: for each $C_i \in C$ with $|C| = n$, a κ-subtree with 8 nodes is inserted, and one node for the root.
- $|M_{JED}| = |T_S| - d = 8n - 2k + 1 - d$, where d is the number of unmapped nodes in T_S. From the JSON tree transformation, we know that $|T_C| \geq |T_S|$. Then, the mapping size is the size of the smaller tree $|T_S|$ subtracted by the number of unmapped nodes in T_S.

Further, there are at most $4(n-k) + 2(3k) + 1$ nodes with identical labels. 1 results from the identical object in the root nodes, $s_1, \ldots, s_m$ and their child literals L may be identical to c_{i_j} and their child literals, and all other subtrees ($n-k$ many) share one object node and three literals L.

The cost γ of the minimal JSON edit mapping M_{JED} between T_S and T_C is bounded as follows:

$$\begin{aligned}
\gamma(M_{JED}) &\geq (|M_{JED}| - (4(n-k) + 2(3k) + 1)) \\
&\qquad + (|T_C| - |M_{JED}|) + (|T_S| - |M_{JED}|) \\
\gamma(M_{JED}) &\geq 4n - 4k - d + (|T_C| - |M_{JED}|) + (|T_S| - |M_{JED}|) \\
\gamma(M_{JED}) &\geq 4n - 2k + (|T_S| - |M_{JED}|) \\
\gamma(M_{JED}) &\geq 4n - 2k + d \\
\gamma(M_{JED}) &\geq 4n - 2k
\end{aligned}$$

Note that the first term is an upper bound on the number of rename operations, the second term is the number deletion, and the third term is the number of insertions.

Since the minimal JSON edit mapping is a visual representation of the node edit operations, we know that $\gamma(M_{JED}) = JED(T_S, T_C)$. Hence, $JED(T_S, T_C) \geq 4n - 2k$. ■

Lemma 3.5.
Given an instance of *X3C* with JSON tree transformations T_S and T_C, the following holds: if there is an exact cover by 3-sets, then $JED(T_S, T_C) \leq 4n - 2k$.

Proof.
Create a JSON edit mapping M^* as follows:

- Map nodes $s_1, s_2, \dots, s_m$ in T_S to a c_{i_j} in T_C with the same label. There must be zero-cost mappings for all $s_i \in S$ as well as their literal children L, since there exists an exact cover by 3-sets. Hence, for each triplet of λ-subtrees (k many), 2 nodes have to be inserted (l_j and an object node). The resulting costs are $2k$.
- Map the remaining μ-subtrees ($n - k$ many) in T_S to the remaining κ-subtrees in T_C with cost 4.

The cost of the resulting mapping is $\gamma(M^*) = 2k + 4(n - k) = 4n - 2k$. Since M^* is a valid but not necessarily minimal JSON edit mapping, $\gamma(M^*) \geq JED(T_S, T_C)$. ■

Lemma 3.6.
Given an instance of *X3C* with JSON tree transformations T_S and T_C, the following holds: if $JED(T_S, T_C) \leq 4n - 2k$, then there is an exact cover by 3-sets.

Proof.
From $JED(T_S, T_C) \leq 4n - 2k$ and Lemma 3.4, we know that $JED(T_S, T_C) = 4n - 2k$. Further, from Lemma 3.4, we know that $d = 0$, hence all nodes in T_S are in the minimal JSON edit mapping M_{JED}. We continue by analyzing which nodes pairs of T_S and T_C are in M_{JED}.

First, the roots of T_S has to be mapped to the root of T_C. Otherwise, the root can only be mapped to an object in a κ-subtree. Due to the ancestor condition, all nodes in T_S have to be mapped to the subtree of the according object node in T_C. Since there are more nodes in T_S than in such a subtree, it contradicts the fact that all nodes in T_S are mapped.

Next, consider the root node of any μ-subtree. Due to the type constraint of the JSON edit mapping, it can only be mapped to the root of a κ-subtree or to a c_{ij}. Mapping it to the root implies a minimal cost of 4 rename operations compared to at least 6 deletion operators otherwise. Therefore, in a minimal mapping, each μ-subtree is mapped to a κ-subtree with cost 4, which results in an overall cost of $4(n - k)$.

Hence, only $2k$ edit operations remain to map $3k$ λ-subtrees to k κ-subtrees. The size difference between these remaining subtrees in T_C and T_S is $8k - 6k = 2k$. Hence, all nodes in T_S must be mapped with cost 0, i.e., all $s_i \in S$ must match exactly one c_{ij} in k κ-subtrees in T_C, which is equivalent to C' and , therefore, there is exists an exact cover by 3-sets.

■

Theorem 3.7.
Computing the JSON edit distance between two JSON trees is NP-complete.

Proof.
We need to show the following two steps:

- $JED \in NP$: A non-deterministic algorithm can guess a JSON edit mapping $M^* \subseteq N(T_S) \times N(T_C)$ between two JSON trees T_S and T_C and verify in polynomial time whether the mapping costs, hence the number of rename, insertion, and deletion operations, are at most i.
- X3C can be reduced to JED: From Lemma 3.5 and Lemma 3.6 immediately follows that there is an exact cover by 3-sets iff $JED(T_C, T_S) \leq 4n - 2k$.

Hence, the computation of the JSON edit distance is NP-complete. ■

While the above proof only holds for JSON trees with degree 3, the proof for trees with degree $deg = q > 3$ results from an according reduction of exact cover by q-sets. In this general case, the claim is as follows: there exists an exact cover by q-sets iff $JED(T_S, T_C) \leq (q+1)n + (q-5)k$.

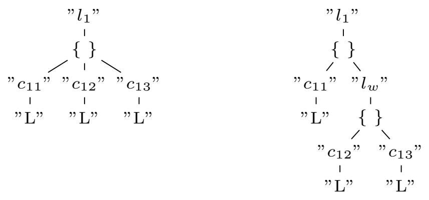

Figure 3.2: Transformation of κ- and μ-subtrees into a JSON tree with degree 2, $l_w \notin S$.

Since exact cover by 2-sets is solvable in polynomial time, we reduce exact cover by 3-sets to show the NP-completeness of trees with degree two. The following two modifications are needed: (1) compared to Figure 3.1, adjust the structures of κ- and μ-subtrees as shown in Figure 3.2, (2) the sizes change as follows $|T_C| = 10n + 1$, $|T_S| = 10(n-k)+2(3k)+1$, $|M| = 10n-4k+1-d$, and there are at most $5(n-k)+2(3k)+1$ nodes with identical labels. Then it holds that $JED(T_C, T_S) \leq 5n - 3k$ iff there is an exact 3-cover. Hence, the JSON edit distance is NP-complete for JSON trees with degree 2 or higher.

Note, that JED for degree 1 trees can be computed using the string edit distance where each node of the tree is considered as a single character in the string. This observations follows from the ancestor condition of JED.

CHAPTER 4

JED Approximations

Computing the JSON edit distance is not feasible, even for small documents, as proven in the previous section. In this section, we present well-defined approximations of JED that are computable in polynomial time. Further, we proof that each approximation is either a lower or an upper bound to JED.

4.1 Node Intersection (NI)

A basic JED approximation is the so-called node intersection. Intuitively, the hierarchical information is disregarded and nodes are mapped with cost 0 if they have the same type and label. Otherwise, it is considered that the remaining nodes can be renamed with cost 1. Note, that we consider the unit cost model for all of the following examples, i.e., each insertion, deletion, and rename have cost 1.

Theorem 4.1 (Node Intersection Lower Bound).
Given two JSON trees T_J and T'_J, let N and N' be the bags of nodes of T_J and T'_J, and $|N \Cap N'|$ be the bag intersection, respectively.

$$JED(T_J, T'_J) \geq max(|N|, |N'|) - |N \Cap N'|$$

Proof.
The right-hand side of the equation is equivalent to the number of nodes in the larger tree that cannot be mapped to a node in the smaller tree with zero cost (i.e., different type and/or label). Consequently, for all of these nodes exactly one edit operation is imposed. All other nodes are considered to be mapped with cost 0, even though they may not be in the minimal JSON edit mapping. The overestimation of 0 cost mappings lead to an overall lower bound. ■

Consider JSON trees T_J and T'_J in Figure 2.1, the node intersection is $19 - 11 = 8$. As shown in Section 3, the computed value is equivalent to JED in this example.

4.2 Ordered JSON Edit Distance (OJED)

The first out of three upper bounds for JED that we present in this paper adapts the edit mapping definition for ordered trees (cf. Definition 2.1) to JSON trees which results in Definition 4.2. The main difference is the additional JSON type constraint (type).

Definition 4.2 (Ordered JSON Edit Mapping).
Given two JSON trees T_J and T'_J and a mapping $M_{OJED} \subseteq N(T_J) \times N(T'_J)$, M_{OJED} is an edit mapping from T_J to T'_J if and only if the following conditions hold for any two node pairs $(v, w), (v', w') \in M_{OJED}$:

- $v = v'$ iff $w = w'$ (one-to-one),
- v is an ancestor of v' iff w is an ancestor of w' (ancestor),
- v is to the left of v' iff w is to the left of w' (order),
- $type(v) = type(w)$ (type).

The resulting distance, called OJED, is equivalent to the minimum cost ordered JSON edit mapping. OJED considers an order among all siblings of a JSON tree, including unordered key-value pairs of object nodes. As a consequence, the considered order may forbid mappings that are part of the minimal JED mapping. However, OJED provides an upper bound to JED as shown in Theorem 4.3.

Theorem 4.3 (OJED Upper Bound).
Given two JSON trees T_J and T'_J, then

$$OJED(T_J, T'_J) \geq JED(T_J, T'_J).$$

Proof.
The only constraint from OJED (cf. Definition 4.2) that differs from JED (cf. Definition 3.1) is the order constraint. However, a total order on all nodes of a tree is stricter than the partial order on array nodes, hence, each ordered JSON edit mapping is a JSON edit mapping and, consequently, OJED is an upper bound of JED. ■

The imposed order between unordered siblings (keys) can highly influence the distance between two JSON trees. A good heuristic for real-world datasets is to sort the keys of object nodes alphabetically by key labels. Note that this results in a consistent ordering since keys are unique among all sibling of a JSON object.

In the example in Figure 2.1, $OJED(T_J, T'_J)$ is 14. Due to the imposed order, the subtree of key `"Layout"` is mapped to the subtree of key `"Description"` with two rename operations. The subtrees of keys `"Model"` and `"OS"` in the left hand tree are deleted, and the subtrees of keys `"Layout"` and `"Model"` in the right hand tree are inserted. The subtree with key `"Size"` is mapped to the subtree of key `"Dimensions"` using one rename and two delete operations.

OJED can be computed by adjusting the cost-model (mapping cost of infinity for nodes of different type) of any existing ordered tree edit distance algorithm. In our experiments (cf. Section 6), we have implemented the state-of-the-art algorithm by Pawlik and Augsten [26] providing a runtime complexity of $O(n^3)$ and space complexity of $O(n^2)$ where n is the size of the larger tree.

4.3 Document Preserving JSON Edit Distance (DPJED)

OJED decreases the algorithmic complexity by neglecting the unordered fashion of key-value pairs. In order to consider unordered siblings and keep a polynomial runtime and space complexity, further constraints are needed.

In JSON trees, each subtree (starting at an array or object node) is a nested JSON document itself. A natural constraint, called document preserving, for JSON trees allows only individual subtrees, hence JSON documents, to be mapped to one another. Adding the described constraint to the JSON edit mapping defines an another approximation of JED (cf. Definition 4.4).

Definition 4.4 (Document Preserving JSON Edit Mapping).
Given two JSON trees T_J and T'_J and a mapping $M_{DPJED} \subseteq N(T_J) \times N(T'_J)$, M_{DPJED} is an edit mapping from T_J to T'_J if and only if the following conditions hold for any two node pairs (v, w), (v', w'), $(v'', w'') \in M_{DPJED}$:

- $v = v'$ iff $w = w'$ (one-to-one),
- v is an ancestor of v' iff w is an ancestor of w' (ancestor),
- $v <_S v'$ and w is comparable to w' in $<_S$ then $w <_S w'$ (array-order),
- $type(v) = type(w)$ (type),
- let $lca(v, v') = V$ and $lca(w, w') = W$; V is proper ancestor of v'' iff W is proper ancestor of w'' (document preserving).

Like OJED, the resulting distance DPJED, is equivalent to the minimum cost document preserving JSON edit mapping. DPJED imposes additional restrictions on the JSON edit mapping and, hence, is an upper bound of JED as shown in Theorem 4.5.

Theorem 4.5 (DPJED Upper Bound).
Given two JSON trees T_J and T'_J, then

$$DPJED(T_J, T'_J) \geq JED(T_J, T'_J).$$

Proof.
Assume that $DPJED(T_J, T'_J) < JED(T_J, T'_J)$, then there exists a DPJED edit mapping M_{DPJED} with cost $c(M_{DPJED}) = DPJED(T_J, T'_J)$ and a JED mapping M_{JED} with cost $c(M_{JED}) = JED(T_J, T'_J)$. Since M_{JED} is (by definition) also a DPJED edit mapping, the cost of M_{JED} is not minimal, which contradicts $c(M_{JED}) = JED(T_J, T'_J)$. ■

The constraints introduced by OJED and DPJED are independent from each other. Therefore, there is no upper or lower bound relation between the two edit distances.

In the example in Figure 2.1, $DPJED(T_J, T'_J)$ is 8 and hence less than the ordered JSON edit distance. DPJED finds the minimum cost mapping between the subtrees of unordered sibling pairs, i.e., `"Model"` to `"Model"`, `"Size"` to `"Dimensions"`, `"Layout"` to `"Layout"`, and `"OS"` to `"Description"`.

In order to compute DPJED, we combined the algorithms for the ordered constraint tree edit distance [36] and the unordered constraint tree edit distance [37]. Next to adjusting the cost-model, we needed to adjust the method of finding the minimum mapping between siblings. For arrays, the order must be considered; therefore, the string edit distance between the ordered sequences of siblings is used. For objects, the best combination of sibling pairs need to be identified using a bipartite graph matching approach, e.g., with the Hungarian Algorithm. Keys only have one child and literals are leave nodes, hence, there children can be mapped trivially.

4.4 Ordered Document Preserving JSON Edit Distance (ODPJED)

The third approach combines the constraints of OJED and DPJED, i.e., all siblings are considered to be ordered and only distinct subtrees can be mapped to one another. The edit mapping of the resulting distance measure, called ODPJED, is shown in Definition 4.6.

Definition 4.6 (Ordered Document Preserving JSON Edit Mapping). Given two JSON trees T_J and T'_J and a mapping $M_{ODPJED} \subseteq N(T_J) \times N(T'_J)$, M_{ODPJED} is an edit mapping from T_J to T'_J if and only if the following conditions hold for any two node pairs (v, w), (v', w'), $(v'', w'') \in M_{ODPJED}$:

- $v = v'$ iff $w = w'$ (one-to-one),
- v is an ancestor of v' iff w is an ancestor of w' (ancestor),
- v is to the left of v' iff w is to the left of w' (order),
- $type(v) = type(w)$ (type),
- let $lca(v, v') = V$ and $lca(w, w') = W$; V is proper ancestor of v'' iff W is proper ancestor of w'' (document preserving).

Like OJED and DPJED, the resulting distance ODPJED, is equivalent to the minimum cost ordered document preserving JSON edit mapping. In Theorem 4.7 and Theorem 4.8, it is shown that ODPJED is an upper bound for OJED as well as DPJED and consequently for JED.

Theorem 4.7 (ODPJED is an upper bound for OJED). Given two JSON trees T_J and T'_J, then

$$ODPJED(T_J, T'_J) \geq OJED(T_J, T'_J).$$

Proof.
Assume that $ODPJED(T_J, T'_J) < OJED(T_J, T'_J)$, then there exists a ODPJED edit mapping M_{ODPJED} with cost $c(M_{ODPJED}) = ODPJED(T_J, T'_J)$ and a OJED mapping M_{OJED} with cost $c(M_{OJED}) = OJED(T_J, T'_J)$. Since M_{OJED} is (by definition) also a ODPJED edit mapping, the cost of M_{OJED} is not minimal, which contradicts $c(M_{OJED}) = OJED(T_J, T'_J)$. ∎

Theorem 4.8 (ODPJED is an upper bound for DPJED).
Given two JSON trees T_J and T'_J, then

$$ODPJED(T_J, T'_J) \geq DPJED(T_J, T'_J).$$

Proof.
Assume that $ODPJED(T_J, T'_J) < DPJED(T_J, T'_J)$, then there exists a ODPJED edit mapping M_{ODPJED} with cost $c(M_{ODPJED}) = ODPJED(T_J, T'_J)$ and a DPJED mapping M_{DPJED} with cost $c(M_{DPJED}) = DPJED(T_J, T'_J)$. Since M_{DPJED} is (by definition) also a ODPJED edit mapping, the cost of M_{DPJED} is not minimal, which contradicts $c(M_{DPJED}) = DPJED(T_J, T'_J)$. ∎

Theorem 4.9 (ODPJED is an upper bound for JED).
Given two JSON trees T_J and T'_J, then

$$ODPJED(T_J, T'_J) \geq JED(T_J, T'_J).$$

Proof.
Immediately follows from Theorem 4.7 or Theorem 4.8. ∎

Similar to the experiments on real-world datasets in Section 6, $ODPJED(T_J, T'_J) = 14$ is equivalent to $OJED(T_J, T'_J)$. In order to compute ODPJED, we use the algorithm for the ordered constraint tree edit distance [36]. Like for OJED and DPJED, the cost-model needs to be adapted to guarantee the JSON type constraint.

CHAPTER 5

Related Work

5.1 JSON Schema

Most of the scientific publications related to the JSON data format deal with schema extraction/inference algorithms. An extracted schema may be used as a description of the dataset or enables schema-based optimization techniques on JSON data. In a recent study by Baazizi et al. [5], a parametric and parallelizable schema inference algorithm was introduced. Klettke et al. [23] presented a schema extraction algorithm to identify structural outliers in large datasets. Further algorithms have been published by Frozza et al. [15] and Spoth et al. [30].

5.2 JSON Diffs

The only work on JSON diffs was published by Cao et al. [12]. In their study they presented an algorithm thats computes a JSON patch based on the edit operations defined in RFC6902 [10]. Even though an experimental study and a comparison to four open source solutions was performed, the runtime and space complexity of the presented algorithm was not discussed. Further, the resulting patch is not minimal. Yahia et al. [33] proposed a YAML based, language for describing change detection strategies on JSON data.

There is a larger number of diff algorithms for other hierarchical data formats. Assuming application domain-specific properties, Chawathe et al. [13] presented an algorithm that computes minimal diffs for Latex as well as HTML documents. The diff consists of the following edit operations: insert a leaf node, delete a leaf node, update the value of any node and move a subtree. The diff between XML documents introduced by Cobena et al. [14] considers insertions of subtrees, deletions of subtrees, value updates of any node, and move a node or a part of a subtree. The diff is computed bottom-up and in linear time. Another algorithm on XML diffs were published by Leonardi and Bhowmick [24] using relational databases. Here, node insertion, node deletion, and leaf update are considered.

While all previously mentioned algorithms compute the diff based on tree-matching, Jang et al. [22] propose a stream-based method using a D-Path algorithm.

5.3 JSON Tree Representations

Different approaches for tree representations of JSON objects have been published recently. Bourhis et al. [8] represent keys and the array order as edges and values as leaves nodes; however, object and array informations is not explicitly encoded in the tree. The approach by Shukla et al. [29] is similar, but keys and the array order are inner nodes instead of edges. In the work of Klettke et al. [23], there are three different types of nodes (object, array, property) additional to the label. Spoth et al. [30] state that the tree contains atomic values at the leaves and complex values in the inner nodes.

Further, tree representations of XML data may be applied in the context of JSON. For example, Augsten et al. [4] use keys as inner nodes and key-value pairs as leaf nodes.

5.4 Tree Edit Distance

A well-known edit distance for trees is the tree edit distance (TED). As pointed out in Section 2.3, there exist algorithms for ordered and unordered trees. The state-of-the-art algorithm for ordered trees by Pawlik and Augsten [26] computes TED in cubic time using quadratic memory. Computing the exact tree edit distance for unordered trees is NP-hard, as shown by Zhang et al. [39]. Typical approaches for polynomial time algorithms are the consideration of certain tree classes, parameterized algorithms or restricted edit operations.

Akutsu et al. [1] considers only special classes of trees, namely stars, caterpillars, and moths. The presented parameterized algorithm runs in $O(2^{(b_1+b_2)}|T_1||T_2|\Delta)$ time, where $|T_1|, |T_2|$ and $|b_1|, |b_2|$ are the sizes of the input trees and their number of branching nodes and Δ is maximum number of children in both trees. Akutsu et al. [2] presented another parameterized algorithm that runs in $O(2.62^k \cdot poly(|T_1|, |T_2|))$, where k is an upper bound on the unordered tree edit distance. Shasha et al. [28] presented an exact algorithm that runs in $O(4^{|l_1|}4^{|l_2|}min(l_1, l_2) \cdot |T_1||T_2|)$, where $|l_1|, |l_2|$ are the number of leaf nodes.

Given an upperbound k and the edit operations node insertion and leaf deletion, Fukagawa and Akutsu [16] presented an algorithm that runs in $O(4^k n)$.

Fukagawa et al. [17] transformed the unordered tree edit computation in a maximum clique problem and applied existing solvers. In their experimental evaluation, they compared their algorithm to an A^* approach by Horesh et al. [21]. Another A^* algorithm was published by Higuchi et al. [20]. In order to improve their solution, they apply lowerbounds, namely tree size difference, label histogram distance, and degree histogram distance.

5.5 Unordered Tree Edit Distance Approximations

Augsten et al. [4] introduce an approximation for the unordered tree edit distance by tree decomposition, called windowed pq-grams. This approach splits a tree into a set of smaller elements which are then compared to the decomposition of another tree. In their experimental evaluation, they show that windowed pq-grams outperform other tree decomposition algorithms, namely binary branches [34], path shingles [11], and valid subtrees [19]).

Torsello and Hancock. [31] present an algorithm that transforms the unordered tree edit distance computation into a series of maximum weighted clique problems and then finds an approximation using relaxation labeling.

A different concept was used by Shasha et al. [28] which is based on hill climbing and bipartite graph matching.

5.6 Further edit-based tree distances

Due to the computational complexity of the tree edit distance between unordered trees, other distance measures have been proposed in literature that impose further restrictions the edit mapping.

One of these distances for unordered trees was introduced by Zhang [37], the so-called constrained edit distance (or isolated-subtree distance). Here, disjoint subtrees can only be mapped to disjoint subtrees. This distance can be computed in $O(|T_1| \cdot |T_2| \cdot (deg(T_1) + deg(T_2)) \cdot log_2(deg(T_1) + deg(T_2)))$ for unordered trees.

Distances for unordered trees that can be computed in linear time are the top-down by [27] and bottom-up distance [32]. Intuitively, two nodes can be mapped, if their parents (resp. all children) are mapped.

CHAPTER 6

Experiments

In this section, we experimentally evaluate the introduced JED upper bounds in terms of runtime and tightness. Therefore, we implemented all approximation algorithms in a unified C++ framework. The experiments are executed single-threaded on an Intel Xeon E5-2630 v3 2.40GHz server with 8 cores and 96GB of RAM, running Debian 4.19.0. The experimental setup is as follows:

- A dataset is a collection of multiple JSON documents. These documents are transformed into according JSON trees which serves as input for our algorithms. To this end, we gathered 20 real-world JSON datasets (cf. Section 6.1).
- We compare the upper bounds OJED, DPJED, and ODPJED in terms of runtime and tightness by performing a pair-wise distance computation of all neighboring JSON trees in a collection, i.e., given a dataset D, compute $ub(T_{J_1}, T_{J_2})$, $ub(T_{J_2}, T_{J_3})$, $\ldots$, $ub(T_{J_{|D|-1}}, T_{J_{|D|}})$, where $ub \in \{OJED, DPJED, ODPJED\}$.

6.1 Real-World JSON Dataset Analysis

We collected a large amount of real-world JSON datasets to verify the quality of the proposed upper bound algorithms in realistic scenarios. In an extensive analysis, we identified insightful characteristics of JSON datasets as shown in Table 6.1. We highlight the most important characteristics in the following paragraphs.

Dataset sizes (#records): We define the size of a datasets as the number of individual JSON documents/trees it contains. In our gathered collection, the sizes vary from 25 up to almost 9 million JSON documents.

Document sizes (#nodes/rec): The size of a document within a dataset is defined as the number of nodes of its according JSON tree. In our gathered collection of datasets, the maximum tree size is around 18k nodes. However, for most of the datasets the tree sizes are less than 1000 nodes. Another interesting property is the size difference within a dataset. For example, the smallest tree in the DBLP dataset has size 14 while the largest one has 651. As a consequence, the size difference between two tree can be used as an efficient and effective JED lower bound.

Type distributions (#objects, #arrays, #keys, #literals): A JSON-specific characteristic is the type distribution of the nodes of a JSON tree. Relative to the number of nodes in a tree, there are at most 20% object nodes, 10% array nodes, 49% key nodes, and 49% literal nodes. This is of special interest since there are similarity algorithms with a complexity that depends on the number of leaves (literal nodes). E.g., the algorithm by Zhang and Shasha [35] can be used to compute OJED in $O(|T_J||T'_J|L_{T_J}L_{T'_J})$, where L_{T_J} and $L_{T'_J}$ denote the number of literal nodes of JSON trees T_J and T'_J.

Degree (object/array degree): The degree of a JSON tree is equivalent to the maximum number of key-value pairs (resp. elements) among all objects (resp. arrays) in a certain document. In our dataset collection, the average degree of both, objects and arrays, is typically less than 20 except for the Device dataset. This characteristic is of special interest since the complexity of DPJED depends on the degree of the JSON tree, i.e., $O(|T_J||T'_J|(deg(T_J) + deg(T'_J))log_2(deg(T_J) + deg(T'_J)))$.

Tree depth (max. depth): The depth of a JSON tree can be seen as the number of nested values within a JSON document. As a result of our analysis, none of our datasets is deeply nested. The maximum depth of 8 can be found in the dataset Twitter 2. Similar to the number of leave nodes, there are algorithms with a complexity that depends on the tree depth, e.g., the algorithm by Zhang and Shasha [35] can be used to compute OJED in $O(|T_J||T'_J|D_{T_J}D_{T'_J})$, where D_{T_J} and $D_{T'_J}$ denote the depth of JSON trees T_J and T'_J.

6.2 Tightness Analysis

In this section, we address the question of how tight the three upper bounds are to JED. Due to the computational complexity of JED, the actual distances between the tree pairs in a dataset are unknown and hence a comparison is not possible.

Given a lower bound lb of JED, we leverage the fact that $ub(T_J, T'_J) = JED(T_J, T'_J)$ if $ub(T_J, T'_J) = lb(T_J, T'_J)$, where ub is an upper bound. As a result, we get a lower bound of the relative number of tree pairs where an upper bound computes the same distance as JED. In our experiments, we use the node intersection lower bound as presented in Section 4.1.

Figure 6.1 shows the experimental results for 14 datasets (x-axis) which are described in Section 6.1. The bars show the relative number of tree pairs within a dataset where the node intersection is equivalent to the upper bound, i.e., OJED, DPJED, or ODPJED.

Overall, the presented bounds perform well on most of the analyzed datasets. For 10 out of 14 datasets, the exact JED distance can be computed in over 75% of the cases. For the datasets *Device* and *Reddit*, the upper bounds are equivalent

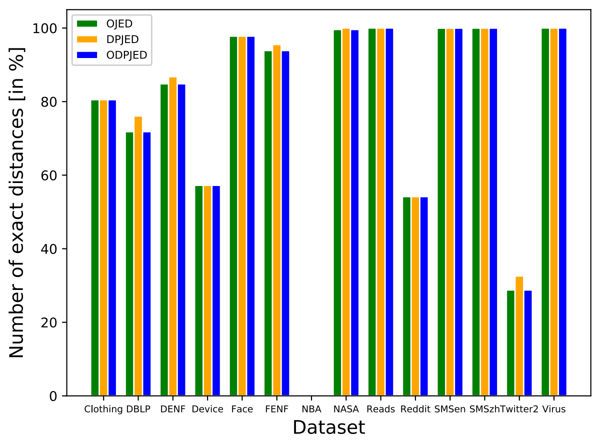

Figure 6.1: Lower bound on the relative number of tree pairs, for which the upper bounds OJED, DPJED, and ODPJED compute the exact distance within a dataset.

to JED for approximately 60% of all considered JSON tree pairs. In the *Twitter 2* dataset, the exact JED distance is computed in 30% of the cases.

In contrast to the other datasets, all bounds perform poorly at the *NBA* dataset, i.e., there is no tree pair in which the lower equals the upper bound. A more fine-grained analysis of this behavior can be seen in Figure 6.3. The y-axis shows the distances computed by the bounds for all considered JSON tree pairs ordered ascending by OJED values. While OJED, DPJED, and ODPJED are rather similar and vary only for some tree pairs, the values of the lower bound are never close to any of the upper bounds. A detailed analysis of the dataset revealed that the trees share a lot of common literals, however, at different positions in the tree. This is a worst case scenario for the node intersection since the hierarchical information is not considered and nodes are mapped incorrectly. The consistent values of the upper bounds suggest that the exact JED value is close their approximation.

An interesting aspect of this experiment is the comparison between all upper bounds. As shown in Figure 6.1, in most of the cases, all approximations perform equal in terms of tightness. From the given upper bound relationships, it is expected that ODPJED is always higher or equal to OJED and DPJED. However, an interesting fact is that there are no cases where OJED is tighter than ODPJED. In turn, this suggests that the document preserving constraint introduced by DPJED and ODPJED may not influence the distances of real-world JSON documents.

The overall winner is DPJED. There is no case, in which OJED (nor ODPJED) is tighter than DPJED. For 4 datasets (DLBP, DENF, FENF, and Twitter 2) the difference is also visible in Figure 6.1. By analyzing the datasets, we have identified a specific use-case, in which DPJED is superior to OJED, that occurs frequently these datasets. OJED as well as ODPJED order the key-value pairs of an object alphabetically by key. As a result, two matching values (nested-documents) may not be mapped since their ordered keys are located at different positions and the according subtrees cannot be mapped. An adapted example from the DBLP dataset

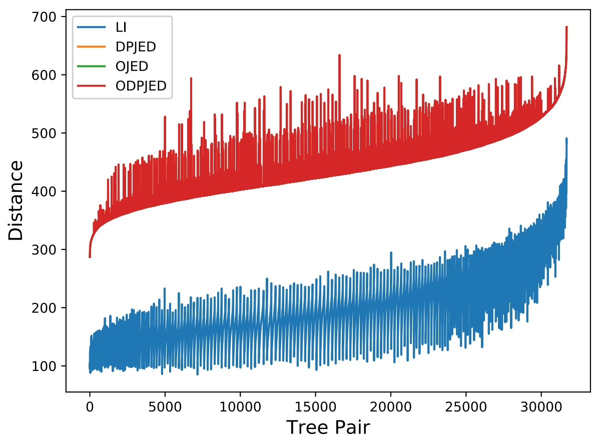

Figure 6.2: Lower and upper bound values for the NBA dataset, ordered by ODPJED.

can be seen in Figure.

Summarizing, the given bounds perform well for real-world JSON datasets. A detailed analysis of more problematic datasets is troublesome since the exact JED distance cannot be computed.

6.3 Runtime Analysis

Next to tightness, we analyze the runtime behaviors of all upper bounds. We consider the same 14 datasets as in the tightness experiment and measure the overall runtime that is needed to perform the pair-wise distance computation explained in Section 6.2. The time that is needed to load the data into memory is not considered.

Figure 6.3 shows the runtime for all upper bound algorithms in milliseconds. Based on the asymptotic complexities of the algorithms, we expect ODPJED to take the least time ($O(max\{|T_J|, |T'_J|\}^2)$). Due to the complexities of OJED ($O(max\{|T_J|, |T'_J|\}^3)$) and DPJED ($O(|T_J||T'_J|(deg(T_J)+deg(T'_J))log_2(deg(T_J)+deg(T'_J))))$, there is no clear winner and the execution time depends on the degree of the trees. In the following paragraphs, we discuss interesting aspects of the given experimental results.

Comparing the results of ODPJED and OJED, the difference in the asymptotic complexity (quadratic vs. cubic) cannot be seen. In fact, OJED executes faster than ODPJED in many cases. This behavior can be explained by analyzing the document sizes within our dataset collection. The average number of nodes per JSON tree is less than 80 for 8 out of 14 considered datasets. Comparing larger JSON trees, e.g., Device (323), NBA (977), Reddit (265), or Twitter 2 (195), the cubic complexity of the OJED algorithm becomes visible in the plot. DPJED has a quadratic complexity in terms of tree sizes. Consequently, DPJED outperforms OJED for large JSON trees, e.g., dataset Device and NBA.

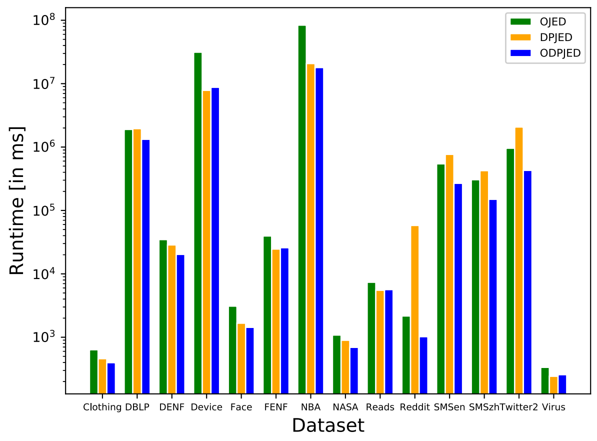

Figure 6.3: Overall runtime in milliseconds for computing the pair-wise distance using OJED, DPJED, and ODPJED.

An interesting dataset for DPJED is Reddit. As shown in Figure 6.3, DPJED takes more than one order of magnitude longer than both, OJED and ODPJED. Again, the behavior can be explained by the characteristics of the dataset. The complexity of the DPJED algorithm depends on the degree of the trees, more specifically the objects degrees. As listed in Table 6.1, compared to all other datasets, Reddit has a high object degree with up to 104 key-value pairs per object.

Overall, ODPJED performs the best and does not run into worst case scenarios. The remaining algorithms provide good results on average; however, OJED is inefficient for large JSON trees and DPJED executes slowly for JSON trees with large object degrees.

		AllCards			Clothing			DBLP			DENF			Device		
#records		20,746			504			1,984,049			7,497			147,899		
#nodes/rec		37	132	298	29	29	51	14	26	651	47	59	283	49	323	3264
#objects		3	12	39	4	4	7	2	2	2	2	2	2	2	2	2
#arrays		8	8	11	3	3	5	0	1.5	5	0	2.7	20	0	1.9	2
#keys		17	59	127	12	12	21	6	11	17	23	27	45	24	29.9	39
#literals		9	54	192	10	10	18	5	11.5	634	22	28	221	23	289	3230
object degree	min	0	0.9	1	2	2	2	1	1	1	0	2.7	20	0	5.9	6
	avg	3	5.2	12	3	3	3	3	5.6	8.5	12	13.4	23	12	14.9	15
	max	16	21.5	28	5	5	5	5	10	16	23	24	25	24	24	24
array degree	min	0	0.03	1	1	1	1	0	1.5	263	0	0.2	1	0	131	1600
	avg	0.3	1.7	21	1.3	1.3	1.6	1	1.9	312	0	0.3	11	0	131	1602
	max	1	7	167	2	2	2	1	2.3	621	0	1.3	95	0	131	1603
max. depth		3	4	4	5	5	5	2	2	2	1	1.3	3	5	5	5
		Face			FENF			Movie			NASA			NBA		
#records		409			13,991			8,765,568			1,000			31,686		
#nodes/rec		29	68	359	47	49	51	15	23	729	15	27	31	659	977	1269
#objects		4	9.3	49	2	2	2	1	1.9	2	1	2	2	21	27	35
#arrays		3	6.5	33	0	0	0	0	0	1	0	1	1	3	3	3
#keys		12	28	147	23	24	25	7	11	13	7	12	14	339	477	619
#literals		10	24	130	22	23	24	7	10	714	7	12	14	332	470	612
object degree	min	2	2	2	0	0	0	1	1.4	11	2	2.1	7	1	1	1
	avg	3	3	3	11.5	12	12.5	4.5	5.8	11	5.5	6.1	7	16	17.4	18.4
	max	5	5	5	23	24	25	7	10	12	7	10.2	12	20	20.5	21
array degree	min	1	1	1	0	0	0	0	0	704	0	1.9	2	2	2	2
	avg	1.3	1.6	1.9	0	0	0	0	0	704	0	1.9	2	5.3	7.5	10
	max	2	3.1	16	0	0	0	0	0	704	0	1.9	2	7	10.7	14
max. depth		5	5	5	1	1	1	1	1.9	2	5	5	5	5	5	5
		Reads			Reddit			Piazza			SMSen			SMSzh		
#records		30,000			25			3756			55,835			31,465		
#nodes/rec		11	11	11	209	265	432	19	21	78	81	81	81	79	80.9	81
#objects		1	1	1	5	13	27	1	1	1	19	19	19	19	19	19
#arrays		0	0	0	6	7.8	10	2	2.3	3	0	0	0	0	0	0
#keys		5	5	5	104	130	206	8	8.6	9	40	40	40	39	39.9	40
#literals		5	5	5	94	115	189	8	9.1	66	22	22	22	21	21.9	22
object degree	min	5	5	5	0	0	0	8	8.6	9	1	1	1	1	1.4	11
	avg	5	5	5	7.6	11.3	20.8	8	8.6	9	2.1	2.1	2.1	4.5	5.8	11
	max	5	5	5	102	103	104	8	8.6	9	9	9	9	7	10	12
array degree	min	0	0	0	0	0	0	0	0.7	15	0	0	0	0	0	0
	avg	0	0	0	0	0.7	1.9	0.5	1.2	29.5	0	0	0	0		0
	max	0	0	0	0	3.9	6	1	1.7	58	0	0	0	0	0	0
max. depth		1	1	1	2	6.2	7	2	2	2	4	4	4	4	4	4
		Standford Dev			Standford Train			Twitter 1			Twitter 2			Virus		
#records		48			442			1,984,049			19,316			500		
#nodes/rec		2212	5379	18,135	294	2597	10,549	17	155	750	115	195	489	19	19	19
#objects		404	988	3303	50	440	1783	4	6.9	68	5	11.4	42	1	1	1
#arrays		118	264	864	28	242	966	0	4.5	65	2	13.2	27	0	0	0
#keys		904	2196	7416	122	1078	4383	8	75	316	55	83	222	9	9	9
#literals		786	1931	6551	94	837	3417	5	68	341	53	87	208	9	9	9
object degree	min	2	2	2	2	2	2	0	0.9	1	0	1.5	2	9	9	9
	avg	2.1	2.2	2.3	2.4	2.45	2.5	1.8	11	14.5	4.4	7.75	11.9	9	9	9
	max	3	3	3	3	3	3	4	37	37	29	34	35	9	9	9
array degree	min	1	2.2	4	1	1	1	0	0.01	2	1	1	2	0	0	0
	avg	3.1	3.7	5.3	1.7	1.8	.9	0	0.6	2.7	1.7	2	2.5	0	0	0
	max	21	43	98	5	43	149	0	1.5	29	2	4.4	16	0	0	0
max. depth		7	7	7	7	7	7	2	4.2	7	3	6	8	1	1	1

Table 6.1: Analysis of the collected JSON datasets.

CHAPTER 7

System Integration

In order to allow users to pose similarity queries on JSON documents as part of a larger query, JED must be integrated into a database. In this section, we introduce a suitable target system, AsterixDB, and discuss two approaches how to integrate the JSON distance.

7.1 AsterixDB

AsterixDB [3] is an open-source big data management system (BDMS) that is designed for computing data-intensive tasks. Its features comprise a JSON-like data model, a declarative query language, a rule-based optimizer, and an execution engine for parallel execution of query plans. The architecture of AsterixDB contains multiple layers of software. In the following, we will give a brief outline of the most relevant parts.

7.1.1 Data Model

The AsterixDB Data Model (ADM) is an extension of the JSON data format. The main differences include: (1) supporting multisets, i.e., an unordered set of values, and (2) providing additional kinds of literals, e.g., circles, polygons, and intervals. Note, that these changes do neither affect the JED definitions nor the presented algorithms. To represent multisets in a JSON tree, a new node type must be introduced.

7.1.2 Query Processing

AsterixDB supports SQL++ [25] queries, a declarative language to query semi-structured data. In the following, we discuss the most-important steps that are performed while processing a query. First, an abstract syntax tree is built while parsing the incoming query. Next, the abstract syntax tree needs to be transformed into a logical plan that serves as input for Algebricks [6], AsterixDB's query compiler. Based on a set of rules, Algebricks optimizes the given logical plan before it translates the logical into a physical plan. The rule-based optimizer is also applied on the physical plan in order to find an even more efficient execution plan. In Algebricks' final step, jobs in the form of directed-acyclic graphs of operators are generated according to the physical plan. Ultimately, the generated jobs are executed in parallel on AsterixDBs runtime engine Hyracks [7].

7.1.3 Similarity Query Support

Currently, AsterixDB provides functionality for string and set similarity queries. Strings can be compared using the Levenshtein distance. Similar to JED, the Levenshtein distance is defined by the number of insert, delete, and rename operations that transform one string into another. For example, the strings *hello* and *shell* have a Levenshtein distance of 2, i.e., delete o, insert s.

Sets can be compared using Jaccard. Given two sets r and s, the Jaccard similarity J is defined as $J(r,s) = \frac{r \cap s}{r \cup s}$. For example, the Jaccards similarity of sets $r = \{1,3,5\}$ and $s = \{1,2,3,4\}$ is $J(r,s) = 0.4$.

In order to enable users to use their own similarity functions, AsterixDB provides user-defined functions (UDF). Implemented UDFs can be called by function name in an SQL++ query; however, the optimizer is not aware of any details of that function and unable to optimize.

7.2 JSON Similarity Queries within AsterixDB

The integration of similarity queries for JSON documents is not straight forward. JED is NP-complete and, therefore, cannot be used in real-world scenarios. In this paper, we present several JED approximation algorithms and give a detailed analysis in an experimental evaluation in Section 6. Since both, OJED and DPJED, have runtime issues for certain tree characteristics, ODPJED may be used in time-critical scenarios. However, DPJED can deal with unordered siblings and is therefore tighter for some datasets. The final decision is user and application dependent and should consider the characteristics of the data.

Next, we discuss two ways of how to integrate JED within AsterixDB. The first approach are AsterixDB's user-defined functions. The major advantage of UDFs is that we do not have to modify the internals of the database system and limit the scope of potential software errors to the newly implemented function. Hence, only the functionality of the according JED approximation has to be implemented. UDFs can be called by its function name using native SQL++ queries.

The second approach is to integrate an edit distance natively within AsterixDB. While UDFs are an efficient way to integrate functionality, they are external code and can therefore not be considered in the optimization process. The function must be implemented inside AsterixDB and further optimization rules must be added. Especially for vast amounts of data or complex similarity queries, an optimized execution plan is crucial for an efficient execution. However, the integration process is more complex and time consuming.

CHAPTER 8

Conclusion and Future Work

In this paper, the unsolved problem of edit-based distance measures for JSON documents is addressed. Based on their hierarchical definition, we represent JSON documents as a hybrid of ordered and unordered trees where each node carries a type and a label. We define the JSON edit distance (JED) as the minimum number of node edit operations, namely insertion, deletion, and rename, that transform one JSON tree into another. We proof that computing JED is NP-complete and present four JED approximation algorithms with lower and upper bound guarantees.

In an experimental evaluation, we analyze and compare the runtime and tightness of the presented approximations. To build the groundwork for our experiments, we gathered 20 real-world JSON datasets and performed an extensive analysis of their characteristics.

We present AsterixDB, a big data management system, as a potential prototype system for JED integration. Further, two integration approaches are discussed , namely UDFs and native code.

So far, the main focus of this work was on distances and algorithms to compute them. However, computing the edit-distance of JSON documents can be applied in various query paradigms, e.g., selection, join, or top-k queries. Each paradigm comes with certain constraints that give rise to improvements over existing approximation algorithms or the introduction of novel algorithms. For example, a given similarity threshold can be leveraged in a join setting.

Acknowledgments. We thank Wail Alkowaileet, Daniel Kocher, Chen Li, Mateusz Pawlik, and Zhihui Yang for valuable discussions. This work was conducted during a research stay of the author at UC Irvine which was supported by the Austrian Marshall Plan Foundation. This work was partially supported by the Austrian Science Fund (FWF): P 29859.

Bibliography

[1] Tatsuya Akutsu, Daiji Fukagawa, Magnús M Halldórsson, Atsuhiro Takasu, and Keisuke Tanaka. Approximation and parameterized algorithms for common subtrees and edit distance between unordered trees. *Theoretical Computer Science*, 470:10–22, 2013.

[2] Tatsuya Akutsu, Daiji Fukagawa, Atsuhiro Takasu, and Takeyuki Tamura. Exact algorithms for computing the tree edit distance between unordered trees. *Theoretical Computer Science*, 412(4-5):352–364, 2011.

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. Asterixdb: A scalable, open source bdms. *arXiv preprint arXiv:1407.0454*, 2014.

[4] Nikolaus Augsten, Michael Böhlen, Curtis Dyreson, and Johann Gamper. Windowed pq-grams for approximate joins of data-centric XML. *VLDB Journal*, 21(4):463–488, 2012.

[5] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive json datasets. *The VLDB Journal*, 28(4):497–521, 2019.

[6] Vinayak Borkar, Yingyi Bu, E Preston Carman Jr, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J Carey, and Vassilis J Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 422–433, 2015.

[7] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1151–1162. IEEE, 2011.

[8] Pierre Bourhis, Juan L Reutter, Fernando Suárez, and Domagoj Vrgoč. JSON: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pages 123–135, 2017.

[9] Tim Bray. The javascript object notation (JSON) data interchange format. RFC 8259, RFC Editor, 11 2017.

[10] Paul C Bryan and Mark Nottingham. Javascript object notation (json) patch. RFC 6902, RFC Editor, 4 2013.

[11] David Buttler. A short survey of document structure similarity algorithms. In *International conference on internet computing*, volume 7, 2004.

[12] Hanyang Cao, Jean-Rémy Falleri, Xavier Blanc, and Li Zhang. JSON patch for turning a pull REST API into a push. In *International Conference on Service-Oriented Computing*, pages 435–449. Springer, 2016.

[13] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *Acm Sigmod Record*, 25(2):493–504, 1996.

[14] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in xml documents. In *Proceedings 18th International Conference on Data Engineering*, pages 41–52. IEEE, 2002.

[15] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. An approach for schema extraction of json and extended json document collections. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 356–363. IEEE, 2018.

[16] Daiji Fukagawa and Tatsuya Akutsu. Fast algorithms for comparison of similar unordered trees. In *International Symposium on Algorithms and Computation*, pages 452–463. Springer, 2004.

[17] Daiji Fukagawa, Takeyuki Tamura, Atsuhiro Takasu, Etsuji Tomita, and Tatsuya Akutsu. A clique-based method for the edit distance between unordered trees and its application to analysis of glycan structures. *BMC bioinformatics*, 12(S1):S13, 2011.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., USA, 1979.

[19] Minos Garofalakis and Amit Kumar. Correlating xml data streams using tree-edit distance embeddings. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 143–154, 2003.

[20] Shoichi Higuchi, Tomohiro Kan, Yoshiyuki Yamamoto, and Kouichi Hirata. An A* algorithm for computing edit distance between rooted labeled unordered trees. In *JSAI International Symposium on Artificial Intelligence*, pages 186–196. Springer, 2011.

[21] Yair Horesh, Ramit Mehr, and Ron Unger. Designing an A* algorithm for calculating edit distance between rooted-unordered trees. *Journal of Computational Biology*, 13(6):1165–1176, 2006.

[22] Bumsuk Jang, SeongHun Park, and Young-guk Ha. A stream-based method to detect differences between xml documents. *Journal of Information Science*, 43(1):39–53, 2017.

[23] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema extraction and structural outlier detection for json-based nosql data stores. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[24] Erwin Leonardi and Sourav S Bhowmick. Detecting changes on unordered XML documents using relational databases: a schema-conscious approach. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 509–516, 2005.

[25] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The sql++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631*, 2014.

[26] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[27] Stanley M Selkow. The tree-to-tree editing problem. *Information processing letters*, 6(6):184–186, 1977.

[28] Dennis Shasha, Jason Tsong-Li Wang, Kaizhong Zhang, and Frank Y Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):668–678, 1994.

[29] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, et al. Schema-agnostic indexing with Azure DocumentDB. *Proceedings of the VLDB Endowment*, 8(12):1668–1679, 2015.

[30] William Spoth, Ting Xie, Oliver Kennedy, Ying Yang, Beda Hammerschmidt, Zhen Hua Liu, and Dieter Gawlick. Schemadrill: Interactive semi-structured schema design. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–7, 2018.

[31] Andrea Torsello and Edwin R Hancock. Computing approximate tree edit distance using relaxation labeling. *Pattern Recognition Letters*, 24(8):1089–1097, 2003.

[32] Gabriel Valiente. An efficient bottom-up distance between trees. In *spire*, pages 212–219, 2001.

[33] Elyas Ben Hadj Yahia, Jean-Rémy Falleri, and Laurent Réveillère. Polly: A language-based approach for custom change detection of web service data. In *International Conference on Service-Oriented Computing*, pages 430–444. Springer, 2017.

[34] Rui Yang, Panos Kalnis, and Anthony KH Tung. Similarity evaluation on tree-structured data. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 754–765, 2005.

[35] Kaizhong Zhang. *The editing distance between trees: algorithms and applications.* PhD thesis, Dept. of Computer Science, Courant Institute, 1989.

[36] Kaizhong Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern recognition*, 28(3):463–474, 1995.

[37] Kaizhong Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15(3):205–222, 1996.

[38] Kaizhong Zhang and Tao Jiang. Some max snp-hard results concerning unordered labeled trees. *Information Processing Letters*, 49(5):249–254, 1994.

[39] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information processing letters*, 42(3):133–139, 1992.