Towards a Data Generation Tool for NoSQL Data Stores

Author: Hasan Mahmud

Reviewers: Prof. Dr. Jens. Lehmann
           Prof. Dr. Stefan Wrobel

Supervisor: Harsh Thakkar, M.S.

A Thesis submitted to the Faculty of Computer Science in partial fulfillment of the requirements for the degree of Masters of Science

Media Informatics
RWTH Aachen University

November 4, 2018
In both industry and academia areas, datasets with different characteristics are needed for experimental purposes. However, real data may be difficult to obtain due to its scarcity and privacy policies. Moreover, there are many situations where researchers cannot make use of the real data due to its availability in incompatible formats and/or insufficient volume (there are many cases that the developers may require large volumes of data, e.g. benchmarking, machine learning, etc). The work presented in this thesis has been undertaken in the context of generating test data onto different data stores format at different volume to create a replacement of real data (or at least generate as real data as possible with respect to the real world application needs). Through this work, we present a concept of data generation approaches where the data is generated by extracting the data schema from the user’s given sample data sets. Then the system uses that extracted schema to generate new data that closely follow the real data key and value pattern for different data stores. By this way, the end user can get the flexibility to create data onto a different data format just from one sample JSON formatted dataset. In our long-term vision, we see our concept of data generation approach as a part in the research community in various domains (such as benchmarking) to get more flexibility on the data manipulation and to test a wider set of scenarios in their application.
Acknowledgements

First of all, I am deeply grateful to my supervisor Harsh Thakkar for his valuable guidance and constant support during this work. Completing this work without his help would have been impossible. His worthy criticism and appreciation of work helped me in transforming my fluid concepts in my thoughts into concrete tasks.

I would like to thanking my examiners Prof. Dr. Jens Lehmann and Prof. Dr. Stefan Wrobel for helping me to successfully graduate from RWTH Aachen University, Germany.

It’s my pleasure to take this opportunity to thank my fellows Mohiuddin Ibne Kawsar, Shakur Shams and Sawgata Biswas for helping me with the crucial task of narrowing down my thesis problem.

My special thanks to my parents and sisters who significantly contributed to my upbringing and brought me to a stage where I can make my humble contributions to human knowledge. I want to dedicate this thesis to my late father Kamal Uddin Ahmed who passed away a couple of months ago. He is my idol and till his last day he keeps faith in me and inspired me. At the same time I can never forget the contributions of my wife for always providing me with the moral support and backing of my decisions.
# Contents

Abstract ii

Acknowledgements iii

1 Introduction 1
  1.1 Motivation Scenario ........................................... 2
  1.2 Motivation Questions ........................................... 3
  1.3 Aims and Objectives ........................................... 3
  1.4 Paper Outline ................................................... 4

2 Literature Review 5
  2.1 Background Knowledge ......................................... 5
    2.1.1 Data Stores ................................................. 5
      Data Stores in Scope of this Paper ......................... 7
    2.1.2 JSON Documents ........................................... 7
      JSON Document in Scope of this Paper ..................... 7
    2.1.3 Terminology ................................................ 8
      Jackson JSON .................................................. 8
      Alphanumeric Pattern ........................................ 9
      URL Pattern ................................................... 10
      Email Pattern ................................................ 10
      Thread Local Random ......................................... 10
      Graphson ....................................................... 11
      XML .......................................................... 12
      Cypher ......................................................... 12
      SQL .......................................................... 13
  2.2 Previous Works in Data Generation .......................... 13
    2.2.1 Previous Approach of Synthetic Data Generation ........ 14
      White’s Data Generator ....................................... 14
      IDSG .......................................................... 14
      Berlin SPARQL Benchmark ..................................... 14
    2.2.2 Data Generation of Different Benchmark Application .... 15
      TPC .......................................................... 15
3 Approach

3.1 Conceptual Model ........................................... 19
3.2 System Architecture ...................................... 21
  3.2.1 User Interface ......................................... 21
  3.2.2 Experiment Manager ................................. 21
  3.2.3 System Sequence Diagram ......................... 21
3.3 Schema Extraction ......................................... 23
  3.3.1 JSON Extraction Process ........................... 23
  3.3.2 Approach of Schema Extraction ................. 24
    Definition of the Key-Value Map .................... 24
  3.3.3 Schema Extraction Algorithm .................... 24
  3.3.4 Example of Schema Extraction Algorithm ...... 27
3.4 Data Generation ............................................ 30
  3.4.1 Data Generation Process ........................... 30
  3.4.2 Identification of Target Data Volume .......... 31
  3.4.3 Identification of Key Frequency per record ... 31
  3.4.4 Identification of Key-Value Relationship ... 35
  3.4.5 Identification of Value Generation Method ... 37
    Mimic Unique ............................................. 37
    Mimic Duplicate ......................................... 37
    Random ................................................... 37
  3.4.6 Generation of Numerical Data ................... 38
    Mimic Unique Number .................................. 38
    Mimic Duplicate Number .............................. 39
    Random Number ......................................... 40
  3.4.7 Generation of Text Data ........................... 41
    Text Generator .......................................... 41
    Example of Text Data Generation .................. 43
  3.4.8 Generate Data from Identified and Extracted Technique .... 44

4 Experiment and Results .................................... 46

4.1 Experimental Setup .................................... 46
  4.1.1 IRIS Dataset ......................................... 46
  4.1.2 MSME Country Indicators Dataset .............. 47
  4.1.3 Task Dataset ......................................... 47
# List of Figures

2.1 Current state of data stores ........................................ 6  
3.1 Conceptual model .................................................. 20  
3.2 System architecture ............................................... 21  
3.3 System sequence diagram 1 ....................................... 22  
3.4 Sequence diagram of JSON extraction process .................. 23  
3.5 Example of schema extraction method first phase key ......... 28  
3.6 Example of schema extraction method with another phase key 29  
3.7 Sequence diagram of data generation process ................ 30  
4.1 Extracted data model of IRIS dataset .......................... 49  
4.2 Extracted data model of MSME Country Indicators dataset .. 51  
4.3 Extracted data model of Task dataset .......................... 52  
4.4 JSON data generation of IRIS dataset .......................... 54  
4.5 JSON data generation of MSME Country Indicators dataset .. 55  
4.6 JSON data generation of Task dataset .......................... 55  
4.7 GraphSON data generation of IRIS dataset .................... 57  
4.8 GraphSON data generation of MSME Country Indicators dataset 57  
4.9 GraphSON data generation of Task dataset .................... 58  
4.10 SQL data generation of IRIS dataset ............................ 59  
4.11 SQL data generation of MSME Country Indicators dataset .. 60  
4.12 SQL data generation of Task dataset ............................ 60  
4.13 Cypher data generation of IRIS dataset ....................... 61  
4.14 Cypher data generation of MSME Country Indicators dataset 62  
4.15 Cypher data generation of Task dataset ....................... 62
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Jackson databind JSONnode method summary</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Data generator comparison</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Sample IRIS Dataset</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>System Configuration</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Analysis of identifying key frequency in IRIS datasets</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Properties of Generated JSON data</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Properties of Generated GraphSON data</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>Properties of Generated SQL data</td>
<td>61</td>
</tr>
<tr>
<td>4.7</td>
<td>Properties of Generated Cypher data</td>
<td>63</td>
</tr>
</tbody>
</table>
# List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Construction of the Key-Value Map</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Walk throw the JSON document</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>Check node of the KV</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Add node to the KV</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Add value to the KV</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Construct KV</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>Identification of Key Frequency:1</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>Identification of Key Frequency:2</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>Identification of Key Frequency:3</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>Identification of key-value relationship:1</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>Identification of key-value relationship:2</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>Mimic unique number</td>
<td>39</td>
</tr>
<tr>
<td>13</td>
<td>Mimic duplicate number</td>
<td>39</td>
</tr>
<tr>
<td>14</td>
<td>Random number</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>Text generator</td>
<td>42</td>
</tr>
<tr>
<td>16</td>
<td>Sentence generator</td>
<td>43</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years the volume of data consumed by many organizations has exploded, due to the explosion of the web. According to [BCM11], in 15 out of the 17 US economic sectors, companies which has more than 1,000 employees store on average over 235 terabytes of data which is more than US Library of Congress data volume. Due to the need to scale and/or processing this data volume has increased and the database world came up with new data management technique with novel data management model which is called NoSQL (which stands for “Not Only SQL”). There exists a vast array of NoSQL data store solutions and in particular more than 225 ¹. Both academia and industry alike are using these data stores at a rapid pace.

In industry and academia in both areas test data is essential for performance testing, security testing, and functional testing and benchmarking [Tha+17b; Kes+17]. [AR+13] mentioned that in the research community synthetic data approaches is widely used in a variety of domains which are named as privacy protection, healthcare, pattern recognition, data mining, etc. According to [Nar17] it is unfortunate that, there is a scarcity of the availability of the real data due to many organizations are not willing to share their data with third parties for privacy concerns, the cost of data transfer, as well as the unavailability of such a huge amount of data. Therefore the development and testing for analytics software are somewhat very much dependent on the synthetic data which ensures reasonable results of testing and development. [ATM12] also mentioned that for the reliable results, it is important to capture the characteristics of the real data. [Nar17] also claims that data generators need to be expressive and scalable in order to generate large quantities of realistic data onto a short period of time. [AR+13] states that researchers are having more flexibility on manipulating data to test a wider range of conditions and scenarios in their applications by generating synthetic data.

In the past, there have been several attempts made to generate test data onto different scenarios of different goals in mind and targeted to the specific application domain, and some of which are discussed in chapter 2. Generating large data and also generating data onto different data stores are equally important. In this paper, we present a practical

¹ http://nosql-database.org/, last accessed on August 15, 2018
Chapter 1. Introduction

approach to a data generator where the data is generated by extracting the data schema from the users given sample data sets. The data generator is able to extract schema from the user’s given sample data sets and generate data that focuses on volume, variety and velocity. By configuring and deploying different number of parallel data generator can produce different volume of data sets. As the data is generated based on the user provided sample data sets by extracting the schema and creating the data model, so different types of data such as structured, semi-structured data can be generated by keeping the important characteristics of raw data.

1.1 Motivation Scenario

To motivate the importance of our proposed data generator, we present the following scenario.

Asif is a computer science doctoral student. He and his supervisor are planning to develop a new efficient mechanism for analyzing big data. His supervisor asked him to start his work by generating a large amount of data related to the e-commerce sector as they don’t have a vast amount of data onto their lab to analyze. Asif has started his work by surveying literature of different data generator and overwhelmed with a large number of data generators to solve the initial problem. After an extensive search process, Asif was able to test some data generator and generate some data. But he found that these generated data are synthetic data and unrealistic too due to the typical approach of a synthetic data generator which generate random data. He also realized that the generated data is not appropriate for his target domain as he wanted e-commerce related data.

After doing another extensive search he observed some more data generator which aims to generate data onto e-commerce sector but some of them are not generating all types of data such as structured, semi-structured and unstructured data rather they are generating only structured data. On the other hand, some data generators generated all types of data but they have a fixed data model which is not suitable for his use-case. After a couple of months of work and extensive searching, Asif was able to manage one e-commerce vendor to get their data whose data model is quite similar of his target use-case. But in this case, he identified that transferring this large amount of data is quite expensive. It is apparent that this task is quite time and effort consuming and also quite expensive.

Almost all developers have written database applications in where test data is a crucial fact in order to be able to check the validity of their applications. [Sal07] claims that a database that is defined, but does not yet contain data, is not really that useful to make sure the design the application will work as intended or not. [Sal07] also states
that in some cases, the developers may have access to test data or an event snapshot of production data. However, many times it is the case that the developers may have a limited set of test data, or perhaps no access to data at all.

1.2 Motivation Questions

We formulate some questions to address the problem and to overcome the problem based on the above mentioned scenario.

- Is a given dataset reasonable?
- Does the predefined datasets reflect my use case?
- How to generate big data from my sample data?
- How to decide which data stores is well suited for my application own use case?

To formulate the answers to the above questions motivate us to define a new approach to generate data where the data is generated by extracting the data schema from the user given sample data sets. The system designed through this work captures the distributions and patterns of attributes, from the given sample JSON data and use these information to generate much larger data for different data stores.

1.3 Aims and Objectives

The aims and objectives of this thesis are:

- Design and development of the concept of a data generator which is able to generate a large volume of data by extracting the data schema from user given sample data sets.

- Generated data should capture the dependencies and distributions of real data, including patterns.

- Design and development of a web based user interface for easy user experience.

- Promote easy access for replication of synthetic to aid third party benchmarking platforms such as LITMUS [Tha+17b; Kes+17], LDBC [Erl+15], etc.
1.4 Paper Outline

The rest of this paper is organized as follows.

- Chapter 2 reviews some of the preliminary concepts and some major related works in the area of data generation techniques.

- Chapter 3 presents the approach and system architecture of our data generation application. The heart of the chapter is the schema extraction approach and algorithm. In addition to this, this chapter also describes the methodology that we follow for extracting and storing the patterns of real data, and generating data from these patterns.

- The goal of chapter 4 is to summarizes the experiments and analysis of the results.

- Chapter 5 includes the conclusion and possible future extensions.
Chapter 2

Literature Review

In this chapter, we provide some background knowledge in the area of data generator and review some related work on data generation techniques. We start to discuss some common terms and definition that is going to be used throughout the paper. Then we introduce one example JSON document, as our data generator aims to extract schema from user given sample datasets and generate data. Throughout this paper, we will use this example JSON document to discuss different approaches. After that, we discuss some of the previous systems that are developed for generating data.

2.1 Background Knowledge

This section aims to discuss the some common terms and definition that is going to be used throughout the paper.

2.1.1 Data Stores

As our proposed benchmark application aims to benchmark NoSQL data stores so in this section we give a brief description of NoSQL data stores. At present, there are more than 225 NoSQL data stores exist which are usually divided into four categories, according to their data model and storage and namely as Key-Value Stores, Document Stores, Column Stores and Graph databases [HJ11].

Key-value Stores persists in pairs of a unique key and an associated value. In this data model, data stores in key-value pair where value can be from a simple string to more complex lists and sets. Because of this data model, there is no concept of schema beyond distinguishing keys and values [HJ11].

Document Stores are advanced key-value stores where it encapsulates key-value pair in JSON or JSON like documents. [Kle+15] refers document stores as loosely structured sets of name-value pairs in JSON (JavaScript Object Notation) format or the binary representation BSON. In this data store key are unique to the document

\footnote{http://nosql-database.org/last accessed on July 15, 2018}
and also within a collection of documents every document has a special key ID [HJ11]. In contrast to key-value stores, values may not be only scalar values or list rather it supports more complex data structure like nested document. It provides more flexibility to add or remove data onto a particular document without affecting other documents.

![Figure 2.1: Current state of data stores. Adapted from [MH13]](image)

**Column Family Stores** are also known as column-oriented stores, extensible record stores and wide columnar stores [HJ11]. It is also an advanced key-value store, where key-value pairs can be grouped together in columns within tables. Column family stores representing data as a table like relational databases with the flexibility of key-value stores. This data model contains rows in where each row of a table consists of a unique row key. In this data model column can only contain atomic value and tables contain wide columns [BV16]. Thus, a column family is like a nested table within a column.

**Graph databases** represent and store data directly as graphs in where data is represented by nodes, edges, and properties. Nodes represent entities, Edges are lining that connect any two nodes which represent the relationship between two nodes and properties provide information to nodes. This database allows easy expression and fast processing of graph-like queries [BV16].

This variety makes it hard to the end users to decide which data stores are well-suited for their own use-case which motivates us to design the benchmark application to support the users to make the decision on which data stores are well-suited for their own use case.
2.1. Background Knowledge

Data Stores in Scope of this Paper

Data onto document stores is often represented in JSON format. The schema extraction and data generation methods we present in this paper mainly focus on JSON-based document stores such as mongodb. However, data onto key-value stores may also be stored in JSON or a similar format and thus it may also be applied to key-value stores. Besides that we also present in this paper data generation approach to graph databases such as neo4j and graphson as they are widely popular data stores. To give a comparison we also present data generation approaches to SQL data stores such as MySQL.

2.1.2 JSON Documents

Several NoSQL databases store JSON documents. In JSON the entities are called objects. Objects are unordered enumeration of properties, consisting of name/value pairs [Kle+15]. The available basic data types are boolean, number, and text. JSON properties can be multi-valued, these structures are then called array which is an ordered list of zero or more values, each of which may be of any type. With square bracket notation and comma-separated elements, arrays are represented. An Object is an unordered collection of name-value pairs, also known as key-value pairs in where the keys are strings. Every array is represented by an object. To separate each pair, objects are delimited with curly brackets. Within each pair, the key-value is separated by the colon ‘:’ character.

JSON Document in Scope of this Paper

In this paper, we do not introduce the complex JSON document. Instead, we provide some intuition using a small example. Throughout this paper, we use this example JSON document. In this example JSON document we present an object book is represented by the properties title, content, author, date, URL, reference and comments. The property comments are defined as an array, consisting of objects. Within the objects the properties content, email, reputation and date occur. This example JSON document is a good starting point because it uses the JSON data types string, number, array, and object. It even contains a hierarchical nesting of objects. It also contains field frequency relationship as in the example JSON document in an array we provide two JSON objects but they are not fully identical.

---

LISTING 2.1: Example JSON Document

```
[
  {
    "title": "Data Generation Technique",
    "content": "Database are ..",
    "author": "John Doe",
    "date": "2017-10-10 23:42",
    "comments": [{
      "email": "user@gmail.com"
    }, {
      "reputation": 4
    }],
    "url": "http://www.db.com",
    "reference": "AAHKJ23"
  },
  {
    "title": "NoSql Databases",
    "author": "Alex Muller",
    "comments": [{
      "content": "A great book..
    }, {
      "date": "2017-10-10 23:42",
      "reputation": 9
    }],
    "url": "http://www.nsdb.com",
    "reference": "http://www.BSDreference.com"
  }
]
```

2.1.3 Terminology

Jackson JSON

Jackson is a popular, high-performance JSON processor for Java which can support almost any JSON need that a user might have. [poi15] states that Jackson is a simple Java-based library to serialize Java objects to JSON and vice versa. [poi15] describes that Jackson processes three different ways to process json which are named as streaming API, tree model, and data binding. The data generator which we present in this paper heavily uses Jackson tree modeling and data binding process to extract schema from the JSON document.
2.1. Background Knowledge


**Data binding** [poi15] also presented that ObjectMapper reads/writes JSON for both types of data bindings which are

- **Simple Data Binding**: which converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans, and null objects [poi15]

- **Full Data Binding**: which converts JSON to and from any Java type. [poi15]

Throughout the paper our data generator use Jackson JSON node to extract schema from JSON document. JSON node is the base class for all JSON nodes, which form the basis of JSON Tree Model that Jackson implements. In here we briefly state some of the JSON node functions that we use in our data generator.

**Table 2.1:** Jackson databind JSONnode method summary, Adapted from [CC13]

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>asBoolean() Method that will try to convert value of this node to a Java boolean.</td>
</tr>
<tr>
<td>double</td>
<td>asDouble() Method that will try to convert value of this node to a Java double.</td>
</tr>
<tr>
<td>int</td>
<td>asInt() Method that will try to convert value of this node to a Java int.</td>
</tr>
<tr>
<td>long</td>
<td>asLong() Method that will try to convert value of this node to a Java long.</td>
</tr>
<tr>
<td>abstract string</td>
<td>asText() Method that will return a valid String representation of the container value, if the node is a value node (method isValueNode() returns true), otherwise empty String.</td>
</tr>
<tr>
<td>abstract JSONNode Type</td>
<td>getNodeType() Return the type of this node.</td>
</tr>
<tr>
<td>Iterator&lt;JSONNode&gt;</td>
<td>iterator() Same as calling elements(); implemented so that convenience &quot;for-each&quot; loop can be used for looping over elements of JSON Array constructs.</td>
</tr>
</tbody>
</table>

**Alphanumeric Pattern**

For pattern matching with regular expressions, Java provides the java.util.regex package. A regular expression is a special sequence of characters that help to match or find other strings or sets of strings, using a specialized syntax held in a pattern. In this paper, our data generator uses this class to match the alphanumerical pattern from a text data. Following is the regex expression that our data generator uses to match the pattern of alphanumerical.
Chapter 2. Literature Review

**Definition 1 (Alphanumeric pattern regex)** Pattern REGEX is defined by
\[ \text{Pattern} \text{.} \text{compile}([A-Z\, a-zA-Z0-9]*) \]

**Url Pattern**

Our data generator uses java pattern matcher class to match an URL pattern of a text data. Following is the regex expression that our data generator uses to match pattern of URL.

**Definition 2 (Url Pattern Regex)** Pattern REGEX is defined by
\[ \text{Pattern} \text{.} \text{compile}((?i)?(: https?|ftp) : /)/(? : \\S + (? : \\S*)?)|(? : (? : 1[6-9]\d\d|1\d\d|00|0)\d?|1[6-9]\d|0|9|8)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-z)\d|a-zA-Z0-9_@]+$)

**Email Pattern**

Our data generator uses java pattern matcher class to match an email pattern of a text data. Following is the regex expression that our data generator uses to match the pattern of email.

**Definition 3 (Email pattern regex)** Pattern REGEX is defined by
\[ \text{Pattern} \text{.} \text{compile}($[A-Z\, a-zA-Z0-9\, @]+$", Pattern.CASEINSENSITIVE$)

**Thread Local Random**

[Vid16] nicely defines the class ThreadLocalRandom (TLR), is an implementation of the interface offered by the Random class of java openJdk. [Vid16] defines Random
is an implementation of the Linear Congruence Method. [Vid16] also states that the authors of ThreadLocalRandom took two algorithms for producing PRNGs and refined their implementation until they reached to what SplittableRandom is (and subsequently ThreadLocalRandom).

**Graphson**

[Bur17] nicely defines graphson as a JSON-based format for individual graph elements (i.e. vertices and edges). The organization and utilization of this elements are written as a complete graph that can also be considered graphson format.

---

### Definition 4 (Graphson:Vertex rules and conventions) Adapted from [Bur17]

- Each vertex has the following structure:
  
  ```json
  {
  "id" : int,
  "label" : "",
  "inE" : {
  "edge1_label1" : [{edge}, {edge}, ...]},
  "outE" : {
  "edge1_label1" : [{edge}, {edge}, ...]},
  "properties" : {}
  }
  ```

- By a unique integer id “id” is mapped (it starts at 1, by convention, and increases sequentially)

- By a string which represents the label associated with that vertex, “label” is mapped.

- “inE” and “outE” map to dictionaries which map the label of an edge to a list of edges of that type that involve the vertex of which these are sub-dictionaries and also “inE” and “outE” encode the directionality of the edges.

- By a dictionary, “properties” is mapped in where the keys represent the labels of the properties and the values are dictionaries which contains the key “id” mapped to an integer ID for that property value and the key “value” mapped to a value for that property.

---

### Definition 5 (Graphson:Edge rules and conventions) Adapted from [Bur17]

- Edges have the following structure:
  
  ```json
  {
  "id" : int,
  "outV" : int,
  "inV" : int,
  "properties" : {
  "property_name" : value
  }
  }
  ```

- “id” maps to a unique integer ID. The rule of starting the edge IDs is immediately after the greatest vertex ID and increase sequentially, so if a graph has 8 vertices and 8 edges, the vertex IDs would be 1-8 and the edge IDs would be 9-16.
• On each side of the edge, “inV” and “outV” map to the vertex IDs. the ID of the vertex from which the edge starts is mapped by “outV”. On the other hand, “inV” maps to the ID of the vertex at the edge ends.

• “properties” maps to a dictionary which maps the label of a property to its value.

XML

[Qui] defines XML stands for Extensible Markup Language and is a simple text-based format for representing structured information such as documents, data, configuration, books, transactions, invoices, and much more.

Definition 6 (XML:Format)  Adapted from [Qui]

<part number="1976">
   <name>Windscreen Wiper</name>
   <description>The Windscreen wiper automatically removes rain from your windscreen, if it should happen to splash there. It has a rubber <ref part="1977">blade</ref> which can be ordered separately if you need to replace it. </description>
</part>

Cypher

[neo] defines Cypher nicely and states that cypher is a declarative, SQL-inspired language which is used to for describe patterns in graphs visually using an ASCII-art syntax. [neo] also states that it allows us to state what we want to select, insert, update or delete from our graph data without requiring us to describe exactly how to do it. Cypher is a pattern-oriented, declarative query language which is a mix of SQL and graph traversal patterns.

Definition 7 (Neo4j Cypher:Fundamental building blocks)  Adapted from [neo]

• Nodes are used to represent entities which can have properties represented by key/value pair
2.2. Previous Works in Data Generation

Data generation is an important tool of many areas, including software testing, machine learning, and privacy protection. [ALM11] describes that synthetically generated inputs can be used to test complex program features and to find system faults in the area of software testing. [ALM11] also states that another application area in where synthetic data can be used to balance training data is the field of machine learning. According to [ALM11] synthetic data is also commonly used in scenarios where collecting real

- **Relationships** are used to represent the relationships between nodes which must have a starting and end node which are encapsulated by relationship type.
- **Properties** Nodes and relationships can have properties (key/value pairs values can be primitives or collections of primitives.)
- **Labels** nodes can have zero or multiple labels

**SQL**

SQL stands for structured query language which usually talks to database server. [poi] states that SQL allow users to access data in the relational database management systems.

**Definition 8 (RDBMS concept) Adapted from [poi]**

- **Table** The data in an RDBMS is stored in database objects which are called as tables which is basically a collection of related data entries and it consists of zero or more columns and rows.
- **Field** A field is a column in a table. The responsibility of field is to maintain specific information about every record in the table.
- **Record** A record/row of data is an individual entry that exists in a table
- **Column** A column is a vertical entity in a table. Column contains all the information associated with a specific field in a table
data is not an option, due to budget, time or privacy concerns. Till date, there are many approaches have been proposed to generate synthetic data. [AR+16] claims that some of those works are general-purpose synthetic data generators and they do not target a specific application area. [AR+16] feels the importance to preserve certain characteristics in the generated datasets to make it realistic depending on the domain and thus, it is suitable for their intended use. In this section, we discuss some of the earlier approaches used for synthetic data generation and also describes some of the data generation tools of different benchmark application.

2.2.1 Previous Approach of Synthetic Data Generation

White’s Data Generator

[Hoa08] explores that in 2004 Jonathan White, a graduate student at the University of Arkansas, developed a special-purpose synthetic data generator which was limited to 50 or so field types associated with a mailing list application, such as name, address, city, state, zip code, area code, phone number, etc. However [Hoa08] claims this generator has the ability to produce very realistic data onto the field types that it supported, but it was not extensible.

IDSG

Later in 2005-2006, a team from the University of California at Riverside and Lucent Technologies developed a synthetic data generator which is called IDSG. IDSG stands for “IDAS Data Set Generator”, and IDAS stands for “Information Discovery and Analysis Systems”[Hoa08]. IDSG is built to produce test data sets for various IDAS and provides a web interface through which clients can select a data set (i.e., “Credit card Data”, “Names”). IDSG then generates data and made a comma-separated value (.csv) files for the client[Hoa08]. However [Hoa08] suggests that IDSG provides no easy way to describe and generate user-specified data sets.

Berlin SPARQL Benchmark

According to [BS09] Berlin SPARQL Benchmark (BSBM) is built around an e-commerce use case in which a set of products are offered by different vendors and consumers have posted reviews about products. The benchmark uses a predefined use case to generate data and defines an abstract data model for this use case together with data production rules[BS09]. According to [BS09] BSBM supports the creation of large datasets using a number of products as the scale factor and can output in an RDF representation as well as relational representation.
2.2. Previous Works in Data Generation

2.2.2 Data Generation of Different Benchmark Application

There has been quite some research on developing a benchmark application for big data benchmarking. There are a number of popular benchmarking tools that that aims to evaluate and compare the performance of different big data systems. All existing benchmark applications are developed with different goals in mind and targeted to the specific application domain. It starts in the 1980s according to [Gha+13, p.3] when Teradata Corporation and other more traditional DBMS vendors appeared as the first generation of commercial system and the need rises for a well defined benchmark that measures the performance of DBMS dealing with very large amounts of data. [Gha+13, p.3] also stated that one of the first major benchmark application developed by the Transaction Processing Performance Council to compare commercial systems.

TPC

TPC developed a series of benchmark application starting with TPC-D at the beginning of the 90s and TPC-H and TPC-R in the dawn of 2000. TPC-DS is TPC’s latest decision support benchmark. [NP06, p.4-5] presented the TPC-DS benchmark application as a decision support benchmark application which mainly models the decision support functions of a retail product supplier and data contains important business information such as order, customer, and product data. [NP06, p.4-5] also stated that the data generator of TPC-DS uses a hybrid approach to the domain and data scaling. TPC-DS handles some aspects of big data such as very large data and system sizes. From the current limit of 100 terabytes, the data generator and schema can be extended to petabytes. Its data size is defined in scale factors and also it partially considers data veracity as it generates a small portion of data sets using more realistic distributions derived from real data besides generating synthetic data [NP06, p.4-5]. Still, it has some lacking and the major lack is it does not consider semi-structured and unstructured data as it only uses a table as the data source.

YCSB

In parallel, the industrial player as well as emerging big data companies has started to develop their own benchmark suites. In 2010 [Coo+10, p.1-2] presented a micro benchmark for big data analytics namely as YCSB. It is Yahoo’s cloud serving benchmark suite to evaluate NoSQL data stores. It is an open source benchmark application to support benchmarking clients for many NoSQL data stores. The core of this suite is YCSB client that generates the data to be loaded into a data store and execute the operations of workload. [Coo+10, p.1-2] explain that this benchmark suite represents a key-value store like data model with one entity, which is called User and has ten attributes per
default. It inserts records of a specific data store with a specified distribution such as uniform, zipfran, latest, multinomial [Coo+10, p.4-5]. In the terms of data generation, this benchmark suite has some major lacking including limited data type and data source using. It only generates table data which is a structured data type as their workload inputs and also not considering real-world data sets in the data generation process.

**Bigbench**

In recent years Bigbench benchmark application which is mainly based on product retailer focuses on end to end benchmark. In 2013 [Gha+13, p.5] presented this benchmark application of a more robust data generator which is basically designed based on Parallel Data Generation Framework(PDGF) technology. [Gha+13, p.3] states that this data generator generates all three kinds of data type namely structured data which is adapted from TPC-DS, semi-structured data (web-logs), unstructured data (review text); and thus this data generator covers volume, variety, velocity aspect of big data. For the structured data generation, Bigbench benchmark application adopts the technology from PDGF and its scale factor is similar to TPC-DS [Gha+13, p.5-7]. On the other hand for semi-structured data generation, this benchmark suite extends PDGF to generate web-log which is similar to Apache web server log and coupled with structured data [Gha+13, p.5-7]. This benchmark suite also generates unstructured data which is a review text and coupled with structured data by extending PDGF technology [Gha+13, p.5-7]. But in both weblog generator and review generator, the data are generated on the basis of table data only and rely on the structured table data. In addition, the data model of this benchmark suite is very much fixed and mainly designed for product retailer, thus it is not well suited for other kinds of application domains.

**BigDataBench**

In latest, ICT Chinese academy of sciences proposed a more complete benchmark application of different data models. [Wan+14, p.4] showed a more robust data generator of big data benchmarks namely BigDataBench by paying equal attention to structured, semi-structured, and unstructured data with a data model that enable to capture and preserve the important characteristics of real data of different types (e.g. text, graph, and table). [Wan+14, p.4] claims that by analyzing different application domains BigDataBench is generating data onto five application domains with 14 real-world data sets and three kinds of big data generator. In addition, [Min+13, p.6] states that this benchmark application chooses data sets from the application domain sets and generate synthetic data from a representative real data sets by adjusting different volume and deploying different numbers of parallel data generator. [Min+13, p.7] also states the
chosen real-world data are Wikipedia Entries, Google Web Graph, and Facebook Social Graph which is unstructured; E-commerce transaction data are structured; and Amazon Movie Reviews and Personal Resumes are semi-structured data and these real data sets are used in three different data generator namely text generator to generate text data, graph generator to generate graph data and table generator to generate table data. This benchmark application tries to generate data by defining different application domains but could not cover all application domains. With the rapid growth of data new innovative application domains are opening everyday. So by redefining the application domain and generating data based on that domain still keep their scopes limited.

Table 2.2: Data generator comparison of different benchmark application, Adapted from [HLX14, p.4]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scalability</th>
<th>Generation Rate</th>
<th>Source</th>
<th>Type</th>
<th>Real Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Status</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HiBench</td>
<td>Fixed</td>
<td>3 UN-controllable</td>
<td>Text</td>
<td>US, S</td>
<td>No</td>
</tr>
<tr>
<td>LinkBench</td>
<td>Fixed</td>
<td>1 Semi-controllable</td>
<td>Graph</td>
<td>S</td>
<td>Partially</td>
</tr>
<tr>
<td>TPC-DS</td>
<td>Scalable</td>
<td>N/A Semi-controllable</td>
<td>Table</td>
<td>S</td>
<td>Partially</td>
</tr>
<tr>
<td>YCSB</td>
<td>Scalable</td>
<td>N/A UN-controllable</td>
<td>Table</td>
<td>S</td>
<td>No</td>
</tr>
<tr>
<td>BigBench</td>
<td>Scalable</td>
<td>3 Semi-controllable</td>
<td>Text, Web-log, Table</td>
<td>S, SS, US</td>
<td>Partially</td>
</tr>
<tr>
<td>BigDataBench</td>
<td>Scalable</td>
<td>8 Semi-controllable</td>
<td>Text, Resume, Graph, Table</td>
<td>S, SS, US</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.2 on page 17 illustrating data generation techniques of different benchmark efforts [HLX14, p.4]. To date, most of the existing benchmark applications generate data as workload inputs in scale factors. But some benchmark applications such as HiBench and LinkBench generate fixed-size data. BigBench, BigDataBench, LinkBench, TPC-DS generate data by considering the control of data generation rate. We can observe that many existing benchmark applications (LinkBench, TPC-DS, YCSB) considering only structured data type and generate table data only, which make them ill-suited to many application domains. Bigbench and BigDataBench consider all kinds of data type and support a variety of data sources but their application domains are much narrow as
Bigbench is designed only for product retailer. Meanwhile, most of these benchmarks like YCSB generates random datasets as workload inputs, in where no real-world data are considered to generate data. On the other hand TPC-DS, Bigbench partially considers real data to generate data. In Bigbench weblogs and reviews are generated on the basis of table data and in TPC-DS only a small portion of data is generated by using more realistic distribution derived from real data. BigDataBench has different data model to consider real data. But still it keeps their scopes limited by predefining certain application domain and generating data according to that domain.
Chapter 3

Approach

In this chapter, we present the approach to our proposed data generation application by showing the sequence diagram and system architecture. Gradually we discuss the technique of data generator which consists of schema extractor and value generator as these are the two major component of our data generator application and the aim at our research is to develop these two components by motivating from some scenarios discussed in chapter 1.

3.1 Conceptual Model

The main goal of our proposed data generation application are to tackle the mentioned scenarios and questions discussed in chapter 1. The application consists of value generator by keeping the real value as much as possible where the value is generated by extracting the data schema from the user given sample data sets. The main concept of our data generator are to create large datasets by extracting the schema and value of user-provided small datasets. While generating new data this data generator aims to generate new data by keeping the original datasets frequency. This generator gives the choice to the user to generate new data by choosing the generation technique among 3 types of technologies which are unique datasets generation, mimic datasets generation and random datasets generation.

Figure 3.1 on page 20 gives an overview of the conceptual model of our Data Generation application. In this model, we present the flow of our application. User uses our application by providing his own sample JSON dataset which runs over on schema extractor, the major function of the data generator. This schema extractor extracts the schema and shows the schema to the user. The user can also choose data generation technique from our 3 types of data generation technique which includes: Mimic data and random data for every field of the generated schema. From user-chosen data generation technique and extracted schema, the application generates a dataset based on user-defined data volume. The application is intelligent enough to generate data by not losing any meaningful data onto user given sample datasets.
Chapter 3. Approach

The data generation application consists of JSON extractor and the data generator. JSON extractor is one of the most important parts of our application which is responsible to extract user provided sample JSON data and get the schema and meaningful value of user data. The core of our application is the data generator whose role is to generate data according to user choice by preserving the user provided sample datasets schema and value.

Figure 3.1: Conceptual model of system
3.2 System Architecture

In this section, we present the architecture for our data generation application, illustrated in Figure 3.2 on page 21, that consists of three main parts:

![System Architecture Diagram]

**Figure 3.2: System architecture of data generation application**

### 3.2.1 User Interface

The User Interface provides the end user with a user-friendly interface where the end user can specify her generation requirements by making a choice of data store selector, data uploader, method selector, volume selector. Data uploader let the user upload her own sample JSON dataset and datastore selector let the user to choose for which data store she wanted to generate data. The user can generation technique by selecting among 3 choices of every data field from the method selector. The user interface gives the end user the functionality to configure the data generation volume.

### 3.2.2 Experiment Manager

The Experiment manager has two components: data generation and Schema Extraction. Schema extractor is able to extract schema from JSON based datasets. On the other hand, component data generation is designed to produce data from user given sample datasets and it consists of value generator which is generating value of every data field by keeping in mind of user provided sample data values.

### 3.2.3 System Sequence Diagram

The data generation application consists of JSON extractor and the data generator. JSON extractor is one of the most important parts of our application which is responsible to extract user provided sample JSON data and get the schema and meaningful value of
Chapter 3. Approach

Figure 3.3: Sequence diagram of system
user data. The core of our application is the data generator whose role is to generate data according to user choice by preserving the user provided sample datasets schema and value.

3.3 Schema Extraction

One of the major function of our data generation application is its JSON extractor which is capable of extract schema from user provided sample JSON datasets and then generate large data sets based on the extracted data schema.

3.3.1 JSON Extraction Process

Figure 3.4 on page 23 shows the JSON extraction process sequence diagram that references the sequence diagram "dataset". The sequence starts at the top left, with the user sending a message to the controller object. At that point, the JSON extraction sequence diagram is called, with provided sample JSON data file as a parameter. The process of JSON schema extraction is discussed later in this chapter. Schema Extractor object sends the message to service. Service sends the sample JSON data onto parser object who is responsible to read user provided JSON data. Walker walks through the whole user sample JSON data and create a map of the node in where the node represents every unique field for sample JSON data. On the other hand value checker check the data.
type for every node and store the type and original value. The resulted schema map then passed to the user.

3.3.2 Approach of Schema Extraction

We present a schema extraction approach mainly focused on JSON based document data stores. Below we discussed the schema extraction approach and algorithm.

We present our approach to schema extraction by a key-value map in where the key represents the parent id and value to represent every child node. Nodes represent JSON properties, parent properties, and depth. As we store the key-value pair in ordered so the hierarchical structure of the JSON documents is preserved.

Definition of the Key-Value Map

Our schema extraction approach can be defined by key-value map structure, with key and nodes labels as the value that capture the complete schema structure by the following definition.

**Definition 9 (Key-value map)** [H] A key-value map $KV = (K, N)$ is a map where

- $V$ is a finite set of keys. Each key $v_i \in V$ is represented by a number ($\text{nodeID}_i$) where:
  - $\text{nodeID}_j : i$ is the ID of the JSON Object of $J_j$ within a Collection $C$ and $i$ is a unique node number within the JSON document.

- $N$ is a finite set of values. Each value $n_i \in N$ is represented by a node where:
  - node represent JSON properties, parent properties and depth.

3.3.3 Schema Extraction Algorithm

In this section, we briefly discuss the schema extraction algorithm of our proposed data generation application.

**Construction of the Key-Value Map**. Algorithm 1 on page 25 presents the building algorithm of a key-value map from input data of the JSON by a method KVM.walker in where we initiate to walk through the whole JSON document recursively. We initiate the parent node as a predefined node name as "root" and set its id 1. We also mark that this the initial depth of a JSON tree as we initialize the depth as 0.
3.3. Schema Extraction

**Algorithm 1** Construction of the Key-Value Map

1: **procedure** KV
2:  \[\text{input data: JSON document collection C:}\]
3:  initialize \(N_j\): \(N_j = \text{JSONdocumentToNode}\);
4:  initialize \(N\): \(N = \emptyset\);
5:  initialize \(M\): \(K = \emptyset; N = \emptyset\);
6:  \(i = 1\); // initialize counter
7:  // initializing the root node of the JSON document
8:  initialize \(P\): \(P = \text{root}\); // root name
9:  initialize \(P_i\): \(P_i = 1\); // root id
10: initialize \(D\): \(D = 0\); // depth
11: KV.walker\((N_j, D, P, P_i, N, M, i)\);

**Walk to throw the JSON document.** Algorithm 2 on page 26 presents a recursive approach in where we walk through the whole JSON document. This approach checks whether the current node are an array or object. If the current node are an array or object we recursively call the function until we find node containing value and field. Adding or extending a node is performed by distinguishing whether the node already exists on KV or not which is represented by KV.checkNode method. If the node does not exist on the map yet then we add the node to the map by assigning the node id, node name, node type, node depth, parent id. After creating the node on the map we get the node and assigned its sample value and value type with the node. At the same time, the algorithm stores the occurrence of a node. At any point, if the same node occurred again the algorithm detects that and get the node key-value mapping from KV map and update the occurrence of that node. At the same time, it also adds the new value of the node by performing the KV.addValue method.
Algorithm 2 Walk through the JSON document

1: procedure KV.walker(Nj, D, P, Pi, N, M, i)
2:     if Nj.isArray then
3:         Iterator iter = Nj.hasElement;
4:         while iter.hasNext do
5:             Nj = iter.next;
6:         if Nj.isArray or Nj.isObject then
7:             KV.walker(Nj, D + 1, P, Pi, N, M, i);
8:     if Nj.isObject then
9:         Iterator iter ← Nj.fieldNames;
10:        while iter.hasNext do
11:           KV.checkNode(D, P, N, name);
12:          if KV.checkNode = 0 then
13:              j = i;
14:              i += 1;
15:              KV.addNode(i, Pi, P, name, D, type, N, M);
16:              Mj=KV.getNode(KV.check, N);
17:              KV.addValue(Mj, Nj);
18:          if Nj isArray or Nj.isObject then
19:              KV.walker(Nj, D + 1, P, Pi, N, M, j);
20:          if KV.check > 0 then
21:              Mj=KV.getNode(KV.check, N);
22:              KV.addValue(Mj, Nj);
23:          if Nj isArray or Nj.isObject then
24:              KV.walker(Nj, D + 1, P, Pi, N, M, KV.check);

Check node of the KV. Algorithm 3 on page 26 presents the check method of a node is already existed on the map not. Adding a node to KV map is performed by distinguishing whether the node already exists on KV or not. If the node already exists on KV then the node is updated by appending the new value of it.

Algorithm 3 Check node of the KV

1: procedure KV.checkNode(D, P, N, name)
2:     for s ∈ N do
3:         n ← N(nodename);
4:         p ← N(parentname);
5:         d ← N(depth);
6:         if D = d and P = p and name = n then
7:             return N(id);
3.3. Schema Extraction

Add node to the KV. Algorithm 4 on page 27 presents the approach to create a node. Every node is stored with the information on a unique node id, node name, parent id, and name, the depth of the node of the JSON document.

Algorithm 4 Add node to the KV

1: procedure KV.addNode(i, Pid, P, name, D, type, N, M)
2: N : (id_n, name, pid_n, pname_n, depth_n, type_n) ← (i, name, Pid, P, D, type)

Add value to the KV. Algorithm 5 on page 27 presents the approach to add the data value to the created node. From N_j node the algorithm gets the data type of the node and then it assigned the type of a string of type_n. Using the type_n as a key the algorithm creates a map where type_n is the key and its associated value stored in value_n.

Algorithm 5 Add value to the KV

1: procedure KV.addValue(M_j, N_j)
2: N : (type_n, value_n) ← (N_j, M_j)

construct KV. Algorithm 6 on page 27 presents the approach to add a key-value pair on the map KV.

Algorithm 6 Construct KV

1: procedure KV.make(i, Pid, P, name, D, type, N, M)
2: M : (pid_n, KV.addNode) ← (Pid, KV.addNode)

3.3.4 Example of Schema Extraction Algorithm

In the following, we provide an example. In a library application, we assume the two book objects of a JSON document from 3.1 on page 27 and 3.2 on page 28. We assign unique node numbers to identify the node. We also assign the depth number. This simple numbering scheme is sufficient here because the numbers only serve to identify the components and depth of schema extraction. We assume that during schema extraction the JSON documents are not changing.

Listing 3.1: JSON document of a book object

```json
{
  "title": "Data Generation Technique",
  "content": "Database are ..",
  "author": "John Doe",
  "date": "2017-10-10 23:42",
}
At first, a key-value map is generated from the above JSON document 3.1 on page 27. We initiate the parent node as a predefined node name as "root" and set its id 1. We also mark that this the initial depth of a JSON tree as we initialize the depth as 0. When we initialize to traverse the JSON document we found new node namely as title, content, author, date, comments and all of these nodes added to our KV map by assigning a unique id and its depth. In here we find comment node is an array of object, therefore we store comment node as an object with its id and depth.

**Figure 3.5: Example of schema extraction first phase**

When a node is added, we distinguish two cases: if the node does not yet exist on the KV, we add the node. The node is stored with a node name, unique node id of the current node and the unique node number i, node depth, parent id, total occurrence and with a list that contains value and its data type. If the node already exists on the KV (since the same node occurred in some other input document) then the current node information is appended to the list that is stored with the node.

**Listing 3.2: JSON document of another book object**

```json
{
  "comments": [{
    "email": "user@gmail.com"
  }, {
    "reputation": 4
  }],
  "url": "http://www.db.com",
  "reference": "AAHKJ23"
}
```
3.3. Schema Extraction

Iterating the whole JSON document we get another book object of the depth 0. At this point our algorithm can detect that bookObj is already added to our KV map and thus we are not now going to add this node to the map. We get the already added node from the map and only add the new value and value data type to the map. While iterating the above JSON document 3.1 on page 27 we get node content and date inside comment object of depth 2, while these are the new node, we add these 2 nodes at depth 2 by assigning a new unique id.

Using our schema extraction approach it is also possible to illustrate the required and optional nodes. Each node is represented by a unique id and depth id. So at each depth, each node can occur only once with the same parent id. In figure 3.6 on page 29, the property title is required because on each sample JSON document the title fields is common. An example to the case of optional is in figure 3.6 on page 29 in where the property reputation for depth 2 is optional because it has the occurrence 1 while it parent comment has occurrence 2 in the whole JSON document.
Chapter 3. Approach

The general idea of schema extraction from JSON documents are introduced into the above algorithm. Its implementation is slightly more involved, since it additionally derives the data type (string, number, boolean, array, or object), stores its data values and also stores the total occurrence of each record of the input JSON documents. If in the input instances the same properties with different data types exist, our algorithm stores both the data type, for instance [{"type":"string","type":"integer"] and its associated values in the KV map.

3.4 Data Generation

The data generator is not settled in any predefined use case. There is no predefined data model. Rather the generator aims to generate data for any data model provided by users own use case. The data generator extracts user provided sample datasets and define a data model for that sample datasets together with data production rule that allows datasets to be scaled to arbitrary size using the total number of target data object or target data volume as a scale factor.

3.4.1 Data Generation Process

Figure 3.7: Sequence diagram of data generation process

Figure 3.7 on page 30 shows the data generation sequence diagram that references the sequence diagram "request". The sequence starts at the top left, with the user sending a message to the controller object. At that point, the data generation sequence diagram
is called, with the extracted schema, target volume, method as a parameter. Data
generation controller sends the message to service. Service sends the request to the
data generation object who is responsible to generate resulted data. At this point data
generator checks which data generation object should be invoked from the user request.
For the target, data generation process method checker reads user request for every node
generation method and sends the request to value generator object to generate value of
the given method. The value generator object generates among random, duplicate or
unique value for every node by preserving the original data value. The resulted data
then passed to the user.

3.4.2 Identification of Target Data Volume
From the graphical user interface the user can define his target data generation of two
choices which are namely data generation by volume and data generation by count. The
user can provide how many target data he wants by providing some number. On the
other hand, the user can also define his desired target data by providing data volume in
megabytes.

Definition 10 (Target data volume generation) Creates $n$ objects in where $n$ is the tar-
get data object and it is calculated by

- $n \times m \leq x$, in where
  - target data volume is $x$ (in MB)
  - every object volume is $m$
- Or $n$ is calculated by total number of desired object $y$ in where $n = y$.

3.4.3 Identification of Key Frequency per record
Usability of a data generator strongly depends on its ability to preserve the patterns of
the original data. Preserving the pattern of actual data plays a vital role in the design of
the data generator. Hence, pattern extraction is not limited to only extract the key-value
relationship but it also crucial to extract the frequency of every key of a document.

On 2.1 on page 8 we give present a sample JSON document. The example JSON
document consists of two book objects in where some key occurs to both objects and some key
occurs to a single object. After extracting the schema from the sample JSON documents
our data generator store all the key occurrence of the KV map which we discussed on algorithm 2 on page 26. In this section, we discuss the methodology of our data generator to generate new data onto the same key frequency which occurred in the real data sets.

**Algorithm 7 Identification of Key Frequency:1**

1: **procedure** \(K.F.\text{init}(O, A, D)\)
2: // \(O\) = map of actual occurrence at sample data
3: // \(A\) = total number of data at sample data sets
4: // \(D\) = target data count
5: \(NO = O.\text{keySet().size()}\)
6: \(CNO = (\text{long})\text{Math.ceil}((NO * D)/A)\)
7: \(CENO = (\text{long})\text{Math.ceil}(CNO/NO)\)
8: initialize map \(NEWO\)
9: for \(o \in O.\text{keySet()}\) do
10: \(NEWO.add(s, CENO)\)

**Identification of key frequency 1.** Algorithm 7 on page 32 represents the approach to identify the target generated data every field occurrence. The algorithm takes the original sample data sets node occurrence of KV map, original data count and target total data count. It then calculates the probability of occurrence of every node for the target data count.

**Algorithm 8 Identification of Key Frequency:2**

1: **procedure** \(K.F.\text{set}(\text{List < VC > vc})\)
2: // \(VC\) consists of
3: // \(id\) = unique id of the node
4: // \(O\) = occurrence of the node
5: // \(L\) = loop of the node
6: // \(VM\) = value map of the node
7: initialize map \(NEWO\)
8: for \(v \in vc\) do
9: initialize map \(m = v.\text{getOccurance()}\)
10: for \(k \in m.\text{keySet()}\) do
11: \(oc = K.F.\text{getMax}(k, vc)\)
12: if \(K.F.\text{isnotmax}(NEWO, k, oc)\) then
13: \(NEWO.add(k, oc)\)
3.4. Data Generation

**Identification of key frequency 2.** Algorithm 8 on page 32 represents the approach to set every node frequency of the target data. Every node store its target occurrence of every loop. In here the loop context is used to represent how many identical object presents in the data sets. All of these properties are stored for every node by assigning its unique id.

**Algorithm 9 Identification of Key Frequency:**

1: procedure $KF.max(k, List < VC > vc)$
2: $t = 0$
3: for $v \in vc$ do
4: initialize map $m = v.getOccurance()$
5: if $m.containsKey(k)$ then
6: $tm = m.getFirst(key)$
7: if $tm > t$ then
8: $t = tm$

**Identification of key frequency 3.** Algorithm 9 on page 33 represents the approach to identify the probable maximum occurrence of each node for the target generated data.

The probability of occurrence of each node is calculated using the relative frequency node in the sample data. For example, in the given sample data on 2.1 on page 8 has two book objects and content, date node is only present at the first book object. On the other hand title, author, URL, reference are present on both of the book object. If the data generator has the target to generate book object four times, the probability of selecting the title, content are:

\[
\frac{\text{title} = 2 \times \text{target} = 4}{\text{actual} = 2} = 4 \quad (3.1)
\]

\[
\frac{\text{content} = 1 \times \text{target} = 4}{\text{actual} = 2} = 2 \quad (3.2)
\]

The data generator identifies and sets that among four-time target book objects, all four of them have the node title and only two of them have the node content.

**Listing 3.3: Example of data generation by identifying key frequency**

```json
[
  {
    "title": "Data Generation Technique",
  },
  
```
The data generator has also the mechanism to generate data without identifying the key frequency. From the graphical user interface if user selects to generate data without identifying the key frequency the data will be generated in a flat hierarchy in where the data generator assumes that every node occurrence is equal in every object.
3.4. Data Generation

3.4.4 Identification of Key-Value Relationship

On 2.1 on page 8 our sample JSON document consists of two book objects in where the key namely reference has occurred in both book object and in every object, it has the different data types. One object has the data type String of the format alphanumeric and another object has the data type string of the format URL. After extracting the schema from the sample JSON documents our data generator store all the key occurrence of the KV maps which we discussed on algorithm 2 on page 26. In this section, we discuss the methodology of our data generator to generate new data by preserving the key-value relationship which occurred in the real data sets.

If in the input instances the same properties with different data types exist, our algorithm stores both the data type, for instance "type": "string", "type": "integer" and its associated values in the KV map.

Algorithm 10 Identification of key-value relationship: 1

1: procedure KVR.init(VC, d)
2: // VC is the map of actual data type and value
3: // d is the target total data
4: initialize map cv
5: initialize a = KVR.actual(VC)
6: for k ∈ VC.keyset() do
7: initialize collection m = VC.get(k)
8: initialize tm = m.size()
9: initialize tx = Math.ceil((tm * d) / a)
10: cv.add(k, tx)

Identification of key-value relationship 1. Algorithm 10 on page 35 represents the approach to identify the target generated data every field key-value relationship. The algorithm takes the the map of actual data type and value from KV map, and target total data count. It then calculates the probability of value data type occurrence of every node for the target data count.

Algorithm 11 Identification of key-value relationship: 2

1: procedure KVR.actual(VC)
2: initialize t = 0
3: for k ∈ VC.keyset() do
4: initialize collection m = VC.get(k)
5: t = t + m.size();
Identification of key-value relationship 2. Algorithm 11 on page 35 represents the approach to calculate the actual total value of the real data sets which is stored in the KV map.

The probability of occurrence of each node value type is calculated using the relative frequency node value type of the sample data. For example, in the given sample data on 2.1 on page 8 has two book objects and node title has value type string on every book object. On the other hand, node reference has value type alphanumeric in one book object and value type URL in another book object. If the data generator has the target to generate book object four times, the probability of value type string/alphanumeric/URL for node title, reference are:

\[
\frac{\text{title} - \text{type} - \text{string}}{\text{actual}} = \frac{2 \times \text{target}}{4} = 4
\]  

\[
\frac{\text{reference} - \text{type} - \text{string/alphanumeric}}{\text{actual}} = \frac{1 \times \text{target}}{2} = 2
\]  

\[
\frac{\text{reference} - \text{type} - \text{string/url}}{\text{actual}} = \frac{1 \times \text{target}}{2} = 2
\]

The data generator identifies and sets that among four-time target book objects, all four of them have the node title with value type string and only two of them have the node reference with value type string in alphanumeric format and another two of them have the node reference with value type string in URL format.

Listing 3.5: Example of data generation by identifying key-value relationship

```json
[
{
  "title": "Data Generation Technique",
  "reference": "AAHKJ23",
},
{
  "title": "NoSql Databases",
  "reference": "AAHKJ24",
},
{
  "title": "Advance Software Development",
  "reference": "http://www.example.com",
},
{
  "title": "Another great article",
}
]
3.4.5 Identification of Value Generation Method

From the graphical user interface, user can choose his desired data generation method of every node. The user can choose a mixture of generation method of his target generated data. The data generator currently supports three types of data generation method of each node which are:

- Mimic Unique
- Mimic Duplicate
- Random

The data generator then generates data according to the selected data generation method.

Mimic Unique

Mimicking data has the characteristics to identify the properties of data, capture the characteristics of data to generate similar synthetic data. Our data generator divides the data mimicking technique into two groups and one is called mimic unique. This method has the responsibility to generate data by mimicking the real data by making it sure that in the generated data no duplicate data exist.

Mimic Duplicate

Another group is called the mimic duplicate. This method has the responsibility to generate data by mimicking the real data in where duplicate data may exist.

Random

Random data generator is just generating data randomly. It does not extract the real data pattern rather it randomly generates data. At most of the cases, we use Java random class to generate the random number. Generating random text uses real data pattern to determine how many words in a sentence contains and generate the random word in that limit.
Chapter 3. Approach

All of this data generation method is going to be discussed in the section next sections.

3.4.6 Generation of Numerical Data

This section explains the method we follow to generate numerical attributes of the data. For each distinct key value of a node, we extract a list of all values for that node from the real data. This list of values is gathered for all node values of all numerical attributes in the real data. Our data generator supports the following numerical data types:

- Integer
- Long
- Float
- Double
- Short

In the previous section, we introduce three kinds of data generation technique to generate data. The data generator extracts the pattern of numerical data onto all of these data generation techniques.

Mimic Unique Number

Algorithm 12 on page 39 represents the approach to generate mimic unique numerical data. In here generating mimic unique number means to generate data from the real data set by keeping the important characteristic of the real data and also at the same time not repeating any data value. All of the generated data in this way will be unique and keep the important characteristic of the real data. From the stored values of the real data, the algorithm identifies the maximum and minimum value of the real data. While generating data it sequentially generates data until the maximum data value occurred. At the same time, it stores the last generated data value identifies the possible next value.
3.4. Data Generation

Algorithm 12 Mimic unique number

1: procedure N.unique(CollectionC, List < CV > cv, List < TCV > tcv, n)
2: // C = already stored read data set values
3: // cv = List of current value after each iteration
4: // tcv = List of temporary current value
5: // n = unique id of node
6: initialize r = range of C by extracting min and max
7: initialize mi = minimum value of the C
8: initialize ma = maximum value of the C
9: initialize num = 0
10: if cv.isEmpty() then
11: num = mi
12: else
13: if CV.isExist(cv, n) then
14: z = CV.getValue(cv, n)
15: if z + 1 <= ma then
16: num = z + 1
17: else
18: num = mi

Mimic Duplicate Number

Algorithm 13 on page 39 represents the approach to generate mimic duplicate numerical data. In here generating mimic duplicate number means to generate data from the real data set by keeping the important characteristic of the real data. In this process, the generated might be duplicated but in the range of real data. The algorithm starts by identifying the maximum and minimum value from the stored values of the real data. It then generated random data onto the range by using threadlocalrandom which we describe on 2.

Algorithm 13 Mimic duplicate number

1: procedure N.duplicate(Collectionvc)
2: initialize mi = vc.iterator().next()
3: initialize ma = vc.iterator().next()
4: for v ∈ vc do
5: calculate min and max of cv
6: num = ThreadLocalRandom.current().nextInt(mi, ma + 1)
Random Number

Algorithm 14 on page 40 represents the approach to generate random numerical data. Just like 13 on page 13 this algorithm identifies the maximum and minimum value from the stored values of the real data and generate random data. In this process instead of generating any random number, our algorithm generates random data in the range of the highest maximum number of the maximum number length.

**Algorithm 14 Random number**

1: procedure `N.duplicate(Collection vc)`
2: initialize `mi = vc.iterator().next()`
3: initialize `ma = vc.iterator().next()`
4: for `v ∈ vc` do
5: calculate min and max of `cv`
6: `s = Integer.toString(ma).length()`
7: `r = (int)Math.pow(10, s − 1)`
8: `num = ThreadLocalRandom.current().nextInt(mi, r * 10)`

On 2.1 on page 8 we give present a sample JSON document. The example JSON consists of two book objects in where node commentObject has node reputation which is an integer number. The data generator gets the minimum number 4 and maximum number 9 from the provided sample dataset. In the case of generating mimic duplicate number, the data generator generates data onto the range [4-9].

**Listing 3.6: Example of numeric data generation**

```json
[
  {
    "comments": [{
      "reputation": 4
    }],
  },
  {
    "comments": [{
      "reputation": 6
    }],
  },
  {
    "comments": [{
      "reputation": 7
    }]
  }
]```
3.4.7 Generation of Text Data

This section explains the method we follow for pattern extraction from text data. For each distinct key value of n node, we extract a list of all values for that node from the real data. This list of values is gathered for all node values of all text attributes in the real data. Our data generator finds the following pattern of a text data:

- Words
- Alphanumeric
- Email
- Url

Text Generator

On chapter 2 we discussed the finding the alphanumeric, email, URL pattern from a text data. In this section, we discuss finding the pattern of a textual data by analysis the total number of words every sentence of the real data.

**Definition 11 (Text generation)** Textual data value consists of

- \( w \) is number of words which are randomly chosen from a dictionary \( d \)
- \( w \) is calculated by \( a \leq w \leq b \) in where:
  - \( a \) is minimum total number of words consists in a single text data
  - \( b \) is maximum total number of words consists in a single text data
  - dictionary \( d \) is constructed from the sample datasets words
Algorithm 15 on page 42 represents the approach to textual data generation. At the very beginning, the algorithm gets all the stored values of sample data. It then creates a dictionary $d$ by splitting every sentence and extracts the words of the sentences. At the next step, the algorithm calculates the length of every word and gets the minimum and maximum length words. After that, the data generator calls 16 on page 43 to construct a new sentence.

Algorithm 15 Text generator

1: **procedure** $T$.generate($CollectionC$, $List < CV > cv$, $List < TCV > tcv$, $n$

2: // $C =$ already stored read data set values
3: // $cv =$ List of current value after each iteration
4: // $tcv =$ List of temporary current value
5: // $n =$ unique id of node
6:     initialize $d$
7:     initialize $mi$
8:     initialize $ma$
9:     for $v \in C$ do
10:         $td = ((String)value).split("\s+")$
11:     $mi =$ calculate minimum $td$ length
12:     $ma =$ calculate maximum $td$ length
13:     for $s \in td$ do
14:         if !$d$.contains($s$) then
15:             $d$.add($s$
16:     $sen = T$.sen()

Algorithm 16 on page 43 represents the approach to construct a new sentence. This is a recursive function where recursively we iterate the dictionary of words and construct a new sentence from the words. At the same time, the algorithm keeps a track of loop of the iteration to construct our resulted sentence. This step is necessary to generate unique data in where this loop tracking makes it possible to generate the next unique sentence.
Algorithm 16 Sentence generator

1: procedure T.sentence(w, ma, cu, cl, ll, d, r)
2:   //w = 0
3:   //ma = maximum word of a sentence
4:   //cu = empty string to store current word
5:   //cl = current loop
6:   //ll = last loop
7:   //d = dictionary of words
8:   //r = result sentence
9:   //Global going, initializing by true
10:   initialize i = 0
11:   while i < d.size() AND going do
12:     if w == ma then
13:       if BigInteger(cl).compareTo(BigInteger(ll)) == 1 then
14:         r.setLoop(cl)
15:         r.setResult(cu)
16:     else
17:       initialize oc = cu
18:       initialize ocl = cl
19:       initialize ow = w
20:       w ++
21:       cu+ = d.get(i) + ””
22:       cl+ = i
23:       T.sen(w, ma, cu, cl, ll, d, r)
24:     w = ow
25:     cu = oc
26:     cl = ocl

Example of Text Data Generation

Our data generator generates a new sentence by using provided sample datasets. On 2.1 on page 8 we present a sample JSON document. The example JSON consists of two book objects in where node title has different text value of each object. After extracting the schema from the sample JSON documents our data generator store all the values of the KV map which we discussed on algorithm 2 on page 26. From that stored values our generator creates a dictionary consisting of all unique word of the sample values. The resulting dictionary might be as follows:
Chapter 3. Approach

The data generator then calculates the minimum words occur to a single value of the sample datasets which are in this example 2: NoSQL Databases and the maximum words occur for a single value in the sample datasets which is in this example 3: Data Generation Technique. The data generator sets that among four-time target book objects, all four of them have the node title and the title value has maximum words of 3.

LISTING 3.7: Example of text data generation

```json
[
    {
        "title": "Data Data Data",
    },
    {
        "title": "Data Data Technique",
    },
    {
        "title": "Data Generation Technique",
    },
    {
        "title": "Data Generation NoSQL",
    }
]
```

3.4.8 Generate Data from Identified and Extracted Technique

After getting the extracted model of sample datasets which we discussed on ?? and using the algorithm of 7 on page 7 and 10 on page 10 our data generator generate unique, duplicate or random data depending on user choice in the graphical user interface. The data generator is able to generate data for the following data stores:

- JSON
- graphson
- XML
- Cypher
- SQL
New textual or numerical data is generated at the above format and stores in a file at the user provided destination. Based on target data store the data generator populates the target file with any of the above formats by new data. While populating the file with data the generator maintains the proper syntax format for the respective file format. The generated file can then be imported to the respective data stores.
Chapter 4

Experiment and Results

In this chapter, we discuss the experiments conducted to test and validate our data generation approach. The main goal of these experiments are to verify that the schema extracted from the sample datasets present in the original datasets and also to verify that the key frequency and key-value relationship which we discussed on section 3.4.3 and section 3.4.4 preserves the patterns present in the original data fed to the data generator. Then we make an experiment to verify the generated text and numeric data patterns present in the original data sets. The first section of this chapter gives the details of the experimental setup, and the three datasets used in our experiments. The next section discusses the experiment result. In the last section, the scalability of the data generation process is discussed.

4.1 Experimental Setup

Our data generation approach was validated using three different datasets which are Iris dataset[Zai13], MSME Country Indicators dataset[Ban18], and a custom dataset which we called task dataset. The characteristics of these datasets are discussed in detail in the following sub-sections.

4.1.1 IRIS Dataset

The Iris flower dataset is a publicly available dataset and the best-known database to be found in the pattern recognition literature[UCI]. This dataset contains values of sepal length, petal length, sepal width, and petal width, for different classes. Among these five attributes, only ‘species’ contains textual value and the remaining attributes contain numerical value (figure 4.1). The dataset we are using in our experiment has 150 objects and the file-size is about 15.8KB
4.1. Experimental Setup

TABLE 4.1: Sample IRIS Dataset Adapted from [Zai13]

<table>
<thead>
<tr>
<th>sepalLength</th>
<th>sepalWidth</th>
<th>petalLength</th>
<th>petalWidth</th>
<th>species</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>6.2</td>
<td>2.2</td>
<td>4.5</td>
<td>1.5</td>
<td>versicolor</td>
</tr>
</tbody>
</table>

4.1.2 MSME Country Indicators Dataset

The second dataset we use in our experiment is MSME country indicator dataset which is adapted from this source [Ban18]. According to [Ban18] MSME-CI is a collection of secondary data of various institutions like: statistical institutes, ministries, international organizations, small business promotion agencies, research institutions. This is a good dataset which consists of a deep nesting of JSON object and also consists of all kind of numerical and textual data values. The dataset has only one record, however, this single object has a very deep nested object structure and the file-size is about 33.3KB. In appendix A we present a sample of this dataset and it shows that this dataset has in total 95 keys which consists of boolean, numerical and textual values.

4.1.3 Task Dataset

The final dataset of our experiment is a Task dataset which is a custom-made dataset of our own. In appendix A we presents a sample of this datasets. This dataset is a good candidate to check our schema extraction process as this dataset has a deep and complex level of nesting. The task dataset has the key named ‘name’, ‘tasks’ and ‘subtasks’. Each task and subtask keys has its own nested key namely as name, worker. Also worker has its nested key company and company also has its nested key name. In this way, our tasks dataset is nested at four levels of depth. This dataset contains only the textual field value which is also a good experimental point to check the data generation time of textual data. As generating textual data is time-consuming. The task dataset has 10 records and the file size is about 24.1 KB.

4.1.4 System Configuration under Test

This system for data generation, schema extraction from JSON data with all the algorithms for identifying key frequency, unique/random data generation discussed in Chapter 3, has implemented using Java version 10.0.21 with Spring-boot version 1.5.92. The graphical user interface is a web-based user interface and implemented using

1https://www.oracle.com/java/, last accessed on August 15, 2018
2https://spring.io/projects/spring-boot, last accessed on August 15, 2018
Angular version 5.2. The experiments were executed on a Linux platform of operating system Ubuntu 18.04.1 LTS Bionic Beaver version. All tests are conducted on an Intel core i7 machine with 2.8 GHz clock rates, 32 gigabyte RAM and a 400 GB SATA hard disks drive. Table 4.2 on page 48 summarizes the system configuration which we used in our experiment.

<table>
<thead>
<tr>
<th>System</th>
<th>Ubuntu 18.04 (bionic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>4.15.0-34-generic</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>32053 MiB</td>
</tr>
<tr>
<td>Storage</td>
<td>400GB / Samsung SSD 850</td>
</tr>
</tbody>
</table>

4.2 Methodology of the Experiment

We applied following test procedure for every dataset

- **Load dataset** We load the dataset at the machine under test and from the graphical interface set the dataset file location. We also give the read permission of the dataset location to the generator.

- **Execute extraction of data models** From the graphical interface we execute to extract the schema of the loaded dataset. After extracting the schema the system shows us a tree representation of the extracted data model which we can use to cross-check the extracted data model on the original dataset data model.

- **Execute tests run** We execute the test run for every dataset by giving the target data volume of megabytes. For every dataset, we execute a test run for 25 times. For every target data-store the test run is executed in both flat and keeping key frequency data generation pattern, and for both patterns, the test run is executed in unique, mimic duplicate, random and mixed value generation techniques.

- **Execute query to database** Our proposed data generator is generating as JSON, SQL, XML and Cypher data format. After executing every test run we import the generated file to target data store namely as: Mongodb, Tinkerpop Gremlin,

---

3[https://angular.io/](https://angular.io/), last accessed on September 14, 2018  
4[http://releases.ubuntu.com/18.04/](http://releases.ubuntu.com/18.04/), last accessed on September 15, 2018  
5[https://www.mongodb.com/](https://www.mongodb.com/), last accessed on August 15, 2018  
Neo4j\textsuperscript{7}, and Mysql\textsuperscript{8} by using the data-store vendor console. We then execute some basic query to make sure whether our generated data file for those formats is error free or not. In addition, for the JSON dataset, we use jsonlint\textsuperscript{9} to check whether the generated JSON syntax has errors or not.

- **Shutdown system, clear caches, and restart system** After every test runs we shutdown the system under test, clear the caches and restart the system again.

## 4.3 Results and Analysis

The section starts with the analysis of schema extraction from original datasets which we use to generate new data. Then in this section, we analyze the generated data using the above three datasets by running our data generator of the above-mentioned system and compare their generated data value to their original counterparts.

### 4.3.1 Analysis of the Schema Extraction from JSON

Let us begin by analyzing the schema extraction from sample JSON dataset. As stated before the concept of our data generator is to extract schema and create the data model from user provided sample JSON dataset. Later this extracted data model will be used to generate new data. We at the first start analyzing the schema extraction from IRIS dataset.

![Figure 4.1: Extracted data model of IRIS dataset](image)

Figure 4.1 on page 49 represents the data model of IRIS dataset after extraction of schema from real IRIS dataset. We can see from the figure that extracted schema has five

\begin{itemize}
\item root
\item sepal Length
\item sepal Width
\item petal Length
\item petal Width
\item species
\end{itemize}

\textit{datatype : double}

\textit{datatype : double}

\textit{datatype : double}

\textit{datatype : double}

\textit{text}

\textsuperscript{7}https://neo4j.com/, last accessed on August 15, 2018

\textsuperscript{8}https://www.mysql.com/, last accessed on August 15, 2018

\textsuperscript{9}https://jsonlint.com/, last accessed on August 15, 2018
keys of the same depth and among them, only ‘species’ has the textual data as value and remaining four key has the numerical data as values. In section 4.1.1 on page 46 we discussed the real IRIS dataset structure and we can see that the extracted schema reserves the same characteristics of the real IRIS dataset.

Next, we analyze the schema extraction of MSME country indicator dataset. Before starting the analysis we have to modify some key’s value such as query, format, rdf-class, geo-coding-attached in the original MSME dataset because in the original dataset these keys contain the empty object and our data generator does not yet give support to an empty object. Figure 4.2 on page 51 represents the data model of MSME country indicator dataset after extraction of schema from the real dataset. We can see from the figure that extracted schema has in total of 95 keys of a different depth. All of the keys contain the same data type as a value identical to their real data set’s value data type. On section 4.1.2 on page 47 we discussed the real MSME dataset structure. We also observed from our analysis that the extracted data model is identical to its original dataset in terms of their node depth of the tree.
FIGURE 4.2: Extracted data model of MSME Country Indicators dataset
At last, we analyze the schema extraction of Task dataset. Figure 4.3 on page 52 represents the data model of Task dataset after extraction of schema from the real dataset. On the other hand on section 4.1.3 on page 47 we discussed the Task dataset structure. We can see from the figure that extracted schema has in total of 19 keys of a different depth. On section 4.1.3 on page 47 we discussed that task dataset only contains textual data value and also in our extracted data model we can observe that the key ‘name’ only contain textual data value which is similar to the real dataset. We also observed from our analysis that the extracted data model is identical to its original dataset in terms of their node depth of the tree.

4.3.2 Analysis of the Data Generation by Identifying Key Frequency per Record

Next let us analyze the data generation by identification key-frequency approaches from sample JSON dataset. In this section we only analyze whether the generated data represents the same key frequency per record of the original dataset or not. In the later chapter we also analyze data generation by key frequency of comparing to flat data generation.
Table 4.3 on page 53 represents the comparison of original IRIS dataset and generated IRIS dataset. For the experiment we modify the real IRIS dataset and the sample IRIS dataset contains only five records. Among them first record does not contain sepal length and sepal width key, second object does not contain petal length, petal width key. Also in the sample dataset third object contains all the keys except species key. However fifth object contains all the keys. We run our system to generate 20 new records of the sample five records. Table 4.3 on page 53 shows that among 20 new records 16 of them contain petal width key. For example we can assume that our data generator generates the first record four times while generating 20 new records. In the generated four times first record none of them contain sepal length and sepal width keys which is identical to real dataset.

<table>
<thead>
<tr>
<th>Appearance in 1st Object in Original data</th>
<th>Appearance in 2nd Object in Original data</th>
<th>Appearance in 3rd Object in Original data</th>
<th>Appearance in 4th Object in Original data</th>
<th>Appearance in 5th Object in Original data</th>
<th>Appearance in 1st Object in Generated data</th>
<th>Appearance in 2nd Object in Generated data</th>
<th>Appearance in 3rd Object in Generated data</th>
<th>Appearance in 4th Object in Generated data</th>
<th>Appearance in 5th Object in Generated data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1</td>
<td>1 1 0 0 1</td>
<td>1 1 1 1 0</td>
<td>1 0 1 0 1</td>
<td>1 1 1 1 1</td>
<td>0 0 1 1 1</td>
<td>1 1 0 0 1</td>
<td>1 1 1 1 0</td>
<td>1 0 1 0 1</td>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>Total Appearance in Original data</td>
<td>Total Appearance in Generated data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 3 4 3 4</td>
<td>16 12 16 12 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At the next section we will analyze the data generation of different format for all three datasets which we discussed previously. For every target data-store the test runs is executed in both flat and keeping key frequency per record data generation pattern, and for both patterns the test run is executed in unique, mimic duplicate, random and
mixed value generation techniques. The flat data generation represents that data will be generated in a flat hierarchy in where every generated record all the key one time. On the other hand keeping key frequency per record data generation represents that data will be generated by maintaining the original datasets key frequency per record. For the unique, random and mimic duplicate data generation we set our target data volumes 10 megabytes. However for the mixed data generation we set our target data volume 1 gigabytes. In here the mixed data generation defines that data will be generated in a mixture of unique, random and duplicate data generation techniques. For the MSME dataset experiment we don’t make the experiment to generate unique data as our data generator is built in a way the target data volume can not be greater than maximum possible data onto unique dataset. For the MSME dataset we mixed unique and duplicate data generation of the phase of unique data generation experiment.

4.3.3 Analysis of the Data Generation in JSON Format

In this section, we will analyze the data generated in JSON format for all three datasets. Figure 4.4 on page 54 shows the data generation of the IRIS dataset in JSON format. The following results are displayed below in seconds.

Figure 4.4: JSON data generation of IRIS dataset

Figure 4.5 on page 55 shows the data generation of MSME dataset in JSON format. The following results are displayed below in seconds.
4.3. Results and Analysis

**Figure 4.5:** JSON data generation of MSME Country Indicators dataset

Figure 4.6 on page 55 shows the data generation of Task dataset in JSON format. The following results are displayed below in seconds.
From the above bars, we can see that generating data by identifying key frequency per record takes longer than flat data generation. However, this behavior is expected because with the identifying key-frequency per record technique the system has to compute the possible next record data model after creating each record. From the above bars, we also find that deeply nested data structure takes a longer time to generate to compare to single depth data structure. As IRIS dataset does not have deep nesting and on the other hand task dataset has deep nesting. Generating data for task dataset takes longer time to compare to IRIS dataset. Table 4.4 on page 56 contains the number of records of our generated dataset for all three datasets. From the table and the plots against we can see that to generate 6617833 records of IRIS dataset the data generator takes about 24 minutes, on the other hand generating 18799 records of MSME dataset the data generator takes about 95 minutes.

<table>
<thead>
<tr>
<th></th>
<th>Flat</th>
<th></th>
<th>With-Key-Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unique</td>
<td>Duplicate</td>
<td>Random</td>
</tr>
<tr>
<td>Size of generated data</td>
<td>10MB</td>
<td>10MB</td>
<td>10MB</td>
</tr>
<tr>
<td>Number of records in generated IRIS data</td>
<td>66033</td>
<td>66840</td>
<td>66193</td>
</tr>
<tr>
<td>Number of records in generated MSME Country Indicators data</td>
<td>X</td>
<td>4057</td>
<td>4518</td>
</tr>
<tr>
<td>Number of records in generated task data</td>
<td>27403</td>
<td>27191</td>
<td>26128</td>
</tr>
</tbody>
</table>

### 4.3.4 Analysis of the Data Generation in GraphSON Format

In this section, we will analyze the data generated in GraphSON format for all three datasets. Figure 4.7 on page 57 shows the data generation of the IRIS dataset in GraphSON format. The following results are displayed below in seconds.
4.3. Results and Analysis

Figure 4.7: GraphSON data generation of IRIS dataset

Figure 4.8 on page 57 shows the data generation of MSME dataset in GraphSON format.

Figure 4.8: GraphSON data generation of MSME Country Indicators dataset
Figure 4.9 on page 58 shows the data generation of Task dataset in GraphSON format.

From the above bars, we can see that generating data by identifying key frequency per record takes longer than flat data generation. From the above bars, we also find that deeply nested data structure takes a longer time to generate compared to single depth data structure. As IRIS dataset does not have deep nesting and on the other hand MSME dataset has deep nesting. Generating data for MSME dataset takes longer time to compare to IRIS dataset. Table 4.5 on page 59 contains the number of records of our generated dataset for all three dataset. From datasets and the bars we can see that to generate 3123114 records of IRIS dataset the data generator takes about 11 minutes, on the other hand generating 89054 records of MSME dataset the data generator takes about 20 minutes. Compare to the JSON data generation which we represent at table 4.4 on page 56 GraphSON data generation takes a smaller amount of time.
### 4.3. Results and Analysis

#### Table 4.5: Properties of Generated GraphSON data

<table>
<thead>
<tr>
<th></th>
<th>Flat</th>
<th>With-Key-Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unique</td>
<td>Duplicate</td>
</tr>
<tr>
<td>Size of generated data</td>
<td>10MB</td>
<td>10MB</td>
</tr>
<tr>
<td>Number of records in generated IRIS data</td>
<td>32396</td>
<td>32587</td>
</tr>
<tr>
<td>Number of records in generated MSME Country Indicators data</td>
<td>X</td>
<td>1694</td>
</tr>
<tr>
<td>Number of records in generated task data</td>
<td>5919</td>
<td>5910</td>
</tr>
</tbody>
</table>

#### 4.3.5 Analysis of the Data Generation in SQL Format

In this section, we will analyze the data generated in SQL format for all three datasets. Figure 4.10 on page 59 shows the data generation of the IRIS dataset in SQL format. The following results are displayed below in seconds.

![Figure 4.10: SQL data generation of IRIS dataset](image)

Figure 4.11 on page 60 shows the data generation of MSME dataset in SQL format.
Chapter 4. Experiment and Results

Figure 4.11: SQL data generation of MSME Country Indicators dataset

Figure 4.12 on page 60 shows the data generation of Task dataset in SQL format.

Figure 4.12: SQL data generation of Task dataset

For the SQL data generation, we only conduct the experiment for flat data generation as our data generation does not provide any support to generate SQL by identifying
key-frequency per record. From the above bars we can see that generating unique data takes longer than generating duplicate and random data. From the table and the bars we can see that to generate 5731324 records of IRIS dataset the data generator takes about 2 minutes, on the other hand generating 323466 records of MSME dataset the data generator takes about 1.5 minutes.

<table>
<thead>
<tr>
<th>Size of generated data</th>
<th>Flat</th>
<th>Unique</th>
<th>Duplicate</th>
<th>Random</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10MB</td>
<td>10MB</td>
<td>10MB</td>
<td>1GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of records in generated IRIS data</td>
<td>57484</td>
<td>57940</td>
<td>56797</td>
<td>5731324</td>
<td></td>
</tr>
<tr>
<td>Number of records in generated MSME Country Indicators data</td>
<td>X</td>
<td>3329</td>
<td>3324</td>
<td>323466</td>
<td></td>
</tr>
<tr>
<td>Number of records in generated task data</td>
<td>12690</td>
<td>12655</td>
<td>12654</td>
<td>1211392</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.6 Analysis of the Data Generation in Cypher Format

In this section we will analyze the data generated in cypher format for all three datasets. Figure 4.13 on page 61 shows the data generation of the IRIS dataset in Cypher format. The following results are displayed below in seconds.
Figure 4.14 on page 62 shows the data generation of MSME dataset in Cypher format.

Figure 4.15 on page 62 shows the data generation of Task dataset in Cypher format.
From the above bars, we can see that generating data by identifying key frequency per record takes longer than flat data generation. Table 4.7 on page 63 contains the number of records of our generated dataset for all three datasets. From the datasets and the bars we can see that to generate 5266037 records of IRIS dataset the data generator takes about 20 minutes, on the other hand generating 119515 records of MSME dataset the data generator takes about 24 minutes.

<table>
<thead>
<tr>
<th>Table 4.7: Properties of Generated Cypher data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Size of generated data</strong></td>
</tr>
<tr>
<td>Flat</td>
</tr>
<tr>
<td>Unique</td>
</tr>
<tr>
<td>10MB</td>
</tr>
<tr>
<td>Number of records in generated IRIS data</td>
</tr>
<tr>
<td>53126</td>
</tr>
<tr>
<td>Number of records in generated MSME Country Indicators data</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>Number of records in generated task data</td>
</tr>
<tr>
<td>9717</td>
</tr>
<tr>
<td>8787</td>
</tr>
</tbody>
</table>

After executing every test run we import the generated file to target data store namely as: Mongodb, Tinkerpop Gremlin, Neo4j and MYSQL by using the data-store vendor console. We then execute some basic query to make sure whether our generated data file for those formats is error free or not. The experiment in this stage shows that we can successfully write our generated data onto the target database. The target of our system is to create data onto different data stores only from one sample JSON dataset and the experiment and analysis conducted above shows that the system is able to generate data onto a different data formats which we use in the different data-stores interface to load the data and execute the different query. This shows us that the generated data is correct in that data format corresponding data stores.

4.4 Constraint

- We assume user give valid JSON file as input
- We assume while generating data user does not provide new JSON file
- Preserving key-frequency of the original data by selecting "original" in the user interface works for all specified target data stores format except SQL
• Selecting "original" in the user interface to preserve key-frequency of the original data not implemented yet with unique value generation.

• Data generation works best if the original data value is correct and only contain English character

• Date generator does not support yet of the empty object

• Data generator partially work for the value generation as an array

• Data generation by identifying key-frequency per node works as expected relative to per node. It only checks whether at this record the key will present or not. At the same record if any nested key has different frequency the data generator takes the maximum frequency to generate data.

• Data generator may fail if the user chooses to generate data by providing target number and also choose keeping the key-frequency while the target number to generate data is too small. It’s best to generate data by volume while choosing to generate data by keeping key-frequency or give target number to generate data larger than original data record count
Chapter 5

Conclusion & Future Work

Through this work, we formulated a concept of a system that can be used to generate data onto different data-stores. The system extracted the data model from the sample JSON dataset and use that data model to generate new data that closely follow the real data key and value pattern for different data stores. By this way, the end user can get the flexibility to create data onto a different data format just from one sample JSON formatted dataset.

The system takes a sample dataset in JSON format. The system is capable of extract schema from the sample dataset and uses a set of dependencies as input to generate new data. The dependency list consists of keeping the original data set key frequency or flat key frequency of the additional input of generating target data value among unique, mimic duplicate and random techniques. Using the value generation technique of unique and duplicate, the system generates data closely to the pattern of real data in where the generated numerical data is in the range of real data and textual data is similar to real data. In this way, the system ensures to generate meaningful data rather than garbage data. Thus the system generates new value of a larger scale of every key corresponding to their data type in the real data.

To evaluate the system, we conducted the experiment on three different datasets: IRIS dataset, MSME country indicator dataset and task dataset which is discussed in section 4.1 on page 46. The experiment and analysis of schema extraction which is discussed at the beginning of section 4.3 on page 49 shows that the extracted data model to follow the same data type and depth of every key of original data. At next the experiment and analysis of data generation by identifying key frequency which is discussed in section 4.3 on page 49 show that the generated data preserves the same key same key distribution pattern of original data. The experiment conducted on section 4.3 on page 49 also showed that the data generated correctly for different data stores. Through the experiment conducted on the machine, we saw that flat data generation performs faster than data generation by keeping key-frequency technique. This behavior is also expected because with the identifying key-frequency technique the system has to compute the possible next record data models after creating each record. The target
of our system is to create data onto different data stores only from one sample JSON dataset and the experiment and analysis conducted on chapter 4 shows that the system is able to generate data onto different data formats which we use in the different datastores interface to load the data and execute the different queries. This shows us that the generated data is correct in that data format corresponding data stores.

In our long-term vision, we see our concept of data generation approach as a part in the research community in various domains to get more flexibility on the data manipulation and to test a wider set of scenarios in their application (such as benchmarking). Thus the concept of our data generation approach can formulate the answers which we discussed in section 1.2 on page 3.

5.1 Future Work

Next, let us discuss some of the possible ways of our concept of data generator enhancement and extension in future.

- Our current approach generating textual and numerical value by mimicking the original dataset. In future, we would like to extract the pattern distribution of original dataset value attributes and generate data very closely to original datasets rather than mimicking data.

- As our data generator system is written in Java and maintaining the code base in a modular way, it is possible to implement a new target data store by using the existing code base. In future, we would like to extend the target data store support.

- The main goal of our data generator concept are to generate new data onto different data stores and thus we are generating and storing the data onto a file for the target data stores supported file format. However, it is also possible to directly write the newly generated data directly to the target database. In future we would like to extend our data generator approach to write record directly to the target database besides generating the file.

- Lastly, we plan to bundle our synthetic data generator with the cross-domain benchmarking framework LITMUS [Tha17; Tha+17b; Kes+17], which will allow an open and fair benchmarking of RDF and Graph databases by enabling automatic execution SPARQL queries over property graphs using Gremlin traversals [Tha+18a; Tha+18b; Tha+17a].
Appendix A

Sample JSON File

A.1 Task dataset

Listing A.1: Sample Task Dataset

```json
{
    "name": "Project1",
    "tasks": [{
        "name": "Task4",
        "worker": {
            "name": "Worker4",
            "company": {
                "name": "Company3"
            }
        }
    }, {
        "name": "Task5",
        "subtasks": [{
            "name": "Subtask4.1",
            "worker": {
                "name": "Worker4",
                "company": {
                    "name": "Company3"
                }
            }
        }
    }
}
}
```
A.2 MSME Country Indicators dataset

Listing A.2: MSME Country Indicators Dataset

```json
{
    "oid": 342286,
    "provenance": "official",
    "columns": [{
        "width": 304,
        "dataTypeName": "text",
        "cachedContents": {
            "top": [{
                "item": "CSD 01 Manhattan",
                "count": 20
            }, {
                "item": "CSD 02 Manhattan",
                "count": 19
            }],
            "smallest": "CSD 01 Manhattan",
            "largest": "CSD 32 Brooklyn",
            "non_null": 32,
            "null": 0
        },
        "fieldName": "jurisdiction_name",
        "position": 2,
        "id": 2989248,
    }
}
```
A.2. MSME Country Indicators dataset

"name": "JURISDICTION NAME",
"tableColumnId": 1501756,
"format": "txt",
"renderTypeName": "text"
],
"rowsUpdatedBy": "txun-eb7e",
"rights": ["read"],
"hideFromCatalog": false,
"viewType": "tabular",
"createdAt": 1311775795,
"publicationDate": 1318116419,
"publicationAppendEnabled": false,
"owner": {
  "screenName": "NYC OpenData",
  "displayName": "NYC OpenData",
  "id": "5fuc-pqz2",
  "profileImageUrlSmall": "/api/users/5fuc-pqz2/profile_images/TINY",
  "type": "interactive",
  "profileImageUrlMedium": "/api/users/5fuc-pqz2/profile_images/THUMB",
  "profileImageUrlLarge": "/api/users/5fuc-pqz2/profile_images/LARGE"
},
"flags": ["default", "restorable", "restorePossibleForType"],
"viewCount": 6595,
"name": "School District Breakdowns",
"newBackend": false,
"category": "Education",
"publicationGroup": 238848,
"rowClass": "row",
"attribution": "Department of Youth and Community Development (DYCD )",
"hideFromDataJson": false,
"downloadCount": 2573,
"id": "g3vh-kbnw",
"totalTimesRated": 0,
"displayType": "table",
"description": "Demographic statistics broken down by school districts",
"metadata": {
"custom_fields": {
    "Legislative Compliance": {
        "Geo-coding Attached?": "yes"
    },
    "Dataset Information": {
        "Agency": "Department of Youth and Community Development (DYCD)"
    },
    "Update": {
        "Update Frequency": "As needed",
        "Automation": "No",
        "Date Made Public": "2011-10-08 23:26:49"
    }
},
"availableDisplayTypes": ["table", "fatrow", "page"],
"rowLabel": "",
"renderTypeConfig": {
    "visible": {
        "table": true
    }
},
"rdfSubject": "0",
"attachments": [{
    "filename": "school district breakdowns.xlsx",
    "assetId": "9a72dfaa-50e8-4008-aec9-e82fad89ef6d",
    "name": "school district breakdowns.xlsx",
    "blobId": 1
}],
"rdfClass": "rdf",
"rowIdentifier": "0"
}
A.2. MSME Country Indicators dataset

"type": "viewer",
"flags": ["public"]
},
"tableId": 244141,
"rowsUpdatedAt": 1318116415,
"tableAuthor": {
  "screenName": "NYC OpenData",
  "displayName": "NYC OpenData",
  "id": "5fuc-pqz2",
  "profileImageUrlSmall": "/api/users/5fuc-pqz2/profile_images/TINY",
  "type": "interactive",
  "profileImageUrlMedium": "/api/users/5fuc-pqz2/profile_images/THUMB",
  "profileImageUrlLarge": "/api/users/5fuc-pqz2/profile_images/LARGE"
},
"numberOfComments": 2
}
Appendix B

User Guide

B.1 Software Installation

B.1.1 Core Technology

- Frontend written by angular 5
- Backend written by java spring framework

B.1.2 System Requirements

- nodejs latest version
- npm latest version
- java
- maven
- tomcat
- angular cli

B.1.3 Installation

In this guide we are only discussing about the dependency installation process on ubuntu machine.

1. **install nodejs**: $ sudo apt-get install -y nodejs

2. **install npm**: $ sudo npm install npm@latest -g

---

1 https://nodejs.org/en/download/
2 https://www.npmjs.com/
3 http://java.com/download
4 https://maven.apache.org/download.cgi
5 https://tomcat.apache.org/download-80.cgi
6 https://cli.angular.io/
3. **install java:**
   
   - $ sudo add-apt-repository ppa:webupd8team/java
   - $ sudo apt-get update
   - $ sudo apt-get install oracle-java7-installer

4. **install tomcat:**
   
   - $ sudo mkdir /opt/tomcat
   - $ sudo chown -R <username> tomcat/
   - $ sudo tar xvf <apache-tomcat-8.5.4.tar.gz> -C /opt/tomcat –strip-components=1
   - $ sudo chgrp -R tocmcat conf
   - $ sudo chmod g+rwx conf

5. **install maven:** $ sudo apt-get install maven

**B.1.4 Run**

- Go to the project frontend folder source code and open proxy.conf.json file
- Set your backend webserver url. Default is set to localhost on port 8083
- From the command line run: $ npm install (to install all frontend dependencies)
- Then from the command line run: $ npm start
- Frontend will start on localhost at port 4200
- Go to the project backend folder path
- From the command line run: $ mvn spring-boot:run -Drun.jvmArguments=-Dserver.port=8083
- Backend will start on localhost at port 8083

**B.2 Software Features**

1. Extract sample json
2. View your json schema tree
3. Choice of target data store:
   
   - JSON
   - Graphson
Appendix B. User Guide

- SQL
- CQL (Cypher Query Language)
- XML

4. Choose your desired data volume

5. Choice of data generation technique:
   - By preserving key frequency of original data
   - Flat data generation

6. Based on your schema 3 types of data generation method supported:
   - **mimick unique**: Calculate from your sample data how many unique data generation possible by mimicking your sample data
   - **mimick duplicate**: mimicking your sample data, duplicate data generation possible
   - **random**: randomly generated data

7. Generate data

B.3 A walk through of the software

1. Provide your sample json data source path in json format

![Figure B.1: upload sample data](image-url)
2. View your extracted json schema

![View schema](image)

**Figure B.2:** view schema

3. Choose for which datastore or format you want to generate data

![Data format choice](image)

**Figure B.3:** data format choice

4. Choose whether do you want to generate data by keeping the kry frequency of original data sets or not
5. Choose whether do you want to generate data by providing volume in MegaBytes or by providing total number of desired object.

6. Set data generation volume by providing the target MB / scale factor how many times the want to generate data.
7. Choose the method for every field data generation technique. For every field according to your schema you can choose either unique, duplicate or random data generation technique. For every choice the window show you maximum how many data for this field by your preferred generation technique is possible.
We provide data generation in three ways: Mimic your sample data uniquely, Mimic sample data duplicate and random data generation. Mimic unique selection might not produced your desired data volume, if you does not care about unique data then choose mimic duplicate technique. Click Next to proceed.

8. Give your output file path location

Please give your sample data file folder path only. Make sure your sample data file location is writable to the program. Create an empty folder and give the path with trailing hash(/). Everytime you generate data the folder will be emptied by deleting all file inside the folder. Click Next to proceed.
9. Generate data

![Figure B.9: data generation](image)

10. Generated json data and tested it with mongo console to import generated json file and query the data. The output of generated json is like this:

```
[{
    "title": "Databases Data NoSql", "content": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.", "email": "un@gmail.com", "reputation": 1333,
    "url": "https://example.com"
},
    {
    "title": "Databases NoSql NoSql", "content": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.", "email": "io@gmail.com", "reputation": 1333,
    "url": "https://example.com"
}]
```

![Figure B.10: generated json data](image)

11. Generated graphson data and tested it with gremlin console to import generated graphson file and query the graph. The output of generated graphson is like this:
12. Generated sql data and tested it with mysql console to import generated sql file and query the data. The output of generated sql is like this:

```sql
CREATE TABLE IF NOT EXISTS root (title VARCHAR(255), content TEXT, author VARCHAR(255), date DATETIME, comments INTEGER NOT NULL, FOREIGN KEY (comments) REFERENCES comments(comments), url VARCHAR(255), reference VARCHAR(255), root INTEGER NOT NULL, PRIMARY KEY (root));
CREATE TABLE IF NOT EXISTS comments (email VARCHAR(255), reputation INTEGER, content VARCHAR(255), date DATETIME, comments INTEGER NOT NULL, PRIMARY KEY (comments));
INSERT INTO root (title, content, author, date, comments, url, reference, root) VALUES (Data, Data, Data, Data, Data, Data, Data, Data);
INSERT INTO comments (email, reputation, content, date) VALUES (AAA@gmail.com, 4, A, great, 2017-10-18, 2);  
```

13. Generated graph data and tested it with neo4j console to import generated graph file and query the data. The output of generated cypher is like this:

```cypher
CREATE (n:Node {name: 'John Doe', age: 25, email: 'john.doe@example.com'})
CREATE (n:Node {name: 'Jane Smith', age: 26, email: 'jane.smith@example.com'})
CREATE (n:Node {name: 'Tim Brown', age: 30, email: 'tim.brown@example.com'})
CREATE (n:Node {name: 'Sarah Lee', age: 35, email: 'sarah.lee@example.com'})
CREATE (n:Node {name: 'David Kim', age: 40, email: 'david.kim@example.com'})
```
14. The output of generated xml is like this:

```xml
<?xml version="1.0" encoding="ISO 8859-15"?>
<root>
  <Databases Data Technique="title">content="cillum"

  <Loreum ut do et dolor consecutetur lorem labore consequat, adipsicing cillum dolore aute ipsum magna aliqua ut sit labore et dolore aliqua.

  <Ut labore dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."

  <Dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."

  <Detect aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur."

  <Incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."

  </content></Databases>
</root>
```

**FIGURE B.14:** generated xml data
Bibliography


