ABSTRACT

SCHEMALYSIS: VISUALIZATION OF SUB-SCHEMAS IN DOCUMENT NoSQL DATABASES

by Andrew Joseph DePero

NoSQL database systems are useful for managing large and diverse data sets associated with Big Data. Highly diverse data sets contain data with different structures, but often there are no readily available schemas describing the structures. The lack of a uniform structure for data may make it difficult to understand and query a database. Recent research and industry software tools extract some aspects of the structures inherent in a NoSQL database; most tools provide a schema that gives the union of attributes across all objects, termed a union schema. Some provide sample values for attributes. We present Schemalysis, a tool for analyzing and displaying the sub-schemas of a document NoSQL database along with example instances. The web application implements an algorithm that reads objects and detects individual sub-schemas of each document in a document database, as well as the database's union schema. We also conduct three different case studies to validate the functionality of Schemalysis with real-world data and compare and contrast to existing tools for extracting schemas.

SCHEMALYSIS: VISUALIZATION OF SUB-SCHEMAS IN DOCUMENT NoSQL DATABASES

A Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Science

by

Andrew Joseph DePero

Miami University

Oxford, Ohio

2022

Advisor: Karen Davis

Reader: James Kiper

Reader: Alan Ferrenberg

This Thesis titled

SCHEMALYSIS: VISUALIZATION OF SUB-SCHEMAS IN DOCUMENT NoSQL DATABASES

by

Andrew Joseph DePero

has been approved for publication by

The College of Engineering and Computing

and

The Department of Computer Science & Software Engineering

Karen C Davis

James Kiper

Alan Ferrenberg

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my advisor, Karen C Davis, for all the guidance, support, and enthusiasm throughout this process. I would also like to thank my committee members, James Kiper and Alan Ferrenberg, for all of their consideration and support throughout my years as a student at Miami University. To all the faculty and staff of the Miami University Department of Computer Science and Software Engineering—thank you for providing such wonderful learning experiences even in the midst of a pandemic and allowing me to become the computer scientist and researcher that I now am. Finally, to Ginger—my small fluffy feline friend and supporter.

Chapter 1

Introduction

With the emergence of Big Data, data is being stored at much greater capacities than ever before. Database systems must be equipped to handle and query millions of data records for some organizations. This scale of data collection and handling can be very difficult to maintain. NoSQL databases have emerged to address the volume and variety of Big Data. The semi-structured and flexible nature of NoSQL data storage is often termed *schemaless*, which makes application development faster but query specification more difficult.

Reverse engineering is a process used to extract a schema, or the structure of the information a database contains. This is a common strategy used to understand the contents of a database, and how to interact with it. After reverse engineering a database, the schema can be represented using a visual model to further improve readability and human understanding. A typical example of this is reverse engineering a relational database into an Entity Relationship (ER) diagram. An ER diagram expresses structures such as entities and relationships that help a database administrator understand the connections between data items that may be difficult to visualize when all data is stored in tables as it is in the relational model.

1.1 Motivation

Traditionally, data has been stored in relational databases and data warehouses. Relational database management systems have drawbacks for Big Data. One such drawback stems from the strictly defined structure of relational database schemata. While this improves organization and human understanding when studying and querying, it can lead to performance issues at scale. Big Data is popularly characterized by the 5 V's: volume, velocity, variety, veracity, and value. The main factor that is limited by a defined structure is variety. This thesis addresses the variety aspect of Big Data.

NoSQL database management systems tend to be schemaless, or semi-structured. This allows data of many different types to be stored in the same database. No longer having a need to format data into a specific structure allows for a much larger scale of data to be stored with less effort. Data does not have to conform to a rigid structure, so new data formats can be more easily integrated without changing the original data. As a result, NoSQL database management systems are being adopted by many organizations and seeing rapid a growth in development. The database management ranking site db-engines.com lists 46 relational systems and 54 non-relational systems among the top 100 most popular database management systems [8].

The semi-structured nature of NoSQL databases can lead to problems of its own. For example, there may be a case where data records have different names for the same piece of information. It can also be difficult in general for a person to locate the right attribute names to query. Different types of NoSQL database management systems, such as graph-based or document-based, use different modeling formats for data. This can make it difficult to develop universal tools for users to understand the data contained in a database.

1.2 Contributions

Relational databases have tools and methods for reverse engineering schemas and providing visual models for people to reference. In contrast, tools for NoSQL database management systems are emerging but lack some features such as identifying sub-schema structures. Therefore, this research creates a tool for reverse engineering a visual model for document-based NoSQL database management systems to improve human understanding of the data contained in a document-based NoSQL database.

Chapter 2

Background and Related Work

This chapter discusses four main areas of background and related research. The first section discusses concepts that provide a foundation for the data models utilized for NoSQL database systems, particularly for forward and reverse engineering. The second section focuses on research that performs reverse engineering with NoSQL systems as well as tools that practitioners can use. The software tools are compared to our application in Chapter 5.

2.1 NoSQL

NoSQL database systems have emerged as an alternative to relational, or SQL, database systems. They have seen an increase in usage due to their flexibility and heterogeneity [9]. Traditionally, SQL database systems are table-based, and have a predefined schema for each data record. On the other hand, NoSQL, defined as “not only SQL,” is unstructured or semi-structured. This means that NoSQL database systems are able to have varying schemas, or different structures for each individual data item. However, NoSQL databases can follow a prescribed schema, but due to their flexibility, it can be difficult to enforce or model effectively. In spite of this drawback, NoSQL models are important when it comes to working with Big Data, as these systems are often the best equipped to handle the information with improved flexibility and scalability. In this case, adopting a NoSQL approach can help developers save time, as the data can evolve and adapt over a period of time.

There are also many different types of NoSQL database systems. Depending on their data model, they can be classified into a few main categories, such as document, graph, column, and key-value database systems. This section goes further into detail on document and graph databases.

2.1.1 Document

Document databases are characterized by data records, called documents, that represent individual objects. These objects may have a complex structure to store attributes and values [9]. One example of a document NoSQL database management system is MongoDB, which allows users to store and query document data.

Document data can also be represented as JSON, or JavaScript Object Notation, and a document database can be represented as an array of JSON objects. As with documents, JSON objects contain information about an object as a set of attributes, or properties, as well as their values. This format is a way to access document databases for applications.

Figure 2.1 shows an example of two documents with different structures.

```
[
    {
        "a": "a1",
        "b": "b1",
        "c": "c1",
        "d": "d1"
    },
    {
        "a": "a2",
        "b": "b2",
        "c": "c2",
        "e": "e2"
    }
]
```

Figure 2.1: Array containing document data with two structures

Union Schemas and Sub-schemas

Since each document can have its own structure, document databases are typically modeled using a *union schema* in existing tools such as Hackolade [10] and MongoDB Compass [11]. This format provides a list of every distinct attribute contained in all documents in the database. Table 2.1 shows an example of a union schema for the documents shown in Figure 2.1. In addition to the attribute names and types, union schemas may also include other information for each attribute, such as whether or not the attribute is present in every document in the database, or how many times the attribute appears in the database. Union schemas are useful to model document databases that are mostly uniform with slight variation.

Table 2.1: An Example of a Union Schema

Attribute	**Type**	**Required**
a	String	*
b	String	*
c	String	*
d	String	
e	String	

While the information provided by a union schema is useful to look at the overall structure of a document database, it does not account for the individual structures of documents. Therefore, we consider the *sub-schemas,* or the structure of individual documents when constructing a visual model. Table 2.2 shows an example of a sub-schema profile for the documents shown in Figure 2.1. The key difference between between a union schema and a sub-schema model is that a union schema is a single schema that includes every attribute in the database, while the sub-schema model provides a list of the entire structure of every document in the database. This model is useful for document databases with a high amount of variation, and allows a viewer to see the structures in which documents in the database appear as.

Table 2.2: An Example of a Sub-schema Profile

Sub-schema	**Attributes**	**Types**
1	a	String
	b	String
	c	String
	d	String
2	a	String
	b	String
	c	String
	e	String

2.1.2 Graph

Graph databases structure data in a different way than document databases. The documents contained in document databases are treated as individual objects with their own structures, but the data items in a graph database are treated as nodes interconnected by edges. This allows the database to depict the way that data is connected through relationships between nodes.

Figure 2.2 shows an example of a graph database model created by Czerepicki [1]. It depicts a set of nodes labeled *W1-Wn*, and a set of edges labeled *K1-Km*. Node *Wn* has attributes *A1-Ap*. There is a directed relationship from node *W2* to node *W1*, shown as the edge *K1*. *K1* has attributes *A1-Ar*. Node *W3* has a directed relationship, labeled *K5*, with itself. Node *W4* has no relationships with any other node, showing that graph databases do not have to be connected.

2.1.3 Heterogeneity

One consequence of databases containing semi-structured data is heterogeneity within the data. This means that heterogeneous, or mixed data with different schemas or structures can be stored in the same database. One strategy used to query heterogeneous data is a query rewriting approach. This section discusses approaches to query rewriting for both document databases and graph databases. Research from each of the approaches is illustrated below.

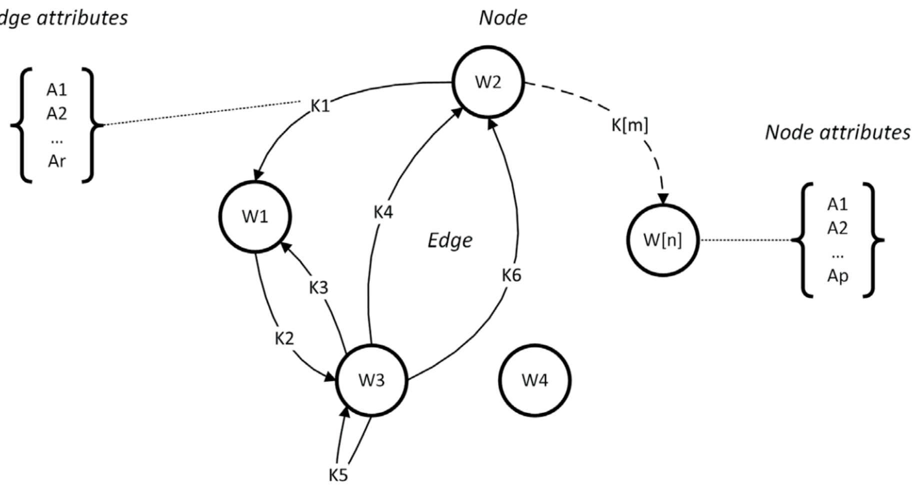


Figure 2.2: Graph Database Model [1]

EasyQ

Document-based NoSQL database management systems contain data that are semi-structured and therefore consists of heterogeneous documents. Querying a database with varying structures requires a user to be familiar with all possible structures for their desired information, or a query processing system that can account for the variations automatically. Research by Hamadou *et al.* proposes an approach for schema-independent queries that are designed for documents with multiple structures [2]. They design the EasyQ, or Easy Query tool. Their approach focuses on JSON-based documents in the MongoDB database management system (DBMS). As an example, they query a series of documents describing information on movies shown in Figure 2.3. Some attributes have values that are relatively constant, such as movie_title. Other attributes, such as language, have values that are not necessarily structured the same way. Document *d1* has an attribute for language, document *d2* has an embedded object called details that contains an attribute for the language, document *d3* does not contain information about the language, and document *d4* has an array of objects called versions that contains the language across multiple releases. Therefore, executing a query on the language attribute would yield incomplete results due to the heterogeneous structure of the documents.

They propose a new mechanism for querying over heterogeneous structures. One of the main components of EasyQ is a dictionary of all possible paths to any given field in the database. There is also a rewriting module that automatically changes the query based on the paths of specified fields. When compared to basic queries, the rewritten queries perform worse on average. This overhead is acceptable when compared to the effort needed to perform the separate queries to retrieve all structures of an attribute [2].

Figure 2.4 shows how a user query is processed by EasyQ. The Figure shows a data structure extractor that updates the dictionary with all paths to the same attributes. The

```
d1: {
        "movie_title":"Fast and furious",
        "year":2017,
        "language":"English"
    },
d2: {
      "movie_title": "Titanic",
      "details":
        {"year":1997,"language":"English"}
    },
d3: {
        "movie_title": "Despicable Me 3",
        "year":2017
    },
d4: {
        "movie_title": "The Hobbit",
        "versions":
        [{"year":2012, "language":"English"},
        {"year":2013, "language":"French"}]
    }
```

Figure 2.3: Unstructured or Semi-structured Document Data [2]

user then enters a query that passes through the query rewriting engine. This engine uses the dictionary to modify the query to include the different paths. The new query is sent to the database and returns the documents to the user.

EasyGraphQuery

Malki *et al.* propose a solution to the complex querying of heterogeneous data in graph-oriented NoSQL databases [3]. The proposed solution utilizes EasyGraphQuery with the Neo4j DBMS. EasyGraphQuery allows users input their query, and the system rewrites it using a dictionary to extract similar attributes.

Figure 2.5 shows the architecture of the EasyGraphQuery system. There is a data structure extractor that is used to populate a dictionary with similar attributes. The dictionary is populated using two similarity matrices that calculate attributes that are syntactically and semantically similar. When a user inputs a query, the query rewriting engine manipulates the query to include similar attributes. This updated query is sent to the database, and the results are returned to the user.

The proposed EasyGraphQuery solution allows for queries to return a more complete data set for semi-structured graph data. Malki *et al.* only considers the structural heterogeneity

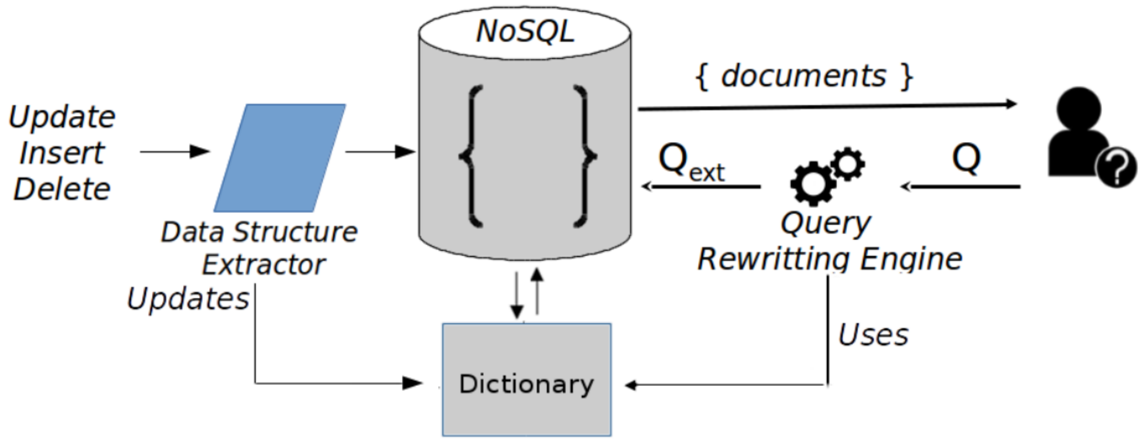


Figure 2.4: Overview of EasyQ's Query Processing [2]

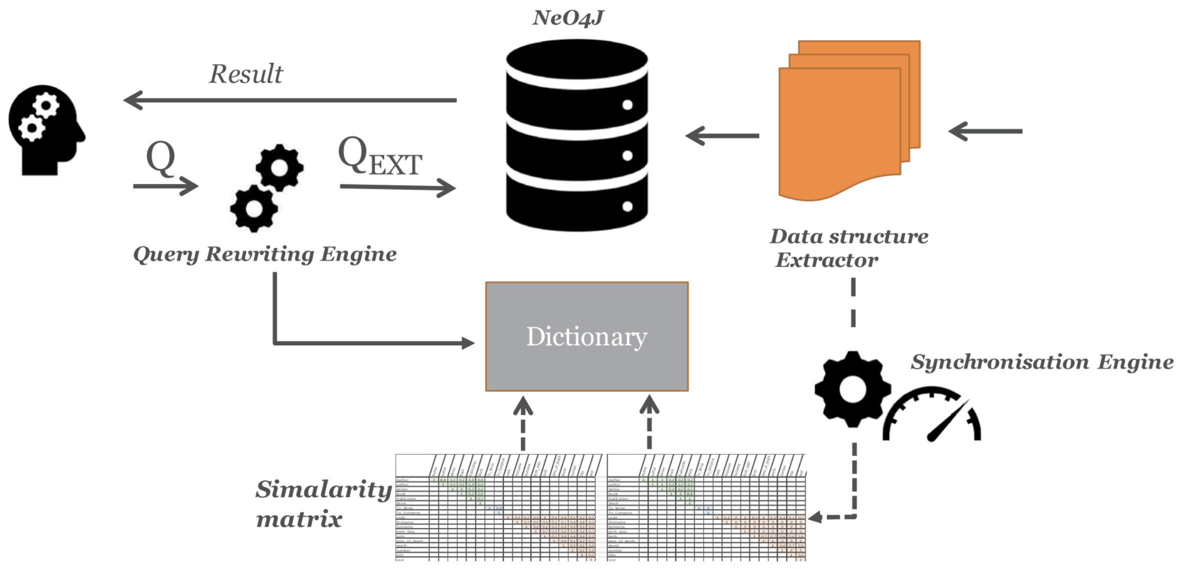


Figure 2.5: Architecture of the EasyGraphQuery System [3]

of the attributes. Future work includes extending their solutions to include more aspects of semantic heterogeneity.

EasyQ and EasyGraph Query have similar approaches to rewriting queries to return a more complete answer from NoSQL databases. They both extract the structure of data into a dictionary that is used by a query rewriting engine to manipulate a query from a user. However, EasyQ focuses on locating different paths to an attribute, while EasyGraphQuery focuses on finding similar attributes.

In our research, we focus on the extraction of the structures in document databases into a descriptive visual model. This allows users to gain knowledge of the structures in the database, providing another approach to locating similar attributes and/or structures in NoSQL document data.

2.2 Forward Engineering

Forward engineering describes the process of converting a database from a high level, abstract model into a physical model that can be used to store data. For relational databases, forward engineering typically takes the form of translating a visual model such as the Entity-Relationship model or UML class diagrams into relational tables. However, this process can be more difficult with semi-structured or unstructured NoSQL database models because the data does not have to follow a strict structure, so it is difficult to account for the heterogeneity of these databases. This section outlines research for forward engineering of NoSQL databases. We investigate forward engineering as a basis for abstraction in NoSQL database visual models and to capture design concepts that can be implemented into an application.

UMLtoGraphDB

UMLtoGraphDB is a system used to transform a UML class diagram into a graph database [4]. With a Model Driven Architecture (MDA) approach the system generates java code to access, update, and verify graph databases from a conceptual schema. It refines platform-independent models to platform specific models.

An overview of the UMLtoGraphDB infrastructure is shown in Figure 2.6. The input to the process is a UML class diagram and a set of constraints. The UML diagram is transformed into a GraphDB model and the constraints are transformed into a Gremlin model. The Gremlin model and the GraphDB model are used to create a physical graph database.

UML to NoSQL

Abdelhedi *et al.* propose UMLtoNoSQL, a system to automatically transform a UML conceptual model into a NoSQL model using an MDA approach [12]. To facilitate the process from a UML conceptual model into a NoSQL physical model, they introduce a platform-independent logical level that is between conceptual and physical. This generic logical model allows for mapping to multiple different NoSQL platforms. The process is divided into two main steps: (1) convert the UML conceptual model into a generic logical model, and (2) convert the generic logical model into a physical model.

The second part of the UMLtoNoSQL transformation algorithm allows for the choice of document, column, or graph database target models [13]. The generic logical model describes data in terms of common NoSQL features to allow for the choice in physical model, and

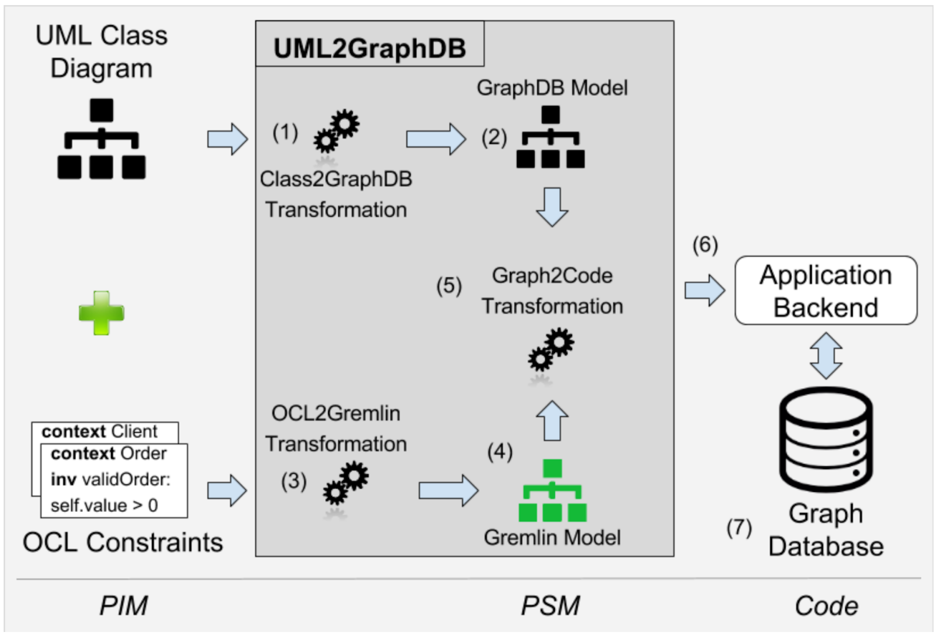


Figure 2.6: Overview of the UMLtoGraphDB Infrastructure [4]

abstracts the technical details so that the logical level remains platform independent. They describe sets of rules for transforming from the generic logical model to the physical models and demonstrate the process for three different NoSQL systems: MongoDB (document), Neo4j (graph), and Cassandra (column). Since the NoSQL database types of the three chosen systems represent data differently, the algorithms to transform the generic logical model into the selected physical model are different from one another.

Daniel *et al.* also propose an approach for mapping UML conceptual models to NoSQL databases called UMLto[No]SQL [5]. Their approach allows a UML class diagram to be mapped to a relational, graph, or document database. To facilitate the mapping of a UML class diagram to a document or graph NoSQL database, they create an intermediate metamodel for each database type.

The DocumentDB metamodel, shown in Figure 2.7, shows information about the document database in the form of a diagram. The database contains a collection of documents, and the documents contain fields. The fields have types, which can be a collection type, a primitive type, or a document type.

The GraphDB metamodel, shown in Figure 2.8, shows information about the graph database in the form of a diagram. This diagram shows that graph databases consist of vertices and edges. Both vertices and edges may have properties and primitive types.

For document databases, the mapping algorithm consists of the following steps [5]. Each document region is mapped to a document database. Each class is mapped to a collection. This produces a collection for each superclass. Each class is mapped to a document schema and its containing collection is set to the mapped collection of the top-level element inheritance hierarchy. This means that classes in the same inheritance hierarchy are contained

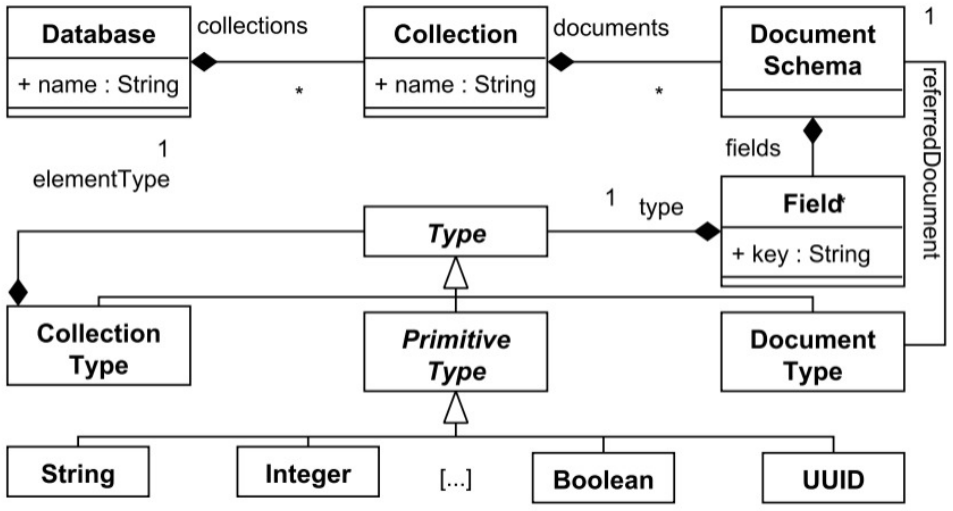


Figure 2.7: DocumentDB Metamodel [5]

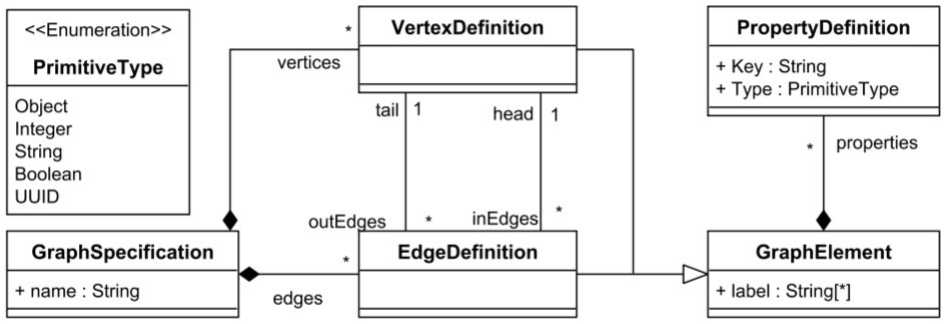


Figure 2.8: GraphDB Metamodel [5]

in the same collection. Each attribute is mapped to a field where the key is the attribute name, the type is the attribute type, and added to the property list of mapped containers, multi-valued attributes are mapped using the collection type. Each association between two classes is mapped to two fields that represent the classes involved in the association and sets the type as a document referring to the other class. Each association between two classes in heterogeneous data stores is mapped to a field. Each association class between classes is mapped to a document with an attribute corresponding to each of the documents involved in that association.

For graph databases, the algorithm consists of the following steps [5]. Each graph region is mapped to a graph specification based on class attribute pairs. Each class is mapped to a vertex definition where the node label is the class name and the name of its parent classes, and is uniquely identified by a set of properties. The set of properties is unique and is used for cross-datastore associations. Each attribute of the class and its parent classes are mapped to a property definition. Each name is mapped to a key, a type to its type, and added to the list of properties. Each association is mapped to an edge definition, where the label is the name, and the tail and head are the vertex definitions of the two associated classes. This also creates edge definitions to support associations involving the parents of each class. Each association is mapped to a property where the key is the association name, the type is the UUID, and add to the property list of its mapped container.

For mapping from UML to NoSQL, we observe two different approaches. The approach proposed by Abdelhedi *et al.* [13] (UMLtoNoSQL) uses an MDA approach to transform the UML conceptual model into a platform-independent logical model. Depending on the type of the target NoSQL database, a specific set of transformations is used to forward engineer the logical model into a physical model. The approach proposed by Daniel *et al.* (UMLto[No]SQL) uses multiple logical models and sets of rules to define the transformations from a UML conceptual model to a physical model. Their approach transforms a UML class diagram into a logical metamodel that is specific to the type of NoSQL database. The metamodel is forward engineered into a physical model using the transformation algorithm that matches the metamodel's type.

ModelDrivenGuide

Mali *et al.* create a guide to deciding which SQL/NoSQL system should be implemented [6]. ModelDrivenGuide introduces an approach that facilitates the transformation of databases. They investigate methods for transforming a relational database into a NoSQL database and also for transforming a conceptual model (ER) to a NoSQL database. They use an MDA approach with 3 types of models: A platform independent conceptual model, a platform independent common logical model (PIM), and a platform specific physical model (PSM).

A visual guide to these transformations is shown in Figure 2.9. The PIM, or Platform Independent Model, is split into two transformations: it starts as a traditional UML diagram. The second level, PIM2, is the 5Families metamodel (relational, document, graph, key-value, and column). This model describes how one logical model can be adapted into another using a set of refinement rules. The PSM, or Platform Specific Model, is obtained by transforming the compatible PIM2 model.

Forward engineering can be used to help to gain an understanding of transformations from higher level models to lower level models. All of the approaches covered in this section made use of a logical intermediate model to facilitate the transformation from a conceptual model to a physical model. We investigated the formal models used in the forward engineering approaches with the intention of adopting one conceptual model as an output model for our application, but we decided to create our own custom output model because none of the conceptual or logical models include example data. The next section discusses work related to the development of a tool to generate a schema profile.

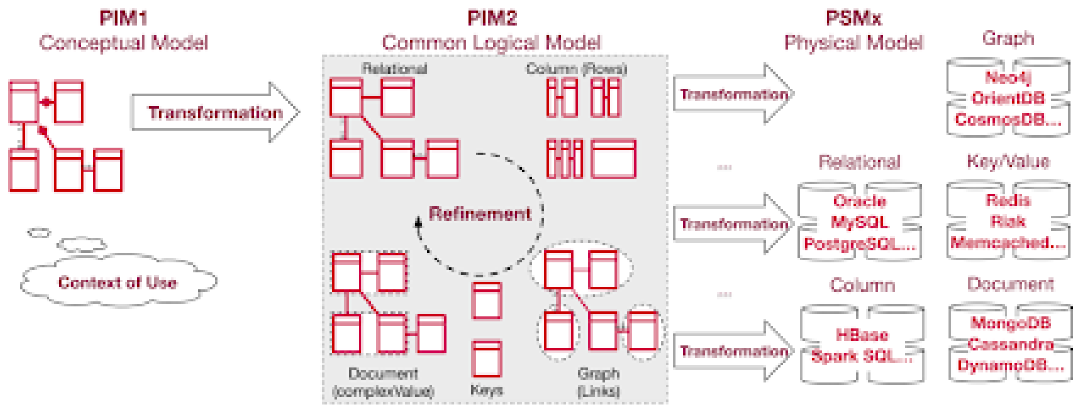


Figure 2.9: ModelDrivenGuide: From Conceptual to Physical Model [6]

2.3 Related Work

This section discusses 3 main topic areas closely related to our work: reverse engineering of NoSQL databases, profiling document database schemas, and software tools used for creating and visualizing schema profiles. Reverse engineering refers to the process of abstracting a physical database into a higher level model. We observe strategies used to reverse engineer NoSQL databases to investigate possible methods of abstracting a document data base into a model to represent a schema profile. Schema profiling refers to the process of describing the schema of unstructured or semi-structured data in NoSQL databases. We investigate schema profiling to investigate strategies used to assist the development of an algorithm to generate a schema profile of a document database. Finally, we investigate current tools that have the ability to profile document databases in order to investigate current functionalities to find what areas need further development.

2.3.1 Reverse Engineering

Reverse engineering is a process of deriving a schema and structure from a given database. This is done to gain an understanding of the database in order to more adequately query and/or work with a database. One method of reverse engineering is to create a conceptual schema. Conceptual schemas are platform-independent models that capture design decisions and depict high-level objects and relationships between them. They usually have a visual (graphical) component. The input to the reverse engineering process can be an implemented database, and the output is a figure that illustrates the database structures and their interconnections. An example of reverse engineering is representing the data stored in a relational data base in an ER or UML class diagram.

Roundtrip Engineering

Roundtrip engineering (RTE) describes the process of both reverse and forward engineering databases, allowing for bi-directional transformations between a physical model and a conceptual model. Akoka *et al.* investigate homogeneous integration between the design and the implementation phases of an RTE process [14]. Their research demonstrates the feasibility of an approach to RTE with an illustrative scenario. They transform a custom conceptual model into a property graph model and describe the transformation from the property graph model into a physical database. This process can be used in the context of NoSQL database maintenance including anomaly detection and migrations to new platforms. Future research includes designing and implementing a maintenance expert module, a set of guiding rules, several transformation rule sets, trace models, and quality evaluation models.

This approach to roundtrip engineering offers insights into producing database models derived from both conceptual and physical models. We initially investigated the possibility of forward engineering a physical database as a feature of our application, but we decided to focus on reverse engineering and document databases rather than graph databases.

Reverse Engineering Approach

Abdelhedi *et al.* investigate reverse engineering of NoSQL databases into a conceptual model with high level, abstract semantics that are easily understood by humans [15]. This reverse engineering approach consists of a series of transformation algorithms that transform a document-oriented NoSQL physical model into a conceptual model. In an experiment converting a NoSQL system into a conceptual model, they conduct a case study using the Eclipse Modeling Framework (EMF). They create a source and target metamodel. Then they build an instance of the source model and test the transformations.

This approach to reverse engineering focuses on the development of a conceptual model that makes it easier for people to understand the data stored in a NoSQL database and assist in querying. In our application, we also focus on providing a high level model to understand the data stored in a document database, but we produce a custom visual model rather than a UML class diagram.

Extraction Process to a Conceptual Model

Brahim *et al.* investigate an extraction based approach to generate a conceptual model from a document database [16]. They propose an MDA-based approach called ToConceptualModel that extracts the attributes of a collection of documents and represents it as a conceptual model. The source is a platform-specific NoSQL physical model, and the target is a platform-independent conceptual model (UML class diagram). To accomplish this task, the researchers implement a set of Query-View-Transformation (QVT) rules.

The transformations to convert the physical model into a UML class diagram consist of four steps: (1) represent each document collection as a class, (2) transform each primitive field in the collection to an attribute and type in the UML class diagram with multi-typed attributes being shown in brackets, (3) each complex field that is not a DBRef (an attribute used to express a link between collections) is transformed into a class with a link to the collection class, and (4) each complex field that is a DBRef is transformed into a link between the two classes linked by the DBRef.

While the target output of our application is not a UML class diagram, the transformations described by Brahim *et al.* can be utilized in the generation of a schema profile by using a schema profile object instead of a UML class diagram to transform the attributes represented in the collection of documents. The next section focuses on research that involves schema profiling of databases.

2.3.2 Schema Profiling

Schema profiling describes the process of extracting a schema from a database. This process is similar to reverse engineering, but it focuses on describing the schema of a database. NoSQL databases are schemaless, so schema profiling is used to understand the structures of the data stored in a database. We investigate strategies used to generate a schema profile through this section of related works.

Variety-Aware Approach

Gallinucci *et al.* present an approach to querying on schemaless document data [17]. Their approach consists of four main stages to generate a query dependency graph: schema extraction, schema integration, functional dependency (FD) enrichment, and querying. Since our topic focuses on generating a schema profile, we investigate the schema extraction and schema integration stages.

The schema extraction stage identifies the set of distinct local schemas from a document database. Local schema refers to the schema of a particular document; we refer to it as a sub-schema. Gallinucci *et al.* describe variety in a document database as either inter-document variety, or intra-document variety. Inter-document variety refers to documents that have different fields. Intra-document variety refers to heterogeneous data that occurs within a document, such as a mixed array of embedded objects. The approach uses a tree structure to store the local schema of each document.

The schema integration stage generates a global schema by using mappings to the local schemas identified in the schema extraction phase. A global schema refers to a single comprehensive view of the collection of the attributes across all documents, which we refer to as a union schema. Gallinucci *et al.* take one local schema as the global schema, and iteratively examine each other local schema and update the global schema accordingly.

While our approach does not use a tree structure, we use the concepts of local schemas (sub-schemas) and a global schema (union schema) during the implementation of our application. Our application considers both inter-document variety by recognizing documents with different attributes and intra-document variety by profiling nested objects within documents.

Schema Profiling with a Decision Tree

Gallinucci *et al.* propose a decision tree based approach to generating a schema profile. They modify the C4.5 classification algorithm, an algorithm used to generate a decision tree developed by Quinlan [18], to generate a decision tree for a schema profile. The tree is generated through the creation of nodes to split data based on certain information. The leaves of the tree represent one or more schemas, and internal nodes refer to attributes. The edges are splits in the data and can be either value-based splits or schema-based splits, and refer to whether the schemas at the leaf nodes satisfy the condition labeled by the edge.

For value based splits, the values can be either numerical or categorical. For numerical, this split is based on the value and splits into two nodes: $x < value$ or $x \geq value$. For categorical values, child nodes and are based on $x = value$ or not. Figure 2.10 shows nodes that are split on the Boolean attribute *has medical school*.

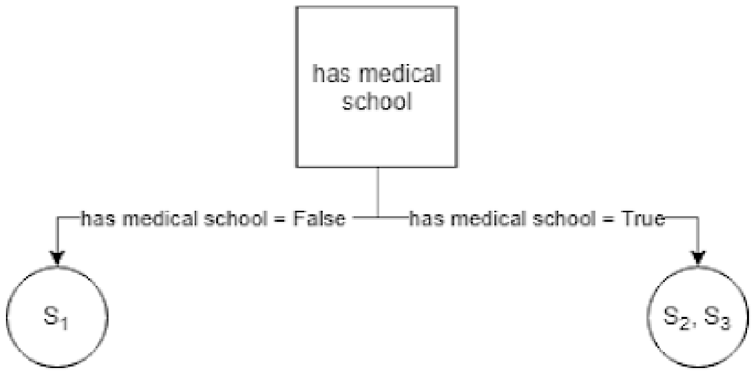


Figure 2.10: Value-based Split [7]

Schema-based splits have two child nodes based on whether or not the schema has a certain attribute. Figure 2.11 shows nodes that are split based on whether or not the schemas contain the attribute *medical school ranking*.

The process for creating the schema profile tends to result in leaves having only one schema; however, a schema can be repeated multiple times in different leaf nodes. This approach is interesting because it considers values as well as the structure of the documents to provide users with a profile of schemas and data within a document collection. For this research, we use a JSON object rather than a tree to store the schema profile and split data using only schema-based splits in order to generate a structural profile. Each of our sub-schemas is only represented once rather than multiple times. The next section covers tools related to profiling the schemas of document databases.

2.3.3 Schema Profiling Tools

Schema profiling tools are applications that allow users to input a document database and generate a schema profile detailing information about the data contained in the database.

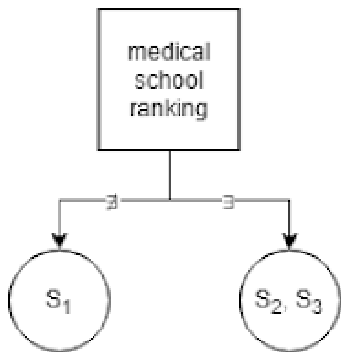


Figure 2.11: Schema-based Split [7]

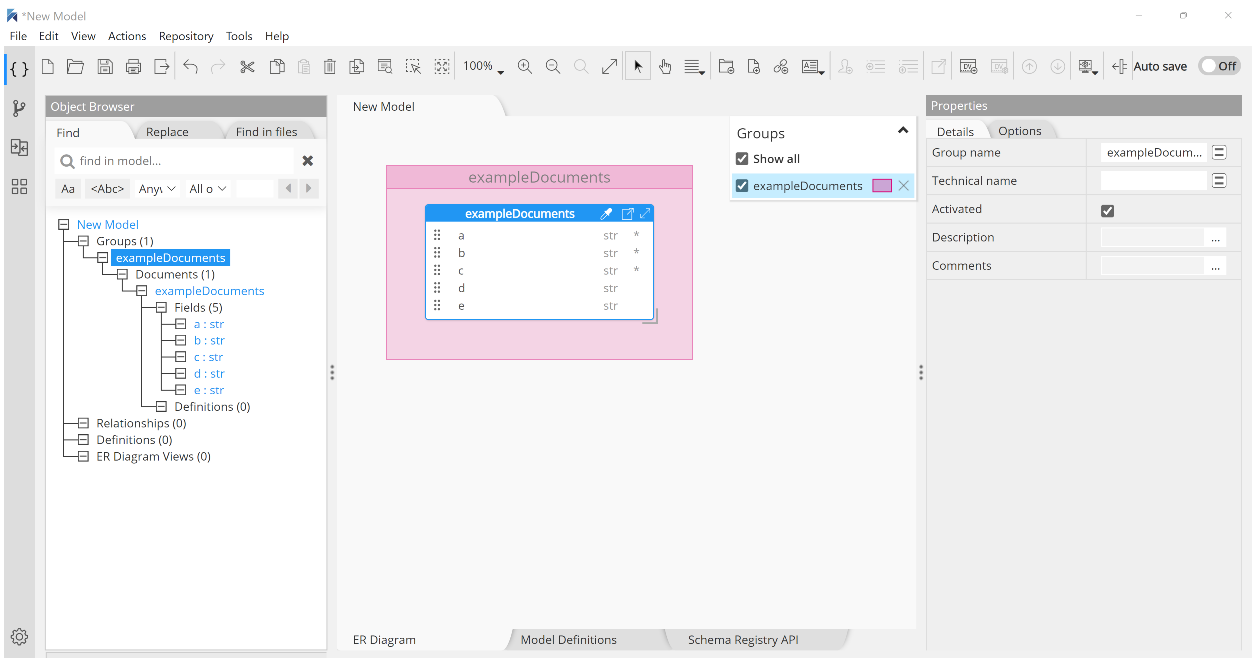
We investigate three applications: Hackolade, MongoDB Compass, and JSON Schema Generator.

Hackolade

Hackolade is a commercial tool that assists in the visualization of NoSQL data founded by Pascal Desmarets in 2016 [10]. This tool is designed for the purpose of providing a visual model for NoSQL data, and seeks to answer the question: “How do you query a database if you don’t know what fields exist that you can query on?” [19].

Its functionalities include creating a NoSQL model from scratch, importing a model, reverse engineering a model from a data set, and forward engineering a data set from a model. There are numerous different data source plugins for Hackolade to support many popular NoSQL models such as MongoDB, MariaDB, Firebase, Neo4j, BigQuery, DynamoDB, and many more. This allows it to be an effective tool that can be applied to many different data targets.

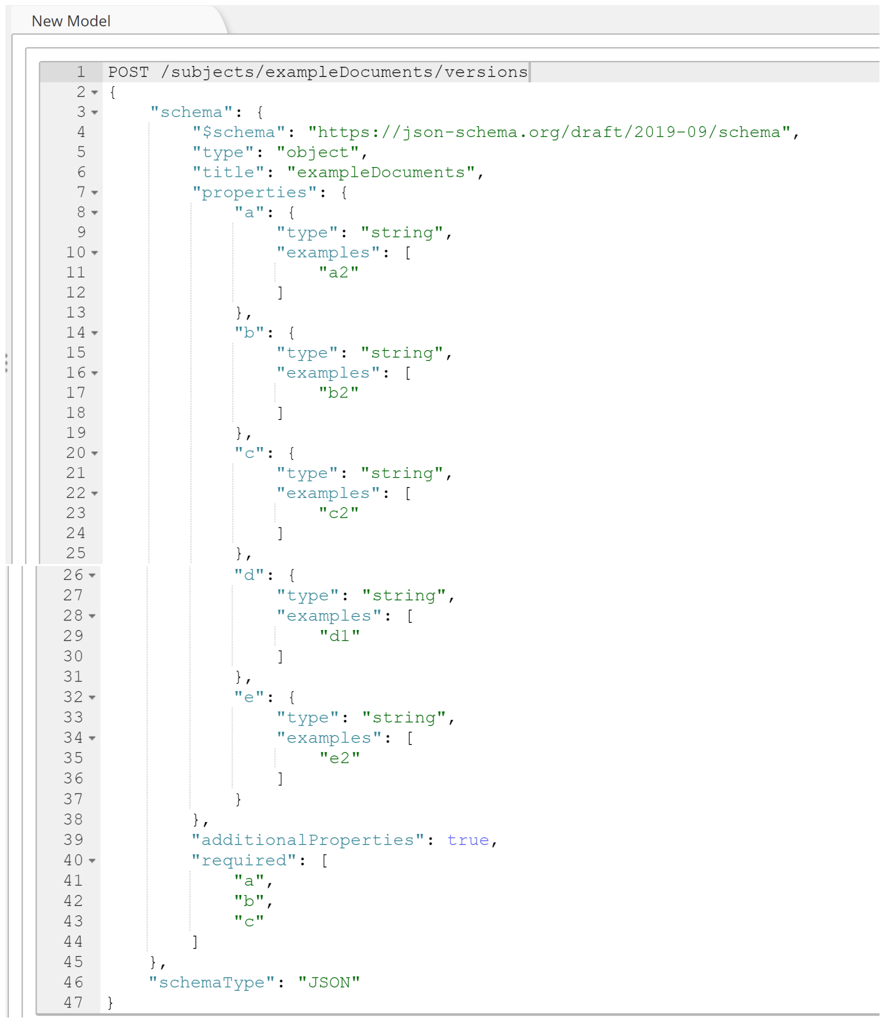
When used for reverse engineering a schema profile from a document database, it presents output in the form of an ER diagram. Figure 2.12 shows an example of the Hackolade interface when used to reverse engineer the sample documents shown in Figure 2.1. In the ER diagram interface, document databases are shown as pink squares, and collections of documents are shown as blue and white squares listing the attributes and types represented by documents in the collection. An asterisk is shown for each attribute that is required in the collection.



Group name	exampleDocum...
Technical name	
Activated	✓
Description	
Comments	

Figure 2.12: The Hackolade Interface with Example Data

Hackolade also provides the JSON Schema of the current model in the application. JSON Schema is a format used to describe the schema of document databases. Figure 2.13 shows the JSON Schema produced by Hackolade from the model shown in Figure 2.12 and presents the attributes in the database with their properties and example values as well as required attributes.


New Model

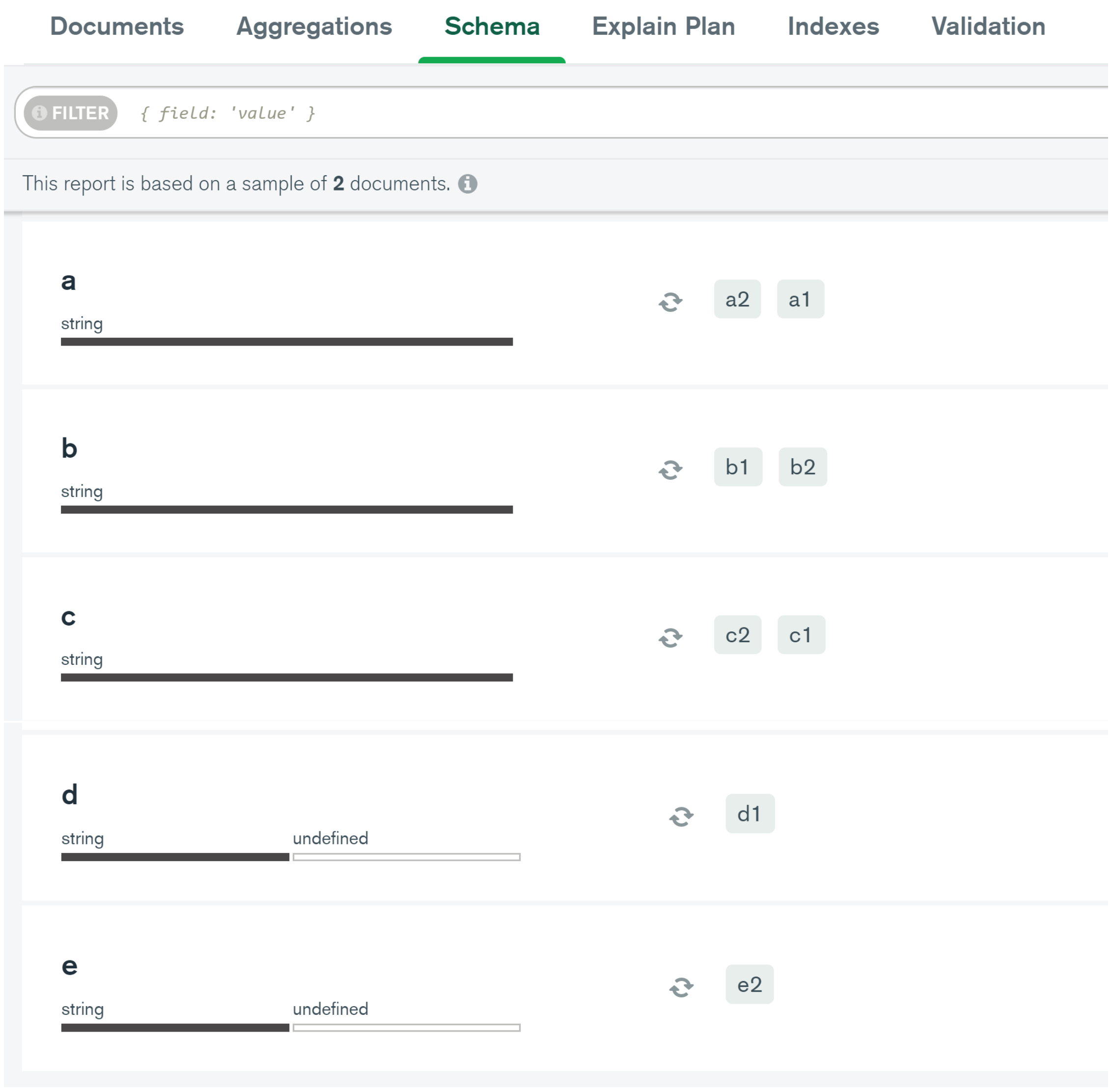
```
POST /subjects/exampleDocuments/versions
{
    "schema": {
        "$schema": "https://json-schema.org/draft/2019-09/schema",
        "type": "object",
        "title": "exampleDocuments",
        "properties": {
            "a": {
                "type": "string",
                "examples": [
                    "a2"
                ]
            },
            "b": {
                "type": "string",
                "examples": [
                    "b2"
                ]
            },
            "c": {
                "type": "string",
                "examples": [
                    "c2"
                ]
            },
            "d": {
                "type": "string",
                "examples": [
                    "d1"
                ]
            },
            "e": {
                "type": "string",
                "examples": [
                    "e2"
                ]
            }
        },
        "additionalProperties": true,
        "required": [
            "a",
            "b",
            "c"
        ]
    },
    "schemaType": "JSON"
}
```

Figure 2.13: JSON Schema Produced by Hackolade

MongoDB Compass

The MongoDB document database management system is the 5th most popular database management system as listed by DB-Engines and the top ranked non-relational database management system [8]. MongoDB Compass is an interactive tool for querying, optimizing, and analyzing data stored in a MongoDB [11] database. Its features include querying the documents in a database, query evaluation, schema analysis, and document validation rules. The schema analysis tool provides a visual union schema of the database and can provide useful insights into the information and structure of the data.

Figure 2.14 shows the output produced by the MongoDB Compass Schema Analyzer for the sample documents shown in Figure 2.1. This output shows each attribute in the database with a bar showing the distribution of types each attribute is represented as and example values of each attribute. Attributes that are not present in all documents are shown as the type undefined as a proportion of the bar matching the proportion of documents that the attribute is missing from. For example, the bar for the attributes d and e in Figure 2.14 are shown as half undefined since they are only present in 1 of 2 documents.



This report is based on a sample of **2** documents.

a
string
a2 a1

b
string
b1 b2

c
string
c2 c1

d
string undefined
d1

e
string undefined
e2

Figure 2.14: Union Schema Produced by the MongoDB Compass Schema Analyzer

JSON Schema Generator

JSON Schema Generator is an online web application used to generate JSON Schema from a JSON document database. JSON Schema is an Internet Draft Internet Engineering Task Force draft standard released in 2021, so it is currently not a JSON standard [20]. JSON Schema is a format used to define the structure of JSON document data used for the purpose of validation and documentation to ensure that all data follows a particular format. JSON Schema Generator receives input in the form of entered text containing a JSON document database and outputs a JSON Schema that describes the database's schema.

Figure 2.15 shows the output of JSON Schema Generator using the sample documents in Figure 2.1. While similar to the output shown in Figure 2.13 with Hackolade, there are a few differences. The JSON Schema produced by JSON Schema Generator lists every attribute as required, including d and e that are not present in both documents. However, the descriptions for d and e in the properties attribute of the JSON Schema include an empty default attribute that is missing from the other required attributes. While this can be slightly misleading, it allows the required attribute to be read as a union schema by displaying every attribute and interpreting an attribute with an empty default value as non-required.

In addition to the example values for each attribute shown in the properties attribute (also shown in the JSON Schema produced by Hackolade), JSON Schema Generator also provides examples of all of the documents in the database. However, the examples attribute is not used by JSON Schema for the purpose of validation. With large data sets, this attribute can become lengthy and reduce readability of the JSON Schema.

```
{
    "$schema": "https://json-schema.org/draft/2019-09/schema",
    "$id": "http://example.com/example.json",
    "type": "array",
    "default": [],
    "title": "Root Schema",
    "items": {
        "type": "object",
        "title": "A Schema",
        "required": [
            "a",
            "b",
            "c",
            "d",
            "e"
        ],
        "properties": {
            "a": {
                "type": "string",
                "title": "The a Schema",
                "examples": [
                    "a1",
                    "a2"
                ]
            },
            "b": {
                "type": "string",
                "title": "The b Schema",
                "examples": [
                    "b1",
                    "b2"
                ]
            },
            "c": {
                "type": "string",
                "title": "The c Schema",
                "examples": [
                    "c1",
                    "c2"
                ]
            },
            "d": {
                "type": "string",
                "default": "",
                "title": "The d Schema",
                "examples": [
                    "d1"
                ]
            },
            "e": {
                "type": "string",
                "default": "",
                "title": "The e Schema",
                "examples": [
                    "e2"
                ]
            }
        },
        "examples": [{
            "a": "a1",
            "b": "b1",
            "c": "c1",
            "d": "d1"
        },
        {
            "a": "a2",
            "b": "b2",
            "c": "c2",
            "e": "e2"
        }]
    },
    "examples": [
        [{
            "a": "a1",
            "b": "b1",
            "c": "c1",
            "d": "d1"
        },
        {
            "a": "a2",
            "b": "b2",
            "c": "c2",
            "e": "e2"
        }]
    ]
}
```

Figure 2.15: Example output produced by JSON Schema Generator

Hackolade produces a union schema that represents a comprehensive list of the attributes and types in a document database with an indication of which attributes are present in all documents. Likewise, the JSON Schema produced by Hackolade lists all attributes and types in the database and provides a list of the required attributes.

The MongoDB Compass Schema Analyzer provides a union schema of the document database with information about the proportion of documents that contain each attribute. It also provides sample values for each attribute that appear in the database.

JSON Schema Generator produces a JSON Schema from a given document database. The output produced by this application can be used to deduce a union schema and sub-schemas in a document database, but it requires an understanding of the output to interpret as a schema profile. The primary purpose of JSON Schema is for validation of data.

Among the related schema profiling tools, we observe the lack of a visual model that provides a schema profile that includes the sub-structures of a document database. Therefore, we seek to develop an application that produces an analysis of the sub-schemas of documents in the database including information about the proportion of documents and a list of sample documents that follow each sub-schema and provide more information about the attributes in the union schema of a document database, such as the number and percentage of documents each attribute appears in, and present the schema profile in a visual model.

The output produced by our schema profiling application, Schemalysis, is compared to the output produced by these three schema profiling tools in Chapter 5.

2.3.4 Comparison

In summary, we combine our findings from the related work into Table 2.3 to compare their contributions and functionalities. This table investigates 5 key insights about each related work.

1. Source Model - the initial input format for the approach.
2. Target Model - The result output format of the approach.
3. Algorithm - An algorithm in the form of pseudocode, description, sets of rules, *etc.* is included.
4. Implementation - a system or tool is discussed that implements the algorithm.
5. Union Schema - the approach outputs a schema profile including the union schema of the database.
6. Sub-schemas - the approach outputs a schema profile including the sub-schemas in the database.

Table 2.3: Comparison of Related Works

Reference	Source Model	Target Model	Algorithm	Implementation	Union Schema	Sub-Schemas
Roundtrip [14]	UML-like NoSQL Model	UML-like conceptual model				
Reverse Engineering Approach [21]	Document Database	UML Class Diagram	✓	✓		
Extraction Process [16]	Document Database	UML Class Diagram	✓	✓		
Variety-Aware Schema Extraction [17]	Document Database	Global Schema Mapping for Dependency Graph	✓	✓		
Decision Tree Schema Profile [7]	Document Database	Tree-based Schema Profile	✓	✓		
Hackolade [10]	JSON Schema or Document Database	ER Diagram, JSON Schema		✓	✓	
MongoDB Compass [11]	Document Database	Custom Visual Schema Profile		✓	✓	
JSON Schema Generator [22]	Document Database	JSON Schema		✓	✓	
Schemalysis	Document Database	Custom Visual Schema Profile	✓	✓	✓	✓

Table 2.3 shows that most of the reverse engineering and schema profiling approaches involve accepting a document database or physical model as input.

Most reverse engineering approaches output a UML class diagram. This type of model provides a depiction of the structure and the attributes of the data. However, we do not consider UML class diagrams to be a union schema or sub-schemas of the database since the purpose of UML class diagrams is to observe the structures and relationships between objects in object-oriented systems. A schema profile focuses on providing a comprehensive view of the attributes and structures that are in a single heterogeneous database.

Most schema profiling approaches output a schema profile or a visual model depicting a schema profile. While the other schema profiling tools covered in this paper focus on generating a union schema that encompasses all attributes represented in a database, Schemalysis creates a visual model to depict all sub-schemas represented in the database as well as the union schema.

Chapter 3 discusses an overview of the development process of our application.

Chapter 3

Research Overview

We seek to expand the ways in which NoSQL document databases can be visualized and analyzed. The structure of these databases are embedded in the semi-structured format of each document and need to be extracted in order to create a visualization. This chapter outlines the research questions, methodologies, and stopping criteria for each research task.

3.1 Research Questions

This research addresses three main questions to consider during the development of an application to provide a sub-schema analysis of a document database.

- RQ1: What reverse engineering techniques are applicable to document-based NoSQL data models and how can they be extended to create visual models?
- RQ2: What would an effective design be for a tool to support reverse engineering?
- RQ3: Does the model correctly and effectively evaluate unit tests and case study data sets to demonstrate functionality of the tool?

3.2 Methodology

This section provides a list of the tasks completed in pursuit of the research questions.

1. Develop a reverse engineering algorithm to derive a visual representation of a document database's schema profile. This algorithm is covered in Chapter 4.1 and outputs a schema profile which is used by the application to display the visual model.
2. Develop a tool to implement the algorithm. After the algorithm has been designed, we develop a web-based application that allows users to input a document database as a file or through text entry so that the algorithm extracts the schema profile and provides the visual model.
3. Perform an evaluation of effectiveness. After the tool has been created, it must then be tested to ensure correct functionality and effectiveness. These functional tests consist of unit tests to ensure correctness, integration tests to apply to a greater variety of data, and case studies to apply to real-world data.

3.3 Stopping Criteria

The stopping criteria for each task determines the point at which the task is considered complete.

Task 1, regarding the implementation of the reverse engineering algorithm, will be complete once we have developed an algorithm that successfully produces a schema profile from a document database that can be utilized to provide a visual model. This algorithm is explained in Section 4.1. **Task 2**, regarding the application, will be complete once a tool has been developed that successfully fulfills the requirements necessary for acceptance. These requirements are that it can: (1) receive input in the form of a document database file or entered as text, (2) extract the inherent structure of documents in the database, (3) output a visual model displaying the sub-schemas and the union schema of the database, and (4) display example instances of both the sub-schemas and the union schema. The application is described in Section 4.2. **Task 3**, regarding the validation of the application, will be complete once the tool has been assessed for correct functionality and comparison to related tools. The validation of the application is performed in Chapter 5.

3.4 Evaluation Methodology and Metrics

We evaluate the tool through the use of functional tests to manually verify that the requirements are met. These consist of unit tests to verify basic functionality, integration tests to verify that the application can handle a larger variety of data, and case studies to demonstrate functionality with real-world data. We also discuss the output produced by our tool and the output of related tools in order to compare and contrast functionalities between them.

Chapter 4

Schemalysis

This chapter covers the main contribution, a web application called Schemalysis, that reverse engineers a document database and provides the database's schema profile in an interactive visual model. This chapter introduces the algorithms used to extract the sub-schemas and union schemas of a document database, as well as the implementation and user interface of the application.

4.1 Algorithms for Schema Profiling

The algorithms used in the reverse engineering application capture a schema profile of the given document database. The input for the application is a document database, which is represented as an array of JSON documents. The output is a visual representation of the sub-schemas and union schema profile along with example instances of both.

We design a structure to store the generated schema profile as a JSON object. Table 4.1 shows the structure of the overall schema profile object. The names of the attributes are in the left column, and their respective types are in the right column.

The schema profile object contains three main attributes: the first one listed is schemas, which captures all of the sub-schemas in the database. It is represented by an array of objects, with each object in the array containing information about each sub-schema. The structure for the sub-schema objects is described in Section 4.1.1. The second object is union, which captures the union schema and is represented as an object. The structure of the union schema object is described in Section 4.1.2. Finally, the count attribute is an integer that captures the total number of items in the database.

Table 4.1: Structure of the Schema Profile Object

Schema Profile	
schemas	Array[object]
union	object
count	integer

The pseudocode shown in Figure 4.1 describes the overall algorithm steps performed by the application in the course of generating the schema profile. Step 1 reads the input from either manually entered text or an uploaded file, and Step 2 creates the empty profile object. Step 3, describing the generation of the sub-schema array, is further elaborated on in

Section 4.1.1, and Step 4, describing the generation of the union, is discussed in Section 4.1.2. Steps 5 and 6 store the completed schema profile so that the application can navigate the user to the results page and display the profile.

```
// Algorithm: Schemalysis Application

// This algorithm extracts a schema profile
//   from an array of JSON objects.

// Input:
// A document database
//   in the form of an array
//   of JSON objects

// Output:
// A visual profile of the
//   sub-schemas and union schema
//   of the database

1. Read document database from input file/text
2. Create empty schema profile JSON object and
   initialize count to zero
3. Extract sub-schemas into schemas attribute
   of schema profile
4. Extract union schema into union attribute
   of schema profile
5. Store schema profile and navigate to results page
6. Display results
```

Figure 4.1: Pseudocode Algorithm for the Overall Application

4.1.1 Extracting the Sub-Schemas

This subsection covers the generation of the sub-schemas component of the schema profile. Table 4.1 shows the sub-schema profile represented as an array of objects called schemas. Each object in the array represents a sub-schema of the database. Table 4.2 displays the structure of a single sub-schema object.

The sub-schema object has three main attributes: The first one is the schema. This is an embedded object whose attributes are the names of the attributes from the original object, and their values are a string representation of the type of the attribute, e.g., “string,” “number,” “boolean,” *etc.* as detected by the JavaScript typeOf() function. The items attribute is an array of objects that contains a sample of up to 10 original documents from the database that match the sub-schema. Finally, there is a count attribute, which is an integer that tracks the number of documents that conform to the sub-schema.

Table 4.2: Structure of a Sub-schema Object

Sub-Schema	
schema	object
items	Array[object]
count	integer

The overall approach to generating the sub-schema can be described in 3 steps: (1) pass through each document in the database and extract its individual schema, (2) compare the individual schema to all previously found schemas, and (3) construct an object that each unique schema.

The pseudocode shown in Figure 4.2 describes the algorithm used to extract the sub-schemas from documents in a database. This is how the array of objects is obtained for the schemas attribute for the schema profile object shown in Table 4.1.

```
// Algorithm: Sub-schema Extraction

// Input:
// An array of document objects

// Output:
// An array of sub-schema objects

1. Create an empty array 'SubArr' to
   store all sub-schema objects
2. Iterate through each document 'Doc' in the database:
     // handle schema profile's count attribute
     - increment the schema profile's total count by one
     // extract Doc's sub-schema
     - create empty schema object 'SchemaObj'
     - iterate through each attribute 'DocAtt' in Doc:
         - create attribute 'SchemaAtt' in SchemaObj
           with the same name as DocAtt
         - set the value of SchemaAtt to DocAtt's type
     // add SchemaObj to SubArr
     - iterate through each sub-schema object 'Sub'
       currently in SubArr:
         - compare Sub's schema to SchemaObj
             - check that both Sub's schema and SchemaObj have
               exactly the same attributes and the
               same types for each attribute
             - if Sub's schema matches SchemaObj:
                 - increment Sub's count by one
                 - add Doc to Sub's sample if the max
                   sample size has not yet been reached
                 - continue to next document in the database
     - if a match for SchemaObj was not found:
         - create a new sub-schema object 'NewSub'
         - initialize NewSub's schema to SchemaObj
         - initialize NewSub's count to one
         - create empty array for sample items
           and add Doc to it
         - add NewSub to SubArr
3. Return SubArr to schema profile object
```

Figure 4.2: Pseudocode Algorithm for the Sub-schema Extraction

4.1.2 Extracting the Union Schema

This subsection covers the details regarding the generation of the union schema. As shown in Table 4.1, this is represented as an embedded object in the schema profile object, shown in Table 4.3. The structure of the union schema is a set of key-value pairs, where the key is the name of each attribute in the database, and the value is an object that provides information about the attributes. The union schema object contains an embedded object for every attribute included in the database. The structure for the object of an attribute in the union schema is shown in Table 4.4. There are three attributes: The first one is required, which is a boolean indication on whether the attribute exists in every document in the database. The second is types, which is a string representation of every type of value the attribute has. This string is generated using a set data structure. This allows for the type of each attribute to be added to it from every document without the set containing duplicates. The final attribute is count, which is the number of documents that the attribute appears in.

The approach to generating the union schema consists of 5 main steps: (1) create the empty union schema object, (2) create an object for each attribute in the first document, and set initial values, with required set to true, (3) iterate through the rest of the documents, adding new attribute objects or updating existing ones as needed, (4) sort the type set of each attribute object and convert it into a string so that it can be displayed in the results, and finally (5) return the completed union schema object to the main schema profile object.

Table 4.3: Structure of the Union Schema Object

Union Schema	
attribute1	object
attribute2	object
...	...

Table 4.4: Structure of an Attribute in the Union Schema Object

Union Attribute	
required	boolean
types	string
count	integer

The pseudocode shown in Figure 4.3 describes the algorithm used to generate the union schema. This is how the data in the union attribute of the schema profile object shown in Figure 4.1 is obtained.

```
// Algorithm: Generate Union Schema

// Input:
// A document database

// Output:
// A union schema object

1. Create empty union schema object ‘Union’
2. Iterate through each attribute ‘FirstAtt’
   in the first document:
     - create empty attribute object ‘FirstAttObj’
       for FirstAtt
     - initialize FirstAttObj’s required
       attribute to true
     - initialize FirstAttObj’s count attribute to one
     - initialize FirstAttObj’s types attribute
       as an empty set object
     - add FirstAtt’s type to the set
3. Iterate through each subsequent document ‘Doc’:
     - iterate through each current attribute
       ‘CurrAtt’ in Union:
         - if CurrAtt does not exist in Doc:
             -set CurrAtt’s required attribute to false
     - iterate through each attribute ‘Att’ in Doc:
         - if Att already exists in Union:
             - increment its count attribute by one
             - add Att’s type to the set
         - if Att does not yet exist in Union:
             - create empty attribute object ‘AttObj’
             - initialize AttObj’s required attribute to false
             - initialize AttObj’s count attribute to one
             - initialize AttObj’s types attribute
               as an empty set object
             - add Att’s type to the set
4. Sort the types set of each attribute in Union
   and convert it into a string
5. Return Union to schema profile object
```

Figure 4.3: Pseudocode Algorithm for the Generation of the Union Schema

4.1.3 Time Complexity

We analyze the time complexity to perform the Schema Profile Algorithm here. The algorithm makes two passes through the databases: one to generate the sub-schema profile and one to generate the union schema.

The Sub-schema Extraction Algorithm

The Sub-schema Extraction Algorithm reads each data record in the database in order to account for every sub-schema present in the database. It also reads each attribute in each data record to generate the document's sub-schema. For n documents in the database, the sub-schema extraction component of the algorithm is linear, or $O(n)$, since time is based on the number of documents.

After extracting each document's sub-schema the Sub-schema Extraction Algorithm also makes a comparison to all other sub-schemas currently found in the database. For databases with less schema variety, this number is small. However, since the sub-schema of every document can potentially be different, resulting in a sub-schema array of size n. Therefore, the time complexity for comparing each sub-schema to the array of current sub-schemas is, at worst, linear $O(n)$.

Combining both the sub-schema extraction and comparison components of the Sub-schema Extraction Algorithm, the time complexity for a worst case database, where every document has a different sub-schema, is:

$$O(n^2)$$

Generate Union Schema Algorithm

Similarly, the Generate Union Schema Algorithm also reads each document in the database, as well as their attributes, in order to capture each attribute and the number of times they appear in the database. For n documents in the database, the union schema extraction component is linear, or $O(n)$, since time is based on the number of documents.

Upon extracting the attributes in each document, the algorithm adds the attributes to the union schema. To accomplish this, it must compare each attribute found in the document to the attributes currently in the union schema. For databases with less variation, the size of the union schema object remains relatively constant. However, each document in the database may have a completely different set of attributes. For n documents in the database with an average number of attributes a, the size of the union schema object would be $n * a$. Since the size of the union schema object is based on the documents in the schema, the time complexity to compare the attributes in each document to the attributes in the union schema is also linear $O(n)$.

Combining both the union schema extraction and comparison components of the Generate Union Schema Algorithm, the time complexity for a worst case database, where every document has a different set of attributes, is:

$$O(n^2)$$

Since the time complexity of both the Sub-schema Extraction Algorithm and the Generate Union Schema Algorithm is $O(n^2)$, the time complexity to perform the overall Schema Profile Algorithm is:

$$O(n^2)$$

4.2 Application

This section covers the implementation of the algorithms described in Section 4.1 as a working application. The implementation framework, including the languages and libraries used to construct the application, is described in Section 4.2.1. The main features and functionalities the application can perform are described in Section 4.2.2. Finally, Section 4.2.3 describes the user interface of the application.

4.2.1 Implementation

Schemalysis is implemented as a web application. Therefore, it is developed using a web page friendly environment. We develop the web pages in HTML and use of Bootstrap to handle the style of the interface.

The algorithms that perform the schema extraction and populate the schema profile object are implemented in JavaScript. This language was chosen for its simplicity and ease of integration into the web page. Additionally, jQuery is used in the JavaScript code to obtain and modify information on the web page interface.

4.2.2 Features

This section covers main features and functionalities that Schemalysis can perform. The list of features is shown in Figure 4.4 and is described in further detail in the rest of this section.

1. Document database input as either file upload or text entry
2. Option to profile lower level objects
3. Sub-schema analysis
   (a) Rank for each sub-schema, based on prevalence in database
   (b) Attributes represented in each sub-schema
   (c) The type of each attribute
   (d) The number of times the sub-schema appears in the database
   (e) The percent of the total documents in the database match the sub-schema
   (f) A sample of up to 10 items from the original database that match the sub-schema
4. Union schema analysis
   (a) Each attribute represented in the database
   (b) All types that each attribute appears as
   (c) Whether or not the attribute is required in the database
   (d) The number of documents the attribute appears in
   (e) The percentage of documents the attribute appears in
5. Interactive results page
6. Documentation describing how to use the application

Figure 4.4: Features of Schemalysis

1. Input

The application accepts input in either the form of a JSON file or manually entered text containing a document database. In both cases, the input must consist of an array, contain only JSON objects, contain at least one item, and maintain proper syntax and successfully parse into a document database, as described in the in-app documentation, partially shown in Figure 4.6. If it fails to successfully parse or load the data, the application stops.

2. Profiling Lower Level Objects

The user also has the option to profile lower level objects within the database, shown as a checkbox field under each input method in Figure 4.5. If this option is selected, then all embedded objects are profiled and their type is portrayed as a string representation of the embedded object created by recursively passing the embedded object into the same function that is profiling the original object.

Array attributes are further processed if the lower level option is selected. When an array is found, the algorithm constructs a set of the types of each item in the array in order to provide a more descriptive view into the information contained in an array.

Furthermore, any arrays found in embedded objects and embedded objects found in arrays are recursively processed until the lower level structures are fully described. However, for extensively complex objects, or deeply nested objects, this process can become lengthy. Therefore, if a user is not interested in lower level objects, and prefers a top-level description of attribute types, they can choose to ignore this option.

3. Sub-schema Analysis

Once a user has loaded a database into the application, they are presented with a sub-schema analysis of the structures of the data contained in the database. This is presented to the user in the form of a table. The table gives 6 key pieces of information about each sub-schema, and the content of the table can be manipulated through the user interface as described in Section 4.2.3.

The first piece of information shown for each sub-schema is its rank. This rank is a number assigned to the sub-schema based on its frequency in the database, with rank 1 being the most frequent. This stat is useful to determine the total number of sub-schemas present in the database, as well as view which ones are the most prevalent so that the user can take this into consideration when querying the database.

The next piece of information is a list of attributes that are present in the sub-schema. This list describes the structure of the sub-schema and the information that it contains.

Alongside the list of attributes, the types of each respective attribute are also shown. This information can be used to assist in querying the attributes in the database or to observe differences in the types of attributes between the sub-schemas.

Another piece of information that is shown for each sub-schema is the number of times that it occurs in the database. This stat helps a user see exactly how many times that documents in the database follow the structure of a given sub-schema.

Furthermore, the count is also used to provide the percentage of the database that follows the given sub-schema structure. This is done by dividing the count of the individual sub-schema with the count of the overall database. This stat helps a user conceptualize how often the sub-schema structure appears in comparison to all others.

Finally, each sub-schema in the analysis table contains a button that allows a user to view a sample of items from the original database that follows the sub-schema structure. This allows a user to see a sample of the data in the database. Sample data can also be viewed in outlier sub-schemas that could be used to locate documents to potentially correct or delete.

4. Union Schema Analysis

In addition to the analysis on sub-schemas, Schemalysis provides an analysis on the overall union schema of the document database. Compared to a sub-schema, a union schema is a single schema that contains all attributes that are present in any object in the entire database. However, Schemalysis also provides additional information about the union schema that allows users to conceptualize the data contained in the database. The layout, format, and examples of the union schema are described in Section 4.2.3.

The union schema is displayed in a web page table and contains a row for each attribute with 5 main points of information. The first is the name of the attribute. The second is a list of all types or structures that the attribute takes within the database. If there are attributes that have the same name but appear as different types or contain variation in their lower level structures, they will all be present in this column. Next is an indicator on whether or not the attribute is a required field, *i.e.,* currently present in every document in the database.

The final two components are the number of documents that the attribute appears in, and the percentage of documents the attribute appears in. These two points of information allow a user gauge how prevalent a particular attribute is in a database for the purpose of performing queries.

5. Interactive Results Page

The interface in which a user views the output of the application is structured in an interactive table. A user is able to manipulate the information shown through the use of inputs and buttons. This is described in greater detail in Section 4.2.3.

6. Documentation

There is in-app documentation that a user can view to understand how to use the application. It can be accessed by clicking on the Instructions button on the home page shown in Figure 4.5, and the documentation is shown in Figure 4.6.

The purpose of the documentation is to help guide a user who is new to Schemalysis. It provides guidelines and examples on the accepted inputs, as well as descriptions and examples of outputs of both the sub-schema and union schema analyses and how they can be interpreted.

4.2.3 User Interface

This section describes the interface of Schemalysis and how a user would navigate it during use.

Home Page

The first page that the user sees is the homepage. This page consists of a title and description to introduce the user to the app, and navigation for the user to view an About page, to read about the origin of the application, or an Instructions page to look into the documentation of the use of the application. Additionally, this page also contains the input fields in which a user can upload the data that they want to profile, select the option to profile lower level objects if they wish, and finally run the application with the Upload button. An image showing this page is shown in Figure 4.5 and an image showing the documentation is shown in Figure 4.6.

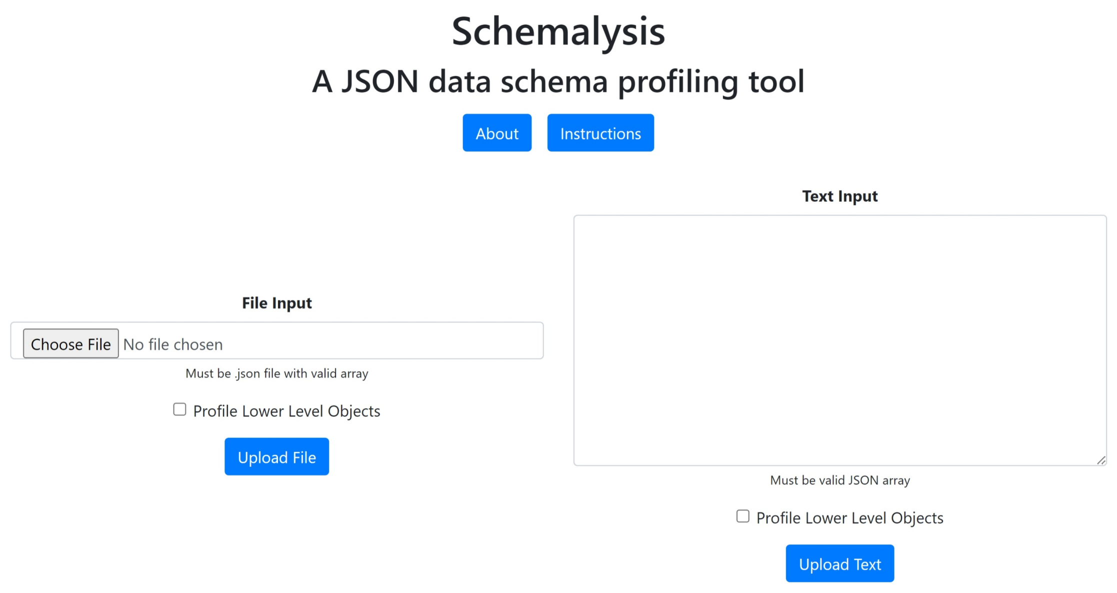

Schemalysis
A JSON data schema profiling tool
About
Instructions
Text Input
File Input
Choose File
No file chosen
Must be .json file with valid array
Profile Lower Level Objects
Upload File
Must be valid JSON array
Profile Lower Level Objects
Upload Text

Figure 4.5: The Home Page of Schemalysis

Sub-Schema Analysis

After the user has entered a document database and run the application, the results page is shown. This is the second main page of Schemalysis. This page is designed to display the results of the sub-schemas from the database, as well as allow the user to manipulate which sub-schemas are shown and view other information about the database and each sub-schema. A header showing the interface of the results page is shown in Figure 4.7, while examples of full output are shown in Chapter 5 in Sections 5.2.1, 5.2.2, and 5.2.3.

As shown in Figure 4.7, the results page shows several key points of information about each sub-schema found, as well as different options to allow the user to customize the information shown. This is done by altering the input fields or clicking the buttons.

The results are shown based on the rank of the sub-schema, which is a number given to it based on how many times the sub-schema occurs, with Rank 1 being the most prevalent sub-schema. By default, Schemalysis shows up to the top 10 sub-schemas by rank (Figure 4.7). Users can also click the Show All button to show every sub-schema in the database (Figure 4.8), the Bottom Ten button to view up to ten of the lowest ranked sub-schemas in reverse order (Figure 4.9), or the Top Ten button to return to showing the top 10 ranked sub-schemas.

Additionally, users can enter a custom range of sub-schemas they would like to see. This can be done by entering the first rank you would like to see in the field labeled Start and the last rank the user would like to see in the field labeled End: resulting in a custom view of the sub-schemas from the entered start and end range, inclusive, after clicking on the Update button. To check which ranks are being shown and the total number of sub-schemas present, a status message appears between the buttons and the start of the table. This message is shown in the form "Showing results X-Y of Z" with X being the start point, Y being the end point, and Z being the total number of sub-schemas present. An image displaying a custom range of sub-schemas from 3 to 7 is shown in Figure 4.10.

If the user enters values for the start and end points that are not in bounds (start value less than 1, or end value greater than the number of sub-schemas), the values are automatically adjusted to the minimum value for the start point or the maximum value for the end point. However, if the user enters invalid inputs for the start and end point (the start value is greater than the end value) the status message will show "Error: Invalid start and end points," and no results will be displayed, as shown in Figure 4.11.

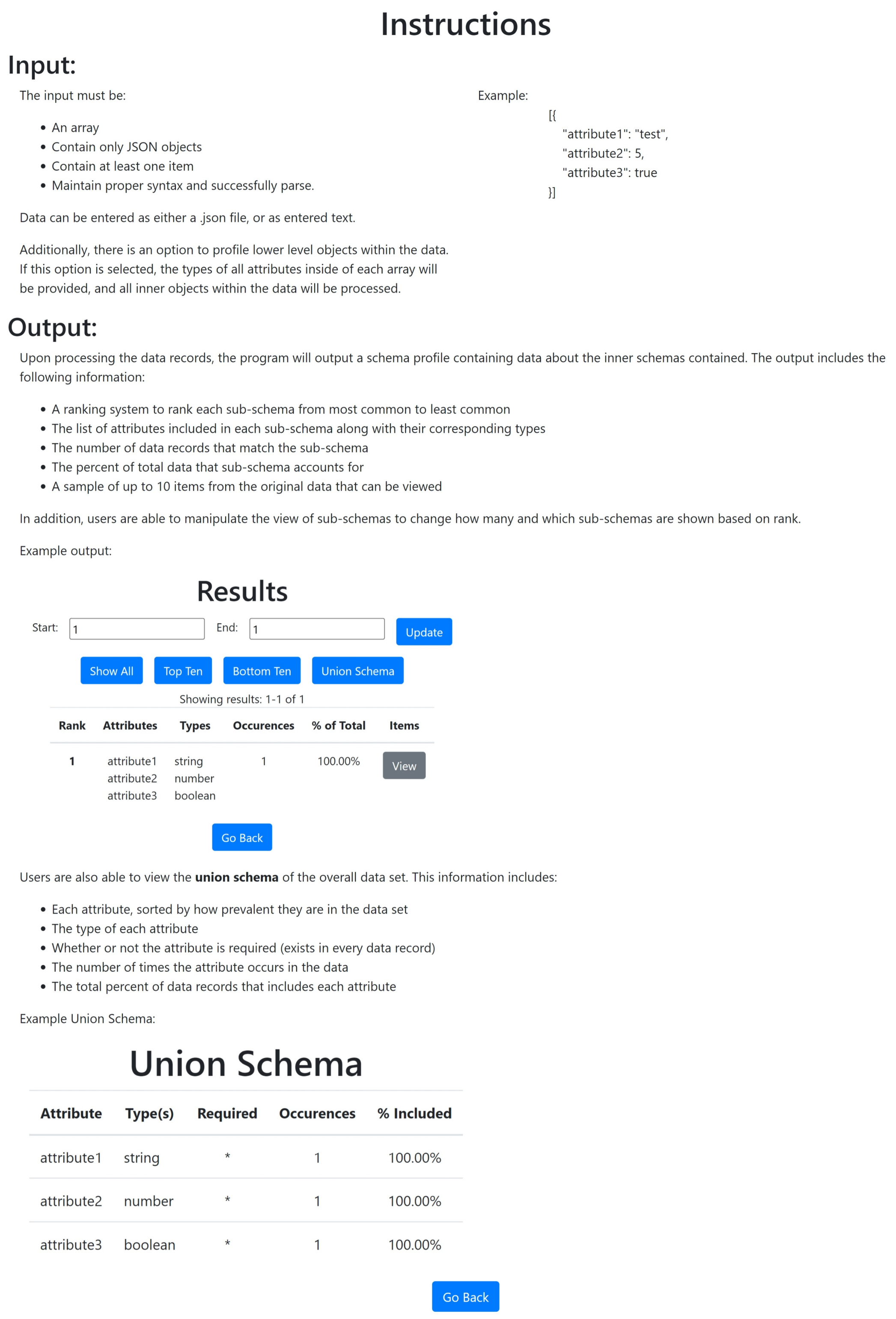
As the last point of information in table, item 3.f of the feature list, shown in Figure 4.4, describes the ability for a user to view a sample of up to 10 original items from the database for each sub-schema shown. On the user interface, this can be accessed by using a button that appears in every row of the table of sub-schemas. The location of this button is specified in Figure 4.12. Clicking this button opens up a new page which shows the rank of the sub-schema whose items are being viewed and a numbered list of the items contained in the sample. An example of this page showing example documents from Case Study 3 in Section 5.2.3 is shown in Figure 4.13.

Finally, at the bottom of the results page, there is a button labeled Go Back. Clicking this button returns the user to the home page so that they can restart the application with different input or options.

Union Schema

The user can access the union schema of the database by clicking on the Union Schema button on the results page. This opens the union schema on a new page. This page consists of a table that contains the information listed in part 3 in Figure 4.4. An example of the union schema is shown in Figure 4.14, and is also shown in Chapter 5 in sections 5.2.2 and 5.2.3.

The application described in this chapter is validated in Chapter 5.



Input:

The input must be:

- An array
- Contain only JSON objects
- Contain at least one item
- Maintain proper syntax and successfully parse.

Example:

```
[{
    "attribute1": "test",
    "attribute2": 5,
    "attribute3": true
}]
```

Data can be entered as either a .json file, or as entered text.

Additionally, there is an option to profile lower level objects within the data. If this option is selected, the types of all attributes inside of each array will be provided, and all inner objects within the data will be processed.

Output:

Upon processing the data records, the program will output a schema profile containing data about the inner schemas contained. The output includes the following information:

- A ranking system to rank each sub-schema from most common to least common
- The list of attributes included in each sub-schema along with their corresponding types
- The number of data records that match the sub-schema
- The percent of total data that sub-schema accounts for
- A sample of up to 10 items from the original data that can be viewed

In addition, users are able to manipulate the view of sub-schemas to change how many and which sub-schemas are shown based on rank.

Example output:

Results

Start: 1 End: 1

Showing results: 1-1 of 1

Rank	Attributes	Types	Occurences	% of Total	Items
1	attribute1 attribute2 attribute3	string number boolean	1	100.00%	View

Users are also able to view the **union schema** of the overall data set. This information includes:

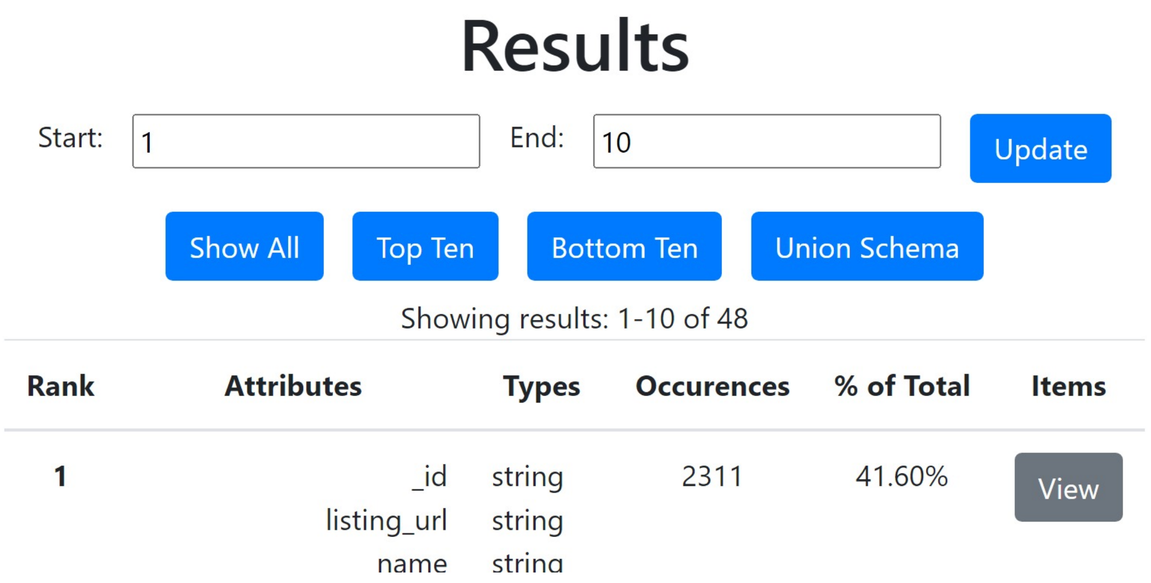
- Each attribute, sorted by how prevalent they are in the data set
- The type of each attribute
- Whether or not the attribute is required (exists in every data record)
- The number of times the attribute occurs in the data
- The total percent of data records that includes each attribute

Example Union Schema:

Union Schema

Attribute	Type(s)	Required	Occurences	% Included
attribute1	string	*	1	100.00%
attribute2	number	*	1	100.00%
attribute3	boolean	*	1	100.00%

Figure 4.6: In-app Documentation

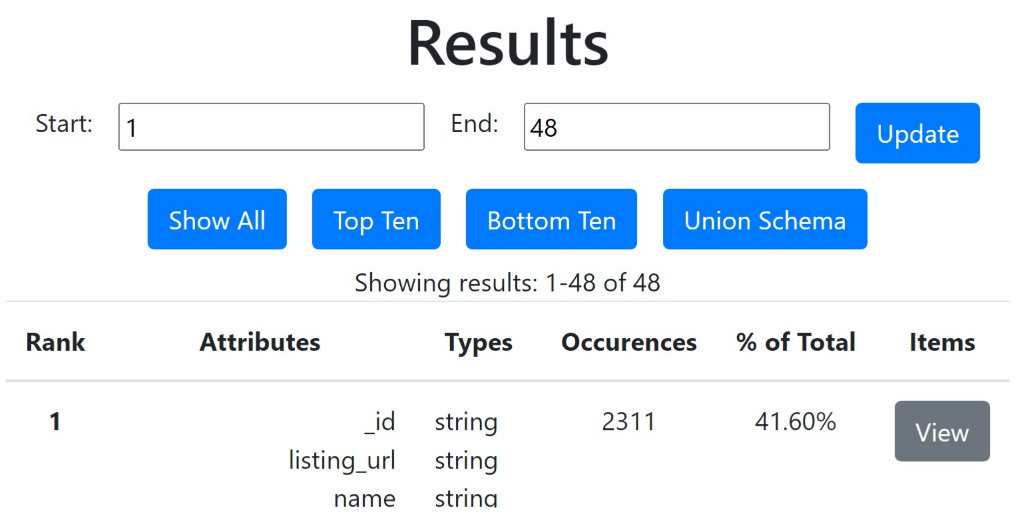

Results

Start: 1 End: 10

Showing results: 1-10 of 48

Rank	Attributes	Types	Occurences	% of Total	Items
1	_id listing_url name	string string string	2311	41.60%	View

Figure 4.7: Options on the Results Page, Showing Default Top 10 Sub-schemas.

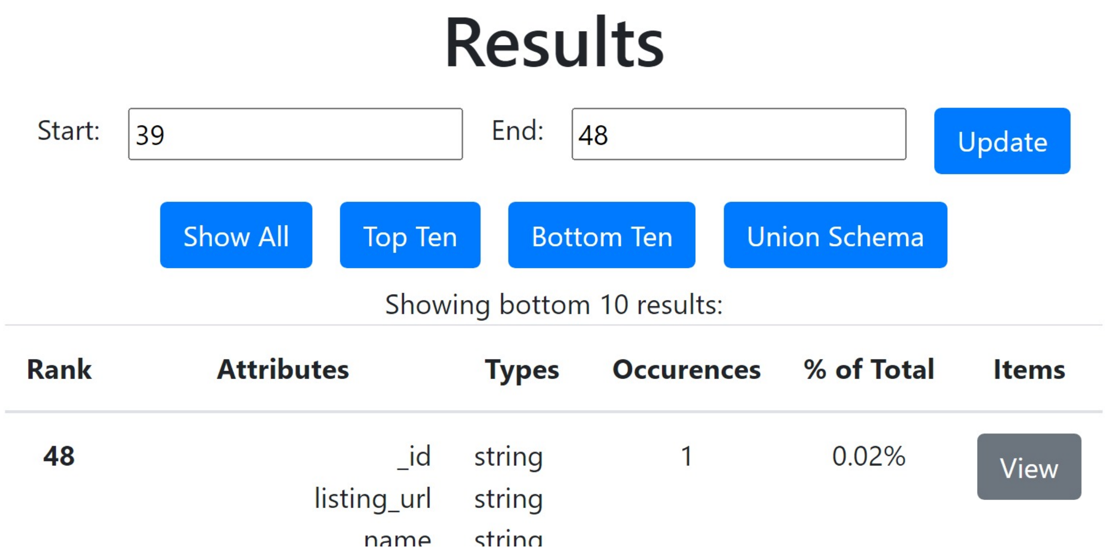

Results

Start: 1 End: 48

Showing results: 1-48 of 48

Rank	Attributes	Types	Occurences	% of Total	Items
1	_id listing_url name	string string string	2311	41.60%	View

Figure 4.8: Showing All Sub-schemas in the Database

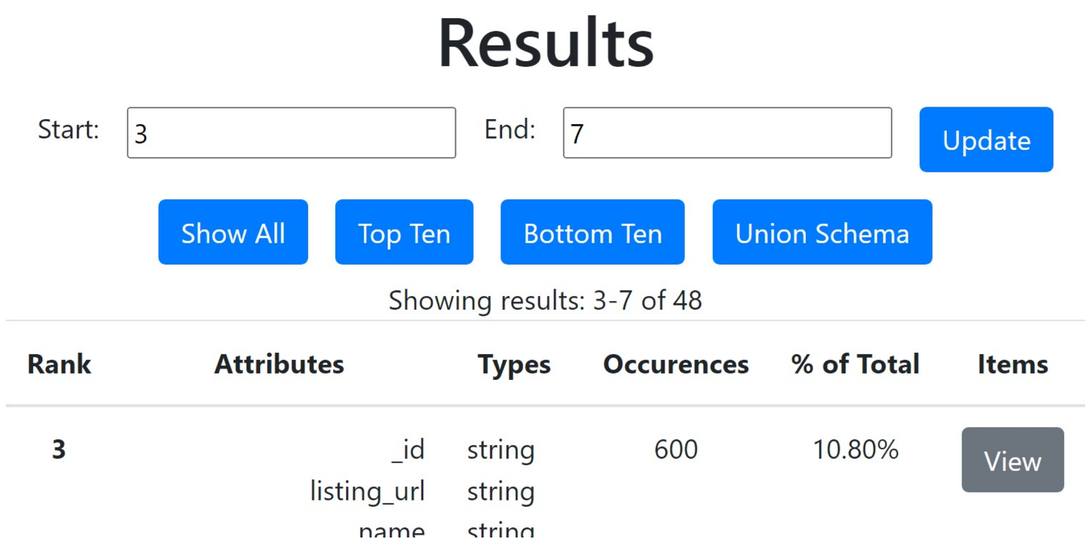


Start: 39 End: 48

Showing bottom 10 results:

Rank	Attributes	Types	Occurences	% of Total	Items
48	_id listing_url name	string string string	1	0.02%	View

Figure 4.9: Showing Bottom 10 Ranked Sub-schemas

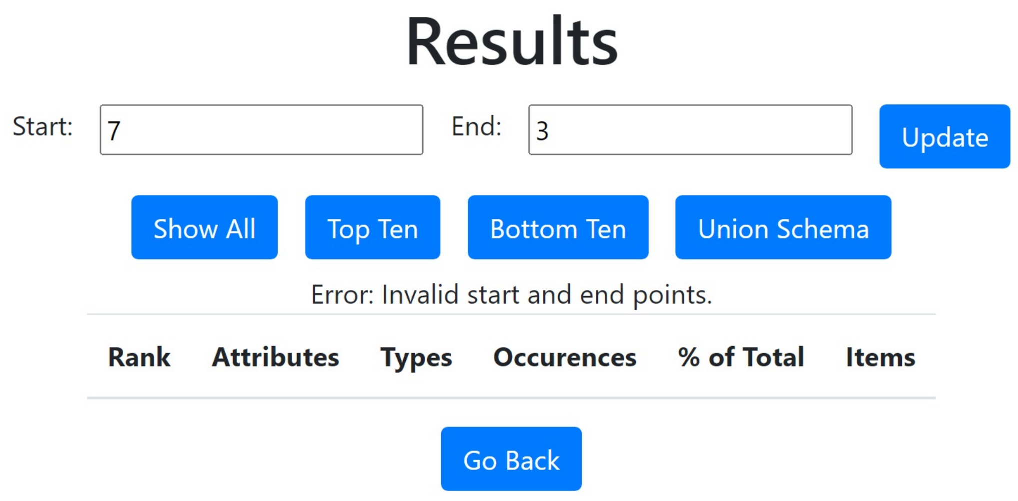


Start: 3 End: 7

Showing results: 3-7 of 48

Rank	Attributes	Types	Occurences	% of Total	Items
3	_id listing_url name	string string string	600	10.80%	View

Figure 4.10: Showing Custom Range of Sub-schemas

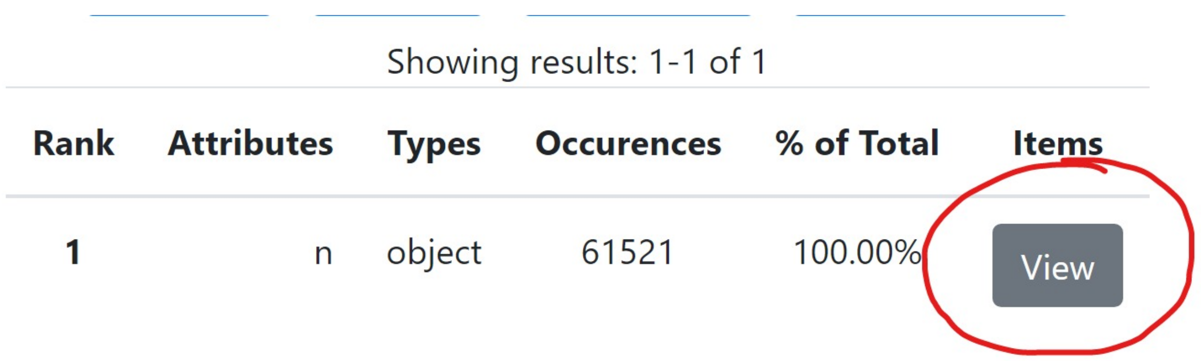

Results

Start: 7 End: 3

Error: Invalid start and end points.

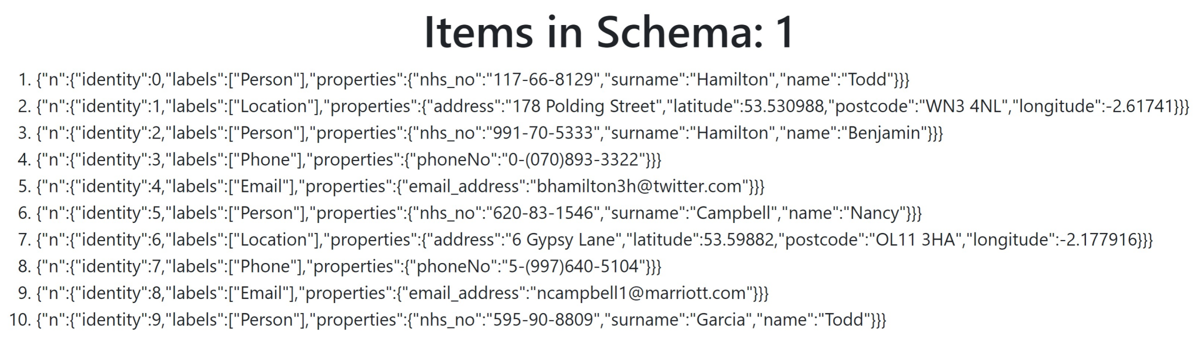
Rank	Attributes	Types	Occurences	% of Total	Items

Figure 4.11: Showing Invalid Input for Custom Range of Sub-schemas


Showing results: 1-1 of 1

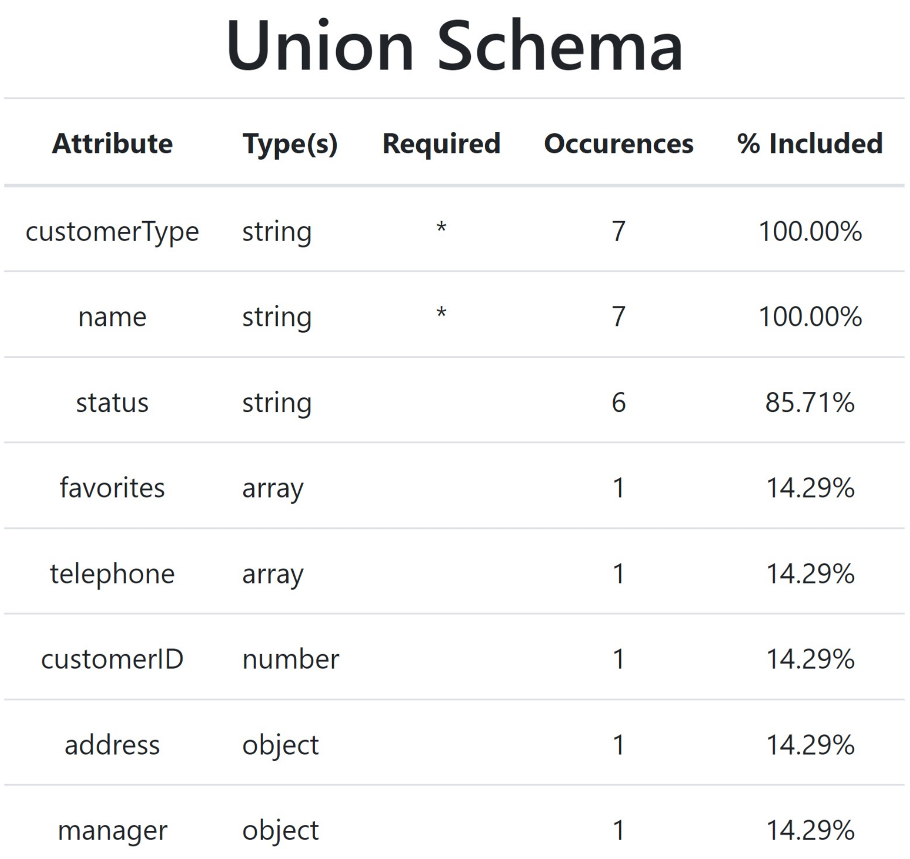
Rank	Attributes	Types	Occurences	% of Total	Items
1	n	object	61521	100.00%	View

Figure 4.12: Location of Button to View Sample Items, Circled in Red


Items in Schema: 1

1. {"n":{"identity":0,"labels":["Person"],"properties":{"nhs_no":"117-66-8129","surname":"Hamilton","name":"Todd"}}}
2. {"n":{"identity":1,"labels":["Location"],"properties":{"address":"178 Polding Street","latitude":53.530988,"postcode":"WN3 4NL","longitude":-2.61741}}}
3. {"n":{"identity":2,"labels":["Person"],"properties":{"nhs_no":"991-70-5333","surname":"Hamilton","name":"Benjamin"}}}
4. {"n":{"identity":3,"labels":["Phone"],"properties":{"phoneNo":"0-(070)893-3322"}}}
5. {"n":{"identity":4,"labels":["Email"],"properties":{"email_address":"bhamilton3h@twitter.com"}}}
6. {"n":{"identity":5,"labels":["Person"],"properties":{"nhs_no":"620-83-1546","surname":"Campbell","name":"Nancy"}}}
7. {"n":{"identity":6,"labels":["Location"],"properties":{"address":"6 Gypsy Lane","latitude":53.59882,"postcode":"OL11 3HA","longitude":-2.177916}}}
8. {"n":{"identity":7,"labels":["Phone"],"properties":{"phoneNo":"5-(997)640-5104"}}}
9. {"n":{"identity":8,"labels":["Email"],"properties":{"email_address":"ncampbell1@marriott.com"}}}
10. {"n":{"identity":9,"labels":["Person"],"properties":{"nhs_no":"595-90-8809","surname":"Garcia","name":"Todd"}}}

Figure 4.13: Page for Viewing Sample Items



Union Schema

Attribute	Type(s)	Required	Occurences	% Included
customerType	string	*	7	100.00%
name	string	*	7	100.00%
status	string		6	85.71%
favorites	array		1	14.29%
telephone	array		1	14.29%
customerID	number		1	14.29%
address	object		1	14.29%
manager	object		1	14.29%

Figure 4.14: Union Schema Page

Chapter 5

Validation

This chapter discusses the validation of the functionality of the app. This is achieved by applying unit tests, integration tests, and case studies, and we compare the outputs to different applications. This chapter outlines the experimental design of the validations in Section 5.1. Then the tests are performed, and the results are presented in Section 5.2. Finally, the results and comparisons are discussed in Section 5.3.

5.1 Experimental Design

To validate that Schemalysis fulfills the functionality we set to achieve, we administer several different testing scenarios to ensure the application achieves correct results. We implement and perform the following:

1. Unit Tests
2. Integration Tests
3. Case Studies

The purpose of the unit tests is to test for basic functionality on simple objects. The integration tests combine all cases from the unit tests as a single input and validates both the sub-schemas and union schema output. Finally, case studies are performed using large data sets of real-world databases to demonstrate functionality using actual data. Further description of each testing scenario is detailed in its respective section.

Additionally, we compare the output provided by Schemalysis to other tools. The applications featured in the discussion are:

- Hackolade [10]
- MongoDB Compass Schema Analyzer [11]
- JSON Schema Generator [22]

Hackolade is an application that allows for both reverse and forward engineering NoSQL databases. It supports more than 50 different NoSQL database systems [10]. This application allows a user to view a visual model of database as a diagram. When reverse engineering a database, Hackolade accepts input as either a document database or a JSON Schema and extracts the union schema of the database and displays it as a diagram. It can also forward engineer a document database from a model in the application. This outputs a document database with a single document containing each attribute with example data.

The application also provides an interactive interface in which a user can modify the visual model. This interface can also be used to construct a new visual database model from scratch by manually creating collections and specifying their attributes and other characteristics. Additionally, Hackolade generates a JSON Schema that corresponds to the database model currently shown in the interface.

We compare union schemas in Hackolade to both union and sub-schemas in Schemalysis for unit tests in Section 5.2.1, and the integration tests in Section 5.2.2.

MongoDB Compass is an application that allows users to interact with data stored in MongoDB, such as browsing and querying. One of these functions, that relates to the reverse engineering of data, is the schema analyzer tool. This tool is used to analyze the selected data set and provides a profile of the schema. MongoDB's schema analyzer is compared to the union schema feature of Schemalysis as part of the integration tests in Section 5.2.2.

JSON Schema Generator is a web-based tool available online for the purpose of generating a JSON Schema from document-based data. This application reads in a document database as text input and outputs its JSON Schema. Due to its lengthy output, we select a single unit test and present an excerpt of the output, which is shown in Figure 5.23.

These three tools have the closest functionality to Schemalysis, but Schemalysis provides additional features that are discussed in the comparisons below.

5.2 Results

This section covers the results of the functionality tests described above. It includes the unit tests in Section 5.2.1 validating correctness of small inputs of different types, the integration tests in Section 5.2.2 validating the combination of all unit tests as a single input, and case studies in Section 5.2.3 demonstrating the application used with large real-world data sets.

5.2.1 Unit Tests

The functionality of Schemalysis is first tested through several unit tests. We formulate unit tests to validate the use of the application with (1) a single level object, (2) adding an additional field, (3) removing a field, (4) adding an embedded object, (5) adding an array, (6) adding an embedded object with an array, (7) adding an array of objects, (8) varying the embedded object, (9) varying the array, (10) varying the array of objects, and (11) changing the type of a field. This yields a total of 11 unit tests. These involve running the app with different variations of an object. Table 5.1 shows each of the unit tests performed with information about each test's title, the control test object which is placed with the unit test object to observe how they differ in the application, and the content of the test object.

We observe how the variation is analyzed by the tool, then manually verify that the varied object is both a) recognized as a different sub-schema than the original object, and b) is accurately portrayed. Additionally, we compare the results of each unit test to Hackolade in order to view the difference in how Schemalysis and Hackolade handle the variation.

Table 5.1: Unit Tests

Test	Title	Control Test Case	Object Content
1	Original	N/A	{ “customerType”: “Corporate”, “status”: “Active”, “name”: “Worldwide Travel Agency” }
2	Add Field	1	{ “customerType”: “Corporate”, “status”: “Active”, “name”: “Worldwide Travel Agency”, “customerID”: 10023145 }
3	Remove Field	1	{ “customerType”: “Corporate”, “name”: “Worldwide Travel Agency” }
4	Add Embedded Object	1	{ “customerType”: “Corporate”, “status”: “Active”, “name”: “Worldwide Travel Agency”, “address”: { “street”: “10 Main Street”, “city”: “New York City”, “state”: “NY”, “zip”: 10001 } }
5	Add Array	1	{ “customerType”: “Corporate”, “status”: “Active”, “name”: “Worldwide Travel Agency”, “favorites”: [“food”, “furniture”, “cleaning”] }

6	Add Embedded Object with an Array	1	{
			“customerType”: “Corporate”,
			“status”: “Active”,
			“name”: “Worldwide Travel Agency”,
			“manager”: {
			“name”: “Steve”,
			“hobbies”: [“golfing”, “fishing”, “volunteering”],
			“remote”: false
			}
			}

7	Add Array of Objects	1	{
			“customerType”: “Corporate”,
			“status”: “Active”,
			“name”: “Worldwide Travel Agency”,
			“telephone”: [
			{
			“type”: “home”,
			“area code”: “619”,
			“phone”: “555 6789”
			},
			{
			“type”: “mobile”,
			“area code”: “619”,
			“phone”: “423 1114”
			}
			]
			}

8	Variation in Embedded Object	4	{
			“customerType”: “Corporate”,
			“status”: “Active”,
			“name”: “Worldwide Travel Agency”,
			“address”: {
			“state”: “NY”,
			“coordinates”: [40.1, -123.2],
			“country”: “USA”
			}
			}

9	Variation in Array	5	{
			“customerType”: “Corporate”,
			“status”: “Active”,
			“name”: “Worldwide Travel Agency”,
			“favorites”: [“food”, 42, true]
			}

10	Variation in Array of Objects	7	{
			“customerType”: “Corporate”,
			“status”: “Active”,
			“name”: “Worldwide Travel Agency”,
			“telephone”: [
			{
			“type”: “home”,
			“area code”: “619”,
			“phone”: “555 6789”
			},
			{
			“number”: “(456) 830-5516”
			}
			]
			}

11	Change type of Field	1	{
			“customerType”: “Corporate”,
			“status”: 200,
			“name”: “Worldwide Travel Agency”
			}

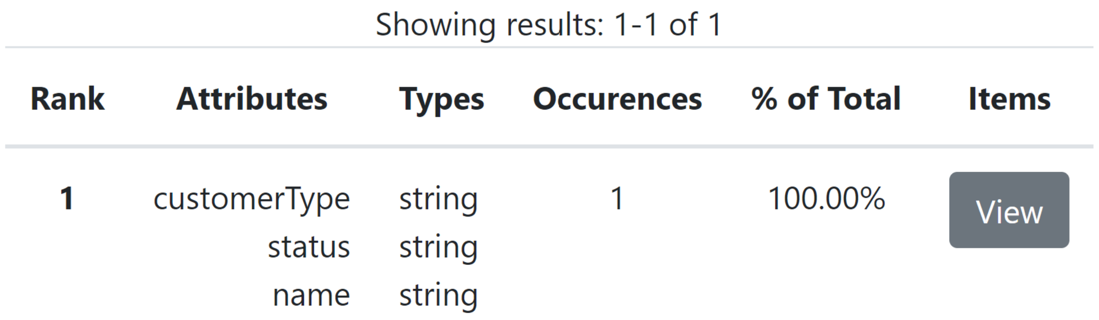
1. Original object

This is the simplest unit test conducted with a single object with primitive attributes. Its purpose is to ensure that the application functions as intended on a basic object. The results are shown in Figure 5.1, and the output from the same input into Hackolade in Figure 5.2 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



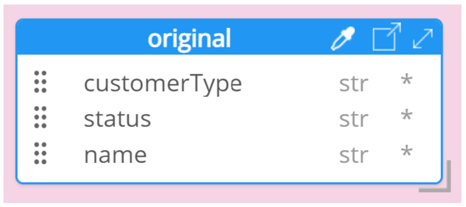
Showing results: 1-1 of 1

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name	string string string	1	100.00%	View

Figure 5.1: Unit Test 1: Original Object

Schemalysis successfully loads the schema of the singular object. Each attribute and type corresponds to the input document.

Comparison to Hackolade



original

customerType	str	*
status	str	*
name	str	*

Figure 5.2: Unit Test 1: Hackolade

For unit test 1, Hackolade produces a very similar output. It provides the same profile of attributes and types, as well as marks each attribute as required.

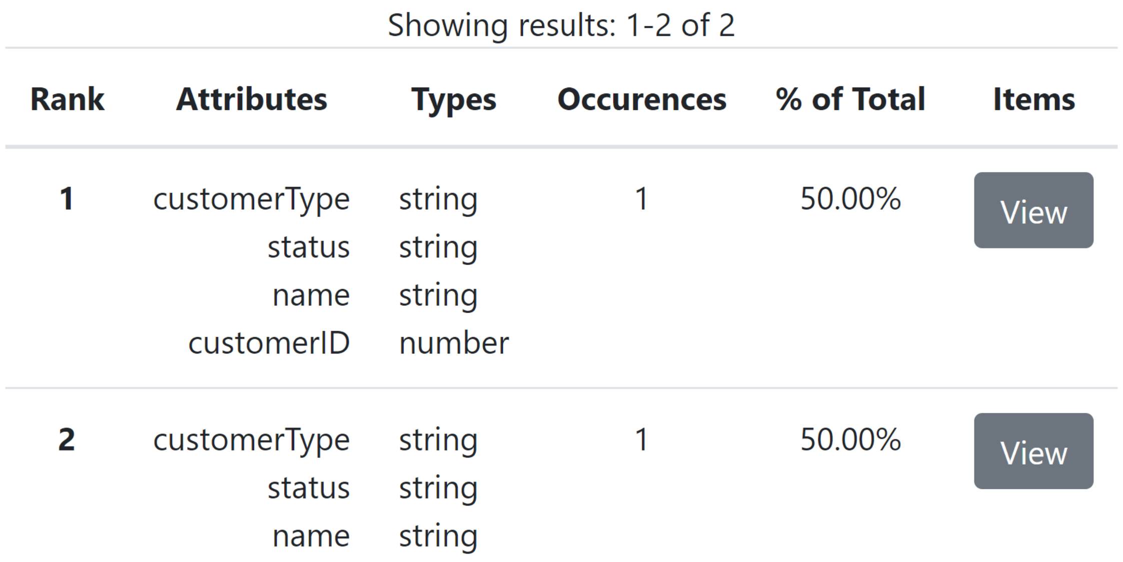
2. Add Field

Unit test 2 is conducted by profiling an item with an additional field added. As a control, the document from unit test 1 is also profiled with it. This is to allow the variations between the test documents to be captured. The results are shown in Figure 5.3, and the output from the same input into Hackolade in Figure 5.4 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "customerID": 10023145
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



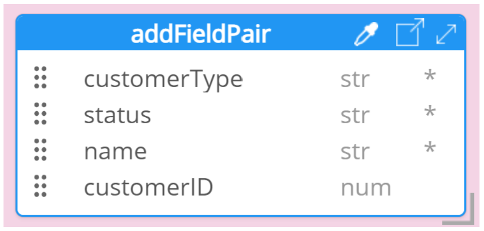
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name customerID	string string string number	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.3: Unit Test 2: Add Field

Unit test 2 is successfully passed as the variation in both documents are recognized, and the attributes and types of both objects are correctly illustrated.

Comparison to Hackolade



addFieldPair

customerType	str	*
status	str	*
name	str	*
customerID	num	

Figure 5.4: Unit Test 2: Hackolade

With a second sub-schema, the output of Schemalysis and Hackolade begin to differ. As Hackolade provides a union schema profile, its output consists of a single schema. Hackolade shows the variation of the objects by not markeing the added field as required since it is not present in the original object.

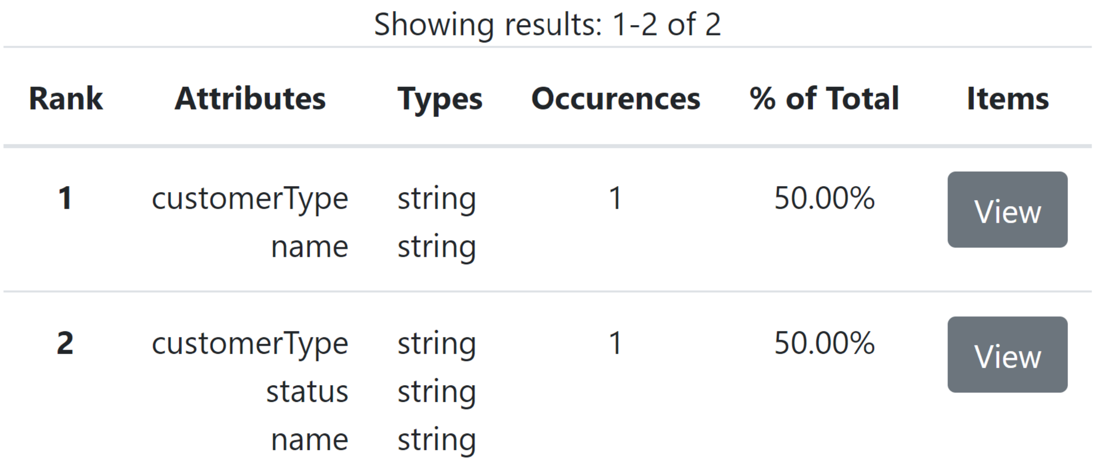
3. Remove Field

Unit test 3 is conducted by profiling an item with a field subtracted from the original document. As a control, the document from unit test 1 is also profiled with it. The results are shown in Figure 5.5, and the output from the same input into Hackolade in Figure 5.6 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "name": "Worldwide Travel Agency"
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



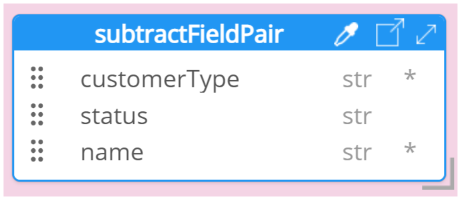
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType name	string string	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.5: Unit Test 3: Remove Field

Unit test 3 is successfully passed as both variations of the document are recognized, and both sub-schemas are correctly profiled.

Comparison to Hackolade


subtractFieldPair

customerType	str	*
status	str	
name	str	*

Figure 5.6: Unit Test 3: Hackolade

The profile that Hackolade provides for unit test 3 is similar to unit test 2 in the way that it shows the variation. However, the key difference is that since the variation document is missing one of the fields from the original document, that field is no longer marked required, while Schemalysis fully captures the sub-schemas of both documents and tracks their data separately.

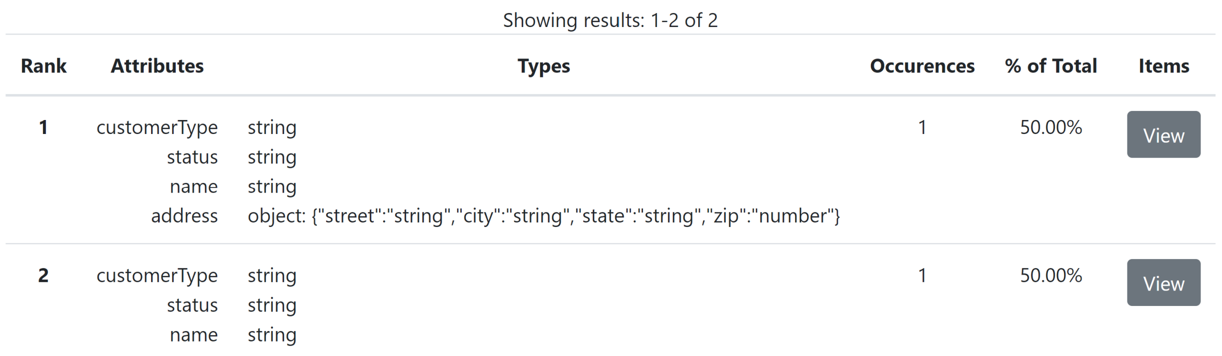
4. Add Embedded Object

Unit test 4 is conducted by adding an embedded object to the original document. The purpose of adding an embedded object is to verify that Schemalysis can handle processing a complex embedded object with multiple properties in addition to simple fields. As a control, the original document from unit test 1 is also included to verify that a variation is recognized. The results are shown in Figure 5.7, and the output from the same input into Hackolade in Figure 5.8 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "address": {
        "street": "10 Main Street",
        "city": "New York City",
        "state": "NY",
        "zip": 10001
    }
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



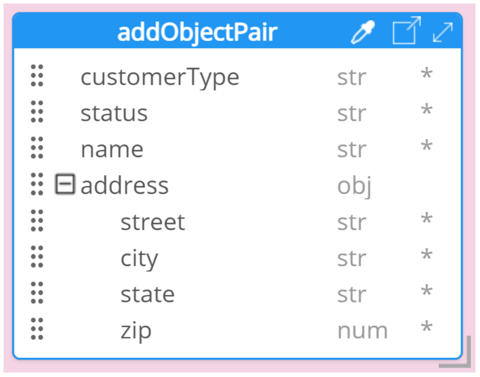
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name address	string string string object: {"street":"string","city":"string","state":"string","zip":"number"}	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.7: Unit Test 4: Add Embedded Object

Schemalysis successfully passes unit test 4 in both recognizing the variation with an embedded object and portraying the schema profile correctly. Furthermore, the application displays the embedded object as a string representation of the object as JSON.

Comparison to Hackolade



addObjectPair

customerType	str	*
status	str	*
name	str	*
address	obj	
street	str	*
city	str	*
state	str	*
zip	num	*

Figure 5.8: Unit Test 4: Hackolade

The way that embedded objects are presented are very different between Schemalysis and Hackolade. Schemalysis represents embedded objects as a string representation of the objects with all attributes and types contained in a single line. Hackolade, on the other hand, presents an embedded document as a collapsible object field, and separately profiles the attributes inside of the embedded objects, so they are also marked required, despite their not existing in the original document.

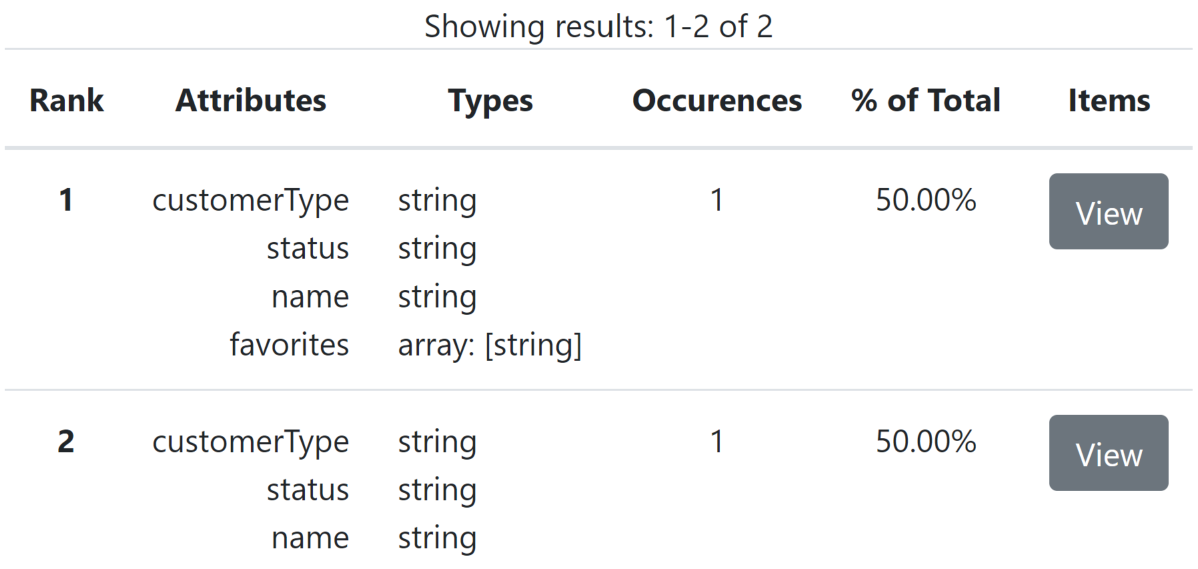
5. Add Array

Unit test 5 is conducted by adding an array, called favorites, to the original document. The purpose of adding an array is to verify that Schemalysis can handle processing attributes that are represented as arrays as well as describe types the types of fields contained in them. The results are shown in Figure 5.9, and the output from the same input into Hackolade in Figure 5.10 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "favorites": ["food", "furniture", "cleaning"]
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name favorites	string string string array: [string]	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.9: Unit Test 5: Add Array

Schemalysis successfully passes unit test 5 in both recognizing the variation with an array and portraying the schema profile correctly. Furthermore, the application displays the attribute, favorites, with the word “array” followed by a list of types included in the array.

Comparison to Hackolade

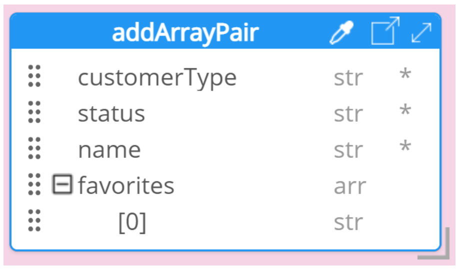


Figure 5.10: Unit Test 5: Hackolade

Hackolade depicts an array as an array field with the types of attributes contained in the array as a sub-field. Schemalysis depicts an array in a string representation and including the types of items in the array and treats it as the type of the attribute.

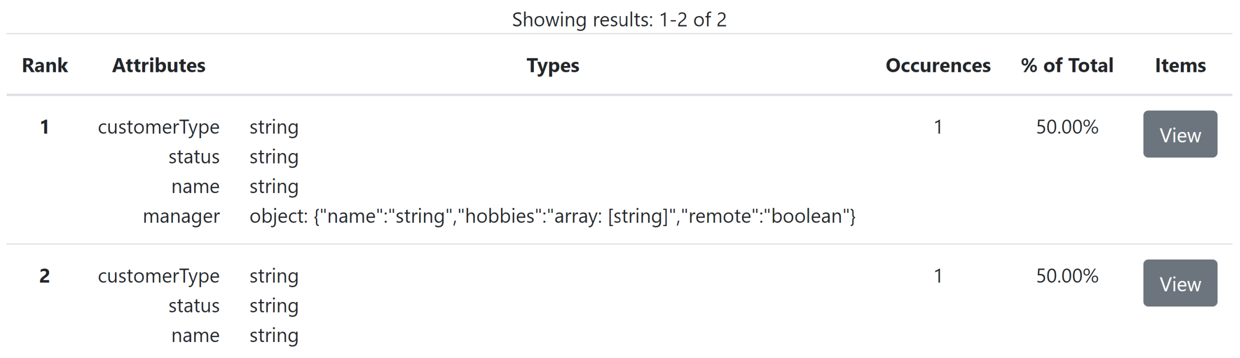
6. Add Embedded Object with an Array

Unit test 6 is conducted by adding an embedded object that contains an array to the original document. The purpose of adding an embedded object containing an array is to verify that Schemalysis can handle processing arrays inside of embedded objects. As a control, the original document from unit test 1 is also included in the input. The results are shown in Figure 5.11, and the output from the same input into Hackolade in Figure 5.12 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "manager": {
        "name": "Steve",
        "hobbies": ["golfing", "fishing", "volunteering"],
        "remote": false
    }
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



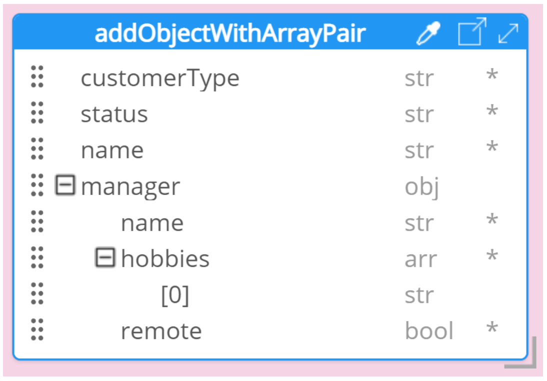
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name manager	string string string object: {"name":"string","hobbies":"array: [string]","remote":"boolean"}	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.11: Unit Test 6: Add Embedded Object with an Array

Schemalysis successfully passes unit test 6 in both recognizing the variation with an embedded object containing an array and portraying the schema profile correctly. Furthermore, the application displays an embedded object with an array the same way that the embedded object is shown in unit test 4 is displayed with the inner array's type matching how arrays are shown in unit test 5.

Comparison to Hackolade



addObjectWithArrayPair

customerType	str	*
status	str	*
name	str	*
manager	obj	
name	str	*
hobbies	arr	*
[0]	str	
remote	bool	*

Figure 5.12: Unit Test 6: Hackolade

Similarly to how Schemalysis handled the array inside of the embedded object, Hackolade also portrayed the embedded object the same way as it did in unit test 4, and the array inside the embedded object is portrayed the same was it was in unit test 5.

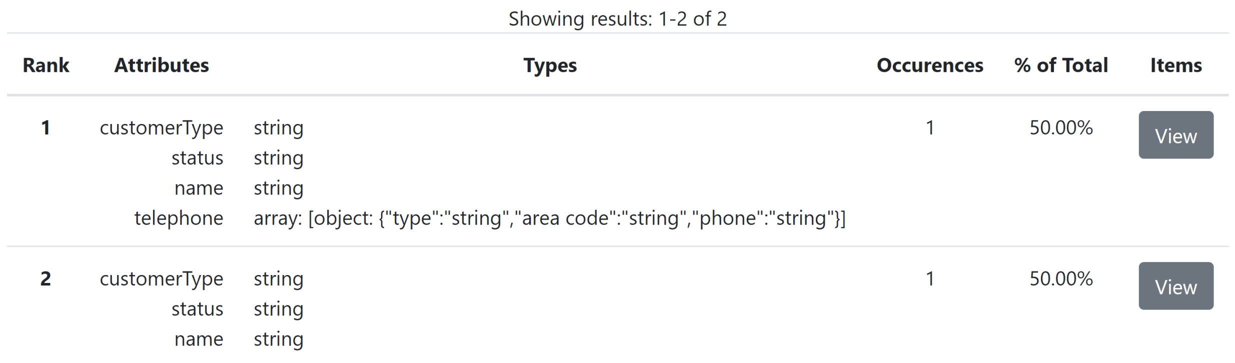
7. Add Array of Objects

Unit test 7 is conducted by adding an array of objects to the original document. The purpose of adding an array of objects is to verify that Schemalysis can handle processing an array containing elements whose types are objects. As a control, the original document from unit test 1 is also included in the input. The results are shown in Figure 5.13, and the output from the same input into Hackolade in Figure 5.14 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "telephone": [
        {
          "type": "home",
          "area code": "619",
          "phone": "555 6789"
        },
        {
          "type": "mobile",
          "area code": "619",
          "phone": "423 1114"
        }
    ]
},
{
  "customerType": "Corporate",
  "status": "Active",
  "name": "Worldwide Travel Agency"
}]
```

Results:



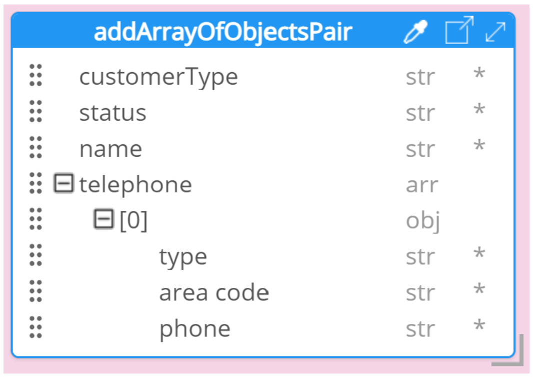
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name telephone	string string string array: [object: {"type":"string","area code":"string","phone":"string"}]	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.13: Unit Test 7: Add Array of Objects

Schemalysis successfully passes unit test 7 in both recognizing the variation with an array of objects and portraying the schema profile correctly. Furthermore the application displays an array of objects the same way that it displays an array shown in unit test 5 with the types inside the array matching how an embedded object is displayed shown in unit test 4.

Comparison to Hackolade



```
customerType    str  *
status          str  *
name            str  *
telephone       arr
  [0]           obj
    type        str  *
    area code   str  *
    phone       str  *
```

Figure 5.14: Unit Test 7: Hackolade

Hackolade also portrays objects inside of arrays the same way that it portrays an array of primitive types, except the type in the array is profiled as an embedded object. Compared to Schemalysis, Hackolade represents this unit test as a sub-profile within the array, while Schemalysis captures the entire attribute into a string representation.

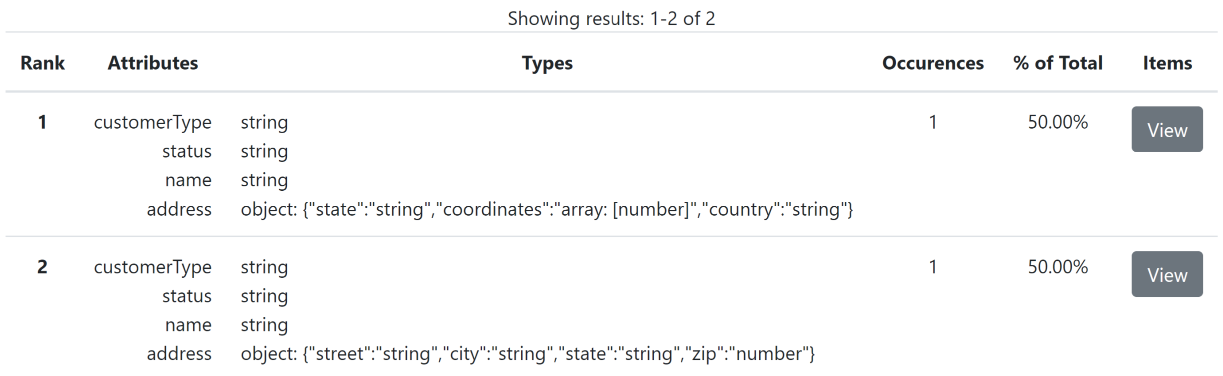
8. Variation in Embedded Object

Unit test 8 is conducted by applying variation to the embedded object from unit test 4. The purpose of this test is to validate that Schemalysis can recognize variations between embedded objects. Unlike previous tests, the control for this unit test is the document from unit test 4 in order to capture the variance between them. The results are shown in Figure 5.15, and the output from the same input into Hackolade in Figure 5.16 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "address": {
        "state": "NY",
        "coordinates": [40.1, -123.2],
        "country": "USA"
    }
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "address": {
        "street": "10 Main Street",
        "city": "New York City",
        "state": "NY",
        "zip": 10001
    }
}]
```

Results:



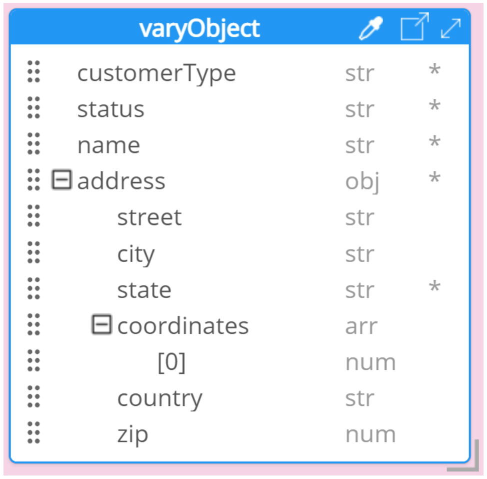
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name address	string string string object: {"state":"string","coordinates":"array: [number]","country":"string"}	1	50.00%	View
2	customerType status name address	string string string object: {"street":"string","city":"string","state":"string","zip":"number"}	1	50.00%	View

Figure 5.15: Unit Test 8: Vary Embedded Object

Schemalysis successfully recognizes both variations of the embedded objects. They are both represented as strings, and since they are different, they are recognized as two different sub-schemas.

Comparison to Hackolade



customerType	str	*
status	str	*
name	str	*
⊟address	obj	*
street	str	
city	str	
state	str	*
⊟coordinates	arr	
[0]	num	
country	str	
zip	num	

Figure 5.16: Unit Test 8: Hackolade

While Schemalysis recognizes the variation in the embedded object as two separate sub-schemas, Hackolade portrays this by providing a profile of the embedded objects. The state field that is marked required is present in both variations, but the others are not.

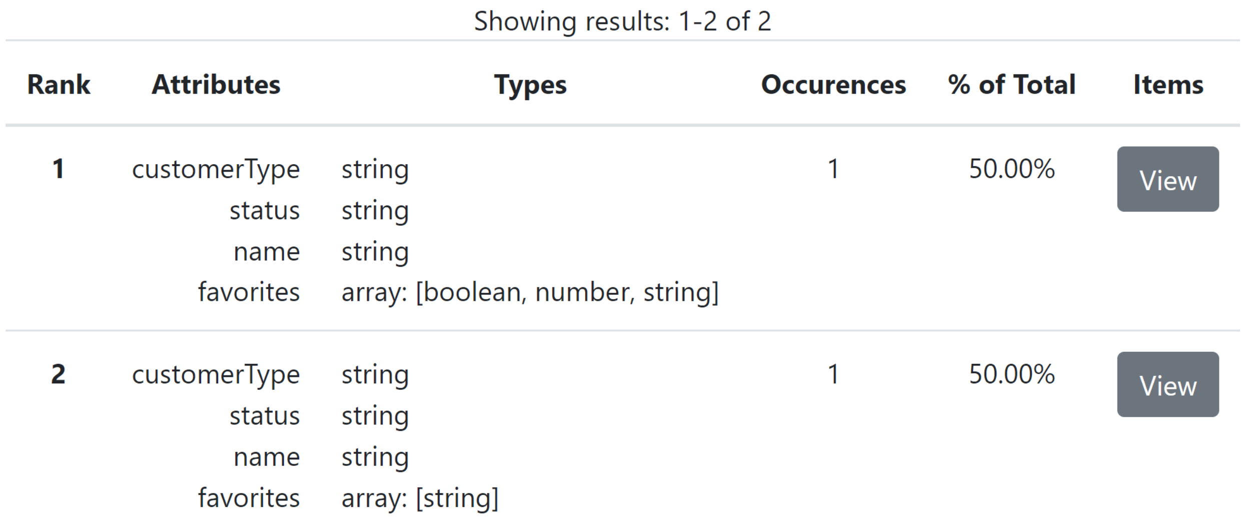
9. Variation in Array

Unit test 9 is conducted by applying variation to the array from unit test 5. The purpose of this test is to validate that Schemalysis can recognize arrays that contain different types of items. The control for this test is document from unit test 5. The results are shown in Figure 5.17, and the output from the same input into Hackolade in Figure 5.18 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "favorites": ["food", 42, true]
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "favorites": ["food", "furniture", "cleaning"]
}]
```

Results:



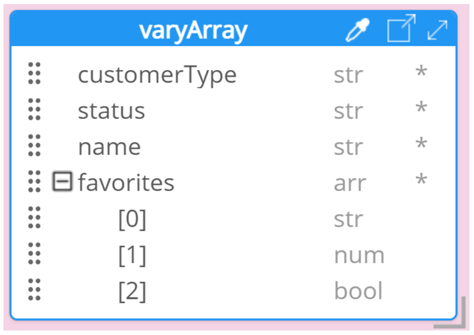
Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name favorites	string string string array: [boolean, number, string]	1	50.00%	View
2	customerType status name favorites	string string string array: [string]	1	50.00%	View

Figure 5.17: Unit Test 9: Vary Array

Schemalysis successfully passes unit test 9 by recognizing the different types of objects inside of the array. Furthermore, this changes the structure of the type of the array so it is portrayed as a separate sub-schema than the control document from unit test 5.

Comparison to Hackolade



varyArray

customerType	str	*
status	str	*
name	str	*
favorites	arr	*
[0]	str	
[1]	num	
[2]	bool	

Figure 5.18: Unit Test 9: Hackolade

Hackolade also recognizes the different types of attributes contained in the array. However, the main difference between this output and the output Schemalysis provides is that Hackolade does not distinguish between the structure of the array between the two documents.

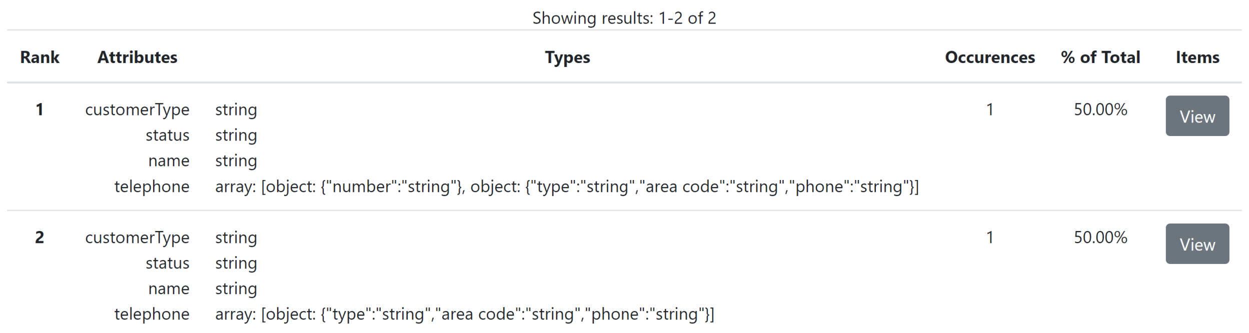
10. Variation in Array of Objects

Unit test 10 is conducted by applying variance to the objects inside of an array. The purpose of this test is to validate that Schemalysis can handle and array of objects that are not the same and see how it portrays this variance. The control for this unit test is the document from unit test 7. The results are shown in Figure 5.19, and the output from the same input into Hackolade in Figure 5.20 followed by a brief comparison.

Input:

```
[{
  "customerType": "Corporate",
  "status": "Active",
  "name": "Worldwide Travel Agency",
  "telephone": [
      {
        "type": "home",
        "area code": "619",
        "phone": "555 6789"
      },
      {
        "number": "(456) 830-5516"
      }
  ]
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency",
    "telephone": [
        {
          "type": "home",
          "area code": "619",
          "phone": "555 6789"
        },
        {
          "type": "mobile",
          "area code": "619",
          "phone": "423 1114"
        }
    ]
}]
```

Results:

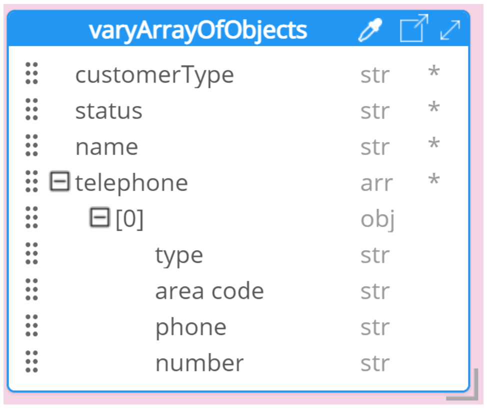

Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name telephone	string string string array: [object: {"number":"string"}, object: {"type":"string","area code":"string","phone":"string"}]	1	50.00%	View
2	customerType status name telephone	string string string array: [object: {"type":"string","area code":"string","phone":"string"}]	1	50.00%	View

Figure 5.19: Unit Test 10: Vary Array of Objects

Schemalysis successfully passes unit test 10 by recognizes both variations of objects inside of the array. This is seen in the type of the array attribute containing a string representation of both objects. This is due to the objects type being processed as a string. Since the strings are different, they are recognized as two separate types within the array, as opposed to one single type in the array in the control document.

Comparison to Hackolade


varyArrayOfObjects

customerType	str	*
status	str	*
name	str	*
telephone	arr	*
[0]	obj	
type	str	
area code	str	
phone	str	
number	str	

Figure 5.20: Unit Test 10: Hackolade

Hackolade also recognizes the variation within the array of objects. However, the type inside the array is a profile of the two objects. This inner profile is a sub-union schema of the embedded objects, and it treats all fields as not required since they share no attributes in common. However, this approach does not allow for the ability to view individual schemas inside of the array.

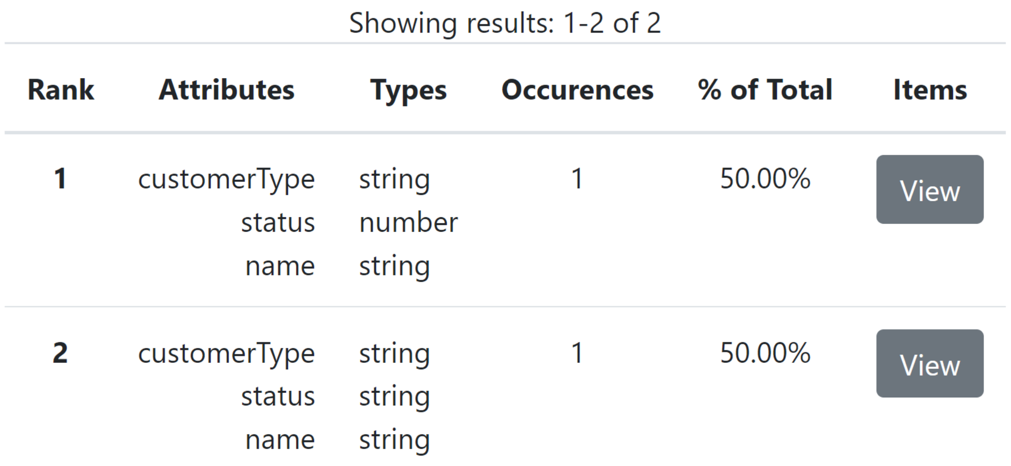
11. Change Type of Field

Unit test 11 is conducted by changing the type of one of the fields in the original object. The purpose of this test is to validate that Schemalysis recognizes when two fields have the same names, but are completely different types. The control for this unit test is the original document from unit test 1. The results are shown in Figure 5.21, and the output from the same input into Hackolade in Figure 5.22 followed by a brief comparison.

Input:

```
[{
    "customerType": "Corporate",
    "status": 200,
    "name": "Worldwide Travel Agency"
},
{
    "customerType": "Corporate",
    "status": "Active",
    "name": "Worldwide Travel Agency"
}]
```

Results:



Showing results: 1-2 of 2

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name	string number string	1	50.00%	View
2	customerType status name	string string string	1	50.00%	View

Figure 5.21: Unit Test 11: Change Type of Field

Unit test 11 is successfully passed since both variations of the document are recognized. When depicting this type of variation, Schemalysis recognizes both documents as having different sub-schemas because, even if they have the same attributes, the types of each attribute are not the same.

Comparison to Hackolade

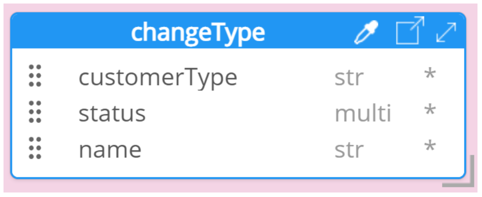


Figure 5.22: Unit Test 11: Hackolade

Hackolade handles this type of variation by describing the type of the varied field as a 'multi' type, indicating it can have different types. While this condenses the output to a single schema, it does not allow for the ability to view what types there are the way that Schemalysis does.

Overall Comparison to Hackolade

When performing reverse engineering in Hackolade, the input is a document database, and the output is a visual model of the database given as a union schema. This union schema gives a profile of all attributes represented in the database, as well as information about each attribute's type and whether it is required.

When adding an embedded object, Hackolade shows this as an optional field with a doc type, and also profiles the items inside of it. When adding an array, Hackolade shows this as an optional field with an arr type, and shows the type of objects in the array. When adding an embedded object with an array, Hackolade shows this using the same format as adding an embedded object, with the array field shown the same as it is on the upper level. When adding an array of objects, Hackolade shows this the same way as an array, but the types inside the array match the output for an embedded object. It also has the ability to profile embedded objects for any documents that contain it.

Comparison to JSON Schema Generator

As a comparison to a different application, we select one unit test to execute using an online JSON Schema Generator. We only select one unit test due to the lengthy output generated by the tool. We compare the output generated in the format of a JSON Schema and the output provided by Schemalysis. The result for unit test 4, adding an embedded object, is shown in Figure 5.23. For brevity, some of the output is not shown. The required sections of the JSON schema output have been highlighted to show that both the original and modified objects and their attributes are recognized.

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "$id": "http://example.com/example.json",
    "type": "array",
    "title": "The root schema",
    "description": "The root schema comprises the entire JSON document.",
    "default": [],
```

...

```
    "items": {
        "$id": "#/items",
        "anyOf": [
            {
                "$id": "#/items/anyOf/0",
                "type": "object",
                "title": "The first anyOf schema",
                "description": "An explanation about the purpose of this instance."
                "default": {},
```

...

```
                "required": [
                    "customerType",
                    "status",
                    "name",
                    "address"
                ],
```

...

```
            {
                "$id": "#/items/anyOf/1",
                "type": "object",
                "title": "The second anyOf schema",
                "description": "An explanation about the purpose of this instance."
                "default": {},
```

...

```
                "required": [
                    "customerType",
                    "status",
                    "name"
                ],
...



                    }
                ]
        }
}
```

Figure 5.23: Output excerpt for Unit Test 4 in JSON Schema Generator

The output in Figure 5.23 provides a few key pieces of information about the document database. The most useful part is the field named items. This field provides a list of the types of items along with information about their types and examples from the original database. The areas marked in boxes in Figure 5.23 show the schemas of the two documents profiled in unit test 4. While this is useful information, the JSON Schema output is not designed for ease of human comprehension. Its primary purpose is to use as validation for future documents entered to ensure that they adhere to the format of the documents described in the JSON Schema.

5.2.2 Integration Tests

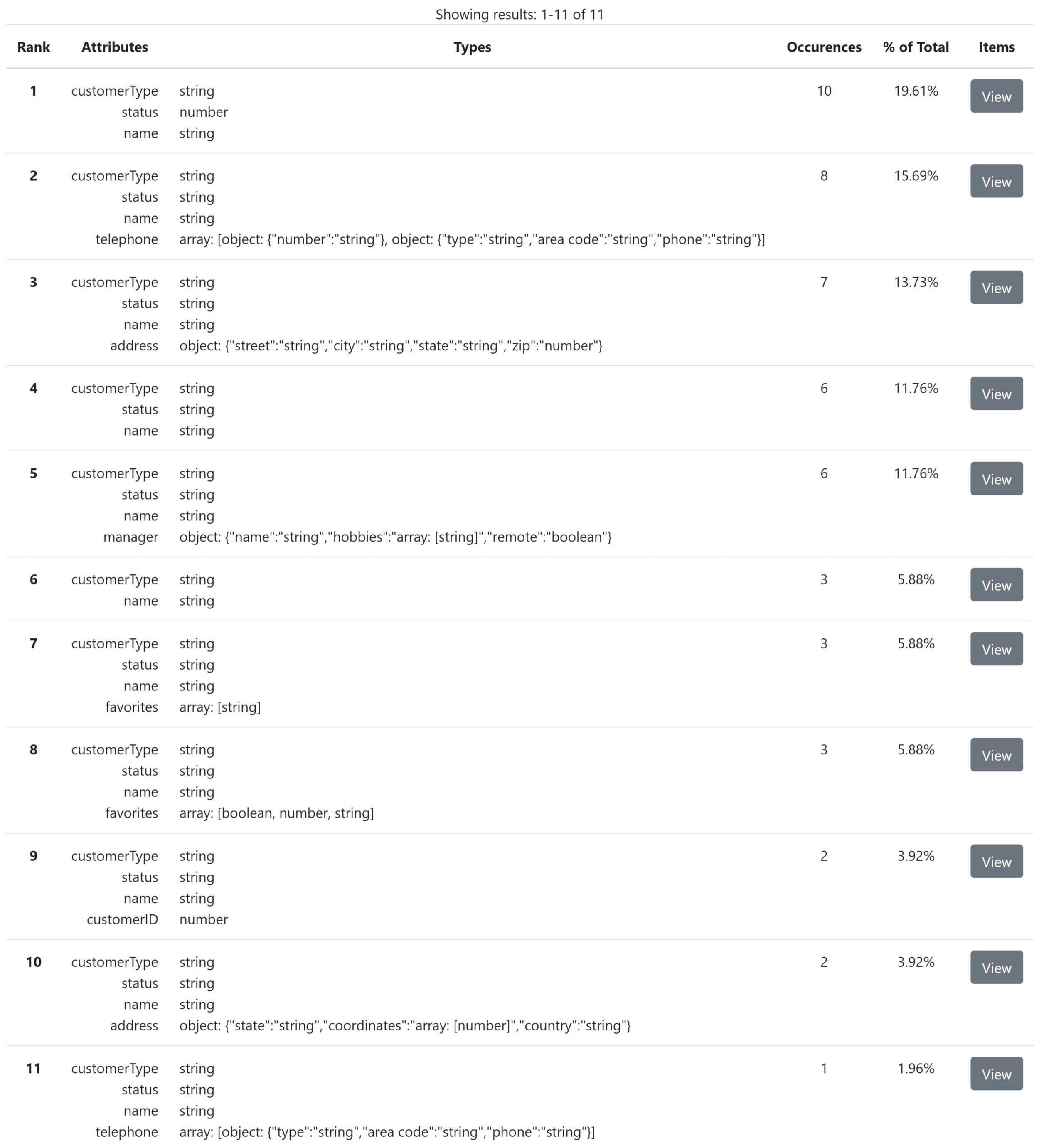
After the completion the unit testing, we combine all of the objects into a single collection to run through Schemalysis. This is to confirm that all of the sub-schemas of the unit test objects are recognized and accurately portrayed by the application. For the integration tests, we divide this into two sections: the sub-schemas and the union schema. Furthermore, we also vary the number of times that each unit test document appears to validate that repeat documents are recognized, and that the count and percentage of each sub-schema/attribute are shown. We use a random number generator to generate a number between 1-10 to decide the number of times each test document occurs. Table 5.2 shows each unit test document and the number of times and percentage it occurs, rounded to the nearest hundredth percent. Since the output provided by Hackolade and MongoDB Compass Schema Analyzer are more comparable to union schemas, they are included in the union schema section of the integration tests.

Table 5.2: Integration Test Object Quantities and Percentages

Test	Title	Count	Percentage
1	Original	6	11.76%
2	Add Field	2	3.92%
3	Remove Field	3	5.88%
4	Add Embedded Object	7	13.73%
5	Add Array	3	5.88%
6	Add Embedded Object with an Array	6	11.76%
7	Add Array of Objects	1	1.96%
8	Variation in Embedded Object	2	3.92%
9	Variation in Array	3	5.88%
10	Variation in Array of Objects	8	15.69%
11	Change type of Field	10	19.61%
	Total	51	100%

Sub-Schemas

First, we validate the application when run with a database containing all items from the unit tests. The result of the integration test is shown in Figure 5.24 and successfully shows all sub-schemas for each variation. The integration test was also performed 20 different times for different permutations of the data, and the same result was achieved, except that sub-schemas with the same count value appeared in the order that the objects appeared in the database.


Showing results: 1-11 of 11

Rank	Attributes	Types	Occurences	% of Total	Items
1	customerType status name	string number string	10	19.61%	View
2	customerType status name telephone	string string string array: [object: {"number":"string"}, object: {"type":"string","area code":"string","phone":"string"}]	8	15.69%	View
3	customerType status name address	string string string object: {"street":"string","city":"string","state":"string","zip":"number"}	7	13.73%	View
4	customerType status name	string string string	6	11.76%	View
5	customerType status name manager	string string string object: {"name":"string","hobbies":"array: [string]","remote":"boolean"}	6	11.76%	View
6	customerType name	string string	3	5.88%	View
7	customerType status name favorites	string string string array: [string]	3	5.88%	View
8	customerType status name favorites	string string string array: [boolean, number, string]	3	5.88%	View
9	customerType status name customerID	string string string number	2	3.92%	View
10	customerType status name address	string string string object: {"state":"string","coordinates":"array: [number]","country":"string"}	2	3.92%	View
11	customerType status name telephone	string string string array: [object: {"type":"string","area code":"string","phone":"string"}]	1	1.96%	View

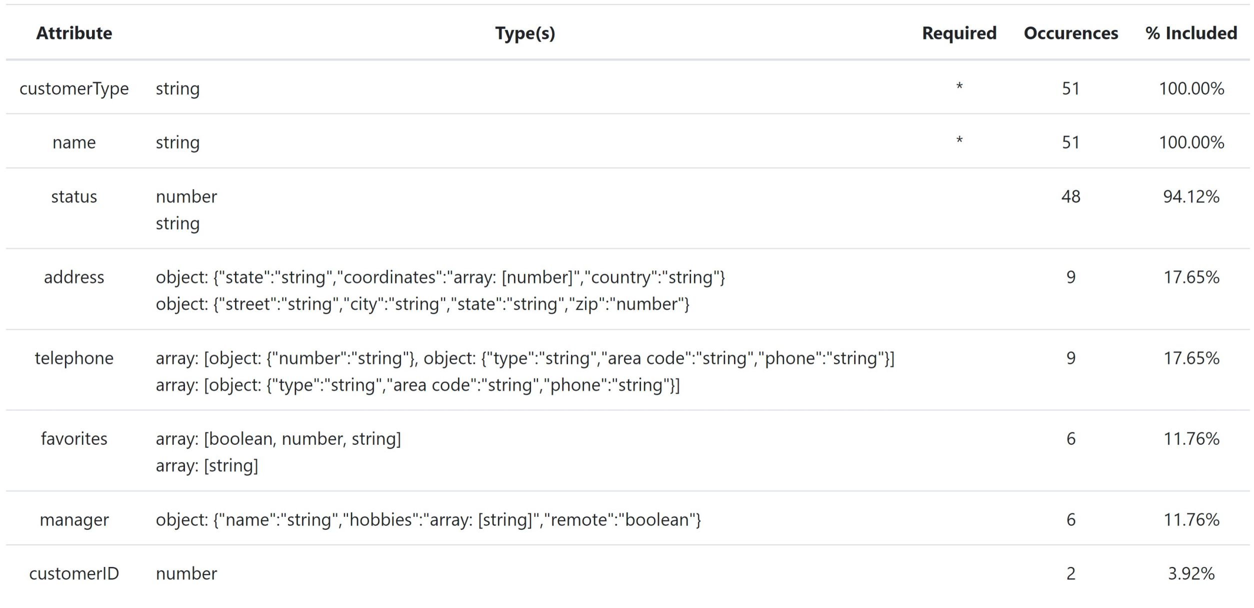
Figure 5.24: Integration Test: Schemalysis Sub-schemas

The output from the integration test matches the input as shown in Table 5.2. Schemalysis recognizes all 11 different sub-schemas, and the counts and percentages match the table perfectly. The integration test was also performed 20 different times for different permutations of the data, and the same result was achieved, except that sub-schemas with the same count value appeared in the order that the objects appeared in the database.

Union Schema

For a holistic view of the attributes represented by document-based data, Schemalysis also offers the ability to view the union schema. This is a single schema composed of every field included by the data. The union schema generated by Schemalysis and MongoDB Compass also provides additional information about the union schema, including whether each field is included in every data object, the number of data objects that include the field, and the overall percentage of data objects that contain the field. This makes it easier to spot outlier attributes or data that is missing a field that most other data has.

To obtain the union schema for this integration test, we input the same input documents described in Table 5.2 and used for the sub-schema section of the integration tests. The union schema for this input is shown in Figure 5.25 and is compared to the output that Hackolade provides shown in Figure 5.26 as well as the output that MongoDB Compass outputs in Figures 5.27 and 5.28.

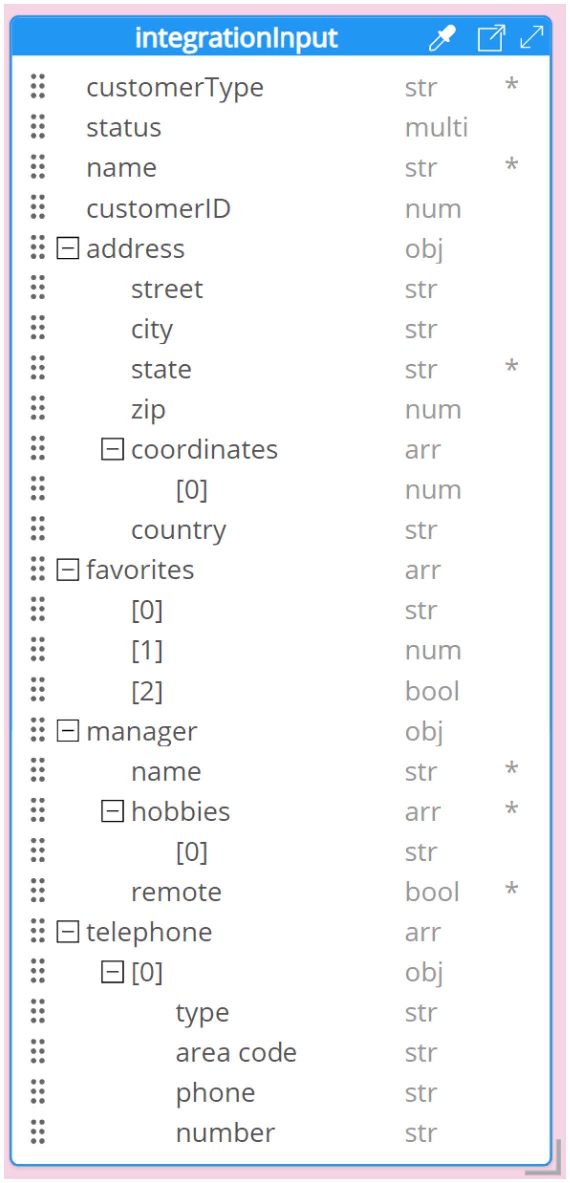


Attribute	Type(s)	Required	Occurences	% Included
customerType	string	*	51	100.00%
name	string	*	51	100.00%
status	number string		48	94.12%

address	object: {"state":"string","coordinates":"array: [number]","country":"string"}
	object: {"street":"string","city":"string","state":"string","zip":"number"}
			9	17.65%

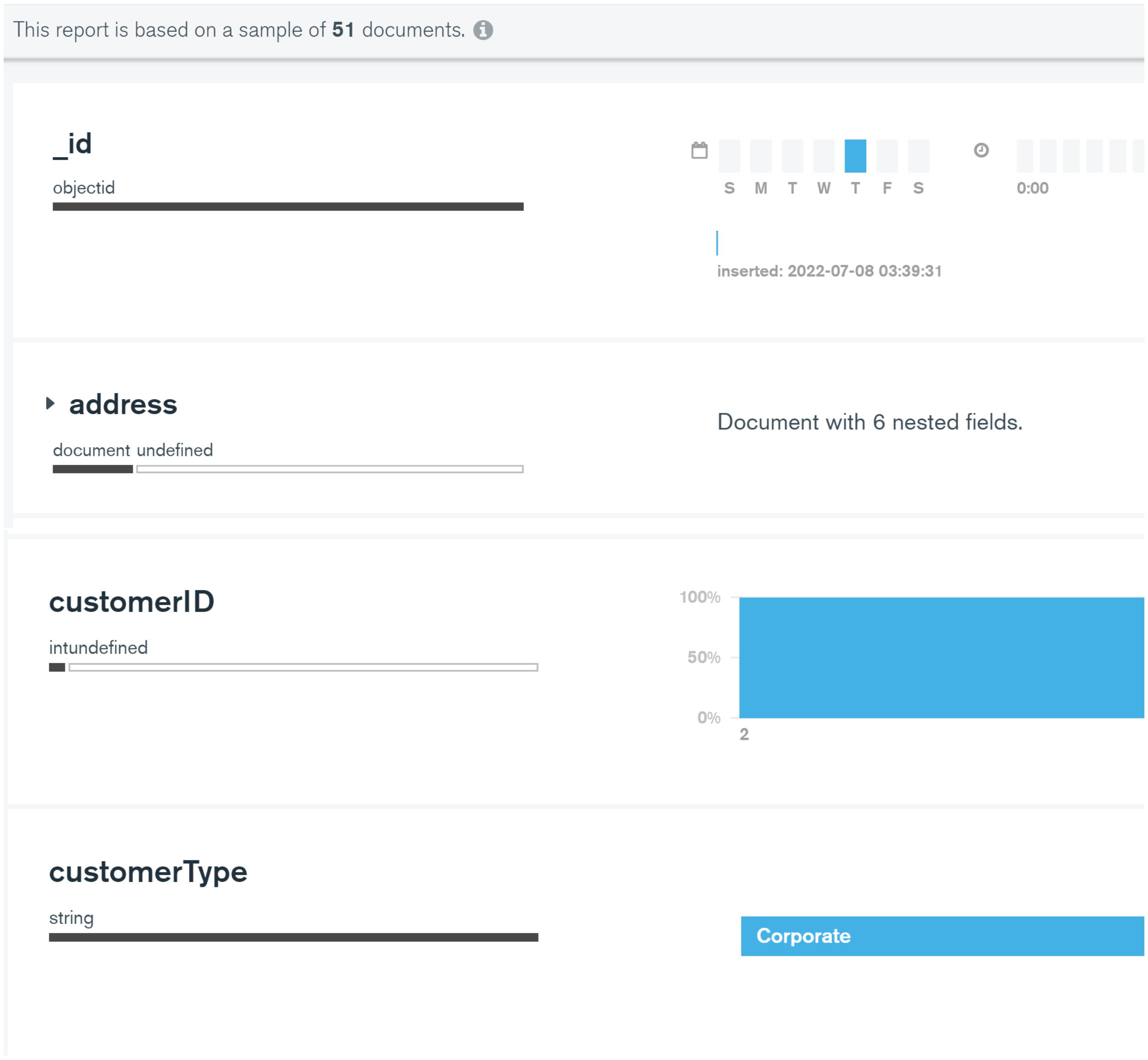
telephone	array: [object: {"number":"string"}, object: {"type":"string","area code":"string","phone":"string"}] array: [object: {"type":"string","area code":"string","phone":"string"}]		9	17.65%
favorites	array: [boolean, number, string] array: [string]		6	11.76%
manager	object: {"name":"string","hobbies":"array: [string]","remote":"boolean"}		6	11.76%
customerID	number		2	3.92%

Figure 5.25: Schemalysis Union Schema Output



```
integrationInput
customerType          str    *
status                multi
name                  str    *
customerID            num
address               obj
    street            str
    city              str
    state             str    *
    zip               num
    coordinates       arr
        [0]           num
    country           str
favorites             arr
    [0]               str
    [1]               num
    [2]               bool
manager               obj
    name              str    *
    hobbies           arr    *
        [0]           str
    remote            bool   *
telephone             arr
    [0]               obj
        type          str
        area code     str
        phone         str
        number        str
```

Figure 5.26: Hackolade Union Schema Output



This report is based on a sample of **51** documents.

_id
objectid

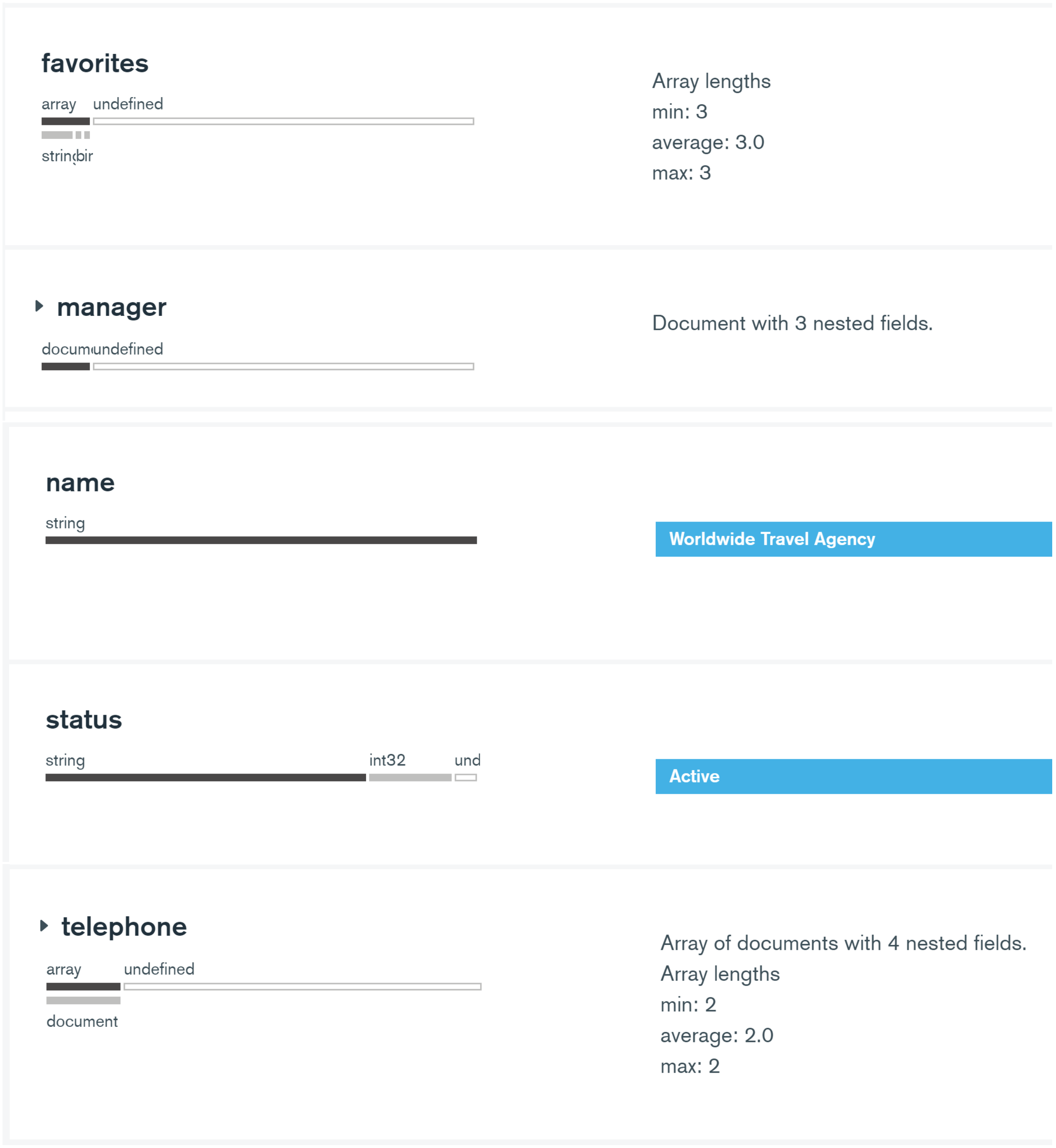
address
document undefined

Document with 6 nested fields.

customerID
intundefined

customerType
string

Figure 5.27: MongoDB Compass Schema Analyzer Output (part 1 of 2)



Array lengths
min: 3
average: 3.0
max: 3

Document with 3 nested fields.

Worldwide Travel Agency

Active

Array of documents with 4 nested fields.
Array lengths
min: 2
average: 2.0
max: 2

Figure 5.28: MongoDB Compass Schema Analyzer Output (part 2 of 2)

These three applications have different approaches to generating a union schema. Starting with Schemalysis, the union schema is displayed in a table. This table contains all attributes included in the database. The quantity and percentage for each attribute match the number of times that they appear in the database. The attributes that are included in every document are marked with an asterisk in the required field, and the attributes are sorted by their prevalence in the database. Additionally, attributes that are varied or appear as different types in the database contain multiple entries in the 'Type(s)' field to capture each type the attribute is represented as. Additionally, all variations of embedded objects are treated as separate types. This allows a user to see all types or variations that an attribute appears as.

Hackolade has a different approach to providing the union schema. The attributes that are included in every document are marked as required with an asterisk. However, for attributes that are not required, there is no indication of how prevalent they are. This could make it difficult for someone to tell the difference between an outlier attribute, or one that is only missing from a few documents. They also portray variance of embedded objects by providing a sub-profile of them. Because of this, there can be fields marked as required even if the attribute for the embedded object is not required, as shown in the manager and the address embedded object.

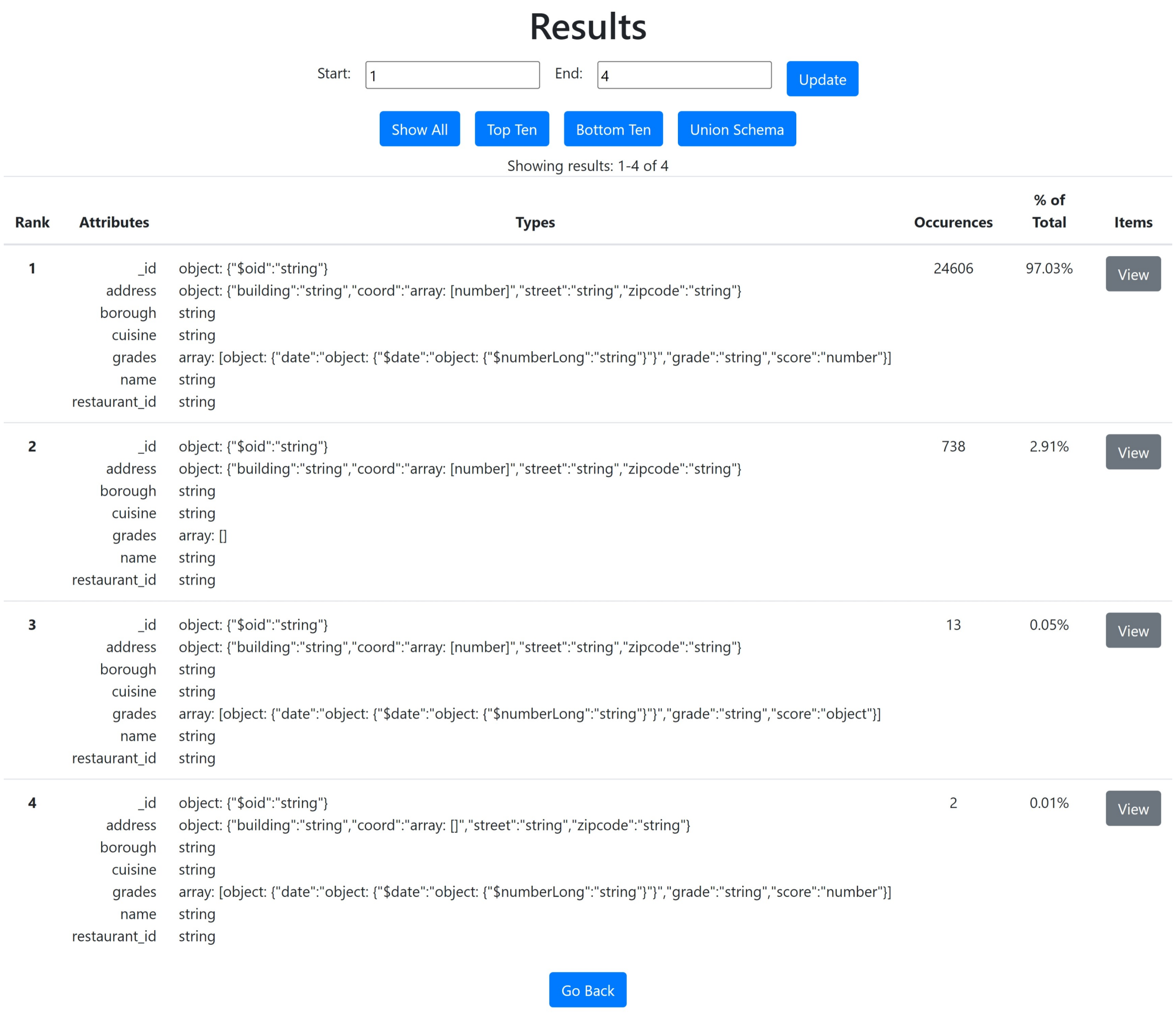
Finally, the MongoDB Compass Schema Analyzer has a similar approach to Schemalysis in terms of separating each attribute with a separate profile. However, MongoDB organizes the attributes in alphabetical order rather than in order of prevalence. For each attribute, it describes its prevalence in the database through a visual bar. For the proportion that the item is present, the bar is marked with the type of the attribute, and where it is missing, it is marked as undefined. Attributes can have multiple types and will have additional segments for each type. Arrays show basic info about the length of the arrays and a second bar is shown to display what types are in the array. Embedded objects are also profiled and a user can click it to view its sub-profile. MongoDB Compass also provides examples from the database for each attribute. This is different from the samples that Schemalysis provides since MongoDB provides examples of only the values of attributes while Schemalysis provides samples of documents that match a certain sub-schema.

5.2.3 Case Studies

To further verify the functionality of Schemalysis, we run the application on a few large datasets that contain real-world data. This tests the processing of many items that may contain unknown and unpredictable variations.

Case Study 1: New York Restaurants Data Set

The New York Restaurants data set is a document database of 25,359 items that contain information about restaurants in New York City, such as location and health inspection scores. It is retrieved from the sample database on MongoDB [11]. This database was chosen because it is mostly uniform with slight variation in lower level objects. This makes it a good candidate to test for cases when there is a large quantity of data that matches a single sub-schema. The result of the sub-schema analysis is shown in Figure 5.29, and the union schema is shown in Figure 5.30.


Results

Start: 1 End: 4 Update

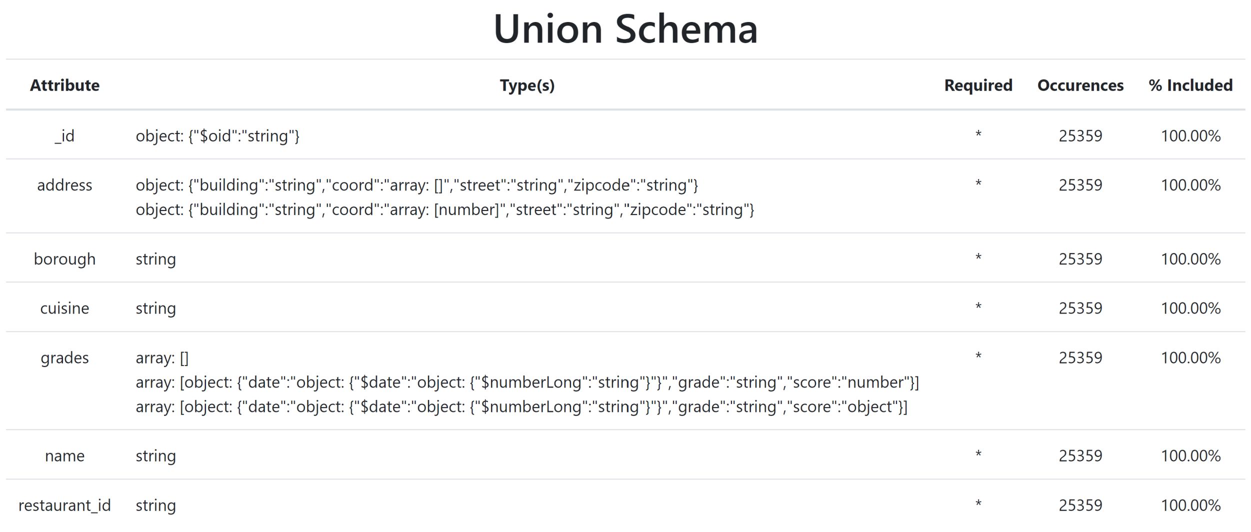
Show All | Top Ten | Bottom Ten | Union Schema

Showing results: 1-4 of 4

Rank	Attributes	Types	Occurences	% of Total	Items
1	_id address borough cuisine grades name restaurant_id	object: {"$oid":"string"} object: {"building":"string","coord":"array: [number]","street":"string","zipcode":"string"} string string array: [object: {"date":"object: {"$date":"object: {"$numberLong":"string"}"}","grade":"string","score":"number"}] string string	24606	97.03%	View
2	_id address borough cuisine grades name restaurant_id	object: {"$oid":"string"} object: {"building":"string","coord":"array: [number]","street":"string","zipcode":"string"} string string array: [] string string	738	2.91%	View
3	_id address borough cuisine grades name restaurant_id	object: {"$oid":"string"} object: {"building":"string","coord":"array: [number]","street":"string","zipcode":"string"} string string array: [object: {"date":"object: {"$date":"object: {"$numberLong":"string"}"}","grade":"string","score":"object"}] string string	13	0.05%	View
4	_id address borough cuisine grades name restaurant_id	object: {"$oid":"string"} object: {"building":"string","coord":"array: []","street":"string","zipcode":"string"} string string array: [object: {"date":"object: {"$date":"object: {"$numberLong":"string"}"}","grade":"string","score":"number"}] string string	2	0.01%	View

Go Back

Figure 5.29: Sub-schemas in the Restaurants Database



Union Schema

Attribute	Type(s)	Required	Occurences	% Included
_id	object: {"$oid":"string"}	*	25359	100.00%

address	object: {"building":"string","coord":"array: []","street":"string","zipcode":"string"}
	object: {"building":"string","coord":"array: [number]","street":"string","zipcode":"string"}
		*	25359	100.00%

borough	string	*	25359	100.00%
cuisine	string	*	25359	100.00%
grades	array: [] array: [object: {"date":"object: {"$date":"object: {"$numberLong":"string"}"}","grade":"string","score":"number"}] array: [object: {"date":"object: {"$date":"object: {"$numberLong":"string"}"}","grade":"string","score":"object"}]	*	25359	100.00%
name	string	*	25359	100.00%
restaurant_id	string	*	25359	100.00%

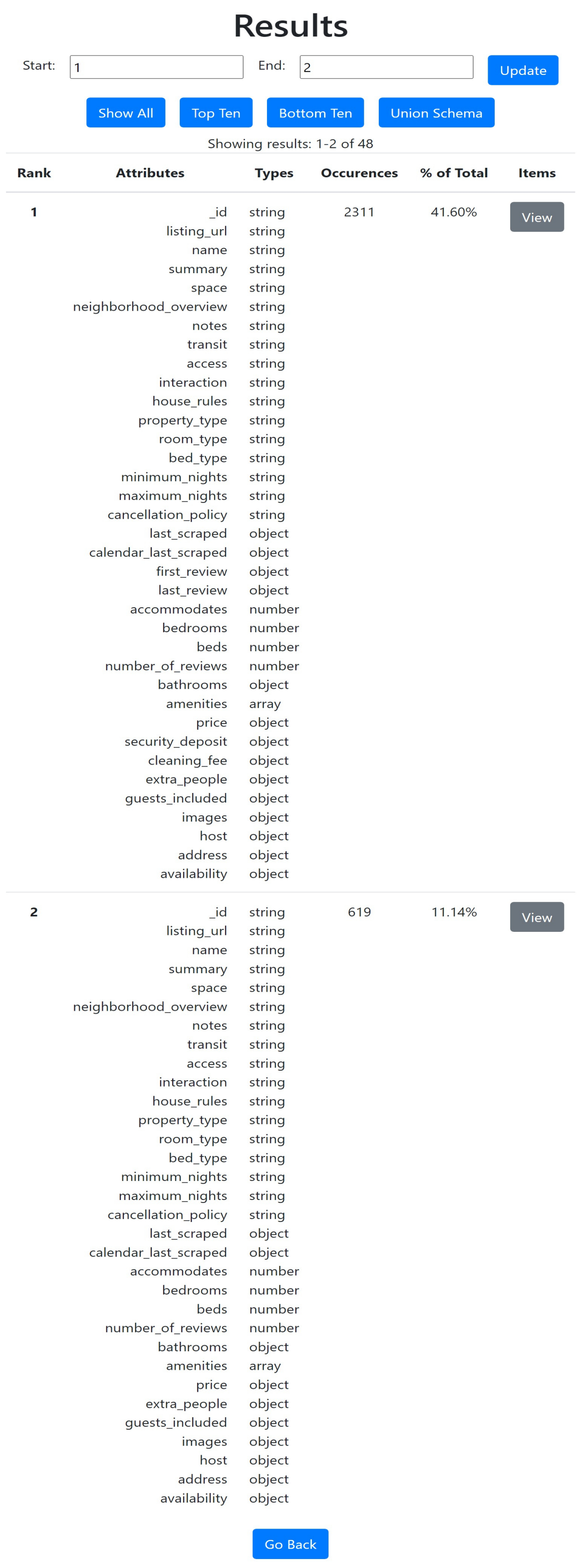
Figure 5.30: Union Schema of the Restaurants Database

From the results, we can see that all items have the same attributes, as shown in Figure 5.30. However, we also see that there is some variation in the arrays and embedded objects, but these are mostly outliers. It might be useful for a user to see that most of the data fits the same pattern, while some of the data has no information about health inspector grades, and a few outliers that are either missing certain information or have a slightly different structure.

Case Study 2: AirBnB Data Set

The AirBnB Data Set contains 5,555 documents with information about property rentals throughout several countries for the company AirBnB. It is retrieved from the MongoDB sample database [11] and is sourced from Inside AirBnB. The data contains large objects with many pieces of information about each rental property, such as the name, price, location, cancellation policy, bedrooms, house rules, *etc.* Several attributes have been removed from the AirBnB data set for this case study. The review_scores attribute is removed due to collision errors when exporting the documents from MongoDB. The description and reviews have also been removed because the values for this attribute in the database are extremely verbose, so when the sample documents are added to the schema profile, the profile becomes too large for JavaScript to store and retrieve in the results page.

This data set was chosen because it has a large amount of variation in its data. This makes it a good candidate to analyze the sub-schemas, and to locate attributes that are prevalent in the data and attributes that are only included in a few data objects. Due to the output size and complexity, we profiled this data set with only top level objects and show a small sampling of the sub-schemas.

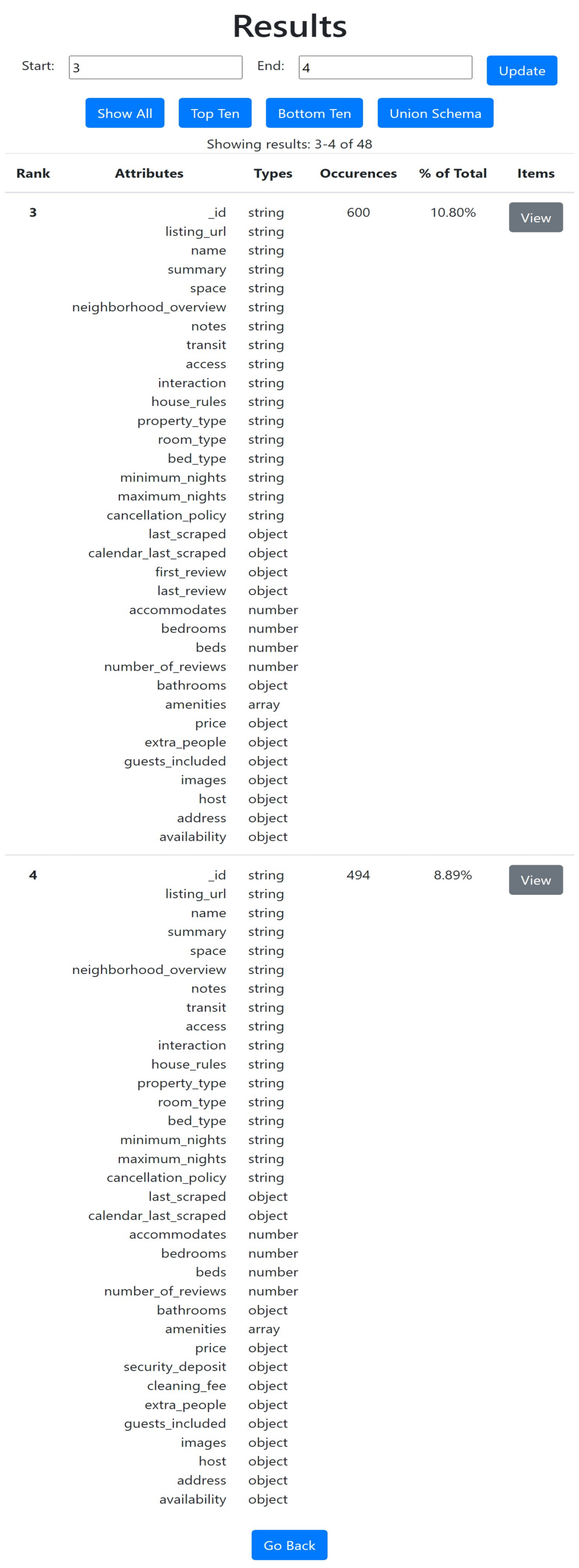


Results

Start: 1 End: 2

Showing results: 1-2 of 48

Rank	Attributes	Types	Occurences	% of Total	Items
1	_id	string	2311	41.60%	View
	listing_url	string			
	name	string			
	summary	string			
	space	string			
	neighborhood_overview	string			
	notes	string			
	transit	string			
	access	string			
	interaction	string			
	house_rules	string			
	property_type	string			
	room_type	string			
	bed_type	string			
	minimum_nights	string			
	maximum_nights	string			
	cancellation_policy	string			
	last_scraped	object			
	calendar_last_scraped	object			
	first_review	object			
	last_review	object			
	accommodates	number			
	bedrooms	number			
	beds	number			
	number_of_reviews	number			
	bathrooms	object			
	amenities	array			
	price	object			
	security_deposit	object			
	cleaning_fee	object			
	extra_people	object			
	guests_included	object			
	images	object			
	host	object			
	address	object			
	availability	object			
2	_id	string	619	11.14%	View
	listing_url	string			
	name	string			
	summary	string			
	space	string			
	neighborhood_overview	string			
	notes	string			
	transit	string			
	access	string			
	interaction	string			
	house_rules	string			
	property_type	string			
	room_type	string			
	bed_type	string			
	minimum_nights	string			
	maximum_nights	string			
	cancellation_policy	string			
	last_scraped	object			
	calendar_last_scraped	object			
	accommodates	number			
	bedrooms	number			
	beds	number			
	number_of_reviews	number			
	bathrooms	object			
	amenities	array			
	price	object			
	extra_people	object			
	guests_included	object			
	images	object			
	host	object			
	address	object			
	availability	object			



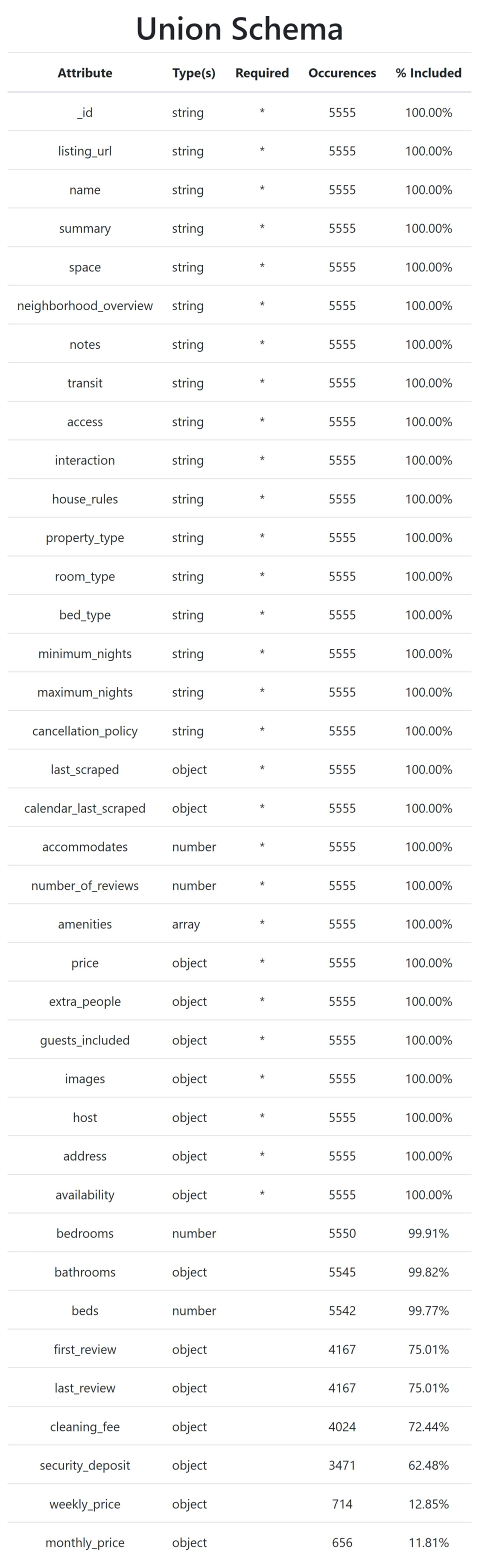
Results

Start: 3 End: 4

Showing results: 3-4 of 48

Rank	Attributes	Types	Occurences	% of Total	Items
3	_id	string	600	10.80%	View
	listing_url	string			
	name	string			
	summary	string			
	space	string			
	neighborhood_overview	string			
	notes	string			
	transit	string			
	access	string			
	interaction	string			
	house_rules	string			
	property_type	string			
	room_type	string			
	bed_type	string			
	minimum_nights	string			
	maximum_nights	string			
	cancellation_policy	string			
	last_scraped	object			
	calendar_last_scraped	object			
	first_review	object			
	last_review	object			
	accommodates	number			
	bedrooms	number			
	beds	number			
	number_of_reviews	number			
	bathrooms	object			
	amenities	array			
	price	object			
	extra_people	object			
	guests_included	object			
	images	object			
	host	object			
	address	object			
	availability	object			
4	_id	string	494	8.89%	View
	listing_url	string			
	name	string			
	summary	string			
	space	string			
	neighborhood_overview	string			
	notes	string			
	transit	string			
	access	string			
	interaction	string			
	house_rules	string			
	property_type	string			
	room_type	string			
	bed_type	string			
	minimum_nights	string			
	maximum_nights	string			
	cancellation_policy	string			
	last_scraped	object			
	calendar_last_scraped	object			
	accommodates	number			
	bedrooms	number			
	beds	number			
	number_of_reviews	number			
	bathrooms	object			
	amenities	array			
	price	object			
	security_deposit	object			
	cleaning_fee	object			
	extra_people	object			
	guests_included	object			
	images	object			
	host	object			
	address	object			
	availability	object			

Figure 5.31: Top 4 Most Prevalent Sub-schemas in the AirBnB Database



Union Schema

Attribute	Type(s)	Required	Occurences	% Included
_id	string	*	5555	100.00%
listing_url	string	*	5555	100.00%
name	string	*	5555	100.00%
summary	string	*	5555	100.00%
space	string	*	5555	100.00%
neighborhood_overview	string	*	5555	100.00%
notes	string	*	5555	100.00%
transit	string	*	5555	100.00%
access	string	*	5555	100.00%
interaction	string	*	5555	100.00%
house_rules	string	*	5555	100.00%
property_type	string	*	5555	100.00%
room_type	string	*	5555	100.00%
bed_type	string	*	5555	100.00%
minimum_nights	string	*	5555	100.00%
maximum_nights	string	*	5555	100.00%
cancellation_policy	string	*	5555	100.00%
last_scraped	object	*	5555	100.00%
calendar_last_scraped	object	*	5555	100.00%
accommodates	number	*	5555	100.00%
number_of_reviews	number	*	5555	100.00%
amenities	array	*	5555	100.00%
price	object	*	5555	100.00%
extra_people	object	*	5555	100.00%
guests_included	object	*	5555	100.00%
images	object	*	5555	100.00%
host	object	*	5555	100.00%
address	object	*	5555	100.00%
availability	object	*	5555	100.00%
bedrooms	number		5550	99.91%
bathrooms	object		5545	99.82%
beds	number		5542	99.77%
first_review	object		4167	75.01%
last_review	object		4167	75.01%
cleaning_fee	object		4024	72.44%
security_deposit	object		3471	62.48%
weekly_price	object		714	12.85%
monthly_price	object		656	11.81%

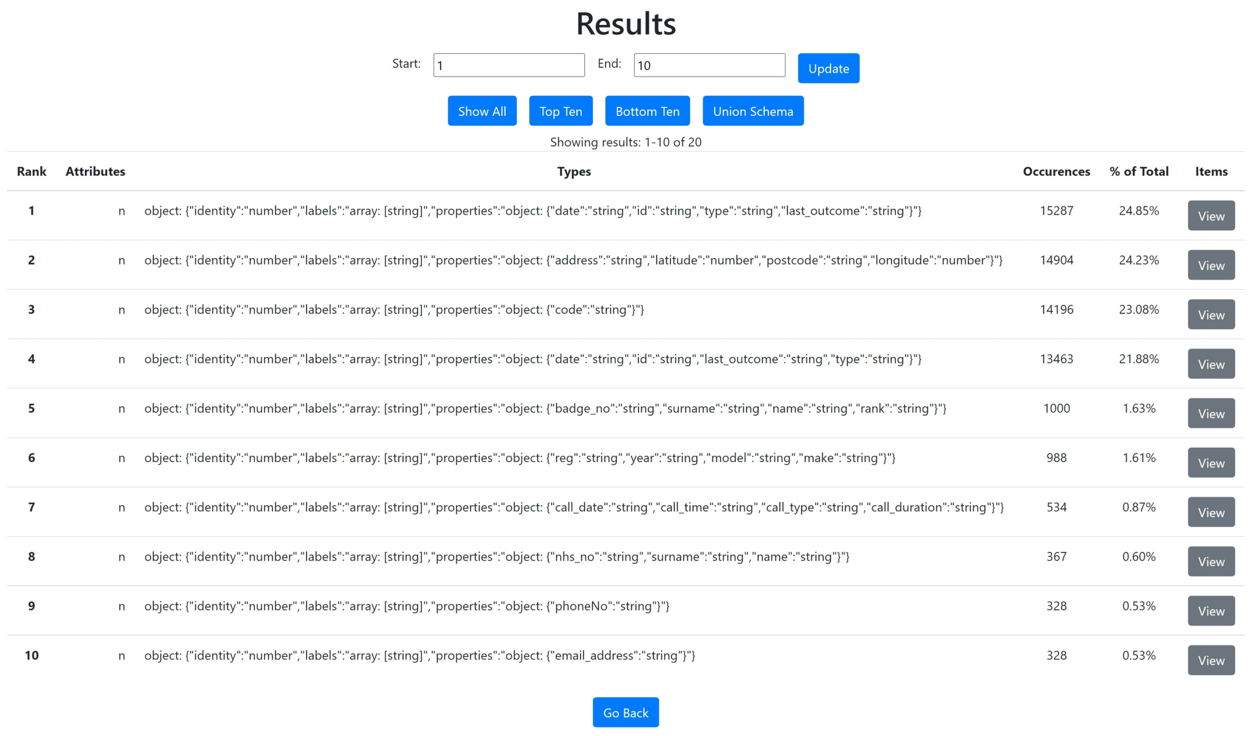
Figure 5.32: Union Schema of the AirBnb Database

As shown in Figure 5.31, this database contains 48 different sub-schemas. Due to the large numbers of attributes in each object and the high level of variation, a user might not find it useful to browse the many variations in the sub-schemas. However, the union schema, shown in Figure 5.32, can be used to analyze the prevalence of attributes to know what portion of the data has particular information. A user could use this information to isolate particular attributes they are interested in and investigate the sub-schemas that contain it.

Case Study 3: Neo4j Crime Data Set

The last case study is performed on a document database containing 61,521 items about crimes. This data set is retrieved from Neo4j sandbox [23]. This data uses crime data for Greater Manchester, UK from August 2017. However, information related to people involved, specific location, or other identifiable information is randomly generated or curated. It should also be noted that while this data originated as graph data including 12 node labels, 18 relationship types, and 32 property keys, it is exported as a JSON file and is treated as a document database in our research.

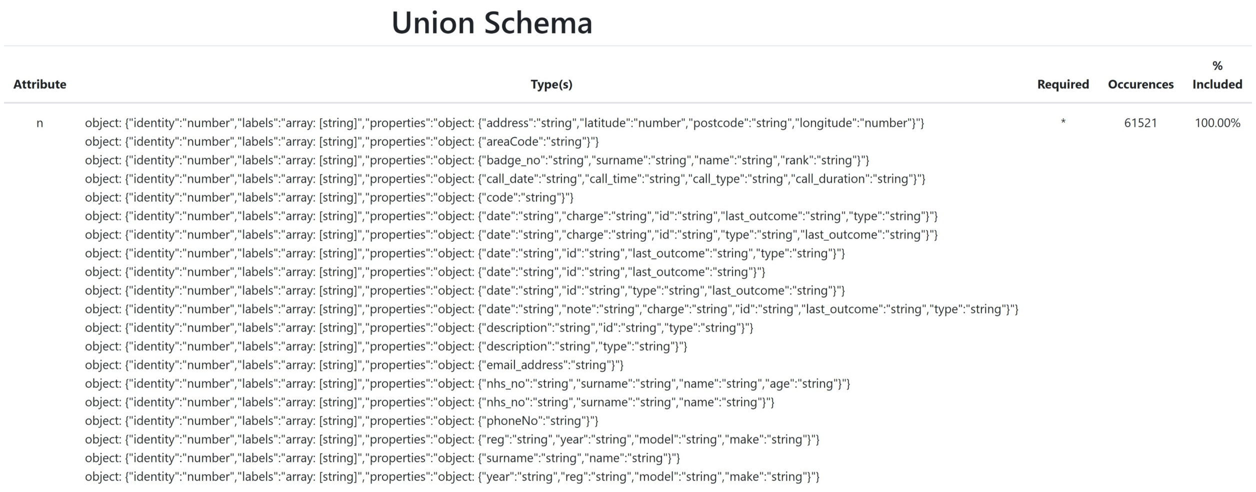
This data set was chosen as a case study due to the variation in lower level objects. Every data object contains only a single embedded object labeled n. However, there is extensive variation within this object, making it a good candidate to test the profiling of lower level objects. The result of the sub-schema analysis is shown in Figure 5.33, and the union schema is shown in Figure 5.34.



Showing results: 1-10 of 20

Rank	Attributes	Types	Occurences	% of Total	Items
1	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","id":"string","type":"string","last_outcome":"string"}"}	15287	24.85%	View
2	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"address":"string","latitude":"number","postcode":"string","longitude":"number"}"}	14904	24.23%	View
3	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"code":"string"}"}	14196	23.08%	View
4	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","id":"string","last_outcome":"string","type":"string"}"}	13463	21.88%	View
5	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"badge_no":"string","surname":"string","name":"string","rank":"string"}"}	1000	1.63%	View
6	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"reg":"string","year":"string","model":"string","make":"string"}"}	988	1.61%	View
7	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"call_date":"string","call_time":"string","call_type":"string","call_duration":"string"}"}	534	0.87%	View
8	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"nhs_no":"string","surname":"string","name":"string"}"}	367	0.60%	View
9	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"phoneNo":"string"}"}	328	0.53%	View
10	n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"email_address":"string"}"}	328	0.53%	View

Figure 5.33: Top 10 Most Prevalent Sub-schemas in the Crimes Database



Union Schema

Attribute	Type(s)	Required	Occurences	% Included
n	object: {"identity":"number","labels":"array: [string]","properties":"object: {"address":"string","latitude":"number","postcode":"string","longitude":"number"}"}	*	61521	100.00%
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"areaCode":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"badge_no":"string","surname":"string","name":"string","rank":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"call_date":"string","call_time":"string","call_type":"string","call_duration":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"code":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","charge":"string","id":"string","last_outcome":"string","type":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","charge":"string","id":"string","type":"string","last_outcome":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","id":"string","last_outcome":"string","type":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","id":"string","last_outcome":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","id":"string","type":"string","last_outcome":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"date":"string","note":"string","charge":"string","id":"string","last_outcome":"string","type":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"description":"string","id":"string","type":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"description":"string","type":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"email_address":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"nhs_no":"string","surname":"string","name":"string","age":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"nhs_no":"string","surname":"string","name":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"phoneNo":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"reg":"string","year":"string","model":"string","make":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"surname":"string","name":"string"}"}			
	object: {"identity":"number","labels":"array: [string]","properties":"object: {"year":"string","reg":"string","model":"string","make":"string"}"}			

Figure 5.34: Union Schema of the Crimes Database

This output reveals how Schemalysis behaves when there is extensive variation in an embedded object. As shown in the union schema in Figure 5.34, only a single attribute is present, with 100% inclusion in the database. However, the union schema also captures the different sub-schemas in the Type(s) column. It would be useful for a user to observe that the structure of most sub-schemas is roughly the same with the exception of the properties attribute, which appears to be the source of the variation. They could then use the information from the sub-schemas, shown in Figure 5.33, to investigate which structures of the properties attribute were the most prevalent.

5.3 Discussion

To provide discussions into the purposes of Schemalysis and the other applications, we have created Table 5.3 with columns of key functionalities corresponding to each application.

Table 5.3: Key Functionalities of Comparable Applications

Tool	Sub-schema Generation	Sub-schema Analysis	Union Schema Generation	Union Schema Analysis	JSON Schema Generation	View Sample Data	Interactive Model
Schemalysis	✓	✓	✓	✓		Document	✓
Hackolade			✓		✓		✓
MongoDB Compass			✓	✓		Attribute	
JSON Schema Generator			✓		✓	Document, Attribute	

5.3.1 Sub-schema Generation and Analysis

Sub-schema generation refers to whether the tool presents a list of the sub-schemas present in the database. Both Schemalysis and JSON Schema Generator provide a schema profile that considers the structure of individual documents rather than a global structure of all documents as a whole.

Schemalysis delves into each data object and retrieves its attribute names and types and compares it to the ones already found. It also keeps a count to find out the proportion of data matches a particular sub-schema. This is useful for finding outliers and erroneous data. It also has the ability to view a sample of the data that matches each sub-schema.

This improves upon the current applications that do not support a visual model to analyze the sub-schemas of a document database. While union schemas are useful to view the overall occurrence of attributes in a database, a sub-schema analysis gives insights into the different structures that objects can take and how often they appear. This structure visualization may allow users to better understand and query the database.

Examples of sub-schema analyses are found in the integration test in Figure 5.24, and the case studies in Figures 5.29, 5.31, and 5.33.

5.3.2 Union Schema Generation and Analysis

This functionality covers the generation of a union schema for a given set of data.

Schemalysis, Hackolade, MongoDB Compass, and JSON Schema Generator all generate a union schema for the data set. However, there are some key differences.

As shown in Figure 5.26, Hackolade provides a union schema with base information including the attributes included and their types, as well as whether or not they are required, or included in every single data object.

Schemalysis provides the same information in the union schema, as well as additional information about the number of times data objects include each attribute and the percentage of objects that contain the attribute, as shown in Figure 5.25. This extra information is useful in determining attributes that occur frequently in the data, but may be excluded in some objects. Compared to only knowing whether an attribute is required or not, knowing the prevalence of an attribute can determine if the attribute is an outlier rather than just missing in a few entries.

Additionally, Schemalysis provides more information on the types of attributes if there are multiple types for the same attribute in different objects. In Schemalysis, this is shown by extra information in the Type(s) column and includes each type, whereas in Hackolade, this is represented by listing the attribute as a "multi" type, as shown in unit test 11 in Figure 5.22.

MongoDB Compass also has a feature for analyzing the schema of a document database. When the schema analyzer is run from the interface, a profile of the union schema is given. This profile provides an analysis of each attribute represented in the database. As shown in Figures 5.27 and 5.28, each attribute has a bar that shows a visual representation of how the attribute appears, similar to how Schemalysis provides this as a percentage. This bar also shows the proportion of each type the attribute is represented as, or undefined if it does not exist in some objects. It also shows isolated examples of each attribute that appear in the database and additional information about arrays and embedded objects.

5.3.3 JSON Schema Validation

Hackolade and JSON Schema Generator both have the ability to generate a JSON Schema representation of the data. An excerpt of a JSON Schema is provided in Figure 5.23. The main difference between the two is that JSON Schema Generator creates a JSON Schema based off of the inputted document database, while Hackolade creates a JSON Schema based off of the current database model shown on the interface.

While this format can be used to view sub-schemas through the items attribute, the main purpose of generating a JSON Schema is validation and preventing future variation in data. Schemalysis is intended to provide a descriptive visual profile of the database's sub-schemas.

5.3.4 View Sample Data

Schemalysis, MongoDB Compass, and JSON Schema Generator allow a user to view sample data after generating a schema profile. However, the sample data shown is in a different format.

MongoDB Compass displays sample data for the union schema. This means that each attribute represented in the database is displayed with a list of example values that correspond to the attribute. This is useful for users to view the ways that attributes appear in the database.

Schemalysis displays sample data for each sub-schema in the database. Rather than examples for just each attribute, Schemalysis displays samples of entire documents. A user can select a particular sub-schema, and Schemalysis shows a sample of up to 10 documents that match the selected sub-schema. This is useful for users who wish to view samples of entire documents that match a particular structure.

5.3.5 Interactive Model

An interactive model refers to the ability of a user to change the view of the model. Schemalysis and Hackolade provide an interactive model as their output.

Schemalysis displays the results of the sub-schema analysis of the document database through an interactive table. With each sub-schema in the database being assigned a rank, a user can manipulate the table to only show which sub-schema ranks that they are interested in. Additionally, users can navigate from the sub-schema analysis page to a page viewing the database's union schema analysis, or to a page viewing the sample items of a particular database.

Hackolade is useful to view databases as a high level model. Once a model has been loaded into the Hackolade interface, it presents the database model in the form of an Entity Relationship (ER) Diagram. A user can then manipulate the diagram to modify the attributes or other characteristics of the data model. It can be used to forward and reverse engineer from a data file, as well as create a diagram to represent a database from scratch. This gives it a highly versatile use, but does not provide as much information as Schemalysis when used for reverse engineering.

The next chapter offers conclusions and future work.

Chapter 6

Conclusions

This chapter concludes our research regarding the development of an application to visualize the sub-schemas of document databases. We provide a summary of our work and contributions, reflect upon the research questions addressed in this thesis, discuss limitations of Schemalysis, and outline potential future work to expand upon this topic.

6.1 Summary

Our contribution through this research is an application that provides a visual model of the sub-schemas in a document database. In summary, we set out to improve the way that unstructured NoSQL document data is visualized. We investigated current research into the modeling and visualization of NoSQL databases. We created a web-based tool called Schemalysis that focuses on the analysis of both the sub-schemas and the union schema of a document database. Our approach involves the extraction of a schema from the individual documents, as well as a union schema of the database. We capture this information in a schema profile that provides a descriptive visual model of the structures in a document database. Finally, we validate the application to ensure correct functionality using test cases and case studies.

6.2 Research Questions

We reflect on the questions this research seeks to answer.

- RQ1: What reverse engineering techniques are applicable to document-based NoSQL data models and how can they be extended to create visual models?
- RQ2: What would an effective design be for a tool to support reverse engineering?
- RQ3: Does the model correctly and effectively evaluate unit tests and case study data sets to demonstrate functionality of the tool?

RQ1: We observed strategies used to generate a schema profile for both the sub-schemas and the union schema of a database through various related works. To generate a union schema, we implement a ladder based approach. We treat the first document as a union schema, then iterate through each other documents to update the union schema. Since no other schema profiling tools give the sub-schemas of a database, we extend the existing functionality by extracting the sub-schema of each document. In order to represent our schema profile as a visual model, we express both the sub-schemas and union schema as a JSON object that is used to populate a visual interactive table that a user can comprehend.

RQ2: We design Schemalysis using a web page that allows users to input either a file or entered text containing a document database. After performing the generation of the schema profile, the user is navigated to the results page where a user can view the sub-schema analysis of the application. This results page allows a user to view the sub-schemas of the database in an easy to understand way that allows users to see each sub-schema in the order of their occurrence in the database. Other information about each sub-schema is shown to allow users to better understand the structure of the documents contained in the database. The union schema shows a comprehensive view of the attributes in a database by listing each distinct attribute represented in the database shown in the order of their occurrence in the database.

RQ3: To validate the functionality of Schemalysis, we conduct unit tests to verify the correctness of the application when used with small inputs, integration tests to verify the correctness of the application when used with a larger and more diverse data set. Finally, we conduct three case studies to demonstrate the output of Schemalysis when used with large, real-world data sets.

6.3 Limitations

One key limitation of the application lies in the storage of the schema profile object in the application. The size of the schema profile object is limited by JavaScript's ability to store it. If there is a very large number of attributes, a high amount of variation, or attribute values are large or complex, it could prevent Schemalysis from being able to store the schema profile and display the results.

6.4 Future Work

We identify 6 main areas where the work reported here could be extended in the future.

The first area of future work consists of conducting a full user study on Schemalysis. This would collect any additional features or changes that would improve the experience for a wider audience. A user study would consist of finding a group of database administrators to test the application. The users would then test the application with sample data and data of their own in order to generate additional feedback to consider to improve the application.

A second area of future work is to extend the application to apply to graph data in addition to document data. This would extend the use of the application to a more broad range of databases.

A third area of future work is to investigate the optimization of the algorithm used to create sub-schemas and union schema in one pass instead of two. This would involve the consolidation of the ladder approach used to generate the union schema, and the iterative approach used to generate the list of sub-schemas. Since the method to extract the type of an attribute is recursive for nested objects, it would also require restructuring this method with additional parameters to differentiate the top level attributes and the attributes of lower level objects. This would prevent the attributes of lower level objects from appearing in the union schema.

A fourth area of future work is to investigate other uses for the schema profile object generated by our algorithm besides a visual model. The EasyQ approach to querying heterogeneous document data proposed by Hamadou *et al.* [2] includes a path dictionary used to rewrite queries to find attributes among different structures in the data. This approach may be able to integrate with a schema profile object to utilize the sub-schemas of the data to locate attributes and perform the query rewriting.

The fifth area of future work is to create a tool that stores a database of the sub-schemas and union in a database and provide an interface that allows a user to search for all possible places that an attribute or sub-schema occurs in the database. This is similar to the path dictionaries described in the EasyQ approach proposed by Hamadou et al. [2] and the EasyGraphQuery approach proposed by Malki *et al.* [3]. However, instead of an automatic query rewriting engine, this would be a descriptive tool used to observe the way that attributes appear in a database.

The last area of future research is to refine the object browser in Schemalysis for deeply nested objects. For functionality, it is currently represented as a string. This could be improved to more clearly convey the structure of deeply nested objects. One idea of how this could be improved is to change the way that embedded objects and arrays are displayed in the results so that they appear on multiple lines in a way that is more readable. This could also be collapsible with a click so that the user can expand nested attributes that they are interested in, similar to the Hackolade interface.

References

[1] A Czerepicki. Application of graph databases for transport purposes. *Bulletin of the Polish Academy of Sciences. Technical Sciences*, 64(3), 2016.

[2] Hamdi Ben Hamadou, Faiza Ghozzi, Andre Peninou, and Olivier Teste. Towards schema-independent querying on document data stores. In Il-Yeol Song, Alberto Abello, and Robert Wrembel, editors, *Proceedings of the 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with 10th EDBT/ICDT Joint Conference (EDBT/ICDT 2018), Vienna, Austria, March 26-29, 2018*, volume 2062 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[3] Mohammed El Malki, Hamdi Ben Hamadou, Max Chevalier, André Péninou, and Olivier Teste. Querying heterogeneous data in graph-oriented nosql systems. In Carlos Ordonez and Ladjel Bellatreche, editors, *Big Data Analytics and Knowledge Discovery - 20th International Conference, DaWaK 2018, Regensburg, Germany, September 3-6, 2018, Proceedings*, volume 11031 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 2018.

[4] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Umltographdb: mapping conceptual schemas to graph databases. In *International Conference on Conceptual Modeling*, pages 430–444. Springer, 2016.

[5] Gwendal Daniel, Abel Gómez, and Jordi Cabot. Umlto [no] sql: mapping conceptual schemas to heterogeneous datastores. In *2019 13th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–13. IEEE, 2019.

[6] Jihane Mali, Faten Atigui, Ahmed Azough, and Nicolas Travers. Modeldrivenguide: an approach for implementing nosql schemas. In *International Conference on Database and Expert Systems Applications*, pages 141–151. Springer, 2020.

[7] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. Schema profiling of document-oriented databases. *Information Systems*, 75:13–25, 2018.

[8] Engines ranking. `https://db-engines.com/en/ranking`.

[9] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the nosql world. *Computer Standards & Interfaces*, 67:103149, 2020.

[10] Pascal Desmarets. Data modeling tool for nosql, storage formats, rest apis, and json in rdbms, 2016.

[11] Mongodb compass. `https://www.mongodb.com/products/compass`.

[12] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh. Mda-based approach for nosql databases modelling. In Ladjel Bellatreche and Sharma Chakravarthy, editors, *Big Data Analytics and Knowledge Discovery - 19th International Conference, DaWaK 2017, Lyon, France, August 28-31, 2017, Proceedings*, volume 10440 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2017.

[13] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh. Umltonosql: Automatic transformation of conceptual schema to nosql databases. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 272–279. IEEE, 2017.

[14] Jacky Akoka and Isabelle Comyn-Wattiau. Roundtrip engineering of nosql databases. *Enterprise Modelling and Information Systems Architectures*, 13:281–292, 2018.

[15] Fatma Abdelhédi, Amal Ait Brahim, Rabah Tighilt Ferhat, and Gilles Zurfluh. Reverse engineering approach for nosql databases. In Min Song, Il-Yeol Song, Gabriele Kotsis, A Min Tjoa, and Ismail Khalil, editors, *Big Data Analytics and Knowledge Discovery - 22nd International Conference, DaWaK 2020, Bratislava, Slovakia, September 14-17, 2020, Proceedings*, volume 12393 of *Lecture Notes in Computer Science*, pages 60–69. Springer, 2020.

[16] Amal Ait Brahim, Rabah Tighilt Ferhat, and Gilles Zurfluh. Extraction process of conceptual model from a document-oriented nosql database. In *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, pages 1–5. IEEE, 2019.

[17] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. Variety-aware OLAP of document-oriented databases. In Il-Yeol Song, Alberto Abelló, and Robert Wrembel, editors, *Proceedings of the 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with 10th EDBT/ICDT Joint Conference (EDBT/ICDT 2018), Vienna, Austria, March 26-29, 2018*, volume 2062 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[18] J Ross Quinlan. *C4. 5: programs for machine learning.* Elsevier, 2014.

[19] Jennifer Zaino. Data modeling lends a hand to nosql databases. `https://www.dataversity.net/data-modeling-lends-hand-nosql-databases/#`, Aug 2016.

[20] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. Json schema: A media type for describing json documents, Jun 2022.

[21] Fatma Abdelhedi, Amal Ait Brahim, Rabah Tighilt Ferhat, and Gilles Zurfluh. Reverse engineering approach for nosql databases. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 60–69. Springer, 2020.

[22] Json schema generator. `https://www.jsonschema.net/`.

[23] Neo4j sandbox. `https://neo4j.com/sandbox/`, Dec 2021.