SCHEMA DECISION TREES FOR HETEROGENEOUS JSON ARRAYS

by

Davis Goulet

B.Sc., University of Northern British Columbia, 2018

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

August 2020

© Davis Goulet, 2020

Abstract

Due to the popularity of the JavaScript Object Notation (JSON), a need has arisen for the creation of schema documents for the purpose of validating the content of other JSON documents. Existing automatic schema generation tools, however, have not adequately considered the scenario of an array of JSON objects with different types of structures. These tools work off the assumption that all objects have the same structure, and thus, only generate a single schema combining them together.

To address this problem, this thesis looks to improve upon schema generation for heterogeneous JSON arrays. We develop an algorithm to determine a set of keys that identifies what type of structure each element has. These keys are then used as the basis for a *schema decision tree*. The objective of this tree is to help in the validation process by allowing each element to be compared against a single, more tailored, schema.

TABLE OF CONTENTS

Abstract ii

Table of Contents iii

List of Tables v

List of Figures vi

List of Acronyms ix

Acknowledgements x

1 Introduction and Motivation 1
- 1.1 Overview 1
- 1.2 Applications 6
  - 1.2.1 NoSQL Databases 7
  - 1.2.2 Similarity Calculations 8
- 1.3 Contributions 9
- 1.4 Thesis Layout 9

2 Overview of Semi-structured Data 11
- 2.1 Types of Data 12
- 2.2 XML 15
- 2.3 Moving Beyond XML 18
  - 2.3.1 Element vs. Attribute 20
  - 2.3.2 Support for Arrays 21
  - 2.3.3 Restricting an Element's Content 22
- 2.4 JSON 22
  - 2.4.1 AJAX 23
  - 2.4.2 JSON Format 24
- 2.5 Schema Documents 28
  - 2.5.1 JSON Schema 28
- 2.6 Popularity of JSON 31

3 Problem Characterization 33
3.1 JSON Arrays and Data Types . . . 34
3.2 Problem Overview . . . 36
3.3 Driving Observation . . . 38
3.4 Schema Decision Trees . . . 39
3.5 Problem Statement . . . 41
3.6 Assumptions . . . 44
3.6.1 Common Structures . . . 44
3.6.2 Existence of Identification Keys . . . 45
3.6.3 Complete Input Data . . . 45
3.6.4 Array of Objects . . . 46

4 Related Works 48
4.1 Related Works Involving JSON . . . 48
4.2 Related Works Involving XML . . . 52
4.3 Related Works Overview . . . 54

5 Solution Details 56
5.1 Syntactic Similarity Scores . . . 56
5.1.1 Tree-edit Distance . . . 59
5.1.2 Disadvantages of Tree-edit Distance . . . 60
5.1.2.1 Similarity of More than Two Trees . . . 60
5.1.2.2 Comparing Unordered Trees . . . 61
5.1.3 Path Distance . . . 61
5.1.4 Path Notation for Nested Arrays . . . 64
5.2 Similarity of a Group . . . 66
5.3 Similarity of a Grouping . . . 67
5.4 Refining the Scoring Criteria . . . 69

6 Algorithm 71
6.1 Starting the Algorithm . . . 74
6.2 GroupNode Algorithm . . . 75
6.2.1 Initializing a New GroupNode . . . 75
6.2.2 Generating Groupings for a GroupNode . . . 76
6.3 SplitNode Algorithm . . . 78
6.3.1 Initializing a New SplitNode . . . 79
6.3.2 Generating Groupings for a SplitNode . . . 80
6.4 Algorithm Remarks . . . 82
6.5 Constructing a Schema Decision Tree . . . 82
6.6 Integrating into JSON Schema . . . 84

7 Evaluation and Analysis 89
7.1 Algorithm Walkthrough . . . 89
7.2 Evaluation . . . 93
7.2.1 Generating the Datasets . . . 95

7.2.2 iTunes Search API Dataset 98
7.2.2.1 Schema Generation Time Comparison 99
7.2.2.2 Schema Validation Time Comparison 100
7.2.3 Open Movie Database (OMDb) API Dataset 101
7.2.3.1 Schema Generation Time Comparison 102
7.2.3.2 Schema Validation Time Comparison 103
7.2.4 Spotify Search API Dataset 104
7.2.4.1 Schema Generation Time Comparison 105
7.2.4.2 Schema Validation Time Comparison 106
7.3 Runtime Complexity Analysis 107

8 Conclusion **110**
8.1 Future Work 111
8.1.1 Validating Identification Keys 111
8.1.2 Dynamic Similarity Threshold 112
8.1.3 Expanded API Study 112

Bibliography **113**

LIST OF TABLES

5.1 A table showing the three scoring criteria being applied to 9 different groupings. The similarity threshold for this example is 0.85. . . 70

6.1 A list of the groups in the best grouping, along with their corresponding *splitKeys* list. 83

7.1 A list of split operations for a single group containing all objects. . 90
7.2 A list of different split operations for one of the groups generated by the *wrapperType* split. 92
7.3 The API Request Endpoint(s) used for each dataset. 95
7.4 An overview of the number of requests made to each API, and the number of objects returned for each request. 96

LIST OF FIGURES

1.1 An illustration comparing an application using a schema and an application not using a schema. 3
1.2 Illustrating the issue with schema design when all elements are assumed to have the same structure. 4

2.1 A valid nested of HTML tags. 16
2.2 An invalid nesting of HTML tags 16
2.3 An element containing attributes. 17
2.4 A sample XML file containing information pertaining to a library. . 19
2.5 An excerpt of figure 2.4 showing one of the book elements. 20
2.6 A modified figure 2.5 showing *id* as an element rather then an attribute. 20
2.7 An excerpt of figure 2.4 showing the *books* element and its content. 21
2.8 A valid XML element showing how the content can contain both text and elements. 22
2.9 A comparison between a traditional web application and an AJAX web application. 25
2.10 A sample JSON document inspired by the iTunes Search API [1]. . 27
2.11 Part of the schema document generated by [2] for the JSON document in figure 2.10. This schema shows how the *results* array is getting interpreted. The array is called *Result* as an auto-generated title referenced in another part of the schema that is not shown. . . 30
2.12 Data from Google Trends [3] showing the relative interest of JSON and XML. 31
2.13 Data from Stack Overflow Trends [4] showing the percentage of questions involving JSON or XML for each month. 31

3.1 Three JSON arrays illustrating the difference between homogeneous and heterogeneous data structures. 34
3.2 Comparison between validating an array through a linear search approach versus validating an array using a schema decision tree. 40

5.1 JSON document in text-based format. 57
5.2 Tree representation of the JSON document in figure 5.1. 60
5.3 Path representation of the JSON document in figure 5.1. 62

5.4 JSON document containing an array. 65
5.5 Three path notations for the JSON document in figure 5.4. 65

6.1 Visualization of the alternating layers of GroupNodes and SplitNodes. 72
6.2 Constructing a schema decision tree for the groups in table 6.1. . . 84
6.3 The section of the resulting JSON schema document containing the definitions for the different sub-schemas tailored to the different types of structure. 86
6.4 Section of the resulting JSON schema document containing a decision node. 88

7.1 Schema decision tree generated from the iTunes Search API dataset. 98
7.2 Comparing the computation time of the iTunes Search API dataset for various similarity thresholds (T) and various input array sizes. 99
7.3 Comparing the validation time of the iTunes Search API dataset for three validation methods. 100
7.4 Schema decision tree generated from the Open Movie Database dataset. 101
7.5 Comparing the computation time of the Open Movie Database dataset for various similarity thresholds (T) and various input data sizes. . 102
7.6 Comparing the validation time of the Open Movie Database API dataset for three validation methods. 103
7.7 Schema decision tree generated from the Spotify Search API dataset. 104
7.8 Comparing the computation time of the Spotify Search API dataset for various similarity thresholds (T) and various input data sizes. . 105
7.9 Comparing the validation time of the Spotify Search API dataset for three validation methods. 106

LIST OF ACRONYMS

API	Application Programming Interface
AJAX	Asynchronous Javascript and XML
AWS	Amazon Web Services
XML	External Markup Language
ECMA	European Computer Manufacturers Association
HTML	Hypertext Markup Language
ISO	International Standards Organization
IETF	Internet Engineering Task Force
JSON	Javascript Object Notation
OOP	Object Oriented Programming
OLAP	Online Analytical Processing
SGML	Standard General Markup Language
SQL	Structured Query Language
UML	Universal Modelling Language
XSD	XML Schema Definition

Acknowledgements

I would first like to express my sincere gratitude to my supervisor, Dr. Alex Aravind, for his support and guidance throughout both my bachelor's and master's degrees. The opportunities and advice he has provided me are invaluable and will continue to benefit me throughout life. I would also like to thank my committee members, Dr. Baljeet Malhotra and Dr. Alia Hemieh, for their advice and insights.

In addition, I would like to thank my friends, Daniel O'Reilly and Rory McClenagan, for helping edit my thesis, and Conan Veitch for always pushing me to pursue new things. Finally, I would like to thank my family for their support and motivation throughout my education.

Chapter 1

Introduction and Motivation

The ever-increasing connectedness of the world has resulted in a vast amount of machine-to-machine communication. Many applications now involve user authentication and the dynamic updating of content; internet-of-things devices constantly upload data collected to a central server, and web APIs have emerged as a way for developers to integrate third party services into their applications. Fundamentally, the communication and transmission of information is an essential component of modern software development.

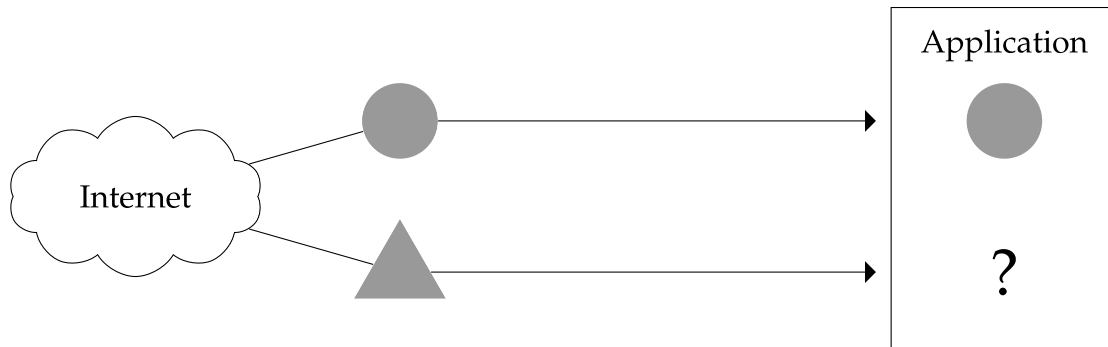
With this, semi-structured data has emerged as the prominent method for structuring the messages used for this communication—largely due to its flexibility and extensibility. This comes as a result of integrating the structural metadata information into the content of the message itself. In this sense, semi-structured data is often characterized as self-describing. Of this type of data, the JavaScript Object Notation (JSON) has become one of the leading formats. This format is primarily based on two data structures: sets of key-value pairs called JSON objects and ordered lists of values called JSON arrays. The true flexibility of the format comes from nesting these structures inside each other resulting in a tree-like structure.

Due to the fact that messages (i.e. JSON documents) can be received from a multitude of different sources (oftentimes sources that you do not even have control over), the need has arisen to first validate the content and structure of a document before processing it. Incorrectly formatted documents will, at best, be detected and handled accordingly; at worst they can cause the entire system to crash or be unknowingly processed and cause logical consistency errors in the application or database.

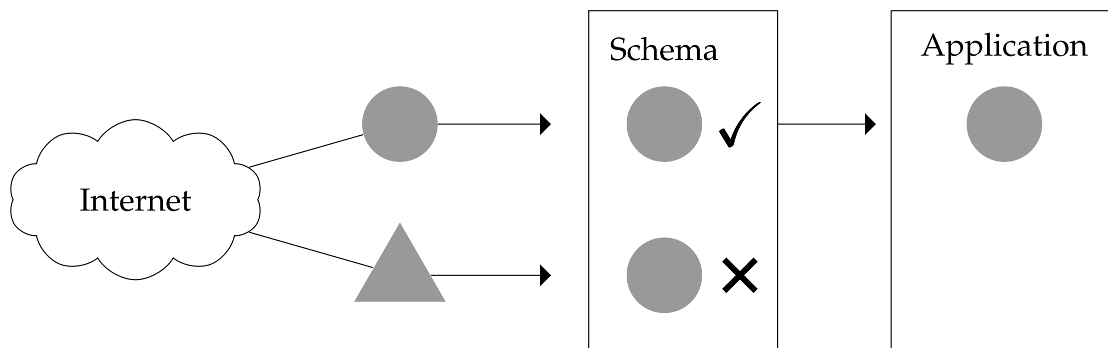
To address this problem, the idea of a schema document emerged as a way of describing what is considered a valid JSON document. Incoming documents are then compared against this schema. If their content and structure match that defined in it, they are passed into the rest of the program to be processed; if not, they are rejected.

Figure 1.1 illustrates the problem of processing incoming documents with and without using a schema. Here, a document's structure is visualised through its shape, i.e. two circles have the same structure, whereas a circle and a triangle have different structures. In figure 1.1a, any document received is processed by the application. Issues may arise if the application is expecting a circle structure but instead receives an ill-structured triangle document. Comparing this to figure 1.1b, documents are first validated against a schema containing information about what is an acceptable structure. Documents satisfying these constraints are passed on to the application.

As creating these schemas are usually tedious and error-prone for humans to do manually, programs have been created to automate the process (examples being [2] and [5]). These programs work by accepting one or more JSON documents as input and generate a schema based on the existing structures found in them. While these programs do generate valid schemas, they run into ambiguity issues when documents contain an array of JSON objects with differing structures.



(a) An application expecting messages to have a circle structure.

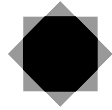


(b) An application that preemptively filters out non-circle structures.

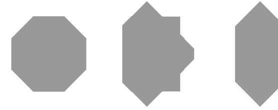
Figure 1.1: An illustration comparing an application using a schema and an application not using a schema.

Consider the array of shapes in figure 1.2a, where each element has one of two possible structure types: square or diamond. Comparing the two types, some parts overlap (the middle portion), whereas some parts only exist in specific types (the points of the diamond or the corners of the square). If a schema for this array were to be generated using current automatic tools, the schema in figure 1.2b would likely be the result. Here, the portion of the structure that is common to all shapes is specified as required (denoted in black), with all other parts being specified as optional (denoted in grey). The reason behind this schema in particular being generated is that current automated tools work off the assumption that all array elements should have the same structure, and any deviations from that structure should be considered optional. These tools do not consider the possibility that

(a) Array of elements consisting of diamond/square structures.



(b) Generated schema for the array in (a).



(c) Shapes that also satisfy the ambiguous schema in (b).

Figure 1.2: Illustrating the issue with schema design when all elements are assumed to have the same structure.

elements in an array may have different structures that are not related.

While this assumption always results in a valid schema, it also leads to unintended side effects in what other structures the schema also accepts. For example, the shapes outlined in figure 1.2c also satisfy the schema in 1.2b. They all contain the portion of the schema that is specified as required with any other components being part of the optional portion. The main issue here is that the schema in figure 1.2b does not specify the relationship between the different optional components—resulting in ambiguity. A better option would be to generate two schemas; one schema for diamonds, and one schema for squares. An element in an array would then be valid only if it satisfies one of the possible schemas.

One potential solution to this problem could be to apply cluster analysis on the elements of an array. The result of this clustering process is a set of groups, where each group contains elements with related structures. A schema can then be generated based on each group. This problem has previously been explored in literature; however, previous work has largely only looked at the scenario of clustering a set containing any JSON documents.

Two downsides exist with this solution. First, the number of groups generated may not match the number of different types of structures that exist. If two types

of structures are very similar to each other, they may get grouped together rather then be placed into their own groups. Second, an increase in the time it takes to validate a document may occur. This increase is due to now potentially having to compare each array element against multiple schemas to determine which one it satisfies. For example, if there are 100 elements in an array and 10 possible types of structures, a potential 1000 schema validation attempts may occur.

This thesis introduces a different approach to clustering the elements of a JSON array. We base this approach off the observation that a JSON array is designed to be processed as a single entity, i.e. a program takes a JSON array and iterates over each element—processing it. When all array elements have the same structure, this is a relatively simple program, as each element is processed in exactly the same way. When elements have different structures however, the program becomes much more complex. Different structures may now require unique ways of being processed. A common method of specifying an element's structure is to include a set of keys within each element whose values identify the type of structure. We call such keys *identification keys*. A program can then use these keys to determine how it should process each element. An example of such a key might be a version number, where the keys value can be used to determine which version of a structure an element has.

Using this idea, we cluster the elements of a JSON array by choosing a key common to all the elements, and partition the elements into a set of groups based on the value of the key. The main advantage of this approach over previous clustering methods is that future elements can be assigned to a cluster in constant time. To do so, a lookup is done on the elements identification keys and, based on their values, they are assigned to one of the clusters. In the context of schemas, this method allows each element to be validated against a single schema versus potentially being compared against multiple schemas. To determine which schema an element

should be compared against, we introduce the concept of a *Schema Decision Tree* based on decision-tree classifiers. With this, each element starts at the root node of the tree and moves to a child node based on the value of the key specified at the current node. This process repeats until the element eventually reaches a leaf node containing the schema it should be compared against.

To construct this schema decision tree, this thesis designs and implements an algorithm that takes a JSON array and recursively partitions the elements based on an operation we introduce called *splitting*. This operation takes a key that is common to all elements and partitions them based on what value they have for that key; all elements with the same value are placed in the same group, and elements with different values are placed in different groups. As more than one key may be in common to all the elements, we measure how good a split operation was by measuring the structural similarity of the resulting groups. This is done through a calculation based on the Jaccard Similarity Index previously discussed in literature. Furthermore, this algorithm works recursively, as groups generated from one split operation may have to be partitioned again. This is to account for the existence of multiple identification keys.

1.2 Applications

The main motivation for the work done in this thesis is to improve schema generation for JSON arrays containing JSON objects with differing structures. Thus, the main application is any problem already involving schema documents. The next two subsections outline two specific applications that would benefit from better schema generation.

1.2.1 NoSQL Databases

One issue with JSON (and semi-structured data in general) is that it can not easily be stored in traditional relational databases. The reason for this is twofold. First, its irregular structure means that different documents can contain different keys. To account for this, a relational table column would have to be created for each possible key. Null values would then be used for documents not having that key. Approaching database design this way is not recommended however [6]. Second, JSONs tree-like structure does not match the flat table structure found in the relational model. Existing techniques for storing JSON in relational databases involve either storing the entire document as a single string in one of the columns or breaking up the tree-like structure and storing each layer separately along with new fields to keep track of parent-child relationships [7].

To address this issue, another kind of database—commonly known as a NoSQL database—quickly gained in popularity. The defining feature of this type of database is that they are able to easily store semi-structured data, as well as provide features tailored to interacting with / managing it. Of this kind of database, MongoDB is one of the most prominent. It is built around the concept of storing semi-structured data in *documents* contained in *collections* [8, 9]. Compared to relational databases, this is analogous to storing *rows* in *tables*; the major difference is that a collection can contain documents with differing structures, whereas a relational table requires a rigid structure with a set number of specific values. For this reason, collections in MongoDB are commonly referred to as having a dynamic schema [9].

While this does give great flexibility in how data can be stored, it also leads to challenges when designing programs and queries to interact with a collection. This is because the structure of all the documents in the collection has to be considered. For example, if a document was entered into a collection with missing

fields, queries designed to extract documents having those fields would pass over this document. This is by virtue of the fact that the query requirements did not account for this ill-structured document. For this reason, MongoDB has a feature that allows a collection to have a schema defined for it. This schema acts like a filter to the collection and only allows documents whose structure meets certain criteria to be entered into it. Improving the schema generation process would be beneficial in preemptively detecting ill-structured documents.

1.2.2 Similarity Calculations

Another application for better schema generation can be found in the problem of calculating the similarity between JSON documents. The goal of a similarity calculation is to take some number of documents and generate a numerical representation based on some notion of similarity. One common way of doing this is to create a mapping between related keys found in the documents. The larger the mapping, the more similar the documents are. A challenge with this approach is how to handle the scenario of two JSON arrays, as the number of elements in each array may differ. Not taking this into account can greatly throw off any calculations even though all the elements in the array have the same structure. To solve this problem, previous literature first generated a schema for each of the documents as a way of reducing the elements of an array. Similarity calculations were then applied to the schemas instead of the documents themselves. Improved schema generation could be applied here to better capture the different types of structures found in each array.

1.3 Contributions

The work in this thesis has four main contributions.

1. We give a comprehensive overview of XML and JSON. This includes looking at the general syntax, as well as the history and motivation behind their creation.
2. We define a new problem, related to cluster analysis, based on the idea of partitioning elements by identification keys.
3. We introduce the concept of a *Schema Decision Tree* to help validate JSON arrays, and we show how this method of partitioning can be used to construct this tree.
4. We design and implement an algorithm capable of determining the best partition from which to construct the schema decision tree.

1.4 Thesis Layout

The rest of this thesis is organized as follows. To begin, Chapter 2 overviews what semi-structured data is and outlines the syntaxes of XML and JSON—the two major formats. The history behind each of these is also discussed, along with issues present in XML that lead to the creation and adoption of JSON. This chapter concludes with a comparison between the popularity of the two formats. Chapter 3 then characterizes and defines the problem addressed in this thesis—namely, better schema generation for heterogeneous JSON arrays. We explain the main observation that this work is based off of and introduce the concept of a schema decision tree. Based on the problem defined in this chapter, chapter 4 then examines existing literature involving both JSON and XML. Next, chapters 5 and 6 look at the

solution we have created. Chapter 5 first discusses the background information that the solution is based on. This includes how we are measuring the similarity between JSON objects, as well as a set of criteria for what is considered the best partition. Chapter 6 then presents an algorithm to compute this partition. To evaluate our algorithm, chapter 7 first walks through the algorithm being applied to a simplified dataset. We then analyze the time it takes the algorithm to run for varying array sizes, as well as analyze how long it takes arrays of varying sizes to be validated against the resulting schema decision tree. An analysis of the runtime complexity is also performed. Finally, this thesis is concluded in chapter 8 along with a discussion of possible future directions for this research.

Chapter 2

Overview of Semi-structured Data

A major component of nearly all software applications involves some variation of storing, manipulating, or transmitting data. Take the example of a banking web application that makes a request to a server to increase the balance of an account. In just a single application, the same data is likely being modelled in three different ways. First, there is the web application itself within the user's web browser that has an internal representation of the data it needs. For it to then send a request to the server, the application has to structure the relevant data in a format that both the client and server will understand. This involves taking the internally stored data and representing it in a format that can be serialized for platform independence (e.g. a memory pointer cannot be passed in a request since the receiving end does not have the data stored at the same memory location). The server then processes the request by manipulating the data stored in a structured database before sending a response back.

Data is so important to computer science that entire subfields are dedicated to studying it in different ways. Database management looks at efficient ways of organizing, storing, and retrieving data [6, 10]. Data science looks at methods of extracting patterns and information from potentially unorganized or messy data

[11]. Big data looks at how to process and analyse large data sets [12]. Parallel systems looks at methods for processing data simultaneously [13]. Fundamentally, computer science is a discipline based on data.

2.1 Types of Data

The category of data can be divided into three main types: structured, unstructured, and semi-structured [14, 15].

Structured data means that the data follows a specific rigid format that has been predetermined [16]. Having this fixed structure results in data that is ideal at being stored in a database [14]. In a relational database for example, each row of a table consists of an n-tuple where the ith element of the tuple has one of the values specified by the ith column of the relation [6, 10]. This row has a rigid structure in the sense that it has exactly n elements arranged in a specific order. Swapping two elements of the row changes their meaning, as each column of the relation also has a specific real-world context behind what it represents. As all values for a column follow the same format, the meta-data information associated with it can be stored a single time outside the table itself. This is in contrast to storing a copy of it in each row individually. In the context of SQL, this is done through a data definition language that defines the columns of a table, its connection to other tables, constraints, data types, etc. [6].

The second type of data is unstructured. This type of data differs from structured data in that there is no predefined structure dictating what the data consists of or how it is arranged [12]. Up to an estimated 80% of all data falls under the unstructured category [17] due to the prevalence of online media formats such as video, text, photo, and audio. While these formats all have structure dictating how they encode data in binary, there is no structure regarding what their contents ac-

tually consists of. To illustrate this, consider the example of a plain text document [14]. Given a paragraph, specific information may be contained within a sentence; however, there is nothing outlining where the sentence occurs in the paragraph or what words were used in it. The same piece of information may even be represented in two different sentences using completely different words.

Finally, semi-structured data falls in between structured and unstructured data. While structured data showcased the extreme of rigid guidelines, and unstructured data showcased the extreme of no guidelines, semi-structured data falls in between by providing structure through a more flexible format [14, 15]. It is because of this flexibility that Serge Abiteboul refers to it as irregular data [14]. The flexibility of semi-structured data comes from including the structural information in each instance itself. This way, two instances can have differing structures. As long as each instance specifies where the common data is located, they can be treated the same. For this reason, semi-structured data is commonly referred to as self-documenting [15, 14, 16]; the information needed to interact with the data is contained within the data itself. This differs from structural data, where the structural information is stored outside each instance of the data.

To illustrate the importance of semi-structured data, consider the scenario of having to integrate data related to bank accounts coming from two different banking sources [14]. Creating a single structured data format that encompasses both sources is a challenging problem primarily for two reasons. First, while the general information related to a bank account is likely to be the same between sources (eg. account owner, account balance, etc.), some data will differ. One bank, for example, may include the field *time-since-last-accessed*, whereas the other bank may not. Creating a single data format to encompass both banking sources leads to two options. Option one is to include all possible fields in the common format, and set a field as null for any data instance that does not have it. Contrarily, option two is

to leave the field out and discard it for data that has it. Neither scenario is ideal as the two data sources are inherently different.

An alternative solution to this problem would be to use semi-structured data. With this, each data source could maintain its original structure without the need to combine them together. Two popular formats for representing semi-structured data are XML and JSON; these are discussed in more details in the following sections.

This banking example also showcases one of the biggest areas where semi-structured data is used; that is, in situations where data is transmitted between machines over a network [14, 16]. In order for two machines (call them A and B) to communicate, they have to agree on a common format (or "contract") for how messages will be sent between them. Machine A needs to know how to send data that machine B can correctly interpret. Likewise, machine B needs to know what data is included in a message from machine A and how to process it. One way to define this communication could be through a hard contract. With this, the exact structure of a message is set, and both machines know how to construct a message and extract the information from it. An example of such a contract might be that a message consists of 20 bits evenly split between two 10 bit values. Now suppose that machine A wants to add a new field to the message so that it can also send it to other machines. As machine B is expecting a message with a very specific structure, any attempts by machine A to change the structure will result in machine B failing. Thus, machine B would also have to be updated to support the new format even though the new fields do not affect it. If this communication contract had been outlined using semi-structured messages, machine A would be able to include additional fields without it affecting machine B. All the information machine B needs to work off of would still be included in each message, and it could ignore the parts it does not need.

2.2 XML

One of the first major semi-structured formats to emerge was the Extensible Markup Language (XML). The goal behind this language was to provide a way of arranging data into structures along with annotations about the data [18]. The creators also wanted a general purpose format that could be applicable to many types of applications. In addition, it should also be fast and easy to create concise documents that both humans and machines could understand [18, 19]. Particular emphasis was also put on using it to transmit information over the world wide web [18].

To discuss the origin of XML and why it became popular involves first discussing the history of the web. When Tim Berners-Lee was developing the world wide web around 1989, he needed a way of marking up documents with hypertext (text containing links to other resources that are directly available from the link itself—usually by clicking [20, 21, 22]). In particular, he needed a format that would be independent of a particular computer, as previous hypertext packages were too computer-specific to be used in a global network run on different types of machines [23].

To solve this problem, he created a computer-independent document format called the Hypertext Markup Language (HTML). This format had primarily two functions: (1) a way of specifying hyperlinks in text, and (2) a way of formatting information for visualization purposes. With this format, an application (ie. web browser) could be created for any computing system to take an HTML file and display it to the user. Even today, the HTML format is the ubiquitous way of displaying information in a web browser.

When Berners-Lee was in the process of creating HTML, rather than inventing his own format, he chose to base it on a previously established standard called the Standard Generalized Markup Language (SGML) [23]. This was a standard from the International Standard Organization (ISO) that outlined a format for defining

markup languages. SGML allowed for documents to be created that separated the content of the document from annotations about the content [24, 25]). For example, the title of a document serves a different purpose then a paragraph; by including this meta-information in the document itself, whoever/whatever is processing the document can be aware of what information falls in what category and, as a result, act accordingly.

The basic structure of an HTML document (and thus an SGML document) consists of a series of *elements* [25, 23]. For the most part, an element has a structure defined by

1. an initial start-tag in the format of <TAG-NAME >,
2. followed by some arbitrary content,
3. and closed by an end-tag in the format of </TAG-NAME >. Further, the TAG-NAME of the end-tag must be identical to the TAG-NAME of the start-tag for the element to be valid.

Elements can also be nested inside each other as long as each start-tag has a corresponding end-tag, and that all inner elements are closed before the outer elements end-tag [26, 25, 24]. For example, figure 2.1 is a valid nesting since the inner tag is fully contained within the content of the outer tag. Figure 2.2 on the other hand is not a valid nesting since the inner tag is started within the content of the outer tag but not closed.

```
<OUTER-TAG> <INNER-TAG> ... Content ... </INNER-TAG> </OUTER-TAG>
```

Figure 2.1: A valid nested of HTML tags.

```
<OUTER-TAG> <INNER-TAG> ... Content ... </OUTER-TAG> </INNER-TAG>
```

Figure 2.2: An invalid nesting of HTML tags

Additionally, elements can have *attributes* included inside a start tag for data that is related to the element [26, 25, 24]. For instance, a title tag may have a font size attribute specifying how large the title should be. Figure 2.3 shows an example of an element whose start-tag contains one attribute.

```
<INNER-TAG attribute-name="value"> ... Content ... </INNER-TAG>
```

Figure 2.3: An element containing attributes.

One of the main distinctions between HTML and SGML involves what is considered a valid tag name for an element [23]. In SGML, there are no restrictions for a tag's name except that it has to consist of characters, begin with a character or underscore, and contain no spaces; any name satisfying these rules is considered valid. HTML on the other hand restricts the name of a tag to a small set of specific strings [26, 23]. The reason for this restricted set is that, in HTML, a tags name has external influence behind how it is visualized in a web browser. For example, the < h1 > tag represented a large heading; < p > represented a paragraph of text; < a > represented an anchor containing a hyperlink. HTML had to restrict the set of valid tag names in its specification because the companies in charge of the web browsers had to program this external information into them.

As the world wide web became more popular over time, a need started to arise for a better format for structuring data. HTML was too limited by its restricted set of tags to account for all the different data scenarios [23]. Further, having each HTML tag associated with some aspect of visualization was not needed for the purpose of strictly structuring data. SGML, on the other hand, was too complex in what was considered valid syntax [19, 27, 28]. As a result, many software implementations only ever accounted for a subset of SGMLs entirety. This led to many implementations not being fully compatible with each other [27]. In essence, both

HTML and SGML were not suited to being general purpose data formats.

To find a solution, a group known as the *SGML Editorial Review Board* (later known as the *XML Working Group*) was formed in mid 1996 with the aim of creating a "slimmed-down SGML" [19, 27, 18]. This group consisted of members such as Jon Bosak from Sun Microsystems, Tim Bray of Textuality and Netscape, Jean Paoli of Microsoft, Steve DeRose, and other prominent members who either were experts on SGML or came from various companies in the field [18]. Many of these individuals had already been working on, and even proposed, more concise versions of SGML [19, 28]; however, these were contained to more specific applications and never resulted in widespread adoption or the creation of new standards. On February 10th, 1998 (about a year and a half after the group's formation), the first version of the XML specification was released [18] and quickly gained in popularity.

Like HTML and SGML, an XML document for the most part consists of a series of elements each containing a start-tag and corresponding end-tag. Figure 2.4 shows an example of a typical XML file. Here, the *library* element is known as the root element since everything else is nested inside it [18]. The content of this element then consists of the *location* and *books* elements. These in turn each have more inner elements. Further, each *book* element contains an attribute called *id*. An important point to also note is that an element can have multiple inner elements with the same name; in the case of *books*, three elements having the name *book* are nested within it.

2.3 Moving Beyond XML

From its initial 1.0 specification release, XML would go on to become the dominant semi-structured data format in the software community [19]. As a result, an entire

```
1  <library>
2    <location>
3      <city>Vancouver</city>
4      <address>123 Park Way</address>
5    </location>
6    <books>
7      <book id="472">
8        <title>XML in a Nutshell</title>
9        <availability>true</availability>
10     </book>
11     <book id="293">
12       <title>The SGML FAQ Book</title>
13       <availability>false</availability>
14     </book>
15     <book id="142">
16       <title>The SGML Handbook</title>
17       <availability>true</availability>
18     </book>
19   </books>
20 </library>
```

Figure 2.4: A sample XML file containing information pertaining to a library.

ecosystem of tools has been developed around it. Major tools include: XPath (a query language for finding data within an XML file [29]), XML Namespaces (a way of preventing collisions between element names in different XML files [30]), and XML Schema (a way of describing the structure of an XML document for validation purposes [31]).

Regardless of its popularity, not everyone in the community felt that XML was the best format for structuring data. One of the main issues with XML stems from the fact that it is first and foremost a markup language for document formatting rather then a data format. This distinction can be found in many of the core features of XML. Three major ones are discussed next.

2.3.1 Element vs. Attribute

In the context of markup languages, having a distinction between an element and an attribute is logical, as attributes can be thought of as properties of the element. In HTML for example, an element defines the general way the content will be displayed; attributes can then be added to the element to influence things like font color, font size, margins, etc.

In the context of only structuring data, this distinction is not nearly important, as for the most part, elements only serve to arrange data into logical units. To illustrate this difference, consider figures 2.5 and 2.6 where *id* is considered an attribute in one and an element in the other.

```
1 <book id="142">
2     <title>The SGML Handbook</title>
3     <availability>true</availability>
4 </book>
```

Figure 2.5: An excerpt of figure 2.4 showing one of the book elements.

```
1 <book>
2     <id>142</id>
3     <title>The SGML Handbook</title>
4     <availability>true</availability>
5 </book>
```

Figure 2.6: A modified figure 2.5 showing *id* as an element rather then an attribute.

Both cases make logical sense for modelling the data. Because of this, there is much debate in the community over what data should be placed in an attribute versus an element. One option is that elements should be "the essential material that is being expressed or communicated in the XML" [32]; however, there is no

hard rules, and much of the choice is left up to the developer. For example, the *title* and *availability* elements could also be modelled as attributes of the *book* elements. However, doing this would decrease the readability of the document as all information is now being stored in a single element.

2.3.2 Support for Arrays

The second issue with XML primarily being a markup language is that there are no native implementations of common data structures such as arrays. Figure 2.7 shows how a makeshift array can be constructed in XML. An outer element is created to act like a wrapper for the array elements. Each inner element then represents one of the array values. While this method does allow an array to be encoded, it also requires any people/tools using the document to have this background information. To illustrate this point, suppose the *books* element only had a single inner *book* element. By just looking at the data, it would be impossible to determine if *books* was an array or not without knowing in advance.

```
1  <books>
2    <book id="472">
3       <title>XML in a Nutshell</title>
4       <availability>true</availability>
5    </book>
6    <book id="293">
7       <title>The SGML FAQ Book</title>
8       <availability>false</availability>
9    </book>
10   <book id="142">
11      <title>The SGML Handbook</title>
12      <availability>true</availability>
13   </book>
14 </books>
```

Figure 2.7: An excerpt of figure 2.4 showing the *books* element and its content.

2.3.3 Restricting an Element's Content

Finally, XML places no restrictions on what the content of an element can contain. For the most part, an element's content consists of either a string or a sequence of inner elements as shown in figure 2.4; however, this is not a restriction the language places on the data, and as a result, a mixture of both can occur and is considered valid. Figure 2.8 shows a valid XML element that contains a mixture of text and elements as its content. Again, this syntax is useful in a markup language to annotate text that appears mid-sentence. As a data format though, it is not needed.

```
<location>Located in <city> Vancouver </city> Canada </location>
```

Figure 2.8: A valid XML element showing how the content can contain both text and elements.

2.4 JSON

In 2001, Douglas Crockford was working at a company he co-founded called State Software. They were attempting to build a web framework for creating dynamic web pages—an idea that was just staring to emerge in the software community at the time. Part of this project involved needing a way of structuring data when transmitting it between the web browser and a server [33]. When considering XML however, he found that it was an inefficient format because of its lack of data structures that were common to nearly all programming languages [34, 35]. Instead, he chose to create a format based on the notation Javascript uses for its data structures. In this sense, the JSON format is a subset of the ECMA-262 Specification that Javascript itself is based on [36, 33, 34, 37, 38]. One of the main benefits of this approach is that it would make parsing JSON very easy on the browser side. When approaching companies about their framework, they were reluctant to use the for-

mat because there was no specification attached to it [34, 35, 33]; thus, Crockford gave it the name Javascript Object Notation (JSON) and created a simple website (json.org). This website gave a description of what JSON was and provided a simple grammar outlining the format.

Crockford claims to have “discovered” JSON rather than ”invent” it, as the main ideas had already existed in various formats before him. For instance, he knew of Netscape using similar ideas with different syntax as early as 1996 [35]. Crockford’s contributions mainly came in formalizing a specification for it and giving it a name and website. Besides that however, he did not do any promotion of the format such as trying to get it adopted at other companies.

JSON’s popularity mainly grew as people came across it and used it for its simplicity. Because of its small grammar, JSON parsers were quickly created in all the main programming languages. Further, its two main data structures (sets and arrays) exist in some form in nearly every programming language; this made it very easy to encode / decode information [37]. However, Crockford believes the biggest contributing factor behind its popularity was likely due to its use in AJAX [35, 33].

2.4.1 AJAX

During the early 2000’s, a series of new tools and techniques were starting to emerge allowing developers to create more dynamic web applications—similar to what State Software was doing in 2001. The collective name for this new approach to web development would become known as *Asynchronous Javascript and XML* (AJAX) as a result of a blog post by Jessie James Garrett discussing the new trend [39].

The goal behind AJAX was to improve the user experience by having the web page dynamically update with new information received from the server. This

was compared to the previous approach where an entirely new web page would be requested for each UI update [39, 40]. For example, Gmail could now refresh the users inbox seamlessly while the user was reading another email; this gave a better experience in comparison to the user having to manually refresh it and wait for a new web page to be displayed.

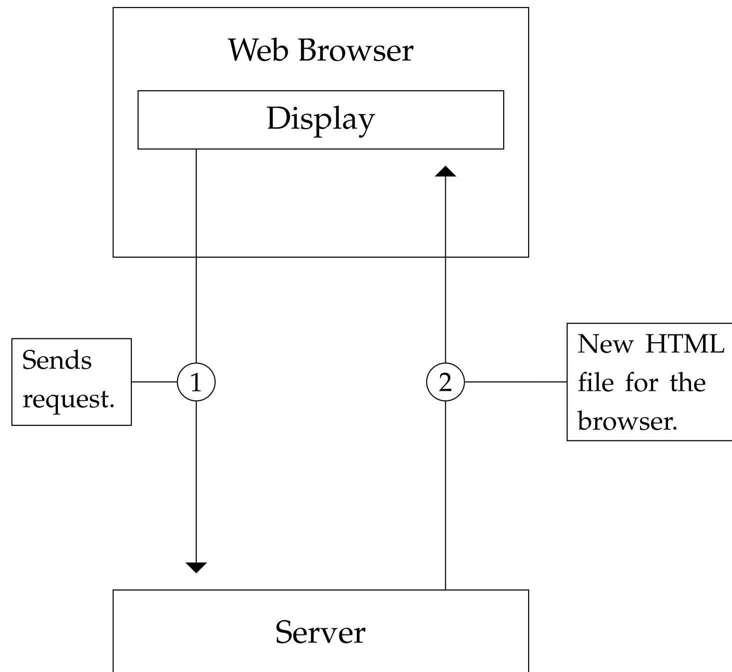
The basic idea for achieving this was to create a middle man that would act like a liaison between the web browser and server. Figure 2.9 shows the difference between the traditional approach (figure 2.9b) and the AJAX approach (figure 2.9a).

In the traditional approach, whenever the web browser wants new information, it sends a request to the server. The server then returns an entirely new web page and possibly all of its attached resources (CSS, JavaScript, images, etc) if they were not cached. Comparing this to the AJAX approach, when the browser makes a request for new information, it instead calls a JavaScript function that initiates an asynchronous request. This request being asynchronous is what allows the web page to still remain active while the request is being processed. Since the web page will be dynamically updated, the server now only has to return the required data using a format like XML or JSON. The asynchronous request then receives the data and manually updates the existing web page's HTML code.

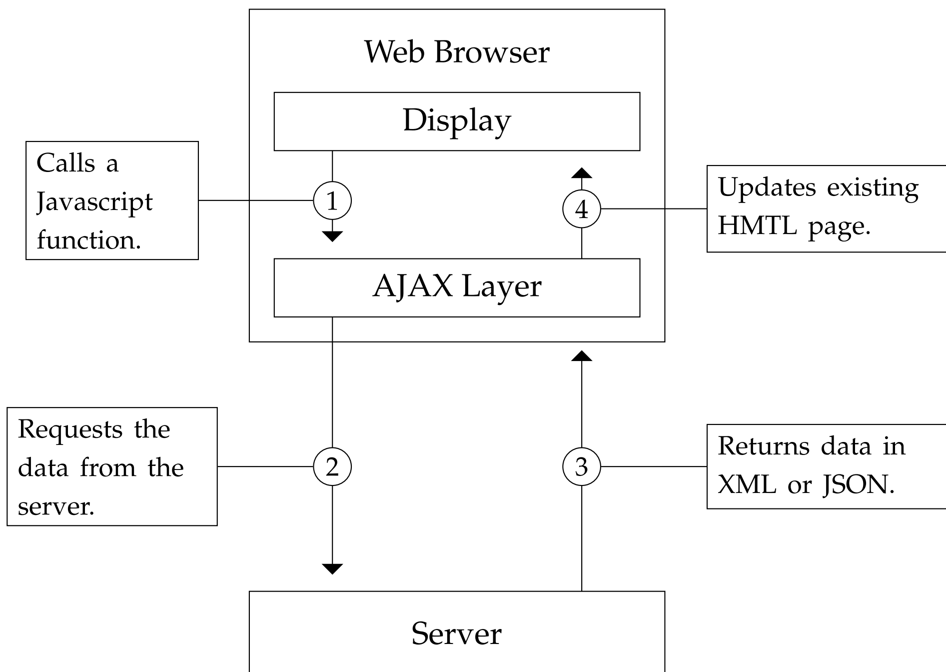
While AJAX was originally used with XML (the 'X' in AJAX even standing for XML), JSON has quickly gained in popularity as a replacement. A large part of this is due to the synergy JSON has with Javascript and its shared notation for data structures [33].

2.4.2 JSON Format

JSON is a semi-structured data-interchange format, primarily used for moving information between different systems. Like XML, JSON is a platform independent format. This means that different systems can use it regardless of what operating



(a) Traditional Web Application



(b) AJAX Web Application

Figure 2.9: A comparison between a traditional web application and an AJAX web application.

system is running, what software is interacting with it, and what programming language the software was written in [35, 37, 41, 38].

Fundamentally, the format is based on two constructs [37].

1. *JSON Object*: A set of key-value pairs enclosed within two curly braces. The order of key-value pairs does not matter. Further, each key must be unique among all other keys in the object, and a value is accessed by performing a lookup on the corresponding key.

2. *JSON Array*: An array of values enclosed within two square brackets. Compared to the JSON object, the order of elements does matter, and a value is accessed through a lookup on its array index.

While a key is limited to only a character string, a value can be any of the available data types. This includes the simple data types listed below [37, 38] in addition to the two complex data types (JSON objects and JSON arrays) discussed above.

1. Integer: Signed decimal number which may include scientific notation.

2. String: A sequence of zero or more Unicode characters surrounded by double quotes.

3. Boolean: Either *true* or *false*.

4. Null: The empty value designated by *null*.

This ability to nest objects and arrays within each other is what gives JSON its tree-like structure so common to semi-structured data. A JSON document then consists of either a single JSON object or JSON array as the root element.

Figure 2.10 contains an example of a typical JSON document. Here, *date*, *successful*, and *resultCount* are examples of key-value pairs with simple data types;

metadata is a key-value pair whose value is another JSON object, and *results* is an array containing two JSON objects as elements. It's important to note that arrays in JSON are heterogeneous meaning that all elements in the array do not need to have the same data type [38].

```
1  {
2    "metadata": {
3      "date": "11/17/19",
4      "successful": true,
5      "resultCount": 2
6    },
7    "results": [
8      {
9        "kind": "song",
10       "collectionId": 585972750,
11       "isStreamable": true,
12       "trackTimeMillis": 241562
13     },
14     {
15       "kind": "podcast",
16       "collectionId": 394775318,
17       "genres": ["Design","Podcasts"]
18     }
19   ]
20 }
```

Figure 2.10: A sample JSON document inspired by the iTunes Search API [1].

The contents of the JSON document in figure 2.10 are inspired by the results returned from the iTunes Search API [1] [1]—a web API that returns information on products available in the iTunes store. The main component of the APIs response is an array of JSON objects, where each object of the array represents a different product. As iTunes sells more then one type of product (examples being songs,

[1]Actual results from the API were not included in the thesis due to their sizeable length taking up multiple pages. Figure 2.10 is instead inspired by the results from the API to showcase the problem tackled by the thesis in a real world scenario.

movies, podcasts, etc.), the structure of each object depends on what type of product it is trying to model. For example, objects representing songs have a different structure compared to objects representing podcasts.

2.5 Schema Documents

While the main idea behind semi-structured data formats is that their structure is flexible in what information can be included and how it is arranged, a need has still arisen to put restrictions on what classifies a valid structure for within a certain application. For example, consider the scenario where a document is passed around to different applications in a distributed system, where each application is able to modify the document before passing it on to the next application. Once all the modification have been completed, it would be very beneficial to have a way of verifying that all the changes have still resulted in a document that the program knows how to process. The document may syntactically be valid, but its structure may not be what the program processing it is expecting. If the program were to process the document, it could potentially produce logical consistency errors or even crash.

To combat this, the idea of a schema document emerged as a way of specifying what is considered a valid structure for a specific application. Other semi-structured data can then be compared against this schema to see if their structure and content match that defined in it. If it does match, the document is considered valid, and otherwise, it is considered invalid [31, 42].

2.5.1 JSON Schema

The *JSON Schema Specification* is a working specification that outlines a format for creating schema documents based on the JSON format. Each schema document

describes the structure/content that it considers valid. Other JSON documents can then be compared against the schema to see if their structure and content match. If it does, the document is considered valid. The JSON Schema Organization is the organization behind the widely accepted implementation [43]. They are currently on their ninth draft as of September 2019 with the goal of being standardized by the Internet Engineering Task Force (IETF) [42].

Figure 2.11 shows part of the schema generated for the JSON document in figure 2.10—in particular, for the objects in the *results* array. What this schema outlines is that an array object is only valid if all of the keys within it match one of the five specified. If a key is present in the object, then the datatype of its corresponding value must also match the datatype listed in the schema. Finally, every object has to have the keys listed in the required section (namely *collectionId* and *kind*) with the others being optional. If an object breaks any of these rules, then the entire JSON document is rejected, as its structure does not conform to the schema. This schema was generated using the Quicktype automated program [2]. This program is listed as one of the generators on the JSON Schema Organization's website [42].

```
...
"Result": {
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "kind": {
            "type": "string"
        },
        "collectionId": {
            "type": "integer"
        },
        "isStreamable": {
            "type": "boolean"
        },
        "trackTimeMillis": {
            "type": "integer"
        },
        "genres": {
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    },
    "required": [
        "collectionId",
        "kind"
    ],
    "title": "Result"
}
...
```

Figure 2.11: Part of the schema document generated by [2] for the JSON document in figure 2.10. This schema shows how the *results* array is getting interpreted. The array is called *Result* as an auto-generated title referenced in another part of the schema that is not shown.

2.6 Popularity of JSON

Since its initial release in 2001, JSON has slowly become one of the two dominant formats (the other being XML) for semi-structured data. Further, JSON has likely overtaken XML as the most popular format based on internet search data.

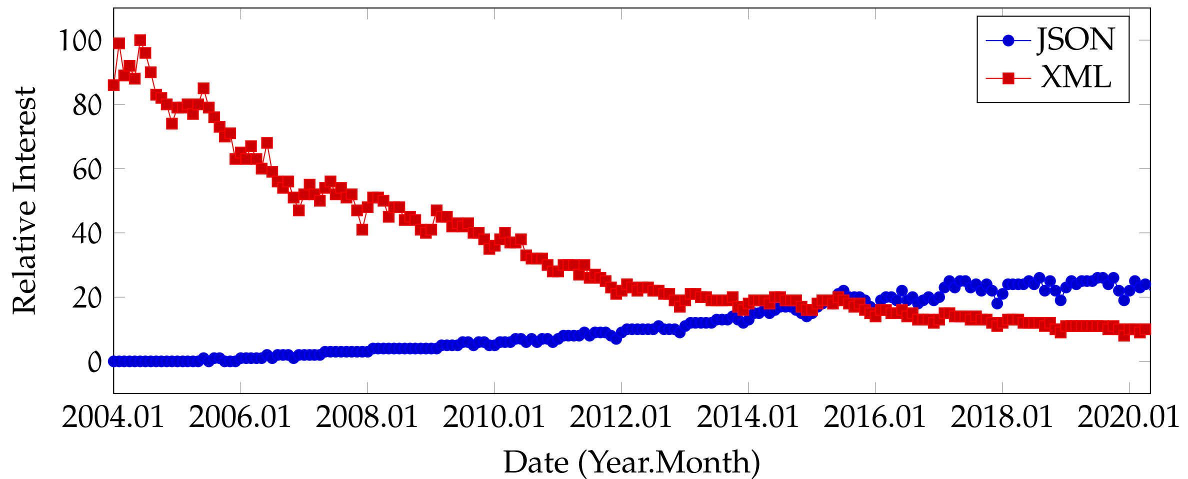


Figure 2.12: Data from Google Trends [3] showing the relative interest of JSON and XML.

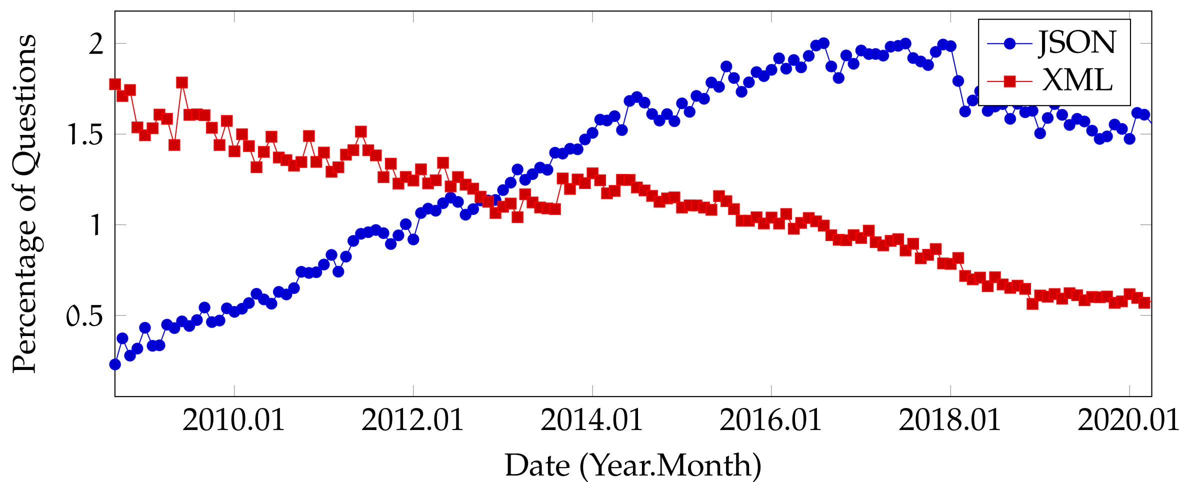


Figure 2.13: Data from Stack Overflow Trends [4] showing the percentage of questions involving JSON or XML for each month.

Figure 2.12 shows the popularity of the JSON and XML search terms on Google Search since 2004. A score of 100 represents the peak popularity of either term, with all other points being in proportion to it. Likewise, figure 2.13 shows the percentage of questions posted to Stack Overflow involving the JSON or XML question tags. In general, what these figures show is that JSON has slowly been gaining in popularity since its creation, while XML has been decreasing in popularity.

As the data does not go back before the creation of JSON, conclusive results cannot be drawn from it. XMLs large decrease in popularity may have been caused by influences not related to JSON. For example, in the early 2000s, XML could have seen exceptionally high popularity due to its novelty and the state of the world wide web. The sharp decrease in figure 2.12 could then be a result of new trends emerging, and shifting interests in the development community. Support for this interpretation also comes from comparing the relatively small interest of JSON in the late 2010s with XML in the early 2000s; even taking the combination of XML and JSON at the present date does not match the interest of XML at its peak.

Chapter 3

Problem Characterization

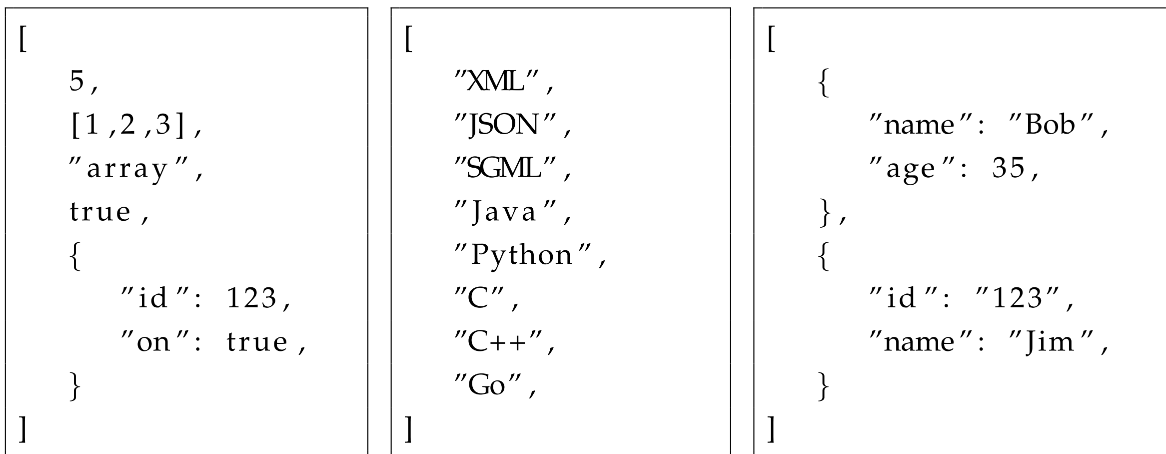
In the shape motivation presented in Chapter 1, the main issue was that a single schema was generated for unrelated types of shapes. Consequently, this schema was ambiguous as it also accepted any combination of the different types. If we were to instead treat each shape type as a JSON object with a particular structure, we see that the same problem arises. Existing JSON schema generation tools work off the assumption that all objects in an array have the same structure, and thus, generate a single schema by merging together the structures of all the objects. This process also results in an ambiguous schema as information on the individual structures is lost.

For example, in the JSON document of figure 2.10, the *results* array has two objects with different structures. When a schema was generated using the QuickType program [2], the two structures got merged together. As a result, objects having the keys *kind*, *collectionId*, *isStreamable*, and *genres* are also considered valid, regardless of the fact that they consist of a combination of both structures. Even worse is the schema generator in [5] [1] which generates a schema for the entire JSON array based only on the object in the first position; every other object—regardless if it has a different structure—is ignored.

[1]Results are not shown in the thesis due to their length.

3.1 JSON Arrays and Data Types

The primary issue with existing schema generation methods comes as a result of treating an array of JSON objects as a homogeneous data structure rather than a heterogeneous data structure. A homogeneous data structure is a data structure that contains data of the same type (e.g. arrays in Java or Golang) [44, 45]. In contrast, heterogeneous data structures can contain data with different types (e.g. lists in Python or Javascript) [44, 45].



(a) Heterogeneous array (b) Homogeneous array (c) Array of unknown type

Figure 3.1: Three JSON arrays illustrating the difference between homogeneous and heterogeneous data structures.

Figure 3.1 shows three examples of a JSON array. The first is a heterogeneous array, as it includes a combination of integers, strings, arrays, and objects; the second is a homogeneous array containing only strings, and the third is an array of two objects. This third array could fall into either the homogeneous or heterogeneous categories depending on what is considered the data type of a JSON object.

One way to view the data type of a JSON object is to assume that all objects have the same type; that is, the type of a JSON object is a JSON object regardless of the contents inside of the object. When viewed this way, the array in figure 3.1c is

a homogeneous data structure. To construct a schema for the general JSON object data type then involves taking the union of all possible keys in the objects. This is how most schema generation tools work due to the fact that it always results in a valid schema. If an array element is a JSON object, it is valid only if each key in the object matches one of the keys in any of the objects used to generate the schema.

The issue with this method is that it is possible to have two JSON objects with no relation to each other. In this scenario, it instead makes sense to treat JSON objects with different structures as if they were different data types. To illustrate this scenario, consider two JSON objects: one having the keys *city*, *province*, and *country*, and the other having the keys *title*, *author*, and *publisher*. It is obvious that the two objects represent different entities, namely, a location and a book. Thus, it would be beneficial to treat them as separate data types.

This idea of two JSON objects having different data types captures the principles found in object oriented programming (OOP). In this paradigm, the object is the central method of "encapsulating state and behaviour" [46]. This is commonly implemented through the construction of abstract templates known as classes. Each class consists of a set of properties used to store data (the state), and a set of methods used to manipulate data (the behaviour). Objects are then instantiated from a class, where each object has its own values for the properties. In this regard, all objects created from the same class have the same data type—the class itself [46]. Two classes can then have the same properties with the same names but still be treated as different data types.

When converting the data encapsulated within an OOP object directly to a JSON object, this data type information is lost leading to the creation of heterogeneous arrays. The receiving end of the JSON object is then responsible for processing each object with the assumption that it knows the data type of the object.

In this thesis, we are assuming that each element of the array given as input

consists of a JSON object that is associated with a particular structure type. A structure type represents a particular structure a JSON document can have, along with some external reason for why that information is grouped together. Two structure types can have similar structures but still be considered different structure types. This is due to the different meaning behind what the structures represent. For example, as discussed in section 2.4.2, the iTunes Search API returns data containing a JSON array, where each object in the array represents a type of product available in the iTunes store. All objects representing the same type of product have very similar structures, as that structure is used to model that product.

3.2 Problem Overview

A better approach to schema design for JSON arrays would be to generate a schema for each of the possible structure types. An object in an array would then be valid if it satisfies at least one of the possible schemas. The benefit of this approach is that it reduces the chance of ambiguity as keys belonging to different types are no longer being mixed together in a single schema. If an object now tries to mix together two structures (as those in the shape example in figure 1.2c), it would be rejected as it does not satisfy the schema for either type.

A potential issue with this method is that it relies on the structure type of each object being known in advance. If this information is not known, then the problem becomes significantly more challenging. This is because there is no completely accurate method of deducing what structure type an object is trying to model—largely because objects may contain optional keys. If two objects have different keys, it cannot be determined whether they represent different structure types or the same type. In the second case, the differences between their structures are due to them including different optional keys. For example, the two objects in figure

2.10's *results* array may actually be the same structure with different optional keys. Without having this external information to influence the decision, any method can only guess with varying confidence what structure type an object has.

One way to avoid this issue is to draw from the existing field of cluster analysis [47]. The goal of this field is to "divide [a] set of objects into homogeneous groups [such that] two arbitrary objects belonging to the same group are more similar to each other than two arbitrary objects belonging to different groups" [48]. Clustering algorithms could be applied to a set of objects to partition them into a set of groups. A schema could then be generated for each group. The assumption behind this method is that two objects with the same structure type should have structures that are more similar compared to two objects having different structure types. This avoids the issue of needing to identify each object's type, as most clustering algorithms work off similarity calculations that compare the objects structures.

Applying clustering algorithms alone does not guarantee that each structure will be identified. If two types of structures are very similar to each other, it is possible that clustering algorithms may place the objects into the same group. Any differences between their structures would then be considered optional. How often this occurs largely depends on the number of groups generated in the clustering process. Regardless, this method does improve the accuracy of a schema as multiple tailored schemas are generated rather then a single schema.

Another downside to this approach is that it may come with a decrease in performance. With a single schema (as in the shape motivation in figure 1.2b or the JSON schema in figure 2.11), each object only has to be compared against one schema to determine its validity. By generating a schema for each possible structure type, an object may have to be compared against multiple schemas to determine which one it satisfies (if any). For example, if there are 1000 objects and 10 different schemas, up to 10000 schema validation attempts may be performed.

3.3 Driving Observation

We introduce a different approach in this thesis based on the observation that heterogeneous JSON arrays usually contain a set of keys within each object for the purpose of identifying its structure type. We will call these keys *identification keys*. For example, each object may have an identification key containing a version number in scenarios where the format has changed over time. This is the major difference between an array of JSON objects and a collection of random JSON objects—the array is designed to hold related content and be processed as a single unit.

This assumption of the existence of identification keys comes from JSON being a data interchange format for transmitting information between two machines. Because of this, a program has to exist on the receiving end to accept the JSON document and automatically process it. With a homogeneous array, this is a simple program as each object has the same data type and can be processed in the same way. When working with a heterogeneous array, different objects may have to be processed in different ways. In the iTunes example, song and movie objects contain different keys representing different concepts. How a program processes a song will be different from how it processes a movie.

One alternative to identification keys could be to hardcode a set of key lookups into the program. This action is performed before deciding how to process an object. Based on which keys are present in the object (i.e. considering only if a key is present or not rather then looking at the value of a key), the program would process the object accordingly. For example, objects with keys `A` and `B` would be processed one way, whereas objects with keys `C` and `D` would be processed another.

This approach has two disadvantages. First, additional key lookups have to be performed on each object to check the different scenarios. Second, this method is not "future proof". For example, suppose that a new structure type with keys `A`, `B`,

and E is added in the future that requires different processing compared to objects with just A and B. Not only would every program have to be manually modified to account for this new structure, missing a program would lead to unintended errors. Objects with keys A, B, and E will be processed as if they only had keys A and B. Over time, this method will likely lead to large if-else chains that are needed to check all possible edge cases that arise. Thus, the easiest method to identify an objects structure is to use identification keys.

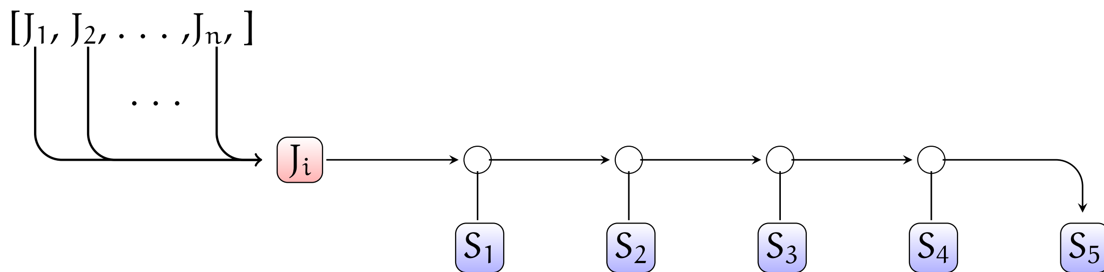
3.4 Schema Decision Trees

In both the clustering approach and the approach discussed in the previous section, the result is a set of groups that partition an array of JSON objects. A schema can then be generated for each group. While both methods ultimately result in a set of schemas, the second approach has an advantage in that it also determines the identification keys of the array. In the traditional clustering approach, this information is unknown, as each group may not correspond to a single structure type. Having this information is beneficial because a schema can be generated for each structure type, and all objects of that type can be validated against that specific schema. This combines the advantages of the other approaches; each object is only compared against a single schema, but that schema more accurately represents one of the possible structure types.

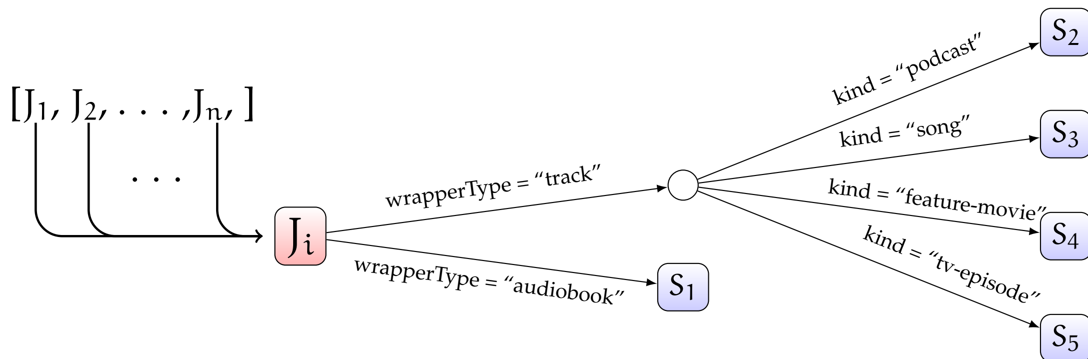
To determine which schema an object needs to be compared against, we introduce the concept of a *Schema Decision Tree*—based on the idea of decision-tree classifiers. This type of classifier consists of a tree structure where each leaf node is associated with a class, and each inner node is associated with a decision. The process of determining which class a piece of data falls under consists of starting at the root node and working down through the tree. At each inner node, the data

is compared against the decision at that node; the result of the decision then effects which child node the data moves to next [6].

Similar to decision-trees, a schema decision tree is a tree structure that holds information about the identification keys of a given array as well as the schemas for the different structure types. Within this tree, each leaf node contains one of the possible schemas. Each inner node contains a decision involving one of the identification keys. This decision is based on the value of the given key, where each possible value is associated with one of the child nodes. An object only needs to compare its values for the specified keys against those found in the tree to determine which schema the object should be compared against.



(a) Validating a JSON array through a linear search approach.



(b) Validating a JSON array using a schema decision tree. This tree is part of the final schema decision tree generated for the iTunes Search API.

Figure 3.2: Comparison between validating an array through a linear search approach versus validating an array using a schema decision tree.

Figure 3.2 shows an array of objects being validated using the two different methods. In 3.2a, each array object J_i is compared against the first schema. If the object satisfies the schema, the program moves on to the next object in the array. If the object does not satisfy the schema, the program compares it against the second schema. This process repeats until the object eventually satisfies one of the schemas or is rejected. With the *Schema Decision Tree* in 3.2b though, a lookup is first performed on the key *wrapperType*. If that key does not exist or its value is not one of the possibilities outlined, the object is rejected. Otherwise, the object moves down the tree based on the value of the key. This process repeats until the object eventually reaches a leaf node containing the schemas it should be compared against.

3.5 Problem Statement

The problem explored in this thesis is improving automatic schema generation for heterogeneous JSON arrays. To do this, we aim to first determine a set of keys that identifies an object's structure. A schema can then be generated for each type of structure that more accurately reflects its content. Based on these keys and resulting schemas, we generate a schema decision tree to help in the validation process.

In order to determine an array's identification keys, we introduce a variation of cluster analysis based on an operation we define called splitting. Let the notation $val(k, J)$ represent the value of the key k in JSON object J. A split operation on a group of objects [2] g is then defined by choosing a key k common to all the objects in g, and partition g into $g_1, g_2, ..., g_n$ such that the following two properties are satisfied.

[2] In this context, a group in considered an array where an objects index position is ignored.

1. $\forall J_1, J_2 \in g_i : val(k, J_1) = val(k, J_2)$

2. $\forall J_1 \in g_i, J_2 \in g_j, i \neq j : val(k, J_1) \neq val(k, J_2)$

This definition says that based on the value of this key k, the objects are partitioned into at least one group. All objects with the same value for the key are placed in the same group (property 1), and objects in different groups have different values for the key (property 2).

Applying the split operation one time assumes that a single identification key exists. As we have seen from the iTunes example in figure 3.2b though, this is not a valid assumption. Based on this, we define two issues that have to be addressed.

1. More than one identification key may exist. For example, the combination of two keys may identify an objects structure.

2. An identification key may only be present in a subset of the objects. For example, one identification key partitions the objects into a set of groups. A second identification key then exists for one of these groups to further partition it. This key may only be present in the objects of that group, and not in the objects of other groups.

As a result of these issues, any algorithm designed to determine a groups identification keys needs to be able to recursively apply the split operation; that is, further apply the split operation to one or more of the groups generated from another split operation. Applying a recursive solution to this problem resolves the aforementioned issues. For issue one, the split operation would be applied to one of the keys. Each of the resulting groups could then be partitioned again based on the second key. This is equivalent to first partitioning on the second key and then on of the first key. In both cases, the result is a series of groups having the same

values for the given keys. Furthermore, issue two is solved because the split operation could be applied to one group in particular. All other groups, not having the second identification key, would be left alone.

While the split operation tells us how to partition the objects for a given key k, it does not tell us how to choose this key. Contained within the objects may be many potential options for k with only a few of these being used for identification purposes. Other keys may just be common to all structure types. Thus, we need a way of determining whether a key falls under the identification key category or the non-identification key category. To solve this problem, we use the assumption that objects with the same structure type should have more similar structures when compared to objects with different structure types. To illustrate this concept, suppose all objects have two keys: *version* and *date*. Splitting based on the *version* key should result in groups where objects in the same group have similar structures. On the other hand, splitting based on the *date* key should result in groups of random objects, as there is no connection between the keys value and its structure type.

Just considering this assumption alone is not enough to determine if a key is used for identification purposes. Consider a scenario where all objects in a group contain an *id* key that acts like a unique identifier (like a primary key in a relational database). Due to being unique, splitting on this key results in a partition where each object is placed into its own group. By only going off the assumption that splitting on an identification key results in groups of similar objects, *id* should be used for identification purposes. Each of the resulting groups has a perfect similarity score due to only containing a single object (i.e. there is no variance in the structure of the objects when the group only contains a single object). For this reason, maximizing the similarly of objects in the same group is not sufficient for determining whether a key is an identification key. The number of groups

generated also has to be considered.

Another problem arises if we were to instead strictly focus on trying to minimize the number of groups. By not considering a group's similarity anymore, the best way to partition the objects would be to simply not partition them at all, and instead, leave them in a single group—regardless of if they have the same structure type or not. This effectively renders no results, as we are back to generating a single schema for the entire array.

In order to generate useful results, a balance between maximizing the similarity of each group and minimizing the number of groups needs to be found. To do this, we introduce a similarity threshold T that represents the minimum similarity a group must satisfy to be considered valid. This allows the best split to be the one with the minimum number of groups such that the similarity of each group is above the threshold.

Furthermore, having a similarity threshold provides a base condition for when to stop recursively splitting. If a split operation results in all but one group being above the threshold, only that one group needs to be further partitioned. The rest of the groups can be left alone, as they have all reached the base condition.

3.6 Assumptions

3.6.1 Common Structures

Related keys have the same name between objects. This removes the need to consider semantic differences when comparing keys. For example, the keys *latitude* and *lat* are assumed to have no relation to one another even though they are semantically related (one is an abbreviation of the other).

Further, related keys will appear in the same location between objects. This removes the need to match sub-structures that appear in different locations. For

example, one object containing the keys *street*, *city*, and *country* in the root object and another containing the same keys nested within an *address* object are assumed to have no relation to one another.

We believe these assumptions to be valid because all objects in an array are originating in the same document from the same source. If a single program is automatically creating the document, it should have common formats for representing information. Furthermore, the document is designed to be processed by a single problem. This means that the program needs to know which keys contain certain information and where they are located in the document. Achieving this can only be done using common names for keys and storing keys in common locations.

3.6.2 Existence of Identification Keys

Each object is associated with a particular structure type and has a set of keys that identifies its structure. The aim of this thesis is to then identify such keys. Further, we assume that two objects of the same structure type have more similar structures compared to objects of different structure types. The reasoning behind this assumption was discussed in sections 3.2 and 3.3. We leave the possibility of removing this restriction as future work, and we discuss a possible method on how this could be done in section 8.1.

3.6.3 Complete Input Data

In order to generate the schema, we assume that the input data is complete in the sense that all possible structure types are included in array. In addition, all optional keys related to a structure type are included in at least one object of that type.

We believe this assumption to be appropriate because a schema cannot be gen-

erated for data that is unknown. If we assume incomplete data (i.e. the data does not include all types of structures), then the possibility arises of the schema rejecting objects that should be considered valid. We cannot automatically determine if a structure that failed to validate against the schema was because the schema did not include its structure type or if it actually was ill-structured. The only other option could be to forward invalid objects to an admin to manually decide which scenario it falls under. Regardless, there is no way of dynamically updating a schema, as doing so defeats the purpose of having a schema in the first place.

3.6.4 Array of Objects

The JSON array given as input contains only JSON objects as elements. Each object can have an arbitrary number of keys with arbitrary data types and be arranged in an arbitrary nested structure.

The reason for this assumption is that each element in an array falls into one of three categories.

1. The element is one of the simple data types (integer, string, etc.).
2. The element is a JSON array.
3. The element is a JSON object.

In the first category, a schema can be easily generated by looking directly at its data type. In the second category, the schema for that element would be computed separately from all other elements—including other JSON arrays. The reason for this is that we cannot say if two arrays contain the same identification keys or not. As such, we do not focus on this type of element as generating a schema for it forms an identical sub-problem. By improving schema generation for an array of JSON objects, we inadvertently improve schema generation for elements that are arrays themselves. Finally, the third category is the topic explored in this thesis.

Based on our observations, the vast majority of situations have an array consisting of either all simple data types or all JSON objects. As computing the schema of an array of simple data types is trivial, we focus on improving schema generation for an array of JSON objects, as that is the core problem.

Chapter 4

Related Works

To our knowledge, no previous work has directly looked at the problem of schema generation for heterogeneous arrays contained within JSON documents. Thus, the works analyzed in this chapter mainly fall into the two categories of general clustering of semi-structured data and software tools involving schema generation. Section 5.1 in the following chapter further discusses different similarity calculations used; the focus of this chapter is instead on the big picture problem.

4.1 Related Works Involving JSON

The first related work is by Izquierdo and Cabot, and it focuses on generating a UML-like model for a collection of JSON documents [49]; in particular, documents returned from the various endpoints of a web API. The goal of their work is to help developers better understand and visualize the global data model hidden behind the API. Developers do not usually have direct access to this model. Instead, they interact with it through overlapping snippets of data returned from the APIs endpoints. By piecing together these related snippets, the authors hope to better understand the entire data model. The running example they use throughout their paper is a transportation API. In this API, one endpoint returns identifiers for the

closest train stops to a given location. These identifiers can then be passed into a second endpoint to get more details on each stop. Although the information returned from the two endpoints is different, it is still related due to a common application domain.

Their algorithm works in three stages. First, each JSON document is converted into a non-JSON representation they call a model format. Secondly, all the models from the same endpoint are combined together to create a new model. The difference with this new model is that it captures the structural variation present in the documents. For example, some documents may contain optional keys not included in other documents. Finally, models from the same endpoint are combined together based on overlapping substructures. The result of this process should be a complete view of how the information and endpoints are connected.

This work is then the bases for their online tool called JSONDiscoverer [50]. Of particular note is the first stage where they extract a single schema for a JSON array. Like other schema generation mechanisms, their tool considers all JSON objects to represent the same structure type and only generates a single schema for all of them.

In Klettke et. al.'s work, they develop a schema extraction algorithm targeted at collections of JSON documents found in NoSQL databases [51]. The first step of their solution involves taking a group of JSON documents and building a tree-like graph based on them. The purpose of this graph is to summarize the parent-child relationships of all the key-value pairs, as well as track which documents each relationship occurs in. Based on this graph, a schema can be generated that accounts for all structural variations (e.g. optional key). In addition to this, the generated graph can also be used for determining structural outliers. These are defined as patters of keys occurring in only a few of the documents. The database admin can then use this structural outlier information to classify if the documents

are ill-structured or not. Finally, this paper discussed a series of calculations for measure the similarity of groups of JSON documents. This is further discussed in section 5.1.

One point of significance briefly discussed in this paper is a preprocessing stage where the document collection is partitioned into smaller groups based on some key (date, timestamp, etc). Their algorithm can then be applied to the resulting groups to improve the accuracy of the generated schemas. This idea is similar to that presented in this thesis; however, they only mention it as something that can be done if those keys are already known. They do not discuss any ways of actually determining said keys.

Next, Spoth et. al. present an OLAP tool called SchemaDrill [52]. The aim of this tool is to help users visualize a collection of JSON documents. From this visualization, users can then create a relational mapping of the documents. Their application works by first displaying a list of all the keys whose value is not a JSON object or array (i.e. the leaf nodes of the tree). Furthermore, each key includes the path from itself to the root node. Users then groups together related keys with these groups forming the bases of a set of relational tables. As the number of keys in the list could range in the hundreds or thousands and overwhelm the user, their application preemptively groups the keys based on two calculation they define. These calculations are called correlation and anti-correlation. Correlation looks at how often two keys appear in the same JSON object. Keys appearing frequently together should then be placed in the same group. Similarity, anti-correlation looks at keys that rarely occur together, and instead, tries to place them in different groups.

While some of the ideas in this paper carry over to our work (such as how they represent keys), the focus is on flattening JSON documents into relational schemas rather then on semi-structured data schema generation. A consequence of this is

that the authors do not give any special consideration for JSON arrays. In fact, they treat arrays as normal JSON objects where each element in the array is assigned a key; the name of this key is just the index of the element in the array.

In [53], [54] and [55], Bazizi et. al. look at the problem of schema generation for very large JSON datasets (in the order of millions of documents). Because of how time consuming it is to sequentially process a collection this large, they develop an algorithm capable of generating a schema in parallel using Apache Spark and the MapReduce paradigm. Their algorithm works in two phases. In phase one (the map phase), each JSON document has its values reduced down to their datatypes. For example, the JSON object {*"A":123*} is reduced down to {*"A":Num*}; similarly, {*"D": [123, true, "abc"]*} is reduce down to {*"D":[Num, Bool, Str]*}. Phase two (the reduce phase) then takes these simplified documents and repeatedly merges them together in a process they call fusing. Fusing works by taking the union of all datatypes at each layer of the documents. If the same key appears in two documents with two different data types, the key gets assigned the union of the two data types. The result of phase two is a single simplified document capturing all the possible structures in the collection. A schema is then generated from this document.

When their algorithm reaches an arrays, it is first passed through a simplification process that reduces the elements datatypes down to their simplest form. This is done by combining all occurrences of the same datatype together. For example, an array containing two strings and an integer gets simplified down to one-or-more strings and an integer. When there is more then one JSON object in the array, they are merged together into a single JSON object. So while this work does consider heterogeneous arrays, they treat all JSON objects as the same type and merge them together.

Next, Perzoa et. al. define a formal definition of the syntactic and semantic meaning behind the JSON Schema format [43]. The purpose of this formal definition is to provide a uniform way for applications to interpret schema documents—especially involving some of the gray areas where interpretation of a schema document may vary between implementations (e.g. recursive schema definitions). To create this formal definition, they create a grammar for the JSON Schema specification outlining how the specification's features should interact with each other.

Finally, DiScala and Abadi design an algorithm to transform hierarchically structured JSON documents into flat structures suitable for relational databases [56]. Their algorithm works by applying unsupervised machine learning to group together keys that likely correspond to relational entities. This way, all instances where those keys occur together can be stored in the same relation.

In order to detect related keys, they first perform functional dependency detection between pairs of keys. The purpose of this is it look for scenarios where the value of one key effects the value of another, as this may correspond to primary key–foreign key relationships. Phase two of their algorithm then looks for reoccurring substructures. Finally, phase three generates a relational schema that maps substructures to tables and connects them through foreign key relationships. Regarding arrays, this work treats them as independent sub-problems that are recursively computed.

4.2 Related Works Involving XML

In addition to the above literature on JSON, it is also advantageous to examine other related work based on different semi-structured data formats. In particular, we now look at XML due to its widespread popularity in both industry and academia. While the two formats are not identical in how they represent informa-

tion, they are similar enough such that work done for one format is still applicable to the other in some fashion.

To begin, one prominent area of research is developing clustering algorithms to group related XML documents together. Motivation for this can be found in applications such as document retrieval systems. These system work by accepting one or more XML documents as input and return other documents related to them. As discussed in the overview paper by Piernik et. al., most clustering applications involve three phases [57]. First, a preprocessing stage occurs to transform the text-based XML documents into more computationally compatible formats. For example, an XML document may be turned into a tree data structure that is implemented for a specific programming language. Second, a similarity calculation can be developed as a way of measuring how related two or more XML documents are to each other. Finally, a clustering algorithm is applied that uses this similarity calculation to group related documents [57].

This problem has been thoroughly explored under the context of different similarity calculations and clustering algorithms. A few examples of such works include [47] where they use a tree-edit distance calculation and hierarchical agglomerative clustering to group XML documents. [58] also takes a tree-edit distance approach, except they restrict edit operations to only leaf nodes and not inner nodes. In [59], the authors also consider the semantic similarity between XML tags through the use of the WordNet English lexical database. Similarly, [60] considers the semantics of entire sub-trees rather then just elements themselves. Finally, [61] approaches the problem by first grouping together all nodes at each level of the tree. These level structures, as the authors call them, are then compared together rather then the trees themselves. [62] expands on this idea but instead considers the edges of the trees rather then the nodes.

Related to clustering, [63] and [64] both look at grouping together XML schemas rather than XML documents. The goal behind their work is to reduce the number of schemas in a collection by merging related schemas together. To determine which schemas should be merged, the authors apply a clustering algorithm to create groups of related schemas. The similarity calculation they consider here is based on both the syntactic and semantic aspects of a tag's name. All schemas in a group are then merged together into a single schema. This approach can be recursively applied depending on how many schemas is desired.

The final area of related work discussed in this chapter deals with the problem of duplicate detection. Based on data cleansing, this problem looks to remove duplicate elements that appear within an XML document. In terms of JSON, this problem can be thought of as the removal of duplicate objects in an array. In [65], two elements are classified as duplicates if they have the same parent element, the same tag name, and similar content. Their algorithm works through a top-down approach. When it comes across two elements having the same parent element and same tag name, their contents are compared using syntactic and semantic similarity calculations. This differs from the approach taken in [66]. In their algorithm, they instead employ a bottom-up approach by first finding matching leaf nodes. Substructures are then built up from these leaf nodes to see how similar their parent elements are.

4.3 Related Works Overview

As discussed in this chapter, all related works have either dealt with the general problem of schema generation, clustering of semi-structures data, or measuring the similarity of JSON or XML documents. The closest work we have found to the problem discussed in this thesis is by Klettke et. al. in [51]. In their paper, they

briefly discuss how the accuracy of a schema can be improved by first partitioning the documents based on some key. However, they only mention this partitioning as something that can be done if the key is already known. Furthermore, they do not consider many of the related problems explored in this thesis. These problems include actually determining identification keys, connecting the resulting schemas together (e.g. schema decision tree), or multiple identification keys existing.

Chapter 5

Solution Details

Based on the problem definition in chapter 3, the main issue to address is partitioning an array of JSON objects into a set of groups by recursively applying the split operation. The smallest set of groups, such that each group has a similarity score over a given threshold, is then the best partition. A schema decision tree can then be generated based on it.

5.1 Syntactic Similarity Scores

The first issue to discuss in our solution is defining the concept of similarity in the context of JSON objects. A similarity score (or similarity measure) is a function that takes two or more items as input and returns an integer representing the similarity between them [67]. What similarity is defined as depends on the application and what type of data the items consists of. To illustrate, consider an application involving 3D coordinates. One way to measure the similarity between two points could be to take the euclidean distance between them. Another method could be to measure the angle between two lines generated by connecting each point to the origin. Both of these calculations, however, are restricted to this type of data and could not be used to measure the similarity of semi-structured data without first

transforming it.

Creating similarity calculations tailored to semi-structured data has been an active research problem ever since the concept first emerged. Most of the calculations created can be classified into either the semantic or syntactic categories.

Syntactic calculations focus on measuring the differences in how the data is arranged in structures. They usually assume that two keys are related if they have the same name and not related if they have different names. Similarity is then based on how often the same keys appear in comparable locations. In figure 5.1 for example, the keys *firstName* and *lastName* are nested within the *name* key; however, they could just as easily be placed directly in the *author* key without the meaning of the rest of the document drastically changing. Syntactic calculations take this structural information into account and would likely score this scenario higher than if *firstName* and *lastName* had instead been nested within the *location* key.

```
1  {
2      "location": {
3          "city": "Vancouver",
4          "country": "Canada"
5      },
6      "author": {
7          "id": 1753,
8          "name": {
9              "firstName": "John",
10             "lastName": "Smith"
11         }
12     }
13 }
```

Figure 5.1: JSON document in text-based format.

On the other hand, semantic calculations focus on measuring the differences between the meaning of the data. Given two keys, these calculations look at prop-

erties such as their names, values, and locations when trying to measure how similar they are. Semantic calculations are particularly useful for comparing data originating from different sources. Different companies often have different naming practices. Thus, assuming two keys are related only if they have the same name is no longer sufficient. For example, one company may use the keys *latitude* and *longitude*, whereas another company may use the common abbreviations *lat* and *long*. There is obviously a shared meaning behind what these keys represent even though they are syntactically different. Semantic calculations look to identify this commonality.

As a result of key names in JSON being restricted to strings, semantic calculations are significantly harder to create due to the involvement of human language. These calculations have to account for different scenarios such as abbreviations (id vs. identification), synonyms (city vs. town), conjoined words (e.g. firstName), etc. Furthermore, common techniques used in natural language processing are based on large data sets. By comparison, semi-structured data often only consists of a small number of keys. Additional information behind the meaning of the data may be provided through external resources. For example, one of the keys in the iTunes Search API data is *artworkUrl60* which contains a URL for related artwork having a size of 60x60 pixels. This meaning behind the key represents is only provided through external developer documentation not accessible to the similarity calculation. Even if the documentation is provided—and even exists in the first place—the formats of these documents range widely with no common page structure or content. Thus, semantic calculation have to try to extract this information from the key's name alone.

For the purpose of this thesis, we are only interested in syntactic similarity calculations. The reason for this is that the primary purpose of a schema document is to validate the structure of other JSON documents. This involves making sure that

specific keys appear in specific locations and have specific data types. Their focus is not on trying to match common meanings as semantic similarity calculations do.

5.1.1 Tree-edit Distance

One of the popular similarity calculations for semi-structured data is the tree-edit distance. This calculation is based on the inherent tree-like structure that results from nesting information inside other information [68, 69, 70, 47]. Each time this nesting occurs, a parent-child relationship emerges that can easily be modelled in a tree. For example, figure 5.2 shows the JSON document in figure 5.1 as a tree structure. At the top level, each JSON document consist of a single nameless root object. This becomes the root of the tree and is given a default name of *root*. Nested directly within this root object are the *location* and *author* keys, and appropriately, they become the direct children of the root node. This process repeats itself until it eventually reaches a node containing a single value rather then more nested information (these are known as leaf nodes). The value of this node can either be stored directly in the node itself (as depicted) or as an single additional child node.

Originally introduced for general tree structures in the early 1970's, the tree-edit distance between two trees is defined as the length of the minimum sequence of node insertions and deletions needed to transform one of the trees into the other [68, 69, 70]. When a node is deleted, all of its children have their parent node set as the deleted nodes parent. This idea of edit operations has further been expanded upon over time to include other operations such as renaming a node and the insertion or deletion of entire subtrees as a single operation [70]. Most solutions for calculating the tree-edit distance are based off of the previously explored string-edit distance problem [69]. This problem looks at how many character insertions and deletions are needed to transform one string into another.

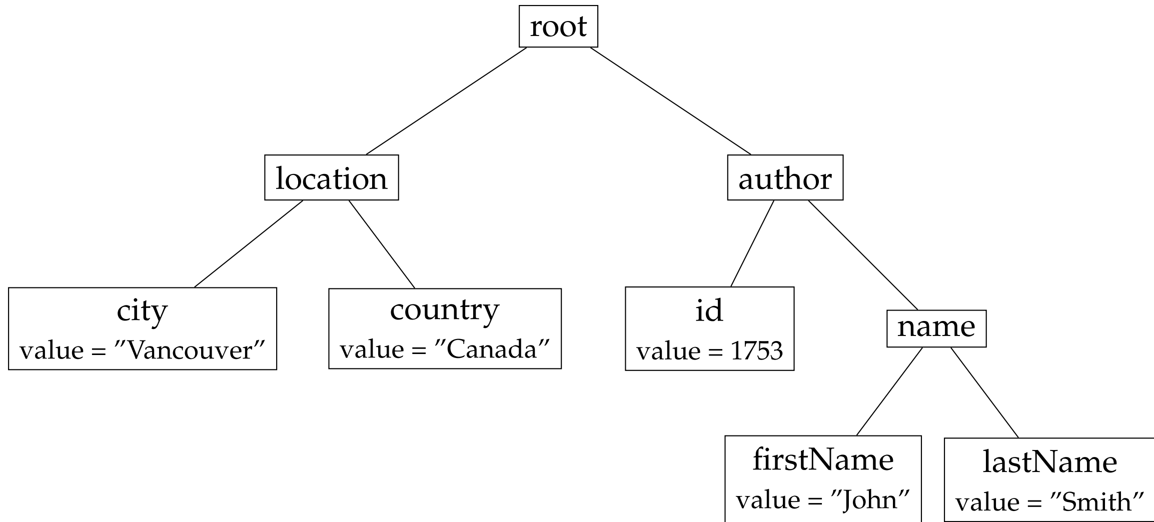


Figure 5.2: Tree representation of the JSON document in figure 5.1.

5.1.2 Disadvantages of Tree-edit Distance

While the tree-edit distance is a popular similarity calculation in previous literature, we find that it is not suited for the problem explored in this thesis; the major reasons for this are outlined next.

5.1.2.1 Similarity of More than Two Trees

The tree-edit distance is designed to measure the similarity between two trees. In previous clustering approaches, this was not an issue as a similarity matrix was usually first created. This matrix contained the similarity scores for each pair of trees, thus allowing the tree-edit distance to be applicable. A clustering algorithm (such as hierarchical agglomerative clustering as used in [47]) would then work off this matrix when creating the clusters.

This differs from our approach as groups are first generated using the split operation. How good a split operation was then depends on the similarity of these groups. As such, we need a similarity calculation capable of measuring the similarity of more then two documents simultaneously.

5.1.2.2 Comparing Unordered Trees

Most algorithms that compute the tree-edit distance are based on ordered trees. This is a type of tree where, for each node, every child is assigned a value indicating its position among the rest of the children. Two nodes are then considered equal only if they have the same children appearing in the same order. On the other hand, unordered trees would consider two nodes equal as long as they have the same children—regardless of their order.

Previous literature on XML has used ordered tree-edit distance algorithms because XML documents are considered ordered trees [18]. XML schemas, for example, have separate notation for when the order of elements does not matter. This is compared to JSON where objects are treated as unordered sets of key-value pairs. As such, tree-edit distance algorithms for ordered trees cannot be applied.

Creating tree-edit distance algorithms for unordered trees has also been previously explored [71, 72]; however, these algorithms are more conceptually complex and computationally expansive. The reason for this is that the assumption that a node's children appear in a specific order allows efficient dynamic programming techniques to be applied. For example, comparing two ordered nodes means that the first child of each node have to match, the second child of each node have to match, etc. Compared to unordered trees, the first child of a node may match any child of the other node. This results in significantly more permutations to consider.

5.1.3 Path Distance

Because of these reasons, we have decided to use a different similarity score that has also been previously discussed in literature. This score is based off the well known Jaccard index used for measuring the similarity between two sets [51, 57]. Given two sets, the Jaccard index returns a rational number inclusively between

0 and 1 based on dividing the size of the intersection by the size of the union. A score of 1 represents the two sets having exactly the same elements. A score of 0 represents the two sets having no element in common.

Before the Jaccard index can be applied to JSON documents, one problem has to be addressed; namely, a JSON document takes the form of a tree-like structure whereas the Jaccard index requires sets as input. To address this, we first apply a transformation to each JSON object to remove its nested structure and "flatten" it down to a linear structure. This notion has also been discussed in past literature [51, 52, 43].

The main concept behind flattening a JSON object is that the useful information within the object is contained in the leaf nodes (ie. the key-value pairs whose value is an integer, boolean, string, or null). All inner nodes (ie. keys whose value is an object or array) mainly exist to arrange the leaf nodes into smaller, more meaningful, structures. Using this concept, a JSON document can be flattened by taking each key that is a leaf node and appending onto it the names of the keys that occur when traversing to it from the root of the tree. For example, the JSON document in figure 5.1 can be transformed into the set of key-value pairs depicted in figure 5.3.

```
1 location/city: "Vancouver"
2 location/country: "Canada"
3 author/id: 1753
4 author/name/firstName: "John"
5 author/name/lastName: "Smith"
```

Figure 5.3: Path representation of the JSON document in figure 5.1.

More formally, let a JSON object J, in path notation, be defined by the following equation.

$$J = \{k_1 : v_1, k_2 : v_2, ..., k_n : v_n \mid k_i \neq k_j \text{ for } i \neq j\} \tag{5.1}$$

Here, $k_i : v_i$ is a key-value pair in the flattened structure (e.g. in figure 5.3, *location/city: "Vancouver"* is a key value pair where the key is *location/city* and the value is *"Vancouver"*). Because J is a set, the set intersection and set union operations can be defined for it. Let the intersection of two JSON objects be defined by equation 5.2.

$$J_1 \cap J_2 = \{k_i \mid k_i \in J_1 \wedge k_i \in J_2\} \tag{5.2}$$

This says that the intersection of two JSON objects is the set of keys common to both objects. The resulting set is not a JSON object, however, as a key is no longer associated with a specific value. The reason for this is that while a key may appear in both J_1 and J_2, the value of the key is likely to differ between them. Thus, it does not make sense to include a value in the resulting intersection, as the two values would either have to be combined together, or two separate key-value pairs would have to be included.

The union of two JSON objects is defined by equation 5.3.

$$J_1 \cup J_2 = \{k_i \mid k_i \in J_1 \vee k_i \in J_2\} \tag{5.3}$$

This says that the union of two JSON objects is the set of keys appearing in either of the objects. Similar to the the intersection operation, the union also results in a set of keys with no values.

With the intersection and union operations now defined, the Jaccard index for

two JSON objects is defined in equation 5.4. We call this equation *sim* for similarity.

$$sim(J_1, J_2) = \frac{|\, J_1 \cap J_2 \,|}{|\, J_1 \cup J_2 \,|} \tag{5.4}$$

5.1.4 Path Notation for Nested Arrays

One concept about path notation that has not yet been discussed is how to represent nested arrays. When only considering objects, as in figure 5.1, a unique path can be created for each key, since two keys cannot have the same name in the same object. As the values in an array are nameless, this property does not hold. Previous literature has dealt with arrays in different ways. Both [52] and [43] insert the element's array index into the path to keep it unique. [51] generates a single schema for the entire array, and thus, merges all objects together resulting in unique paths.

Three main ways exist to deal with nested arrays. Figure 5.5 shows the resulting path notation based on the array in figure 5.4 for each of the three ways.

The first approach is to insert the elements array index into the path (like [43] and [52] did). Figure 5.5a showcases the resulting path notation. The downside of this method is that the number of elements in the array then plays a factor when computing the intersection and union operations. For example, one array containing 10 elements and another containing 100 elements would result in a low similarity score regardless of if each element had the same structure.

The second approach is to instead take the union of all keys and treat the array as a single object (like [51] did). Figure 5.5b shows this option. While this does solve the issue of having arrays of different sizes, it also introduces its own problems. First, the value of the key is lost as a result of each key likely having a different value in each occurrence. Second, by merging together all array elements and treating it as an object, we lose the distinction on whether a key originally

existed in an array or an object.

```
1  {
2    "books": [
3      {
4        "title": "Intro to JSON",
5        "author": "John Smith",
6        "numSold": 173
7      },
8      {
9        "title": "Intro to XML",
10       "author": "Jane Doe"
11     }
12   ]
13 }
```

Figure 5.4: JSON document containing an array.

```
1 books/0/title: "Intro to JSON"
2 books/0/author: "John Smith"
3 books/0/numSold: "173"
4 books/1/title: "Intro to XML"
5 books/1/author: "Jane Doe"
```

(a) Path notation that includes an objects array index.

```
1 books/title
2 books/author
3 books/numSold
```

(b) Path notation that takes the union of the keys.

```
books: Array
```

(c) Path notation that treats an entire array as the keys type.

Figure 5.5: Three path notations for the JSON document in figure 5.4.

The third approach, and the method used in this thesis, is to treat the entire array as a single JSON array data type and not break it down any further. This is shown in figure 5.5c. The reasoning for this is that there is an intrinsic difference between a JSON object and a JSON array. The purpose on an array is to act like a container for when the number of elements is unknown. An object, on the other hand, has a more fixed structure with more meaning behind what keys are included. Furthermore, identification keys are unlikely to exist in a nested array as they would have to appear in every element. For our purposes, we find it satisfactory that two keys having the same name and an array as a value should represent the same concept.

5.2 Similarity of a Group

Unlike the tree-edit distance calculation, the Jaccard index can easily be expanded to measure the similarity of more than two sets. This is a result of the set intersection and set union operations being associative and commutative. Let a group g of JSON objects be defined as follows:

$$g = \{J_1, J_2, ..., J_n\} \tag{5.5}$$

The similarity of g can then be defined by expanding equation 5.4 to consider the intersection and union of all the sets in g rather than just two sets. In both cases, $0 \leqslant sim \leqslant 1$ as the size of the intersection has a minimum value of 0 and a maximum value of the size of the union. This expanded equation is defined by:

$$sim(g) = \frac{|\bigcap_{J_i \in g} J_i|}{|\bigcup_{J_i \in g} J_i|} \tag{5.6}$$

Klettke et. al. discuss this equation in the context of JSON [51]. One point they

bring up is that this equation can be highly influenced by a few objects in a group. If one object in a group has no keys in common with any of the other objects, the score will always be 0—regardless of how similar the remaining objects are. The reason for this is that the equation involves the intersection operation. Thus, if one object has nothing in common with any of the other objects, the intersection will always result in the empty set and a size of 0. While the authors argue this is a downside to the equation for the purpose of measuring a groups similarity, we consider it an advantage. Schema documents have to consider the structure of a group as a whole. Whether a group contains 2 objects with identical structures or 1000, the result is still a single schema. If a group contains two objects with completely different structures, the resulting schema has to include all the keys in both. By having a similarity score that considers the differences in structures within a group rather than how many objects have the same structure, we are able to generate more accurate schemas.

5.3 Similarity of a Grouping

Based on our assumptions discussed in chapter 3, we assume that an identification key is a key whose value corresponds to the type of structure an object has. Based on this assumption, a set of split operations is considered good if the resulting groups each have a high similarity score. The higher the score, the more accurate the generated schema for that group will be. We now define a calculation to measure the similarity among a set of groups which we call a grouping. The definition of a grouping G is defined as:

$$G = \{g_1, g_2, ..., g_n\} \tag{5.7}$$

We use the notation of lowercase letters to represent groups and uppercase letters to represent groupings. The similarity of a grouping G is then defined as:

$$simGroup(G) = \begin{cases} 0 & \text{if } \exists g \in G : \\ & sim(g) < T \\ \frac{\sum_{g \in G} sim(g)}{|G|} & \text{otherwise} \end{cases} \tag{5.8}$$

Here, T is the given similarity threshold that a group must satisfy where $0 \leqslant T \leqslant 1$. What this equation says is that if all the groups in the grouping meet the similarity threshold, then the similarity score of the grouping as a whole is just the average of the similarity scores of the groups. However, if one of the groups does not meet the threshold, then the similarity score of the grouping is just 0.

The reason for introducing a similarity threshold is that a scenario would arise where a grouping with one large group of low similarity and many small groups of high similarity would still have an overall high grouping similarity score because each group was weighted equally. The large amount of small groups were artificially bringing up the average. Another option was to have a weighted average where the score considers the number of objects in each group; however, this calculation has the opposite issue where a grouping with one large group of high similarity would ignore small groups of low similarity. Introducing a similarity threshold gave two advantages:

1. All groups in the grouping must have a satisfactory similarity score.

2. Less groupings have to be considered when computing the best grouping. If a group does not satisfy the similarity threshold, then we know it is not part of the best grouping.

5.4 Refining the Scoring Criteria

Building off these definitions, we can now refine our criteria for selecting the best grouping. Discussed in the problem statement, the best grouping is the set of splits that results in the fewest possible groups, such that each group has a similarity score over a given threshold. Of all the possible groupings, we find the best grouping by applying these three filtering criteria in the give order:

1. Filter out groupings whose `simGroup` score is below the specified threshold.
2. Filter out the groupings that do not have the fewest number of groups.
3. Filter out the groupings that do not have the highest similarity score.

Criteria number one removes all invalid groupings. Criteria number two then removes all the groupings that are likely to overfit the data to the groups (i.e. it removes groupings from keys like *id* that result in one group per object). Criteria number three then chooses the grouping with the highest similarity score. Although unlikely, it is possible for more then one grouping to meet all three criteria. In this case, any of the resulting groupings can be used as the basis for the schema decision tree.

Table 5.1 shows the criteria being applied to 9 different groupings. While the first 3 groupings have a low number of groups, they do not meet the similarity threshold and are filtered out in the first criteria. Of the remaining groups, the last 4 do not have the minimum number of groups and are filtered out in second criteria. The grouping with the higher similarity score is then chosen as the best grouping.

Num. Groups	Similarity Score	Criteria One	Criteria Two	Criteria Three
1	0.2	✘	—	—
5	0.6	✘	—	—
7	0.72	✘	—	—
8	0.87	✓	✓	✘
8	0.90	✓	✓	✓
9	0.86	✓	✘	—
15	0.92	✓	✘	—
32	0.98	✓	✘	—
44	0.99	✓	✘	—

Table 5.1: A table showing the three scoring criteria being applied to 9 different groupings. The similarity threshold for this example is 0.85.

Chapter 6

Algorithm

Based on the calculations and scoring criteria defined in the previous chapter, an algorithm can now be developed that takes an array of JSON objects and computes the best grouping. One simple approach could be to use a brute force method and first generate every possible grouping. The filtering criteria could then be applied to narrow this list of groupings down to only the best one. Generating all possible groupings could be done by initially placing all objects into a single group and applying the split operation on each key common to all of them. This process then repeats for any group with a similarity score below the threshold until every group is above it, or there are no more keys left to split upon. The possible groupings are then generated by taking all the combinations of the the different split operations, such that no two sibling split operations are included (i.e. if two different split operations are applied to the same group, a grouping cannot contain groups from both splits, as that would duplicate the objects).

The results of this approach can be visualized as a tree structure exemplified by figure 6.1. Each layer of this tree alternates between two types of nodes.

- GroupNodes (denoted by g_i) that represent a group containing a subset of all the objects.

- SplitNodes (denoted by s_i) that represent the groups generated by applying the split operation on a specific key.

In figure 6.1, g_1 is the initial group containing all the object. Among these, two keys are in common to all the objects, and a split operation is applied for each one. Splitting on s_1 partitions g_1 into g_2, g_3, and g_4, whereas splitting on s_2 partitions g_1 into g_5 and g_{10}. g_5 is then further split based on s_3 and s_4. Assuming this is the final tree, three possible groupings can be generated from it. (g_2, g_3, g_4) is generated by splitting on s_1; (g_6, g_7, g_{10}) is generated by taking all the groups in s_3 and the right group of s_2, and (g_8, g_9, g_{10}) is generated by taking all the groups in s_4 and the right group of s_2. Note that (g_8, g_9) alone cannot be a valid grouping, as it does not contain all the objects in g_1. This is because g_1 was split into g_5 and g_{10}, but the grouping (g_8, g_9) is only derived from g_5. Furthermore, the grouping (g_2, g_3, g_4, g_{10}) is invalid because both (g_2, g_3, g_4) and (g_5, g_{10}) contain the same objects—just partitioned into different groups. Any combination of these groups then results in objects being duplicated.

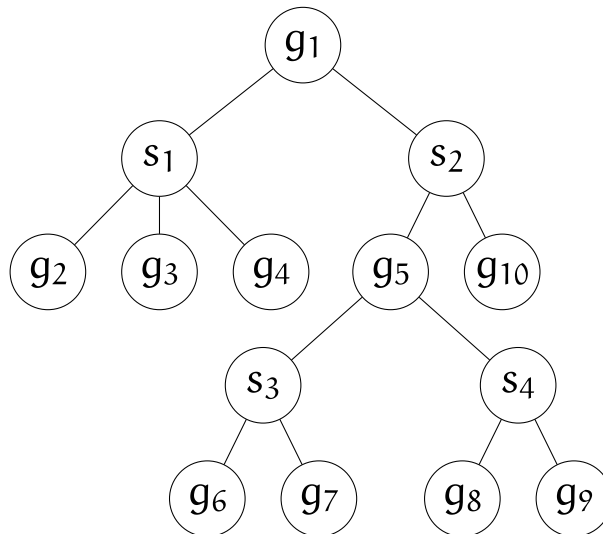


Figure 6.1: Visualization of the alternating layers of GroupNodes and SplitNodes.

While the brute force approach does generate the desired results, more efficient algorithms can be developed based around the same GroupNode/SplitNode tree structure. Instead of first generating all valid grouping and then applying the scoring criteria, we instead integrate the criteria into the generation processes itself. This allows us to preemptively filter our groupings that we know will not be the best grouping and reduce the amount of computation required.

To increase the efficiency of the algorithm, we use the optimization that once a grouping is found whose `simGroup` score is above the similarity threshold, any other partially created groupings, already containing a larger number of groups, can be preemptively discarded. The reason for this is that the second scoring criteria is minimizing the number of groups. Thus, if the size of a grouping is already larger than a previously found valid grouping, it cannot be the best one. By considering this optimization, entire branches/subtrees do not have to be generated.

In order to incorporate this optimization, we introduce an upper bound variable into our algorithm that keeps track of the maximum number of groups a grouping can have. When the algorithm is generating the GroupNode/SplitNode tree structure, it can compare the current number of groups to the upper bound. If the number of groups is greater, the algorithm can preemptively stop recursively splitting down the current branch. Furthermore, if better results are found mid computation, the upper bound can be lowered to reflect the new maximum.

Another optimization to improve efficiency is to first examine the splits containing the fewest number of groups. The reason for this is that those splits will initially give the best chance of lowering the upper bound the furthest. For example, if two groupings exist containing 2 and 20 groups respectfully, it makes sense to first examine the split containing only 2 groups. Applying a split operation to a group always results in an least one group. Thus, the grouping with 20 groups can only ever result in more then 20 groups. By first examining the grouping with

only 2 groups, we run the chance of finding a valid grouping with a size under 20. If this occurs, the grouping with 20 groups can then be discarded without computing it, as it will never be the minimum. On the other hand, if we had first examined the grouping with 20 groups, all of the work would be rendered useless once the grouping with 2 groups is examined. This optimization of starting with the smallest groupings can be thought of as working up from a lower bound.

By using the combination of lower and upper bounds, an algorithm can work up from the lower bound while simultaneously trying to lower the upper bound. Once the two bounds meet, the best grouping has been found. More details regarding the specifics of this method are now explained in the pseudocode shown in the following sections. Section 6.1 first describes the initialization of the algorithm. Section 6.2 shows the algorithms for initializing a new GroupNode and generating the possible groupings from it. Section 6.3 then shows the algorithms for initializing a new SplitNode and generating the groupings from it.

Based on the best groupings generated by the algorithm, we then construct a schema decision tree. This is shown in section 6.5. Section 6.6 then shows how to integrate a schema decision tree into the JSON schema specification.

6.1 Starting the Algorithm

When the algorithm is started, a single GroupNode is created and initialized with all the objects of the array. It is assumed that each object has already been converted into its path notation representation. In addition to the objects, an empty list is also passed in for the *splitKeys* parameter. This list represents that no split operations have been applied to the group yet. A function call to the nodes *genGroupings* function is then performed to generate the best groupings. An initial upper bound of infinity is passed in as an argument—representing that no best group-

ings have yet to be generated. Pseudocode for this startup procedure is given in the following algorithm.

Algorithm 1: Initializing and Starting the Algorithm

Input: objects– List of JSON objects in path notation.
Output: List containing the best groupings.

1 root ← A new GroupNode initialized with objects and an empty list for splitKeys.
2 results ← root.genGroupings(∞)

6.2 GroupNode Algorithm

The purpose of a GroupNode is to represent a set of objects. When a split operation is performed on a GroupNode, the result is a set of GroupNodes. Generating the possible groupings for a GroupNode consists of performing a split operation on each key common to all objects in the node. The set of possible groupings is then the union of possible groupings generated for each split operation.

6.2.1 Initializing a New GroupNode

Algorithm 2: GroupNode: Initialization

Input: objects– List of JSON objects in path notation.
splitKeys– List of keys that have been used to filter all the array objects down to this group.

1 this.objects ← objects
2 this.splitKeys ← splitKeys
3 this.similarity ← sim(objects)

- Lines 1–2: Initialize *objects* and *splitKeys* to those passed into the function.
- Line 3: Calculate the similarity score of the objects in this group. This is performed once on initialization rather then computing it every time the similarity score is needed.

6.2.2 Generating Groupings for a GroupNode

Algorithm 3: GroupNode: genGroupings()

Input: upperBound– Max number of groups this group can be split into.
Output: List containing the best groupings.
Global : threshold– Minimum similarity requirement.

```
1  if this.similarity >= threshold then
2  |   Return 1 and a list containing a list with this group in it.
3  end
4  groupings ← empty-list, splits ← empty-list
5  splitOptions ← (⋂_{J∈this.objects} J) – this.splitKeys
6  foreach key in splitOptions do
7  |   Add a new SplitNode to splits passing in this.splitKeys, key, and
   |     this.objects.
8  end
9  foreach split in splits do
10 |   if split.similarity >= threshold ∧ split.numGroups < upperBound then
11 |   |   upperBound ← split.numGroups
12 |   end
13 end
14 foreach split in splits do
15 |   if split.numGroups > upperBound ∨ split.numGroups = 1 then
16 |   |   Remove split from splits.
17 |   end
18 end
19 Sort splits in ascending order by numGroups.
20 foreach split in splits do
21 |   splitNum, splitGroupings ← split.genGroupings()
22 |   if splitNum = 0 then
23 |   |   Continue to next iteration.
24 |   end
25 |   if splitNum = upperBound then
26 |   |   Add splitGroupings to groupings.
27 |   else if splitNum < upperBound then
28 |   |   upperBound ← splitNum
29 |   |   groupings ← splitGroupings
30 end
31 Filter groupings to only contains those with the highest similarity score.
32 if groupings is empty then  return 0, empty-list
33 else  return upperBound, groupings
```

- Lines 1–3: If the group itself already has a similarity score over the threshold, there is no need to further apply the split operation. Return a list containing a list with this group inside it. This represents that there is one possible grouping consisting of all the objects in one group. If the group does not have a similarity score over the threshold, then the group has to be split further. In this case, continue on with the algorithm.

- Lines 5–8: Determine the possible keys to split on by finding the set of keys that are common to all the objects. Remove keys that have already been split on to reach this point. For each of these keys, generate a new SplitNode.

- Lines 9–13: Check to see if any of the newly created SplitNodes satisfy the similarity threshold without needing to be split any further. Of the ones that due, find the one with the fewest groups, and set the number of groups in it as the new upper bound.

- Lines 14–18: Remove any SplitNode that results in more groups than the upper bound. These can be discarded as a better best grouping has already been found. Also, remove any SplitNode that results in only one group, as there is no reason to split on a path that results in the same group.

- Line 19: Sort the remaining SplitNodes in ascending order by the number of groups in the SplitNode. This results in the following loop first examining the SplitNode with the fewest groups. It then works its way up to the SplitNode with the most groups. This method increases the likelihood of lowering the upper bound the most, allowing more SplitNodes to be preemptively discarded.

- Line 21: Generate the possible groupings of the given SplitNode by calling its *genGrouping* function. Pseudocode for this algorithm is given in algorithm

5. This function returns the number of groups in the possible groupings and a list of the groupings.

- Lines 22–24: A return value of 0 for splitNum indicates that all valid groupings that can be generated by the current SplitNode have more groups then the current upper bound. All results from this SplitNode can be discarded as a better best grouping has already been found.
- Lines 25–26: If the SplitNode has resulted in groupings with the same number of groups as the upper bound, add these to the list of current groupings.
- Lines 27–29: If the SplitNode has resulted in groupings containing fewer grounds then the current upper bound, discard the current list of groupings, and set the list to the list of newly generated groupings. Lower the upper bound to the value of *splitNum* to reflect the new best grouping.
- Line 31: Filter out groupings that do not have the highest similarity score. This satisfies criteria 3 of the scoring criteria.
- Lines 32–33: If all the groupings with a similarity score over the threshold have more groups then the upper bound, return 0 and an empty list to indicate that this GroupNode cannot generate a better best grouping. Otherwise, return the new upper bound and the new groupings.

6.3 SplitNode Algorithm

The purpose of a SplitNode is to represent the GroupNodes generated when applying the split operation on a specific GroupNode for a given key. Generating the possible groupings for a SplitNode consists of generating the different combinations of the groups resulting from the split. For each group, either the group

itself can be included, if it satisfies the similarity threshold, or the group can be partitioned further with the resulting groups being included.

6.3.1 Initializing a New SplitNode

Algorithm 4: SplitNode: Initialization

Input: splitKeys– List of keys that have filtered all of the array objects down to the group this split operation is being applied on.
key– Key to split objects on.
objects– List of JSON objects in path notation.

```
1 this.splitKeys ← splitKeys
2 this.key ← key
3 this.objects ← objects
4 this.groups ← empty-list
5 foreach group in partition(objects, key) do
6     Add a new GroupNode to this.groups—passing in group and the
      concatenation of splitKeys with key.
7 end
8 this.numGroups ← |groups|
9 this.similarity ← simGroup(this.groups)
```

- Lines 1–3: Initialize *splitKeys*, *key*, and *objects* to the corresponding value passed into the function.
- Line 4: Initialize the *groups* variable to en empty list.
- Lines 5–7: Partition the objects based on the value of the given key. This means that all objects having the same value for the given key form a group. Create a new GroupNode for each of the resulting groups—passing in the objects of the new group, along with a list of the past split keys concatenated with the new key.
- Line 8: Set *numGroups* to the number of groups in the partition.
- Line 9: Calculate the similarity score for the set of groups. This is performed once on initialization and used whenever the similarity score is required.

6.3.2 Generating Groupings for a SplitNode

Algorithm 5: SplitNode: genGroupings()

Input: upperBound– Maximum number of groups that any group in this split can be partitioned into.
Output: List containing the best groupings.
Global : threshold– Minimum similarity requirement.

```
1  if this.numGroups > upperBound then
2  |  return 0, empty-list
3  end
4  if this.numGroups = upperBound ∧ this.similarity >= threshold then
5  |  return upperBound, this.groups
6  end
7  numAccounted ← this.numGroups
8  groupGroupings ← empty-list
9  foreach group in this.groups do
10 |  bound ← upperBound − numAccounted + 1
11 |  splitNum, validGroupings ← group.genGroupings(bound)
12 |  if splitNum = 0 then
13 |  |  return 0, empty-list
14 |  end
15 |  numAccounted ← numAccounted + splitNum − 1
16 |  Append validGroupings to groupGroupings
17 end
18 groupings ← empty-list
19 Add a list of groups to groupings for each permutation of groups in
     groupGroupings.
20 return numAccounted, groupings
```

- Lines 1–3: If the number of groups in this SplitNode is already greater then the upper bound, return 0 and an empty list to indicate that no better grouping can be generated.

- Lines 4–6: If the number of groups in this SplitNode is equal to the upper bound, then return a grouping consisting of the groups in this SplitNode.

- Line 7: *numAccounted* keeps a running tally of the number of groups already accounted for. Initially, this is set to the number of groups in the SplitNode

as splitting a group can only in more groups.

- Line 8: *groupGrouping* stores the possible groupings for each of the groups in the SplitNode. If a group itself is above the similarity threshold, *groupGrouping* just stores the group. If the group is below the threshold, it stores the best groupings for that group. For example, the first element of *groupGrouping* represents the possible groupings for the first group of the SplitNode, the second element represents the possible groupings for the second group, etc.

- Line 10: Calculate the new upper bound for the group currently being examined. This is done by taking the existing upper bound and subtracting the number of groups already allocated to the previously examined groups. Add 1 since one group is already accounted for (i.e. taking one group and splitting it into two groups only yields one additional group). For example, if the upper bound is ten and the first group examined requires three groups to get above the similarity threshold, any partition of second group can only consist of a maximum of seven groups.

- Lines 12–14: A return value of 0 for *splitNum* from the function indicates that the group cannot be partitioned into a grouping that is above the similarity threshold without exceeding the upper bound. Because of this, return 0 and an empty list indicating that this SplitNode cannot yield better results.

- Lines 15–16: Update the number of groups accounted for and add the list of possible groupings to *groupGrouping*.

- Lines 18-20: *groupGrouping* now contains a list of possible groupings for each group. To construct the possible groupings for the SplitNode, create a grouping for each combination of groups. This is done by choosing one grouping option for each group in the SplitNode.

6.4 Algorithm Remarks

A benefit of having the root node contain all objects in the array is that minimal computation is required for homogeneous arrays (ie. arrays where the objects all have the same structure). This is because the root node itself will have a similarity score over the threshold, and thus, return itself as the best grouping right away. Because of this, there is little overhead for checking most arrays. Only if the array is heterogeneous does the algorithm start splitting the group.

Another note about the algorithm is that it is possible for multiple best groupings to be returned; however, this is rare. The algorithm only returns multiple best groupings if two groupings have the same number of groups and the same similarity scores. In this scenario, either grouping can be used as the basis for the schema decision tree.

6.5 Constructing a Schema Decision Tree

Given an best grouping, a schema decision tree can be generated using the list of *splitKeys* contained within each GroupNode. The basic idea behind the construction process is to start with an empty decision tree, and add each group to the tree one at a time. Adding a group to the tree consists of starting at the initial decision node and iterating through the GroupNode's *splitKeys* list. Every time the group was partitioned during the generation process, the key used to partition it was appended to its *splitKeys* list. This means that the list is, in a sense, ordered; the first key in every *splitKeys* list is the key used to partition the initial GroupNode that contained all the objects.

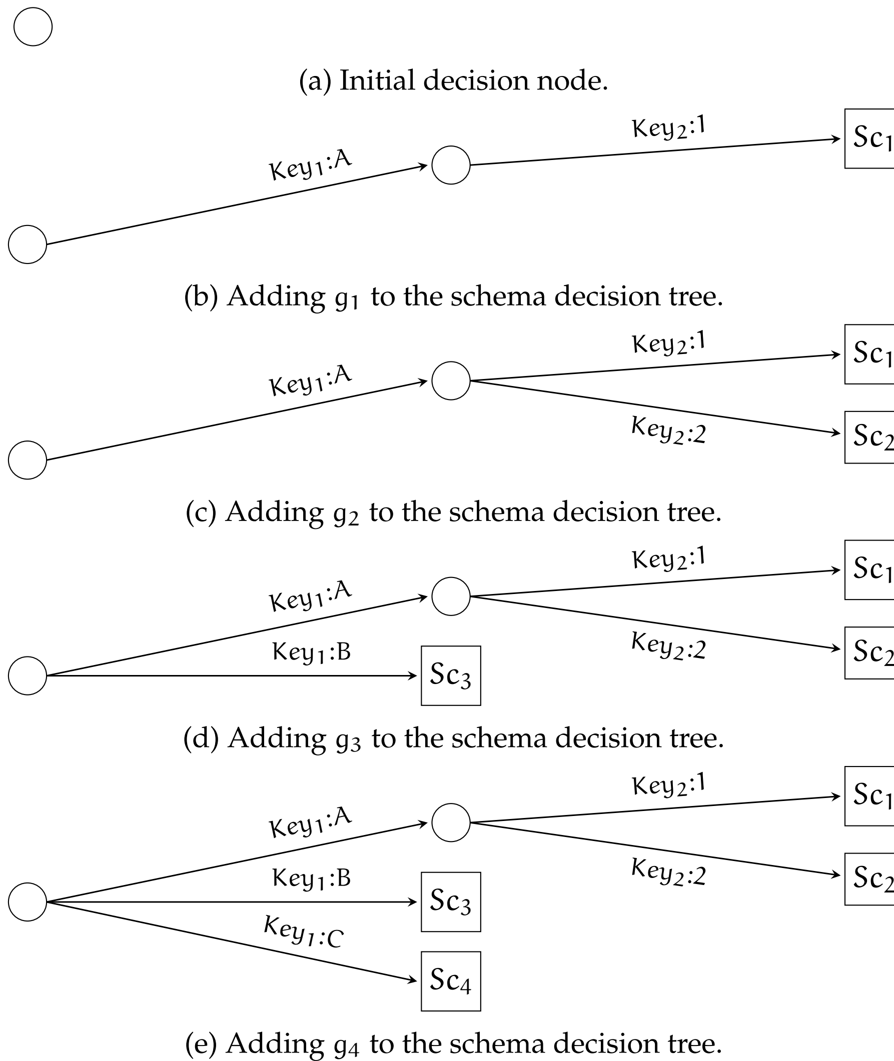
For each key in a group's *splitKeys*, if the key (and corresponding value) is already present in the tree, then the group moves down the tree to the next decision node. If the key is not present in the tree, a new branch is added to the current de-

cision node with the give key. If this was the last key in the list, the branch points to a schema generated for the objects in the group. Otherwise, the branch points to another decision node.

Figure 6.2 shows an example of the construction process for the 4 groups shown in table 6.1. Initially, the schema decision tree starts with an empty decision node as shown in 6.2a. In figure 6.2b, the first group, g_1, is added to the tree by taking the first key in its *splitKeys* list and checking if it is already present at the decision node. Since it is not present, a new branch is created pointing to another decision node. This process repeats for the second key, except this time, the branch points to a schema, as it is the last key in the list. This schema is generated by passing the objects of the group into a traditional schema generation tool. When adding g_2 to the tree in figure 6.2c, the group sees that the first key is already present at the initial decision node. It instead uses the existing branch to move down the tree rather then creating a duplicate branch. Once this process has been performed on every group, the final schema decision tree is generated (as shown in figure 6.2e).

Group	*splitKeys* list from the GroupNode
g_1	Key_1:A, Key_2:1
g_2	Key_1:A, Key_2:2
g_3	Key_1:B
g_4	Key_1:C

Table 6.1: A list of the groups in the best grouping, along with their corresponding *splitKeys* list.



(a) Initial decision node.

(b) Adding g_1 to the schema decision tree.

(c) Adding g_2 to the schema decision tree.

(d) Adding g_3 to the schema decision tree.

(e) Adding g_4 to the schema decision tree.

Figure 6.2: Constructing a schema decision tree for the groups in table 6.1.

6.6 Integrating into JSON Schema

Based on the schema decision tree constructed in figure 6.2, we now show how it can be integrated into a JSON schema document. As the actual document would span multiple pages and be difficult to read, we instead include snippets showcasing the two main components. These components are the definition section where individual schemas are stored, and decision nodes specifying how to move down the tree.

First, the definition section of a JSON schema document is designed for defining sub-schemas. These can then be referenced throughout the rest of document using the notation *"$ref": "#/definitions/SubSchemaName"*. As a schema is generated for each type of structure, we choose to store these schemas in this section and reference them throughout the document. The purpose for this is to break up the document into more manageable pieces and improve readability.

Figure 6.3 contains the definition section for two sub-schemas named *Schema1* and *Schema2*. *Schema1* is defined in lines 2–17, and *schema2* is defined in lines 18–33. Lines 3 and 19 specify that each schema is for a JSON object. Lines 4–12 and 20–28 then contain a list of properties that an object must contain to satisfy the given schema. The identification keys are specified in lines 5–10 and 21–26. Their corresponding enum value specifies the value of the identification key for the given schema. Lines 11 and 27 are where the rest of the schema for the given structure type would go. Finally, lines 13–16 and 29–32 specify that the identification keys are required to appear in the object. Additional keys can be included here depending on what the rest of the schema consists of.

```
1  "definitions": {
2    "Schema1": {
3      "type": "object",
4      "properties": {
5        "key1": {
6          "enum": ["A"],
7        },
8        "key2": {
9          "enum": ["1"],
10       },
11       ...
12     },
13     "required": [
14       "key1",
15       "key2"
16     ],
17   },
18   "Schema2": {
19     "type": "object",
20     "properties": {
21        "key1": {
22         "enum": ["A"],
23       },
24       "key2": {
25         "enum": ["2"],
26       },
27       ...
28     },
29     "required": [
30       "key1",
31       "key2"
32     ],
33 },
```

Figure 6.3: The section of the resulting JSON schema document containing the definitions for the different sub-schemas tailored to the different types of structure.

Second, each decision node is implemented using a combination of the *anyOf* and *allOf* syntaxes. The *anyOf* syntax specifies that at least one of the following schemas has to be satisfied for the data to be valid. Likewise, the *allOf* syntax specifies that all of the following schemas must be satisfied for the data to be valid. By nesting these within each other, we are able to group validation conditions together.

Figure 6.4 shows the schema for the second decision node in figure 6.2. For any JSON object to be valid, it has to satisfy either the schema defined in lines 2–17 or the schema defined in lines 18–33. Both of these schemas then consist of an *allOf* syntax with two further schemas defined. Lines 4–10 and 20–26 are first used to perform a lookup on the given identification key. If its value matches that specified in the enum, it moves on to the second part of the *allOf* syntax in lines 13–15 and 29–31. This schema consists of a reference that sends the document on to either another decision node or one of the schemas defined in the definition section.

```
1  "anyOf": [
2      {
3        "allOf": [
4          {
5            "type": "object",
6            "properties": {
7              "key2": {
8                "enum": ["1"]
9              }
10           },
11           "required": ["key2"]
12         },
13         {
14           "$ref": "#/definitions/Schema1"
15         }
16       ]
17     },
18     {
19       "allOf": [
20         {
21           "type": "object",
22           "properties": {
23             "key2": {
24               "enum": ["2"]
25             }
26           },
27           "required": ["key2"]
28         },
29         {
30           "$ref": "#/definitions/Schema2"
31         }
32       ]
33     }
34   ]
```

Figure 6.4: Section of the resulting JSON schema document containing a decision node.

Chapter 7

Evaluation and Analysis

7.1 Algorithm Walkthrough

To showcase the logic behind the algorithm, we now work through the process the algorithm takes when partitioning a dataset. This dataset consists of an array of 50 JSON objects returned from the iTunes Search API. To simplify the problem for explanation purposes, only 5 structure types were included in the dataset; however, the process is the same regardless of the number of structure types. For this example, we assume a similarity threshold of 0.70.

To begin, all objects of the array are placed into a single group. As this group has a similarity score of 0.2 and a simGroup score of 0, we know that the split operation has to be applied at least once. Common to all the objects are 8 keys resulting in 8 SplitNodes being generated—one for each key. Table 7.1 shows what key each SplitNode was generated from, how many groups resulted from the split, what the average similarity of the groups was, and what the *simGroup* score was. Looking at this table, we see that the splits based on *collectionExplicitness* and *wrapperType* each resulted in only a few groups being generated; however, they do not satisfy the similarity threshold.

Of the options that do satisfy the threshold, *collectionPrice* has the lowest number of groups at 18. As such, this grouping becomes the current best grouping, and the lower bound is reduced down to 18. Any grouping having more then 18 groups can be discarded. Two scenarios now exist. Scenario one is that either the groups generated by splitting on *wrapperType* or *collectionExplicitness* can be further split into less then 18 groups that all satisfy the threshold. In this case, the results from *collectionPrice* can be discarded, as better groupings have been found. Scenario two is that none of the groups generated by splitting on *wrapperType* or *collectionExplicitness* can be further split. In this case, *collectionPrice* is the best grouping.

Split Key	Num. Groups	Average Similarity	SimGroup Score
collectionPrice	18	0.905	0.905
artistName	35	0.986	0.986
releaseDate	41	0.984	0.984
collectionExplicitness	3	0.461	0
artworkUrl60	46	0.988	0.988
wrapperType	2	0.672	0
primaryGenreName	24	0.972	0.972
artworkUrl100	44	0.988	0.988

Table 7.1: A list of split operations for a single group containing all objects.

The only way to determine which scenario is true involves further applying the split operation to the groups generated by *wrapperType* and *collectionExplicitness*. We first start with *wrapperType* as it has the fewest groups. Splitting on *wrapperType* results in two groups. One has an average similarity of 0.391, and the other has an average similarity of 0.952. As the second group already satisfies the threshold, it does not have to be split further; only the first group does. Furthermore, since

the combination of the 2 groups has an upper bound of 18 groups, the first group can, at most, be split into 17 groups. Because of this, 17 can be thought of as the new upper bound when examining that group. Table 7.2 shows the different split operations that can be applied to this group. Of note is the increase in options for split keys. In addition to the unused keys carried down from table 7.1, additional keys now exists. This is due to the existence of keys that are in common to all the objects of this group but not all the objects in general. As such, they did not appear in the first table but do in this table.

Looking at this table, we see that 6 groupings have resulted in less than 18 groups. Of these, splitting on *kind* has resulted in only 4 groups that are all over the threshold. Before we can say this is the new best grouping and return to the previous recursive layer, we need to check two other options. Both *trackExplicitness* and *collectionExplicitness* resulted in fewer groups but did not meet the similarity threshold. Thus, we need to verify that they cannot be further split.

We first check to see if *trackExplicitness* can be split into at most 4 groups. This grouping consists of 3 groups having average similarity scores of 0.537, 0.600, and 0.914. Both the first and second groups need to be split further as they are currently below the threshold. However, doing this results in at least 5 groups which is already above the threshold. As such, splitting on *trackExplicitness* cannot result in a better grouping then we have already found. Likewise, splitting on *collectionExplicitness* also results in 3 groups having average similarity scores of 0.537, 0.600, and 0.914. For the same reasons as with *trackExplicitness*, splitting on *collectionExplicitness* cannot result in a better grouping either. Thus, no groupings exist with fewer then 4 groups, meaning that *kind* is the current best best grouping.

Split Key	Num. Groups	Average Similarity	SimGroup Score
trackName	28	0.977	0.977
trackViewUrl	30	0.984	0.984
trackExplicitness	3	0.684	0
collectionPrice	9	0.939	0.939
trackCensoredName	28	0.977	0.977
artistName	23	0.978	0.978
kind	4	0.928	0.928
trackId	30	0.984	0.984
trackPrice	6	0.933	0.933
releaseDate	25	0.976	0.976
artworkUrl30	24	0.978	0.978
collectionExplicitness	3	0.681	0
artworkUrl60	24	0.978	0.978
primaryGenreName	17	0.965	0.965
artworkUrl100	24	0.978	0.978

Table 7.2: A list of different split operations for one of the groups generated by the *wrapperType* split.

Going back up a layer of recursion to table 7.1, we had originally split *wrapperType* into two groups. The second group already satisfied the threshold, and we just found that the first group can be split into 4 groups. This means that a valid grouping now exists with 5 groups, and the upper bound can be further lowered from 18 down to 5.

The only remaining option left to check is now *collectionExplicitness*. It currently has 3 groups with average similarity scores of 0.295, 0.600, and 0.486. As all these groups require at least one further split operation, the minimum number of groups

we could receive is 6. As such, we know that *collectionExplicitness* cannot result in a better split then we have already found.

At this point, we have explored every option that could result in fewer groups, meaning that the lower bound has reached the upper bound. Thus, the best grouping is generated by first splitting on *wrapperType* and then splitting one of the groups on *kind*.

7.2 Evaluation

Two main areas exist that we are interested in examining. The first area is the time it takes the algorithm in chapter 6 to generate the best grouping, and the second area is the time it takes to validate a JSON array using a schema decision tree. To analyze these areas, we run our algorithm against three different datasets and analyze the results in the following sections. Dataset one is based on the iTunes Search API that has been used as a running example throughout this thesis; dataset two is based on the Open Movie Database API, and dataset three is based on the Spotify Search API. Web APIs were the main source of data due to their ability to generate dataset of differing sizes with relative ease.

For each dataset, we present two graphs. The first graph analyzes the runtime of the algorithm for various similarity thresholds and array sizes. This is done in figures 7.2, 7.5, and 7.8. The second graph analyzes how long it it takes JSON arrays of varying sizes to be validated. This is done in figures 7.3, 7.6, and 7.9. In each of these graphs, each array was validated using one of three different methods.

Method one consists of a single schema. This schema was generated by taking the schemas created for the different structure types and arranging them in a linear order. In this sense, the schema for each structure type can be thought of as a sub-schema. An object is validated against the schema by comparing it to the first

sub-schema. If that sub-schema does not validate the object, it is compared against the second sub-schema. This process repeats until the object either satisfies one of the sub-schemas or is rejected.

Method two again consists of a single schema. Instead of arranging the sub-schemas in a linear order, this schema integrates the schema decision tree into it. This is done using the approach outlined in section 6.6. The sub-schemas used in this tree are the same sub-schemas used in the previous method.

One major drawback with the JSON schema format is that is does not support variables. This is important because it prevents the full potential of the schema decision tree from being used. Instead of performing a single key lookup and reusing the value, the schema in method two can only check to see if a key has a specific value. Because of this, a key lookup has to be performed for each value an identification key can have.

Based on this drawback, method three consists of a program that models the schema decision tree. For each object, the program first performs a lookup on the identification key and stores the result in a variable. This variable is then compared against the possible values the identification key can have to determine which branch of the tree it should take. If the branch leads to another decision node, the process repeats. If the branch leads to a schema, then that schema is used to validate the object. This method differs from method two in that there is no longer a single schema document. Instead, the sub-schemas are kept separate, and the program decided which schema an object should be compared against.

All experiments were performed on a 2014 MacBook Air containing a 1.4 GHz Intel Core i5 processor and 8GB memory. Each data point consists of the average time of 10 runs. Each dataset was also verified to contain all structure types. The generated schema decision trees are based on the best groupings found using a similarity threshold of 0.7. This number was chosen based on observations during

the algorithm development process. We find that it gives a good balance between determining the best grouping while still allowing the possibility of some variation within a particular structure type.

7.2.1 Generating the Datasets

For each of the three APIs we are using in our analysis, we first generated a dataset consisting of 15000 JSON objects obtained from heterogeneous arrays in the API's responses. As we are only focusing on JSON arrays, we only extracted the array from the response and discarded the rest.

Table 7.3 shows the API endpoint URLs used for each of the datasets. For the iTunes Search API, we used the search endpoint and the *term* parameter. The value of this parameter is the search query, and the API will return results that it thinks are relevant to it. No authentication is required to use this API

Dataset	API Endpoint URL
iTunes Search API	• https://itunes.apple.com/search?term=<TERM>
OMDB API	• http://www.omdbapi.com/?s=<TERM> &apikey=<KEY> &page=<NUM> • http://www.omdbapi.com/?i=<ID> &apikey=<KEY>
Spotify Search API	• https://api.spotify.com/v1/search?q=<TERM> &type=album,track,artist,playlist,show,episode

Table 7.3: The API Request Endpoint(s) used for each dataset.

For the OMDB API, we used two endpoints. The first is the search endpoint, and it returns information that the API thinks is relevant to the given term. Each response only contains an array of 10 objects with more being available through subsequent requests. This is done by including the optional *page* parameter. For

example, setting *page* to 2 gives the next 10 results. This endpoint only returns a few value for each object in the response, namely, a title, year, ID, type, and poster. To get the rest of the information for each item, we performed an addition request to their ID endpoint that returns the full JSON object for a given ID. An API key is required to use this API, and can be obtained for free through their website. This key is passed in through the *apikey* parameter in each request.

For the Spotify Search API, we used the search endpoint and the *q* parameter to pass in a search query term. In addition, we also specified a *type* parameter. This parameter tells the API what type of information we want in our results. As we are collecting a dataset of all structure types, we list all options so that we receive information for each type. An API key is required to use this endpoint, and can be obtained for free through their website. This key is passed in as a HTTP header for each API request.

Table 7.4 shows the number of request made to each API, as well as the number of objects returned in a response. For the iTunes Search API, we made 300 requests and received 50 objects back per response (50 is the default number returned).

Dataset	API Requests	Objects Per Response	JSON Objects
iTunes Search API	300	50	15000
OMDB API	27	23–3080	15000
Spotify Search API	50	300	15000

Table 7.4: An overview of the number of requests made to each API, and the number of objects returned for each request.

For the OMDB API, we made 17 requests to the search API and received between 23 and 3080 objects per response. The term *blue* gave us the largest at 3080, and the term *sponge* gave us the smallest at 23. Finally, for the Spotify Search API,

we made 50 requests and received 300 objects per response (50 objects for each of the 6 types we listed in the request).

Search terms for all APIs were made to ensure that we got a variety of data that included all structure types. Furthermore, when selecting which data to pass into our algorithm, we randomly chose n objects from the dataset and then verified that all structure types were included. If not, then we repeated the process until this condition was satisfied.

7.2.2 iTunes Search API Dataset

The iTunes Search API is a web API that returns information regarding products available in the iTunes store [1]. Figure 7.1 shows the generated schema decision tree consisting of two identification keys. Objects are first split based on the *wrapperType* key. Objects then having the value *track* are split again based on the *kind* key.

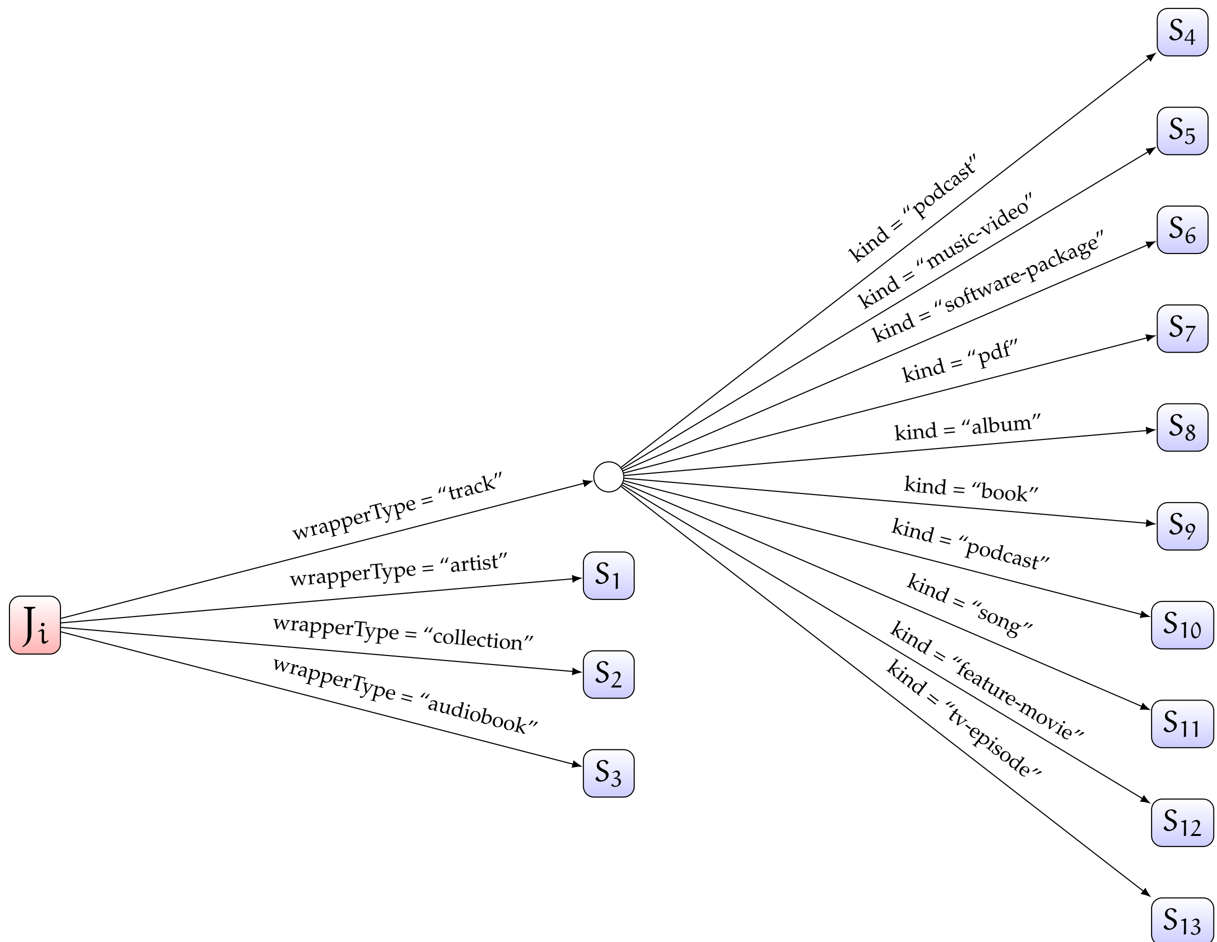


Figure 7.1: Schema decision tree generated from the iTunes Search API dataset.

7.2.2.1 Schema Generation Time Comparison

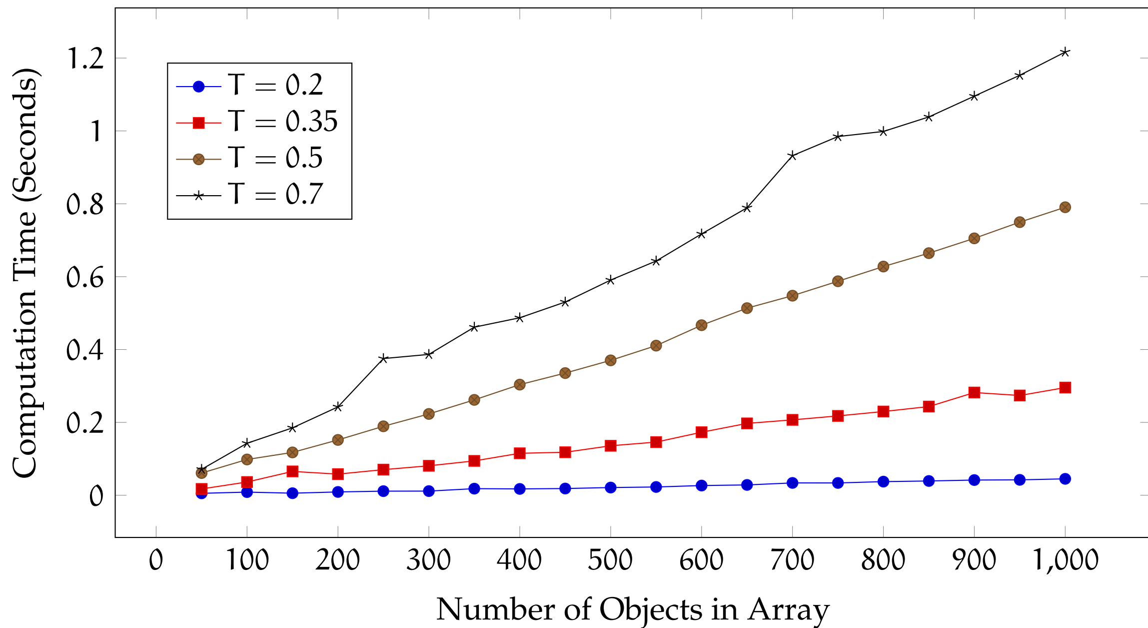


Figure 7.2: Comparing the computation time of the iTunes Search API dataset for various similarity thresholds (T) and various input array sizes.

Figure 7.2 shows how long it took the algorithm in chapter 6 to generate the best grouping for various similarity thresholds and input array sizes. Setting the threshold to 0.2 results in a single group containing all the objects. Because of this, the only computation involved is a single similarity calculation resulting in a near constant runtime. When the threshold is increased to 0.35, the best grouping consists of four groups generated by splitting only on *wrapperType*. A similarity threshold of 0.5 results in six groups generated by first splitting on *wrapperType* and then further splitting one of the groups based on *trackExplicitness*. Finally, a similarity threshold of 0.7 results in the groups used to generate the schema decision tree in figure 7.1.

7.2.2.2 Schema Validation Time Comparison

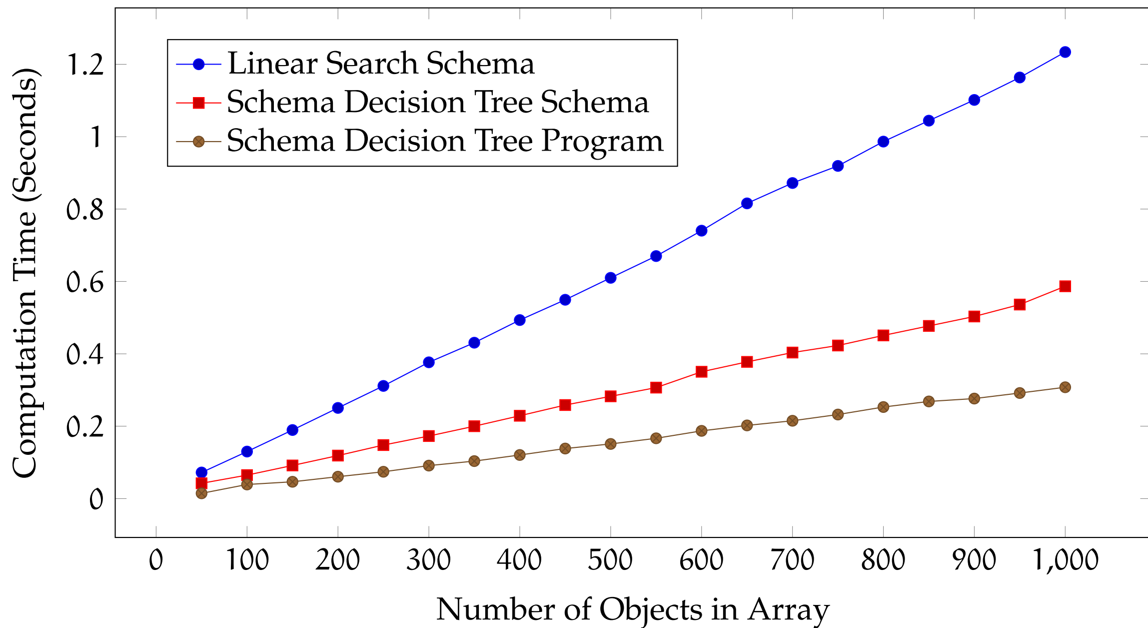


Figure 7.3: Comparing the validation time of the iTunes Search API dataset for three validation methods.

Figure 7.3 now shows the time it took to validate arrays of varying sizes using the three methods. Based on this graph, we see a significant improvement when the schema integrates the schema decision tree compared to when the schema arranges the sub-schemas in a linear order. This is likely due to a few factors. First, there is a significant overlap of keys among the different structures. This means that the time it takes to invalidate an object increases, as the overlapping part may be checked before the non-overlapping part. Second, having a decision node allows multiple schemas to be skipped over.

Comparing the schema implementation of the schema decision tree to the program implementation, we see another improvement. This is likely due to the reduction in key lookups that have to be performed. For example, in the second decision node, only a single key lookup has to be performed to determine which

schema the object should be compared against. In the schema implementation, up to 10 key lookups may have to be performed—one for each sub-schema.

7.2.3 Open Movie Database (OMDb) API Dataset

Dataset two is based on the Open Movie Database API [73]. This is a web API that provides information about different movies and television series—such as actors, producers, etc. Figure 7.4 shows the resulting schema decision tree. Compared to the tree generated for the iTunes data set, this tree is relatively simple with only a single identification key and three structure types.

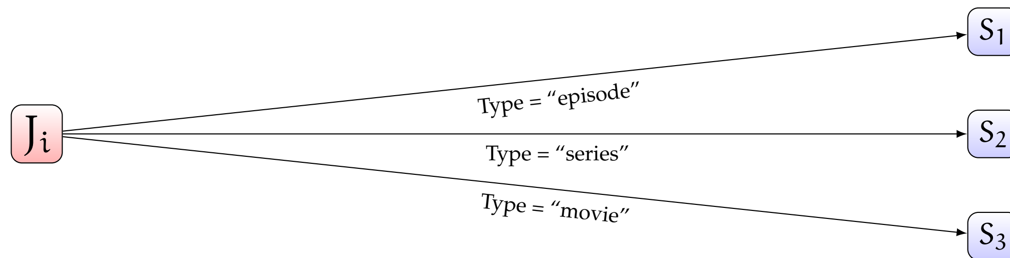


Figure 7.4: Schema decision tree generated from the Open Movie Database dataset.

7.2.3.1 Schema Generation Time Comparison

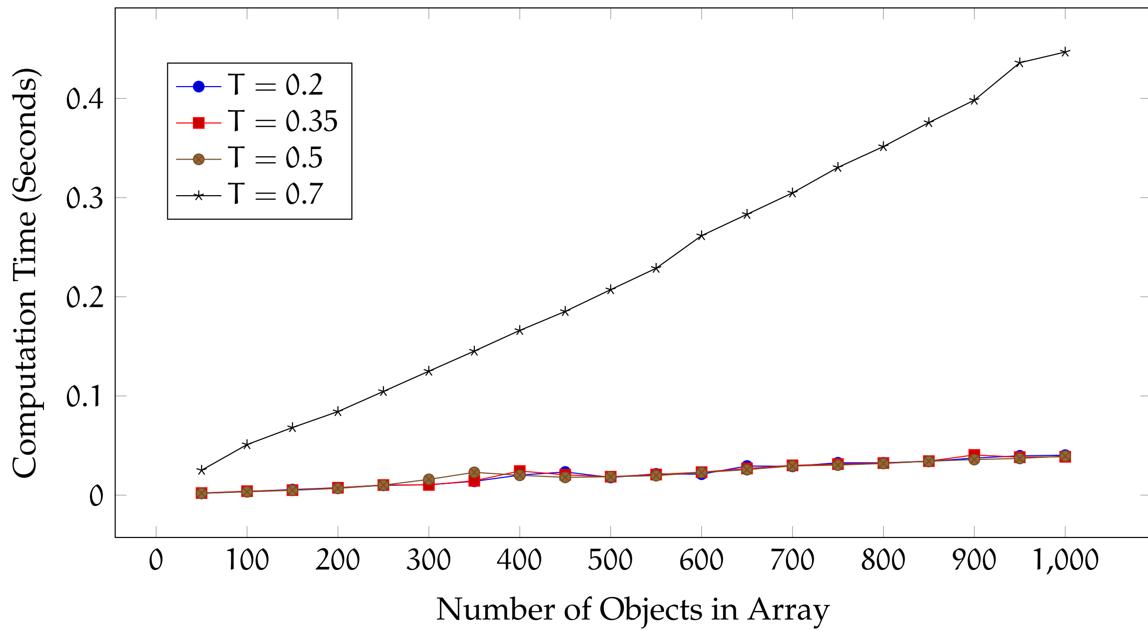


Figure 7.5: Comparing the computation time of the Open Movie Database dataset for various similarity thresholds (T) and various input data sizes.

Figure 7.5 shows the time it took the algorithm to generate the best grouping. When the similarity threshold was set to 0.2, 0.35, or 0.5, all objects were placed into a single group. Because of this, they all have near identical execution times. When the similarity score was 0.7, a single split based on *Type* occurred resulting in three groups.

7.2.3.2 Schema Validation Time Comparison

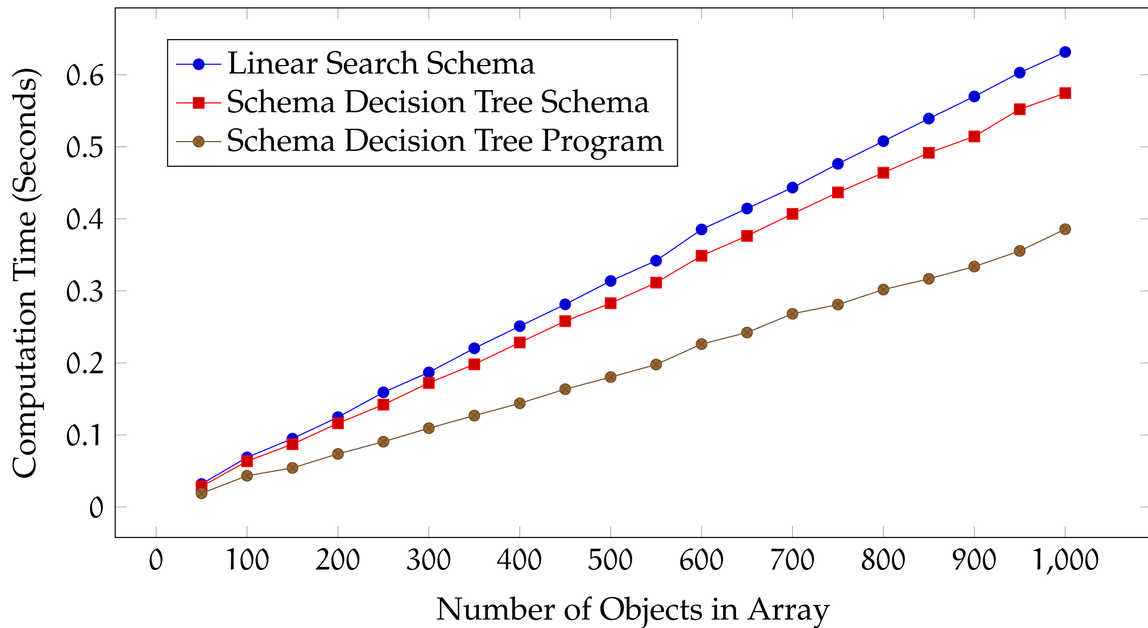


Figure 7.6: Comparing the validation time of the Open Movie Database API dataset for three validation methods.

Figure 7.6 shows the time it took to validate arrays of varying sizes using the three methods. We see that the schema decision tree version only had slightly reduced validation times. We believe the reason for this is due to two factors. First, only having three structure types means that an object can be invalidated relatively quickly. Second, the schema decision tree version still has to perform a key lookup to determine the identification key's value.

Comparing the program implementation of the schema decision tree to the other two, we see a significant improvement. This provides support for the second factor having a major contribution for the similar runtimes found in the schema versions.

7.2.4 Spotify Search API Dataset

The third dataset was generated from the Spotify Search API [74]. Like the iTunes Search API, this API returns information related to different songs, artists, shows, etc. available on the Spotify platform. Figure 7.7 shows the resulting schema decision tree. Like in the previous dataset, this decision tree consists of a single identification key and six structure types.

One note for this dataset is that Spotify does not combine together different types of structure into a single array. Instead, an API response contains an array for each type of structure (i.e. an array just for objects of the artist structure type, an array just for objects of the album structure type, etc). As each structure also contains a key identifying its type, we still felt this was a useful dataset to evaluate the schema decision tree concept. Thus, the dataset was generated by combining together the different arrays into a single array.

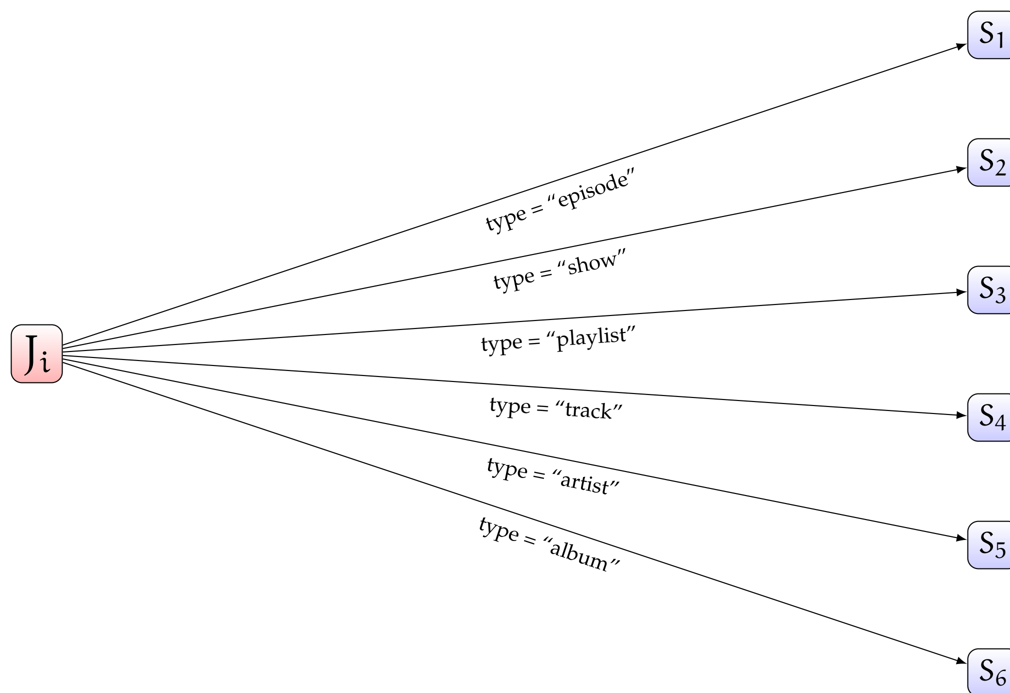


Figure 7.7: Schema decision tree generated from the Spotify Search API dataset.

7.2.4.1 Schema Generation Time Comparison

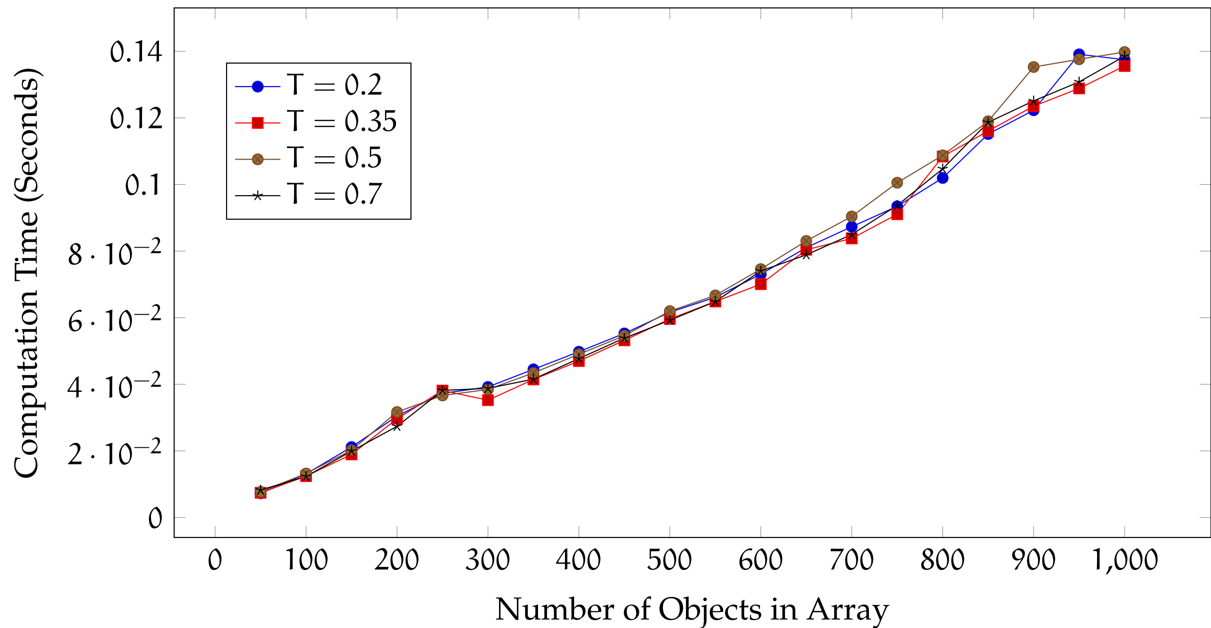


Figure 7.8: Comparing the computation time of the Spotify Search API dataset for various similarity thresholds (T) and various input data sizes.

Figure 7.8 shows the time it took to generate the best grouping for various similarity thresholds and various input array sizes. All similarity thresholds resulted in roughly equal execution times due to there being very little overlap among the different types of structures. For instance, only four keys are in common to all the objects.

7.2.4.2 Schema Validation Time Comparison

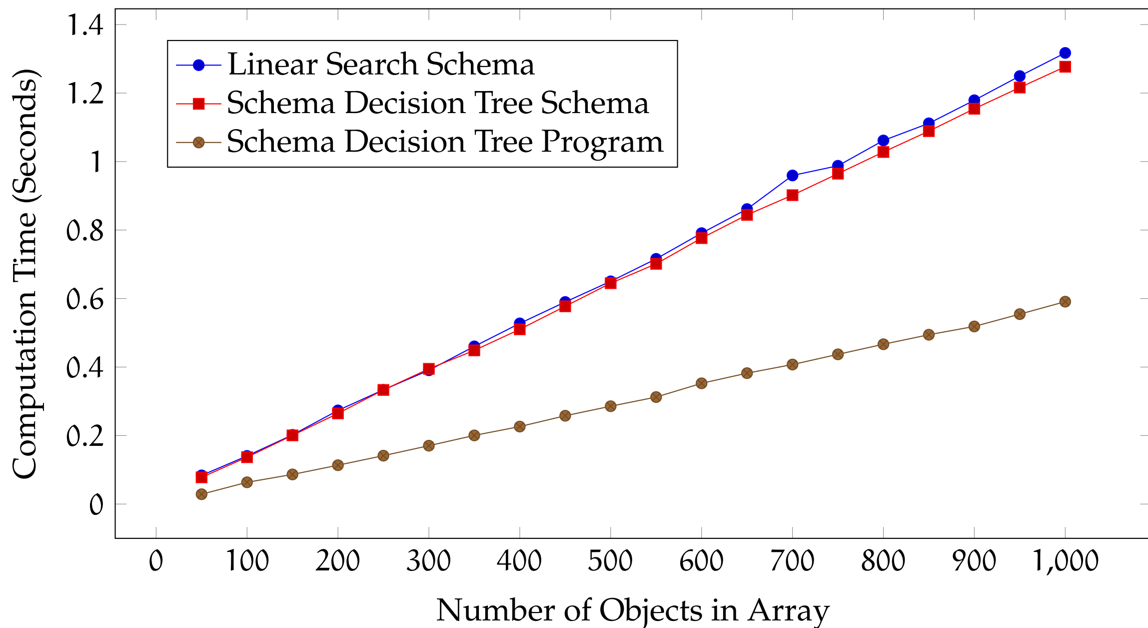


Figure 7.9: Comparing the validation time of the Spotify Search API dataset for three validation methods.

Figure 7.9 shows the time it took to validate arrays of varying sizes using the three methods. Looking at the graph, we see very similar validation times between the schema implementations. This slight reducing is comparable to the validation times for the schemas in the OMDB dataset (figure 7.6). We believe this is also due to similar reasons discussed there.

Unlike the OMDB dataset, however, the validation times for the two schemas in the Spotify dataset are nearly identical. This is likely due to very little overlap among the structure types. Because of this, an object can be invalidated in roughly the same time it takes to check the value of the identification key. This is shown when looking at the validation times for the program implementation. The program used the same sub-schemas but sees a significant improvement due to no longer having to check each value for an identification key.

7.3 Runtime Complexity Analysis

The challenge with determining the runtime complexity of the algorithm is that it is largely dependent on the structure of the data passed into it. Let J be the number of JSON objects in an array and n be the maximum number of keys in one of the objects.

The best case scenario occurs when the initial group (containing all the objects) already has a similarity score above the threshold. In this case, no split operations have to be performed. Calculating the runtime complexity of the algorithm is then simply calculating the runtime complexity of the similarity calculation. As each JSON object is treated as a set of keys, this problem is equivalent to determining the complexity of the intersection and union of J sets. Both of these operations can be done in $\mathcal{O}(Jn)$ time through the use of data structures that have constant time key lookups (e.g. hash sets in Java). This process works by treating the first set as the base set. All other $J-1$ sets are then iterated through. Each key within a set is compared against the base set to see if it is present. In the case of the union operation, the key is added to the base set if it is not already present. Once all the sets have been iterated over, the base set contains the union. In the case of the intersection operation, an integer count is associated with each key in the base set and is incremented when a lookup on that key is performed. After iterating over all the sets, the intersection of the sets consists of the keys in the base set having a count equal to the number of sets. As each set has at most n keys, the runtime complexity of the similarity calculation is $\mathcal{O}(Jn)$.

Based on this, the runtime complexity of a single split operation can now be calculated. For one split operation, there is a cost of $\mathcal{O}(J)$ to partition all the objects into their corresponding groups. Calculating the similarity score of the resulting groups then has a cost of $\mathcal{O}(Jn)$ as all the objects of the original group are still present. Adding these two complexities together gives a final complexity of $\mathcal{O}(Jn)$.

When we are dealing with the same dataset, we can treat n as a constant, as objects of the same structure type have roughly the same number of keys. With this assumption, the runtime complexity of the entire algorithm would then be dependent on the number of objects in the array and the number of split operations performed. We can again assume that the number of split operations will be a constant value due to the same groups being generated every time. If we let s be the number of split operations performed for a given dataset, we end up with a linear runtime complexity of $\mathcal{O}(Jns)$ with J and s being constants. This is shown in figures 7.2, 7.5, and 7.8, where all datasets have a near linear runtime.

Trying to define s in terms of J and n is a significantly more challenging problem. As such, we leave it as a future research direction and instead discuss some of the major factors that would need to be considered. The main issue is that multiple factors influence how many split operations are performed.

The first factor is comparing the size of the intersection to the size of the union for a given group. This is important because these are the two components of the similarity score. Maximizing the number of splits for a group involves maximizing the size of the intersection. If the intersection is too large, the groups similarity score will already be above the threshold, and thus, does not need to be split further. This means that the most split operations occur when the similarity score is just underneath the threshold.

The issue with only considering this is that a split operation always results in at least two groups—both having a higher similarity score. If the original group's similarity score was just underneath the threshold, it is very likely that the resulting groups will have similarity scores over the threshold, and thus, no further split operations are performed. This is largely due to the fact that the size of the union set will shrink, as objects containing unique keys are separated into different groups.

Consequently, there is a trade-off between having a high initial similarity score that results in many split operations occurring early on, and having a low initial similarity score that results in many layers of recursion. In essence, the first option results in a wide but short tree, whereas the second option results in a thin but tall tree.

The second factor that influences the number of split operations is how many groups are generated for each split. To maximize the number of splits, it is ideal to have a large number of groups that can each be split further. However, each split has to partition the same number of objects. This results in either many groups containing a few objects or a few groups containing many objects. The importance of this consideration is that the maximum number of split operations that can be recursively performed on a large group is greater then a small group. This is due to the fact that a split operation always results in groups of fewer objects. As such, the number of objects in a group influences the maximum layers of recursion that could be applied.

Similar to the first factor, initially generating many groups in a split results in a wide by short tree, whereas generating a few groups with many objects results in a thin but tall tree. In both factors, the solution that results in the most split operations is likely somewhere between the two extremes.

Chapter 8

Conclusion

The Javascript Object Notation document format is one of the prominent formats for modeling semi-structured data. Because of its popularity, the idea of a schema document emerged as a method of validating the structure and content of other JSON documents. While the general problem of schema generation has been previously explored, the problem of schema generation for heterogeneous JSON arrays has not been adequately addressed. Existing schema generation tools have worked off the assumption that all JSON objects in an array should have the same structure. As a result, these tools only generate a single schema that combines all objects together—regardless of if they have different types of structures.

This thesis looks to address this problem by instead generating a schema for each of the structures types found in a JSON array. As an array is designed to be processed as a single unit, objects in a heterogeneous array usually contain a set of keys whose purpose is to identify the objects structure. This allows programs processing the array to know how to proceed for each object. In order to detect these identification keys, we design an algorithm that recursively partitions the objects of an array based on a split operation we define. In essence, this operation chooses a key and partitions the objects based on the value for that key; objects having the

same value are placed in the same group. Using the identification keys generated by our algorithm, we build a schema decision tree to help in the validation process.

8.1 Future Work

8.1.1 Validating Identification Keys

The main challenge addressed in this thesis is determining which keys are used for identification purposes. To do this, we work off the assumption that the groups generated by splitting on an identification key will have a greater similarity score when compared to groups generated by splitting on a non-identification key. While we believe this assumption to be largely valid, it is still possible for the algorithm to return a grouping that split on a non-identification keys. Determining when this occurs is predominantly based on the structure and values of the data passed into it.

One idea to help detect when this occurs is to adopt an approach commonly used in machine learning and statistics. This approach is where the input data is first partitioned into two sets. The first set is used to train the model, and the second set is used to test the accuracy of it [75]. Using this idea, objects of a given JSON array could be first partitioned into two sets. The first set would be used to generate the schema decision tree. Objects of the second set would then be passed through the tree to see if they are validated correctly. If one of the objects is not validated, then either the resulting grouping applied the split operation on a non-identification key or the object represented an entity that was not part of the training data set. How to determine which scenario is true could be based on how many objects were not validated. If many objects were not valid, option one is likely. If only a few objects were not valid, option two is more likely.

8.1.2 Dynamic Similarity Threshold

The algorithm presented in this work assumes a static similarity threshold. This, however, runs the risk of either being too low and not detecting all identification keys or being too high and having non-identification keys be part of the best grouping. Determining when this occurs is mainly dependent on the dataset. A different approach could instead be to make the threshold dynamic. This would allow the threshold to take on different values for different datasets—or even change during the algorithm itself.

This idea of a dynamic threshold could also be used in conjunction with the validation method discussed above to better refine the results. If it was determined that the best grouping includes a non-identification key, the dynamic threshold could be lowered to see if more accurate results can be found. Likewise, if the best grouping validates all the objects in the test data, the dynamic threshold could be raised to see if more identification keys could be detected. Furthermore, the algorithm itself could potentially be extended to return groupings at different similarity thresholds.

8.1.3 Expanded API Study

For the analysis done in this thesis, we focused on datasets generated by three web APIs, namely, the iTunes Search API, the Open Movie Database API, and the Spotify Search API. While these datasets do showcase how our algorithm performs on real word data, we think it would be beneficial to perform a more in-depth analysis on a larger sample size. In particular, the three data sources for our analysis are focused on the music / entertainment sectors. We would like to expend this to include other industries and vendors, as they might use the JSON format in different ways.

Furthermore, the focus on our work was on heterogeneous JSON arrays, and as such, we only generated the schema for that part of the JSON documents. Another area we would like to look at in the future is on expanding our work to identification keys in general rather then focusing on identification keys only present in arrays. For example, Amazon Web Services (AWS) allows users to define Identity and Access Management (IAM) policies through JSON documents. Over time, the format for these documents has evolved, and now, two different version exist. Users can define which version of the format they are using by defining a *version* key in the root of the JSON document that either contains the date *2008-10-17* or *2012-10-17*. This key acts like a unique identifier for the JSON document even though the data is not part of a heterogeneous array. We think it would be very beneficial to expand our work to also cover scenarios like this as it would open up more data sources that we could use for analysis.

Bibliography

[1] Apple Inc., "iTunes Search API," 2019. Accessed on 02-11-2019.

[2] Quicktype, "Json schema generator," 2019. Accessed on 18-11-2019.

[3] Google Inc., "Google Trends," 2019. Accessed on 04-11-2020.

[4] Stack Overflow Inc., "Stack Overflow Trends," 2019. Accessed on 04-11-2020.

[5] JSONschema.net, 2012. Accessed on 18-11-2019.

[6] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. USA: McGraw-Hill, Inc., 6 ed., 2011.

[7] R. Bahta and M. Atay, "Translating json data into relational data using schema-oblivious approaches," in *Proceedings of the 2019 ACM Southeast Conference*, ACM SE '19, (New York, NY, USA), p. 233–236, Association for Computing Machinery, 2019.

[8] "MongoDB." `https://www.mongodb.com/`. Accessed: 2020-03-28.

[9] K. C. nd Michael Dirolf, *MongoDB: The Definitive Guide*. " O'Reilly Media, Inc.", 2010.

[10] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, p. 377–387, June 1970.

[11] V. Dhar, "Data science and prediction," *Commun. ACM*, vol. 56, p. 64–73, Dec. 2013.

[12] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[13] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*. USA: Benjamin-Cummings Publishing Co., Inc., 1989.

[14] S. Abiteboul, "Querying semi-structured data," in *Proceedings of the 6th International Conference on Database Theory*, ICDT '97, (Berlin, Heidelberg), p. 1–18, Springer-Verlag, 1997.

[15] P. Buneman, "Semistructured data," in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, (New York, NY, USA), p. 117–121, Association for Computing Machinery, 1997.

[16] S. Abiteboul, P. Buneman, and D. Siciu, *Data on the Web: From Relations to Semistructured Data and XML*. USA: Morgan Kaufmann Publishers, 2000.

[17] C. C. Shilakes and J. Tylman, "Enterprise information portals," Nov 1998. https://web.archive.org/web/20110724175845/http://ikt.hia.no/perep/eip_ind.pdf.

[18] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0*, 2 1998. Available at https://www.w3.org/TR/1998/REC-xml-19980210.html.

[19] D. C. Denison, "The road to xml: Adapting sgml to the web," *World Wide Web Journal*, vol. 2, pp. 5–12, 1997.

[20] J. Bollen, *A cognitive model of adaptive web design and navigation*. PhD thesis, Vrije Universiteit Brussel, 10 2001. Available at https://www.cs.odu.edu/~jbollen/diss_pdfs/chapter2.pdf.

[21] V. Bush and J. Wang, "As we may think," *Atlantic Monthly*, vol. 176, pp. 101–108, 1945.

[22] Hypertext., *Merriam Webster Dictionary*. Merriam Webster Inc. Available at https://www.merriam-webster.com/dictionary/hypertext. Accessed: 2020-05-12.

[23] D. Raggett, J. Lam, I. Alexander, and M. Kmiec, *Raggett on HTML 4 (2nd Ed.)*. USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[24] ISO Central Secretary, "Information processing — Text and office systems — Standard Generalized Markup Language (SGML)," Standard ISO 8879:1986(en), International Organization for Standardization, Geneva, CH, 1986.

[25] C. F. Goldfarb, *The SGML Handbook*. USA: Oxford University Press, Inc., 1991.

[26] HTML Working Group, "HyperText Markup Language Specification - 2.0," Standard RFC 1866, Internet Engineering Task Force (IETF), Fremont, USA, 1995.

[27] E. R. Harold, W. S. Means, and L. Petrycki, *XML in a Nutshell: A Desktop Quick Reference*. USA: O'Reilly Associates, Inc., 3 ed., 2004.

[28] S. J. DeRose, *The SGML FAQ Book: Understanding the Foundation of HTML and XML*. USA: Kluwer Academic Publishers, 1997.

[29] World Wide Web Consortium (W3C), *XML Path Language (XPath) 1.0*, 11 1999. Available at `https://www.w3.org/TR/1999/REC-xpath-19991116/`.

[30] World Wide Web Consortium (W3C), *Namespaces in XML 1.0 (Third Edition)*, 12 2009. Available at `https://www.w3.org/TR/REC-xml-names/`.

[31] World Wide Web Consortium (W3C), *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, 4 2012. Available at `https://www.w3.org/TR/xmlschema11-1/`.

[32] U. Ogbuji, "Principles of xml design: When to use elements versus attributes," Mar 2004.

[33] Douglas Cockford, "The JSON Saga." 2011. [Online] Available: https://www.youtube.com/watch?v=-C-JoyNuQJs, last accessed on 10/23/2019.

[34] D. Crockford, *How Javascript Works*. USA: Virgule-Solidus LLC, 2018.

[35] C. Severance, "Discovering javascript object notation," *Computer*, vol. 45, pp. 6–8, apr 2012.

[36] "ECMAScript 2019 Language Specification," Standard ECMA-262, ECMA International, Geneva, CH, 1995.

[37] Douglas Crockford, "Introducing JSON." [Online] Available: https://www.json.org/, last accessed on 04/23/2020.

[38] ECMA International, *The JSON Data Interchange Syntax*, 12 2017. Available at `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[39] J. J. Garrett, "Ajax: A new approach to web applications," Feb 2005.

[40] N. C. Zakas, J. McPeak, and J. Fawcett, *Professional Ajax, 2nd Edition*. GBR: Wrox Press Ltd., 2007.

[41] L. Bassett, "Introduction to javascript object notation: A to-the-point guide to json," (USA), O'Reilly Associates, Inc., 2015.

[42] J. S. Org, "Json schema," 2019. Accessed on 12-11-2019.

[43] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, (Republic and Canton of Geneva, CHE), p. 263–273, International World Wide Web Conferences Steering Committee, 2016.

[44] B. Kommadi, *Learn Data Structures and Algorithms with Golang*. Birmingham, UK: Packt Publishing, March 2019.

[45] G. J. Bronson, *A First Book of C++*. Boston, MA, USA: Course Technology Press, 4th ed., 2011.

[46] K. B. Bruce, *Foundations of Object-Oriented Languages: Types and Semantics*. Cambridge, MA, USA: MIT Press, 2002.

[47] A. Nierman and H. V. Jagadish, "Evaluating structural similarity in xml documents," vol. 2, pp. 61–66, 2002.

[48] S. Wierzchon and M. Klopotek, *Modern Algorithms of Cluster Analysis*. Cham, CH: Springer International Publishing, 2018.

[49] J. L. C. Izquierdo and J. Cabot, "Discovering implicit schemas in json data," in *Proceedings of the 13th International Conference on Web Engineering*, ICWE'13, (Berlin, Heidelberg), pp. 68–83, Springer-Verlag, 2013.

[50] J. Canovas Izquierdo and J. Cabot, "JSONDiscoverer: Visualizing the schema lurking behind JSON documents," *Knowledge-Based Systems*, vol. 103, pp. 52–55, 04 2016.

[51] M. Klettke, U. Störl, and S. Scherzinger, "Schema extraction and structural outlier detection for json-based nosql data stores.," 03 2015.

[52] W. Spoth, T. Xie, O. Kennedy, Y. Yang, B. Hammerschmidt, Z. H. Liu, and D. Gawlick, "Schemadrill: Interactive semi-structured schema design," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA'18, (New York, NY, USA), pp. 11:1–11:7, ACM, 2018.

[53] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Counting types for massive json datasets," in *Proceedings of The 16th International Symposium on Database Programming Languages*, DBPL '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[54] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Parametric schema inference for massive json datasets," *The VLDB Journal*, vol. 28, pp. 497–521, Aug 2019.

[55] M.-A. Baazizi, H. Ben Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani, "Schema Inference for Massive JSON Datasets," in *Extending Database Technology (EDBT)*, (Venise, Italy), Mar. 2017.

[56] M. DiScala and D. J. Abadi, "Automatic generation of normalized relational schemas from nested key-value data," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), pp. 295–310, ACM, 2016.

[57] M. Piernik, D. Brzezinski, T. Morzy, and A. Lesniewska, "Xml clustering: a review of structural approaches," *The Knowledge Engineering Review*, vol. 30, no. 3, p. 297–323, 2015.

[58] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. Sellis, "A methodology for clustering xml documents by structure," *Inf. Syst.*, vol. 31, p. 187–228, May 2006.

[59] A. Tagarelli and S. Greco, "Semantic clustering of xml documents," *ACM Trans. Inf. Syst.*, vol. 28, Jan. 2010.

[60] J. Tekli and R. Chbeir, "A novel xml document structure comparison framework based-on sub-tree commonalities and label semantics," *Web Semant.*, vol. 11, p. 14–40, Mar. 2012.

[61] R. Nayak and S. Xu, "Xcls: A fast and effective clustering algorithm for heterogenous xml documents," in *Proceedings of the 10th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD'06, (Berlin, Heidelberg), p. 292–302, Springer-Verlag, 2006.

[62] P. Antonellis, C. Makris, and N. Tsirakis, "Xedge: Clustering homogeneous and heterogeneous xml documents using edge summaries," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, (New York, NY, USA), p. 1081–1088, Association for Computing Machinery, 2008.

[63] A. Algergawy, E. Schallehn, and G. Saake, "A schema matching-based approach to xml schema clustering," in *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '08, (New York, NY, USA), pp. 131–136, ACM, 2008.

[64] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang, "Xclust: Clustering xml schemas for effective integration," in *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, (New York, NY, USA), p. 292–299, Association for Computing Machinery, 2002.

[65] M. Weis and F. Naumann, "Detecting duplicate objects in xml documents," in *Proceedings of the 2004 International Workshop on Information Quality in Information Systems*, IQIS '04, (New York, NY, USA), pp. 10–19, ACM, 2004.

[66] S. Puhlmann, M. Weis, and F. Naumann, "Xml duplicate detection using sorted neighborhoods," in *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, (Berlin, Heidelberg), p. 773–791, Springer-Verlag, 2006.

[67] X. L. Wang and X. Wang, "An efficient approximate emst algorithm for color image segmentation," in *Machine Learning and Data Mining in Pattern Recognition* (P. Perner, ed.), (Cham), pp. 147–159, Springer International Publishing, 2018.

[68] S. Lu, "A tree-to-tree distance and its application to cluster analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, pp. 219–224, April 1979.

[69] k. Zhang and D. Shasha, “Simple fast algorithms for the editing distance between trees and related problems,” *SIAM journal on computing*, vol. 18, pp. 1245–1262, Feb 1989.

[70] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” *SIGMOD Rec.*, vol. 25, pp. 493–504, June 1996.

[71] T. Akutsu, T. Mori, T. Tamura, D. Fukagawa, A. Takasu, and E. Tomita, “An improved clique-based method for computing edit distance between unordered trees and its application to comparison of glycan structures,” in *Proceedings of the 2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, CISIS ’11, (USA), p. 536–540, IEEE Computer Society, 2011.

[72] T. Mori, T. Tamura, D. Fukagawa, A. Takasu, E. Tomita, and T. Akutsu, “A clique-based method using dynamic programming for computing edit distance between unordered trees,” *Journal of computational biology : a journal of computational molecular cell biology*, vol. 19, pp. 1089–104, 10 2012.

[73] Brian Fritz, “The Open Movie Database API,” 2020. Accessed on 05-23-2020.

[74] Spotify Inc., “Spotify Search API,” 2020. Accessed on 05-23-2020.

[75] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.