JSON: Data model and query languages

Pierre Bourhis, Juan L Reutter, Domagoj Vrgoč

JSON: Data model and query languages

Pierre Bourhis

CNRS CRIStAL UMR9189, Lille; INRIA Lille

Juan L. Reutter

Pontificia Universidad Católica de Chile and IMFD Chile

Domagoj Vrgoč

Pontificia Universidad Católica de Chile and IMFD Chile

Abstract

Despite the fact that JSON is currently one of the most popular formats for exchanging data on the Web, there are very few studies on this topic and there is no agreement upon a theoretical framework for dealing with JSON. Therefore in this paper we propose a formal data model for JSON documents and, based on the common features present in available systems using JSON, we define a lightweight query language allowing us to navigate through JSON documents, study the complexity of basic computational tasks associated with this language, and compare its expressive power with practical languages for managing JSON data.

Keywords: JSON, Schema languages, Navigation

1. Introduction

JavaScript Object Notation (JSON) [29, 19] is a lightweight format based on the data types of the JavaScript programming language. In their essence, JSON documents are dictionaries consisting of key-value pairs, where the value can again be a JSON document, thus allowing an arbitrary level of nesting. An example of a JSON document is given in Figure 1. As we can see here, apart from simple dictionaries, JSON also supports arrays and atomic types such as numbers and strings. Arrays and dictionaries can again contain arbitrary JSON documents, thus making the format fully compositional.

```
{
  "name": {
     "first": "John",
     "last": "Doe"
     },
  "age": 32,
  "hobbies": ["fishing","yoga"]
}
```

Figure 1: A simple JSON document.

Due to its simplicity, and the fact that it is easily readable both by humans and by machines, JSON is quickly becoming one of the most popular formats for exchanging data on the Web. This is particularly evident in Web services communicating with their users through an Application Programming Interface (API), as JSON is currently the predominant format for sending API requests and responses over the HTTP protocol. Additionally, JSON format is much used in database systems built around the NoSQL paradigm (see e.g. [38, 3, 42]), or graph databases (see e.g. [48]).

Despite its popularity, the coverage of the specifics of JSON format in the research literature is very sparse, and to the best of our knowledge, there is still no agreement on the correct theoretical framework for JSON, and no formalisation of the core query features which JSON systems should support. While some preliminary studies do exist [40, 33, 10, 44, 27], as far as we are aware, no attempt to describe a theoretical basis for JSON has been made by the research community. Therefore, the main objective of this paper is to formally define an appropriate data model for JSON, identify the key querying features provided by the existing JSON systems, study the complexity of basic tasks associated to these query features such as evaluation and satisfiability, and compare existing JSON languages with respect to the type of features they support.

In order to define the data model, we examine the key characteristics of JSON documents and how they are used in practice. As a result we obtain a tree-shaped structure very similar to the ordered data-tree model of XML [8], but with some key differences. The first difference is that JSON trees are deterministic by design, as each key can appear at most once inside a single nesting level of a dictionary. This has various implications at the time of

querying JSON documents: on one hand we sometimes deal with languages far simpler than XML, but on the other hand this key restriction can make static analysis more complicated. Next, arrays are explicitly present in JSON, which is not the case in XML. Of course, the ordered structure of XML could be used to simulate arrays, but the defining feature of each JSON dictionary is that it is unordered, thus dictating the nodes of our tree to be typed accordingly. And finally, JSON values are again JSON objects, thus making equality comparisons much more complex than in case of XML, since we are now comparing subtrees, and not just atomic values. We cover all of these features of JSON in more detail in the paper, and we also argue that, while technically possible (albeit, in a very awkward manner), coding JSON documents using XML might not be the best solution in practice.

Having a formal data model for JSON documents in place, we then consider the problem of querying JSON. As there is currently no agreed upon query language in place, we examine an array of practical JSON systems, ranging from programming languages such as Python [23], XPath analogues for JSON such as JSONPath [24], or fully operational JSON databases such as MongoDB [38], and isolate what we consider to be key concepts for accessing JSON documents. As we will see, the main focus in many systems is on navigating the structure of a JSON tree, therefore we propose a navigational logic for JSON documents based on similar approaches from the realm of XML [21], or graph databases [4, 32]. We then show how our logic captures common use cases for JSON, extend it with additional features, and demonstrate that it respects the "lightweight nature" of the JSON format, since it can be evaluated very efficiently, and it also has reasonable complexity of main static tasks. Interestingly, sometimes we can reuse results devised for other similar languages such as XPath or Propositional Dynamic Logic, but the nature of JSON and the functionalities present in query languages also demand new approaches or a refinement of these techniques.

Finally, since theoretical study of JSON is still in its early stages, we close with a series of open problems and directions for future research.

Related work. We argue in this paper that JSON needs a formal model and that we need to study logics for JSON query processing. However, the analysis of this logic is related to, and draws from, a wide body of research about XML documents, XPath query answering, tree automata, and its various extensions. We remark that in this paper we are more interested in the complexity of evaluating formulas in our logic, and (to a lesser degree) in the

problem of checking whether such formulas are satisfiable. It would be also interesting to have a complete analysis of particular operators of our logic, and how including or excluding them from the language affects the evaluation and the satisfiability problem, similarly as it was done for the case of XPath (see e.g. [5, 21]). There is plenty of work on XML processing that could be reused for this analysis, but we think such considerations are out of the scope of this paper.

On the other hand, we believe that JSON offers some unique challanges that were not considered in the XML setting. First of all, JSON introduces a kind of deterministic navigation, stemming from the restriction that key:value pairs in JSON are unique in a specific level of the dictionary. In turn, this means that the evaluation problem that we study has a different flavour than the usual XML evaluation problem, as far as we are aware (see e.g. [26, 18, 5]). Similarly, to process the equality operators in the JSON logic we propose, one must deal with subtree comparisons. Nonetheless, we can still extend some of the known techniques that work in the XML setting to JSON. For instance, to evaluate a query that uses subtree comparisons, we can preprocess our document by deploying a DAG-like compression such as the one introduced in [14], in order to reduce subtree comparisons to data comparisons. For satisfiability we know that this operator can easily lead to undecidability as long as some form of recursion is allowed [49, 34], and even local equality tests can be troublesome [20, 22]. We confirm this with a simple proof in this paper. The extensions on the basic JSON logic with nondeterminism and/or recursion looks much more like known languages such as Propositional Dynamic Logic or XPath. For the evaluation of this logic we offer a strategy based on compiling subtree equalities as node data tests, and then invoke known bounds for XML [9] or PDL [2, 15]. For satisfiability we offer just two results, one for the full logic and another one for the full logic without equalities, but there are several results for XML that could help us pinpoint the behaviour of each particular operator, starting with [36] for a simple versions in which arrays are allowed nondeterminism, or porting any of the fragments studied in e.g. [39, 18, 5, 22]. One could also try to study more complex problems such as minimization of queries which were considered in the XML context (see e.g. [17]), however, that is not the main focus of our paper.

Organisation. We introduce the required notation, and define the appropriate data model for JSON is discussed in Section 2, and in Section 3 we

present our logic, together with an inspection of JSON query languages that fuelled the design choices of the logic. Section 4 presents a complete analysis of some of the algorithmic properties of our logic, namely the evaluation and satisfiability problems. Our conclusions and the directions for future work are discussed in Section 5.

Remark. Some of the results in this paper have previously appeared in a conference proceedings [11]. However, this paper introduces substantial new material. The analysis of JSON languages has been expanded, and this version includes a series of backwards comparisons between our logic and languages developed for JSON systems. The formalisation of MongoDB's projection is also new to this paper, as well as some of the proofs for the results about evaluation and satisfiability.

2. Data model for JSON

In this section we propose a formal data model for JSON documents whose goal is to closely reflect the manner in which JSON data is manipulated in industry, and to provide a framework in which we can formalise JSON query languages. We begin with a description of navigational primitives for JSON, including what we call *JSON navigation instructions* and how these are used to explore documents. Then we introduce our formal model, called JSON trees, discuss how this model reflects the navigational primitives used in practice, and discuss the main differences between JSON trees and other well-studied tree formalisms such as data trees or XML.

We start by fixing some notation regarding JSON documents. The full specification defines seven types of values: objects, arrays, strings, numbers and the values true, false and null [12]. However, to abstract from encoding details we assume in this paper that JSON documents are only formed by objects, arrays, strings and natural numbers.

Formally, denote by Σ the set of all unicode characters. JSON values are defined as follows. First, any natural number $n \geqslant 0$ is a JSON value, called a *number*. Furthermore, if $\mathtt{s}$ is a string in Σ^*, then $\mathtt{"s"}$ is a JSON value, called a *string*. Next, if $v_1, \ldots, v_n$ are JSON values and $s_1, \ldots, s_n$ are pairwise distinct string values, then $\{s_1 : v_1, \ldots, s_n : v_n\}$ is a JSON value, called an *object*. In this case, each $s_i : v_i$ is called a key-value pair of this object. Finally, if $v_1, \ldots, v_n$ are JSON values then $[v_1, \ldots, v_n]$ is a JSON value called an *array*. In this case $v_1, \ldots, v_n$ are called the *elements* of the array. Note that in the case of arrays and objects the values v_i can again be

objects or arrays, thus allowing the documents an arbitrary level of nesting. A *JSON document* (or just document) is any JSON value. From now on we will use the term JSON document and JSON value interchangeably.

2.1. Navigational Primitives

Arguably all systems that process JSON documents use what we call *JSON navigation instructions* to navigate through the data. The notation used to specify JSON navigation instructions varies from system to system, but it always follows the same two principles:

- If J is a JSON object, then, besides accessing a specific part of this object, one should be able to access the JSON value in a key-value pair of this object.
- If J is a JSON array, then one should be able to access the i-th element of J.

In this paper we adopt the python notation for navigation instructions: If J is an object, then $J[key]$ is the value of J whose key is the string value `"key"`. Likewise, if J is an array, then $J[n]$, for a natural number n, contains the n-th element of J[1].

As far as we are aware, all JSON systems use JSON navigation instructions as a primitive for querying JSON documents, and in particular it is so for the systems we have reviewed in detail: Python and other programming languages [23], the mongoDB database [38], the JSONPath query language [24] and the SQL++ project that tries to bridge relational and JSON databases [41].

Iterating through JSON values. So how does one iterate through JSON documents? Suppose we have a document J that is an object, say of the form $\{s_1 : v_1, \ldots, s_n : v_n\}$, and we wish to check whether any of the values $v_1, \ldots, v_n$ is equal to the string `Alice`. There are two standard ways to do this.

1. First, JSON libraries in programming languages are equipped with an iterator on the values of an object, so one can issue an instruction of the form `for v in J: check(v)` in order to call the function `check`

[1]Some JSON systems use the *dot* notation, where J.key and J.n are the equivalents of J[key] and $J[n]$.

over each of $v_1, \ldots, v_n$. We also mention that some JSON databases also feature FLWR expressions that support this type of iteration (see e.g. [41, 10]).

2. JSON query languages normally provide a declarative alternative for this iteration in the form of a wildcard or a similar argument that would be matched to any key in a navigation instruction, and therefore extend navigation instructions so that they return a set of values. For example, the instruction `J[*]` would retrieve the set $\{v_1, \ldots, v_n\}$ of JSON values, since the wildcard `*` matches any of the keys $s_1, \ldots, s_n$.

To incorporate these primitives into our model, we add two more principles for navigating documents:

- If J is a JSON object, then one should be able to iterate over all of its values.
- If J is a JSON array, then one should be able to iterate over all of its elements.

At this point it is important to note a few important consequences of processing JSON according to these primitives, as these have an important role at the time of formalising JSON query languages.

First, note that we do not have a way of obtaining the keys of JSON objects. For example, if `J` is the object `{"first":"John", "last":"Doe"}` we could issue the instructions `J[first]` to obtain the value of the pair `"first":"John"`, which is the string value `"John"`. Or use `J[last]` to obtain the value of the pair `"last":"Doe"`. However, there is no instruction that can retrieve the keys inside this document (i.e. `"first"` and `"last"` in this case).

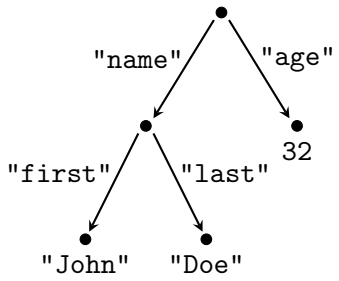
Similarly, for the case of the arrays, the access is essentially a *random access*: we can access the i-th element of an array using an expression of the form `J[i]`, and most of the time there are additional primitives to access the first or the last element of arrays. For instance, `J[-1]` usually defines the last element of the array, and `J[-k]`, the kth element from last to first. Most of the time, it is also possible to navigate through arrays by using a range of indices. For instance, `J[3,6]` navigates from index 3 to index 6 (returning all these elements in this particular order; i.e. returning a sub-array of `J`). Likewise, `J[0,-1]` navigates from the first to the last element in the array. Sometimes it is also possible to navigate the array in reverse using negative indices; e.g. `J[-1,-4]` returns the last four elements of `J` in reverse order.

2.2. JSON trees

JSON objects are by definition compositional: each JSON object is a set of key-value pairs, in which values can again be JSON objects. This naturally suggests using a tree-shaped structure to model JSON documents. However, this structure must preserve the compositional nature of the JSON specification. That is, if each node of the tree structure represents a JSON document, then the children of each node must represent the documents nested within it. For instance, consider the following JSON document J.

```
{
  "name": {
     "first": "John",
     "last": "Doe"
     },
  "age": 32
}
```

as explained before, this document is a JSON *object* which contains two keys: `"name"` and `"age"`. Furthermore, the value of the key `"name"` is another JSON document and the value of the key `"age"` is the number 32. There are in total 5 JSON values inside this object: the complete document itself, plus the literals `32`, `"John"` and `"Doe"`, and the object `"name": {"first":"John", "last":"Doe"}`. So how should a tree representation of the document J look like? If we are to preserve the compositional structure of the JSON specification, then the most natural representation is by using the following edge-labelled tree:



The root of tree represents the entire document. The two edges labelled `"name"` and `"age"` represent two keys inside this JSON object, and they lead to nodes representing their respective values. In the case of the key `"age"` this is just a number, while in the case of `"name"` we obtain another JSON object that is represented as a subtree of the entire tree.

Finally, we need to enforce the property that no object can have two keys with the same name, thus making the model deterministic in some sense, since each node will have only one child reachable by an edge with a specific label. Let us briefly summarise the properties of our model so far.

Labelled edges. Edges in our model are labelled by the keys forming the key-value pairs of objects. This means that we can directly follow the label of edges when issuing JSON navigation instructions, and also means that information about keys is represented in a different medium than JSON values (labels for the former, nodes for the latter). This is inline with the way JSON navigation instructions work, as one can only retrieve values of key-value pairs, but not the keys themselves. To comply with the JSON standard, we enforce that two edges leaving the same node cannot have the same label (but edge labels may be repeated across the document as long as they leave from different nodes).

Compositional structure. One of the advantages of our tree representation is that any of its subtrees represent a JSON document themselves. In fact, the five possible subtrees of the tree above correspond to the five JSON values present in the JSON document J.

Atomic values. Finally, some elements of a JSON document are actual values, such as numbers or strings. For this reason, the leafs of the tree that correspond to numberss and strings will also be assigned a value they carry. Nodes that are leafs and are not assigned a value represent empty objects: that is, documents of the form {}.

Although this model is simple and conceptually clear, we are missing a way of representing arrays. Indeed, consider again the document from Figure 1 (call this document J_2). In J_2 the value of the key `"hobbies"` is an array: another feature explicitly present in the JSON standard that thus needs to be reflected in our model.

As arrays are ordered, this might suggest that we can have some nodes whose children form an ordered list of siblings, much like in the case of XML. But this would not be conceptually correct, for the following two reasons. First, as we have explained, JSON navigation instructions use random access to access elements in arrays. For example, the navigation instruction used to retrieve an element of an array is of the form `J2[hobbies][i]`, aimed at obtaining the i-th element of the array under the key `"hobbies"`. But more importantly, we do not want to treat arrays as a list because lists naturally suggest that one can readily navigate from one element to its neighbours.

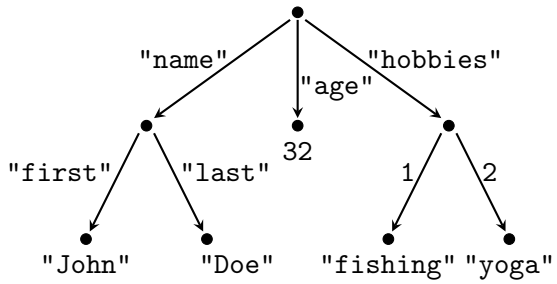


Figure 2: Tree representation of document J_2

On the contrary, in JSON the only way to obtain the next element in an array is via the iteration of all its elements (or by randomly accessing this element through its position). We choose to model JSON arrays as nodes whose children are accessed by axes labelled with natural numbers reflecting their position in the array. Namely, in the case of JSON document J_2 above we obtain the representation shown in Figure 2.

Having arrays defined in this way allows us still to treat the child edges of our tree as navigational axes: before we used a key such as `"age"` to traverse an edge, and now we use the number labelling the edge to traverse it and arrive at the child.

Formal definition. As our model is a tree, we will use tree domains as its base. A *tree domain* is a prefix-closed subset of $\mathbb{N}^*$, where each node of the form $n \cdot i$, with $i \in \mathbb{N}$, represents a child node of n. Without loss of generality we assume that for all tree domains D, if D contains a node $n \cdot i$, for $n \in \mathbb{N}^*$ then D contains all $n \cdot j$ with $0 \leqslant j < i$.

Let Σ be an alphabet. A **JSON tree** over Σ is a structure $J = (D, \mathit{Obj}, \mathit{Arr}, \mathit{Str}, \mathit{Num}, \mathcal{A}, \mathcal{O}, \mathit{val})$, where D is a tree domain that is partitioned by Obj, Arr, Str and Num, $\mathcal{O} \subseteq \mathit{Obj} \times \Sigma^* \times D$ is the object-child relation, $\mathcal{A} \subseteq \mathit{Arr} \times \mathbb{N} \times D$ is the array-child relation, $\mathit{val} : \mathit{Str} \cup \mathit{Num} \to \Sigma^* \cup \mathbb{N}$ is the string and number *value* function, and where the following holds:

1. For each node $n \in \mathit{Obj}$ and child $n \cdot i$ of n, $\mathcal{O}$ contains exactly one triple $(n, w, n \cdot i)$, for a word $w \in \Sigma^*$.
2. The first two components of $\mathcal{O}$ form a *key*: if $(n, w, n \cdot i)$ and $(n, w, n \cdot j)$ are in $\mathcal{O}$, then $i = j$.
3. For each node $n \in \mathit{Arr}$ and child $n \cdot i$ of n, with $i \in \mathbb{N}$, $\mathcal{A}$ contains the triple $(n, i, n \cdot i)$.

4. If n is in *Str* or *Num* then D does not contain nodes of form $n \cdot u$.
5. The value function assigns to each string node in *Str* a value in Σ^* and to each number node in *Num* a natural number.

The usage of a tree domain is standard, and we have elected to explicitly partition the domain into four types of nodes: *Obj* for objects, *Arr* for arrays, *Str* for strings and *Num* for numbers. The first and the second condition specify that edges between objects and their children are labelled with words, but children are uniquely identified by the word on the edge from their parent (i.e. the edge label acts as a key to identify a child). The third condition specifies that the edges between arrays and their children are labelled with the number representing the order of children. The fourth condition simply states that strings and numbers must be leaves in our trees, and the fifth condition describes the value function *val*.

Throughout this paper we will use the term JSON tree and JSON interchangeably; that is, when we are given a JSON document we will assume that it is represented as a JSON tree. As already mentioned above, one important feature of our model is that when looking at any node of the tree, a subtree rooted at this node is again a valid JSON. We can therefore define, for a JSON tree J and a node n in J, a function $json(n)$ which returns the subtree of J rooted at n. Since this subtree is again a JSON tree, the value of $json(n)$ is always a valid JSON.

2.3. JSON and XML

Before continuing we give a few remarks about differences and similarities between the JSON and XML specifications, and how these are reflected in their underlying data models. We start by summarising the differences between the two formats.

1. *JSON documents mix ordered and unordered data.* JSON objects are completely without order. Even within the same programming environment and the same computer, one can not expect that a JSON object will be iterated two times in the same order. On the other hand, for arrays we can do random access to retrieve the i-th element, and therefore we can also iterate over all elements in an array in a fixed order, starting with element 0 and iterating until there are no more elements. We could simulate both behaviours within an XML document with some ad-hoc rules, but this is precisely what we do in our model in a much cleaner way.

2. *JSON trees are deterministic.* The property of JSON trees which imposes that all keys of each object have to be distinct makes JSON trees deterministic in the sense that if we have the key name, there can be at most one node reachable through an edge labelled with this key. On the other hand, XML trees are nondeterministic since there are no labels on the edges, and a node can have multiple children. As we will see, the deterministic nature of JSON documents makes some problems easier and others more difficult than in the XML setting.
3. *Value is not just in the node, but is the entire subtree rooted at that node.* Another fundamental difference is that in XML when we talk about values we normally refer to the value of an attribute in a node. On the contrary, it is common for systems using JSON data to allow comparisons of the full subtree of a node with a nested JSON document, or even comparing two nodes themselves in terms of their subtrees. To be fair, in XML one could also argue this to be true, but unlike in the case of XML, these "structural" comparisons are intrinsic in most JSON query languages, as we discuss in the following sections.[2]

On the other hand, it is certainly possible to code JSON documents using the XML data format. In fact, the model of ordered unranked trees with labels and attributes, which serves as the base of XML, was shown to be powerful enough to code some very expressive database formats, such as relational and even graph data. However, both models have enough differences to justify a study of the JSON standard on its own. This is particularly evident when considering navigation through JSON documents, where keys in each object have to be unique, thus allowing us to obtain values very efficiently. On the other hand, coding JSON data as XML data would require us to have keys as node labels.

3. Navigational queries over JSON data

As JSON navigation instructions are too basic to serve as a complete query language, most systems have developed different ways of querying JSON documents. Unfortunately, there is no standard, nor general guide-

[2]This is not to say that the techniques involved to study problems arising from JSON subtrees must be fundamentally different of those developed for XML or tree automata. In fact, comparing subtrees has already been studied for automata in e.g. [49, 34, 16].

lines, about how documents are accessed. As a result the syntax and operations between systems vary so much that it would be almost impossible to compare them. Hence, it would be desirable to identify a common core of functionalities shared between these systems, or at least a general picture of how such query languages look like. Therefore we begin this section by reviewing the most common operations available in current JSON systems.

Here we mainly focus on the subdocument selecting functionalities of JSON query languages. By subdocument selecting we mean functionalities that are capable of finding or highlighting specific parts within JSON documents, either to be returned immediately or to be combined as new JSON documents. As our work is not intended to be a survey, we have not reviewed all possible systems available to date. However, we take inspiration from MongoDB's query language (which arguably has served as a basis for many other systems as well, see e.g. [3, 42, 47]), as well as JSONPath [24] and SQL++ [41], two other query languages that have been proposed by the community.

Based on this, we propose a navigational logic that can serve as a common core to define a standard way of querying JSON data. We then define several extensions of this logic, such as allowing nondeterminism or recursion. To justify our claim that our logic is a common core of the languages, we show how our logic captures the navigational functionalities of the languages we originally reviewed.

3.1. Accessing documents in JSON databases

Here we briefly describe how JSON systems query documents.

JSON navigation instructions. As we mentioned, JSON navigation instructions are the most basic way of retrieving certain specific nodes from JSON documents. Formally, a JSON navigation instruction is a sequence $N = [w_1][w_2][w_3]...[w_n]$ of words and/or numbers. The idea is that when evaluated over a document J, such an instruction retrieves the node corresponding to $J[w_1][w_2][w_3]...[w_n]$, that is, the node reached when we start form the root of J and iteratively traverse down to the node connected through an edge labelled with w_i. We define the evaluation of navigation instructions by induction: the evaluation of an instruction $[w]$ over a document J, that we denote by $J \cdot [w]$, or just $J[w]$, is:

- The JSON value J' in a key pair $w : J'$ in J, if J is an object and w is a string, or

- the w-th value of J, if J is an array and w is an integer, or
- the evaluation is empty in any other case.

MongoDB's find function. Although technically speaking MongoDB works over their proprietary BSON format (which is just a specific binary encoding of JSON data, with support for a few extra datatypes and where other additional fields have been added), the similarity between both paradigms makes it appropriate to treat MongoDB as a JSON database system. We do highlight that there are some important differences introduced when moving to BSON. For example, in MongoDB the order of documents matters: values `{"first": "John", "last": "Doe"}` and `{"last": "Doe", "first": "John"}` are not the same. For the sake of generality, we ignore these differences in our formalisation, but refer the reader to the official BSON specification for more details [13].

The basic querying mechanism of MongoDB is given by the *find* function [38], therefore we focus on this aspect of the system[3]. The find function receives two parameters, which are both JSON documents. Given a collection of JSON documents and these parameters, the find function then produces an array of JSON documents.

The first parameter of the find function serves as a *filter*, and its goal is to select some of the JSON documents from the collection. The second parameter is the *projection*, and as its name suggests, is used to specify which parts of the filtered documents are to be returned. Since our goal is specifying a navigational logic, we focus for now on the filter parameter, and on queries that only specify the filter. We return to the projection in Section 3.4. For more details we refer the reader to the current version of the documentation [38].

The basic building block of filters are what we call *navigation conditions*, which can be visualised as expressions of the form $N : C$, where N is a JSON navigation instruction and C is a comparison operator. MongoDB uses dot notation for its navigation instructions, and uses JSON syntax itself for its comparison operators; navigation conditions are then expressed as key:value pairs, where the key is a navigation instruction and the value is a comparison object built as shown in Figure 3.

[3]For a detailed study of other functionalities MongoDB offers see e.g. [10]. Note that in [10] the authors do not consider the find function though.

Operator	Semantics (when evaluated over a document J)
N: `$exists:true`	true if $J \cdot N$ is nonempty
N: `$exists:false`	true if if $J \cdot N$ is empty
N: `$eq:D`	true if $J \cdot N$ is equal to `D`
N: `$ne:D`	true if $J \cdot N$ is not equal to `D`
N: `$in:A`	true if $J \cdot N$ is equal to one of the elements in array `A`
N: `$nin:A`	true if $J \cdot N$ is different from all elements in array `A`
N: `$type:<type>`	true if $J \cdot N$ is of type `<type>`
N: `$all:A`	true if $J \cdot N$ is an array containing all elements in `A`
N: `$size:s`	true if $J \cdot N$ is an array with `s` elements
N: `$elemMatch:Q`	true if $J \cdot N$ is an array and one of its elements matches a new query query `Q`

Figure 3: Some comparison operators in Mongo. These comparisons are used together with a navigation condition N, and here D is any JSON document and A is a JSON array. We are omitting `$gt`, `$gte`, `$lt` and `$lte` (greater, greater-or-equal, lower and lower-or-equal than), comparison of strings and regular expressions, operators for bitwise comparison, geospatial operators and operators that deal with comments.

Example 1. *Assume that we are dealing with a collection of JSON documents containing information about people and that we want to obtain the one describing a person named Sue. In MongoDB this can be achieved using the following query* `db.collection.find({name: {$eq: "Sue"}},{})`. *The initial part* `db.collection` *is a system path to find the collection of JSON documents we want to query. Next,* `"name"` *is a simple navigation instruction used to retrieve the value under the key* `"name"`. *Last, the expression* `{$eq: "Sue"}` *is used to state that the JSON document retrieved by the navigation instruction is equal to the document/string* `"Sue"`. *Since we are not dealing with projection, the second parameter is simply the empty document* `{}`.

Navigational conditions, as shown in Figure 3, have a boolean semantics, and they can be combined using boolean operators with the standard meaning [4]. Then we say that a condition $N : C$ selects, or returns, a document J whenever condition C is true when evaluated over $J \cdot N$. Thus, the filter part of the find function always returns entire documents. The second argument

[4]Strictly speaking, negation is not allowed in this fashion, but it can be simulated using the allowed NOR operation.

JSONPath	Intended meaning
`$`	The root of the tree
`@`	The current node being processed
`*`	Any child
`..`	Any descendant
`.'key'`	Value of the key `'key'`
`['key`$_1$`',...,'key`$_k$`']`	Value of any of the keys `'key`$_i$`'`
`[i]`	The `i`th element of an array
`[i`$_1$`,...,i`$_k$`]`	All elements `i`$_j$ $(1 \leqslant j \leqslant k)$ of the array
`[i:j]`	Any element between position `i` and `j`

Table 1: Atomic navigation expressions in JSONPath. When selecting the ith element of the array, a negative i means the ith element from the last (see Section 2.1). Similarly, negative indices in the expression `[i:j]` mean traversing the array in reverse.

of the find function is *projection*, and is used to construct and return different documents. We will show how to use our framework to formalise mongoDB queries with projection later on, in Section 3.4.

Query languages inspired by XPath. The languages we analysed thus far offer very simple navigational features. However, people also recognized the need to allow more complex properties such as nondetermnistic navigation, expression filters and allowing arbitrary depth nesting through recursion. As a result, an adaptation of the XML query language XPath to the context of the JSON specification, called JSONPath [24, 1] was introduced and implemented.

JSONPath uses expressions to traverse JSON documents in the same way XPath works with XML trees. Expressions start with the context variable $, signalling the outermost layer of the document, and navigate the tree repeatedly using one of the atomic navigational axes in Table 1. A navigational JSONPath expression is simply a concatenation of atomic navigational expressions, and its semantics is equivalent to executing these expressions in sequence.

Just as with XPath, JSONPath expressions allow for filtering paths, by the usage of the expression `?()`. Filters can be either paths or a comparison between JSONPath expressions and/or values. They are constructed by using a context variable `@` to refer to the current node in the navigation. For example, we can use `$..book[?(@.isbn)]` to obtain all objects with a key

`book`, and such that this object contains a key-value pair with `isbn` as key. Or we can use `$..book[?(@.price<10)]` to specify that the book contains a key-value pair of the form `"price":n`, with n a number less than 10. Finally, context variables can be mixed inside filters, so for example the expression `..[?($.key1=@.key2)]` searches for the nodes where it is true that the value of the key `key2` of the current node (retrieved using the operation `@.key2`) equals the value of the key `key1` of the root of our document (the operation `$.key1`).

JSONPath does not offer an official semantics other than a considerable number of examples, and thus we can only give an overview of the semantics of this language. We view JSONPath expressions as queries that return a number of nodes from a given document. Given a JSONPath expression α, the evaluation $\alpha(J)$ over a document J would then be all nodes that can be reached by navigating through the axes of Table 1, as if they were JSON navigation instructions (one can understand these axes as a form of non-deterministic navigation instruction). If a filter is encountered during the navigation, then a check must be performed to make sure that the current node satisfies this filter. In turn, a filter of the form `?( e )`, for an expression `e`, is satisfied if `e` returns at least one value. Likewise, a filter of the form `?( e1 = e2 )` is satisfied if there are nodes n_1 and n_2 belonging to the evaluation of `e1` and `e2`, respectively, and $n_1 = n_2$. Other comparisons are defined in the same way. Note that JSONPath also offers a special built-in function `length` such that `@.length` returns the length of the current node in case that it is an array (and an error otherwise).

Query languages inspired by FLWR or relational expressions. There are several proposals to construct query languages that can merge, join and even produce new JSON documents. Most of them are inspired either by XQuery (such as JSONiq [45]) or SQL (such as SQL++ [41]). These languages have of course a lot of intricate features, and to the best of our knowledge have not been formally studied. However, in terms of JSON navigation they all seem to support basic JSON navigation instructions and not much more.

Based on these features, we first introduce a logic capturing basic queries provided by navigation instructions and conditions, and then extend it with non-determinism and recursion so that it captures more powerful querying formalisms.

3.2. Deterministic JSON logic

The first logic we introduce is meant to capture JSON navigation instructions and other deterministic forms of querying such as most of MongoDB's find function conditions. We call this logic *JSON navigation logic*, or JNL for short. We believe that this logic, although not very powerful, is interesting in its own right, as it leads to very lightweight algorithms and implementations, which is one of the aims of the JSON data format.

As often done in XML [21] and graph data [32], we define our logic in terms of unary and binary formulas.

Definition 1 (JSON navigational logic). *Unary formulas φ, ψ and binary formulas α, β of the* JSON navigational logic (JNL for short) *are expressions satisfying the grammar*

$$\begin{array}{rcl} \alpha, \beta & := & \langle \varphi \rangle \mid X_w \mid X_i \mid \alpha \circ \beta \mid \varepsilon \mid r \\ \varphi, \psi & := & \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid [\alpha] \mid EQ(\alpha, A) \mid EQ(\alpha, \beta) \end{array}$$

where w is a word in Σ^*, i is a natural number and A is an arbitrary JSON document.

Intuitively, binary operators allow us to move through the document (they connect two nodes of a JSON tree), and unary formulas check whether a property is true at some node of our tree. For instance, X_w and X_i allow basic navigation by accessing the value of the key named w, or the ith element of an array respectively, and r is used to access the root of a document. They can subsequently be combined using composition or boolean operations to form more complex navigation expressions. Unary formulas serve as tests if some property holds at the part of the document we are currently reading. These also include the operator $[\alpha]$ allowing us to test if some binary condition is true starting at a current node (similarly, $\langle \varphi \rangle$ allows us to combine node tests with navigation). Finally, the comparison operators $EQ(\alpha, A)$ and $EQ(\alpha, \beta)$ simulate XPath style tests which check whether a current node can reach a node whose value is A, or if two paths can reach nodes with the same value. The difference with XML though, is that this value is again a JSON document and thus a subtree of the original tree.

The semantics of binary formulas is given by the relation $[\![\alpha]\!]_J$, for a binary formula α and a document J, and it selects pairs of nodes of J:

- $[\![\langle \varphi \rangle]\!]_J = \{(n, n) \mid n \in [\![\varphi]\!]_J\}$.

- $[\![X_w]\!]_J = \{(n, n') \mid (n, w, n') \in \mathcal{O}\}$.
- $[\![X_i]\!]_J = \{(n, n') \mid (n, i, n') \in \mathcal{A}\}$, for $i \in \mathbb{N}$.
- $[\![\alpha \circ \beta]\!]_J = [\![\alpha]\!]_J \circ [\![\beta]\!]_J$.
- $[\![\varepsilon]\!]_J = \{(n, n) \mid n \text{ is a node in } J\}$.
- $[\![r]\!]_J = \{(n, n) \mid n \text{ is the root of } J\}$.

For the semantic of the unary operators, let us assume that D is the domain of J.

- $[\![\top]\!]_J = D$.
- $[\![\neg\varphi]\!]_J = D - [\![\varphi]\!]_J$.
- $[\![\varphi \wedge \psi]\!]_J = [\![\varphi]\!]_J \cap [\![\psi]\!]_J$.
- $[\![\varphi \vee \psi]\!]_J = [\![\varphi]\!]_J \cup [\![\psi]\!]_J$.
- $[\![[\alpha]]\!]_J = \{n \mid n \in D \text{ and there is a node } n' \text{ in } D \text{ such that } (n, n') \in [\![\alpha]\!]_J\}$
- $[\![EQ(\alpha, A)]\!]_J = \{n \mid n \in D \text{ and there is a node } n_1 \text{ in } D \text{ such that } (n, n_1) \in [\![\alpha]\!]_J \text{ and } \mathit{json}(n_1) = A\}$
- $[\![EQ(\alpha, \beta)]\!]_J = \{n \mid n \in D \text{ and there are nodes } n_1, n_2 \text{ in } D \text{ such that } (n, n_1) \in [\![\alpha]\!]_J, (n, n_2) \in [\![\beta]\!]_J, \text{ and } \mathit{json}(n_1) = \mathit{json}(n_2)\}$.

Example 2. *Consider the formula $\varphi = X_{hobbies} \circ X_1$, and the JSON tree J_2 in Figure 2. Then $[\![\varphi]\!]_{J_2}$ corresponds to a single pair of nodes (r, f), where r is the root of the tree and f is the node corresponding to the value* `"fishing"`. *We can also use JNL to select specific documents from a collection. For example, if we'd like to select all documents referring to John Doe, we can issue the instruction*

$$\psi = [X_{name} \circ \langle EQ(\varepsilon, \texttt{\{"first": "John", "last": "Doe"\}})\rangle],$$

or equivalently,

$$\psi' = EQ(X_{name}, \texttt{\{"first": "John", "last": "Doe"\}}).$$

When evaluated over J_2, we verify that $[\![\psi]\!]_{J_2} = [\![\psi']\!]_{J_2} = \{r\}$, where r is again the root of J_2.

Typically, most systems allow jumping to the last element of an array, or the j-th element counting from the last to the first. To simulate this we can allow binary expressions of the form X_i, for an integer $i < 0$, where -1 states the last position of the array, and $-j$ states the j-th position starting from the last to the first. Having this dual operator would not change any of our results, but we prefer to leave it out for the sake of readability.

JNL and JSON navigation instructions. As promised, we can easily encode JSON navigation instructions with JNL: we iteratively replace every entry of the form $J[\text{key}]$ with X_{key} and $J[n]$ with X_n.

Example 3. *The navigation instruction $J[hobbies][i]$, aimed at retrieving the i-th element of the array under the key* `"hobbies"`*, is expressed in JNL as $X_{hobbies} \circ X_i$.*

The following easy result now follows.

Proposition 1. *For every JSON navigation instruction N one can construct a binary JNL expression φ_N such that for every JSON tree J with root r, the result of evaluating N over J contains node n if and only if the pair (r, n) is in $[\![\varphi_N]\!]_J$.*

3.3. Extensions

Although the base proposal for JNL captures the deterministic spirit of JSON navigation, it is somewhat limited in expressive power. First of all, it fails to express one of the basic query primitives we desire from a JSON language: iteration through JSON values (see Subsection 2.1). Second, JNL as defined in Section 3.2 can not capture neither MongoDB's find function nor the base proposal of JSONPath [24], as it can not traverse arbitrary paths. Here we propose two natural extensions: the ability to non-deterministically select which child of a node is selected, and the ability to traverse paths of arbitrary length. In the following subsection we show how these extensions add enough expressive power to capture the most popular JSON query languages in existence today.

Non-determinism. The path operators X_w and X_i can be easily extended such that they return more than a single child; namely, we can permit matching of regular expressions and intervals, instead of simple words and array positions. Similarly, we can add disjunction to binary formulas to allow them to choose which path they traverse.

Formally, *Non-deterministic JSON logic, or NJNL*, extends binary formulas of JNL by the following grammar:

$$\alpha, \beta \quad := \quad \langle \varphi \rangle \mid X_e \mid X_{i:j} \mid \alpha \circ \beta \mid \alpha \cup \beta \mid \varepsilon$$

where e is a subset of Σ^* (given as a regular expression), and $i \leqslant j$ are natural numbers, or $j = +\infty$ (signifying that we want any element of the array following i). The semantics of the new path operators is as follows:

- $[\![X_e]\!]_J = \{(n, n') \mid \text{there is } w \in L(e) \text{ s.t. } (n, w, n') \in \mathcal{O}\}$.
- $[\![X_{i:j}]\!]_J = \{(n, n') \mid \text{there is } i \leqslant p \leqslant j \text{ s.t. the triple } (n, p, n') \text{ is in } \mathcal{A}\}$.
- $[\![\alpha \cup \beta]\!]_J = [\![\alpha]\!]_J \cup [\![\beta]\!]_J$.

Notice that NJNL can easily express iteration through JSON values: to iterate over of the values of some object we use the expression X_{Σ^*}, and to iterate through the elements of an array, we simply use the expression $X_{1:+\infty}$. As when defining JNL, here we assume that the indices in $X_{i:j}$ are positive. It is straightforward to extend their semantics in order to allow iterating backwards, as discussed previously (see Section 2.1).

Recursion. While NJNL is enough to capture Mongo's find function, we are missing one more ingredient to be able to fully capture JSONPath: the ability to traverse down to any descendant of a node, not just to its children. We go a bit further, and allow exploring not just the descendant query, but any path of arbitrary length that can be described by subsequently applying any formula an arbitrary number of times. We capture this by adding a Kleene star to our logic. *Recursive JNL, or RJNL*, allows $(\alpha)^*$ as a binary formula (as usual we normally omit the brackets when the precedence of operators is clear). The semantics of $(\alpha)^*$ is given by

$$[\![(\alpha)^*]\!] = [\![\varepsilon]\!]_J \cup [\![\alpha]\!]_J \cup [\![\alpha \circ \alpha]\!]_J \cup [\![\alpha \circ \alpha \circ \alpha]\!]_J \cup \ldots .$$

We also define the *Recursive, non-deterministic JSON logic, or RNJNL*, as the logic that augments JNL with both of the extensions mentioned in this section.

Additional node tests. Several proposals for querying JSON data rely on different unary formulas that do not feature navigation, but rather are specific tests that can be processed simply by looking at the current node

of the tree (and its neighbours). For example, MongoDB includes primitives to see that a certain array contains n elements, or that a node is of a given type (string, object, array, etc.); other proposals include testing whether all the elements of an array are different [30], testing that a certain property is present in an object, etc. Since our goal is to understand the navigational capabilities, we do not cover these node tests in full detail. However, adding some of them to our logic is straightforward: we simply create more unary predicates that are to be tested in a given node, just as we did when including the equality predicate EQ.

3.4. Using JSON logic to formalize MongoDB's find function

Just as we did with navigation instructions, we want to show that JSON logic can capture the find function of MongoDB, in the sense that for every such instruction one can construct a logical expression that retrieves the same document. We also use our framework to formalise MongoDB's projection, although such operators are outside the scope of our logic. As we hinted, the presence of non-deterministic navigation such as `all` in MongoDB's conditions demand to use the non-deterministic version of the JSON logic.

The filter part (first parameter). Since both NJNL and MongoDB's find function allow for boolean combinations, we just need to show how to encode a navigation condition of the form $N : C$ in NJNL. Recall that these instructions, when evaluated over a document J, are meant to either select or not select document J, that is, they evaluate to true or false when evaluated over (the root of) J. Thus, for every navigation condition of the form $N : C$ we will construct a unary NJNL formula $\psi_{N:C}$, that evaluates to true over a root of the document if and only if the document is selected by the condition $N : C$. The encoding makes use of the fact that for each navigation expression N, we can find an equivalent (binary) JNL formula φ_N (Proposition 1). In Table 2 we provide the complete encoding of a navigational condition $N : C$ by the formula $\psi_{N:C}$.

Note that the only operators that need non-deterministic traversal to be captured are the parts that navigate to any element in an array. With this encoding we have:

Proposition 2. *For every JSON navigation condition $N : C$ one can construct a unary non-deterministic JNL expression $\psi_{N:C}$ such that for every*

Condition $N : C$	Equivalent NJNL expression $\psi_{N:C}$
N: `{$exists: true}`	$[\varphi_N]$
N: `{$exists: false}`	$\neg[\varphi_N]$
N: `{$eq: D}`	$EQ(\varphi_N, D)$
N: `{$ne: D}`	$\neg EQ(\varphi_N, D)$
N: `{$in: A}`	$EQ(\varphi_N, a_1) \vee \cdots \vee EQ(\varphi_N, a_n)$
N: `{$nin: A}`	$\neg EQ(\varphi_N, a_1) \wedge \cdots \wedge \neg EQ(\varphi_N, a_n)$
N: `{$all: A}`	$EQ(\varphi_N \circ X_{1:+\infty}, a_1) \wedge \cdots \wedge EQ(\varphi_N \circ X_{1:+\infty}, a_n)$
N: `{$size: s}`	$[\varphi_N \circ X_s] \wedge \neg[\varphi_N \circ X_{s+1}]$
N: `{$elemMatch: Q }`	$[\varphi_N \circ X_{1:+\infty}\langle\varphi_Q\rangle]$

Table 2: Encoding a navigation condition of the form $N : C$, where N is a JSON navigation instruction and C a condition with a NJNL expression. Here φ_N is the JNL expression obtained from N in Proposition 1. Note that all operators except for `all` and `elemMatch` require only deterministic navigation.

JSON tree J with root r, the result of evaluating $N : C$ over J returns true if and only if the root r of J is in $[\![\psi_{N:C}]\!]_J$.

The projection part (second parameter). While not dependent on our logic, our formal framework can also be used to formalise the way MongoDB projects over documents, and so we present it here for completeness. Interestingly, our formalisation immediately raises a set of questions and possible extensions that we believe important for future work.

In its essence, the idea of the projection argument is to select only those subtrees of input documents that can be reached by certain navigation instructions. The simplest way of defining this is by stating a set of navigation instructions. Let us come back to the *John Doe* example presented in Section 2. If we want to project out the hobbies, to obtain just the name in this document, we would write:

```
db.collection.find({},{name:1}).
```

Note how we are now using the second argument of the find function. The first argument is left empty, which means that we select all JSON documents. The input to the projection function is either a set of key-value pairs of the form `<path>:1`, or a set of key-value pairs of the form `<path>:0`, where `<path>` is a JSON navigation instruction.

In the former case, the semantics for the projection, applied to a document J, is as follows. For each pair `<path>:1`, we evaluate the instruction `path`,

obtaining a single node n from J. We then prune from J everything which is not an ancestor or a descendant of n. The final projection combines all the structures obtained for each of the `<path>:1` pairs, to form the desired query answer.

Thus, if we want to retrieve only the first name in our *John Doe* document, plus his age, we would write

```
db.collection.find({},{"name.first":1, age:1}),
```

and obtain the document

```
{
  "name": {
     "first": "John"
     },
  "age": 32,
}
```

We define the result of the expression `<path>:1` over a JSON tree J, denoted $J_{\texttt{path}}$, formally as follows. First, let $\varphi_{\texttt{path}}$ be the JNL formula obtained from the navigational expression `path` in Proposition 1. If (r, n) is not in $[\![\varphi_{\texttt{path}}]\!]_J$ for any node n, then $J_{\texttt{path}}$ is the empty data structure. Otherwise, let n be the unique node such that $(r, n) \in [\![\varphi_{\texttt{path}}]\!]_J$. Let $D_{\texttt{path}}$ be the set containing all the nodes from J that: (i) either lie on the unique path from the root of J to n; (ii) or are in the subtree $json(n)$. The expression `<path>:1`, when evaluated over the JSON tree $J = (D, Obj, Arr, Str, Num, \mathcal{A}, \mathcal{O}, val)$, produces a structure $J_{\texttt{path}}$, which is the substructure of J with the domain $D_{\texttt{path}}$ (that is the elements from Obj in this structure are $Obj \cap D_{\texttt{path}}$, and similarly for other elements in J). Notice that, while $J_{\texttt{path}}$ has a tree structure, it is not necessarily a JSON tree (e.g. when the expression `path` selects an element of an array different from the initial element $D_{\texttt{path}}$ is not even a tree domain).

To define the semantics of the projection operator, we have to show how to combine the results of combining multiple paths. More precisely, if the projection operation consists of pairs $\texttt{<path}_1\texttt{>:1, <path}_2\texttt{>:1}$ ... $\texttt{<path}_n\texttt{>:1}$, then denote by $\{J_1, \ldots, J_n\}$ the set $\{J_{\texttt{<path1>}}, \ldots, J_{\texttt{<path}n\texttt{>}}\}$ obtained by evaluating $\texttt{<path}_1\texttt{>:1, <path}_2\texttt{>:1}$ ... $\texttt{<path}_n\texttt{>:1}$ over a document J as described above. Furthermore, denote each data structure J_i as $J_i = (D_i, Obj_i, Arr_i, Str_i, Num_i, \mathcal{A}_i, \mathcal{O}_i, val_i)$. Let J^* be the tuple

$$\big(\bigcup D_i, \bigcup Obj_i, \bigcup Arr_i, \bigcup Str_i, \bigcup Num_i, \bigcup \mathcal{A}_i, \bigcup \mathcal{O}_i, \bigcup val_i\big),$$

where we abuse notation by treating each function val_i as the relation containing all pairs $(x, val_i(x))$ (so that it can be unified as the rest of the components). Since all the expressions are evaluated over the same document J, te structure J^* is a tree, in the sense that it has exactly one root and the children of each of its nodes is given by one of the relations $\bigcup Obj_i$ or $\bigcup Arr_i$. However, J^* is not a JSON document yet: the set $\bigcup D_i$ is not necessarily a tree domain, and because of this the array-child relation $\bigcup \mathcal{A}_i$ does not satisfy the condition of our definition, as one may find e.g. a node n with a child $n \cdot 2$ but no child $n \cdot 0$, nor $n \cdot 1$. However, we can relabel the nodes in the domain and update all relations accordingly. That is, let D^* be the (unique) tree-domain such that the tree represented by D is isomorphic to the tree represented by $\bigcup D_i$, and let h be the isomorphism from D^* to $\bigcup D_i$. Now define Obj^* as, informally, the union $\bigcup h(Obj_i)$: the set that contains a node $h(n)$ for each node n in any of the Obj_i's. Define Arr^*, Str^* and Num^* in the same way. Furthermore, let $\mathcal{A}^*$ map each node $h(n)$ to any of the children included in any of the relations $\mathcal{A}_i$: a set that contains the triple $(h(n), i, n \cdot i)$ for each triple (n, j, n') in any of the $\mathcal{A}_i$'s, assuming $h(n') = h(n) \cdot i$. Likewise, define $\mathcal{O}^*$ as the set that contains a triple $(h(n), w, h(n'))$ for each triple (n, w, n') in any of the $\mathcal{O}_i$'s. Finally, let val^* be the function that assigns to each node n the value $val_i(h^{-1}(n)$, if there exists an $1 \leqslant i \leqslant n$ for which $val_i(h^{-1}(n)$ is defined, and is undefined otherwise.

The resulting JSON document is then defined as the tree

$$(D^*, Obj^*, Arr^*, Str^*, Num^*, \mathcal{A}^*, \mathcal{O}^*, val^*).$$

Note that the projection of a document is again a single JSON document. When applied to a collection of JSON documents, the result of the projection is simply the new set containing the projection of each individual document in the collection.

For the case when the projection is a set of pairs `<path>:0`, the definition is much simpler. For a single document, one instead removes from J all subtrees rooted at the nodes selected by each of the paths in the pair. Thus, if we only want the hobbies of our document from Figure 1, we write

```
db.collection.find({},{name:0, age:0}).
```

Again, when applied to a collection of JSON documents, the result of the projection is simply the new set containing the projection of each individual document in the collection.

JSONPath	Intended meaning	RNJNL equivalent
`$`	The root of the tree	r
`@`	The current node being processed	ε
`*`	Any child	$X_\Sigma \cup X_{1:+\infty}$
`..`	Any descendant	$(X_\Sigma \cup X_{1:+\infty})^*$
`.'key'`	Value of the key `'key'`	$X_{\texttt{key}}$
`['key`$_1$`',...,'key`$_k$`']`	Value of one of the keys `'key`$_i$`'`	$X_{\texttt{key}_1} \cup \ldots \cup X_{\texttt{key}_k}$
`[i]`	The `i`th element of an array	X_i
`[i`$_1$`,...,i`$_k$`]`	The element `i`$_j$ of the array	$X_{i_1} \cup \ldots \cup X_{i_k}$
`[i:j]`	Any element between position `i` and `j`	$X_{i:j}$

Table 3: Atomic navigation expressions in JSONPath and their equivalents in RNJNL.

It is easy to see that the semantics is well defined in both cases, and that one can compute the JSON document resulting of these projections in Ptime. However, one naturally wonders whether one could extend these instructions to, for example, allow for a combination of pairs `<path>:1` and `<path>:0`; or allowing a finer interplay between filtering and projecting, to issue instructions such as *remove all nodes where the age is less than 18*. We believe this is an interesting ground for future work, as there are also fundamental questions regarding the expressive power of these transformations, and the possible interactions with schema definitions.

3.5. JNL to formalize JSONPath

The navigational power of non-deterministic JNL goes a bit beyond the basic expressions of JSONPath, and indeed we can simulate almost all navigation axes with non-deterministic JNL[5]. In Table 3 we show an equivalent RNJNL formula φ_E, for an atomic navigation expression E of JSONPath. Since every JSONPath navigational expression is simply a concatenation of atomic navigational expressions, we easily obtain the following:

Proposition 3. *For every JSONPath navigational expression E, there is a binary RNJNL formula φ_E such that n is selected by E when evaluated over J, if and only if the pair (r, n) belongs to $[\![\varphi_E]\!]_J$.*

As we mentioned, the only axis for which we need recursion is the descendant `...` As far as equality tests are concerned, the most common use

[5]Here we assume that JNL expressions X_i and $X_{i:j}$ can use negative indices, and extend their semantics accordingly.

of filters in JSONPath, which test if the current node, or some of its descendants equals either a fixed value or another descendant of the current node, is precisely the same as the semantics of the *EQ* operation in JNL, so one can easily obtain a similar result for JSONPath formulas that include equality comparisons. To support more involved tests, and the use of the in-built function `.length`, we would have to extend RNJNL with these capabilities explicitly, as is usually the case.

4. Algorithmic properties of JNL

As promised, we show that JNL is a logic particularly well behaved for database applications. For this we study the evaluation problem and satisfiability problem associated with JNL. The Evaluation problem asks, on input a document J, a JNL unary expression φ and a node n of J, whether n is in $[\![\varphi]\!]_J$. The Satisfiability problem asks, on input a JNL expression φ, whether there exists a document J such that $[\![\varphi]\!]_J$ is nonempty. We begin with the deterministic version of our logic, and then move to study its extensions.

4.1. Basic deterministic JNL

We start with evaluation, showing that JNL indeed matches the "lightweight" spirit of the JSON format and can be evaluated very efficiently:

Proposition 4. *The* Evaluation *problem for JNL can be solved in time $O(|J| + |\varphi|)$, with J being the JSON tree, and φ the JNL formula. If φ does not use the $EQ(\alpha, \beta)$ operator, then the problem can be solved in time $O(|\varphi|)$.*

Before proving this result, let us remark that we cannot immediately transfer the techniques for XPath evaluation (see e.g. [43, 25]). First, we need to deal with the $EQ(\alpha, \beta)$ operator; for this we need a preprocessing phase along the lines of the compression schema used in [14]. Furthermore, to obtain the $O(|\varphi|)$ bound we rely on the determinism of JNL. First, each unary JNL formula φ is a boolean combination of operators of the form $\top$, $[\alpha]$, $EQ(\alpha, \beta)$ and $EQ(\alpha, A)$, so evaluating this formula is proportional to checking whether each of these operators is true at the node n. Second, for each binary formula α, due to determinism of the tree J, there is at most one node n' with $(n, n') \in [\![\alpha]\!]_J$. Using these two facts, we can then evaluate all the nested subformulas of φ in a recursive manner, needing to process each operator of φ only once.

Proof: For this proof we will assume that the JSON tree is stored through a series of pointers (i.e. in a way that the tree data structure is usually implemented). Each pointer in this representation will additionally carry its label (i.e. the key), so that we can directly access a child of a node using a particular label. For instance, we can ask for the `name` child of the root of the JSON tree from Figure 2 to obtain the node to which the pointer labelled `name` points to. Note that we can easily implement this in such a way that asking for a child with a specific key has cost $O(1)$, assuming that the key itself is treated as a single symbol. The latter is a reasonable assumption, since in practice the size of the JSON document is almost never dominated by the size of the keys that are used. Alternatively, we could simply put a fixed upper bound on the number of symbols used in each key for this assumption to hold true.

We will furthermore assume that the JNL query is given by its parse tree, which is again a reasonable assumption, as database systems usually store the queries in this form internally.

Next, we make three observations that will be key to obtaining the evaluation algorithm. First, note that it is sufficient to solve the evaluation problem for the root a JSON tree. Indeed, testing if $n \in [\![\varphi]\!]_J$ for some formula φ is the same as testing if this hold at the root of the tree $json(n)$, i.e. the subtree of J rooted at n. Second, given a binary formula α and a node n in a JSON tree J, there can exists at most one node n' in J such that $(n, n') \in [\![\alpha]\!]_J$. The latter follows from the fact that path formulas in JNL are simply concatenation of symbols (or unary tests which hold at one precise node), and that a JSON tree is deterministic, i.e. it can have at most one path with a pre-defined labelling of the edges. Third, we will heavily rely on the fact that each unary JNL formula is a boolean combination of the operators $\top$, $[\alpha]$, $EQ(\alpha, \beta)$ and $EQ(\alpha, A)$.

To make the presentation easier to follow, we will present the algorithm for JNL fragments of increasing complexity, starting with the simplest fragment not allowing the EQ operator, nor the use of $\langle\rangle$ inside the operator $[\alpha]$; that is, we do not allow the nesting of unary and binary formulas. The syntax of this fragment is given below:

$$\begin{aligned} \alpha, \beta &:= X_w \mid X_i \mid \alpha \circ \beta \mid \varepsilon \\ \varphi, \psi &:= \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid [\alpha] \end{aligned}$$

The key observation here is that our formula is a boolean combination of the operators $\top$ and $[\alpha]$, where α is a simple concatenation of key names or

array positions (and ε). We can therefore view it as a propositional formula where all the variables are different, and each operator $[\alpha]$ or $\top$ corresponds to one variable. What then remains is to evaluate each $[\alpha]$, seeing whether it holds true at the root of the tree J, and using this information evaluate the propositional formula. Since the tree J is deterministic, and α is simply a concatenation of key names, array positions and ε, checking whether $[\alpha]$ is true can clearly be done in time $O(|\alpha|)$ by following the appropriate pointers from the root of J. Therefore, the total time needed to evaluate φ corresponds to the sum of $O(|\alpha|)$, for each operator $[\alpha]$ in φ, plus the time needed to evaluate the propositional formula, which can be done in time $O(|\varphi|)$, thus giving the total time bounded by $O(2 \cdot |\varphi|) = O(|\varphi|)$.

Next we show how this algorithm can be extended when we allow nesting of subformulas; namely, if we allow the operator $\langle\psi\rangle$ inside binary formulas. In this case, we can proceed similarly as when nesting is not present. More precisely, each binary subformula $\langle\psi\rangle$ occurring in φ is again a boolean combination of $\top$ and $[\alpha]$, with the difference that α can again contain $\langle\psi'\rangle$, with ψ' a unary formula, and so on recursively. In this case, we can use the same algorithm as when nesting is not present, but with potential recursive calls to itself. That is, our formula φ is going to be a boolean combination of $\top$s and $[\alpha]$s, so we proceed by treating it again as a propositional formula where we need to find the value of $[\alpha]$s. When processing each $[\alpha]$ we rely on the fact that α is a concatenation of key names, array positions, ε, *and* $\langle\psi\rangle$, for ψ a unary formula. We therefore process α by remembering where we currently are in the document J (i.e. we remember the unique node where each part of the concatenation in α starts), and if we encounter $\langle\psi\rangle$ in α, we evaluate ψ recursively at this node using the same procedure, until its value can be computed because it contains no further nesting (i.e. we are treating a formula with no $\langle\psi'\rangle$ inside its binary subformulas). It can be shown by an easy induction on the nesting depth that this indeed works in time $O(|\varphi|)$. The base case is when φ contains no nesting, so the result follows from the algorithm above. If we assume that the claim holds for each formula where the nesting depth is n, then a formula of nesting depth $n+1$ can easily be evaluated, since each nested subformula $\langle\psi\rangle$ can be evaluated in time $O(|\psi|)$ by the induction hypothesis, so our evaluation algorithm, which simply treats φ as a propositional formula whose truth value depends on the value of (top level) operators $[\alpha]$, can process each α as before (i.e. by following the required concatenation of symbols inside the tree), while each of its $\langle\psi\rangle$ subformula is evaluated in time $O(|\psi|)$, giving the total time of

$O(|\varphi|)$.

Finally, we show how to treat the equality operators $EQ(\alpha, \beta)$, and $EQ(\alpha, A)$. For his, observe that comparing if two JSON trees J_1 and J_2 are equal can be done in time $O(max\{|J_1|, |J_2|\})$ by doing a traversal of the two trees in parallel (by following pointers from the root). Therefore, to evaluate $EQ(\alpha, A)$ inside a formula, we will be starting at some node n of the tree J. We can then compute the unique node n' such that $(n, n') \in [\![\alpha]\!]_J$ as in the case when no equality tests are used (or recursively as above if α contains equality tests). Having this n', we can then compare the subtree of J rooted at n' with A in time $O(|A|)$ as described above, so it is bounded by the size of the subformula. If the two trees are equal we mark the node corresponding to $EQ(\alpha, A)$ in the parse tree of φ with true, and if they are not equal, or n' does not exist, with false. Notice that we still maintain the $O(|\varphi|)$ bound when processing these types of queries.

Handling $EQ(\alpha, \beta)$ cannot be done in a similar manner, because it would involve checking for equality of two JSON subtrees, which can take time comparable to the size of those trees, an arbitrary number of times. To reduce the complexity, if any $EQ(\alpha, \beta)$ operator is present then we first preprocess the tree, assigning colors to nodes such that n and n' are assigned the same color if the subtrees starting from this nodes is the same (in other words, if $json(n) = json(n')$). Notice that this preprocessing phase only needs $|J|$ colors, and we can do it in linear time in the size of J by first processing all leafs of J, then all nodes one level above, and so on: checking for a level only needs to take into account the children of the nodes at this level, and their respective colors. Now, to evaluate $EQ(\alpha, \beta)$, we simply need to check whether the node retrieved by α is colored with the same color as the node retrieved by β, which can be done in constant time. This gives us a total time in $O(|J| + |\varphi|)$, as we first need to do the linear preprocessing and then proceed with the evaluation of φ. □

Next, we move to satisfiability. Here we can easily get NP-hardness from the fact that JNL can emulate propositional formulas. However, we show that this lower bound holds even for the positive fragment of JNL. It might be surprising that the positive fragment is not always trivially satisfiable, but this holds due to the fact that each key in an object is unique, so a formula of the form $[X_a \circ \langle [X_1] \rangle] \wedge [X_a \circ \langle [X_b] \rangle]$ is unsatisfiable because it forces the value of the key a to be both an array and a string at the same time. We also show a matching NP-upper bound, which is again not direct due to the

presence of numbers in our logic.

Proposition 5. *The* SATISFIABILITY *problem for JNL is* NP*-complete. It is* NP*-hard even for formulas neither using negation nor the equality operator.*

Proof: **Membership**. This is a standard guess and check algorithm for NP. More precisely, if φ is a satisfiable formula, the document satisfying φ can have height at most $|\varphi|$, and its width at each level is at most the number of operators appearing at this level in the formula. A small complication is caused by having array positions written in binary, so constructing this many array elements might be exponential in the length of the input number. However, not all the array elements need to be materialized, as we can simply sort the required numbers at each level of our formula and start enumerating them from 1 again. This gives us a polynomial size witness for the formula, since we only need to materialize the array elements that are mentioned in the formula itself. It is straightforward to see that a formula where the numbers are reassigned in this way is satisfiable if and only if the original formula is satisfiable. We can now simply guess a witness of polynomial size and using Proposition 4 check whether it satisfies the formula.

Hardness. Reduction is from 3SAT. Let φ be a propositional formula in 3CNF using variables $p_1, \ldots, p_n$ and clauses $C_1, \ldots, C_m$. For each p_i we define the formula $\theta_{p_1} = (X_{p_1}\langle X_1\rangle) \vee (X_{p_1}\langle X_w\rangle)$, with w a fresh string, with the intention of allowing, as models, all valuations of each of the p_i's: if the object under key p_1 is an array, then we will interpret this as p_i being assigned the value true, and if it is an object we will interpret that p_i was assigned the value false. Moreover, for each of the clauses C_j that uses variables a, b and c, define γ_{C_J} as $X_aS_a \vee X_bS_b \vee X_cS_c$, where each S_a, S_b and S_c is either $\langle X_1\rangle$, if a (respectively, b or c) appears positively in C_j, and $\langle X_w\rangle$ otherwise.

Recall that all edges leaving from array nodes are labelled with natural numbers, and all edges leaving from object nodes are labelled with strings. This means that for any document J it must be the case that $[\![\langle X_1\rangle]\!]_J$ and $[\![\langle X_a\rangle]\!]_J$ are always disjoint. Moreover, recall that an object cannot have two pairs with identical keys, so a node in a JSON trees cannot have two children under an edge with the same label. With these two remarks it is then immediate to see that φ is satisfiable if and only if the following JNL

expression is satisfiable:

$$\bigwedge_{1 \leqslant i \leqslant n} [\theta_{p_i}] \wedge \bigwedge_{1 \leqslant \ell \leqslant m} [\gamma_{C_\ell}]$$

□

4.2. Adding non-determinism and recursion

So what happens to the evaluation and satisfiability when we extend this logic? If we disallow the $EQ(\alpha, \beta)$ operator then we can retain a linear algorithm (in the data) by using the classical PDL model checking algorithm [2, 15] with small extensions which account for the specifics of the JSON format, and using again the linear preprocessing to deal with the $EQ(\alpha, A)$ operators. On the other hand, if $EQ(\alpha, \beta)$ is allowed then we can use the preprocessing to transform this operator into something that resembles XML data equality tests, and then invoke the linear algorithm in [9].

Proposition 6. *The evaluation problem for Recursive, non-deterministic JNL (RNJNL) can be solved in time $O(|J| \cdot |\varphi|)$, when $EQ(\alpha, \beta)$ constructs are not allowed, and in time $O(|J| \cdot 2^{|\varphi|})$ in general. One can also solve the evaluation problem in in time $O(|J|^3 \cdot |\varphi|)$.*

Proof: When our formula does not use the $EQ(\alpha, \beta)$ operator, we can reuse the classical model checking algorithm from PDL [2, 15] that runs in time $O(|J| \times |F|)$, since our logic is a syntactic variant of PDL and JSON trees can be viewed as a generalisation of Kripke structures. Some small changes are needed though in order to accommodate the specifics of the JSON format and of our syntax. First, arrays can be treated as usual nodes, with the edges accessing them being labelled by numbers. Second, for the formula X_e, where e is a regular expression, JNL traverses a single edge (i.e. the regular expression is not applied as the Kleene star over the formulas). To accommodate this, we can mark each edge of our tree with an expression e such that the label of the edge belongs to the language of e. Since checking membership of a label l in the language of the expression e can be done in $O(|e| \cdot |l|)$, and the sum of the length of all the labels is less than the size of the model (as we have to at least write them all down), this can be done in $O(|e| \cdot |J|)$. We now repeat this for every regular expression appearing in our JNL formula. Since the number of expressions is linear in

the size of the formula the preprocessing takes linear time. Finally, we need the preprocessing that employs as many colors as the number of $EQ(\alpha, A)$ operators appearing in φ, and that colors the nodes n such that $json(n) = A$. With this preprocessing, we can treat the fact that $json(n) = A$ as a unary node test, and then process $EQ(\alpha, A)$ as the formula that process α and then checks that this node satisfies this unary test. We can now run the classical PDL model checking over this extended structure treating regular expressions and numbers as ordinary edge labels.

When the $EQ(\alpha, \beta)$ operator is used we need again the linear preprocessing to reduce subtree equality to data tests (checking if the color of a node is the same to the color of another node). With this model, the evaluation of RNJNL can be dealt with using the same techniques for evaluating XPath under said data tests. We recall the $O(|J| \cdot 2^{|\varphi|})$ bound from [9].

Interestingly, we can also show a $O(|J|^3 \cdot |\varphi|)$ bound, and actually this bounds hold even for the more general problem that takes as input a JSON tree J, a JNL formula (with recursion and equality tests) F, and computes the relation (or set if F is unary) $[\![F]\!]_J$.

Before describing the algorithm for evaluation, we give some observations. First, notice that for a JSON tree J, we measure the size $|J|$ as the number of nodes in J (we implicitly assume that the size of the edge labels is dominated by this number; alternatively, one could also count the sizes of edge labels to contribute to the size of the model without changing much). This remains true if we view J as a graph, since we will have a graph with $|J|$ nodes and $|J|-1$ edges (since J is a tree). Therefore standard graph traversal algorithms such as breadth and depth first search will run in time $O(|J|)$ over a JSON tree. Next, we will assume that there is an ordering of the elements of a JSON tree J. This is easily obtained by e.g. running a graph traversal algorithm over our tree, and will allow us to sort the results of our query. Finally, we will assume that there is a total ordering on the key names and array positions, with numbers coming before any string, and strings being ordered according to the lexicographical ordering.

Considering this, the first step of our algorithm will be to pre-compute the equality relation with a linear preprocessing, assigning colors to nodes with the same subtrees just as we did in the proof of Proposition 4, so that we can test if $json(n) = json(n')$ in constant time, for two nodes n, n'.

Next we compute the relation $E(A)$, of all the nodes equal to the fixed JSON document A, for each operator $EQ(\alpha, A)$ appearing in F. We can color all nodes equivalent to a single document A just as we did with the

linear preprocessing above. We therefore need one new color per document A appearing in φ, which can be done in $O(|J| \cdot |\varphi|)$.

Finally, we will pre-compute the relations $[\![X_e]\!]_J$, for every operator X_e appearing in our input formula F. This can easily be done with a single pass over the tree J, where checking if the label of the edge coming into the current node belongs to e can be done in time $O(|e|)$. Therefore the entire algorithm takes time at most $O(|J| \times |F|)$. Similarly, we pre-compute the relation $[\![X_{i:j}]\!]$, for each operator can be done in $O(|J| \times |F|)$. Note that all of these relations have size at most $O(|J|)$, since they are defined over the edges of our tree J.

We now solve the evaluation problem using a dynamic programming algorithm that processes the parse tree of our formula F in a bottom-up fashion, and computes, for every binary sub-expression α of F, the binary relation $[\![\alpha]\!]_J$. Similarly, we compute, for every unary sub-expression φ of F, the set $[\![\varphi]\!]_J$. Clearly, if each such relation can be computed within time $O(|J|^3 \cdot |\varphi|)$, the evaluation problem can be solved within the required time.

We now discuss how to obtain the desired time bound. The algorithm is similar to an algorithm used for evaluating regular expressions on graphs [31, 32], and can be described inductively as follows.

The base cases for binary expressions, that is, computing $[\![\alpha]\!]_J$ where α is one of ε, X_e, or $X_{i:j}$ are straightforward, since ε simply defines the diagonal relation, and we already have all the X_e, and $X_{i:j}$ pre-computed. Similarly, the base cases for unary expressions, that is, computing $[\![\top]\!]_J$ is trivial as well.

For the induction step we need to consider binary expressions of the form $\langle\varphi\rangle$, $\alpha \circ \beta$, $\alpha \cup \beta$, and α^*. Also, we need to consider node expressions of the form $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $[\beta]$, $EQ(\alpha, \beta)$, and $EQ(\alpha, A)$.

In the case of binary expressions, the case $\langle\varphi\rangle$ is trivial because $[\![\varphi]\!]_J$ contains at most $|J|$ elements. For $\alpha \cup \beta$ we can first sort both relations $[\![\alpha]\!]_J$ and $[\![\beta]\!]_J$ (costing $O(|J|^2 \log |J|)$ time since they are of the size at most $|J|^2$) and then compute $[\![\alpha \cup \beta]\!]_J$ while performing a single pass over $[\![\alpha]\!]_J$ and $[\![\beta]\!]_J$. For $\alpha \circ \beta$ the relation $[\![\alpha \circ \beta]\!]_J$ is the composition $[\![\alpha]\!]_J \circ [\![\beta]\!]_J$, which can be obtained by computing the natural join of $[\![\alpha]\!]_J$ with $[\![\beta]\!]_J$ by sorting the first relation on the second attribute, and the second one on the first one. Computing $[\![\alpha^*]\!]^G$ amounts to computing the reflexive-transitive closure of $[\![\alpha]\!]^G$ which can be done in time $|J|^3$ by Warshall's algorithm.

In the case of unary expressions, $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, and $[\beta]$ are straightforward to evaluate in the desired time. The most interesting cases here involve

the operator EQ.

Computing $[\![EQ(\alpha, \beta)]\!]_J$ from $[\![\alpha]\!]_J$ and $[\![\beta]\!]_J$ can be done similarly to how one performs a sort-merge join. First, we sort the relations $[\![\alpha]\!]_J$ and $[\![\beta]\!]_J$ on the first attribute in time $O(|J|^2 \times \log |J|)$. Then, for each of the $|J|$ possible nodes n (in increasing order), we can compute in time $O(|J|)$ the sets $D_{n,1} = \{n_1 \mid (n, n_1) \in [\![\alpha]\!]_J\}$ and $D_{n,2} = \{n_2 \mid (n, n_2) \in [\![\beta]\!]_J\}$. Since both $D_{n,1}$ and $D_{n,2}$ have at most $|J|$ elements, and using our pre computed equality relation we can test in constant time if $json(n_1) = json(n_2)$, it can be tested in time $O(|J|^2)$ if the two sets have an element with the same value. The result contains all such elements, and can therefore be computed in time $O(|J|^3)$. The case of $EQ(\alpha, A)$ is similar, since we have pre-computed the relation $E(A)$ containing all the elements in J equal to A.

□

For satisfiability the situation is radically different, as the combination of recursion, non-determinism and the binary equalities ends up being too difficult to handle. The following proposition can be shown by adapting similar results for tree automata using equality and inequality constraints (see e.g. [16], proposition 4.2.10), but for completeness we give here a reduction from the halting problem of two counter machines.

Proposition 7. *The* SATISFIABILITY *problem is undecidable for recursive, non-deterministic JNL formulas (RNJNL), even if they do not use negation.*

Proof: The proof is by a reduction from the halting problem of two-counter machines [37, 28]. A two-counter machine can be defined as a non-deterministic finite state automaton equipped with two counters C_1 and C_2 that hold non-negative integer values. That is, a two-counter machine is a tuple $M = (Q, q_0, q_f, \delta, C_1, C_2)$, where Q is a finite set of states, q_0 is the initial state, q_f is the final state, C_1 and C_2 are the two counters, and δ is the transition relation. The transition relation δ contains instructions of the following form:

- $\delta(q) = (INC\ C_i, q')$, stating that, when in the state q, the machine should increase the counter C_i by one, and move to the state q';
- $\delta(q) = (DEC\ C_i, q')$, stating that, when in the state q, the machine should decrease C_i and move to the state q'. In case that $C_i = 0$, the counter remains unchanged;

- $\delta(q) = (IF\ C_i = 0\ THEN\ q_1, ELSE\ q_2)$, stating that, when in the state q, the machine should move to the state q_1 if the counter C_i contains a zero, and move to the state q_2 otherwise.

The triple consisting of the current state of the machine, and the values in the two counters is called the configuration of the machine. The machine starts in the initial configuration where the state is q_0 and $C_1 = C_2 = 0$. It then starts executing the instructions of the relation δ. The halting problem for two-counter machines asks if there is a sequence of transitions in δ starting with the initial configuration, and such that the configuration reached at the end of this sequence has the state q_f and $C_1 = C_2 = 0$. The configuration with the state q_f and $C_1 = C_2 = 0$ is called accepting. As shown in e.g. [28], this problem is undecidable.

For our reduction, given a two-counter machine M, we need to define a formula φ that is satisfiable if and only if M can reach the accepting configuration when started in the initial configuration. We will encode a configuration (inside a run) of a two-counter machine M using a JSON object that has precisely four keys:

- the key c_1, whose value represents the value of the counter C_1;
- the key c_2, whose value represents the value of the counter C_2;
- the key *state*, whose value is the current state of the machine represented as a string; and
- the key *next*, whose value is another JSON object representing a configuration of our machine.

The values of keys c_1 and c_2 will be nested JSON objects with a single key a, whose depth represents the value of the corresponding counter. The empty counter is represented by the string `"0"`. For instance, if $C_1 = 2$, our coding will have c_1`:{"a":{"a":"0"}}`, and if $C_2 = 0$, then our encoding will have c_2`:"0"`. Therefore, if we are in a configuration where q is the current state and the values of the counters are $C_1 = C_2 = 1$, then we will code this configuration with the following JSON document:

```
{
   "state": "q",
   "c1": {"a": "0"},
```

```
   "c2": {"a": "0"},
   "next": {...}
}
```

The idea here is that the key `"next"` is either an encoding of a valid configuration that follows the current one, or is not present in the document (signalling that we have reached an accepting configuration).

We now define a formula φ_M that will accept precisely such encodings of valid runs of a counter machine M. We let $\varphi_M = \langle F_{init} \circ F_{transition} \circ F_{accept} \rangle$, where:

- $F_{init} = \varepsilon \langle EQ(X_{c_1}, "0") \wedge EQ(X_{c_2}, "0") \wedge EQ(X_{state}, "q_0") \rangle \circ X_{next}$, checks that we are in the initial configuration at the root of our JSON document;
- $F_{accept} = \varepsilon \langle EQ(X_{state}, "q_f") \rangle$, checks that at the end we reach the accepting configuration; and
- $F_{transition} = (\varepsilon \langle \bigvee_q \varphi_q \rangle \circ X_{next})^+$, checks that the transition from one configuration to the next is done correctly according to δ.

The formulas φ_q code the transition $\delta(q)$ as follows:

- if $\delta(q) = (INC\ C_i, q')$, then

$$\varphi_q = EQ(X_{c_i}, X_{next} \circ X_{c_i} \circ X_a) \wedge$$
$$EQ(X_{state}, "q") \wedge EQ(X_{next} \circ X_{state}, "q'")$$

- if $\delta(q) = (DEC\ C_i, q')$, then

$$\varphi_q = EQ(X_{state}, "q") \wedge EQ(X_{next} \circ X_{state}, "q'") \wedge$$
$$\big(EQ(X_{c_i} \circ X_a, X_{next} \circ X_{c_i}) \vee EQ(X_{c_i}, "0")\big)$$

- if $\delta(q) = (IF\ C_i = 0\ THEN\ q_1, ELSE\ q_2)$ then

$$\varphi_q = EQ(X_{state}, "q") \wedge$$
$$\Big(\big(EQ(X_{c_i}, "0") \wedge EQ(X_{next} \circ X_{state}, "q_1")\big) \vee$$
$$\big([X_{c_i} \circ X_a] \wedge EQ(X_{next} \circ X_{state}, "q_2")\big)\Big) \bigwedge$$
$$EQ(X_{c_i}, X_{next} \circ X_{c_i})$$

The idea here is to use the current state to check that the value of the key next is correct according to the transition function δ. With this definition at hand it is straightforward to see that M has an accepting run if and only if the formula φ_M is satisfiable, since every JSON document satisfying φ_M has to contain as a subdocument a JSON document that correctly codes a satisfying run. □

As it is usually the case in PDL-like languages such as XPath [5, 35], the undecidability is caused by the presence of the equality test $EQ(\alpha, \beta)$. Indeed, it was shown several times in the literature that XPath without data tests has EXPTIME-complete satisfiability problem. Most notably, the techniques of [5] can be transferred directly to our case.

Proposition 8 ([5, 35]). *The* SATISFIABILITY *problem is* EXPTIME*-complete for Recursive, non-deterministic JNL (RNJNL) without the* $EQ(\alpha, \beta)$ *operator.*

5. Future perspectives

In this work we present a first attempt to formally study the JSON data format. To this end, we describe the underlying data model for abstracting JSON documents, and introduce logical formalisms which capture the way JSON data is accessed and controlled in practice. Through our results we emphasise how the new features present in the JSON standard affect classical results known from the XML context. While some of these features have been consider in the past (e.g. comparing subtrees [6, 49], or an infinite number of keys [7]), it is still not entirely clear how these properties mix with the deterministic structure of JSON trees, thus providing an interesting ground for future work.

Apart from these fundamental problems that need to be tackled, there is also a series of practical aspects of the JSON format that we did not consider. In particular, we identify three areas that we believe warrant further study, and where the formal framework we propose could be of use in understanding the underlying problems.

MongoDB's projection. Our formalisation of MongoDB's projection opens up several lines of work. First there is the issue of understanding up to what extend can this projection operators be extended without loosing the good algorithmic properties of the operator. Indeed, the projection

in MongoDB is quite limited in expressive power, and does not allow a lot of interaction between filtering and projecting. There are also fundamental questions regarding the expressive power of these transformations, and the possible interactions with schema definitions. Finally, there is the issue of understanding the real need for a JSON-to-JSON query language, and to what extent does JNL, augmented with projection, satisfies these needs.

A standard query language for JSON data. The popularity of the format constantly feeds the creation of new systems capable of dealing with JSON data. More often than not, these systems come up with their own version of a query language. The lack of a standardised query language taxes the JSON environment with the cost of learning several of these languages, makes benchmarking these systems a rather complicated effort, and prevents the community from adopting or refining the fastest querying algorithms available at hand. Of course, the creation of a standard language requires finding common ground amongst the different languages currently available, which would be much easier to do in our framework, using JNL as a common language to compare them. We are strongly convinced of JNL as an interesting core for a future standardisation of a JSON query language.

Streaming. Another important line of future work is streaming. Indeed, the widespread use of JSON documents as a means of communicating information through the Web demands the usage of streaming techniques to query JSON documents or validate document against schemas. Streaming applications most surely will be related to APIs, in order to be able to query data fetched from an API without resorting to store the data (for example if we are in a mobile environment). In contrast with XML (see, e.g., [46]), we suspect that deterministic JNL might actually be shown to be evaluated in a streaming context with constant memory requirements when tree equality is excluded from the language.

Acknowledgements

Reutter and Vrgoč were funded by the Millennium Institute for Foundational Research on Data. Bourhis and Vrgoč were partially funded by the STIC AMSUD project Foundations of Graph Structured Data (Fog). Bourhis was partially funded by the DeLTA project (ANR-16-CE40-0007). Reutter was also funded by CONICYT FONDECYT regular project number 1170866.

References

[1] Jayway JsonPath. `https://github.com/json-path/JsonPath`, 2017.

[2] N. Alechina and N. Immerman. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337, 2000.

[3] AranbgoDB Inc. The ArangoDB database. `https://www.arangodb.com/`, 2016.

[4] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. Relative expressiveness of nested regular expressions. In *Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, Ouro Preto, Brazil, June 27-30, 2012*, pages 180–195, 2012.

[5] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)*, 55(2):8, 2008.

[6] Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, pages 161–171, 1992.

[7] Adrien Boiret, Vincent Hugot, Joachim Niehren, and Ralf Treinen. Deterministic automata for unordered trees. In *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014.*, pages 189–202, 2014.

[8] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and xml reasoning. *Journal of the ACM (JACM)*, 56(3):13, 2009.

[9] Mikołaj Bojańczyk and Paweł Parys. Xpath evaluation in linear time. *Journal of the ACM (JACM)*, 58(4):17, 2011.

[10] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of mongodb queries. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, pages 9:1–9:23, 2018.

[11] Pierre Bourhis, Juan L Reutter, Fernando Suárez, and Domagoj Vrgoč. Json: data model, query languages and schema specification. In *PODS*, pages 123–135, 2017.

[12] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. 2014.

[13] Official BSON specification. `http://bsonspec.org/`, August 2017.

[14] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed xml. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 141–152. VLDB Endowment, 2003.

[15] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[16] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[17] Wojciech CzerwiŃski, Wim Martens, Matthias Niewerth, and Paweł Parys. Minimization of tree patterns. *Journal of the ACM (JACM)*, 65(4):26, 2018.

[18] Wojciech Czerwiński, Wim Martens, Pawel Parys, and Marcin Przybylko. The (almost) complete guide to tree pattern containment. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 117–130. ACM, 2015.

[19] ECMA. The JSON Data Interchange Format. `http://www.ecma-international.org/publications/standards/Ecma-404.htm`, 2013.

[20] Diego Figueira. Satisfiability of downward xpath with data equality tests. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 197–206. ACM, 2009.

[21] Diego Figueira. *Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)*. PhD thesis, École normale supérieure de Cachan, France, 2010.

[22] Diego Figueira. Decidability of downward xpath. *ACM Transactions on Computational Logic (TOCL)*, 13(4):34, 2012.

[23] Python Software Foundation. Python programming language. `https://www.python.org/`, 2016.

[24] Stefan Gössner and Stephen Frank. JSONPath. `http://goessner.net/articles/JsonPath/`, 2007.

[25] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[26] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of xpath query evaluation and xml typing. *Journal of the ACM (JACM)*, 52(2):284–335, 2005.

[27] Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-logic: Logical foundations for json querying. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 137–149. ACM, 2017.

[28] Oscar H Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM (JACM)*, 25(1):116–133, 1978.

[29] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/html/rfc7159`, March 2014.

[30] json-schema.org: The home of json schema. `http://json-schema.org/`.

[31] Katja Losemann and Wim Martens. The complexity of evaluating path expressions in SPARQL. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 101–112, 2012.

[32] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14, 2016.

[33] Zhen Hua Liu, Beda Christoph Hammerschmidt, and Doug McMahon. JSON data management: supporting schema-less development in RDBMS. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1247–1258, 2014.

[34] Christof Löding and Karianto Wong. On nondeterministic unranked tree automata with sibling constraints. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[35] Maarten Marx. XPath with Conditional Axis Relations. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, pages 477–494, 2004.

[36] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.

[37] Marvin L Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. *Annals of Mathematics*, pages 437–455, 1961.

[38] MongoDB Inc. The MongoDB3.0 Manual. `https://docs.mongodb.org/manual/`, 2015.

[39] Frank Neven and Thomas Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *International Conference on Database Theory*, pages 315–329. Springer, 2003.

[40] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Caine*, 2009:157–162, 2009.

[41] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.

[42] OrientDB LTD. The OrientDB database. `http://orientdb.com/`, 2016.

[43] Pawel Parys. Xpath evaluation in linear time with polynomial combined complexity. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 55–64, 2009.

[44] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 263–273, 2016.

[45] Jonathan Robie, Ghislain Fourny, Matthias Brantner, Daniela Florescu, Till Westmann, and Markos Zaharioudakis. JSONiq: The JSON Query Language. `http://www.jsoniq.org/`, 2016.

[46] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming xml documents against dtds. In *International Conference on Database Theory*, pages 299–313. Springer, 2007.

[47] The Apache Software Foundation. Apache CouchDB. `http://couchdb.apache.org/`, 2015.

[48] The Neo4j Team. The Neo4j Manual v3.0. `http://neo4j.com/docs/stable/`, 2016.

[49] Karianto Wong and Christof Löding. Unranked tree automata with sibling equalities and disequalities. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, pages 875–887, 2007.