Schema Extraction and NoSQL Data Profiling

Master Thesis

written by

Felix Beuster

born Dec 18th, 1990 in Bützow, matriculation number 211207722

University of Rostock
Faculty of Computer Science and Electrical Engineering
Institute of Computer Science

1st Evaluator:	apl. Prof. Dr.-Ing. habil. Meike Klettke
2nd Evaluator:	Prof. Dr.-Ing. habil. Peter Forbrig
Advisor:	apl. Prof. Dr.-Ing. habil. Meike Klettke

August 27th, 2018

Contents

Contents I

List of Algorithms III

List of Figures IV

List of Listings V

List of Tables VI

1 Overview 1

2 Related Work 2
- 2.1 Data Generation . . . 2
  - 2.1.1 Available data generators . . . 2
  - 2.1.2 Combination of existing generators . . . 4
- 2.2 Schema Extraction . . . 4
  - 2.2.1 Structure Identification Graph . . . 5
  - 2.2.2 Extraction process . . . 5
  - 2.2.3 Reduced Structure Identification Graph . . . 6
- 2.3 Schema Evolution . . . 7
- 2.4 Clustering . . . 8
  - 2.4.1 Data based clustering . . . 8
  - 2.4.2 Node based clustering . . . 9

3 Distribution of Structure Identification Graph Extraction 11
- 3.1 Problem definition . . . 11
- 3.2 Requirements . . . 12
- 3.3 Process overview . . . 12
  - 3.3.1 Distribution and merge procedure . . . 12
  - 3.3.2 Time complexity . . . 14
- 3.4 Evaluation . . . 14
  - 3.4.1 Equality . . . 14
  - 3.4.2 Performance . . . 17

4 Schema Evolution Sampling 18
- 4.1 Problem definition . . . 18
- 4.2 Requirements . . . 19
- 4.3 Fixed Rate Sampling . . . 19
  - 4.3.1 Determining the sampling rate . . . 20

4.4 Dynamic Rate Sampling . . . 20
4.4.1 Decrease sampling rate on unchanged schemata . . . 21
4.4.2 Backtracking to find first occurrence of change . . . 22
4.5 Performance evaluation . . . 23

5 Schema clustering **25**
5.1 Problem definition . . . 25
5.2 Requirements . . . 26
5.2.1 Time . . . 26
5.2.2 Reduction of optional elements and finding connected schemata . . . 26
5.3 Construction of schema cluster graph . . . 26
5.3.1 Relationship between two extracted schemata . . . 26
5.3.2 Storing of schemata in nodes . . . 27
5.3.3 Schema saving procedure . . . 27
5.3.4 Example graph result . . . 31
5.4 Clustering in schema cluster graph . . . 32
5.4.1 Evaluation of clustering methods . . . 32
5.4.2 Applied clustering approach . . . 33
5.5 Quality and performance evaluation . . . 36

6 Comparison of outlined techniques **38**
6.1 Document collection `VideoSpace` . . . 38
6.2 Processing results . . . 38
6.2.1 Distributed Schema Extraction . . . 38
6.2.2 Schema Evolution . . . 39
6.2.3 Schema Clustering . . . 39

7 Conclusion and Outlook **41**

A Related Work **42**
A.1 Data generation: json-maker.com Backend . . . 42
A.2 Schema Extraction . . . 43
A.3 Schema Evolution . . . 44

B Performance evaluation **45**
B.1 Distributed schema extraction . . . 45
B.2 Sampled schema evolution extraction . . . 47
B.3 Schema clustering . . . 47

C Comparison of outlined techniques **49**
C.1 Different entity types . . . 49
C.2 Entity changes . . . 51

Statutory Decleration **VII**

Abbreviations **VIII**

Bibliography **X**

List of Algorithms

3.1 Build Hash Set For Tree . . . 12
3.2 Merge Extracted Schemata . . . 13

4.3 Schema Evolution with Dynamic Rate Sampling . . . 21
4.4 Decreasing the sampling rate . . . 22
4.5 Binary search to find first schema change . . . 23

5.6 Schema Clustering Process . . . 28
5.7 Searching for a given schema in the schema cluster graph . . . 28
5.8 Method for adding an edge to the schema cluster graph . . . 29
5.9 DFS method for algorithm 5.8 . . . 29
5.10 Removing direct edges from graph . . . 30
5.11 Removing outliers schemata from schema cluster graph . . . 33
5.12 Finding sources in schema cluster graph . . . 34
5.13 Finding components in schema cluster graph . . . 34
5.14 Calculating indegrees for nodes . . . 35
5.15 Collecting clusters in independent components . . . 36

List of Figures

2.1 Example SG for JSON document listed in Listing 2.2 5
2.2 Example RG for JSON document listed in Listing 2.2 6
2.3 Example schema version graph 7
2.4 Example graph for spectral clustering 10

3.1 Distribution of the schema extraction (concept) 11

4.1 Timeline of documents, different schema versions illustrated by symbols 18
4.2 Timeline from Figure 4.1 with applied sampling, lighter documents are skipped 18
4.3 Timeline from Figure 4.1 with applied fixed rate sampling, lighter documents are skipped 19

5.1 Different extracted schemata organized by relationship 25
5.2 Example graph for schemata listed in Listing 5.1 32
5.3 Schema cluster graph for Listing B.4 36

6.1 Extracted clusters in Darwin tool 40
6.2 Schema cluster graph for collection `VideoSpace` 40

List of Listings

2.1 Sample schema for next.json-generator.com . . . 3
2.2 Sample document for schema from Listing 2.1 . . . 3
2.3 dummy-json `repeat` command for optional elements . . . 4
2.4 dummy-json simplified `optional` command . . . 4
2.5 `faker` command for faker.js integration . . . 4
2.6 JSON Schema representation of `age` node from Figure 2.1 . . . 6
2.7 Example schema evolution operations . . . 8

5.1 Example schemata for illustration of schema graph . . . 31
5.2 Minimum example for clustering . . . 37
5.3 Minimum example for clustering . . . 37

A.1 Definition of helper methods for json-maker.com . . . 42
A.2 Extracted schema from Figure 2.1 . . . 43
A.3 Schema evolution example File 1 . . . 44
A.4 Schema evolution example File 2 . . . 44
A.5 Schema evolution example File 3 . . . 44

B.1 Example document from the people entity . . . 45
B.2 Example document from the video entity . . . 46
B.3 Example document from the product entity . . . 47
B.4 Minimum example for clustering . . . 47

C.1 VideoSpace user entity document . . . 49
C.2 VideoSpace video entity document . . . 50
C.3 Property `comments` added to video entity . . . 51
C.4 Property `responses` removed from video entity . . . 51
C.5 Property `friends` removed from user entity . . . 51

List of Tables

3.1 Distributed schema extraction runtimes for people data set (see B.1) 17
3.2 Distributed schema extraction runtimes (in seconds) for video data set (see B.1) 17

4.1 Sampled schema evolution extraction for document collection `products` 23
4.2 Sampled schema evolution extraction for document collection `products` 24

6.1 Distributed schema extraction runtimes (in seconds) for `VideoSpace` data set 39
6.2 Extracted evolution from `VideoSpace` collection without sampling 39

Chapter 1

Overview

In recent years, data has become more and more important. Big Data is used to improve user experiences like recommending videos to watch, train A.I. systems such as self driving cars, or to get 360° views of customers. It is often in the interest of a company to store data pro-actively. However analyzing these large amounts of data provides some challenges. Schemas of the documents within the same collection can vary a lot. There are a number of ways to uncover these differences and changes. This thesis is aimed to improve some of these processes to make them faster.

In the first part of this thesis, we take a look at schema extraction from a large data set. This is usually not a time critical process, and is done independently from daily operations. However improvements of the overall runtime can be made by using the available data center infrastructure and introduction of a distribution step.

The following second part covers schema evolution. Similar to schema extraction, this process is usually not time critical. Still, runtime improvements can be made to save cost and resources. In this case we look at a sampling approach, so instead of analyzing all documents, only subset of documents will be used. The goal is that the extracted evolution for this subsets is as close as possible to the extracted evolution of the entire subset.

The final part of this thesis will focus on schema clustering. Instead of extracting one universal schema from the data set, the goal is to extract multiple schemas and find the connections between them. Schemas that are similar enough, shall then be combined into one schema, while schemas that differ from each other, shall be kept separate.

As a part of the related work chapter, we also take a look at test data generation, specifically *JavaScript Object Notation* (JSON) document generation. Researchers don't always have access to document collection like companies. Additionally a document generator can offer more control over the schema of the data. This allows testing approaches and algorithms with different kinds of data in a controlled way. We look at existing generation tools and their short comings, which is why a newly developed generator is also part of this thesis.

Chapter 2

Related Work

2.1 Data Generation

To develop fast and efficient algorithms for big and complex data sets, the first challenge is defining and accumulating good test data. Since those sets require hundreds or thousands of files to see meaningful differences between algorithms of different complexity, they cannot be written by hand in a timely manner.

One possible approach is the use of existing data sets, that were made publicly available. Some examples for those data sets can be found in the collection of open data by the U.S. Government [U.S18], a list of American movies from Wikipedia [Rus16], or can be taken from the list of "Awesome JSON Datasets" by Justin Dorfman on GitHub [Dor18].

Those and other data sets have their disadvantages too. The use may be restricted by certain licenses, or they may include personal data and the processing of such data would require consent by each individual. Furthermore, by using existing data sets, the testing of developed algorithms is limited to the structure of those data sets. For instance, the movie list by Justin Dorfman's offers over 100,000 documents, each of which is a single layered object with six properties, that don't vary from document to document. This would make the list a good test data set for algorithms, that are optimized for flat and homogeneous data sets. However if your focus is on data sets, where the schema evolved over time and the data became very heterogeneous, the list would not be a good test data set.

Because of those restrictions, it can be a better approach, to generate a new test data set, that is representative for real world use cases, and is as simple or complex structured as it is required for the research project.

2.1.1 Available data generators

There are many generators for data sets available. Since this thesis is working with JSON data, I will focus on JSON data generators here.

Those generators are available as online tools, like json-generator.com [Oma16], or as libraries and donwloads, that can be used as part of other applications, like dummy-json [Swe17]. They share the concept of using a user-defined schema, from which the JSON documents will be generated. However, the syntax and available helper functions to generate data items, such as names or numbers, vary between generators. One example schema can be found in Listing 2.1.

Listing 2.1: Sample schema for next.json-generator.com

```
[
  {
    'repeat(5, 10)': {
      _id: '{{objectId()}}',
      age: '{{integer(20, 40)}}',
      name: {
        first: '{{firstName()}}',
        last: '{{surname()}}'
      },
      address: '{{integer(100, 999)}} {{street()}}, {{city()}}, {{state
          ()}}, {{integer(100, 10000)}}',
      friends: [
        {
          'repeat(0,3)': {
            id: '{{index()}}',
            name: '{{firstName()}} {{surname()}}'
          }
        }
      ]
    }
  }
]
```

This schema generates between five and ten documents in an array, one of them cam be found in Listing 2.2. It contains an object ID, age of a person, the name as an object, address information, as well as a list of friends, represented with an array of objects.

Listing 2.2: Sample document for schema from Listing 2.1

```
{
  "_id": "5b01ecb20889f254af2c31ce",
  "age": 38,
  "name": {
    "first": "Roy",
    "last": "Buckley"
  },
  "address": "546 Newton Street, Santel, Georgia, 3191",
  "friends": [
    {
      "id": 0,
      "name": "Craft Baird"
    },
    {
      "id": 1,
      "name": "Lloyd Bush"
    }
  ]
}
```

The types of data that those generators provide are often limited and centered around person related date such as names and address information, dates and the simple data types boolean, integer, float and strings. For many applications, more specific data is needed, which is were tools like faker.js [Mar18] are used. It provides more data sets than the mentioned generators, such as financial data, or image data. Furthermore, some data can be localized to be country specific, and the data sets are larger as well. While dummy-json contains only 100 first names [Swe17], faker.js contains over 3,000 first names just for the English language [Mar18].

2.1.2 Combination of existing generators

In order to test different JSON analyzing methods, such as observing the schema evolution or finding clusters in extracted schemas, a diverse data set is needed. One way to achieve this is having optional properties in the JSON object. However optional properties is not a feature, that is supported by available online JSON generators. For this reason I created json-maker.com, that is available for free [Beu18].

The new tool uses the previously mentioned library dummy-json, that can create optional elements by using a special configuration of the `repeat` command, as seen in Listing 2.3. To simplify the usage for users, I added the `optional` command that can now be used as seen in Listing 2.4.

Listing 2.3: dummy-json `repeat` command for optional elements

```
{{#repeat 0 1 comma=false}}
  "property_name" : "property_value",
{{/repeat}}
```

Listing 2.4: dummy-json simplified `optional` command

```
{{#optional}}
  "property_name" : "property_value",
{{/optional}}
```

To provide more data options as described above, I fully integrated faker.js. Data can be accessed in command similar to the original dummy-json generator. Users will use the added `faker` command with the data category and data item name as argument, illustrated in Listing 2.5.

Listing 2.5: `faker` command for faker.js integration

```
{
  "name" : "{{faker 'category.entry'}}"
}
```

The definition of the helper methods for the optional command and the integration of faker.js can be found in Listing A.1.

2.2 Schema Extraction

Approaches of existing *Extensible Markup Language* (XML) schema extraction tools [MLN00], have been extended and adapted to work with JSON document collections as well. The extraction process from [KSSR15] will be the base for this thesis and shall be explained in this section.

In general, schema extraction works on a selection of documents from a NoSQL database. This selection will be called “data set” from now on for this section. While it is possible to work with all documents of the collection, the data set might only include a subset of documents, based on a version number, a date or specific properties, for example a store location [KSSR15].

2.2.1 Structure Identification Graph

The information gathered during the extraction is stored in a graph structure $SG = (V, E)$, called *Structure Identification Graph* (SG). In this graph, V contains all properties from the documents, stored as nodes $(nodeLabel, nodeIDList)$. A node contains information about the property name and its path in the document, both stored in the *nodeLabel*. The *nodeIDList* is a list of documents IDs, that contain this property. The set $E \subseteq V \times V$ contains all connections between the nodes, with an edge $(v_1, v_2, nodeIDList)$ representing "v_1 is a parent node of v_2". *nodeIDList* again contains a list of document IDs, in which this connection occurs [KSSR15]. An example for a SG can be seen in Figure 2.1

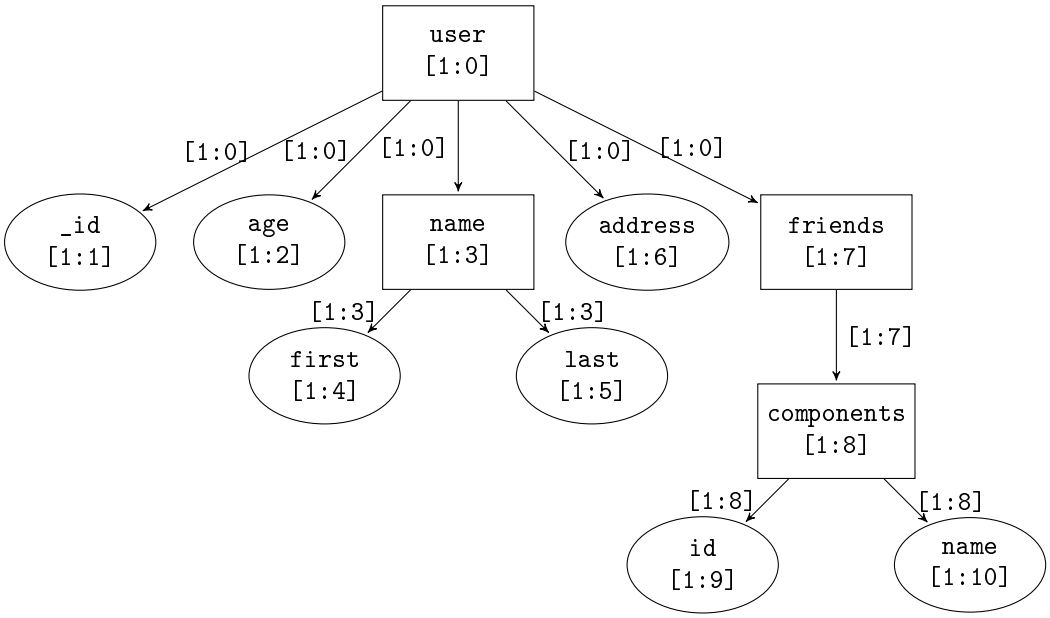

Figure 2.1: Example SG for JSON document listed in Listing 2.2

2.2.2 Extraction process

Construction of SG from documents

For the construction of the graph, the JSON documents are traversed in preorder. This ensures, that parent nodes are processed before their child nodes.

At each node in the document, a new node is either added to the SG with the current path, name, data type and document identifier as information, or the existing node is being updated by adding the current document identifier to the node.

Edges from the current node to its parent node are added in a similar way. If it already exists in the graph, the stored information is updated accordingly. If it doesn't exist yet, it is added with the current document identifier as information.

Extraction of schema from SG

To retrieve the schema from the SG, the tree is traversed in postorder, as the information of the child nodes is needed in the parent node. From each node a textual representation is calculated, following the guidelines of [Sch18]. The information to define name and type in the representation is stored in the node directly. An example for the `age` node from Figure 2.1 can be found in Listing 2.6

Listing 2.6: JSON Schema representation of `age` node from Figure 2.1

```
"age" : {
  "type" : "number"
}
```

Determining if the node is required or optional, requires comparing the length of the document id list of the parent node, with the length of the document id list in the edge connecting the parent with the current node. If those two list are equally long, the node is required, and optional otherwise. The complete extracted schema from Figure 2.1 can be found in Listing A.2.

2.2.3 Reduced Structure Identification Graph

To reduce runtime and memory consumption of the extraction process, [KSSR15] proposes a variant of the SG, the *Reduced Structure Identification Graph* (RG).

This variant removes the *nodeIDList* from both, the nodes and the edges. Instead the node has a simple counter that represents, how often a property is used in the data set. Differentiating between optional and required is still possible, by comparing the counter of a child node to the counter of its parent. An example for a RG can be seen in figure 2.2

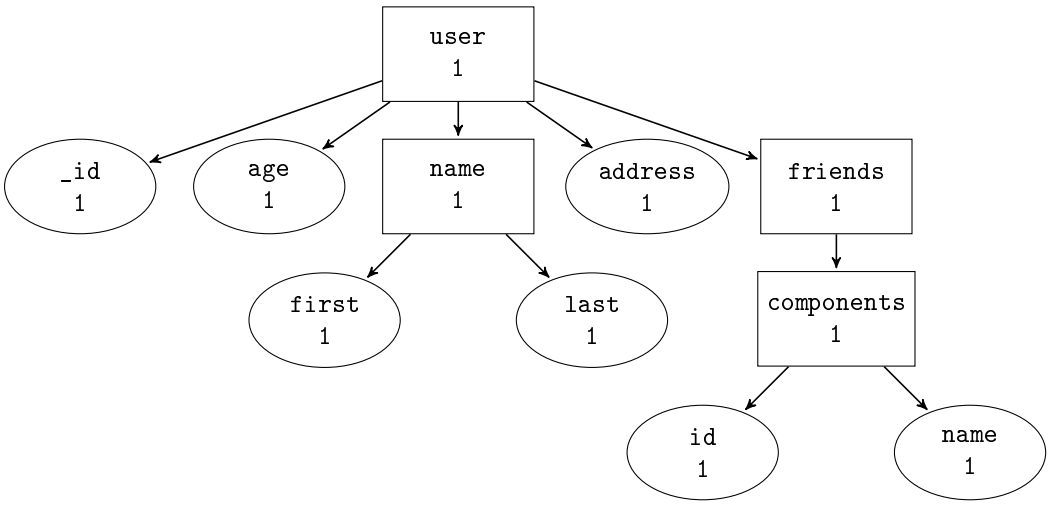

Figure 2.2: Example RG for JSON document listed in Listing 2.2

2.3 Schema Evolution

In a set of documents, that are ordered by time of creation, we can track the changes to the schema over time. Extracting this evolution as described in [KAS+17] is a three step process, and results in a list of operations. Each operation can be applied to the previous schema to get the current schema version at certain time.

Extraction of the schema version graph Each document in the overall document set represents a certain entity, eg. users, products, events, etc. In this first step, a schema version graph is extracted for each entity. For the extraction, a process similar to the extraction in section 2.2.2 can be used. The construction of the schema version graph can be done like the construction of a RG in 2.2.3, however instead of saving the document identifier in the nodes, we now keep track of the timestamps of the documents. You can find an example schema version graph in Figure 2.3, based on the documents from section A.3.

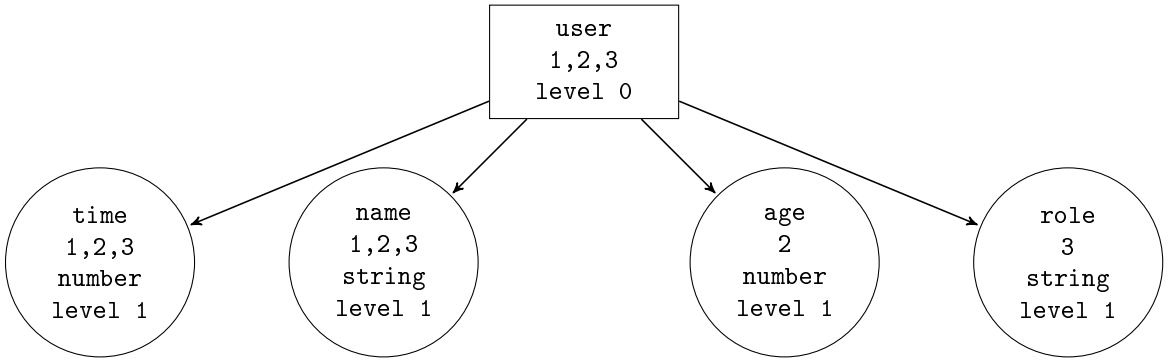

Figure 2.3: Example schema version graph

Extract schema evolution operations The second step analyses the schema version graphs and finds the operations to evolve one schema version to the next. As presented in [SKS13], the basic operations are

- **add** - Adding a property to an entity
- **delete** - Removing a property from an entity
- **rename** - Renaming a property in an entity

Additionally, [KAS+17] adds two more operation for inter-entity operations:

- **copy** - Copying a property from one entity to another
- **move** - Moving a property from one entity to another

The operations can be found by iterating the timestamps and comparing the timestamp lists in the nodes. For example if the running timestamp cannot be found in the node any longer, this can be evaluated as a **delete** operation. Listing 2.7 shows the three schema evolution operations derived from the schema version graph in Figure 2.3.

Listing 2.7: Example schema evolution operations

```
1 add number user.age
2 delete number user.age
3 add string user.role
```

Resolution of ambiguities Since this evolution extraction process only looks at the structure of the data, but not the data and meaning of it itself, user interaction is required in the third step. If in the previous step, alternative operations are generate, eg. adding a property vs. copying the property from another entity, user input is now used to decide on the right operation. This operation is then included in the list of operations for the evolution.

2.4 Clustering

There are different ways to find clusters in graphs and in data. In this section, a few of them will be presented.

2.4.1 Data based clustering

k-means

As introduced in [M+67], k-means is a process to divide a data set into k sets S_i. These sets are centered around a sample s_i, which are taken from the entire data set.

In a first step, those k samples s_i need to be defined. This is done by random selection of data points from the data set. Each selection is independent from the previous one, with the exception that the same data point cannot be selected twice.

The second step is the assignment step, which partitions the remaining data set based on the selected samples. For each remaining data point p, the distance between it and all the samples s_i is calculated with a distance function $dist$. The distance function needs to be defined specifically for the data set. It should define how similar two data points are. The lower the value is, the more similar the two data points are, with $dist = 0$ implying that two data points are the same.
The data point p is then assigned to the set S_i, where $dist(s_i, p)$ has the minimal value.

The k-means problem can be solved in $\mathcal{O}(n^{dk+1})$ time for a fixed number of dimensions d and number of clusters k [IKI94]. However, if either k oder d is unknown, the problems becomes NP-hard [MNV09][ADHP09]. In those cases, a heuristic algorithm is needed, such as Lloyd's algorithm, which has a runtime of $\mathcal{O}(nkdi)$, with n, k and d as above, and i being the number of iterations [HW79]. In addition to the assignment step mentioned above, Lloyd's algorithm adds an update step, that calculates new centers for the sets. The assignment step and update step are then repeated i-times, or until no data points are being moved from one set to the other.

k-medoids

Similar to k-means, k-medoids also divides a data set into k sets S_i, based on a center point s_i in the set. In this case the medoids are not chosen in a random manner, but in a way that they would be the most central points in the cluster.

In the second step, all remaining data points are again assigned to the medoid with the smallest distance. After all data points are assigned, the medoid of each set as adjustes, such that the sum of the distances

is minimized [FHT01]. Assignment and update step are repeated until sums of the distances do not decrease any further.

Hierarchical Clustering

In addition to finding distinct clusters in data, this method tries to find a hierarchy of those clusters. Similar to the two previous methods, a distance function *dist* is needed to express the similarity of two data points. This function is used to decide on splitting or merging the clusters. There are two general approaches to hierarchical clustering [RM05]:

Agglomerative This approach considers each data point to be an individual cluster in the beginning. Each iteration of this bottom up approach then merges a pair of two clusters based on *dist* into one cluster. This is repeated until one cluster containing all data points is formed in the end. With a time complexity of $\mathcal{O}(n^3)$ and a space complexity of $\mathcal{O}(n^2)$ it is not a cheap process.

Divisive This approach is the opposite Agglomerative Hierarchical Clustering and a top down process. Initially all data points are within one cluster. Each iteration then splits the cluster based on *dist* into two halves, until each data point is within its own cluster. However, since there are 2^n ways to split a set of n elements, the time complexity of this process is $\mathcal{O}(2^n)$. Therefore heuristics are used to determine the split of a cluster. For example the DIANA algorithm splits on the data point that has the maximum average *dist* to all other points in the cluster [KR09].

2.4.2 Node based clustering

Min-cut

In a graph G, the min-cut is the minimum sum of edge weights or edge costs of edges, that after removal of those edges from G, the graph will be partitioned into two independent components [W+01]. While not specifically a clustering method, it can be useful when a network needs to be divided into separate groups at minimal cost, or the maximum flow of a flow network needs to be calculated [FF56].

Spectral Clustering

Spectral clustering[VL07][SM00] is a commonly used approach to find clusters within data sets. In a first step, the data points are converted into nodes in a similarity graph $G = (V, E)$. Edges are added to the graph based on the similarity of the data points.

As the next step, the Laplacian matrix L needs to be calculated, which is defined as

$$L = D - W$$

where D is the degree matrix and W is the weight matrix of the graph G. in an unweighted graph, W is replaced with the adjacency matrix A. Given a graph as in Figure 2.4, the resulting matrices would be

$$D = \begin{pmatrix} 6 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}, \; W = \begin{pmatrix} 0 & 1 & 5 & 0 \\ 1 & 0 & 1 & 2 \\ 5 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix}$$

$$L = D - W = \begin{pmatrix} 6 & -1 & -5 & 0 \\ -1 & 4 & -1 & -2 \\ -5 & -1 & 6 & 0 \\ 0 & -2 & 0 & 2 \end{pmatrix}$$

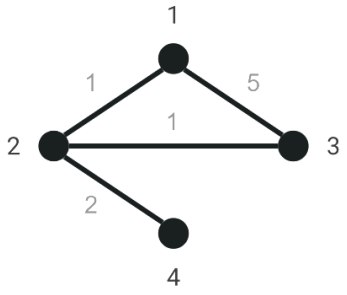

Figure 2.4: Example graph for spectral clustering

Step 3 calculates the eigenvectors and eigenvalues of the Laplacian matrix, which in this case are the eigenvectors

$$\lambda_1 = 0 \qquad \lambda_2 = \frac{\sqrt{17}}{2} + \frac{7}{2} \qquad \lambda_3 = 11 \qquad \lambda_4 = -\frac{\sqrt{17}}{2} + \frac{7}{2}$$

with corresponding eigenvectors

$$x_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \qquad x_2 = \begin{pmatrix} -\frac{1}{8} + \frac{\sqrt{17}}{8} \\ -\frac{\sqrt{17}}{4} - \frac{3}{4} \\ -\frac{1}{8} + \frac{\sqrt{17}}{8} \\ 1 \end{pmatrix} \qquad x_3 = \begin{pmatrix} -1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \qquad x_4 = \begin{pmatrix} -\frac{\sqrt{17}}{8} - \frac{1}{8} \\ -\frac{3}{4} + \frac{\sqrt{17}}{4} \\ -\frac{\sqrt{17}}{8} - \frac{1}{8} \\ 1 \end{pmatrix}$$

From those values the second smallest eigenvalue and corresponding eigenvector, which are λ_4 and x_4 in this example.

The last step number 4 of the spectral clustering splits the eigenvector into two groups at 0 or a median value, thus creating to clusters. For the example the extracted clusters would be

$$\{1, 3\} \quad \{2, 4\}$$

To extract a total of k clusters from the graph, the bi-partitioning can be repeated recursively on the subsets created in this step, or multiple eigenvectors are used to place the data points into a k-dimensional space and a k-means clustering algorithm is applied.

Chapter 3

Distribution of Structure Identification Graph Extraction

3.1 Problem definition

The schema extraction from large data sets of hundreds of thousands of documents creates some challenges. Current approaches run on single machines, since temporary structures, such as the SG oder RG, and data are held in main memory, until the final schema and statistics can be obtained [KSSR15].

To reduce the time the extraction needs and also reduce the main memory requirements, a distributed extraction is desirable. It can also reduce the amount of data that needs to be transferred between data processing locations. Instead of sending a complete data set, only the partial result would need to be transferred for combination with other partial results.

While the data set can easily be split into smaller parts to be processed on different machines, a method to merge the results of each extraction is needed, and will be discussed in this chapter.

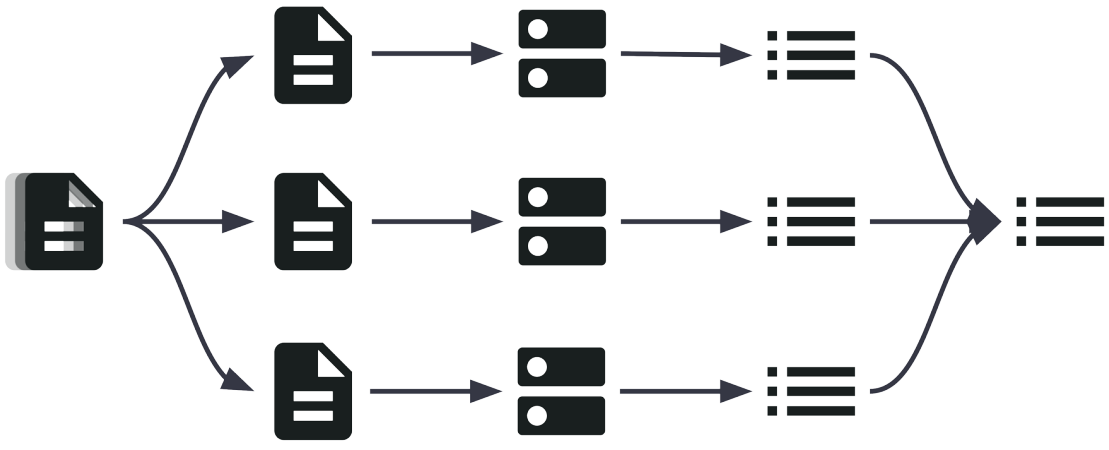

Figure 3.1: Distribution of the schema extraction (concept)

3.2 Requirements

Before designing a process to distribute the processing, it is important to point out the requirements for the result of this process. In this case, the requirements are as follows:

Equality

The result of a distributed schema extraction process should not be different from the result of a non-distributed schema extraction process.

Time

Since the distributed process uses parallel computing, it should of course have a lower overall runtime compared to a non-distributed schema extraction. The merge process of the distributed schema extraction needs to have a low time complexity, so that the schema extraction runtime gains are not consumed by the runtime of the merge process. A linear time complexity is desirable.

3.3 Process overview

3.3.1 Distribution and merge procedure

This section describes the process for distributing the data set, extracting of the schemata, and merging of the extracted schemata into a single schema. The process has four steps:

1. As a first step, the data set will be split into k disjoint partitions, that can be processed on individual computing nodes. The extraction of the schemata shall then be done in a process as described in section 2.2. The result should be a list of k extracted schema trees, represented by RGs. These graphs are now going to be processed further one a single computing node.

2. The second step picks the **master schema**. All other extracted schemata will be merged into the master schema tree. This selection can be done at random. As outlined in step four, by the end of the process we will have walked through all schema trees once, therefor selecting a specific tree at this point is not making a difference.

3. In the third step, we calculate a hash set HS for the master schema tree, where each node stores its path from the root. This can be done by using a *Breadth-first search* (BFS) and allows a faster lookup of nodes. This step is also described in algorithm 3.1.

Algorithm 3.1 Build Hash Set For Tree

```
1:  procedure BuildHashSet(SchemaTree t)
2:      Q := new QUEUE;
3:      HS := new HASH SET;
4:      HS.put(t.root.path);
5:      Q.add(t.root);
6:
7:      while Q ≠ ∅ do
8:          n := RemoveFirst(Q);
9:          HS.put(n.path);
10:         for all c ∈ SortedChildren(n) do
11:             Q.add(c);
        return HS;
```

4. The fourth step is an iteration over the remaining $k-1$ schema trees. For each schema tree t we walk through its nodes by using a BFS with a queue, that is initially populated with the root of the schema tree t. For each node n in the queue, we add its children to the queue. We then check if the path of the node n is listed in the hash set HS.

   In case the path of n can be found in HS, and we can update n in the tree *master* with the data from n in the tree t.

   If n is not listed in HS, we add the node n to the tree *master*, so that the parent of n in *master* is the same as the parent of n in the tree t. We also add the path of n to the hash set HS for reference in the following iterations.

After those four steps, the result is a combined schema tree, stored in *master*. The complete pseudo code for this process is listed in algorithm 3.2.

Algorithm 3.2 Merge Extracted Schemata

```
1:  procedure MergeSchemas(SchemaTree[] schema_list)
2:      master := getRandomTree(schema_list)
3:      HS := BuildHashSet(master);
4:
5:      for SchemaTree t in schema_list do
6:          Q := new QUEUE;
7:          Q.add(t.root);
8:
9:          while Q ≠ ∅ do
10:             n := RemoveFirst(Q);
11:             for all c ∈ Children(n) do
12:                 Q.add(c);
13:             if HS.contains(n.path) then
14:                 master.updateNode(n);
15:             else
16:                 master.insertNode(n, n.path);
17:                 HS.put(n.path);
18:
```

Updating an existing node

Updating a node that already exists in the master schema tree requires further explanation about what and how they are updated.

A node in the master schema tree contains information about the number of occurrences in the data set, to determine if it is required or not[KSSR15]. When the node is updated, the number of occurrences of the current node n in the schema tree t is added to the number of occurrences of the node n in *master*.

Nodes also store information about the data type of the field that they represent from the document. If a node n is encountered, that already exists in the tree *master* with different type, we update the occurrence information as above, and add the type of n in the tree t, to the list of types of n in the tree *master*.

3.3.2 Time complexity

In the process above we extracted k schemata, with each of them having n nodes. Based on that, we can work out the time complexity of the process.

As shown in [CLRS09], BFS has a time complexity of

$$\mathcal{O}(|V| + |E|)$$

Since a tree with n nodes always has $n - 1$ edges, walking through the nodes of a schema tree t with a BFS approach has a time complexity of

$$\mathcal{O}(n)$$

At each node of the tree t, the node either needs to be inserted into the tree *master*, or the corresponding node needs to be updated. This update can be done in constant time $\mathcal{O}(1)$. The insert of a node into the tree *master* and the hash set HS can also be done in constant time $\mathcal{O}(1)$.

As a result the time requirement is

$$\mathcal{O}(2 \cdot n)$$

which equals to the complexity of

$$\mathcal{O}(n)$$

as the time complexity to walk each extracted schema tree and merge it into the master schema tree.

Since we have extracted k different schemata, from which we picked one master schema tree, the time complexity for merging is

$$\mathcal{O}((k - 1) \cdot n)$$

And if we include the build of the master schema tree hash table, the time complexity ends up begin

$$\mathcal{O}(k \cdot n)$$

Thus the merge process achieves the desired linear time complexity from 3.2 for a defined number of partitions.

3.4 Evaluation

3.4.1 Equality

Two trees are equal if the root nodes of each tree are equal, and when they have an equal set of children, therefor having identical subtrees. Thus tree equality can be shown recursively.

In case for schema trees, two nodes are equal when

1. node names are identical
2. node occurrence counts are identical
3. node data type lists are identical
4. node child node lists are identical

The fourth point of the list will include be the recursive check.

Root node equality

With a variation of [KSSR15] by adding the data type list, a node for a RG is defined as a tuple such as

$$(nodeLabel, occurrenceCount, dataTypeList)$$

For the root node in the schema tree of a non-distributed schema extraction, the node label will be the path to the node (for instance represented by /), followed the chosen entity name. The occurrence count matches the total number of documents, and the data type list contains `object` as the only entry.

Since the schema extraction method itself is not changed in the distributed schema extraction, each of the $k$partitions' schema tree's root node will have the same path and entity name as root label. The same applies to the data type list.

$$nodeLabel_{original} = nodeLabel_{root_1} = ... = nodeLabel_{root_k}$$
$$dataTypeList_{original} = dataTypeList_{root_1} = ... = dataTypeList_{root_k}$$

As the labels of the root nodes are identical, inserting the partition trees into a chosen master tree will not create new nodes and merge them into a single node. As the data types lists all contain the same type, no additional types will be added to the list.
Since

$$nodeLabel_{root_1} = ... = nodeLabel_{root_k} = nodeLabel_{merge}$$
$$dataTypeList_{root_1} = ... = dataTypeList_{root_k} = dataTypeList_{merge}$$

it follows

$$nodeLabel_{original} = nodeLabel_{merge}$$
$$dataTypeList_{original} = dataTypeList_{merge}$$

The occurrence count in the root nodes of the partition trees matches the number of documents in those partitions. For a partition p, the number of documents can be described as

$$|p|$$

As document collection was split into k disjoint partitions, it can be assumed

$$occurenceCount_{original} = |documentCollection| = \sum_{i=1}^{k} |p_i| = \sum_{i=1}^{k} occurrenceCount_{root_i}$$

Since the merge process is adding the occurrence counts of all partitions' root nodes, it follows

$$\sum_{i=1}^{k} occurrenceCount_{root_i} = occurenceCount_{merge}$$

and thus

$$occurenceCount_{original} = occurenceCount_{merge}$$

As the three elements of the root nodes of the original schema tree and the merged schema tree are identical, they can be considered equal.

Child nodes equality

For the schema tree *original* of the non-distributed schema extraction and the merged schema tree *merge* to be equal, the child nodes of the root nodes also need to be equal.

Child nodes can be selected from the set of edges E into a set C by

$$C = \{v | (root, v) \in E\}$$

When a schema tree t is merged into the *merge* schema tree, already existing child nodes of root will be updated and no changes to the graph are made. However if a child node c is added to the root node, a corresponding edge is added to E_{merge}, which can be expressed as

$$\{c\} \cup \{v | (root_{merge}, v) \in E_{merge}\}$$

thus after comparing and potentially adding all child nodes from the root t to the root of *merge*, the set is

$$C_{merge} = \bigcup_{i=0}^{k} \{C_i\}$$

As the document collection is split into k partitions, each property of the root node in the JSON schema can be found in at least one partition, and it's corresponding node and edge are added to the schema tree i. Because of that, the set of edges from a root node in *original* can be split such as

$$C_{original} = \bigcup_{i=0}^{k} \{C_i\}$$

Child nodes are merged in the same way as root nodes. Thus, by using the child nodes as root nodes for a subtree, equality can be shown recursively. As a result of that

$$C_{original} = C_{merge}$$

3.4.2 Performance

Testing environment

The testing prototype implementations for this thesis are integrated into the *Darwin* middleware [SMKS17], to make a future complete integration more easily. All performance comparison tests are run on a local Windows 10 machine equipped with an AMD Ryzen 1700 CPU (8 cores/16 threads) @ 3.0GHz, 32GB DDR4 of RAM @ 3000MHz, and SSD storage.

Data and results

Since the regular schema extraction is a single node process and Java typically runs in a single thread, multiple nodes for the distributed extraction are simulated by using multiple threads that can run in parallel. The comparison test will run with 1, 2, 4, 8 and 16 threads.

Threads	1	2	4	8	16
Avg. time per thread	3.906	1.022	0.208	0.088	0.074
Merge time	0.002	0.002	0.002	0.002	0.001
Total time	3.944	1.090	0.270	0.142	0.131

Table 3.1: Distributed schema extraction runtimes for people data set (see B.1)

As it can be seen in the runtime data in Table 3.1, distributing the document achieves the desired effect of reducing the overall time needed for the extraction process. The merge time appears to be constant, with the chosen precision of milliseconds, for this type of document schema.

Threads	1	2	4	8	16
Avg. time per thread	15.257	5.184	1.009	0.430	0.265
Merge time	0.004	0.004	0.004	0.004	0.003
Total time	15.300	5.410	1.129	0.530	0.365

Table 3.2: Distributed schema extraction runtimes (in seconds) for video data set (see B.1)

Using a document collection with a more complex schema as in Table 3.2 shows roughly the same factors of time improvement for the distributed schema extraction. Merging those schemata together again appears to be a constant time factor, though due to the increased schema complexity, the merging takes longer.

Chapter 4

Schema Evolution Sampling

4.1 Problem definition

The schema of JSON documents in a document collection can evolve over time. In some cases evolutions can happen on a continuous basis, where updates are applied regularly. In other cases, schema evolutions occur in burst, for example when a new software version is released.

Figure 4.1: Timeline of documents, different schema versions illustrated by symbols

In case of rarely changing schemata, it is not necessary to process all documents to construct the history of changes. This chapter will explore a fixed and a dynamic sampling approach to extract the schema changes only from documents that have changes, but skip those documents that don't have any changes. This illustrated in Figure 4.2.

Figure 4.2: Timeline from Figure 4.1 with applied sampling, lighter documents are skipped

A different use case for the using sampling during the evolution extraction is generating previews of the entire evolution. When the entire set of documents has some filtering or pre-processing applied, we can generate a preview to evaluate a quick preview of the evolution and tweak the configuration without long waiting times. This also saves valuable time in a cloud environment.

4.2 Requirements

Time

The original time complexity of the evolution extraction process described in [KAS+17] is $\mathcal{O}(n \log n)$. In case of the fixed rate sampling, time complexity should be completely unaffected, but the overall runtime should decrease.

For the dynamic rate sampling, a backtracking mechanism will be added. This mechanism should not exceed the time complexity of the original extraction process. As with the fixed rate sampling, overall runtime should decrease.

Accuracy

When only a subset of the original set of documents is used for the evolution extraction process, it is to be expected that a number of the documents, that are not in subset, contain changes in the schema. The goal is to achieve a high accuracy in the sampling process, so the number of missed changes is kept at a minimum.

Other Prerequisites

In order to extract a meaningful evolution from a document collection, it needs to be sorted in the order of the time of creation of the documents. This can be done via a timestamp attribute in the document. To keep the accuracy of the sampling high, the documents still need to be sorted when sampling is applied.

4.3 Fixed Rate Sampling

The first approach to reduce overall runtime to extract the schema evolution, is using a fixed rate sampling, with a rate of $k\%$. As a result, instead of using the entire set of n documents, only every $\frac{100}{k}$-th document will be used in the extraction process.

Using a fixed rate, we can achieve runtime improvements, that are independent from diversity of the schemata in the set of documents. Since only $k\%$ of documents are processed, and the overall evolution extraction process remains unchanged, the average runtime should improve to $k\%$ of the original runtime.

Figure 4.3: Timeline from Figure 4.1 with applied fixed rate sampling, lighter documents are skipped

The disadvantage of the fixed sampling rate is illustrated in Figure 4.3. The example has an applied sampling rate of 20%, meaning every 5th document will be processed. Documents with the second schema (illustrated by the plus symbol) will not be processed, therefor the change of the schema at that point will not be noticed and not reflected in the extracted evolution.
Depending on the number of schema changes and the sampling rate, the number of missed schema changes can vary. This results in a reduced accuracy of the process. The approach in the next section will improve accuracy.

4.3.1 Determining the sampling rate

For each document collection, there is an ideal sampling rate, that ensures that every schema change is found, while minimizing the number of documents that need to be processed.

One way to obtain this value, is to find the schema version with minimal use s in the document collection $DOCS$. The number of documents using this schema version than becomes the sampling rate $k\%$.

$$k\% = \frac{100}{|\{d|d \in DOCS \wedge Schema(d) = s\}|}\%$$

Since there are at least k documents using each schema, and every $k-th$ document is processed, it is guaranteed that at least one document of each schema version is processed.
However, if a schema version is only used by very few documents, the sampling rate will become higher and more documents are being processed again.

A second approach of finding the ideal sampling rate can be used, if the different schema versions are evenly or almost evenly distributed in the document collection. In case we have an even distribution, the sampling rate $k\%$ can be calculated by

$$k\% = \frac{|SchemaVersions|}{100}\%$$

Using this sampling rate, the document collection is divided into k partitions, and only the first document in each partition is processed.
In case schema versions are almost evenly distributed, the number of schema versions can still be used for the sampling rate, but needs to be increased to ensure coverage of all schema version. Assuming k schema versions in the document collection $DOCS$, P_{max} is the set of documents with a schema version with the most documents and P_{min} is the set with the fewest documents, the needed number k_n to ensure a coverage of all schema changes can be calculated by

$$k_n\% = \left\lceil \frac{|DOCS|}{|P_{max}| - |P_{min}|} \right\rceil \%$$

This method works as long as the difference of $|P_{min}|$ and $|P_{max}|$ is smaller than half the number of documents in the collection. However at that point the distribution of schema version would also not be consider "almost even" any longer.

Using the described methods in an automated system would require a processing of the complete document collection to find the ideal sampling rate, thus they can only be used, if they are retrieved from a user, that has knowledge of the document collection.

4.4 Dynamic Rate Sampling

Instead of using a fixed rate to sample the documents, this approach is more dynamic and adjusts the rate based on whether changes in the schema are found or not.

For this process, we cannot use an intuitive probability approach. Assuming a set of n documents with k changes in the schema, the probability of finding a schema change in any read document would be $\frac{k}{n}$. Every document that is read and doesn't contain a change, increases the probability of finding a change in the next document to $\frac{k}{n-1}$. Based on this observation, the dynamic sampling should start with a sampling rate of any size and increase the rate for every document without a change. However

this increases the number of documents read, until all documents are used again, to extract the schema evolution. This is the opposite of what the sampling wants to achieve, as documents without any changes should be skipped in the extraction.

Instead the sampling rate needs to be decreased on documents that don't have any schema changes. The general process looks as follows:

Algorithm 4.3 Schema Evolution with Dynamic Rate Sampling

```
1:  procedure DynamicEvolution(Document[] document_set)
2:      s := initiateSchemVersionGraph();
3:
4:      for Document d in document_set do
5:          if !inSample(d) then
6:              nextDocument();
7:
8:          else
9:              extractSchemaToVersionGraph(d, s);
10:             t := getTimestamp(d);
11:             e := findEvolutionStep(s, t);
12:
13:             if empty(e) then
14:                 decreaseSamplingRate();
15:
16:             else
17:                 resetSamplingRate();
18:                 backtrackChange();
```

For each processed document, the evolution step needs to be extracted. If no evolution step can be extracted, the schema is unchanged and the sampling rate can be decreased. If there is an evolution step, the sampling rate is set back to the default value the backtracking for the change begins. Both parts are now explained further.

4.4.1 Decrease sampling rate on unchanged schemata

The sampling rate for this process starts with a value of 100%, meaning all documents are going to be processed. The decrease of the sampling rate upon a no-change in the documents can be done immediately or after a certain amount of documents have seen no change, to build confidence that the schema is consistent over a period time, for example by requiring a schema stays unchanged for 0.1% of the document count in the collection. This **sensitivity factor** should be kept low, otherwise it would work against the idea of sampling. It should also be exposed as a setting in implemented application, so that it can be adjusted to the needs for a specific document collection. For example in collection with 1,000 documents, the above sensitivity factor equals to one document, while in a collection with 100,000,000 documents, that factor would requires 100,000 documents to have the same schema, before the sampling would start.

Lowering the sampling rate can be done by setting it to half of the current value. For example the sampling rate of 100% would be lowered to 50%. In that case, instead of processing every document, every second document will be processed. This process of decreasing can be done until the end of the

document collection is reached, with the **minimal sampling rate** reaching

$$\frac{100}{2^{\log_2(|documents|)}}\% = \frac{100}{|documents|}\%$$

a the document collection can be cut in half

$$\log_2(|documents|)$$

times. If rounded up to the nearest integer value, this also represents the number of documents that is processed, assuming a sensitivity factor that equals to one document, and the sampling rate is halved each time.

Algorithm 4.4 Decreasing the sampling rate

```
1: procedure DecreaseSamplingRate
2:     current := getCurrentSamplingRate();
3:     minimum := getMinSamplingRate();
4:     decrease := getSamplingRateDcreaseFactor();
5:
6:     if current / decrease >= minimum then setSamplingRate(current / decrease);
7:
```

In an implementation, the **minimal sampling rate** and **sampling rate decrease factor** should also be exposed as settings to the user, so that the process can be fine tuned to a specific document collection, or to ensure that a certain number of documents will be processed.

4.4.2 Backtracking to find first occurrence of change

In case there is a schema change between the currently processed document *current* and the last processed document *previous*, the sampling rate set back to its original 100%. More importantly, a backtracking process is started, to find the first document after *previous* with a schema change. This change can be a change into the schema that was found in *current* or a different schema change. Backtracking the timeline of documents to the first occurrence of a change after *previous* is done via binary search (see [CLRS09]), as outlined in algorithm 4.5.
When the middle schema between the left and right boundary of the search window has no change to the left one, the mid schema becomes the new left boundary. In case we do find an evolution step and that mid schema is different from the left schema, the right boundary is moved to the middle. This process narrows the search window down until the left boundary is the last schema in the timeline, that has the same schema as *previous*. The right boundary is then the first different schema, which can be equal to *current* or be another schema.

The backtracking adds a complexity of

$$\mathcal{O}(\log(n))$$

to the dynamic rate sampling process. However since this is lower than $\mathcal{O}(n)$, and less than the entire document collection is processed, it the dynamic rate sampling is still a performance improvement compared to the full evolution extraction.

While the fixed rate sampling is able to find persistent schema changes, the dynamic rate sampling with backtracking can also find the exact timestamp of the change, and also find different changes that occurred between *previous* and *current*. However it should be noted, that it is still possible, that this

Algorithm 4.5 Binary search to find first schema change

```
1:  procedure FindFirstChange(Schema previous, Schema current)
2:      left := previous;
3:      right := current;
4:
5:      while left.getTimestamp() < right.getTimestamp() do
6:          mid := getCenterSchemaBetween(left, right);
7:          step := findEvolutionStep(left, mid);
8:
9:          if step.isEmpty() then                          ▷ mid schema equals left schema
10:             left := mid;
11:
12:         else                                            ▷ mid schema is different from left schema
13:             if right.isNextDocumentAfter(left) then return right;
14:             else
15:                 right := mid;
          return right;
16:
```

process missed non-persistent changes, if they are between the $left$ and mid schema. For example, if after the $left$ schema a property is added, but removed before the mid schema, the change will be unnoticed.

In alternative to binary search backtracking, a linear search could be used, which would increase the backtracking complexity from $\mathcal{O}(\log(n))$ to $\mathcal{O}(n)$. Another option could be to not use a backtracking method, but to reset the sampling rate to 100% and continue the processing at *previous*.

4.5 Performance evaluation

The performance evaluation is done on the product data set, which can be found in section B.2.

Fixed rate	Time	Accuracy	Timestamp		
			1st change	2nd change	3rd change
100%	0.245s	100%	2324	2433	3938
50%	0.110s	100%	2324	2434	3938
25%	0.095s	100%	2324	2436	3940
10%	0.096s	100%	2330	2440	3940
1%	0.085s	100%	2400	2500	4000
0.1%	0.082s	33%	-	-	4000

Table 4.1: Sampled schema evolution extraction for document collection `products`

As can be seen in the test results, changes in the schema are recognized at later timestamps, when the sampling rate is decreased. If the sampling rate gets too low, 0.1% in this case, some changes are missed entirely.

Time improvements are noticeable but minor for this example document collection, for which there are two explanations: First, the schema for the documents is very small, so extracting changes can be done quickly. And secondly, the current implementation of the prototype is not ideal, since even with an active filtering, there is still an iteration over all documents, filtered documents are just not further processed

and simply call `next()` on the iterator.

With a more complex schema, and a better implementation of the sampling process, time improvements will be more noticeable.

Table 4.2 shows the result for the same document collection with an applied dynamic sampling. The test compares two different sensitivity factors with various decrease factors. In this case, the sensitivity factor of 0.25% requires 20 documents to be unchanged, before the sampling rate is decreased, 0.125% requires 10 unchanged documents.

Sensitivity factor	Decrease factor	Time	Accuracy	Used documents	Timestamp		
					1st change	2nd change	3rd change
0.25	4	0.199s	33%	210	-	-	3938
0.25	2	0.232s	100%	313	2324	2433	3938
0.25	1.5	0.187s	33%	133	-	-	3938
0.25	1.25	0.194s	33%	154	-	-	3938
0.25	1.125	0.206s	100%	280	2324	2433	3938
0.125	4	0.180s	33%	135	-	-	3938
0.125	2	0.180s	33%	131	-	-	3938
0.125	1.5	0.167s	33%	111	-	-	3938
0.125	1.25	0.182s	33%	119	-	-	3938
0.125	1.125	0.208s	100%	224	2324	2433	3938

Table 4.2: Sampled schema evolution extraction for document collection `products`

The results are quite interesting and show how dynamic sampling can miss changes in the schema. As the first change is adding a property to the schema, that is removed only 109 documents later, it is easy to miss that short term change when the sampling rate aligns in such a way, that a document before the different schema, and the next after it are processed. In that case, no change is seen and thus no backtracking is started.

Chapter 5

Schema clustering

5.1 Problem definition

Extracting a single schema from a data set is a good approach if files are relatively similar in their structure. For data sets that are very heterogenic or have evolved significantly over time, a single schema is not ideal, as the extraction process would mark many attributes as optional [KSSR15]. In extreme cases, or when a document collection with multiple entities is extracted into a single schema, all attributes could become optional.

Such a general schema is not ideal, if it is supposed to be used for validation of the correctness of documents, or if a document uses a recent schema. Generating documents from data based on a general schema is difficult, too. For instance, let's assume your schema represents a user object and lists the name as an optional string attribute, but also as an optional object attribute in which firstname and lastname are separated into string attributes. To generate a user file, that is not only valid on a syntax level, but also can be used by your application, you need additional information about which name attribute to choose.

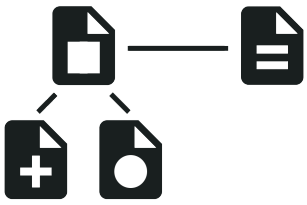

Figure 5.1: Different extracted schemata organized by relationship

To avoid a large number of optional elements, clustering of schemata can be a solution. Similar schemata, that have only minor differences, will be grouped together, while schemata that differ a lot will be in separate groups. This should reduce the percentage of optional attributes in the extracted schemata and also allows for outliers detection on a file, but also schema level.

5.2 Requirements

5.2.1 Time

The schema extracting part of the process should be similar to a regular schema extraction outlined in 2.2, so the time complexity for that part should not change. However collecting the different schemata and subsequent clustering will add complexity, which of course should be kept at a minimum by using common and efficient methods.

5.2.2 Reduction of optional elements and finding connected schemata

The process should find all different schemata within the document collection and organize them based on their relationships to each other. After this organization the process should then merge connected schemata in a way that reduces the number of optional elements and separates different entities within the document collection into separate schemata.

However the consequence of a low number of optional elements should not be an inflation in the number of schemata, e.g. by avoiding any optional elements by returning all found schemata without any grouping. The amount of optional elements should be balanced with the number of returned schemata.

5.3 Construction of schema cluster graph

Instead of building a universal schema during the process of schema extraction, each schema is being stored separately. Since the goal is a clustering of those schemata, we need to express the relations between them (see 5.3.1). Therefor a graph is chosen as the structure to store the extracted schemata.

At first the extracted schema T needs to be defined, which can be done similar to the definition of an SG in section 2.2.1.

$$\begin{aligned} T &= (V_S, E_S) \\ V_S &= (path, name, occurrenceCount) \\ E_S &= \{(v_1, v_2) | v_1, v_2 \in V_S \wedge v_1 \text{ isParentNodeOf } v_2\} \end{aligned}$$

The set T_{SET} contains all extracted schemata T_i. The graph of extracted schemata can then be defined as follows:

$$\begin{aligned} G &= (V, E) \\ V &= \{(t, count) | t \in T_{SET} \wedge count \in \mathbb{N}\} \\ E &= \{(v_1, v_2) \,| v_1, v_2 \in V\} \end{aligned}$$

5.3.1 Relationship between two extracted schemata

Comparing two schemata

The defined set of edges E is very general. For this application of schema clustering, the relations between the schemata are of interest, such as parent-child-relationships, which will be stored as edges in E. A schema t_1 is a parent of another schema t_2, when the schema tree of t_2 is a subgraph of the schema tree

of t_1, and the two schema trees have the same root node. For this purpose, the operator $\rightarrow^p$ is defined as

$$\begin{aligned} t_1 \rightarrow^p t_2 : & V_S(t_2) \subset V_S(t_1) \\ & \wedge E_S(t_2) \subset E_S(t_1) \\ & \wedge rootName(t_1) = rootName(t_2) \end{aligned}$$

With that, the definition of the edge set becomes

$$E = \{(v_1, v_2) \,|v_1, v_2 \in V \wedge v_1 \rightarrow^p v_2\}$$

Refinement of edge definition

The above definition is still very broad and would also include edges between grandparents and grandchildren, as is allows transitive relations. As a result, the smallest, most general schema in the graph could have a large number of edges, whereas the largest, most specific schema would have a very small number of edges associated with it. This is not an ideal basis to apply clustering algorithms on the graph, as it puts a bias on those general schemata and would center clusters around them. To even out the bias, any edges based on transitive shall be excluded. Furthermore, an edge (v_1, v_2) should not be included in the set of edges, if there is a longer path to go from node v_1 to node v_2.

$$v_1 \rightarrow^p v_2 \Rightarrow \nexists v_i, ..., v_j \in V \text{ where } v_1 \rightarrow^p v_i \rightarrow^p ... \rightarrow^p v_j \rightarrow^p v_2$$

Now we can redefine the set of edges E as follows:

$$E = \{(v_1, v_2) \,|v_1, v_2 \in V \wedge v_1 \rightarrow^p v_2 \wedge \nexists v_i, ..., v_j \in V \text{ where } v_1 \rightarrow^p v_i \rightarrow^p ... \rightarrow^p v_j \rightarrow^p v_2\}$$

5.3.2 Storing of schemata in nodes

Each node in the schema graph needs to store the schema as a tree structure, as well as the number of occurrences. Storing the schema as a tree has the benefit, that the comparison of schemata can be done very easily in linear time, in an approach similar to BFS. Section 5.3.3 describes this in more detail.

This storage method also keeps all data in memory. If memory space becomes an issue, the schemata could be stored outside of the graph G in files as strings. The graph G would then store a hash value of the schema. If the schema is needed for comparison, it needs to be be reconstructed as a graph from the file, which in turn would slow down the process.

5.3.3 Schema saving procedure

To generate the schema cluster graph SCG, the process iterates over all documents in the collection. For each individual document, the schema is extracted and added to the graph. In case a schema already was added to the graph, the *count* in the node is updated accordingly, to reflect the number of occurrences. This process can be seen in algorithm 5.6.
The algorithm above needs further detailing in two key parts: Firstly searching the graph to see if a schema already has been added, and secondly adding all edges to the graph. Those two parts can be found in algorithm 5.7 and 5.8 respectively.

Searching for schema in schema cluster graph

Searching for a given schema in the schema cluster graph, can be done with an BFS approach. Each node in the schema cluster graph is visited and the stored schema is compared with the given schema.

Algorithm 5.6 Schema Clustering Process

```
1: procedure GenerateSchemaClusterGraph(Document[] document_set)
2:     SCG := initiateSchemClusterGraph();
3:
4:     for Document d in document_set do
5:         s := extractSchema(d);
6:
7:         if SCG.hasSchema(s) then
8:             SCG.update(s);
9:
10:        else
11:            SCG.addNode(s);
12:            SCG.addEdges(s);
13:
14:    SCG.removeDirectEdges();
```

Algorithm 5.7 Searching for a given schema in the schema cluster graph

```
1: procedure SchemaClusterGraph.hasSchema(SchemaTree s)
2:     for SchemaTree t in SchemaClusterGraph.getNodes() do
3:         if s == t then
4:             return true;
5:
6:         return false;
```

This process has a time complexity of $\mathcal{O}(n)$ for n schemata in the schema cluster graph. However at each node a comparison of two trees will be required, which has a complexity of $\mathcal{O}(m)$ if the larger schema tree has m nodes. Thus searching for a schema in the schema cluster graph has a time complexity of $\mathcal{O}(m \cdot n)$.

Adding edges to the schema cluster graph

Finding the edges for a newly added schema s requires a scan of all schemata, that are currently in the schema cluster graph. This can be done with a *Depth-first search* (DFS) approach starting at the root nodes of the schema cluster graph, meaning the nodes without any parent nodes. The time complexity of the DFS approach to traverse the schema cluster graph is

$$\mathcal{O}(|V| + |E|)$$

For each visited schema, a check will be done to determine if the new schema is a potential parent or child schema. As mentioned earlier, the visited schema is considered a parent of the new schema, when the tree of the new schema is a subgraph of the tree of the visited schema. This test can be achieved in

$$\mathcal{O}(m)$$

when m is the number of nodes in the schema tree. The findings of these parent-child-checks will be added to a `parentOf` and a `childOf` list. At the end of the schema cluster graph traversal, the sets

$$\{(s, p) | p \in \texttt{parentOf}\}$$
$$\{(c, s) | c \in \texttt{childOf}\}$$

can be added to the set of edges E, as outlined in algorithm 5.8. At this point, the list of edges does not align with the definition in 5.3.1 yet. E can still contain transitive edges, that need to be removed. This will be done in a cleanup step after all schemata and edges have been added to the schema cluster graph. The process of finding edges has a time complexity of

$$\mathcal{O}(m \cdot |V| + |E|)$$

Algorithm 5.8 Method for adding an edge to the schema cluster graph

```
1:  procedure SchemaClusterGraph.addEdges(SchemaTree s)
2:      VISITED := new HashSet();
3:      PARENTS := new List();
4:      CHILDREN := new List();
5:
6:      for Node n in SCG.getRoots() do          ▷ Searching relationships of schema
7:          DfsWalk(n, s);
8:
9:      for p in PARENTS do                      ▷ Adding edges from parent to schema
10:         SCG.addEdge(p, s);
11:
12:     for c in CHILDREN do                     ▷ Adding edges from schema to children
13:         SCG.addEdge(s, c);
```

Algorithm 5.9 DFS method for algorithm 5.8

```
1:  procedure DsfWalk(Node n, SchemaTree s)
2:      VISITED.add(n);
3:
4:      if s.isParentOf(n.getSchema()) then
5:          CHILDREN.add(n);
6:
7:      else
8:          if n.getSchema().isParentOf(s) then
9:              PARENTS.add(n)
10:
11:     for Node c in n.getChildNodes() do
12:         if VISTED.isUnvisited(c) then
13:             DsfWalk(c, s);
```

Reduce number of edges

Since the collection E contains all possible parent-child relationships at this point, a cleanup is needed to align with the definition in 5.3.1. As stated there, the collection of edges cannot contain any direct edge (a, b), if there is a path $PATH(a, b)$ with a length greater than two. Algorithm 5.10 outlines this process.

Having a separate step to remove these edges helps to reduce the complexity of adding edges when a new schema is introduced to the schema cluster graph. This is important during the iteration through

Algorithm 5.10 Removing direct edges from graph

```
1: procedure SchemaClusterGraph.RemoveDirectEdges
2:     REMOVALS = new LIST();
3:
4:     for Node a in SCG do
5:         for Node b in SCG do
6:             if a != b and SCG.hasEdge(a,b) and getLongestPath(a,b) > 2 then
7:                 SCG.removeEdge(a,b);
```

large document collections with potentially many different schemata. However it has to be noted that this cleanup process has a complexity of

$$\mathcal{O}(|E|)$$

for iterating all edges and

$$\mathcal{O}(|V| + |E|)$$

for the path lookup, as it can be done with DFS. In combination the step to reduce the number of edges has a complexity of

$$\mathcal{O}(|E| \cdot (|V| + |E|))$$

Performance improvements

The processes outlined above can be improved in different ways at different points. If combined all together, the complexity of iterating the documents results in

$$\mathcal{O}\left(|docs| \cdot (|V| \cdot |nodesInSchema| + |V| \cdot |nodesInSchema| + |E|) + |E| \cdot (|V| + |E|)\right)$$

Hash sets To avoid searching the complete schema cluster graph SCG for the schema to be added to it, a hash table can be used. Given a good hashing method, the lookup time whether a the schema already exists or not, would reduced from the prior $O(|V| \cdot |nodesInSchema|)$ to $O(1)$, and the overall complexity reduced to

$$\mathcal{O}\left(|docs| \cdot (|V| \cdot |nodesInSchema| + |E|) + |E| \cdot (|V| + |E|)\right)$$

The same hash table can be used find the parent-child relationships. Instead of searching the schema cluster graph with DFS, an iteration over all hash table entries is sufficient, reducing the matching time complexity from $\mathcal{O}(m \cdot |V| + |E|)$ to $\mathcal{O}(m \cdot |V|)$, with an overall reduction to

$$\mathcal{O}\left(|docs| \cdot |V| \cdot |nodesInSchema| + |E| \cdot (|V| + |E|)\right)$$

Distributed processing In a similar way to chapter 3, distribution can be introduced as another way of improving performance. For this application, the process could be outlined as follows:

1. Distribute document collection to separate computing nodes
2. Construct independent schema cluster graph on each computing node without edges
3. Bring independent schema cluster graphs together and merge into new graph *master*. For each *node* from schema cluster graph A to be merged into *master*

(a) if *node* exists

- Update *node* in *master* with info from A

(b) else

- Add *node* from A to *master*
- Add edges described in this section

4. Remove direct edges from combined graph *master* as before

5.3.4 Example graph result

For an example illustration of the graph described in this section, assume we have the schemata 0 to 7 in Listing 5.1, with `a`, `h`, `p`, `s1` and `s2` being properties of the schema.

Listing 5.1: Example schemata for illustration of schema graph

```
0: {n,a,h}
1: {n,a,h,p}
2: {n,a,  p}
3: {n,a,  p,s1}
4: {n,a,  p,   s2}
5: {n,a,h,p,s1}
6: {n,a}
7: {n}
```

The extracted schemata would be stored the nodes 0 to 7, making our set of nodes

$$V = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

for our graph G. The initial set of edges

$$\begin{aligned} E = \{&(0,6),(0,7),\\ &(1,0),(1,2),(1,6),(1,7),\\ &(2,6),(2,7),\\ &(3,2),(3,6),(3,7),\\ &(4,2),(4,6),(4,7),\\ &(5,0),(5,1),(5,2),(5,3),(5,6),(5,7),\\ &(6,7)\} \end{aligned}$$

will then be reduced to the $\rightarrow^p$ relationships, resulting in the following set of edges

$$E = \{(0,6),(1,0),(1,2),(2,6),(3,2),(4,2),(5,1),(5,3),(6,7)\}$$

A visual representation of this graph can be found in Figure 5.2.

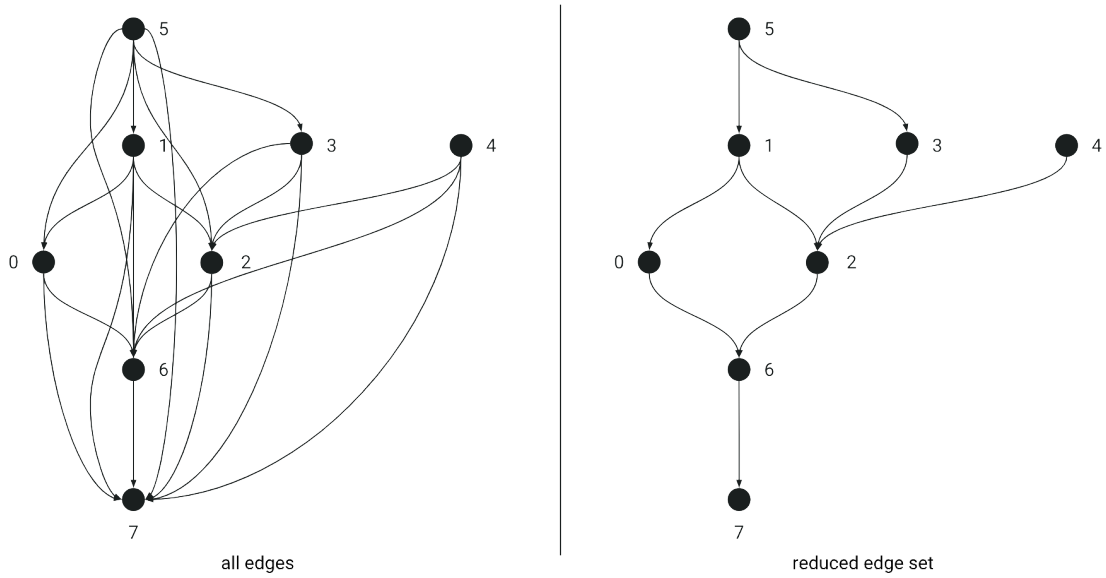

Figure 5.2: Example graph for schemata listed in Listing 5.1

5.4 Clustering in schema cluster graph

Based on the prior constructed schema cluster graph, the process now tries to find different clusters within that graph. This section evaluates clustering approaches from 2.4 for this specific application and then outlines the used clustering approach.

5.4.1 Evaluation of clustering methods

The approaches of k-means, k-medoids and hierarchical clustering all require a distance *dist* function to determine the similarity of data points. For the clustering of schemata, *dist* could be defined as the number of generations between two schemata or the number of changed properties between them.

Using k-means or k-medoids to find groups within the schema cluster graph is not ideal however, as a number of clusters need to be set in advance. Unless the exact or expected numbers of schemata in the document collection is known, k-means and k-medoids are not viable.

Hierarchical clustering seems intuitive to be applied here, as the schemata are organized based on their parent-child relationships, and thus a hierarchy is form. However hierarchical clustering is not used because of it's time and space complexity, and also because the granularity is not needed.

Min-cut would be a good application if the number of different schemata is known before the extraction, or can be at least estimated. Based on that, the same number of cuts could be made in the graph to divide the schemata from each other. However this would still require a weight assigned to the edges.

Similar to the methods above, spectral clustering has limitations regarding the number of clusters. A set of k clusters can extracted by using k eigenvectors, which again requires knowledge or an educated

guess for the number of clusters beforehand. Using a recursive approach it would be possible to search for clusters for a stable state at which no further separation can be done. However with the $\mathcal{O}(n^3)$ complexity [YHJ09] of spectral clustering, this will not be feasible.

5.4.2 Applied clustering approach

Outliers removal

The clustering process' first step is separating outliers schemata, that are used in under a certain number of documents. This threshold t should also be configurable by a user, to adjust for the specific data set. Removing outliers schemata helps reducing the number of optional elements in the schema that is generated per cluster. This also improves the quality of the schemata representing the majority of the document collection. The outliers schemata can be found by iterating over the nodes of the schema cluster graph in $\mathcal{O}(m)$ time, and checking

$$\frac{|documentsWithSchema(s)|}{|documents|} < t$$

as seen in algorithm 5.11. The removed schemata are exported as individual clusters with their own schemata.

Algorithm 5.11 Removing outliers schemata from schema cluster graph

```
1: procedure SCG.RemoveOutlier
2:     count := SCG.getDocumentCount();
3:     threshold := getThreshold();
4:
5:     for Node m in SCG.getNodes() do
6:         if (m.getOccurrence() / count) < threshold then
7:             SCG.removeNode(m);
8:             SCG.removeEdges(m);
9:             m.exportAsJSONSchema();
```

Finding components

The second step of the clustering approach finds the independent components of the schema cluster graph. This separates different entities into separate schemata, as it is unlikely, that there are schema relationships between the entities.

With an undirected graph, independent components could be found via BFS or DFS, but these approaches cannot be applied directly on a directed graph here. To find the components, the used process starts by finding the sources of the graph. Sources are the nodes without any incoming connections. To find them, all nodes of the graph can be added to a list *sources*. During an iteration of the hash set of edges (see 5.3.3), all reachable nodes are removed from *sources*. After the iteration, *sources* contains all nodes without incoming connections, as outlined in algorithm 5.12. This process can be done in $\mathcal{O}(|V| + |E|)$. Starting from the source nodes, the schema cluster graph can now be traversed to find the components. For each source, all reachable nodes are assigned to same component and stored in a list. This can be done by following the edges in a DFS approach, which can be done in $\mathcal{O}(|V| + |E|)$.

If two components intersect and share the same nodes, those node lists of components are merged together. Calculating the intersection of two lists has a complexity of $\mathcal{O}(m * n)$ if the lists have m and n entries. By using hash sets as data structures, this complexity can be reduced to a linear $\mathcal{O}(m)$, as the

Algorithm 5.12 Finding sources in schema cluster graph

```
1: procedure SCG.GetSources
2:     List sources := SCG.getNodes();
3:
4:     for Edge e in SCG.getEdges() do
5:         sources.remove(e.target);
6: return sources;
```

lookup for the second hash set is $\mathcal{O}(1)$.

The process is outlined in algorithm 5.13. Overall, finding the components for a schema cluster graph with s sources and a maximum of m nodes per component, requires

$$\mathcal{O}((|V| + |E|) + s \cdot (|V| + |E| + m))$$

or simplified

$$\mathcal{O}(s \cdot (|V| + |E|))$$

Algorithm 5.13 Finding components in schema cluster graph

```
1: procedure SCG.GetComponents
2:     List components := new List();
3:     List sources := SCG.getNodes();
4:
5:     for Node source in SCG.getSources() do
6:         List component := SCG.getReachableNodes(source);
7:         connected := false;
8:
9:         for List c in components do
10:            if component.intersects(c) then
11:                c.addAll(component.getNodes());
12:                connected := true;
13:        if !connected then
14:            components.add(component);
15: return components;
```

Clustering within the components

The next part in the clustering process is the actual composition of the clusters within each component. Clusters are based on the nodes with the highest indegrees and contain all nodes on the paths to those high degree nodes. Using this approach accomplishes two points:

- The number of schemata is reduced, as the most connected schemata merge themselves and their ancestor schemata into a single schema.
- The number of optional elements in a schema is reduced, as broader child schemata are not included in the schema merge from above.

This cluster composition can be divided into six steps:

1. In the first step calculates the indegrees for the nodes for easier access during the process. The complexity of this is $\mathcal{O}(m^2)$ for m schemata in the graph, as shown in algorithm 5.14.

Algorithm 5.14 Calculating indegrees for nodes

```
1: procedure SCG.GetInDegrees
2:     Map indegrees := new Map();
3:
4:     for Edge e in SCG.getEdges() do
5:         if inDegress.hasKey(edge.target) then
6:             indegrees.put(edge.target, indegrees.get(edge.target) + 1);
7:         else
8:             indegrees.put(edge.target, 1);
9:
10:    for Node n in SCG.getNodes() do
11:        if !indegrees.hasKey(n) then
12:            indegrees.put(n, 0);
13: return indegrees;
```

2. The second step marks all nodes unvisited.
3. In the third step, the node with the highest indegree, and that is still unvisited, is selected. This node will become the base for the cluster.
4. Step four is collecting all paths to the just selected node, by finding all paths between all source nodes and the selected node. While the search for all paths between two nodes could be done in $\mathcal{O}(|V| + |E|)$ time, doing so for multiple source nodes will increase the complexity to $\mathcal{O}(|V|^2)$, assuming the sources are still saved from 5.4.2.
5. In step five the schemata from the nodes of the collected paths, as well as the selected node, are merged into a single schema. This process can be done in the same way as the merge process described in section 3.3.1.
6. The sixth step marks all nodes collected in step 4 as visited. Also this step needs to update the list of sources, before going back to 3 and continue repeating these steps until all nodes are visited.

The result of the clustering process is a list of schemata, each representing a cluster in the component. The process is outlined in algorithm 5.15.

Other clustering options in

Apart from the clustering approach above, two other ideas are worth mentioning here.

One of them is to remove the schemata with the highest occurrence rate in the document collection, and save them as individual clusters, before the clustering of the rest of the component starts. By doing so, the majority of the documents would be represented by a schema with no optional elements. However this can lead to an increase in the number of schemata, if the frequently used schema was highly connected.

The second idea makes a slight change to the clustering approach in section 5.4.2. Instead of collecting all paths leading to the selected node, this approach would be similar to k-means or k-nearest neighbor approaches and collect only paths of a length of up to k. This way the highly connected schemata are still helping to reduce the overall number of schemata for the component, but the number of optional elements would be reduced, as less of the more specific schemata are merged into the selected node's schema.

Algorithm 5.15 Collecting clusters in independent components

```
1:  procedure SCG.GetClusters
2:      List clusters := new List();
3:      Map indegrees := SCG.getInDegrees();
4:      List unvisited := SCG.getNodes();
5:
6:      while unvisited.hasNode() do
7:          List currentCluster := new List();
8:          Node currentMaxNode := unvisited.getNodeWithMaxInDegree();
9:          List paths := SCG.getPathsTo(currentMaxNode);
10:
11:         for Path p in paths do
12:             for Node n in p.getNodes() do
13:                 currentCluster.add(n);
14:                 indegrees.remove(n);
15:                 unvisited.remove(n);
16:
17:         clusters.add(currentCluster);
18: return clusters;
```

5.5 Quality and performance evaluation

Listing B.4 provides a minimum example of 6 different schemata and two entity types, that would have a schema cluster graph as in Figure 5.3. Using the clustering approach described in this chapter, the schemata in Listing 5.2 and 5.3 can be extracted.

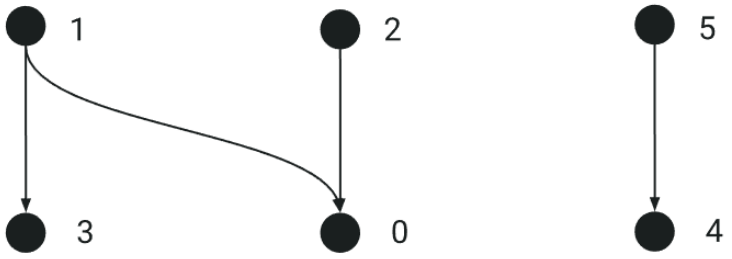

Figure 5.3: Schema cluster graph for Listing B.4

Listing 5.2: Minimum example for clustering

```
{
  "type" : "object",
  "$schema" : "http://json-schema.org/draft-07/schema",
  "description" : "Occurrences: 4",
  "title" : "testing_clustering",
  "properties" : {
    "name" : {
      "type" : "string",
      "description" : "in_documents 3"
    },
    "id" : {
      "type" : "integer",
      "description" : "in_documents 4"
    },
    "job" : {
      "type" : "string",
      "description" : "in_documents 1"
    },
    "age" : {
      "type" : "integer",
      "description" : "in_documents 2"
    }
  }
}
```

Listing 5.3: Minimum example for clustering

```
{
  "type" : "object",
  "$schema" : "http://json-schema.org/draft-07/schema",
  "description" : "Occurrences: 2",
  "title" : "testing_clustering",
  "properties" : {
    "product" : {
      "type" : "string",
      "description" : "in_documents 2"
    },
    "color" : {
      "type" : "string",
      "description" : "in_documents 1"
    },
    "id" : {
      "type" : "integer",
      "description" : "in_documents 2"
    }
  }
}
```

Chapter 6

Comparison of outlined techniques

The techniques described in the previous three chapters are intended to extract different types of information about a document collection. This chapter will provide a better understanding of the differences, as all three techniques are applied to the same document collection

6.1 Document collection `VideoSpace`

The example collection of documents will contain two different entity types: `video` and `user`. In a typical schema extraction, as in 2.2, you would process those separately, otherwise it is likely to have a large percentage of optional elements in the schema, even in the case that the schema from each entity would stay unchanged throughout the collection. The distributed schema extraction approach would have the same result in a faster time, however the schema clustering technique should be able to separate the different entities into separate schemata.
The extraction of schema evolution as in 2.2 would also typically be done on each entity individually, this also is the case for the outlined sampling techniques. Therefor the evolution will be applied on the entities `video` and `user` separately. Samples of the different entity types can be found in C.1

In order to have variety in the extracted schemata and the evolution of the schemata, the sample collection has changes in between the documents. This will create optional elements in all described schema extraction methods. The typical schema evolution extraction will also find the changes as evolution steps, and it is to be expected that the schema evolution extraction with fixed rate and dynamic rate sampling will find these evolution steps as well. However the fixed rate sampling is like to have different timestamps for the first occurrence of a change. In C.2 you can find an outline of the applied changes.

6.2 Processing results

6.2.1 Distributed Schema Extraction

As shown in Table 6.1, the runtimes reduces drastically for this document collection. However there is a noticeable gap between the average time per extraction, and the overall runtime. This comes down to the distribution of the different schema versions throughout the collection, and that some threads include more schema versions then just their own. Extracting the schemata for the entities separately evens out the average extraction times.

Threads	1	2	4	8	16
Avg. time per thread	59.734	14.182	6.113	1.456	0.605
Merge time	0.003	0.003	0.005	0.005	0.005
Total time	59.772	22.550	18.019	3.623	1.088

Table 6.1: Distributed schema extraction runtimes (in seconds) for `VideoSpace` data set

6.2.2 Schema Evolution

Extracting the schema evolution without any sampling uncovers all evolution steps as expected (see Table 6.2). The points of schema changes, timestamp 5001, 10001, 15001 and 20001 are all extracted correctly.

Time	**Operation**	**Path**
5001	Add	/comments
10001	Remove	/responses
15001	Add	/subscriber
15001	Add	/phrase
15001	Add	/user_name
15001	Add	/contact
15001	Add	/friends
15001	Remove	/shares
15001	Remove	/comments
15001	Remove	/downloads
15001	Remove	/meta_information
15001	Remove	/ratings
15001	Remove	/uploader
15001	Remove	/views
20001	Remove	/friends

Table 6.2: Extracted evolution from `VideoSpace` collection without sampling

Fixed sampling rates lower than 100% are still able to find all operations while decreasing the runtime, though the time of uncovering varies. For example a sampling rate of 15% finds the changes at 5006, 10004, 15002, and 20007. The fixed rate sampling with 15% takes 0.948s.

In comparison, a dynamic rate sampling with a decrease factor of 1.5 and sensitivity factor of 0.008% (two documents need the same schema before decreasing the sampling rate further) requires the processing of 205 documents in 0.802s, which approximates to an overall sampling rate of 0.82%. A fixed sampling rate with the same value takes the same time, but won't find the first occurrences of the schema change.

6.2.3 Schema Clustering

Since the document collection contains two different entity types, the clustering process should separate them into different schemata, and reducing the number of optional elements by clustering.
Running the clustering process on this document collection reveals five different schemata, with three connections between them, as illustrated in Figure 6.2. This schema cluster graph can be exported into two separate clusters, as seen in Figure 6.2.

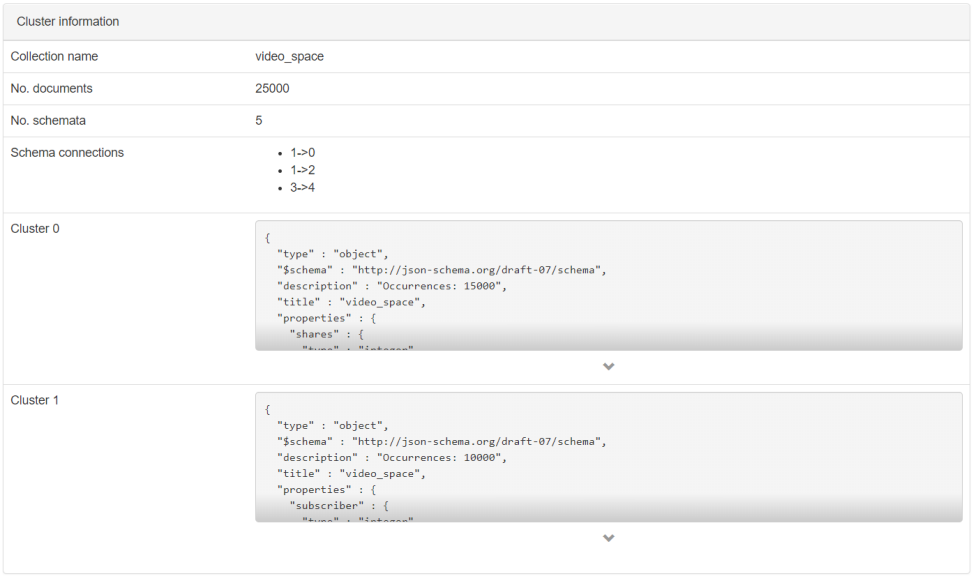

Figure 6.1: Extracted clusters in Darwin tool

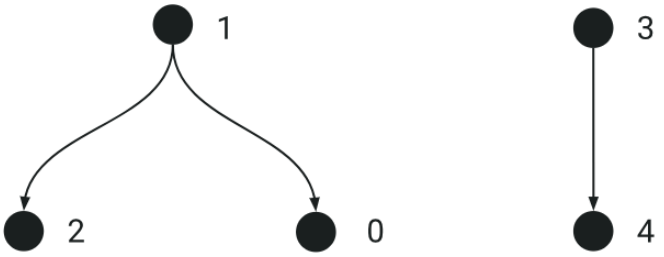

Figure 6.2: Schema cluster graph for collection `VideoSpace`

Chapter 7

Conclusion and Outlook

Schema extraction from large data collections still has lots of opportunities for improvements. This thesis has shown a general distribution and merge process to decrease the runtime of a schema extraction, which has promising results.

It also became clear that some areas, such as sampling for schema evolution extraction, optimizations need to be tied closer to the data or have to make trade-offs in terms of the accuracy.

As an outlook, there are areas where the proposed processes of this paper could be improved. During the merge process of the schema extraction, the separate schema trees could be quite different, for example when they come from different sites of a company. So far they would be merged together into one schema with lots of optional elements, but an improvement could be to check if one schema is the child of another, or even apply a clustering of schemata.

The clustering itself also would benefit from improvements. For example the timestamps of the documents could be taken into account, so that old and not any longer used schemata are removed as outliers before the clusters are built.

Appendix A

Related Work

A.1 Data generation: json-maker.com Backend

Listing A.1: Definition of helper methods for json-maker.com

```
var faker     = require('faker');
var dummyjson = require('dummy-json');
var helper    = {
  faker: function(data) {
      if (!data || typeof data !== 'string' || data.length == 0) {
          server.error = 'faker.js method is missing';
          return server.error;
      }

      var parameters = data.split('.');

      if (parameters.length < 2) {
          server.error = 'faker.js method is missing';
          return server.error;
      }

      if (typeof faker[parameters[0]] === 'undefined') {
          server.error = "faker.js module '" + parameters[0] + "' does
              not exist";
          return server.error;
      }

      if (typeof faker[parameters[0]][parameters[1]] === 'undefined') {
          server.error = "faker.js method '" + parameters[1] + "' does
              not exist";
          return server.error;
      }

      return faker.fake('{{' + data + '}}');
  },

  optional: function(data) {
```

```
31        data.hash = data.hash || {};
32        data.hash.comma = false;
33        return dummyjson.helpers.repeat(0, 1, data);
34    }
35  }
36
37  var result    = dummyjson.parse(input_schema, {helpers: helper});
```

A.2 Schema Extraction

Listing A.2: Extracted schema from Figure 2.1

```
1  {
2    "$id" : "user",
3    "$schema": "http://json-schema.org/draft-07/schema#",
4    "type" : "object",
5    "properties" : {
6      "_id" : {
7        "type" : "string"
8      },
9      "age" : {
10       "type" : "number"
11     },
12     "name" : {
13       "type" : "object",
14       "properties" : {
15         "first" : {
16           "type" : "string"
17         },
18         "last" : {
19           "type" : "string"
20         }
21       },
22       "required" : ["first", "last"]
23     }
24     "address" : {
25       "type" : "string"
26     },
27     "friends" : {
28       "type" : "array",
29       "items" : {
30         "type" : "object",
31         "properties" : {
32           "id" : {
33             "type" : "number"
34           },
35           "name" : {
36             "type" : "string"
37           }
```

```
        },
        "required" : ["id", "name"]
      }
    }
  },
  "required" : ["_id", "age", "name", "address", "friends"]
}
```

A.3 Schema Evolution

Listing A.3: Schema evolution example File 1

```
{
  "time" : 1,
  "name" : "Alice"
}
```

Listing A.4: Schema evolution example File 2

```
{
  "time" : 2,
  "name" : "Bob",
  "age" : 21
}
```

Listing A.5: Schema evolution example File 3

```
{
  "time" : 3,
  "name" : "Carol",
  "role" : "admin"
}
```

Appendix B

Performance evaluation

This chapter is listing example documents from the collections used during for the performance evaluation. The complete data collection, along with the schema for the data generation with `json-maker.com`, can be found in a GitHub repository at `https://github.com/fbeuster/nosql-data-profiling`

B.1 Distributed schema extraction

The first document collection `people` for the distributed schema extraction, contains small documents with a few personal information of a user for an online platform. The collection has 10,000 documents, an example can be found in Listing B.1.

Listing B.1: Example document from the people entity

```
{
{
    "name" : "Ellie Schmeler",
    "street" : "365 Tremblay Divide",
    "city" : "New Eltonside",
    "zip" : "33062-9828",
    "user_name" : "Caleigh_DuBuque9",
    "catch_phrase" : "If we bypass the transmitter, we can get to the
       USB system through the auxiliary THX firewall!",
    "user_id" : 0,
    "user_mail" : "Marty_Kuhn44@gmail.com",
    "favorite_colors" : [
        "#6a5325",
        "#3f7a74",
        "#558034",
        "#777a0e",
        "#235008"
    ],
    "reg_date" : "Tue Nov 25 2014 15:17:06 GMT+0100 (CET)",
    "sign_ups" : {
        "newsletter" : false
    }
}
```

The videos document collection describes videos on a video sharing platform. Each document contain meta information about the video, statistical data, and user generated content in the form of comments and answers. The collection contains 10,000 documents, an example can be found in Listing B.2.

Listing B.2: Example document from the video entity

```
{
  "videoid" : 0,
  "meta_information" : {
    "title" : "I&#x27;ll bypass the open-source SDD circuit, that should
        hard drive the CSS application!",
    "description" : "Voluptas nam explicabo...",
    "tags" : [
      "est",
      "nostrum"
    ],
    "monetized" : false
  },
  "views" : 888,
  "ratings" : {
    "likes" : 46,
    "dislikes" : 41
  },
  "uploader" : "FTUPwzcMAoIg5JK",
  "comments" : [
    {
      "uid" : "evuctVeh0OcUQCB",
      "date" : "Mon Jan 01 2018 00:00:00 GMT+0100 (CET)",
      "content" : "Aut non sed sed illo velit similique necessitatibus
         reprehenderit.",
      "ratings" : {
        "likes" : 5,
        "dislikes" : 1
      },
      "answers" : [
        {
          "uid" : "n03zr_P5pXF6XLi",
          "date" : "Mon Jan 01 2018 00:00:00 GMT+0100 (CET)",
          "content" : "Earum rerum voluptatum."
        }
      ],
      "ratings" : {
        "likes" : 1,
        "dislikes" : 1
      }
    },
    ...
  ],
  "downloads" : 24
}
```

B.2 Sampled schema evolution extraction

The example document collection contains 7948 small documents with product information, an example document can be found in Listing B.3, the entire documen collection is available on GitHub. The following changes are made to the schema throughout the collection:

- Add property `material` at timestamp 2324
- Remove property `material` at timestamp 2432
- Add property `description` at timestamp 3938

Listing B.3: Example document from the product entity

```
{
  "id" : 2614,
  "name" : "Generic Steel Ball",
  "color" : "orange",
  "price" : 835.00
}
```

B.3 Schema clustering

Listing B.4: Minimum example for clustering

```
[
  {
    "id" : 0,
    "name" : "Alice"
  },
  {
    "id" : 1,
    "name" : "Bob",
    "age" : 42
  },
  {
    "id" : 2,
    "name" : "Carla",
    "job" : "SWE"
  },
  {
    "id" : 3,
    "age" : 21
  },
  {
    "id" : 4,
    "product" : "car"
  },
  {
    "id" : 5,
```

```
26      "product" : "car",
27      "color" : "red"
28    }
29  ]
```

Appendix C

Comparison of outlined techniques

The entire document collection `VideoSpace`, along with the schemata to generate the documents, can be found at my GitHub page at `https://github.com/fbeuster/nosql-data-profiling`.

C.1 Different entity types

The following two documents are examples taken for each entity from the `VideoSpace` collection. The example documents contain all possible properties from each entity.

Listing C.1: VideoSpace user entity document

```
{
    "id" : 17331,
    "user_name" : "Bryon85",
    "phrase" : "You can&#x27;t input the feed without generating the
       optical ADP interface!",
    "subscriber" : 178,
    "contact" : {
      "phone" : "(684) 598-6814 x7793",
      "user_mail" : "Daisy.Schaefer47@gmail.com"
    },
    "friends" : [
        609,
        796,
        2359,
        2053,
        3468,
        3008,
        2311,
        3992,
        3967,
        991,
        4867,
        4212,
        1073
    ]
  }
```

Listing C.2: VideoSpace video entity document

```
{
  "id" : 6666,
  "meta_information" : {
    "title" : "If we copy the interface, we can get to the XML monitor
       through the cross-platform ADP microchip!",
    "description" : "Aut ipsum ipsam...",
    "tags" : [
      "ducimus",
      "a",
      "quae"
    ],
    "monetized" : true
  },
  "views" : 936,
  "ratings" : {
    "likes" : 39,
    "dislikes" : 38
  },
  "uploader" : "ksBRZK2ReD_0kae",
  "comments" : [
    {
      "uid" : "Du7PMDDNyTWi38W",
      "date" : "Mon Jan 01 2018 00:00:00 GMT+0100 (CET)",
      "content" : "Earum aliquid et.",
      "ratings" : {
        "likes" : 2,
        "dislikes" : 3
      },
      "answers" : [
        {
          "uid" : "aPzyNtpERoyN789",
          "date" : "Mon Jan 01 2018 00:00:00 GMT+0100 (CET)",
          "content" : "Sequi iste eum."
        }
      ],
      "ratings" : {
        "likes" : 5,
        "dislikes" : 2
      }
    }
  ],
  "responses" : [
      6049,
      9235,
      6625
  ],
  "shares" : 48,
  "downloads" : 29
}
```

C.2 Entity changes

The following listings illustrate the properties that are affected by the evolution in the document collection `VideoSpace`.

Listing C.3: Property `comments` added to video entity

```
{
  ...
  "comments" : [
    {
      "user_id" : 21,
      "message" : "Hello world!"
    },
    ...
  ]
  ..
}
```

Listing C.4: Property `responses` removed from video entity

```
{
  ...
  "responses" : [
    "X0gQ6kzBvD8",
    "kQgNqtecAM4",
    ...
  ]
  ..
}
```

Listing C.5: Property `friends` removed from user entity

```
{
  ...
  "friends" : [
    17,
    4,
    ...
  ]
  ..
}
```

Statutory Decleration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Felix Beuster Rostock, August 27th, 2018

Abbreviations

A.I.	Artificial Intelligence
BFS	Breadth-first search
DFS	Depth-first search
JSON	JavaScript Object Notation
RG	Reduced Structure Identification Graph
SG	Structure Identification Graph
XML	Extensible Markup Language

Bibliography

[ADHP09] Aloise, Daniel, Amit Deshpande, Pierre Hansen and Preyas Popat: *NP-hardness of Euclidean sum-of-squares clustering.* Machine learning, 75(2):245–248, 2009.

[Beu18] Beuster, Felix: *JSON Maker*, May 2018. `https://json-maker.com` last visited on Jun 18th, 2018.

[CLRS09] Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest and Clifford Stein: *Introduction to algorithms.* MIT press, 2009.

[Dor18] Dorfman, Justin: *jdorfman/awesome-json-datasets: A curated list of awesome JSON datasets that don't require authentication.*, April 2018. `https://github.com/jdorfman/awesome-json-datasets` last visited on May 20th, 2018.

[FF56] Ford, Lester R and Delbert R Fulkerson: *Maximal flow through a network.* Canadian journal of Mathematics, 8(3):399–404, 1956.

[FHT01] Friedman, Jerome, Trevor Hastie and Robert Tibshirani: *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:, 2001.

[HW79] Hartigan, John A and Manchek A Wong: *Algorithm AS 136: A k-means clustering algorithm.* Journal of the Royal Statistical Society. Series C (Applied Statistics), 28(1):100–108, 1979.

[IKI94] Inaba, Mary, Naoki Katoh and Hiroshi Imai: *Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering.* In *Proceedings of the tenth annual symposium on Computational geometry*, pages 332–339. ACM, 1994.

[KAS+17] Klettke, Meike, Hannes Awolin, Uta Störl, Daniel Müller and Stefanie Scherzinger: *Uncovering the evolution history of data lakes.* In *Big Data (Big Data), 2017 IEEE International Conference on*, pages 2462–2471. IEEE, 2017.

[KR09] Kaufman, Leonard and Peter J Rousseeuw: *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[KSSR15] Klettke, Meike, Uta Störl, Stefanie Scherzinger and OTH Regensburg: *Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores.* In *BTW*, volume 2105, pages 425–444, 2015.

[M+67] MacQueen, James et al.: *Some methods for classification and analysis of multivariate observations.* In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[Mar18] Marak: *Marak/faker.js: generate massive amounts of fake data in Node.js and the browser*, May 2018. `https://github.com/marak/Faker.js` last visited on May 21th, 2018.

[MLN00] Moh, Chuang-Hue, Ee-Peng Lim and Wee-Keong Ng: *DTD-Miner: a tool for mining DTD from XML documents.* In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, pages 144–151. IEEE, 2000.

[MNV09] Mahajan, Meena, Prajakta Nimbhorkar and Kasturi Varadarajan: *The planar k-means problem is NP-hard.* In *International Workshop on Algorithms and Computation*, pages 274–285. Springer, 2009.

[Oma16] Omanashvili, Vazha: *JSON Generator – tool for generating random JSON data*, April 2016. `https://next.json-generator.com` last visited on May 20th, 2018.

[RM05] Rokach, Lior and Oded Maimon: *Clustering methods.* In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[Rus16] Rust, Peter: *prust/wikipedia-movie-data: JSON data on American movies scraped from wikipedia*, February 2016. `https://github.com/prust/wikipedia-movie-data` last visited on May 20th, 2018.

[Sch18] Schema, JSON: *JSON Schema*, 2018. `https://json-schema.org` last visited on Aug 16th, 2018.

[SKS13] Scherzinger, Stefanie, Meike Klettke and Uta Störl: *Managing schema evolution in nosql data stores.* arXiv preprint arXiv:1308.0514, 2013.

[SM00] Shi, Jianbo and Jitendra Malik: *Normalized cuts and image segmentation.* IEEE Transactions on pattern analysis and machine intelligence, 22(8):888–905, 2000.

[SMKS17] Störl, Uta, Daniel Müller, Meike Klettke and Stefanie Scherzinger: *Enabling Efficient Agile Software Development of NoSQL-backed Applications.* Datenbanksysteme für Business, Technologie und Web (BTW 2017), 2017.

[Swe17] Sweetman, Matt: *webroo/dummy-json: Generates random dummy JSON data in Node.js*, January 2017. `https://github.com/webroo/dummy-json` last visited on May 20th, 2018.

[U.S18] U.S. Government: *Datasets - Data.gov*, May 2018. `https://catalog.data.gov/dataset` last visited on May 20th, 2018.

[VL07] Von Luxburg, Ulrike: *A tutorial on spectral clustering.* Statistics and computing, 17(4):395–416, 2007.

[W+01] West, Douglas Brent et al.: *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[YHJ09] Yan, Donghui, Ling Huang and Michael I Jordan: *Fast approximate spectral clustering.* In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 907–916. ACM, 2009.