SCHEMA-AWARE INDEXES FOR JSON DOCUMENT COLLECTIONS

Thesis

Submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

UMA PRIYA D

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE - 575 025

April, 2023

DECLARATION

by the Ph.D. Research Scholar

I hereby declare that the Research Thesis entitled **Schema-Aware Indexes for JSON Document Collections** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfilment of the requirements for the award of the Degree of **Doctor of Philosophy** in Department of Computer Science and Engineering is a bonafide report of the research work carried out by me. The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.

Uma Priya D, 177057CO007

Department of Computer Science and Engineering

Place: NITK, Surathkal.

Date: April 18, 2023

CERTIFICATE

This is to certify that the Research Thesis entitled **Schema-Aware Indexes for JSON Document Collections** submitted by **Uma Priya D** (Register Number: 177057CO007) as the record of the research work carried out by her, is accepted as the Research Thesis submission in partial fulfilment of the requirements for the award of degree of **Doctor of Philosophy**.

Dr. P. SANTHI THILAGAM
Professor & Head
Department of Computer Science & Engineering
National Institute of Technology Karnataka, Surathkal
P.O. Srinivas... MANGALORE INDIA - 575025

Prof. P. Santhi Thilagam
18/04/2023
Research Guide

(Signature with Date and Seal)

24/04/23
Dr. Manu Basavaraju
Chairman - DRPC

(Signature with Date and Seal)

Chairman
~~DUGC~~ / ~~DPGC~~ / DRPC
Dept. of Computer Engg.
NITK - Surathkal
Srinivasnagar - 575 025

ACKNOWLEDGEMENTS

It is a great pleasure to thank the people who have supported and helped me so much throughout the period of my research. First and foremost, I would like to express my sincere gratitude to my research supervisor Dr. P. Santhi Thilagam, Professor, Department of Computer Science and Engineering, for constantly supporting me throughout my research journey with her knowledge, experience, and patience. I deeply thank her for her valuable guidance, timely suggestions, and advice. Beyond the subject area of this thesis, I have learned a lot from her that, I am sure, will be useful in the later stages of my career as well as my life.

I would like to extend my sincere thanks to the members of my Research Progress Assessment Committee (RPAC) members, Dr. M. Venkatesan and Dr. Jaidhar C. D., for their insightful feedback and constructive suggestions throughout my research work. I would also extend my sincere thanks to the Doctoral Research Programme Committee (DRPC) members and other faculties of the Department of Computer Science and Engineering for their help and support throughout my research. Nevertheless, I am also grateful to the technical and administrative staff of the department for their timely help and cooperation in carrying out the research work.

I would like to thank NITK for providing the infrastructure and facilities for the smooth conduct of my research. I also thank Mr. Varun and Mr. Arjun for their useful discussions and the support they extended during this period.

I would like to extend my gratitude to my friends Dr. D.V.N. Sivakumar, Dr. Srinivasa K, Dr. Amit Praseed, Mrs. Rashmi Adyapady, Dr. Alkha Mohan, Mr. Saran Chaitanya, Mrs. Sujitha, Mr. Ajnas, Mr. Zubair, Mrs. Lavina, Mr. Kondaiyaa, Mr. Siva Kumar for their constant support and encouragement throughout the course of my research. I would like to express my gratitude to my other fellow doctoral students and

friends for their constructive criticism and valuable cooperation.

This thesis would not have been possible without the exceptional support of my family. I am deeply indebted to my husband, Mr. Gourish, who encouraged me to pursue a Ph.D. and has been a constant pillar of emotional and motivational support till the end of my Ph.D. I am equally thankful to my son Jai Dharshan, father Mr. Dharmaraj, mother Mrs. Poongodi, sister Mrs. Arun Priya, and in-laws Mrs. Vasantha and Mrs. Siva Priya for being with me all through and being very supportive towards the course of study. I am highly obligated to my family members for all their sacrifices for my better future.

Finally, I would like to thank all of them whose names are not mentioned here but have helped me in some way to accomplish the work. I would like to thank the Almighty for granting me the health, strength, and wisdom to carry out my research work.

Uma Priya D

ABSTRACT

Web applications, IoT devices, and other real-time applications generate an abundance of multi-structured data every day, increasing the complexity of data storage and management. Large organizations such as Amazon, Google, and Facebook use NoSQL databases to store these large sets of diverse data. NoSQL databases offer an efficient architecture for meeting the performance and scale requirements of big data compared to relational databases. NoSQL document stores adopt the JSON format as the de-facto standard for storing multi-structured data. The *data first, schema later* approach of document stores greatly enhances the use of the JSON data format in modern applications. However, this flexibility poses several challenges for data management and knowledge discovery tasks.

A JSON collection does not have an explicit schema to describe the internal structures of documents; instead, the schema is implicit in the data, allowing the documents to have various structures. Therefore, knowledge of the implicit schemas is essential to understand the data stored in the collection. This schema information can be helpful for efficient data retrieval, data integration, query formulation, etc. In this direction, existing research extracts schemas from JSON documents using their structural relatedness and generates either global schema or schema variants. The global schema is the structural representation of the whole collection that summarises the unique attributes in a collection. This information is generally used for JSON document validation, query formulation, etc. As the global schema does not capture the different sets of attributes available in each document, it does not support various data management tasks such as data integration, query optimization, etc. To overcome this limitation, few studies focus on extracting schema variants from the collections. Schema variants represent the schema versions or distinct schemas of JSON collections that support the above-mentioned data management tasks effectively. Most literature focuses on extracting the schema versions from a collection using schema class types (entities) manually embedded in the documents. Due to the dynamic nature and sheer size of JSON documents,

the manual embedding of class types in each document is not feasible in a real-time scenario. To address this issue, researchers employ clustering approaches to automatically identify the class types of a JSON collection in two steps. The primary step is to extract the schemas from a collection and then cluster the documents using the structural similarity of extracted schemas. However, differently annotated JSON schemas are not only structurally heterogeneous but also semantically heterogeneous. Literature shows that the automatic identification of class types of JSON documents based on structural and semantic similarity of JSON schemas is still in its infancy. To address these research gaps, this research employs both syntactic and semantic relationships of JSON schemas to capture the contextual information. In this work, we propose (i) *Schema Embeddings for JSON Documents* (*SchemaEmbed*) model to capture the contextually similar JSON schemas, (ii) *Embedding-based Clustering* approach to group the contextually similar JSON documents, and (iii) *Schema Variants Tree* (*SVTree*) to represent the schema variants of each cluster. As *SVTree* contains information about the core (common) and schema-specific attributes in a cluster, it supports efficient data retrieval. The proposed approach is evaluated with real-world and synthetic datasets. The results and findings demonstrate that the proposed approach outperforms the current approaches significantly in grouping the contextually similar JSON documents. In addition, the impact of clustering in constructing a compact *SVTree* is also studied.

The heterogeneous nature of JSON documents increases the complexity of the efficient retrieval of data. Indexes have traditionally been used to improve the speed of data retrieval. Existing indexing techniques for JSON data use global schema to identify the unique attributes in a collection and support exact (lexical) matching of path-based queries. However, they suffer from huge index sizes and data retrieval time. As JSON schemas are annotated differently, providing semantic support increases the search relevancy. Existing work on the semantic search of JSON documents uses knowledge bases such as WordNet. However, they capture the abstract meaning of JSON attributes rather than their context. To bridge these research gaps, this research proposes efficient and compact index structures, namely *JSON Index* (*JIndex*) and *Embedding-based JIndex* (*EJIndex*), to support both lexical and semantic matching of path-based queries. With

the help of core and schema-specific attributes of schema variants stored in *SVTree*, the proposed indexes reduce the index size by storing only a subset of attributes rather than all the attributes in a collection. Experimental results demonstrate that the proposed indexes outperform the existing approaches in retrieving both lexically and semantically relevant results, significantly reducing index size and data retrieval time.

As JSON documents evolve and change over time, the implicit schemas must be extracted and updated in the database to support dynamic data retrieval. Existing approaches focus either on maintaining the history of schema versions in data lakes or updating the global schema. Nevertheless, the schema variants must be updated to provide the latest documents for the user queries. In this work, we propose an *Incremental SchemaEmbed* model to generate schema embeddings for new schema variants of the latest documents while preserving the knowledge of old schema variants. The *Incremental Embedding-based Clustering* approach assigns the latest documents to the respective clusters based on the contextual similarity of their schema variants. Consequently, the *JIndex* and *EJIndex* are updated incrementally to support the retrieval of the latest documents for the user queries. The experimental results on diverse datasets show that the proposed work is efficient in updating the schema variants and the indexes.

Keywords: JSON, Schema extraction, Schema variants, JSON Indexing, Semantic search

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviations	Expansion
ACID	Atomicity, Consistency, Isolation, and Durability
AMI	Adjusted Mutual Information
ARI	Adjusted Rand Index
API	Application Programming Interface
CAS	Content and Structure
BSON	Binary JavaScript Object Notation
BSP	Build Schema Profile
DB	Davies-Bouldin
DBLP	DataBase systems and Logic Programming
DTD	Document Type Definition
ELMo	Embeddings from Language Models
ETL	Extract, Transform, and Load
GenICHARE	Generalized CHAin Regular Expression with Interleaving
GKS	Generic Keyword Search
GML	Geography Markup Language
HBaSI	HBase Schema Inference
IMDb	Internet Movie database
IoT	Internet of Things
JSON	JavaScript Object Notation
k-OIRE	k-occurrence Interleaving Regular Expression
KML	Keyhole Markup Language
LCA	Lowest Common Ancestor
LTRe	Learning To Retrieve
MAP	Mean Average Precision
MDE	Model Driven Engineering
NMI	Normalised Mutual Information
OLAP	OnLine Analytical Processing
RDBMS	Relational Database Management Systems
RDF	Resource Description Framework
RoBERTa	Robustly optimized BERT Approach
SC	Silhouette Co-efficient
TF-IDF	Term Frequency - Inverse Document Frequency

Abbreviations	Expansion
XML	eXtensible Markup Language
XSD	XML-Schema
UCIS	Updatable Compact Indexing Scheme
UQP	Unified Query Processing
USE	Universal Sentence Encoder
WSD	Word Sense Disambiguation

CHAPTER 1

INTRODUCTION

The advancement of web applications, online businesses, the emergence of social network organizations, and the adoption of hand-held computerized gadgets, Internet of Things (IoT) devices, etc., contribute to the phenomenal growth of semi-structured and unstructured data. As per SeedScientific (2021) statistics, more than 44 zettabytes of data are being generated every day, and this number is expected to reach 463 exabytes of data globally in the coming years. Managing such huge semi-structured and unstructured data in relational databases is challenging because it does not support scalability, schema evolution, efficient data storage, management, etc. The schema-less nature of NoSQL databases provides the flexibility to store and manage massive multi-structured data in a distributed way. In addition, NoSQL databases offer efficient architecture in meeting the performance and scale requirements of big data compared to traditional relational databases. Most large organizations such as Google, Twitter, Microsoft, Amazon, Facebook, and others use NoSQL databases to store and manage this huge semi-structured and unstructured data efficiently.

Four types of NoSQL databases, namely key-value stores, document stores, column stores, and graph databases, are popularly used to manage semi-structured and unstructured data. Although all the types of NoSQL databases share standard features such as schema flexibility, scalability, etc., the structure of the data stored in these databases and their applications vary significantly. Key-value stores represent each data element as a key-value pair consisting of an attribute name and a value. Document stores employ the

Table 1.1: Types of NoSQL databases and their features

Features	NoSQL Databases			
	Key-value stores	Document stores	Wide-column stores	Graph databases
Storage Type	Collection of key-value pairs	Documents using JSON, BSON, XML data formats	Standard-column family, Super-column family	Property graphs, RDF graphs
Querying	No query language Use CRUD (create, retrieve, update, and delete) commands for key and value	custom query language for each database like MQL, N1Q1, etc.	custom query language for each database like CQL, Jaspersoft, etc.	custom query language for each database like Cypher, Gremlin, etc.
Added support for JSON format	✓	provides native support	✓	✓
Examples	Redis, Amazon DynamoDB, etc.	MongoDB, Couchbase, etc.	HBase, Cassandra, etc.	Neo4j, OrientDB, etc.
Applications	Shopping carts, user profiles, etc.	Content management systems, IoT devices, Real-time data analytics, etc.	Logging, storing sensor logs, user preferences, etc.	social networks, fraud detection, knowledge graphs, etc.

concept of key-value stores, in which data is stored in the form of documents. Columnar databases organize data into columns. Graph databases store data in graphs where the nodes represent the data elements, and the edges represent their inclusive relationships. The key characteristics of the NoSQL databases are given in Table 1.1.

Over the years, eXtensible Markup Language (XML) has been the standard data interchange format over the web and is used to represent semi-structured data generated by web applications. Recently, JSON has become the de facto format for data interchange over the web because JSON is lighter, simpler to read, and faster to parse than XML (Liu et al. 2014). The JSON format also supports faster communication between web applications than XML. For instance, it is observed from Table 1.2 that XML format has redundant opening and closing tags to represent the *author* information. In contrast, JSON syntax is minimal, which makes JSON format lighter and faster. JSON also provides intrinsic data type support, which is not supported in XML. The significant differences between XML and JSON formats given in Table 1.2 make JSON as the primary data interchange format over the web. Moreover, large organizations, like Google, Yahoo, Facebook, Twitter, etc., use JSON to store and manage multi-structured data in NoSQL databases.

1.1 JSON DOCUMENT REPRESENTATION

JSON is a text-based data format that uses JavaScript object syntax to describe data in a structured format. A JSON document collection is a group of documents within a database. It is formally defined in Definition 1.1.1.

Definition 1.1.1. A Collection $G = \{D_1, D_2, ..., D_n\}$ is a set of JSON documents where the document D_i is a JSON object.

JSON documents (objects) are dictionaries comprised of key-value pairs in which the value may be another JSON object, allowing for an arbitrary nesting level. Besides simple dictionaries, JSON supports arrays and primitive types such as string, number, and Boolean. As dictionaries and arrays can include another JSON object, the JSON format is compositional. Definition 1.1.2 provides a formal definition of a JSON document.

Table 1.2: Features of XML and JSON data formats

Features	Data formats	
	XML	JSON
Document	Made up of XML elements where attributes describe the elements	Made up of a set of unordered attributes
Document Representation	Tree (hierarchical)	Map (key-value pairs)

Structure of a document	<root>
	<author>
	<firstName>John</firstName>
	<lastName>Smith</lastName>
	</author>
	<author>
	<firstName>Noam</firstName>
	<lastName>Zeilberger</lastName>
	</author>
	<book>
	<title>Trademaker</title>
	<year>2014</year>
	</book>
	<publisher>Science of Science</publisher>
	</root>
		{
		”author ”:
		[{
		”firstName”: ”John”,
		”lastName”: ”Smith”
		},
		{ ”firstName”: ”Noam”,
		”lastName”: ”Zeilberger”,
		}],
		”book”:
		{
		”title”: ”Trademaker”,
		”year ”: 2014
		},
		”publisher”: ”Science of Science”
		}

Datatypes	text, number, images, charts, graphs, etc.	primitive data types: text, number, boolean complex data types: object, array
Schema	XSD and DTD	JSON-Schema
Ordering	Enforces a strict order between the nodes at each level	Mixes both ordered and unordered data
Uniqueness of Keys	XML elements are non-deterministic	JSON attributes are deterministic
Attribute value	simple data types such as number, text, etc.	both primitive and complex data types

Definition 1.1.2. A JSON document (object) D is made up of a set of unordered fields (key-value pairs) $\{F_1, F_2, ..., F_m\}$. A field *F* has a key (also known as attribute) k_a and the value v_a where the type of the value can be represented as *type*(v_a) = $\{type^{prim} \cup type^{comp} \cup$ *null*$\}$. The primitive data types, $type^{prim} \in \{$*string, number, boolean*$\}$, and the complex data types $type^{comp} \in \{$*array, object*$\}$ where *array* represents the ordered lists of values and *object* is an unordered set of key-value pairs.

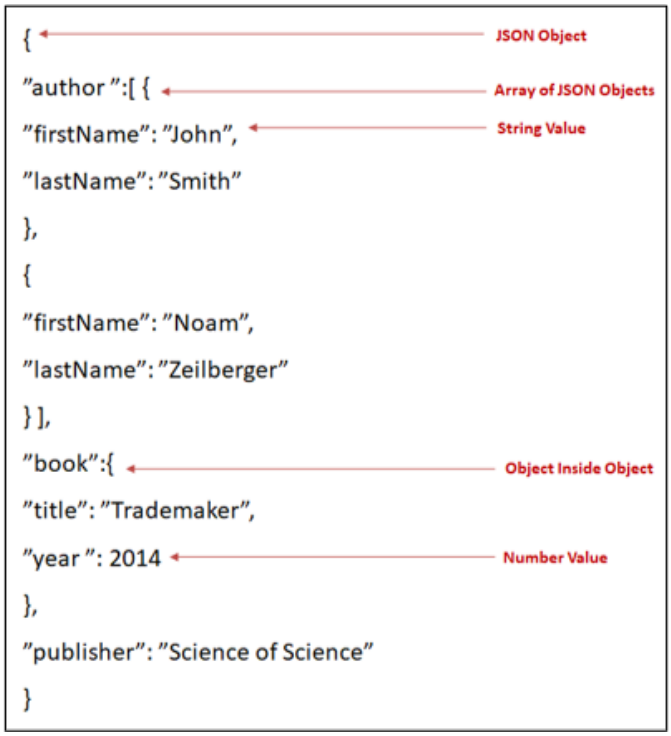


Figure 1.1: A Sample JSON document

Figure 1.1 depicts an instance of a JSON document D which has three keys at the first level, such as *author*, *book*, and *publisher*. The value of the key *author* is of an array type with two JSON objects (also known as an array of objects). In addition, the key *book* has another JSON object with two fields, such as *title* and *year*. The value of the key *publisher* is *Science of Science* with string type.

The JSON document can also be represented as a tree form in two different ways:

- **Node-labeled tree:** The nodes (vertices) represent the JSON attributes, and the edges represent the inclusive relationship between the attributes. JSON arrays are modeled as nodes, and their children are accessed via nodes labeled with numbers that correspond to their position in the array. Defining arrays in this way allows for preserving the structure of an array of objects as well as supporting efficient data access by navigating the child nodes (Shang et al. 2021).

- **Edge-labeled tree:** The leaf nodes represent the JSON value, and edges represent the JSON attributes. JSON arrays are represented as nodes, and their children are accessed by axes labeled with numbers matching their array position. (Bourhis

et al. 2017). As edges carry the key information, constructing space-efficient indexes to support efficient data retrieval is a daunting task.

In this work, JSON documents are represented as node-labeled trees in order to preserve all the features of JSON format as well as to support efficient data retrieval.

Figure 1.2 depicts the tree representation of a JSON document shown in Figure 1.1. The root of a JSON tree describes the whole document. However, the JSON format does not have a specific root attribute as in XML. Therefore, in this work, the common *Root* attribute is created for a JSON document. The *author* attribute is of array type with two objects. The nodes labeled with 0 and 1 represent the child objects of *author*.

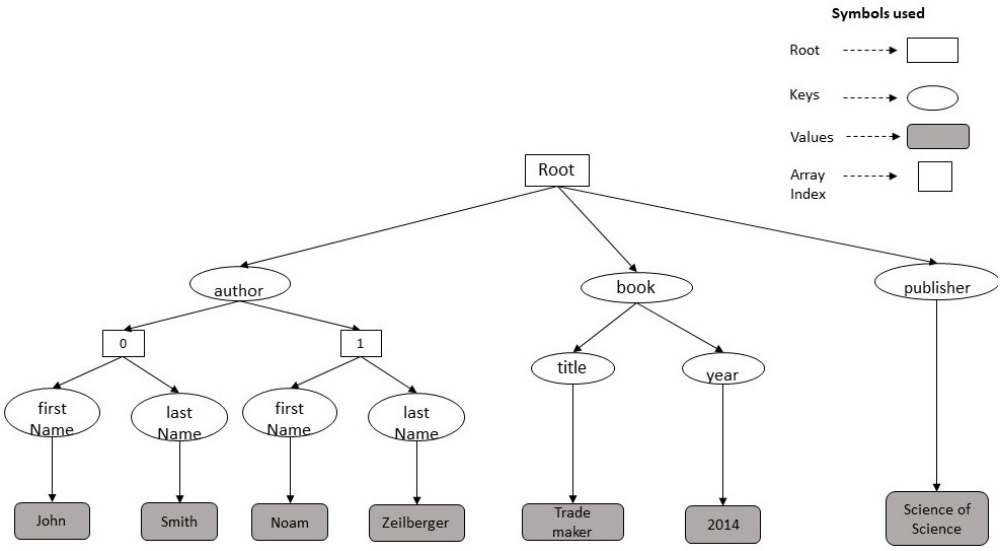


Figure 1.2: Tree representation of a JSON document in Figure. 1.1

1.2 JSON SCHEMA EXTRACTION

NoSQL database systems have emerged rapidly in recent years because of their schema-less nature. The *data first, schema later* approach of NoSQL document stores provides flexibility in storing multi-structured documents in JSON format. However, this flexibility poses several challenges for data management and knowledge discovery tasks. A JSON collection does not have an explicit schema to describe the internal structures of documents; instead, the schema is implicit in the data, allowing the documents to have various schemas. Therefore, knowledge of the implicit schemas is essential to un-

derstand the data stored in the collection. This schema information can be helpful for efficient data retrieval, data integration, query formulation, etc.

The schema represents the structural description of documents stored in the database. Schema extraction is the primary challenge in NoSQL database management and is referred to as the process of extracting the implicit structural description of a JSON document. Schema extraction has already been addressed in the context of various data formats, such as XML (Mlỳnková 2008) and Resource Description Framework (RDF) data (Kellou-Menouer et al. 2021). Although XML and JSON share similar properties like tree-shaped data format, self-describing nature, etc., the vital differences in Table 1.2 make the XML schema extraction approaches challenging to apply to JSON format (Bourhis et al. 2017). For instance, the XML schemas are generated in the form of regular expressions from XML elements. Whereas a JSON document has both ordered (arrays) and unordered (nesting object) data, which must be represented as XML elements in order to use the XML schema extraction approaches for JSON data. While the structures of XML elements and JSON arrays are different, XML schema extraction approaches may not always capture the complete structural information of a JSON document. Therefore, schema extraction approaches for JSON data must preserve the features of a JSON document, such as unordered dynamic data, different nesting levels, arrays, and its deterministic nature.

JSON schemas are useful for data organization, indexing, integration, query formulation, optimization, big data analytics, etc. Therefore, researchers have started to pay attention to schema extraction approaches that facilitate schema generation in the absence of an explicit one. The existing literature on JSON schema extraction has focussed primarily on extracting schemas in three different forms:

1. **Extraction of Global Schema:** Global schema is constructed by taking the union or intersection of JSON attributes present in a collection. However, the global schema may miss prominent attribute information, such as capturing different data types for the same attribute. The loss of such information causes incorrect results for user queries. Hence, the global schema is less informative to appli-

cation developers and users to analyze the implicit structure of data (Gallinucci et al. 2019).

2. **Extraction of Reduced Schema:** While the data type of JSON attributes changes the structure conceptually, the reduced schema is derived by fusing the objects based on equivalent data types (Baazizi et al. 2019).

3. **Extraction of Schema Variants:** Schema variants represent the distinct schemas of JSON collections. The schema variants summarize the co-occurrence of the attributes present in each document that promote data management tasks like data integration, data migration, etc. (D and Santhi Thilagam 2022).

Some of the challenges that are encountered while extracting schemas from JSON collection are highlighted below:

- JSON documents are generated from unknown sources with no detailed information about the data. This is crucial for the data analyst to search for schema information either in the underlying application code or in the data itself (Klettke et al. 2016).
- A JSON document contains both ordered (array) and unordered (nesting object) data, which introduces high structural heterogeneity in a collection.
- A document contains the same attribute corresponding to more than one data type (Habib et al. 2019).
- Schemas of the same entity in a collection might also vary due to data evolution (Sevilla Ruiz et al. 2015).
- Using simple solutions such as intersection or union of all schemas does not work well in practice (Wang et al. 2015).
- There is no unique generic algorithm for JSON schema extraction, which is crucial for researchers to select an appropriate algorithm based on the type of schemas they discover.

Due to these challenges, developing a generic schema extraction approach for JSON documents is difficult. Hence, JSON schema extraction approaches must be formulated for each application based on the type of schemas to be extracted while preserving the features of the JSON format.

1.3 APPLICATIONS OF JSON SCHEMA EXTRACTION

This section discusses the different uses of JSON schema extraction.

1. **Data Exploration:** Data exploration is an important application for JSON documents. Due to the dynamic and heterogeneous nature of JSON documents, database administrators, developers, and researchers are often confronted with different schemas present in a collection about which they do not have prior knowledge. Therefore, in order to understand what data is stored in a collection, the schema extraction approach helps in exploring the knowledge of the implicit schemas (Gómez et al. 2016).

2. **Query Formulation:** The flexibility of schema-less data models introduces complexity in data retrieval. In order to retrieve the required information from a collection, the user must formulate a query with the attributes that exist in the collection. Therefore, the schema extraction approach has the scope to fulfill the user requirements to formulate a query. Most NoSQL document stores, such as MongoDB (mongodb schema 2019), Couchbase (Couch Spark Connector 2019), etc., have adopted this method to ease query processing. For instance, in MongoDB, the schema is represented as a Binary Javascript Object Notation (BSON) document that follows the hierarchical structure like a BSON object. In order to build user-defined indexes, the users must have knowledge about the implicit schema of a collection, which will help them to write the path of an attribute when writing a query (Zhang et al. 2019). In the absence of schema information, query preprocessing and reformulation techniques are needed to retrieve the relevant results for a user query (Tekli et al. 2019). These techniques are computationally extensive due to the dynamic and heterogeneous nature of JSON documents.

3. **Query Optimization:** In general, the query engine uses different schemas for query planning and optimization. The queries are divided into sub-queries, and the query optimizer uses the sub-queries to create an optimized query execution plan (Quilitz and Leser 2008). As JSON collection has different schemas, the co-occurrence of attributes in each schema assists in writing the optimized query execution plan.

4. **Distributed Query Decomposition:** As NoSQL databases are designed for large-scale data storage in a distributed environment, the documents in a collection are stored on different servers. To support efficient retrieval of this data, the queries must be distributed to the servers to fetch the relevant data with less query processing time. For instance, in the MongoDB distributed environment (mongodb schema 2019), the documents are distributed to the shards based on shard keys and partitioning methods. Config Server maintains the metadata information about each shard, which helps the mongos (query router) direct the query to the respective shards. For example, consider a query $Q = \{author, journal\_name\}$. Suppose the query does not contain a shard key; then mongos broadcast the query to all the shards and retrieve the results. Broadcasting queries to all shards (broadcast operation) increases the query processing cost. In order to minimize the number of broadcast operations, instead of sending the query to every shard, the query can be decomposed into sub-queries in such a way that the query attributes can be sent only to the specific shards containing the query attributes instead of directing them to all the shards (Quilitz and Leser 2008). To accomplish this task, it is required to know what data is stored in each shard, the different schemas followed in each shard, and so on.

5. **Data Integration:** Data integration is a long-standing research area in database management. It is crucial when the data needs to be integrated from different sources. For instance, the tourism management system needs to integrate data about hotels, tourist places, and so on. However, JSON data is generated from different sources with different schemas. In order to build a complete application, the varied structured data must be integrated. This integration provides a global

virtual schema for all the data sources so that the database can be accessed in a unified way (Amghar et al. 2019; Curé et al. 2011).

6. **Big Data Analytics:** Data lakes are the large-scale central repository for storing relational and non-relational data. A lack of knowledge about the structure and semantics of data stored in a data lake would facilitate inefficient data access. In order to address this issue, data scientists and business analytics use various machine learning and data discovery techniques to extract the knowledge from data stored in a data lake (Klettke et al. 2017). In this direction, schema extraction helps in understanding the underlying data so that efficient data access can be performed.

7. **Data Organization:** To improve the scalability of a database management system, the documents are distributed horizontally across the cluster in a distributed environment (Couch Spark Connector 2019; mongodb schema 2019). The common way to partition the data into shards is traversal-based partitioning for graph databases and a key range or key hash strategy for other data models. However, these partitioning strategies encounter an issue related to user query latency. i.e., the number of requests to many partitions for a user query is high. A possible solution is to group the similar JSON documents based on structural or semantic similarity of schemas and store each group in each shard (Priya and Thilagam 2022). This kind of partitioning method helps reduce query latency by forwarding the query requests to the specific shards rather than all the shards. Therefore, knowing the implicit schemas of each shard reduces the query latency effectively.

8. **Data Indexing:** In general, index structures help in improving the performance of data retrieval. Schema extraction methods help the user to know the different attributes that exist in a collection. While designing the application, this schema information helps the user to create an index on primary attributes, such as common and schema-specific attributes present in the documents, that improve the overall data retrieval performance (Liu et al. 2014).

1.4 MOTIVATION

Most NoSQL systems are based on a distributed architecture, where the document collection is partitioned intelligently across the instances in the cluster. Even though most schema-less NoSQL stores provide flexibility when importing data, formulating relevant queries requires full knowledge of the underlying document schemas that promote efficient query processing. In this direction, major NoSQL document stores such as CouchBase (Couch Spark Connector 2019), MongoDB (mongodb schema 2019), and third-party tools such as variety.js (2019), Elastic Search (2019), Apache Drill (2019), etc. have already employed some notion of schema extraction mechanisms to understand the underlying structures. In addition, researchers extract various forms of schemas from the JSON collection, such as global schemas (Chouder et al. 2017; Frozza et al. 2018; Klettke et al. 2015; Wang et al. 2015), and reduced schemas (Baazizi et al. 2017, 2019). These approaches provide the summarization of attributes in a collection that would fail to assist various tasks such as efficient query processing, data integration, better data organization, etc. Therefore, there is a need for an approach capable of identifying schema variants in JSON collections to summarize the co-occurrence of the attributes that promote the aforementioned data management tasks effectively.

Despite the schema variants in a collection, the ability to provide fast and flexible access to data is a crucial problem in the database management system. Although the data is organized in a systematic way, scanning the whole dataset for each access is expensive. Index structures play a major role in speeding up the search process. The most widely used indexing structure for keyword-based queries is the Inverted Index (Hsu and Liao 2020), which keeps a map of an attribute and its value list. The literature on existing indexing schemes in related fields, such as XML, uses labeling schemes to index documents by preserving the structural relationships of XML documents. Consequently, the content index includes the content list for every node in the path index, which continues to suffer from a large index size, resulting in increased query processing time (Dhanalekshmi and Asawa 2018; Hsu and Liao 2020). Only very few studies exist for JSON documents to provide memory-efficient index structures by storing the

keys, values, and long strings in different indexes (Shang et al. 2021). Nevertheless, the huge posting list size for every JSON attribute still increases the storage space. Therefore, there is a need to improve the performance of JSON data retrieval by considering the index storage space and query processing time. Hence, this work proposes compact index structures using extracted schema variants that open up opportunities to address the above-discussed issues effectively.

JSON documents are not only structurally heterogeneous but also semantically heterogeneous. However, existing research on indexing JSON documents focuses on providing the exact matches for the given query (Budiu et al. 2014; Shang et al. 2021; Shukla et al. 2015). The major challenge in a traditional information retrieval system is a vocabulary-mismatch problem, which occurs when documents relating to a particular query are not retrieved because different terms are used to represent the same attribute. To address this issue, extant research in XML data provides semantic support on indexes by various methods, such as query rewriting, post-processing query results by re-ranking, etc. (Tekli et al. 2019). However, these additional processes further increase the complexity of data retrieval. In this work, we focus on processing JSON path-based queries efficiently by considering the semantic relationships of JSON schemas. However, the standard semantic measures of words using traditional knowledge bases like WordNet (Miller 1995) are not efficient in revealing the context of the query keyword. Therefore, in this work, we use the neural embedding-based index to facilitate context-based semantic search.

As JSON documents evolve and change over time, the implicit schemas must be extracted and updated in the database to support dynamic data retrieval. Most existing approaches focus on maintaining the history of schema versions in data lakes (Klettke et al. 2017, 2016; Scherzinger et al. 2013). In order to provide the latest documents to user queries, there is a need to update the schema variants. Therefore, this work proposes an approach to handle the dynamic data by updating the schema variants and indexes effectively.

In summary, this work is primarily concerned with schema variant extraction from JSON document collections and constructing compact index structures to support effi-

cient data retrieval. Our motivation for conducting this research is threefold:

1. As the JSON attributes are annotated differently in the collection, there is a need to organize a large JSON collection based on the contextual similarity of JSON schemas. The exact schema variants in each cluster must be identified to support the efficient retrieval of data.

2. There is a need to construct compact and efficient index structures for JSON collection. The indexes must support both lexical and semantic matching of path queries, related to which no study has been conducted until now.

3. There is a need to update the schema variants and indexes incrementally to handle the dynamic JSON data.

1.5 ORGANIZATION OF THE THESIS

The rest of the thesis is structured as given below:

- Chapter 2 provides a taxonomy of JSON schema extraction approaches and a detailed survey of existing research work that deals with the schema extraction approaches and tools for NoSQL data models. The related works of JSON and XML indexing techniques are also provided. Furthermore, the chapter provides a list of research challenges in JSON schema extraction.

- Chapter 3 describes the research problem and objectives of the thesis.

- Chapter 4 describes how the schema variants are extracted and summarized in the proposed data structure. The chapter also provides the experimental evaluation of our approach in terms of efficiency and effectiveness.

- Chapter 5 discusses how the schema variants are used to construct compact index structures for JSON documents. The chapter also discusses how the proposed scheme exploits the lexical and contextual relationship of queries with JSON documents for efficient data retrieval. Lastly, the experimental results of our proposed work are analyzed.

- Chapter 6 discusses how the schema variants and the index structures can be updated incrementally to support the dynamic nature of JSON documents. The performance of the proposed work is presented along with the necessary analysis and discussion.
- Chapter 7 summarizes the research work presented in this thesis and provides some insights into future work.

CHAPTER 2

LITERATURE REVIEW

The majority of the literature related to schema extraction on semi-structured data is concerned with XML and RDF data. This is due to the fact that they were the popular data interchange format over web applications. Recently, JSON has been used as the de facto format for data exchange in almost all web applications. JSON data format started gaining public attention only in the past decade. In fact, the first mention of schema extraction on JSON data only appeared in literature in 2013 (Cánovas Izquierdo and Cabot 2013). Since then, few research works have attempted to understand and propose different approaches to infer the schema.

The flexibility of the schema-less nature of NoSQL Databases ends up with major challenges in querying the data stored. Even though the document stores, such as MongoDB, CouchDB, Terrastore, etc., do not follow the fixed schema to store the data, they require a knowledge of the underlying structures for retrieval of data. Since XML and JSON allow one to represent tree-shaped data, this chapter includes the recent related works of schema extraction and indexing methods of XML and JSON data formats. The first section presents the schema extraction approaches for XML data. The second section discusses the methods and tools for JSON schema extraction and presents the research challenges associated with schema extraction. The third section presents a brief review of the significant research works on structural indexing of XML and JSON data with strong attention to structure-based and structure and content-based queries.

2.1 XML SCHEMA EXTRACTION

The problem of schema extraction from a given schema-less data has been studied for a decade, mainly for XML and JSON data formats. Even though XML and JSON are two distinct data formats, both of them can be described as hierarchical semi-structured data formats at a high level. This section describes the recent related works of schema extraction on XML data formats.

Almost all the research works on XML schema extraction rely on two popular methods, such as the heuristics approach and grammar inference. The research works on these methods can be differentiated by way of constructing a result and its type. In a heuristics approach, the result does not belong to any class of grammar. In grammar inference methods, the result belongs to a particular class of language with specific features. The early efforts on schema extraction from XML documents uses the above methods to learn the restricted classes of regular expressions as content models of the Document Type Definition (DTD) or XML-Schema Definition (XSD) (Abelló et al. 2018; Bex et al. 2006; Li et al. 2019, 2018; Mlýnková and Nečaský 2009; Wang and Chen 2019). For unordered content models (Ciucanu and Staworko 2013; Maatuk et al. 2015), it is required to generate all the possible combinations of dependencies among elements, resulting in an exponential number of candidate solutions. Each combination of elements in a collection can define an unexpected solution, which should be identified and verified for its existence in the class hierarchy. Thus, the process is expensive concerning performance and processing.

Table 2.1 shows the summary of the recent works on XML schema extraction. Zhang et al. (2018) proposed Generalized CHAin Regular Expression with Interleaving (GenICHARE) algorithm to analyze the conciseness of regular expressions and infer the schema from unordered XML documents. However, this algorithm works for a single occurrence of elements in a document. Only the work proposed by Abelló et al. (2018) generated DTD for unordered XML elements considering the name and type of an element. However, the element with a different parent in the same or different document is not considered. Conversely, this feature is the primary cause of heterogeneity. Janga and Davis (2019) proposed a grammar-based approach to generate both schema

Table 2.1: A comparison of existing research works on XML schema extraction methods

Research Article	Ordered/ Un-ordered	Schema-match	Outcome	Compatibility to JSON format
Ciucanu and Staworko (2013)	Unordered	name	Global Schema	Learning restricted classes of regular expressions from XML elements is challenging to apply to JSON arrays, and preserving the deterministic nature of JSON format is crucial.
Maatuk et al. (2015)	Ordered	name, type	Global Schema	
Abelló et al. (2018)	Unordered	name, type	Global DTD	
Li et al. (2018)	Ordered	name	Regular Expressions	
Zhang et al. (2018)	Unordered	name	Regular Expressions	
Janga and Davis (2019)	Unordered	name, type	DTD & XSD	
Li et al. (2020)	Unordered	name, type	Regular Expressions	

languages XSD and DTD. Unlike other existing works focussed on schema extraction on XML homogeneous collections, Janga and Davis (2019) supports schema extraction on heterogeneous XML collections. Li et al. (2020) inferred k-occurrence interleaving regular expressions (k-OIREs) by supporting interactive schema design and recommendation. k-OIREs are extracted from both positive and negative samples of XML data. To support efficient schema design, the author has used word embedding models that capture the semantics of the context and helps in predicting the XML elements while designing the schema.

It is noted from Table 2.1 that most existing works focused on extracting schemas from unordered data. Although it is equivalent to the unordered nature of JSON documents, XML and JSON formats have vital differences. As seen in Table 1.2, (i) JSON document consists of arrays data type which is not present in XML format (ii) the value of any XML element is of primitive types such as string, number, etc. However, the value of a JSON attribute could be another JSON object or array. While the XML schema extraction methods typically learn the schemas as restricted classes of regular

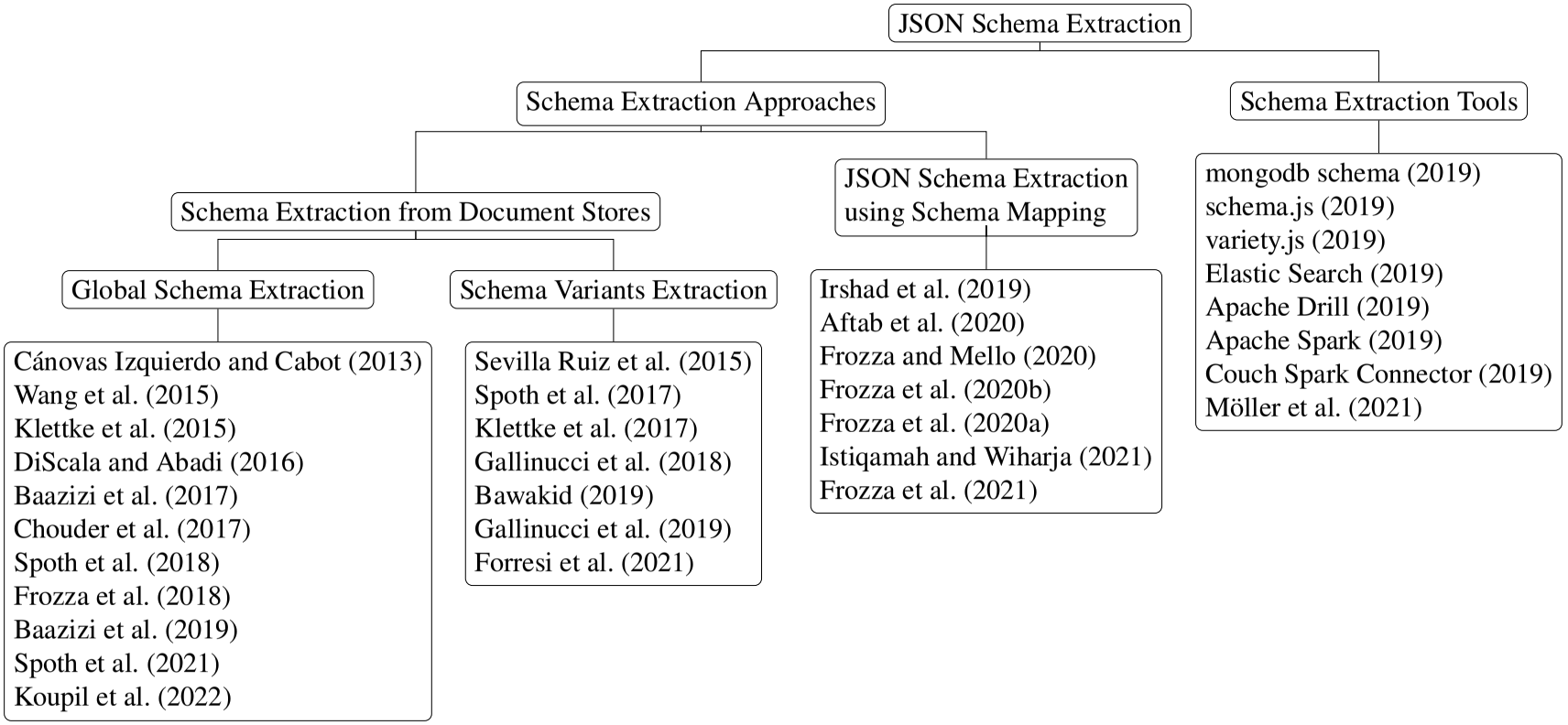


Figure 2.1: Taxonomy of schema extraction approaches and tools on JSON Data

expressions, the complex data structures such as arrays in JSON format introduce complexity in learning the regular expressions. Therefore, it is concluded that the schema extraction methods of XML format are difficult to apply to JSON.

2.2 JSON SCHEMA EXTRACTION

In the past decade, many approaches and tools have been developed for schema extraction in different application scenarios. Therefore, there is an increasing demand for a systematic review of the work done in the field of schema extraction on JSON data. This section presents a taxonomy of the schema extraction approaches and tools available for JSON data.

2.2.1 Approaches

With the popularity of JSON in web and real-time applications, JSON not only acts as the primary format to store and manage data in document stores but also the canonical data representation for other NoSQL databases, such as wide column stores, key-value stores, and graph databases. Therefore, the various approaches that have been used for schema extraction can be categorized into two groups:

1. *Schema extraction from document stores:* It describes schema extraction from JSON documents stored in document stores. Based on the nature of the output, schema extraction approaches can be classified into two classes:

    - Global schema extraction
    - Schema variants extraction

2. *Schema extraction using schema mapping:* It discusses the approaches that generate JSON Schema from NoSQL databases other than document stores, such as columnar databases, relational databases, and so on. The schemas generated from these databases are transformed into JSON schemas using schema mapping techniques.

The taxonomy of the schema extraction approaches and tools is presented in Figure 2.1.

A summary and comparison of the schema extraction approaches for generating global schema and schema variants are presented in Tables 2.2 and 2.3. The survey tables are constructed based on the features of the existing approaches, such as their ability to handle arrays, JSON object representation, methodologies used, and so on.

- **Arrays:** A key characteristic of JSON data types is arrays. An array is a complex data type and is an ordered set of primitive or complex data types. Arrays of objects and arrays of arrays contribute to high schema heterogeneity and introduce complexity in schema design.

- **Schema-match measure:** In general, a JSON document collection comprises different schemas. In order to identify the uniqueness of the two schemas, the attribute name, types, and values are compared. The schemas are merged based on the attribute information.

- **Data Representation:** Extracting the structure information and analyzing them to generate global schema or schema variants requires the data must be represented in some form. In this context, JSON documents could be considered as a naive key-value pair, the root-to-leaf path of attributes, meta models, trees, graphs, vectors, and so on.

- **Clustering/Classification:** In order to analyze the different properties of JSON attributes that contribute to schema heterogeneity, clustering algorithms or classification techniques were used by some existing works.

- **Approach:** To explore the underlying structure of JSON documents, existing research works have used several approaches such as Model Driven Engineering (MDE), probabilistic approaches, hierarchical summarization of schemas, map-reduce based approach, and so on.

- **Output:** Although JSON Schema[1] is a specification to define the structure of JSON documents, various approaches have defined their output in different forms such as skeleton schema, r-schema, c-schema, schema variants, and so on. This

[1] http://json-schema.org/

is due to the fact that the JSON data format does not have a formal data model, and few researchers have laid the formal foundation for the JSON Schema and the query languages (Baazizi, Mohamed-Amine and Colazzo, Dario and Ghelli, Giorgio and Sartiani, Carlo 2019; Bourhis et al. 2017) recently.

2.2.1.1 Global Schema Extraction

Global schema generates a single schema for a document collection that is adequate to describe the structure of attributes present in a collection. The global schema is constructed by merging multiple individual schemas into a single schema using schema integration techniques. Table 2.2 provides a review of the existing research works by considering specific features of JSON data such as arrays and schema-match measures.

Cánovas Izquierdo and Cabot (2013) proposed the unified meta-model for the JSON objects generated by the same or different Application Programming Interface (API) services. The model generates a global schema by taking the union of attributes from each service. Wang et al. (2015) proposed a concept of *skeleton* that clusters similar schemas and generates a summarized representation of heterogeneous schemas. However, a skeleton is constructed based on the frequency of attributes, and hence skeleton may lack certain attributes that are included in some documents but are not considered in skeleton construction. In addition, the path information of the attributes may not be captured by the skeleton and the cost of skeleton construction is high. Klettke et al. (2015) aims at measuring the heterogeneity of schema in a collection. Two graph structures are proposed to model the union of all attributes in JSON object collection. Baazizi et al. (2017) determined the structural irregularities of the JSON data and produced the global schema as a result. Chouder et al. (2017) find the multi-dimensional structures by mining the approximate functional dependencies of data. Spoth et al. (2018) focussed on visualizing the schema summary as data guides to the user, and the schema can be further refined based on user feedback. This is different from his previous approach in providing a summary of a schema collection rather than multiple schemas and considers the attribute values as objects, including arrays.

There are many research works (Baazizi et al. 2019; Gallinucci et al. 2018, 2019;

Table 2.2: Summary of existing research works in the field of global schema extraction from JSON data

Research Article	Arrays	Schema-match	Clustering/ Classification	Data Representation	Approach	Output	Applications
Cánovas Izquierdo and Cabot (2013)	×	name, type, values	×	Meta Model	MDE	Class Diagram	Data Exploration, Query Formulation
Wang et al. (2015)	×	sub trees	Clustering	Tree	Canonical Form based method	E-Sibu Tree	
Klettke et al. (2015)	✓	name, type	×	Graph	Extract Schema and append in a graph	Structural Identification Graph	
DiScala and Abadi (2016)	✓	name	×	Functional Dependencies	Transform denormalized, nested data in to normalized data	Relational Schema	
Chouder et al. (2017)	✓	name	×	Functional Dependencies	Mining approximate functional dependencies from data	c-schema	
Spoth et al. (2018)	✓	name, type	×	Tree-based data guides	Hierarchical Summarization	Relational Schema	
Frozza et al. (2018)	✓	name, type	×	Models	Extract raw schemas and build a tree	Global-Tree Schema	
Baazizi et al. (2019)	✓	name, type	×	Key-Value Pair	Apache Spark based distributed algorithm	Reduced Schema	
Spoth et al. (2021)	✓	name, type	Clustering	Key-Value pair	heuristics-based algorithm	Compact global schema	
Koupil et al. (2022)	✓	name, type	×	Key-Value Pair	Apache Spark based schema inference approach	Reduced Schema	

Sevilla Ruiz et al. 2015) considered the type of values to find the similarity between schemas. An enhanced work by Baazizi et al. (2019) goes a step further by inferring the schema for a massive collection along with type equivalence. The author uses two equivalence relations to fuse the objects based on the name and type of the JSON fields and generate the reduced schema such as kind equivalence, and label equivalence. Kind equivalence fuses types if two field values are objects or basic types or arrays. Label equivalence combines types only if the set of keys is the same. Though the approach is flexible and parallelizable, it is still restrictive. However, equivalence relations provide reduced schemas rather than an informative schema.

DiScala and Abadi (2016) addressed the issue of automatically converting the denormalized JSON documents into normalized relational data using the schema generation algorithm. The algorithm uses functional dependencies of attributes and learns the normalized relational schema from the JSON documents suitable for storing in relational databases. Frozza et al. (2018) generated schema for JSON and extended JSON documents. The authors have proposed a hierarchical structure for summarizing the schema. While other existing approaches generate global schema by considering the properties of JSON data format, Spoth et al. (2021) proposed a schema discovery system called JXPLAIN, which generates a tight and descriptive schema by reducing the ambiguity in schema discovery. Given a collection of documents with N types, the author merged these types into a new schema definition that closely approximates the ground truth schema. In order to analyze the JSON documents in a better way, NAMBA (2021) focussed on semantic information of metadata in terms of static and dynamic keys. The static keys are common attributes present in a collection, and dynamic keys represent that the structure of keys is not dynamically changing (not unique). The author has used a predefined word embedding model to predict the classes of keys. However, the impact of static and dynamic keys in schema discovery is not discussed. While other existing approaches focus on inferring the schema from JSON data format, Koupil et al. (2022) proposed MM-Infer, which has inferred schema from multi-models such as structured vs. semi-structured, order-preserving vs. order-ignoring, and so on. The schemas from each model are extracted and then merged locally using Apache Spark.

All the current works finally generate a canonical/global schema from a candidate set of schemas. Some research works took the union of attributes and generated a global schema. In contrast, some research works focus on reducing the type of an attribute and generating a reduced global schema. Even though extant literature focuses on generating a single schema for a collection, the global schema fails to provide the co-occurrence of attributes present in a specific document. This analysis is needed for applications such as data integration, query optimization, distributed query formulation, etc.

Application-based Comparison: Data Exploration, Query Formulation: Data exploration is a preliminary step for a data analyst to understand what data is stored and its characteristics, such as data completeness, relationships between data, and so on. In order to understand what data is stored, the primary step is to understand the implicit schemas of data which gives the structural description of a collection. To accomplish this, almost all the approaches have given a path for data exploration and query formulation in different ways.

2.2.1.2 Schema Variants Extraction

Global schema summarizes the structure of attributes present in a collection. However, for applications such as data integration, decomposing and optimizing queries in distributed databases, data analysis, etc., requires to know the different schemas present in a collection. We refer to this feature as *schema variants*. However, researchers have used different terms such as versioned schema (Sevilla Ruiz et al. 2015), personalized schema (Spoth et al. 2017), and so on. To define these terms uniquely in a taxonomy (ref. Figure 2.1), in this work, schema variants is referred to as more than one schema in a collection.

Sevilla Ruiz et al. (2015) discovered schema versions available in a collection. Nevertheless, the internal structure of arrays is not described. The different approach by Spoth et al. (2017) proposed *schema on query* using the probabilistic method. The generated schema is shown to the user and can be modified based on user feedback. Hence, the resulting output is a personalized schema. Gallinucci et al. (2018) find

the variants of the schema using association rules. Unlike previous methods, the hidden rules based on value and schema are captured to determine the schema variants in a collection. However, in the case of massive heterogeneous data where the values cannot be predicted, accessing the value from every document increases complexity. Gallinucci et al. (2019), an extended version of Gallinucci et al. (2018), proposed a decision tree-based Build Schema Profile (BSP) that captures the schema variants present in a collection. However, BSP does not look into the content of the arrays. Instead, it marks the presence of the array. Klettke et al. (2017) focussed on extracting the schema versions by capturing the structural changes over time. To reduce the ambiguity of schema versions, the schema evolution history is maintained by mapping the schema versions. Bawakid (2019) identified unique schemas by applying a clustering algorithm to the inferred attributes. The author has employed Term Frequency-Inverse Document Frequency (TF-IDF) vectors to identify similar attributes, with the goal of discovering unique schemas. Schema extraction plays a key role in OnLine Analytical Processing (OLAP) applications as well. Modern ideas for polyglot systems consist mostly of multi-stores and polystores, depending on whether they provide a single or several interfaces for cross-database management system querying. To support better query processing on multi-stores, schema heterogeneity must be taken into account to get consistent results. Forresi et al. (2021) extracted the exact schemas available in the collection for efficient retrieval of documents from multi-stores. D and Santhi Thilagam (2022) used a distributed formal concept analysis algorithm to identify the exact schema variants from large JSON document collections.

Application-based Comparison: Query Optimization, Distributed query decomposition, Data organization, Big Data Analytics: These applications rely on entities and their variants in a schema. The approach by Sevilla Ruiz et al. (2015) is most suitable for these applications because the approach discovered different entity versions of each entity in a collection. However, the approach has an assumption that entity information is embedded in the data source. As per Table 2.4, no such dataset provides such useful schema-related information in the JSON datasets. Therefore, it will be interesting, and more precise schemas can be extracted if entity information is embedded automatically

Table 2.3: Summary of existing research works in the field of schema variants extraction from JSON data

Research Article	Arrays	Schema-match	Clustering/ Classification	Data Representation	Approach	Output	Applications
Sevilla Ruiz et al. (2015)	✓	name, type	×	Meta-Model	MDE	Versioned Schema	Data Exploration, Query Formulation, Query Optimization, Distributed Query Decomposition, Data Organization, Big Data Analytics, Data Indexing
Spoth et al. (2017)	✓	name	×	Functional Dependencies Graph	Probabilistic approach	Personalized multi-schemas	
Klettke et al. (2017)	✓	name, type, entity type	×	Graph	Hierarchical Summariza-tion	Schema version graph	
Bawakid (2019)	✓	name, type	Clustering	Vector	Extract metadata and cluster them	Data Profile	
Gallinucci et al. (2018, 2019)	×	name, type	Classification	R-Schema	Extract R-Schema and build decision tree	Schema profile	
Frozza et al. (2021)	✓	name, type	×	Set of root-to-leaf attribute paths	Schema Integration for OLAP applications	Schema Variants	
D and Santhi Thilagam (2022)	✓	name, type	Clustering	Set of root-to-leaf attribute paths	Formal Concept Analysis	Schema Variants	

in the data source. In addition, Gallinucci et al. (2018, 2019) generated a schema profile by capturing the hidden rules that describe each schema. A schema profile reflects the probability for a schema to describe an instance of a class. D and Santhi Thilagam (2022) describes the exact schema variants available in a collection that captures the attribute co-occurrence in each schema variant. This information helps in the efficient data organization and indexing of large document collection.

2.2.1.3 Schema Extraction using Schema Mapping

The increasing popularity of JSON data formats has attracted attention in storing and processing them in relational databases, graph databases, OLAP applications, and so on. Mapping non-JSON schemas to JSON Schemas and vice versa allows conventional business applications and query optimization techniques to be applied to JSON datasets.

In general, the database transactions are validated in transactional relational systems based on the Atomicity, Coherence, Isolation, and Durability (ACID) properties of relational databases. With the advancement of NoSQL databases in almost all web applications, Irshad et al. (2019) proposed a hybrid solution for this issue by combining the properties of relational and NoSQL database systems. The JSON schemas are extracted, split, and stored in relational databases using the mapping algorithm. Aftab et al. (2020) presented and assessed an effective Extract, Transform, Load (ETL) method to migrate NoSQL to relational databases automatically that dynamically maps schema from NoSQL to relational schema. variety.js (2019) tool was used to extract the schema from the NoSQL database, and the ETL process was invoked to transform the schema that meets the needs of the relational databases.

While existing approaches used the primitive and complex data types of JSON documents for schema extraction, Frozza and Mello (2020) proposed JS4Geo (an extension of JSON Schema) that allows the definition of schemas for geographic data in well-known geographic data formats such as Keyhole Markup Language (KML), and Geography Markup Language (GML). The proposed work extracts the JS4Geo schema from each source and further generates a JS4Geo global schema using schema integration techniques. Different from existing works that focussed on schema extraction

of JSON data stored in web applications and NoSQL Document Stores, Frozza et al. (2020b) generates a JSON schema from NoSQL graph models. The input is a graph where the edges between the vertices represent the relationship of attributes. The vertices are extracted and grouped based on the labels. Based on the vertices and edges of each group, different JSON schemas are generated instead of a single schema.

Apart from NoSQL graph models, Frozza et al. (2020a) has explored the implicit structure of data in columnar databases as well. Columnar databases stores data in columns. The column data types are defined by the applications. Therefore, in order to infer the data types from each table, Frozza et al. (2020a) has proposed an HBase Schema Inference (HBaSI) process that analyses the hierarchical structure of the database. HBase stores data as byte arrays. HBase namespace is given as input for HBaSI, and JSON Schema is generated as output. The formal definitions and the canonical representation of the prototype developed by Frozza et al. (2020a) have been included in their extended version (Frozza et al. 2021).

Even though several works use some mapping techniques to generate JSON schema, they do not consider the important features of JSON documents, such as nesting depth and arrays. This is because data models other than document stores were designed for a specific purpose that may not meet the requirements of the JSON format.

2.2.2 Tools

Recent research efforts have focused on JSON schema inference in NoSQL systems. There are some tools and solutions available for schema extraction, such as mongodb-schema (mongodb schema 2019), schema.js (2019), variety.js (2019), Elastic Search (2019), Apache Drill (2019), and Couch Spark Connector (2019) to perform schema detection on NoSQL databases.

MongoDB-schema (mongodb schema 2019), a JavaScript library for JSON data format, infers a probabilistic global schema for the document collection, which is similar to the approach used by Klettke et al. (2015). Global schema is constructed by the union of all distinct schemas present in a collection. If the same attribute A appears in many schemas with different types, then the union of the data types is taken

to differentiate the structure of an attribute. This union type defines the *type* field of MongoDB-schema. *Count* attribute represents the number of documents containing the attribute A and *probability* represents the probability of the attribute A in the document. The schema in Elastic Search (2019) is a mapping technique that outlines the fields of JSON documents, their data type, and how they should be indexed in the underlying Lucene indexes. To be precise, Elasticsearch extracts the underlying structure of document collection in order to know the different attributes present in a collection. Static mapping involves the exact structure information of attributes, while dynamic mapping predicts the datatype of a new attribute if it is not present in an index. Variety (variety.js 2019), a schema generator for MongoDB, was implemented in a JavaScript framework called Node. Variety generates metadata about a JSON document collection that specifies the structure of attributes, its data types, and the number of occurrences.

The Apache Spark Inference mechanism (Apache Spark 2019) produces a precise global schema for a collection of documents. The schema shows the compact representation of a set of attributes with its associated type. However, while generating the global schema, Spark lacks the *nullable* property. The *nullable* flag for every attribute in a document is always set to *true*. Suppose two documents have the same set of attributes in an array differed in *null* type, irrespective of the different structure, Spark always shows as *string* type, which makes the structure as same in those documents. However, the structure is different due to the presence of the *null* type. Besides, Spark raises an error called a *corrupted record*. Hence, Spark fails to represent structural heterogeneity, especially in the case of arrays.

Couch Spark Connector (2019) proposed a schema inference module for query formulation for a user. Couchbase infers schemas from each document and identifies the distinct schemas present in a collection. The distinct schemas are stored in a hashtable and will be displayed to the user to formulate queries. If two schemas have the same set of attributes with the same data types, they are unique. Hence, schema inference in Couchbase is used for searching the documents. Profiling JSON data helps in optimizing the queries effectively with the help of metadata statistics like cardinalities. Apache Drill (2019) infers schemas dynamically and uses them during query planning and ex-

ecution. Möller et al. (2021) proposed JHound, which provides the empirical study JSON files. JHound explores the JSON document collections and understands the regularity, nesting depth, data types, and completeness of the data. JHound has been tested on real-life datasets and analyzed the metadata with schema evolution.

Even though several tools have been developed for the schema extraction of JSON documents, they collect the union of the attributes and derive a unique schema. Despite the advantages of the mentioned tools, they are generally used for validating the software.

2.2.3 Evaluation Strategy

This section explains how experiments with state-of-the-art approaches were conducted and how their findings were analyzed and verified.

2.2.3.1 Datasets

An abundance of JSON datasets is available for schema extraction. Table 2.4 describes the characteristics of JSON datasets used in recent research works. Several inferences can be drawn from the data in Table 2.4 as follows:

- The complex structures in JSON format are arrays and nesting depth. It is observed from Table 2.4 that all the datasets support attributes of primitive data types. While some datasets include an array of primitive data types, they lack in addressing the array of objects and array of array types. However, datasets supporting these two types are suitable for the approaches that measure schema heterogeneity. For instance, applications such as data integration and query optimization require knowing the schema variants available in a collection.

- Most datasets do not support data type heterogeneity of attributes. i.e., an attribute named *author* with *string* type in one document can be present as *array* in another document. Therefore, attribute heterogeneity plays a key role in measuring the efficiency of the schema extraction approaches. Baazizi et al. (2019) have determined the data type heterogeneity and generated a reduced schema. Reduced schema summarizes the attributes supporting different types.

- The most important feature of the schema extraction process in any data format is to capture the schema heterogeneity in nested objects. Approaches to identify the different schemas from datasets of nesting depth 2 give very good results. However, the unordered heterogenous property present in nested objects showcases the real complexity of the approach. Few schema extraction approaches (Baazizi et al. 2019; Chouder et al. 2019; Wang et al. 2015) performed an evaluation with datasets of more than two nesting depths and improved their performance.

- It is noted from Table 2.4 that few JSON datasets have flattened attributes, i.e., the attributes with nested objects and arrays are flattened by concatenating the parent and child attribute nodes of a JSON tree. The flattening process at the dataset level may help to identify the unique attributes in a JSON collection. However, the efficiency of the approach in handling nesting depth and arrays cannot be measured. Therefore, flattening the nesting objects and arrays at the data parsing level illustrates the complexity of the approach in handling all the characteristics of the JSON dataset.

- Although there is a large number of schema extraction approaches available in literature, very few research works have used the same datasets for evaluation. For instance, Klettke et al. (2015) and Frozza et al. (2018) have used datasets from Movie, Company, and Drugs scenarios and compared the number of schemas discovered by each approach. Similarly, Baazizi et al. (2019) and Chouder et al. (2019) have used GitHub, Twitter, Wikidata, and NYTimes datasets. However, the results of these approaches were not compared to evaluate them due to their different output. Consequently, the results after comparing the approaches for the same datasets are meaningful and complete.

2.2.3.2 Evaluation Measures

Multiple measures have been adopted to evaluate the schema extraction approaches. As JSON documents are application-specific, existing approaches use various evaluation measures to determine the performance of their approaches. Table 2.5 shows the summary of evaluation measures used in the literature, which eventually helps the readers

Table 2.4: Characteristics of datasets used in existing research works

Dataset Name	Datatypes Supported			#Documents	#Nesting Depth	Scenario	Approaches used
	Primitive Datatypes	Arrays of Objects or Arrays	Attribute Hetero-geneity				
Wendelstein 7-X project (Spring et al. 2012)	✓	×	×	30	-	Record configuration of plasma experiments	(Klettke et al. 2015)
Pagebeat project (Finger et al. 2014)	✓	×	×	800	-	Statistical data of Web Performance Measures	(Klettke et al. 2015)
lights (DiScala and Abadi 2016)	✓	Flattened	×	13,000,000	Flattened	Flight information collected from US Bureau of Transportation Statistics	(DiScala and Abadi 2016)
GitHub (DiScala and Abadi 2016)	✓	×	×	1,000,001	5	GitHub Web Service	(Baazizi et al. 2017, 2019; DiScala and Abadi 2016)
Twitter'16 (DiScala and Abadi 2016)	✓	✓	✓	9,901,087	4	*tweet* records describing metadata about tweets	(Baazizi et al. 2017, 2019; Chouder et al. 2019; DiScala and Abadi 2016)
NYTimes [2]	✓	✓	✓	1,184,943	6	metadata about the NYTimes articles	(Baazizi et al. 2017, 2019)
Wikidata [3]	✓	✓	×	49,013,568	8	Wikidata 16/06/2018 snapshot	(Baazizi et al. 2017, 2019)
Twitter'18 [4]	✓	✓	✓	1,945,365	6	tweet records collected during the 2018 Russian election campaign	(Baazizi et al. 2019)

(continue on the next page)

[2]https://developer.nytimes.com

[3]https://dumps.wikimedia.org/wikidatawiki/entities/.

[4]https://www.kaggle.com/borisch/russian-election-2018-twitter.

Table 2.4: Characteristics of datasets used in existing research works (cont.)

Dataset Name	Datatypes Supported			#Documents	#Nesting Depth	Scenario	Approaches used
	Primitive Datatypes	Arrays of Objects or Arrays	Attribute Heterogeneity				
VK [5]	✓	✓	✓	3,036,654	5	User interaction in VK social networking site about 2018 Russian election	(Baazizi et al. 2019)
Core [6]	✓	✓	×	123,986,577	4	Research articles aggregated from many open repositories	(Baazizi et al. 2019)
Freebase (Hassanzadeh et al. 2013)	✓	✓	✓	3,888	4	Drug	(Wang et al. 2015)
DBPedia (Hassanzadeh et al. 2013)	✓	✓	×	3,662	2	Drug	(Frozza et al. 2018; Wang et al. 2015)
DrugBank (Hassanzadeh et al. 2013)	✓	✓	×	4,774	3	Drug	(Wang et al. 2015)
Freebase (Hassanzadeh et al. 2013)	✓	×	×	84,530	2	Movie	(Wang et al. 2015)
DBPedia (Hassanzadeh et al. 2013)	✓	×	×	30,332	2	Movie	(Frozza et al. 2018; Wang et al. 2015)
IMDb (Hassanzadeh et al. 2013)	✓	✓	✓	7,435	3	Movie	(Wang et al. 2015)
Freebase (Hassanzadeh et al. 2013)	✓	✓	✓	74,970	6	Company	(Wang et al. 2015)

(continue on the next page)

[5] https://vk.com/dev/streaming_api_docs_2.

[6] https://core.ac.uk/services#dump-structure.

Table 2.4: Characteristics of datasets used in existing research works (cont.)

Dataset Name	Datatypes Supported			#Documents	#Nesting Depth	Scenario	Approaches used
	Primitive Datatypes	Arrays of Objects or Arrays	Attribute Hetero-geneity				
DBPedia (Hassanzadeh et al. 2013)	✓	×	×	24,367	2	Company	(Frozza et al. 2018; Wang et al. 2015)
SEC (Hassanzadeh et al. 2013)	✓	×	×	1,981	5	Company	(Wang et al. 2015)
RD1 (Gallinucci et al. 2019)	✓	×	×	50,00,000	5	fitness	(Gallinucci et al. 2019)
RD2 (Gallinucci et al. 2019)	✓	×	×	767,000	5	fitness	(Gallinucci et al. 2019)
RD3 (Gallinucci et al. 2019)	✓	×	×	473,000	2	Error log from software companies	(Gallinucci et al. 2019)
Games (Valeri 2014)	✓	Flattened	×	32,000	4	Sports Reference	(Chouder et al. 2019)
DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017)	✓	×	✓	2,000,000	2	Publication	(Chouder et al. 2019)
TwitterW and TwitterM (Spoth et al. 2017)	✓	Flattened	×	100,000	NA	Data extracted from Twitter's Firehose	(Spoth et al. 2017)

to decide on the performance evaluation methods in different aspects. Table 2.5 offers various insights as follows:

- Extant approaches have calculated the execution time of the schema extraction process, which is the basic metric in this field. However, many approaches have not compared their execution time with other approaches.
- Although most approaches have evaluated their approaches with real datasets, it is observed from Table 2.5 that very few approaches have compared their results with other existing approaches in the literature. Most works compared the performance of their approaches with the baseline approach.
- Many approaches have extracted all the schemas from a collection and then constructed a single global schema. Comparing the intermediate results, i.e., the number of schemas with existing approaches, would determine the efficiency of the parsing methods used. However, these intermediate results have not been evaluated by most of the works cited in the literature.
- Although extant approaches designed schema extraction algorithms to support array data type, few approaches have evaluated their algorithm using datasets that do not contain either an array of objects or an array of arrays.

2.2.4 Challenges and Research Directions

In this thesis, we have examined many approaches and tools for schema extraction from JSON data. Despite the wide variety of methods available in the literature, this field has significant scope for future research. This section discusses the open issues and research challenges in the field of JSON schema extraction.

Unlike the conventional schema extraction approaches on XML or RDF data where the complex structures are less, JSON data is highly dependent on two complex structures, such as objects and arrays. Moreover, the presence of arrays introduces high structural heterogeneity in a collection. This makes the schema extraction approaches of JSON data differ from other semi-structured data formats.

Table 2.5: Summary of evaluation measures used in existing research works

Research Article	Output Format	Type of Dataset	Dataset with Arrays ?	Experimental Measure	Comparison with alternative solutions
Cánovas Izquierdo and Cabot (2013)	Class	-	-	-	-
Sevilla Ruiz et al. (2015)	Versioned Schemas	-	-	-	-
Wang et al. (2015)	-	Real	✓	Effectiveness	Baseline
Klettke et al. (2015)	JSON	Real	×	Execution time, Schema & Attribute Existence	Baseline
Gallinucci et al. (2018)	Tree	Real & Synthetic	×	Precision, Conciseness, and Expressiveness	Baseline
Baazizi et al. (2017)	JSON	Real	✓	Effectiveness, Execution Time, Scalability	-
Gallinucci et al. (2019)	Tree	Real & Synthetic	×	Effectiveness & Efficiency	Baseline
Baazizi et al. (2019)	JSON	Real	✓	Efficiency, Succinctness, Precision	Baseline
Chouder et al. (2019)	md-schema	Real	×	Querying	×
Frozza et al. (2018)	ROrd (Schemas)	Real	×	Number of Schemas, Execution Time	Wang et al. (2015)
D and Santhi Thilagam (2022)	Schema Variants	Real	✓	Number of Schemas, Execution Time	Gallinucci et al. (2019)

Similar to the traditional schema extraction problem, schema extraction from JSON documents is also highly application-specific. The majority of the approaches were developed by keeping a set of requirements and constraints in mind. Hence, a quantitative comparison among the approaches is difficult in practice as they generate schemas in different forms. Furthermore, considering the different approaches presented, decid-

ing a particular algorithm for schema extraction is challenging because the algorithm depends on the type of data model being examined and the type of schema (global or schema variants or relational schema) expected.

The JSON schema extraction encounters the following research challenges and research gaps specific to JSON data:

- **Computational Complexity:** In this big data era, the dynamic nature and massive size of data generated pose challenges for the schema extraction approach to extract precise and concise schemas. In particular, to capture the array structure, the extraction algorithms have to parse the complete nesting depth of arrays (array of objects and array of arrays), causing the computational complexity of the extraction algorithms to be high. Consequently, it is challenging to develop an efficient and scalable method for schema extraction from massive JSON datasets. Few existing works (Baazizi et al. 2019) have addressed this issue using Apache Spark. However, they focus on generating a single schema for a collection. Identifying the schema variants supports various data management tasks effectively. Developing a computationally efficient method for schema extraction is a promising research direction in this area.

- **Dynamic Schema Extraction:** The dynamically generated JSON documents can be stored in two ways: (i) inserting new documents with new schema variants, which can also be referred to as schema versions, usually maintained in a data lake, and (ii) modifying the old schema variants by updating the documents already stored in the collection. Even though there are existing works in supporting schema evolution updates (Klettke et al. 2017, 2016; Scherzinger et al. 2013), we observe that the research work related to the dynamic schema extraction method for modifying the existing schema variants is still in its infancy. Data management tasks such as data indexing, data organization, and so on require the latest documents to be updated to support relevant data retrieval. Therefore, apart from maintaining the schema evolution, the schema extraction methods should support dynamic operations as well.

- **Benchmarking JSON Datasets:** It is observed from Section 2.2.3 that although most approaches evaluated their work with real datasets, almost all approaches use different datasets, which could be the major reason for the lack of comparison with other alternative solutions. Although few datasets exist for the schema extraction process, it is required to build benchmark datasets that support all the characteristics of JSON data which will eventually help in identifying the features covered by the existing approaches.

- **Performance Evaluation:** Due to the dearth of benchmark datasets containing explicit ground truth values, it is generally difficult to evaluate schema extraction approaches. In many instances, researchers are required to conduct their experimental study on synthetic data by deciding the number of schemas prior or have to investigate the different schemas using domain expertise manually. Furthermore, there is no recognized standard to evaluate schema extraction approaches developed for JSON data.

- **Semantic Schema Extraction:** JSON documents not only have different schemas based on structural similarity but also differ semantically, i.e., two schemas with the same contextual or semantic meaning use different attributes. Semantic schema extraction helps in grouping the semantically equivalent schemas that return relevant results for user queries efficiently. Literature shows that the structural and semantic similarity of JSON schemas is still in its early stage.

- **Efficient Data Indexing:** With the rapid emergence of JSON data generated and stored in NoSQL document stores, it becomes vitally important to use indexes for efficient data access. Existing JSON indexing techniques use full-text search or indexing JSON attributes by extracting schema. Full-text search incurs long query latency (Shang et al. 2021) and indexing JSON attributes suffer from huge index sizes by storing all distinct paths of JSON document collection in an index. Therefore, identifying the primary attributes for indexing the documents that causes for large index size is a promising research area.

- **Supporting Semantic Search:** JSON documents are not only structurally hetero-

geneous but also semantically heterogeneous, which further demands the need for using the semantic properties of heterogeneous JSON schemas to support semantic search. While traditional semantic models capture the linguistic knowledge of data, the similarity score of queries and the terms in the documents is based on the abstract meaning of attributes which does not capture the context of data. Dense Retrieval (Guo et al. 2022) is a promising alternative approach that matches query terms and schema variants in a low-dimensional embedding space and is as efficient in finding the contextual meaning of data. While existing work focussed on dense retrieval of unstructured data (Uma Priya and Santhi Thilagam 2020), finding the contextual similarity in hierarchical data like JSON is still in the preliminary stage.

- **Schema Inference and Advanced learning techniques:** Majority of the literature used traditional techniques to extract and analyze schemas. Gallinucci et al. (2018, 2019) generated schema profile using the decision tree. Although few approaches (Baazizi et al. 2019) have used distributed processing frameworks to extract schemas from a large collection, combining the distributed techniques with advanced learning methods such as machine learning and deep learning enhances the performance of the approaches in terms of execution time. In addition, these techniques provide a way to analyze the large collection in an efficient way.

2.3 XML AND JSON INDEXING TECHNIQUES

The indexing techniques of XML can be applied to JSON with some enhancements suitable for the nature of JSON data. Hence, this section discusses the related work of XML and JSON data indexing techniques, as seen in Table 2.6, with strong attention to structure-based and structure and content-based queries, respectively.

2.3.1 Syntactic Search

The path-based index is constructed by summarizing the paths of XML data. The early effort in designing a path-based index is a data guide. Data guides describe the structural summary of the XML/JSON data by integrating the forest of tree-shaped schemas. Although it was originally developed for XML (Goldman and Widom 1997), similar

approaches have been applied to JSON data (Klettke et al. 2015; Liu and Gawlick 2015; Liu et al. 2016) as well. Qadah (2017) designed index structures for processing containment queries on interlinked XML documents. Dhanalekshmi and Asawa (2018) reduce the number of index entries by combining the terminal siblings at the same level into a single path. Hsu and Liao (2020) designed UCIX, an updatable compact index structure for XML documents using a branch map labeling scheme. UCIX adopts the structural summary method to provide information about the XML documents. The XML nodes are encoded based on the presence of XML elements in a document. However, the size of each index node structure is large, which increases the complexity of a large dataset. Wellenzohn et al. (2020) proposed a dynamic interleaving scheme that merges the path and values of an attribute in an index. Hence, the approach has achieved robust performance for content and structure queries. However, the substructure retrieval or recursive queries can not be performed due to the distribution of path and value bytes in an interleaved key.

In case of JSON, relational databases such as Oracle (Liu et al. 2014) and SAP extend an RDBMS to support JSON data format. Budiu et al. (2014) has designed a unified query language and indexing data structure for answering JPath queries. The author has modeled JSON documents as unordered labeled trees and handled the arrays of objects by splitting them into different objects and numbering them in a sequence. The set of the same structured JSON objects in an array is grouped to reduce the index size. However, the array still suffers from redundant attributes if there is a mismatch in comparing the structure of JSON objects. Shukla et al. (2015) designed the CAS index for Microsoft Azure's DocumentDB. The path and value are concatenated, and the result is stored in Bw-tree, which results in high path redundancy in index keys. These problems are addressed by Shang et al. (2021), which is an enhancement of Shukla et al. (2015), by storing the keys, values, and long strings in different indexes. Their index pruning policy, on the other hand, retains index keys based on user preference, which creates complexity in handling queries that are not present in the index. Wahyudi et al. (2019) used the vector space model to find similar JSON documents. However, they have focussed on the contents of a small JSON collection rather than the structure of JSON

Table 2.6: A comparison of existing research works on XML and JSON Indexing

Research Article	Data Format	Index Type	Advantages	Limitations
Budiu et al. (2014)	JSON	Structure and content-based Index	Summarized index tree is constructed for merging the similar nodes	Threshold determines the merging phase and substructure redundancy in arrays increases index size
Shukla et al. (2015)	JSON	Structure and content-based Index	Enables automatic indexing of documents without requiring a schema	Huge memory due to concatenation of path and value
Agarwal et al. (2016)	XML	Semantic Index	It returns meaningful information from any XML node, which contains a subset of keywords in the search query Q	Increased Index size
Dhanalekshmi and Asawa (2018)	XML	Structure-based Index	Reduced Index entries by combining path	All XML nodes are indexed
Tekli et al. (2019)	Semi-structured	Semantic Index	Incorporate semantics at indexing level	Context is not customized for different datasets
Hsu and Liao (2020)	XML	Structure-based Index	Reduced Index Size with new labelling scheme	All XML nodes are indexed
Wellenzohn et al. (2020)	Hierarchical Data	Structure and content-based Index	Proposed a well-balanced integration of paths and values in a single index	Pre-processing time before index construction is high
Shang et al. (2021)	JSON	Structure and content-based Index	Achieved less memory by storing the key, values, and long strings in different indexes	The user preferred queries are stored in indexes which introduces complexity in handling queries that are not present in the index

data. Jiang et al. (2020) built bitwise structural indices to provide a memory-efficient index for vast JSON documents. Subramaniam et al. (2019) proposed D-DGReLab$^+$ for efficient processing of user queries in a distributed environment. D-DGReLab$^+$ also includes the pruning technique that removes the irrelevant nodes.

In summary, the space requirement of the content index in XML/JSON data is directly proportional to the number of nodes in the index tree or the number of unique paths or attributes in a collection. This significantly improves the precision of similarity search as well as the amount of storage space required for the index. Considering the schema-less model of JSON documents, the collection may have different schemas, resulting in large number of attributes which results in space overhead at the index. Therefore, there is a requirement to reduce the space overhead of indices for large datasets.

2.3.2 Semantic Search

While database systems have emphasized the integration of syntactic keyword-based search functionality, the information retrieval community has made few efforts to extend syntactic processing toward the semantic full-text search of semi-structured data through semantic indexing techniques. These techniques attempt to address the semantic relatedness problem by encoding both queries and documents into semantic representations using an external knowledge base and performing a retrieval in the semantic space (Kumar et al. 2012; Tekli et al. 2019). However, when the user query is not present in the index, then these techniques employ query pre-processing techniques such as query disambiguation (Navigli 2009; Tekli 2016), and query post-processing techniques such as query relaxation and refinement (Allan et al. 2012; Carpineto and Romano 2012) to find the indexed terms that match the query. This process uses WordNet (Miller 1998) to find the possible matches.

Alghamdi et al. (2014) built schema and data index to handle structure-based queries and the value index to handle content-based queries. However, it ends up in a large index size as each attribute in the XML dataset acts as an index key. Agarwal et al. (2016) presented Generic Keyword Search (GKS) for XML documents wherein the

meaningful information is retrieved not only from Lowest Common Ancestor (LCA) nodes but also from nodes containing a subset of keywords in the search query. Tekli et al. (2019) designed a semantic aware indexing system to provide a generic keyword query model for structured, semi-structured, and unstructured documents. While few research studies focus on semantic search in a hierarchical structure like XML, the existing approaches use the knowledge resources such as WordNet (Miller 1998). In case of semi-structured data where the attributes preserve the ancestral relationship, capturing the context rather than lexical information gives efficient results for all kinds of datasets.

Given that our research explores the dense retrieval of JSON data, it is necessary to discuss related work that has attempted to use neural embeddings to retrieve data efficiently. In the embedding-based retrieval model, queries and documents are trained in the same or different embedding space (Lashkari et al. 2019; Liu et al. 2021a,b; Zhan et al. 2021, 2020) and compare their inner product. Since the document embeddings are precomputed and indexed, the search operation is faster using approximate nearest neighbor search algorithms. Lashkari et al. (2019) jointly train the terms, semantic type, entities, and documents which improves the retrieval efficiency and effectiveness. The posting list is constructed based on the similarity of term vectors rather than term occurrences. Huang et al. (2020) introduced a unified embedding framework to model semantic embeddings for facebook search using the inverted index. The author focussed on improving the recall efficiency to satisfy the user's search instead of giving exact results. Zhan et al. (2020) presented Learning To Retrieve (LTRe), which uses the pretrained encoder for document embeddings and constructs the index beforehand. The query embeddings are generated and updated at each training iteration. By updating the parameters, the model retrieves better relevance than rerank. Tonellotto and Macdonald (2021) improves the performance of dense data retrieval by pruning the non-relevant query embeddings. This approach enhances data retrieval relevance by reducing the number of irrelevant documents retrieved.

While there is a lot of research on embedding-based retrieval of unstructured data, this field is still in its infancy for hierarchical data formats such as XML and JSON.

It is observed from the literature that extant research works focus on either the lexical or semantic matching of XML/JSON documents. Most existing works on JSON data need further improvement in reducing the redundancy incurred by arrays during index construction, which will significantly impact data retrieval time. Considering the nature of JSON data in real time, there is a need for an index structure that supports both lexical and semantic matching of documents for better information retrieval. The index structure must be compact without losing any information and provide better response time.

2.4 SUMMARY

In this chapter, a structured review of the various approaches used in schema extraction of XML and JSON data is presented. Exploring the structural information from JSON documents is a challenging task due to the heterogeneity in schemas. Over the last decade, a wide variety of approaches developed for schema extraction with different goals and outcomes. This chapter organized the state-of-the-art approaches into different categories based on the nature of the output. Finally, the various research challenges and directions in this field are discussed. When choosing an appropriate approach, different aspects of the application must be considered, such as the type of schema required and the types of the dataset that support different features of JSON format, and so on. This comprehensive review provides a better understanding of the several schema extraction approaches and their applications.

This chapter also provides a review of the different indexing methods available for XML/JSON document retrieval. From the literature, it has been found that the existing index structures suffer from huge index sizes and, thereby, data retrieval time. In addition, the support for the semantic search of JSON documents is still in the preliminary stage. Therefore, there is a requirement for designing compact structures for JSON document collection, which supports both exact and semantic matching of user queries.

CHAPTER 3

PROBLEM DESCRIPTION

JSON is the primary format for storing multi-structured documents in NoSQL document stores. Due to the lack of a strict schema enforced on documents, the JSON documents in the collection can have different schemas. The absence of schema information increases the complexity of accessing and managing heterogeneous documents. Therefore, knowledge of the implicit schemas is essential to understand the data stored in the collection. This schema information can be helpful for efficient data retrieval, data integration, query formulation, etc. Most literature on schema extraction of JSON documents focuses on generating global schema, which does not capture the different sets of attributes present in a collection. Therefore, there is a need to extract schema variants that captures the co-occurrence of attributes and promote the above-mentioned tasks effectively. In addition, JSON documents are not only varied by structural heterogeneity but also by semantic heterogeneity. Therefore, the complexity of handling structural and semantic heterogeneity of large data volume is rising. To address this research gap, this work clusters the JSON documents based on the contextual similarity of JSON schemas. The schema variants are extracted at each cluster which captures the common and schema-specific attributes. In order to efficiently manage and access the large collection of JSON documents, it becomes important to use efficient index structures. This work proposes a compact indexing scheme that supports both lexical and semantic matching of JSON path-based queries efficiently. In addition, data is growing with the continuous addition, modification, or deletion of attributes or documents in

the existing collection. Therefore, there is a need to support dynamic data retrieval by updating the schema variants and indexes efficiently.

3.1 OBJECTIVES

The work is subdivided into the following objectives:

1. *Designing an embedding-based clustering approach to cluster the contextually related JSON documents and proposing a tree structure to store the schema variants:* This objective aims to identify schema variants by partitioning the contextually similar JSON documents into clusters and then provide the summarized schema variants representation in the form of *SVTree* by considering all the features of JSON format such as primitive data types, arrays, and nesting objects. The problem is formally stated as: Consider a JSON document collection $G = \{D_i\}_{i=1}^{n}$ and $D_i = \{A_{j,s}\}_{s=1}^{t_j}$ where $A_{j,s}$ represents the attribute A_s in document j, t_j represents the number of attributes in a document j and n denotes the size of the collection. The proposed work aims to assign G into K clusters $\{C_1, C_2, ..., C_K\}$ where C_i contains equivalent JSON documents based on context. On each cluster, the schema variants $S = \{S_i\}_{i=1}^{n}$ are identified and stored in a new data structure called *SVTree*. The structural similarity of JSON schemas in each cluster provides the exact schema variants.

2. *Proposing a compact indexing scheme that exploits the lexical and contextual relationship of queries with JSON documents for efficient data retrieval:* The second objective of this work is to design a compact schema-aware indexing scheme that supports both lexical and semantic matching of JSON path-based queries over JSON collections. The extracted schema variants allow us to analyze the co-occurrence of attributes and identify the common and schema-specific attributes, which helps in constructing compact index structures for efficient data retrieval. The problem is formally stated as: Consider a JSON document collection $G = \{D_i\}_{i=1}^{n}$ and $D_i = \{A_{j,s}\}_{s=1}^{t_j}$ where $A_{j,s}$ represents the attribute A_s in document j, t_j represents the number of attributes in a document j and n denotes the size of the collection, *SVTree* provides the common and schema-specific

attributes at each cluster C_i. The proposed work aims to build space-efficient indexes $I \in \{$*JIndex, EJIndex*$\}$ for efficient retrieval of JSON data, where *JIndex* denotes lexical match and *EJIndex* denotes semantic match. It also aims to improve the quality of search relevance and query processing time.

3. *Extending the approach to update the schema variants and indexes for dynamic data retrieval:* The third objective of this work is to update the schema variants and index structures to provide the latest results to user queries. Consider an updated JSON document collection $G' = \{D'_i\}_{i=1}^{n}$ and $D'_i = \{A'_{j,s}\}_{s=1}^{t_j}$ where $A'_{j,s}$ represents the attribute A_s in document j, t_j represents the number of attributes in a document j and n denotes the size of the collection, the proposed work aims to develop an *Incremental embedding-based clustering* approach that extracts different JSON schemas in the collection G' and analyses the contextual similarity among them to group the similar JSON documents based on the old K clusters ie., the contextual similarity of G' is compared with the K clusters. The output is an updated *SVTree* that represents the summarization of updated schema variants available at each cluster C_i along with its core and schema-specific attributes. In addition, *JIndex* and *EJIndex* are updated incrementally to support the retrieval of the latest documents for user queries.

CHAPTER 4

SCHEMA VARIANTS EXTRACTION

4.1 INTRODUCTION

The problem of finding the schema variants is related to the problem of finding the distinct schemas available in a collection. The challenges involved in identifying the schema variants from heterogeneous data collection are (i) JSON stores its data in map format (key-value pairs), and its specific constructs are nesting objects and arrays. (ii) The complexity of handling structural and semantic heterogeneity of large data volume is rising. The former issue is that the unordered nature of JSON data introduces structural heterogeneity in the collection. Although arrays are ordered lists of values, JSON arrays have no restrictions on their type of values (Bourhis et al. 2017). The lack of data type restriction creates numerous structural variations in arrays, making it difficult to find exact schema variants. In the latter issue, JSON documents are not only structurally heterogeneous but also semantically heterogeneous. In addition, the sheer size of data collection comprises a large set of attributes with different types. However, a schema variant includes a subset of attributes. Hence, the analysis of the interesting distribution of attributes from a large set takes a huge computation time to identify the exact schema variants.

Currently, the literature on schema extraction of JSON data focuses on different goals such as global schema (Cánovas Izquierdo and Cabot 2013; Chouder et al. 2017; DiScala and Abadi 2016; Klettke et al. 2015), reduced schema (Baazizi et al. 2017) and versioned schema (Kellou-Menouer and Kedad 2017; Sevilla Ruiz et al. 2015).

The problem of schema extraction is also focused on finding the core and schema-specific attributes available in a collection (Bawakid 2019). All the above-mentioned solutions on schema extraction facilitate schema integration, query processing, and so on. However, these approaches are based on naive structural features of JSON data. The performance of information retrieval can be further improved by including the semantics of attributes. Furthermore, as the number of heterogeneous JSON sources grows, it is very challenging to manage them in an organized manner to support the efficient retrieval of data.

Clustering provides a way to organize and summarize the large data collection by grouping similar documents in one cluster. Many approaches have been developed for clustering semi-structured data in XML (Aïtelhadj et al. 2012; Costa and Ortale 2013, 2017; Piernik et al. 2016, 2015) format. Despite being the most popular data representation and interchange format, the research on JSON document clustering is very sparse due to its complex hierarchical structure. Most of the existing works use naive structural features for finding similar structured JSON documents. However, JSON documents not only have varied structures but also differ semantically, i.e., two schemas with the same contextual meaning use different attribute names. For instance, consider a *Publication* scenario in the form of JSON document collection from various publishers such as DBLP, IEEE, ACM, and so on. Each publisher uses a different set of attributes to represent the classes, such as *conference, journal, book*, etc. For example, schemas *a* and *c* in Table 4.1 represent the class *conference*; however, they have varied structures but are related by context.

The following example illustrates the need for JSON document clustering based on contextual similarity. Let us consider the simple JSON schemas with different attributes as illustrated in Table 4.1. Schemas *a* and *c* are associated with *conference*, while *b* belongs to *news article*. The task is to find a similar schema for Table 4.1 *(a)*. The schemas *a* and *b* share the root attribute *Paper* whereas *a* and *c* share a different root. Despite the presence of *Paper*, schemas *a* and *b* are semantically identical, whereas *a* and *c* are similar by context, i.e., *conference*. Although the attributes *HeadLine* and *Title* give the same meaning, the context is different. Also, the same set of attributes

present in *a* are annotated differently in *c*, and hence they belong to the same cluster irrespective of their structures. In this case, the contextual similarity is identified by the surrounding attributes in a schema. Therefore, context-based clustering of JSON documents facilitates the efficient organization of data, which would further enhance the performance of query processors by reducing the search space. For example, a query is to *retrieve the author names who published papers in conference proceedings in 2020* then grouping the documents that share a similar context *conference* improves the performance of information retrieval. This work does not use JSON values (contents) to determine similarity; instead, it concentrates on clustering JSON documents based on the contextual similarity of JSON schemas.

Table 4.1: A sample collection of JSON documents and their schema representation

S.No	JSON Documents	Schema

a	”Paper”:
	{
	”Title”: ”NDC-Study”,
	”Authors”: ”Smith”,
	”Conference Name”: ”ICoSiam”,
	”Year”:2004
	}
		Paper_Title
		Paper_Authors
		Paper_Conference Name
		Paper_Year

b	”Paper”:
	{
	”Headline”: ”A model for positive change”,
	”Reporter”: ”Sebastian”,
	”Company”: ”ACD”,
	”Date”:2007
	}
		Paper_Headline
		Paper_Reporter
		Paper_Company
		Paper_Date

c	”Article”:
	{
	”Authors”:
	{
	”Author”:”John Meyer”
	}
	”Conference”:
	{
	”Name”:”CoPD”
	”Year”:1997
	}
	}
		Article_Authors_Author
		Article_Conference_Name
		Article_Conference_Year

The performance of document clustering is determined by the quality of relevant feature extraction and efficient document representation. The most frequently used tree representation of JSON often involves information loss and complex computations. Reducing complex computations consists of reducing the number of relationships between nodes in a JSON tree, which results in information loss. Vector representations of semi-structured data have been increasingly used to represent a range of features, such as paths, subtrees, attributes, and queries (Piernik et al. 2015). In traditional vector representations, each word is assigned its own dimension, resulting in a high-dimensional sparse representation. In addition, when the data is not appropriately distributed in the feature space due to high intra-variance, conventional clustering algorithms struggle to produce satisfying performance (Song et al. 2014). Because they employ a linear mapping function to transform the original data into a feature vector which causes the relevant data to be scattered around the feature space. Existing research (Xie et al. 2016; Yang et al. 2017) represents documents as TF-IDF vectors and then obtains non-linear mapping via autoencoders. Since TF-IDF vectors are based on word frequency, they generate inefficient word representations in comparison to distributed word representations (word embeddings) (Park et al. 2019). Word embeddings, which are often trained using neural networks, represent words in a low-dimensional vector space and capture their syntactic and semantic relationships (Mikolov et al. 2013). Hence, these models are more effective than TF-IDF vectors at capturing semantically relevant embeddings.

More recently, the distributed representation for embeddings is modeled with pre-trained language models (Park et al. 2019; Peters et al. 2018), which generate rich contextualized embeddings than conventional word embeddings. While the efficiency of clustering is determined by the quality of feature representations (Zhou et al. 2020), incorporating contextualized embeddings naturally enhances the performance. However, the existing works on the word or document embeddings focussed on the sequence of words or sentences in unstructured data (Park et al. 2019; Uma Priya and Santhi Thilagam 2020). Hence, the benefit of contextualized embeddings on semi-structured data is not yet explored in the literature. Therefore, this work is the first attempt to construct context-based embeddings in a hierarchical structure, i.e., JSON documents.

To address the aforementioned challenges, in this work, we designed an *embedding-based clustering* approach for grouping the JSON documents based on the contextual similarity of JSON schemas. The schema variants at each cluster are summarized in a data structure.

Our major contributions in this chapter are as follows:

1. Studying the problem of clustering JSON documents and extracting schema variants from JSON documents.

2. Proposing a *SchemaEmbed* model to learn the syntactic and semantic relationship of attributes from JSON documents by considering their ordered and unordered properties.

3. Devising an embedding-based approach for clustering JSON documents based on contextual similarity of JSON schemas

4. Proposing *SVTree* to store schema variants and exhibit schema variants summarization

5. Evaluating the proposed approach on both real and synthetic datasets

The rest of this chapter is organized as follows. Section 4.2 describes the problem statement. Sections 4.3 and 4.4 present the proposed approach in detail. Section 4.5 presents the experimental study and performance analysis. Section 4.6 presents a summary of the chapter.

4.2 PROBLEM DESCRIPTION

This section presents the design of our approach to identify schema variants from a JSON collection. Consider a JSON document collection $G = \{D_i\}_{i=1}^{n}$ and $D_i = \{A_{j,s}\}_{s=1}^{t_j}$ where $A_{j,s}$ represents the attribute A_s in document j, t_j represents the number of attributes in document j and n denotes the size of collection. The proposed work aims to develop an embedding-based approach that extracts different JSON schemas in the collection G and analyses the contextual similarity among them to group the

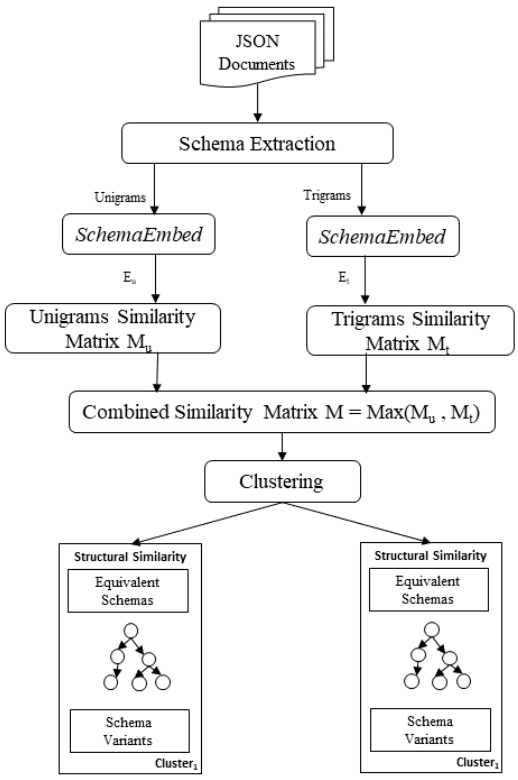


Figure 4.1: Flow diagram of schema variants extraction

contextually similar JSON documents at K clusters $\{C_1, C_2, ..., C_K\}$. The output is a tree-based data structure that represents the summarization of schema variants available at each cluster C_i along with its core and schema-specific attributes. Figure 4.1 shows the flow description of the proposed schema variants extraction approach. Table 4.2 illustrates the set of all the symbols used in this chapter.

The proposed work is divided into two phases, such as (i) Embedding-based clustering approach and (ii) Identification of schema variants. The description of each phase is explained in the following sections.

4.3 EMBEDDING-BASED CLUSTERING APPROACH

The objective of this phase is to design a model for capturing the contextual similarity of JSON schemas and group the contextually similar documents. Given a collection of JSON documents represented as $G = \{D_1, D_2, ..., D_n\}$, and its N-gram attributes N, *SchemaEmbed* model attempts to learn the contextual relationship of JSON schemas

by constructing the unigrams and trigrams vocabularies from the set of flattened JSON attributes. The JSON documents are clustering based on schema embeddings generated by the *SchemaEmbed* model. Four major steps such as JSON schema extraction, feature extraction, constructing *SchemaEmbed* model to generate schema embeddings, are involved in generating the contextualized vectors for a whole JSON collection. Finally, the JSON documents are clustered using an *embedding-based clustering* approach.

Table 4.2: Symbols

Symbols	Definitions
G,n	JSON Document Collection and its size
D	JSON Document
S	JSON Schemas
A	JSON Attributes
U	Unigram Attributes Set
T	Trigrams Attributes Set
F	Embedding Matrix
W, d	Vocabulary and dimension size of Word2Vec Model
P	Probability function used in Word2Vec model
v_a, v_a'	Input and Output vector for an attribute *a*
X, Y, Z	Input, Output(Encoded) and Decoded vector of deep autoencoder
H_1, H_2, H_3	Hidden layer vectors of deep autoencoder
M_u, M_t	Unigram and Trigram similarity matrix
E_u, E_t	Unigram and Trigram schema embeddings
M	Combined similarity matrix to be given to clustering algorithm
N	N-grams Vocabularies
C	Clusters
B	Similarity Graph obtained in spectral clustering algorithm
V, E	Vertices and Edges of B
H	Degree Matrix in spectral clustering algorithm
L	Laplacian Matrix in spectral clustering algorithm
I	Eigen Vectors in spectral clustering algorithm

4.3.1 JSON Schema Extraction

The JSON document can also be represented as a tree where the nodes (vertices) represent the JSON attributes and edges represent the inclusive relationship between the

attributes. In order to preserve the ancestral relationship of attributes, every attribute in a JSON tree must preserve its parent information. Hence, in this thesis, the inclusive relationship of attributes is maintained by flattening the attributes. i.e., the parent attributes are concatenated with the child.

As a first step, the JSON documents are flattened by parsing the JSON tree in a depth-first order and concatenating the parent and child attributes. More precisely, the flattened representation of an attribute denotes the root-to-leaf path of an attribute. Hence, the schema comprises a set of flattened attributes extracted from a JSON document. Table 4.1 depicts the sample JSON document collection and their schema representation. The schema S is formally defined in Definition 4.3.1.

Definition 4.3.1. Given a document $D \in G$, a schema S is a set of flattened attribute names K such that $K = \{{k_a}^{prim} \cup {k_a}^{comp}\}$ where ${k_a}^{comp}$ is the unique pathname obtained by concatenating the pathname from root r to k_a.

4.3.2 Feature Extraction

JSON documents comprise both ordered (array) and unordered (nesting object) attributes. In order to support both ordered and unordered properties of JSON, this work takes unigram and trigram attributes as input and feeds them to *SchemaEmbed* separately to generate schema embeddings E_u and E_t respectively.

Unigrams Extraction: Given JSON attributes $\{A_1, A_2, ..., A_m\}$, the *SchemaEmbed* model aims to extract schema embeddings E_u for G. In this work, the attributes in JSON schemas are flattened. Hence, the set of unordered flattened attributes or unigrams U together forms a document. For instance, the unigram set U = { *Paper_Title, Paper_Authors, Paper_Conference Name, Paper_Year, Paper_Headline, Paper_Reporter, Paper_Company, Paper_Date*} for the schemas a and b in Table 4.1.

Trigrams Extraction: When the unigram attributes are considered for representing the attribute embeddings, the skip-gram model generates dense representation for un-

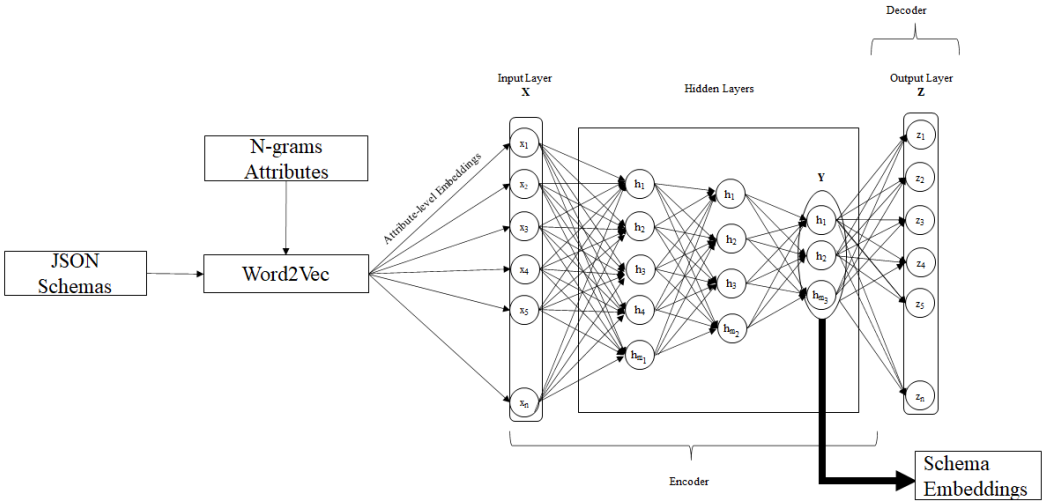


Figure 4.2: Flow description of *SchemaEmbed* model

ordered attributes and ignores the substructure of attributes in a schema. However, a JSON schema comprises unordered and ordered attributes, and hence capturing the substructures of the schema preserves the ordered nature of the JSON schema. Hence, in this work, the schema embeddings are generated using the n-gram attributes of schemas. In general, the size of the n-grams ($n > 1$) might be treated as a parameter. Nevertheless, we chose to set it to 3 because preliminary tests revealed that trigrams performed well and were consistent. This is due to the fact that the frequent ordered attributes are captured due to the presence of the array. Hence, non-array attributes are unordered in most cases. Increasing the n value beyond three results in a smaller n-gram set, which may not effectively capture the hidden semantics among the n-grams.

Given a collection of JSON documents represented as $G = \{D_1, D_2, ..., D_n\}$ where D_i comprises of a set of flattened JSON attributes $\{A_1, A_2, ..., A_m\}$, the trigrams $t_i = \{t_{i_1}, t_{i_2}, ..., t_{i_x}\}$ for each D_i where $t_{i_i} = (A_p, A_q, A_r)$ are extracted. The trigram set T comprises all the trigrams extracted from all schemas. For instance, the trigram set T = { *[Paper_Title, Paper_Authors, Paper_Conference Name], [Paper_Authors, Paper_Conference Name Paper_Year], [Paper_Headline, Paper_Reporter, Paper_Company], [Paper_Reporter, Paper_Company, Paper_Date]*} for the schemas a and b in Table 4.1.

4.3.3 SchemaEmbed Model

In general, the pre-trained models capture different meanings of a word based on the context. The contextualized representations help in improving the clustering quality. Hence, in this work, the pre-trained models are used to preserve the key contexts of the document. The *SchemaEmbed* model is proposed using the Word2Vec and a deep autoencoder. Algorithm 4.1 shows the pseudocode of the *SchemaEmbed* model, and it has been depicted pictorially in Figure 4.2.

Algorithm 4.1: *SchemaEmbed* Model

```
1  S:= Schemas, N:= N-Grams Vocabularies;
2  Function SchemaEmbed(S, N):
3      initialize w2v_dimension, epoch;
4      w2v_model := Word2Vec(N, w2v_dimension);
5      foreach S_i ∈ S do
6          Extract the N-Grams N_i;
7          foreach N_i ∈ S_i do
8              attributeEmbedding[S_i] :=
                 attributeEmbedding[S_i].append(w2v_model[N_i]);
9          end
10         schemaEmbedding[S_i] := concatenated attributeEmbedding[S_i] ;
11     end
       /* Deep Autoencoder                                          */
12     Let X := schemaEmbedding[S] ;
13     Let A, B, C, D be the Weight Matrix and a, b, c, d be the bias for the layers ;
14     H_1 := tanh(AX + a);
15     H_2 := sigmoid(BH_1 + b);
16     Y := sigmoid(CH_2 + c);
17     Z := ReLU(DY + d);
18     L(x_i,z_i) := Σ_{i=1}^{n} x_i log(z_i) + (1-x_i)log(1-z_i);
19     Train the deep autoencoder from lines 13 to 16 for the given epoch to minimize L ;
20 return Y
```

Word2Vec aims to learn an embedding matrix $F \in \mathbb{R}^{W \times d}$ for attributes A where W and d represent the vocabulary size and the dimension, respectively. i.e., For each target attribute a_t, the pre-trained Word2Vec model employs a skip-gram model to learn distributed representation of attributes. The objective of the skip-gram model (Mikolov et al. 2013) is to maximize the average log probability by

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(A_{t+j}|A_t) \tag{4.1}$$

where T represents the number of training attributes, and c represents the training context size.

Each attribute A in the document is represented as a vector, and for each input vector a_t, the probability of predicting the output vector a_{t+j} using the softmax function can be found as

$$P(a_{t+j}|a_t) = \frac{exp({v'_{a_{t+j}}}^T v_{a_t})}{\sum_{a \in V} exp({v'_a}^T v_{a_t})} \tag{4.2}$$

where the input and output vectors of the attribute A are represented by v_a and v'_a respectively. This model attempts to predict the surrounding attributes based on a target attribute. i.e. the attributes $author, conference\ name$ and $conference\ year$ belongs to a context $conference$. Unlike regular words, the words in this work are unordered flattened attribute names where the words $authors\_author$ and $authors$ are more similar. Since the skip-gram model does not preserve the word order explicitly, the surrounding attributes predict the context for a schema.

While Word2Vec learns the dense representation from the context of attributes, the autoencoder learns the dense representation from the context of schemas. Hence, the attribute vectors are concatenated for each schema and given as input to the autoencoder.

Basic Autoencoder: The encoder-decoder, an unsupervised model, learns to map an input to output through a two-step process. (i) Encoding: The encoder compresses input x into latent vector representation y such that $y = f(x)$. (ii) Decoding: The decoder predicts output z from y such that $z = x' = g(y)$. The best encoding-decoding scheme is learned through iterative optimization. The decoded output is compared with the input at each iteration and minimizes reconstruction error between input and output across all the samples by backpropagation. The objective function of a basic autoencoder is given by

$$L' = \sum_{i=1}^{n} L(x; g(f(x))) \tag{4.3}$$

where L represents the loss function, n represents the number of documents, x represents the input vector, $f(x)$ represents the encoded vectors in latent space, and $g(f(x))$ represents the decoded vector which is as close as the input vector. The underlying

semantics of the input x is likely to be captured in the latent representation y by minimizing L' between input and output across all the samples.

Deep Autoencoder: The deep autoencoder has two purposes: (i) generate low-dimensional data and (ii) learn contextualized schema embeddings. The deep autoencoder works in four simple steps:

1. Get the concatenated attribute embeddings associated with each attribute in a schema
2. Pass the resulting embeddings through the hidden layers with non-linear activation functions
3. Train the encoder by back-propagating the errors
4. Continue iterating until the minimum error is achieved

Let X, Y, and Z represent the input vector, encoded and decoded latent representations, respectively. Given the input samples $X = \{x_1, x_2, ..., x_n\}$, this work aims to determine the non-linear mapping function $f : X \rightarrow Z$ and $g : Z \rightarrow X$ such that, for $x_i \in X$, the corresponding decoded representation $z_i \in Z$, and reconstructed sample $x'_i \in X'$ is obtained.

The intuition behind the deep autoencoder is to increase the degree of abstract representation of the input. The non-linear activations on attribute embeddings generate compact schema-level embeddings with respect to their corresponding cluster centers. The number of hidden layers varies for deep autoencoder. In this work, the deep autoencoder has an input layer, three hidden layers, and an output layer — the more deep the autoencoder, the less the reconstruction loss.

The number of neurons in the input and output layers is proportional to the input size, i.e., the number of schemas. The hidden layers have a varying number of neurons, such as 160, 80, and 40, respectively. The number of neurons progressively decreases as the network structure deepens. Each layer in the autoencoder is fully connected with the next layer. For an input vector $x_i \in X$, rather than passing this representation directly

to an output layer, we add additional transformations on x_i by adding more hidden layers before clustering. For a five-layer auto-encoder, say input layer, an output layer, and three hidden layers (H_1, H_2, and H_3), the mapping function for a fully-connected autoencoder is computed as

$$\begin{aligned} H_1 &= f_1(AX + a) \\ H_2 &= f_2(BH_1 + b) \\ Y \equiv H_3 &= f_3(CH_2 + c) \\ Z &= f_4(DY + d) \end{aligned} \tag{4.4}$$

where A, B, C and D represent the weight matrices, X represents the input vectors, H_1, H_2, H_3 represents the hidden layer vectors at each layer. Z represents the output at the decoder layer. The bias terms at each layer are given as a, b, c, and d. For an input vector $x_i \in X$, each node in the hidden layers is regulated by non-linear activation functions $f_1(.), f_2(.), f_3(.)$ and $g(.)$ such as $tanh, sigmoid$, etc.

The efficiency of the deep autoencoder is estimated by reducing the reconstruction error as follows:

$$L' = \sum_{i=1}^{n} L(x_i; g(Df_3(Cf_2(Bf_1(Ax_i + a) + b) + c) + d)) \tag{4.5}$$

where L is defined as

$$L(x_i, x_i') = \sum_{i=1}^{n} x_i \log(x_i') + (1 - x_i) \log(1 - x_i') \tag{4.6}$$

For all the hidden layers of the encoder and decoder, we use $tanh$ and $sigmoid$ activation functions for non-linearity. The non-linear mappings enhance the data representation of the input. The decoded vector is output through the $ReLU$ activation function and batch normalization. The dimensions of the resulting vector Z are the same as those of the input dimension. Batch normalization is applied after each layer to normalize the output (activation) of a previous layer using zero mean and unit standard deviation. The difference between the expected and actual values is calculated, and the error is propagated from the output layer to the first hidden layer. The weight matrix is adjusted to the input layer and continues iterating until the error falls into the required range or after a certain number of iterations has been completed. The backpropagation

aims to reduce the reconstruction error.

Schema Embeddings: The unigrams U and trigrams T are considered as vocabularies to *SchemaEmbed*. The *SchemaEmbed* encoder generates E_u and E_t as output which will capture the contextual similarity between schemas based on unordered and ordered attributes as well.

4.3.4 Clustering

SchemaEmbed learn the non-linear mappings from the concatenated unigram and trigram attribute embeddings. The encoder of *SchemaEmbed* model generates the cluster-friendly schema level embeddings $E_u = \{u_1, u_2, ..., u_n\}$ and $E_t = \{t_1, t_2, ..., t_n\}$ respectively. Since both E_u and E_t captures the contextual similarity of schemas, the maximum of two similarity scores determines the clusters. Therefore, the pair-wise similarity matrices M_u and M_t are computed for E_u and E_t, respectively. The similarity matrix M_u is represented as

$$M_u = [d_{u_i,u_j}]_{n \times n} \tag{4.7}$$

where $u_i, u_j \in E_u$, and

$$d_{u_i,u_j} = \frac{(u_i)^T (u_j)}{\parallel (u_i) \parallel \parallel (u_j) \parallel} \tag{4.8}$$

The similarity matrix M_t is also computed in the same way as equations 4.7 and 4.8. The final similarity matrix $M \in \mathbb{R}^{n*n}$ is computed as

$$M = Max(M_u, M_t) \tag{4.9}$$

Given the similarity matrix M, the task of *embedding-based clustering* approach is to learn a set of K clusters $C = \{C_1, C_2, ..., C_K\}$, $\bigcup_{i=1}^{K} C_i = M, C_i \cap C_j = \phi$ for $1 \leqslant i \neq j \leqslant K$ in the feature space M. In this work, similar schemas are clustered using spectral clustering algorithm (Von Luxburg 2007) with M. The clustering problem can now be reformulated as an undirected similarity graph $B = (V, E)$ where $V = \{v_1, v_2, ..., v_n\}$ represents the vertices and E represents the weighted edge between v_i and v_j. The edge is weighted using $s_{ij} \in M$. The Degree matrix H is termed as $\{h_1, h_2, ..., h_n\}$ is a diagonal matrix where the degree h_i of a vertex $v_i \in V$ is defined as

$$h_i = \sum_{j=1}^{n} s_{ij} \tag{4.10}$$

Given a similarity matrix M, the normalized Laplacian matrix $L = \mathbb{R}^{n*n}$ is defined as

$$L = H^{-1/2}(H - M)H^{-1/2} \tag{4.11}$$

Compute the first k's eigenvectors $I = \{i_1, i_2, ..., i_k\}$ associated with L where $I \in \mathbb{R}^k$ and let the vectors represent the columns of I. The i[th] row of I is represented as $j_i \in \mathbb{R}^k$. The data points $\{j_1, j_2, ..., j_n\}$ in $\mathbb{R}^k$ are clustered using K-Means clustering. Algorithm 4.2 provides the pseudocode of clustering JSON documents using *SchemaEmbed* model.

Algorithm 4.2: Clustering JSON documents

Input: JSON Document Collection D = {D_1, D_2, ..., D_n}
Output: Clusters C = {C_1, C_2, ..., C_K}
initialize w2v_dimension := 50, epoch := 50 ;
Let Unigrams Similarity Matrix $M_u \in \mathbb{R}^{n \times n}$, Trigrams Similarity Matrix $M_t \in \mathbb{R}^{n \times n}$, and Combined Similarity Matrix $M \in \mathbb{R}^{n \times n}$;
Extract Schemas S = {S_1, S_2, ..., S_n} from D ;
`/* Feature Extraction */`
foreach $S_i \in S$ **do**
    construct unigrams and add to U;
    construct trigrams and add to T;
end
E_u = SchemaEmbed(S, U);
E_t = SchemaEmbed(S, T);
`/* Construction of Similarity Matrices */`
foreach *(i, j)* $\in E_u$ **do**
    M_u := cosine(i,j);
end
foreach *(i, j)* $\in E_t$ **do**
    M_t := cosine(i,j);
end
foreach *(i, j)* $\in n$ **do**
    $M[i][j] := Max(M_u[i][j]), M_t[i][j]$
end
`/* Clustering */`
Let clusters $C = \{C_1, C_2, ..., C_K\}$ and centroids $c = \{c_1, c_2, ..., c_K\}$;
Cluster the schemas and the documents using spectral clustering algorithm ;

4.4 IDENTIFICATION OF SCHEMA VARIANTS

The embedding-based clustering approach clustered the JSON documents based on the contextual similarity of JSON schemas. This section describes the identification of schema variants at each cluster and how the new data structure aids in summarizing the

schema variants.

In this thesis, we examine the power of contextual and structural similarity in determining similar JSON schemas. Contextual similarity measure aids in determining the semantic similarity of JSON schemas based on the context in which the attributes are involved. In addition, finding the structural similarity prior to contextual similarity limits the vector space. Hence, in this work, the contextual similarity of JSON schemas is identified prior to structural similarity.

Finding the structural similarity determines the core attributes and schema-specific attributes present in a cluster. The strength of *SchemaEmbed* model using Word2Vec and deep autoencoder lies in handling large collections and training the word vectors efficiently. The purpose of using Word2Vec in this work has two advantages: (i) Word2Vec generates a low-dimensional dense representation of attributes by considering the presence of surrounding attributes. (ii) Due to the use of word frequencies, the vocabulary list in the Word2Vec model for each dataset helps to identify the core attributes A_c present in a cluster. However, with only the word frequencies, Word2Vec fails to determine the schema-specific attributes A_s in each schema variant. As a result, a well-known Inverse Document Frequency (IDF) approach is used to estimate the attributes specific to a schema. The IDF measure can be calculated as

$$IDF_a = log\frac{G}{(df_a)} \tag{4.12}$$

where df_a is the number of documents containing an attribute *a*, and *G* represents the number of documents in a collection. The core and schema-specific attributes are defined formally in Definition 4.4.1.

Definition 4.4.1. For a JSON document *D*, the set of attributes *A* can be represented as $A = \{A_c \cup A_s\}$ where A_c and A_s represents the core and schema-specific attributes respectively. The core attributes are shared by all schema variants of a collection, and schema-specific attributes are unique to a schema variant.

4.4.1 SVTree

SVTree is an ordered tree capable of representing the schema variants in a compressed form. It is built by reading attributes at each schema extracted sequentially and mapping each attribute to a path in an *SVTree*. Schema variants may have core attributes, and hence the path or branch of a tree may overlap. As the more branches overlap, the more compression we achieve in *SVTree*. For instance, given two schemas $b_1 = \{b_{a_1},\ b_{a_2},\ ...,\ b_{a_m}\}$ and $b_2 = \{b_{b_1},\ b_{b_2},\ ...,\ b_{b_n}\}$ where $m \leq n, \forall i \in [1, m], b_{a_i} = b_{b_i}$, then b_1 is called a subset or prefix of b_2.

The *SVTree*, $SVT = (r, V, E)$ is a directed tree where V is a finite set of nodes (vertices). Each interior node $v_i \in V$ is represented by its unique identifier of the node $\{nodeName_i\}$. The radix node $v_k \in V$ is represented by the 2-tuple $\{nodeName_i,\ attList\}$ and each leaf node $v_j \in V$ is represented by the 3-tuple $\{nodeName_j,\ SVNum,\ DocIDList\}$ where

- $nodeName_j$ is the unique identifier of the node j
- $attList$ is the list of core attributes A_c
- $SVNum$ is the unique identifier of the schema variant
- $DocIDList$ is the list of document IDs following the specific schema variant (branch) in a tree

E is a set of directed edges of SVT where $e \in E$ is represented by a tuple (v_m, v_n) with $v_m, v_n \in V$ and $v_m \neq v_n$.

The size of *SVTree* has a significant impact on mining the tree for incremental updates or any search operation. Hence, in order to compress the size of *SVTree*, schema restructuring is used to reorganize the core and schema-specific attributes in sorted order so that maximum attributes in a branch can be overlapped. In other words, the maximum prefix can be obtained. Schema restructuring is important since the frequency-descending order of attributes can result in a higher degree of prefix sharing. This sort of order can be used to re-order the attributes during restructuring. However, the restruc-

turing not only provides the compact tree but also the structure of attributes (inclusive relationship) in each schema variant is preserved.

We next describe the construction of *SVTree* at each cluster to show the summarised representation of schema variants available in a collection. The flow of *SVTree* construction works in two phases:

1. Schema restructuring phase:

   (a) Sort the core attributes based on lexicographic order in order to follow the core sequence of attributes at all schemas in a cluster.

   (b) Sort the schema-specific attributes in increasing order of IDF score at each schema because the lower the score, the probability of the attribute in other documents are high.

2. Insertion phase:

   (a) Create root and radix node in *SVTree*. Insert all core attributes in the radix node. In general, each attribute corresponds to a node in *SVTree*. However, the large number of attributes increases the nodes in *SVTree*. Hence, in this work, the core attributes are compressed into an array and stored in a single radix node.

   (b) For each schema-specific attribute A_j in S_i,

      i. traverse SVT

      ii. if A_j already exists in *SVTree* then continue else create a new branch and insert all the remaining schema-specific attributes of S_i in *SVTree*.

Algorithm 4.3 describes the construction of *SVTree* where the vertices (nodes) and edges are set to ϕ. Initially, the root node is inserted in *SVTree*. The first branch of a tree is constructed by the first schema in a collection where the core attributes are marked as $radixNode$, and the attributes in the list of schema-specific attributes correspond to the interior node in *SVTree*. The end of schema-specific attribute list is marked as

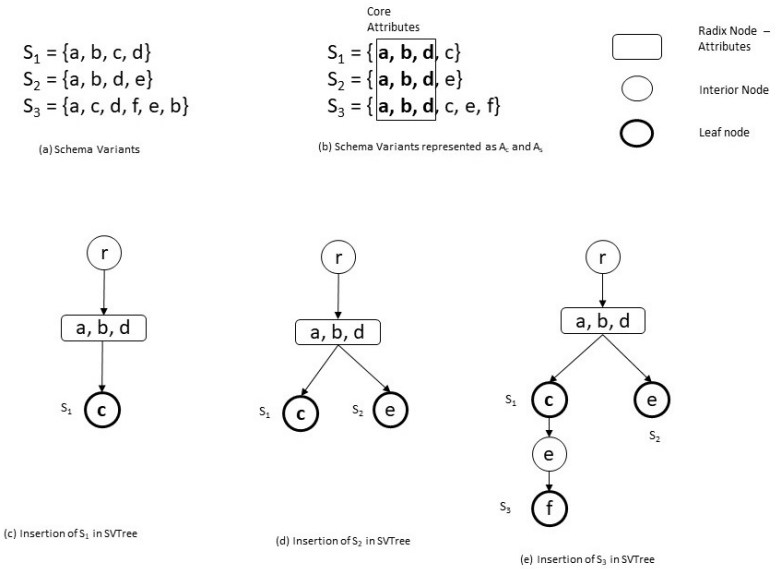


Figure 4.3: Construction of *SVTree*

$leafNode$. The core attributes are inserted only once in *SVTree*. The algorithm gets the schema-specific attribute list of each schema and continues insertion.

Algorithm 4.4 describes the insertion of schema-specific attributes in two ways: (i) If a branch already exists in *SVTree*: Iterate over the nodes in a specific branch and check for the same sequence of attributes. In the case of *SVTree*, the interior node could also be marked as $leafNode$ when one schema variant is a subset of another, i.e., if the nodes in the branch is a subset of a new schema variant that is being inserted, then continue inserting nodes in the same branch and mark the leaf nodes appropriately. For instance, in Figure 4.3, S_1 is a subset of S_3. In this case, the interior node of S_3 is a leaf node of S_1. Hence, the interior node acts as a leaf node to represent the schema variant. However, the leaf nodes contain children, and it is explained pictorially in Figure 4.3. If the nodes in a branch are a superset of the new schema variant that is being inserted, then one of the interior nodes in a branch must be a leaf node for the new schema variant (ii) Create a new branch in *SVTree*: If the attributes in new schema variant are not present in *SVTree*, then create a new branch from $radixNode$ and insert the remaining attributes.

The identification of schema variants using *SVTree* is explained briefly in algorithms 4.3 and 4.4. Figure 4.3 illustrates the *SVTree* constructed for the given schemas. Figure

Algorithm 4.3: Construction of *SVTree* - Summarized schema variants

Input: JSON Schema Variants S = {S_1, S_2, ..., S_n}
Output: *SVTree* SVT

initialize SVT: V = ϕ, E = ϕ;
SVNum=0;
currentNode := SVT.addRootNode($root$);
V := V ∪ currentNode;
Read S_1 ;
/* Insertion of Core Attributes $S_1 = A_c \cup A_{s_1}$ */
childNode := SVT.addRadixNode(A_c);
V := V ∪ childNode;
newEdge := SVT.addEdge(currentNode, childNode);
E := E ∪ newEdge;
SVT.currentNode = childNode;
SVT.radixNode := SVT.currentNode;
/* Insertion of Schema-specific Attributes */
foreach $a \in A_{s_1}$ **do**
    childNode := SVT.addNode(a);
    V := V ∪ childNode;
    newEdge := SVT.addEdge(SVT.currentNode, childNode);
    E := E ∪ newEdge;
end
SVNum := SVNum + 1;
SVT.leafNode := childNode;
SVT.leafNode.DocIDList += S_1;
SVT.leafNode.SVNum := SVNum;
foreach $S_i \in S$ **do**
    traverse SVT till SVT.radixNode;
    SVT.currentNode := SVT.radixNode;
    SVT.currentNode := SVT.nextNode;
    SVT := InsertSS(A_{s_i}, S_i, SVT, SVT.currentNode, SVT.radixNode);
end

4.3(a) represents the schema variants $\{S_1,\ S_2$ and $S_3\}$. After finding core and schema-specific attributes, the attributes are re-ordered in Figure 4.3(b). Figure 4.3(c) describes the construction of *SVTree*: Parse the attributes in S_1 and insert in *SVTree* if not exist. The rectangle node represents the radix node containing core attributes, circle nodes in bold represent the end of the schema. Figure 4.3 (d) describes the insertion of S_2 in *SVTree* after S_1: Parse the attributes in S_2 and insert in *SVTree* if not exists. e is not in *SVTree*. So, create a new node after the radix node. Now radix node has two children, and both are leaf nodes (end of schema). Figure 4.3 (e) describes the insertion of S_3 in *SVTree* after S_2: Parse the attributes in S_3. The schema-specific attributes are

Algorithm 4.4: *SVTree* Insertion

```
1  Function InsertSS(A_s, S, SVT, SVT.currentNode, SVT.radixNode):
2      foreach a ∈ A_s do
3          if a := SVT.currentNode then
4              if SVT.nextNode != φ then
5                  SVT.currentNode := SVT.nextNode;
6              else
7                  SVT := InsertSVT(a, A_s, S, SVT, SVT.currentNode)
8              end
9          else
10             SVT := InsertSVT(a, A_s, S, SVT, SVT.radixNode)
11         end
12     end
13 return SVT
14 Function InsertSVT(a, A_s, S, SVT, SVT.currentNode):
15     if a is end of A_s then
16         childNode := SVT.addNode(a);
17         V := V ∪ childNode;
18         newEdge := SVT.addEdge(SVT.currentNode, childNode);
19         E := E ∪ newEdge;
20         SVNum := SVNum + 1;
21         SVT.leafNode := childNode;
22         SVT.leafNode.DocIDList += S;
23         SVT.leafNode.SVNum := SVNum;
24     else
25         childNode := SVT.addNode(a);
26         V := V ∪ childNode;
27         newEdge := SVT.addEdge(SVT.currentNode, childNode);
28         E := E ∪ newEdge;
29         SVT.currentNode = childNode;
30     end
31 return SVT
```

(c, e, f). While traversing SVT, the core attribute nodes and schema-specific attribute c are already in the tree. Hence, S_3 follows the first branch and added (e, f) immediately below c. f is marked as the leaf node.

In general, the number of nodes determines the size of the *SVTree*. In the worst case, the maximum number of nodes in *SVTree* is 2^k, where k represents the number of attributes present in distinct schema variants. The worst case is due to the high number of unique attributes in each schema variant. However, the size is too large to contain all the unique schema variants in an *SVTree*. Nevertheless, in a real scenario, the set of

schema-specific attributes in a specific schema variant may be shared with other schema variants. Figure 4.3 illustrates this scenario. In such a context, the shared attributes overlap with the nodes that already exist in a tree. Hence, the maximum number of nodes in *SVTree* can be written in different scenarios as: The number of nodes is 2^k in the worst-case where k represents the number of attributes in distinct schema variants. On average, there are $(1 + n)$ nodes where n represents the number of schema-specific attributes. If the subset of schema variants is present in a collection, then $(1 + \mathrm{A} + \mathrm{B})$ nodes are present in *SVTree* where $\alpha \subset \mathrm{A}$, $\mathrm{B} = (n - \mathrm{A})$. A is a superset variant of α and 1 represent the radix node. The average case is due to the reasonable core attributes present in a collection, and the best case occurs when there are subsets of schema variants.

4.5 EXPERIMENTAL EVALUATION

This section presents the experimental results of the proposed approach with existing approaches.

4.5.1 Datasets

We use both real-world data used in literature and synthetic dataset for our experiments.

1. DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017): This real-world dataset contains 2 million XML documents scraped from DBLP and converted to JSON. In this work, we have chosen 2,00,000 documents randomly from 2 million documents. The documents are flattened and include eight types of publications, such as conferences, journals, theses, etc. The dataset comprises 11 schema variants and 25 unique attributes.

2. The synthetic dataset (SD) [1] is also populated for publication scenarios with references from various publications such as IEEE[2], ACM[3], and so on. The JSON documents are flattened, and eight types of publications are supported, such as patent, conference, journal, book, miscellaneous, thesis, and so on. There are 39

[1] https://github.com/umagourish/Synthetic-Datasets
[2] www.ieee.org
[3] www.acm.org

schema variants and 77 unique attributes in this collection. The number of schema variants does not correlate with the number of different types of publications.

Both datasets together have 50 schema variants, i.e., different structures and 13 classes. The schema variants are classified based on the presence or absence of attributes. The number of clusters for measuring the similarity is decided based on the number of classes in both datasets. The data collection includes arrays and nesting levels. Hence, the real dataset and synthetic dataset have different hierarchical structures. As both DBLP and synthetic datasets belong to the publications scenario, the schema variants have shared attributes.

4.5.2 Experimental Setup

The proposed approach uses Python language and Keras framework for implementing deep autoencoder. The dimension and window sizes for the Word2Vec model are set to 300 and 3, respectively. The deep autoencoder is designed with an input layer, three hidden layers, and an output layer. The network dimensions are x-160-80-40-x, where x represents the size of the input. The weight matrix for every layer is randomly assigned from a Gaussian distribution. Each activation layer is followed by batch normalization of size set to 50. The fully connected deep autoencoder is fine-tuned for 1500 iterations without dropout. These parameters are set to achieve a reasonable reconstruction loss for the dataset. The model is optimized with *adam* optimizer. The proposed deep autoencoder is trained using a backpropagation algorithm according to the values of the loss function obtained through processing iterations.

4.5.3 Evaluation Metrics

Typically, there are two types of metrics used to evaluate the efficiency of document clustering algorithms: intrinsic and extrinsic metrics. In this work, we employ Silhouette Co-efficient (SC) to evaluate unsupervised clustering. The most common extrinsic metrics, such as precision, recall, and F1-Measure, depend on the alignment of cluster labels to ground truth labels that are problematic for a wide variety of labels. In this instance, the measures such as Normalised Mutual Information (NMI) score, Adjusted Mutual Information (AMI), and Adjusted Rand Index (ARI) score are more appropriate

because they are not influenced by the absolute label values (Vinh et al. 2010). In addition, this work uses a cosine similarity score to measure the contextual similarity of the schemas.

4.5.4 Results and Discussion

This section begins with an introduction to the research questions and then reports on our findings in relation to each research question.

Research Questions: We conduct a systematic evaluation of our proposed approach using the following research questions (RQ):

1. **RQ1:** How does the proposed work compare with the baseline approaches in terms of efficiency perspective, i.e., the contextual similarity of JSON schemas?
2. **RQ2:** How does the proposed work compare with the baseline approaches in terms of effectiveness perspective, i.e., measuring clustering performance with standard evaluation measures?
3. **RQ3:** What is the impact of clustering in identifying and storing the JSON schema variants in *SVTree*?

4.5.4.1 RQ1: Evaluation of Efficiency

In this chapter, efficiency evaluates the performance of the proposed approach by estimating the contextual similarity score of JSON schemas. It is observed from the literature that few research works have focussed on the semantic similarity of JSON data. To study the effect of contextual representations of JSON schemas, the proposed approach is compared not only with existing work such as JSONGlue (Blaselbauer and Josko 2020) but also with the baseline word-level and document-level word embedding models such as Word2Vec (Mikolov et al. 2013), Doc2Vec (Le and Mikolov 2014), InferSent (Conneau et al. 2017), USE (Cer et al. 2018), Embeddings from Language Models (ELMo) (Peters et al. 2018), and RoBERTa (Liu et al. 2019). JSONGlue uses the knowledge base WordNet (Miller 1998) for identifying the semantic relationships between attributes. All of the baseline models are pre-trained models of supervised/unsupervised learning of words or sentences.

Table 4.3: Cosine similarity of the baseline models and proposed work

S.No	Schemas		Context	Word2Vec (Mikolov et al. 2013)	Doc2Vec (Le and Mikolov 2014)	InferSent (Conneau et al. 2017)	USE (Cer et al. 2018)	ELMo (Peters et al. 2018)	RoBERTa (Liu et al. 2019)	JSONGlue (Blaselbauer and Josko 2020)	*SchemaEmbed*
1	S_1: *book author title year url timestamp biburl bibsource*	S_2:*book author booktitle year volume number pages doi*	book	0.93	0.73	0.75	0.46	0.75	0.84	0.54	0.95
2	S_1:*book author title year url timestamp biburl bibsource*	S_3:*book editor_0 title year publisher address volume isbn abstract edition*	book	0.72	0.77	0.81	0.25	0.8	0.7	0.6	0.95
3	S_4:*inproceedings author_0 author_1 title pages year ee url crossref booktitle*	S_5:*proceedings editor_0 editor_1 title volume year isbn booktitle series publisher url*	conference	0.91	0.68	0.88	0.38	0.82	0.8	0.54	0.96
4	S_6:*manual author title language edition howpublished organization address month year note url*	S_7:*patent author assignee title language address nationality type number day dayfiled month monthfiled yearoryearfiled note url*	dissimilar	0.59	0.24	0.8	0.57	0.84	0.8	0.7	0.18
5	S_8:*article author_0 title year volume number issn url doi journal issuedate address abstract month articleno numpages keywords*	S_9:*article author_0 title year volume number issn url doi journal timestamp biburl bibsource pages*	journal	0.66	0.65	0.83	0.74	0.94	0.95	0.71	0.97

Table 4.3 illustrates the cosine similarity of different pairs of schemas belonging to the same context, i.e., Schemas S_1, S_2 and S_3 belongs to *book* context, S_4 and S_5 belongs to *conference* and so on. The analysis of the similarity score is categorized based on the presence of core attributes:

1. *High core attributes in pair:* In order to show the effectiveness of the proposed approach, the core attributes are further classified into ordered and unordered core attributes.

- **Ordered:** The number of core attributes is high in the pair (S_8, S_9). Still, the proposed approach shows a better score than other models. This is due to the reason that the trigrams captured the subsequence of attributes present in a schema, and this structure contributes to a better similarity score. However, other document embedding models consider the whole structure of a schema, and hence the score of *SchemEmbed* is high compared to the existing approaches.
- **Unordered:** Considering the pair (S_6, S_7), the presence of schema-specific attributes distinguishes the context, and hence the score is 0.18. In addition, the core attributes are present randomly in the pair. While Doc2Vec shows the next least score, other models have shown a high score. Nevertheless, in the case of Word2Vec, the similarity is based on the individual attribute vectors without performing any arithmetic operation on vectors, and hence it shows high similarity. However, Doc2Vec works on the concatenated embeddings, and hence the score is less compared to Word2Vec.

2. *High schema-specific attributes in pair:* The pairs (S_1, S_2), (S_1, S_3) and (S_4, S_5) have more schema-specific attributes, i.e., the attributes are unique to the schema. It is observed that (S_1, S_3) contains more than 50% of schema-specific attributes for which all models yield less score compared to (S_1, S_2) and (S_4, S_5). Word2Vec has achieved 0.93 for (S_1, S_2), and the use of autoencoding further improves its similarity in the proposed work.

The proposed *SchemaEmbed* model uses pre-trained models of Word2Vec and encodes the vectors with different activation functions. By comparing the similarity of Word2Vec and the *SchemaEmbed* model, the use of autoencoding in the proposed model further enhances the performance of embeddings in a feature space. The attribute embeddings generated by the Word2Vec model are based on the co-occurrence of attributes rather than their sequence. However, the document embeddings created by advanced models retain the attribute order in each schema, which results in a high similarity score even when the schemas are not identical by context. From Table 4.3, it is found that the scores of RoBERTa, ELMo and InferSent are close to the proposed work. However, these three models show higher similarity for (S_6, S_7), which actually must be less. This is due to the presence of the same set of subsequences such as $title\ language$ and $month\ year\ note\ url$. However, the presence of subsequences does not affect the similarity score of *SchemaEmbed* because *SchemaEmbed* uses trigrams to capture the subsequence which is not present in (S_6, S_7). Therefore, it is observed that the proposed work shows better performance on the set of unordered as well as ordered flattened attributes. In contrast, other existing models work better on the order of words and sentences.

JSONGlue (Blaselbauer and Josko 2020) constructs a bipartite graph to compare the similarity in schemas. This thesis implements the semantic module of JSONGlue to compare it with the proposed work. In comparison with JSONGlue, it is observed from Table 4.3 that the proposed approach works exceptionally well on all the schema pairs. In the pair (S_6, S_7), the score of JSONGlue is 0.7 because JSONGlue compares the meaning of attributes rather than their context. Furthermore, the efficiency of WordNet relies on its design and word coverage which produces null synsets for unknown words in WordNet. This property distinguishes the proposed approach in finding the contextual relationship based on the co-occurrence of attributes rather than the already existing words in the knowledge base.

The original JSONGlue has calculated the semantic similarity for JSON schema matching. In this work, JSONGlue has been extended such that the semantic similarity of JSON schemas with a threshold of 0.7 has been considered for clustering the

schemas. A similarity score of less than 0.7 is not considered a similar schemas. It is observed from Table 4.3 that the contexts are better captured and exhibit higher effectiveness in grouping the schemas using the proposed approach. One can see that the results of the proposed work are close and better than those InferSent, USE, ELMo, and RoBERTa for each case we have tried.

Results show that the proposed approach performs well in finding contextually relevant JSON documents compared to the baselines. The performance of the proposed *embedding-based clustering* approach demonstrates the power of a deep autoencoder on JSON data. The major advantage of the proposed approach is in handling both ordered as well as unordered attributes.

4.5.4.2 RQ2: Evaluation of Effectiveness

Effectiveness is measured based on the inter-cluster and intra-cluster similarity of JSON schemas. In this work, the K-Means algorithm is used to group the documents into clusters for the dataset taken. Experiments are carried out with varying numbers of clusters.

Table 4.4: Clustering performance to determine the contextual similarity of JSON schemas using external clustering validity metrics

Models	Metrics		
	NMI	AMI	ARI
Word2Vec (Mikolov et al. 2013)	0.72	0.61	0.3
Doc2Vec (Le and Mikolov 2014)	0.4	0.39	0.13
InferSent (Conneau et al. 2017)	0.63	0.62	0.30
USE (Cer et al. 2018)	0.64	0.63	0.35
ELMo (Peters et al. 2018)	0.58	0.57	0.29
RoBERTa (Liu et al. 2019)	0.71	0.51	0.53
JSONGlue (Blaselbauer and Josko 2020)	0.57	0.25	0.13
SchemaEmbed	0.75	0.69	0.58

We evaluated the quality of clusters using external validity measures such as NMI, AMI, and ARI. It is observed from Table 4.4 that the proposed approach outperforms

the baselines over the datasets. It is clear from Table 4.4 that Word2Vec and RoBERTa have shown a close score to the proposed approach on all three metrics. Interestingly, all methods had a major drop in scores between AMI and ARI measures. For instance, Word2Vec gave a high NMI score of 0.72, which is close to the proposed approach. However, it gave an ARI score of 0.3, which has a large drop comparatively. Doc2Vec has shown less score compared to other methods on all three metrics. Although the pre-trained Word2Vec vectors are used in the proposed work, we are able to show improved performance in all the metrics. Figure 4.4 shows how the proposed approach changes NMI, ARI, and AMI values from epoch to epoch. The number of epochs is decided based on the training loss of the *SchemaEmbed* model. Every evaluation metric has an obvious upward trend. This result demonstrates that the proposed *SchemaEmbed* model based on the pre-trained model and deep autoencoder structure work towards the desired direction.

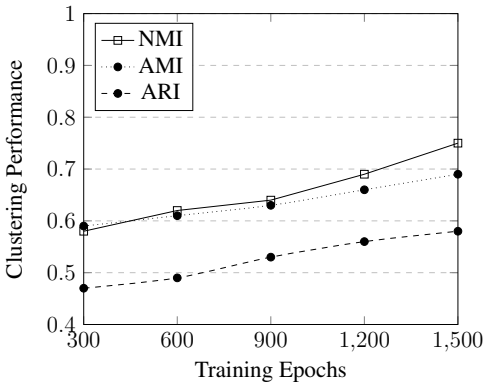


Figure 4.4: Clustering Performance Vs. Training Epochs

Structural Similarity: Most of the existing approaches cited in Chapter 2 use structural properties of JSON documents to extract global, skeleton, or reduced schema. As the BSP approach performs the classification of schema variants based on attribute node conditions, the performance of the proposed approach is compared with the BSP approach.

Both approaches are evaluated for the above-given datasets. The BSP approach is assessed using entropy and sEntropy values, whereas the proposed approach is evaluated with SC, NMI, AMI, and ARI. It is found from Table 4.5 that the BSP approach has

achieved null entropy and sEntropy for the above datasets, ensuring that the approach has achieved maximum precision and concise representation. Although BSP is efficient in describing the schema variants, it does not consider one of the significant properties of JSON data, such as arrays. However, array structures significantly change the category of documents. Furthermore, in schema-based split, the presence or absence of an attribute determines the split, which results in using the structural similarities of JSON schemas for classifying the schema variants. In comparison with the BSP, the proposed approach goes one step further in finding the structural and semantic relationship of JSON schemas, including the internal structures of arrays.

Table 4.5: Clustering performance to determine the structural similarity of JSON schemas

Evaluation Metrics	BSP (Gallinucci et al. 2019)	TF-IDF (Bawakid 2019)	Proposed Work
entropy	0		
sEntropy	0		
SC		0.6	0.68
NMI		0.65	0.75
AMI		0.64	0.69
ARI		0.32	0.58

In addition, the proposed approach is also compared to the conventional TF-IDF-based clustering approach (Bawakid 2019). In contrast to the proposed approach, TF-IDF-based approach performs clustering based on the structural similarity of JSON schemas (frequency of attributes). It extracts the exact schema variants rather than clustering contextually similar schema variants. It is evident from Table 4.5 that the proposed approach achieves better performance than TF-IDF-based approach on all the evaluation measures.

4.5.4.3 RQ3: Evaluation of Schema Variants

This section describes the impact of using clustering on schema variants identification. We also discuss the ways in which the proposed approach is different from existing

approaches in the field of JSON schema extraction.

The efficiency of *SVTree* is evaluated based on core and schema-specific attributes on clustered and unclustered data. Figure 4.5 illustrates the efficiency of *SVTree* before and after clustering. Before clustering, the equivalent schemas are not identified, and hence the number of core attributes is too low with a high number of schema-specific attributes. This increases the number of nodes in *SVTree*. However, after identifying the equivalent schemas, there will be a sufficient number of core attributes in the subset of documents compared to the whole document collection, which introduces node reduction in *SVTree*. For unclustered data, the JSON schemas are parsed and inserted in *SVTree* by traversing over each path in a single pass. However, it incurs a large number of nodes due to the lack of core prefixes in a tree. In Figure 4.5, the unsorted dataset has only one core attribute, i.e., $\_id$, and 392 schema-specific attributes. Although the number of unique attributes is less, the high number of schema-specific attributes is due to the presence of repeated nodes for an attribute. For instance, in Figure 4.3e, e belongs to both S_2 and S_3, and hence the node is repeated. Creating a single node for an attribute that is not in the core attributes list generates a global schema which is not the focus of this work. Whereas if the whole collection is sorted based on core and schema-specific attributes, the number of nodes is probably less than unsorted. It is proved from Figure 4.5 that there are 297 schema-specific attributes in sorted data compared to 392 in unsorted. However, it is still high compared to clustered data. This is due to the fact that the set of core attributes is determined on a large collection. A collection may have distinct schema variants which have fewer core attributes. However, a small collection may contain a large number of core attributes. Hence, clustering has an impact on reducing the search space and introduces the list of core and schema-specific attributes efficiently. In Figure 4.5, clustered schema variants have 184 schema-specific attributes in all clusters, which is 46% lesser than the unclustered tree.

Figure 4.5 is evident for the reduction in the size of the *SVTree* after clustering. Figure 4.3 shows the existence of node redundancy for varying sizes of schema variants. However, the presence of redundancy even after clustering is due to the different sequences of attributes in schema variants. The main reason behind node redundancy is

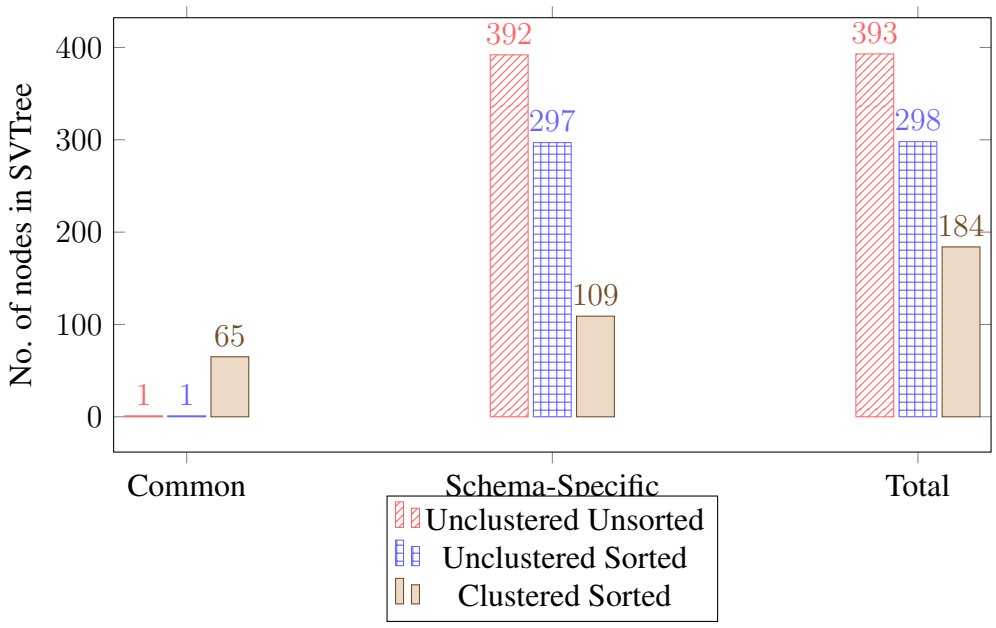


Figure 4.5: Efficiency of *SVTree*

the nature of datasets.

4.6 SUMMARY

In this chapter, an *embedding-based clustering* approach is proposed for discovering schema variants from a heterogeneous JSON data collection by capturing the structure and semantics of attributes in JSON documents. Throughout the literature, all existing works concentrates on schema extraction by considering the structural similarity of JSON schemas. In comparison to the state-of-the-art methodologies discussed in the survey, this research pioneers the use of both structural and semantic similarity for JSON schema inference using a clustering algorithm. The proposed *SchemaEmbed* model produced the JSON schema embeddings. Clustering contextually relevant JSON schemas addressed the semantic matching issues in JSON structures. Literature has seen a few works (Klettke et al. 2015; Wang et al. 2015) designed a data structure to store the global schema. In general, the number of nodes in a tree for constructing a global schema is equivalent to the number of unique attributes in a collection. Hence, the final representation does not explicitly show the exact schema variants that exist in a collection. To address this issue, we presented *SVTree*, a new data structure to provide the summarized representation of schema variants. *SVTree* captures the hidden schema

information, such as core and schema-specific attributes present in a collection. Using the schema variants extracted, any database system can build efficient indexes in order to improve the performance of data retrieval. The extensive experiments on real and synthetic datasets show the effectiveness of the proposed approach with baseline and existing approaches.

CHAPTER 5

SCHEMA-AWARE INDEXING

5.1 INTRODUCTION

Over the past decade, JSON has established itself as the de facto format for storing multi-structured data in NoSQL Document Stores. As documents evolve over time, there is increased complexity in efficiently retrieving data. Indexes have traditionally been used to speed up the search process in hierarchical data. Considering the nature of JSON data where the value of an attribute has different substructures in their nesting level, the parent-child (P-C) and ancestor-descendant (A-D) relationship of attributes must be preserved for efficient retrieval of path queries. Of the wide variety of indexing techniques available for hierarchical data, path-based indexes are particularly effective for fast data access (Sasaki et al. 2020). As the number of heterogeneous JSON sources grows, indexes often consume ample space and degrade data retrieval performance. Thus, there is a demand for fast and space-efficient index structures for JSON documents.

Most search indexes are based on inverted indexes, which use a bag of words model to store and retrieve the structural relationship between terms and documents and their statistical measures of relevance. Inverted index associates each term in the document with JSON objects containing the term, represented as *(term, DocIDs)*. When a search query is executed, the index is queried in a single scan of the index, and the candidate results are identified (Baeza-Yates et al. 1999). Nonetheless, inverted indexes are solely used for finding exact keyword (lexical) matches in each document.

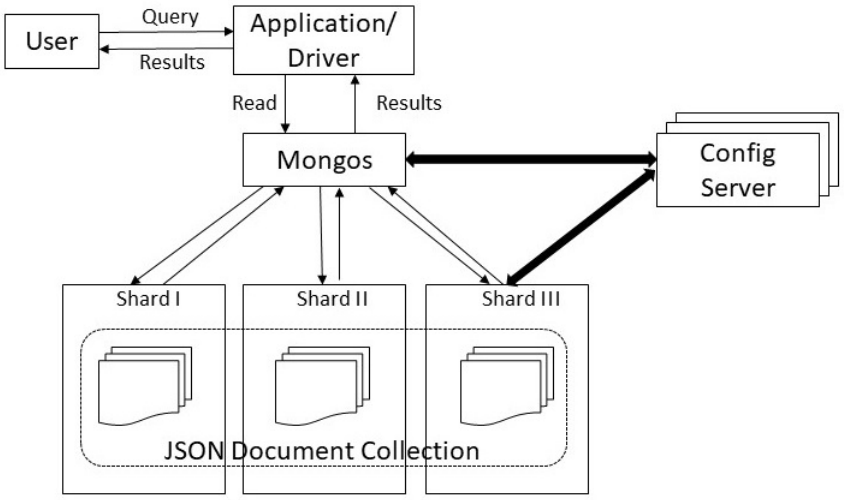


Figure 5.1: Query processing in MongoDB architecture (mongodb schema 2019)

Motivation

Figure 5.1 shows the process of query processing in MongoDB (mongodb schema 2019) distributed environment. The documents are distributed to the shards based on the shard key and partitioning methods. Config Server maintains the metadata information about each shard which helps the mongos (query router) direct the query to the respective shards. For instance, a query $Q = \{conference\ name="IPM"\}$. Suppose if Q is not a covered query, i.e., Q does not contain a shard key, then mongos must broadcast the query to all the shards to retrieve the data. Frequent broadcasting of a query introduces a high query processing cost. This can be improved by constructing the most appropriate index at each shard and config server by identifying the primary attributes that contribute to limiting the amount of data a query must process. This scenario has motivated us to design an efficient and compact index structure that can be placed at each shard and config server for efficient JSON data retrieval.

The following example illustrates the need to explore the contextual relationship of JSON attributes for better information retrieval. JSON documents not only have varied structures but also differ semantically. For instance, consider a *Publication* scenario in the form of JSON document collection, as shown in Table 4.1, from various publish-

ers such as DBLP[1], IEEE[2], ACM[3], and so on. Each publisher uses a different set of attributes to represent the class or category, such as *conference, journal, book*, etc. For example, schemas S_1 and S_3 in Table 4.1 represent the class *conference*, while S_2 belongs to *news article*. In other words, schemas S_1 and S_3 are structurally different but contextually similar, i.e., the same set of attributes are annotated differently in S_1 and S_3. In order to return the results for Q, the naive lexical search returns D_1. However, Q is semantically relevant to D_3 as well. Therefore, the task is to find the relevant schemas for a given query and retrieve the respective documents, such as D_1 and D_3. Because the relevant documents use terms that are similar but different from exact query terms, this problem can be classified as a *vocabulary mismatch problem*. A traditional lexical-based index is unable to deal with this situation. As a result, there is a requirement to address this problem to improve the performance of relevant data retrieval.

Most existing works of JSON data (Budiu et al. 2014; Hamadou et al. 2019; Shukla et al. 2015) focussed on the path-based indexing approach, and all the paths in a JSON document collection act as an index key. In addition, arrays have repeated substructures which increases the size of an index. Therefore, there is a requirement to compress the index for efficient data retrieval without losing essential information. Moreover, they ignore the semantics of data during the construction of the index. While there has been less research effort (Tekli et al. 2019) put into semantic search, existing works have used word sense disambiguation (WSD) to capture the semantic relationship between attributes by employing synsets, hypernyms, and other techniques. WordNet includes linguistic information about the attributes, and the similarity score is based on the abstract meaning of attributes. However, the context assumes more than just linguistic knowledge, i.e., the syntactic and semantic information found on the relationship of surrounding attributes in a schema is included as well. In case of massive JSON data where the structure varies according to the presence or absence of unordered attributes at different levels (schema variants), the contextual similarity is more appropriate in providing more relevant results for applications such as information retrieval.

[1] www.dblp.org
[2] www.ieee.org
[3] www.acm.org

Contributions

In order to address the above-mentioned research problems, this work aims at answering the research question: "*How to efficiently retrieve JSON documents addressing both lexical and semantic matches of queries with improved performance in index size and data retrieval time?*." To address this question, this work proposes a compact indexing scheme for efficiently processing the query with less index size. The performance improvement in index size is achieved by identifying the primary attributes that cause large index sizes and removing them from the index. By using the dense retrieval method, the approach captures the contextual relationship between the query and the schema variants. Dense retrieval is a promising alternative approach that matches query terms (attributes) and schema variants in a low-dimensional embedding space and is as efficient in finding the contextual meaning (Guo et al. 2022). It often uses deep neural networks to learn the dense contextual representations of queries and schemas that contain valuable insights such as semantic relationships, linguistic styles, etc. These low-dimensional dense vectors are frequently referred to as word embeddings. The inner product or cosine similarity between a query and schema embedding is regarded as the relevance score. In this work, the words *semantic* and *context* are used interchangeably.

The major contributions of this work include:

- Proposing a JSONPath index table for indexing the structural information of JSON documents using a novel *JDewey* labeling scheme
- Proposing a compact *JSON Index* (*JIndex*) for lexical matching of queries with improved index size and response time
- Designing an approach to exploit the semantics of schema variants and construct *Embedding-based JIndex* (*EJIndex*) to support efficient semantic search
- Conducting an empirical study to demonstrate the effectiveness and robustness of the proposed work vs. the lexical-only and semantic-only approaches

The rest of the chapter is organized as follows: The preliminaries are given in Section 5.2. Section 5.3 describes the problem along with research objectives, and the proposed

work is explained in Section 5.4. Section 5.5 presents the experimental results and performance analysis. Finally, the chapter is summarized in Section 5.6.

5.2 PRELIMINARIES

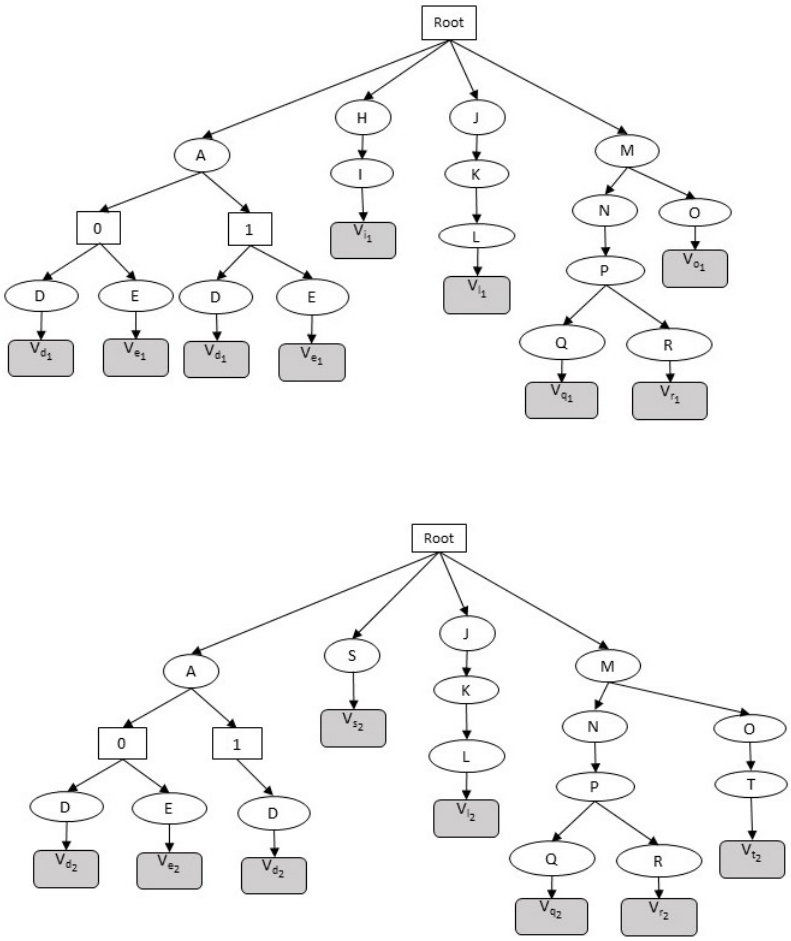


Figure 5.2: Tree Representations of two JSON documents

5.2.1 JDewey Labeling Scheme

In case of tree-shaped data such as XML, in order to retain the structure of XML documents, most of the existing works (Hsu and Liao 2020; Qtaish and Ahmad 2016) have concentrated on labeling the nodes, i.e., the XML nodes are given unique identifiers by preserving the ancestral relationship of the attributes. Similarly, in this work, the prefix labeling scheme (Min et al. 2009) is used to exploit the structural information of the

JSON tree. The commonly used prefix labeling scheme is Dewey order (Tatarinov et al. 2002), where the nodes are given unique labels that describe the path from the root to the node. Unlike XML, the most sensitive part of JSON is an array type representing the set of ordered objects or any primitive data types. Thus, the Dewey order of XML data can not be directly applied to JSON data. Hence, in this work, we propose *JDewey* for JSON documents, an extension of Dewey order for XML data that preserves the P-C and A-D relationship of JSON attributes as well as the array representation of JSON data. In this approach, each node is associated with a vector of integers that corresponds to the path from the document root to the specific node in the document tree. *JDewey* is briefly described as follows:

1. *JDewey(Root)=0*
2. if node *q* is the child of node *p*, then *JDewey(q)=JDewey(p)+"."+q*
3. if node *q* is the n^{th} child of node *p* where *p* is a root node, then *JDewey(q)=n*
4. if node *q* is the n^{th} child of node *p* where *p* is not a root node and of non-array type, then *JDewey(q)=JDewey(p)+"."+n*
5. if node *q* is the n^{th} child of node *p* where *p* is not a root node and of array type, then *JDewey(q)=JDewey(p)+"#"+n*

Figure 5.3 depicts the *JDewey* scheme for index tree representation of JSON documents in Figure 5.2. Unlike XML, JSON documents do not have root attributes. Hence, this work introduces a common root node labeled 0. The ancestors of node Q labeled $4.1.1.1$ are nodes labeled $4.1.1$, 4.1 and 4. In case of an array, the ancestor of node D labeled $1\#1$ is A labeled 1. A is of array type, and hence its immediate children are appended with $\#$.

5.3 PROBLEM DESCRIPTION

Consider a JSON document collection $G = \{D_1, D_2, ..., D_n\}$ where n denotes the size of the collection, and *SVTree*, the proposed work aims to build space-efficient indexes

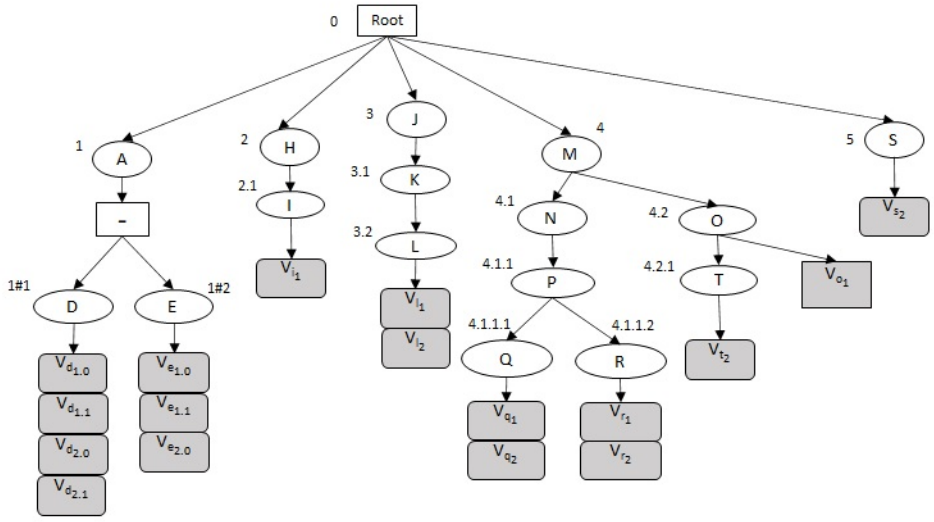


Figure 5.3: *JDewey* scheme for Index tree representation of JSON documents in Figure 5.2

$I \in \{J, E\}$ for efficient retrieval of JSON data, where J denotes lexical match, and E denotes semantic match. The proposed work is also focused on the quality of search relevance and data retrieval time. The following research objectives have been identified to solve the problem described above:

- Analyzing the heterogeneous structure of JSON documents and identifying the primary attributes to build an efficient index structure
- Building a space-efficient index for lexical matching of queries
- Exploiting the contextually similar JSON schemas by analyzing their structural components and building a semantic index with limited size

5.4 LEXICAL AND SEMANTIC INDEXING

This section explains (i) the construction of JSONPath index to determine the core and schema-specific attributes at each nesting level in the documents, (ii) *JIndex* to retrieve exact document matches, (iii) *EJIndex* to retrieve the contextually related JSON documents in a collection, and (iv) a data retrieval phase to process the queries efficiently. The workflow of the proposed work is depicted graphically in Figure 5.4. The description of each phase is given below:

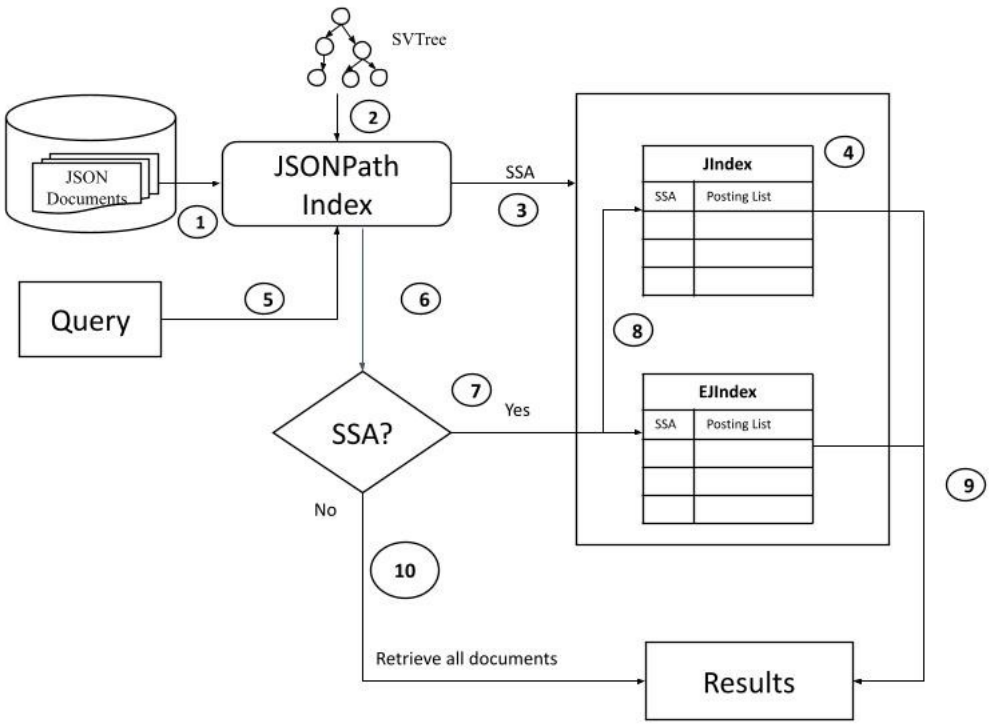


Figure 5.4: Flow diagram of the proposed indexing scheme

5.4.1 JSONPath Index

The proposed JSONPath index aims to minimize the size of lexical (*JIndex*) and semantic (*EJIndex*) indexes and to facilitate efficient query evaluation. In order to have efficient querying of JSON documents, the index must preserve the structural relationships of attributes. This work employs *JDewey* labeling scheme (as described in Section 5.2) to preserve the P-C and A-D relationship of the attributes. However, including these relationships for every attribute in the index incurs space overhead. As a result, the path information for all attributes is efficiently stored in a path table called JSONPath index. The JSONPath index ensures that the parent and children information for each attribute in a collection is used to efficiently handle simple path queries and recursive queries.

Furthermore, the idea of building a JSONPath index is to exploit the core attributes in all the nesting levels of documents in a collection. Although *SVTree* contains the core and schema-specific attributes present in a collection, they represent the root-to-leaf path of attributes. Using this information, the JSONPath index captures the core and schema-specific attributes at each nesting level of documents. This is because core

Table 5.1: JSONPath Index table for JSON documents in Figure 5.2

attributeLabel	attributeName	Parent	Children	Common_Flag
0	Root	-	{1, 2, 3, 4, 5}	1
1	A	0	{1#1, 1#2}	1
1#1	D	1	-	1
1#2	E	1	-	1
2	H	0	{2.1}	0
2.1	I	2	-	0
3	J	0	{3.1}	1
3.1	K	3	{3.2}	1
3.1.1	L	3.1	-	1
4	M	0	{4.1}	1
4.1	N	4	{4.1.1}	1
4.1.1	P	4.1	{4.1.1.1, 4.1.1.2}	1
4.1.1.1	Q	4.1.1	-	1
4.1.1.2	R	4.1.1	-	1
4.2	O	4	{4.2.1}	1
4.2.1	T	4.2	-	0
5	S	0	-	0

attributes exist in a whole collection and hence incorporating them for structure-based queries increases data retrieval time and index size. Furthermore, JSON trees can contain many repeated subtree structures in the form of arrays. We can take advantage of these repeated subtree structures to minimize size and speed up the data retrieval time. The tree representation of the JSONPath index is shown graphically in Figure 5.5.

Assuming that the JSON document is represented as a tree where the attributes are represented as nodes and the edges represent the inclusive relationship between attributes, the JSONPath index is represented by a 5-tuple {*attributeLabel, attributeName, Parent, Children, Common_Flag*} where

- $attributeName_j$ is the name of the node j
- $attributeLabel_j$ is the unique identifier of the node j. It is represented by *JDewey* labels, and therefore, the path of a node j from the root is preserved
- *Parent* of node j denotes the $attributeLabel_i$ of the parent node i

Algorithm 5.1: Construction of JSONPath Index

Input: JSON Document Collection D = $\{D_1, D_2, ..., D_n\}$, SVTree SVT
Output: JSONPath Index P

```
1  initialize P: V = φ;
2  V := V ∈ Root ;
3  Root.attributeLabel := 0 ;
4  Root.attributeName := Root ;
5  Root.Parent := null ;
6  Root.Children := null;
7  Root.CommonFlag := 1;
8  levelCount := 0;
9  foreach D_i ∈ D do
10 |  Insert(D_i, P);
11 end
```

- *Children* of node j contains the $attributeLabel_i$ of the child nodes of node $attributeName_i$
- *Common_Flag* represents whether the attribute is a core or schema-specific attribute. This information is obtained from *SVTree*.

The *JDewey* labels are used to denote the *Parent* and *Children* fields of a JSONPath index, which aids in the identification of the appropriate parent and children nodes within the same index structure. The '-' in the *Children* field indicates that the attribute is a leaf node. The value of *Common_Flag* field is either 0 or 1, where 0 represents the schema-specific attribute, and 1 represents the core attribute.

The JSONPath index table is populated by reading attributes at each document sequentially and mapping each attribute to the entries in the index. Documents may have core attributes, and hence the path or branch of a JSON tree may overlap. As the more paths overlap, the more compression we achieve in the JSONPath index. Algorithm 5.1 explains the construction of the JSONPath index table. Table 5.1 shows the JSONPath index for the documents in Figure 5.2. Since the *Root* node is common for all JSON documents, the table is initialized with *Root* and *attributeLabel* as 0, as per the *JDewey* label rules given in Section 5.2. The *Children* is initialized as null at this step.

Algorithm 5.2 explains the insertion of nodes in the JSONPath index. The JSON trees are traversed in depth-first order, and the process of JSONPath index construction

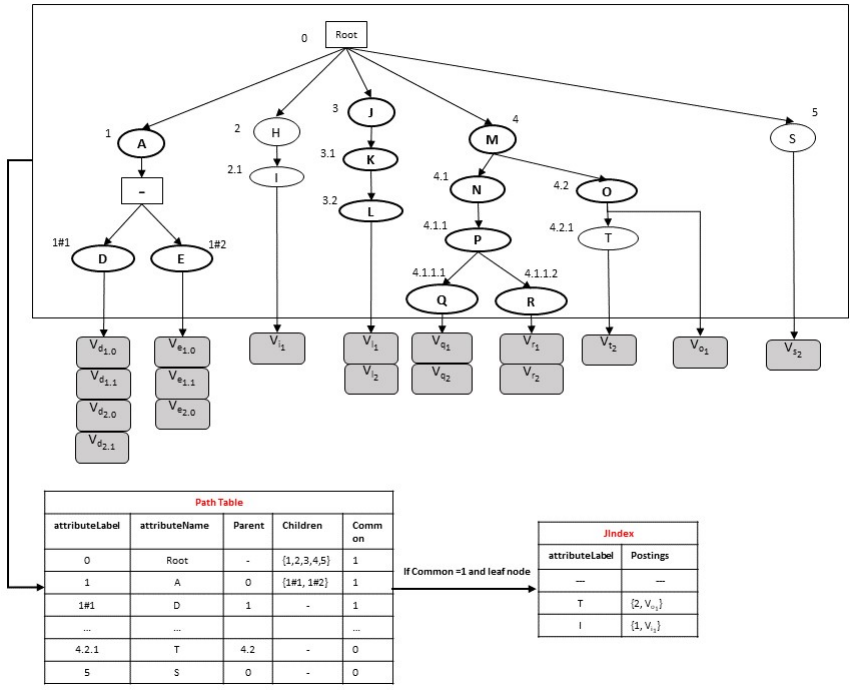


Figure 5.5: A Complete index tree representation for documents in Figure 5.2 with a part of JSONPath Index and JIndex

is as follows:

1. The root node of JSON trees is labeled as '0'.

2. While parsing the node N of the JSON tree,

   (a) if N is not present in the JSONPath index and N is not a leaf node, then insert N in the JSONPath index. Update the *Parent* field and increment the *Count* value. Update the *Children* field of N's parent with N's label. Update the *Children* field of N when the respective child node is parsed.

   (b) else if N is not present in JSONPath index and N is a leaf node, then insert N in JSONPath index. Update the *Parent* field and increment the *Count* value. Update the *Children* field of N's parent with N's label. Update the *Children* field of N with '-.'

   (c) for each attribute A_i in JSONPath index table, if $A_i \in A_c$ then the value of *Common_Flag* is 1 else 0.

Algorithm 5.2: JSONPath Index Insertion

```
1  Function Insert(D, P):
2      childList := [] ;
3      parse D in depth first order;
4      foreach A_j ∈ D do
5          if A_j ∉ P then
6              if A_j is in level-1 then
7                  levelCount := levelCount + 1;
8                  A_j.attributeLabel := levelCount;
9                  A_j.attributeName := A_j;
10                 A_j.Parent := 0;
11                 checkType(A_j);
12             else
13                 checkType(A_j);
14             end
15         else
16             checkType(A_j);
17         end
18     end
19 return P
20 Function checkType(A):
21     if type(A) ∈ object then
22         childCount := 0;
23         for x ∈ A do
24             childCount := childCount + 1;
25             if x ∉ P then
26                 x.attributeLabel := A.attributeLabel + "." + childCount;
27                 x.attributeName := x;
28                 x.Parent := A.attributeLabel;
29                 A.Children := A.Children.append(x.attributeLabel);
30                 if x ∉ SVT.radixNode then
31                     x.CommonFlag := 0
32                 end
33                 checkType(x);
34             end
35         end
36     else if type(A) ∈ array then
37         aCount := 0;
38         foreach x ∈ A do
39             aCount := aCount + 1;
40             if type(x) ∈ object then
41                 childCount := 0;
42                 for y ∈ x do
43                     childCount := childCount + 1;
44                     if y ∉ P then
45                         y.attributeLabel := x.attributeLabel + "#" + childCount;
46                         y.attributeName := y;
47                         y.Parent := x.attributeLabel;
48                         x.Children := x.Children.append(y.attributeLabel);
49                         if x ∉ SVT.radixNode then
50                             x.CommonFlag := 0
51                         end
52                         checkType(y);
53                     end
54                 end
55             else
56                 x.Children := null;
57             end
58         end
59     end
60     else
61         x.Children := null;
62     end
63 return P
```

The complexity arises when parsing the array of objects since an array contains a set of the same or different structured objects. In this case, irrespective of the number of object occurrences, the P-C relationship is considered to decide the path. Hence, this work contributes to compressing the array structure without losing its A-D relationship. Figure 5.5 shows the complete index tree representation for documents in Figure 5.2, in which bold oval-shaped nodes represent the core attributes, and the value list is shown as a set of rectangles. In case of an array of objects, the node A in the first document is

of array type with two objects, each of which contains a set of attributes such as D and E. Since the objects share the same structure, these attributes are mapped together with '-' and represented as D and E with A as the array parent. The respective value nodes are listed in the order of parsing the document.

When the second document is accessed, the nodes A, D, E, and so on are already present in the JSONPath index. As the node S was not previously indexed, the new nodes are placed according to the rules we established. The array parent A has two objects with D, E, and D. Although there is a missing attribute in the second object of A, the presence of already existing D and E nodes in the JSONPath index simplifies the insertion process. The first document has a child node for node O in the JSONPath index, whereas O has a value node in the other. As a result, the JSONPath index is updated to meet the requirements of new documents. The use of the *JDewey* label helps to find the P-C relationship without losing the essential information such as array representation.

5.4.2 JIndex

The most popular index structures available for XML and JSON documents are tree and hash tables. While the path expression plays a vital role in structure-based queries of hierarchical data format such as JSON, tree structures are not efficient in processing partial matching queries (Hsu and Liao 2013). Recursive queries require all interior and leaf nodes to be stored in the index, and hence the index size is huge. Therefore, this work employs a hash-based table to store *JIndex* and *EJIndex* for fast access. In addition, the value of the JSON attribute belongs to a primitive or complex type, i.e., the JSON attribute node must be either a leaf node or an interior node with a subtree as its child. In this case, path indexing allows for faster processing of queries instead of node indexing.

In this thesis, *JIndex* is constructed only for the leaf nodes of the JSONPath index, specifically for the schema-specific attributes A_s. Algorithm 5.3 explains the construction of *JIndex* from the JSONPath Index table. The structure of *JIndex* is represented by a tuple *(attributeLabel, posting list)* where the *attributeLabel* of the leaf node de-

Algorithm 5.3: Construction of JIndex and EJIndex

Input: JSONPath Index P, JSON Document Collection D = $\{D_1, D_2, ..., D_n\}$
Output: JIndex, EJIndex

```
   /* Construction of JIndex                                          */
1  foreach D_i ∈ D do
2      foreach K ∈ D_i do
3          parse the flattened D_i;
4          if P[K.A_p].CommonFlag == 1 then
5              if K.A_p ∉ JIndex then
6                  JIndex[K.A_p] := φ ;
7                  JIndex[K.A_p].postingList := {D_i, K.A_pv};
8              else
9                  JIndex[K.A_p].postingList := JIndex[K.A_p].postingList ∪ {D_i,
                     K.A_pv};
10             end
11         end
12     end
13 end
   /* Construction of EJIndex                                         */
14 Φ(Q) := E_Q(Q) ;
15 Φ(S) := E_S(S) ;
16 foreach P[K.A_p].CommonFlag == 1 do
17     EJIndex[K.A_p] := φ ;
18     sSchema := top-k nearest neighbors of K.A_p ;
19     sDocs := DocIDs of sSchema ;
20     EJIndex[K.A_p].postingList := sDocs;
21 end
```

scribes the path of the leaf node attribute, and the posting list is represented by a set of $\{$*DocID, value*$\}$. Hence, the leaf nodes containing A_s in the JSONPath index become an index key in *JIndex*. While traversing the JSON documents, the posting list of *JIndex* is updated for each path in the document.

Index Pruning: JSONPath index identifies core and schema-specific attributes efficiently. JSONPath index also acts as a pruning technique that prevents core attribute nodes from being processed during query evaluation. In *JIndex*, the leaf nodes are identified by the *Children* field of an attribute. If *Children* is $null$ for an attribute, then the attribute belongs to a primitive data type, and hence it is a leaf node. The *attributeLabel* of leaf nodes determines the path of an attribute and is stored as an index key. The value list and the respective *DocIDs* appended to the leaf node are the posting list for

Algorithm 5.4: Search *JIndex* and *EJIndex*

```
1  Function SearchJIndex(q, θ, JIndex):
2      foreach term ∈ JIndex do
3          if q == term and θ == null then
4              W := JIndex[term].postingList ;
5          else if q == term and θ != null then
6              foreach x ∈ JIndex[term].postingList do
7                  if x == θ then
8                      W := DocID ;
9                  end
10             end
11         end
12     end
13 return W
14 Function SearchEJIndex(q, EJIndex):
15     foreach term ∈ EJIndex do
16         if q == term then
17             W := EJIndex[term].postingList ;
18     end
19 return W
```

the respective index key. In naive existing approaches, all the leaf nodes have an entry in the index. This actually increases the index size. To overcome this problem, in this work, the hash table comprises leaf nodes that are schema-specific attributes.

5.4.3 EJIndex

This section describes the process of constructing *EJIndex* for the semantic retrieval of JSON data collection. The flow of the proposed *EJIndex* construction is explained in Algorithm 5.3. Our workflow includes multiple steps: As a first step, we learn the embedding space for attributes and documents separately by using *SchemaEmbed* model. In this work, query terms are represented as attributes. Since the attributes and documents are represented as a vector, finding their similarity is not challenging. In the second step, each schema-specific attribute A_s observed in the document collection is added as an index key, and its *DocID* is populated as a posting list in the inverted index. The structure of *EJIndex* is similar to *JIndex* and represented by a tuple *(attributeLabel, posting list)* where the posting list is represented by a set of $\{DocID\}$ belongs to the top-k relevant schema variants for a query. In order to identify the most similar docu-

ments for the query, an approximate neighbor search is used that finds the top-k similar schema variants for the query. The steps of this procedure are explained as follows:

5.4.3.1 Learning the embedding space

We assume that queries are attributes from a document collection. Our proposed idea is to represent the query and schema variants as embeddings in order to effectively identify the contextual relationship between the query term and the schema variants. The model comprises three components: a query encoder $E_Q = f(Q)$, which produces a query embedding; a schema encoder $E_S = g(S)$, which makes a schema embedding; and a similarity function has a score between query Q and schema S. In this work, the two models M_Q and M_S are two distinct networks for Q and S that can share parameters. The inner product between Q and S determines the contextual relevance, and the top-k related schema variants are identified. This document retrieval method assumes that the top-k nearest documents in the embedding space will be the most relevant documents to the query term.

Schema Embeddings: The schema embeddings E_t generated in Section 4.3 is used to calculate the contextual similarity with the query embeddings. Hence $\Phi(S) = E_t$.

Query Embeddings: Given a query term Q, the model M_Q is designed with the parameters of M_S where the query encoder E_Q generates query embeddings $\Phi(Q)$ after back-propagation. Various research works have jointly learned query and document embeddings, improving the semantic relevance between queries and documents. However, in our case, we have a structured query containing an attribute or set of attributes. Jointly learning query and schema variants increases the number of core attributes and simultaneously reduces the performance of semantic relevance. The resultant documents are merely equivalent to finding exact matches for a query. Therefore, this work learns query and schema embeddings separately using the same parameters.

5.4.3.2 Building EJIndex

Given a query embedding $\Phi(Q)$ and schema embedding $\Phi(S)$, a relevance estimation function is formally measured as follows:

$$R(\Phi(Q), \Phi(S)) = \frac{\Phi(Q)^T \Phi(S)}{\parallel \Phi(Q) \parallel \parallel \Phi(S) \parallel} \tag{5.1}$$

All documents in the collection can be inversely sorted and posted for each index key. However, due to the sheer size of the document collection, it is not appropriate to include all schema variants in each posting list. As a result, the top-k most similar schema variants can be included in each posting list, excluding the remaining results. For each index key, an approximation of the nearest neighbor search locates and returns *k* schema variants with the highest degree of vector similarity. Hence, the length of the posting list would be determined by the size of the *k* chosen. In this work, the postings are represented by the *DocIDs* of the top-k schema variants returned by the nearest neighbor search. The postings within each posting list may or may not contain the index key, but they are the results that are most similar to the index key based on the learned embeddings.

5.4.4 Query Evaluation

The indexes are built before data retrieval begins. The P-C and A-D relationships are denoted by $/$ and $//$ as defined in Definition 5.4.1. This work aims to answer two types of queries: structure-based queries (SQ) and content and structure-based queries (CSQ) as defined in Definitions 5.4.2 and 5.4.3. These query types are evaluated separately for arrays and nesting levels.

Definition 5.4.1. A query path q is represented by $q = eA_1eA_2...eA_x$ where $e \in \{/, //\}$ and $A_i \in V$.

Definition 5.4.2. An SQ $Q(q)$ consists of query path q of a leaf attribute $A_i \in J$. Q returns JSON Documents $\{D_1, D_2, ...D_k\}$ satisfying q.

Definition 5.4.3. A CSQ $Q(q, \theta)$ consists of query path q and value predicate θ of a leaf attribute $A_i \in J$. Q returns JSON Documents $\{D_1, D_2, ...D_k\}$ satisfying q and θ.

SQ contains a path predicate where the path is expressed by P-C and A-D relationships. SQ comprises simple path query and recursive query as described in Table 5.2. For example, an array query $Q_1 = /A/\#/\mathbf{E}$ where the bold attribute is the target node. E is present in two occurrences of the first document and one occurrence of the second document (refer Figure 5.5). Although the attribute has a different number of occurrences in a document collection, the JSONPath index table assures that the P-C relationship of E is the same for both documents. Therefore, the respective results are returned to the user. Consider a nesting level query $Q_3 = /M/N/P/\mathbf{Q}$. The attribute Q is present in both documents, i.e., $Q \in A_c$, and hence both documents are retrieved as a result.

Algorithm 5.5 shows the pseudocode for evaluating the recursive query. The function *RQuery* evaluates the query path *q* by using the JSONPath index table to determine its children nodes. Furthermore, we need a buffer *bufferP* (initially empty) to hold the path from root to the current node *n*. The algorithm starts with the current node *n* (initially root) and identifies the branches matching *q*. The function *MatchPath* matches the query predicate *q* with the current nodes in *bufferP*. If a branch does not match with *q*, then *q* is an invalid query path; otherwise, *n* in *bufferP* will be evaluated further till it reaches the leaf node. When *n* is a leaf node, the path in *bufferP* is added to *queryList*. Hence, *RQuery* function returns the set of query paths named *qList*. The *qList* returns all mappings such that *n* maps to the start node of all the paths. The search function determines the respective results from *JIndex* and *EJIndex* for every query path in *qList*. Consider a recursive query $Q_5 = /M/\mathbf{O}//$. The attribute O in Q_3 has two different structures, namely the value V_{O_1} and its child node T, as in Figure 5.3. O is a core attribute, and it has different sub-structures. Hence, all the documents in the collection are returned as a result of Q_5. CSQ contains both path and value predicates. Algorithm 5.5 shows the pseudocode for evaluating recursive content and structure queries on *JIndex*. Since *EJIndex* evaluates only the path predicate q, *RCSQ* is not applicable on *EJIndex*. The function *RCSQ* is called with the query path predicate q and value path predicate θ. The complete path of q is obtained from *RQuery* function, and the value predicate θ is matched with the *Search* function. Algorithm 5.4 shows the search function on *JIndex*

Algorithm 5.5: Query Evaluation

```
1  Function RQuery(q, J):
2      n := currentNode ;
3      (Initially, n := root) qList := φ ;
4      match_p := MatchPath(q, J, n) ;
5      if match_p found then
6          bufferP := bufferP.append(n) ;
7          foreach child ∈ n do
8              if n is not a leaf node then
9                  bufferP := bufferP.append(n) ;
10                 RQuery(n) ;
11             end
12             else if n is a leaf node then
13                 qList := qList.append(bufferP) ;
14             end
15         end
16     else
17         return invalid query ;
18     end
19 return qList
20 Function RCSQ(q, θ):
21     qStart := start node of q ;
22     q_1 := RQuery(qStart, J) ;
23     SearchJIndex(q_1, θ, JIndex) ;
24 return W
```

and *EJIndex*. *SearchJIndex* function returns all the *DocIDs* matching the path q if θ is null otherwise the respective *DocIDs* matching θ is retrieved. Since *EJIndex* evaluates only the path predicate q, it returns all the documents belonging to top-k contextually related schemas.

Table 5.2: Query Conditions

No.	Query Type	Query Condition	Representation
QT_1	Simple Path Query	Array	/A#**B**
QT_2	Simple Path Query	Array	/A#**B=”xyz”**
QT_3	Simple Path Query	Nesting Level	/A/B/**C**
QT_4	Simple Path Query	Nesting Level	/A/B/**C=”pqr”**
QT_5	Recursive Query		/A//**C**
QT_6	Recursive Query		/A//**C=”pqr”**

5.5 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of the proposed work in comparison with the existing approaches. We implemented the modules in Python language and Keras framework for implementing deep autoencoder. The results are analyzed with respect to the index size, data retrieval time, and the number of relevant documents retrieved.

5.5.1 Datasets

We use both real-world data used in literature and synthetic dataset (SD) [4] (as given in Section 4.5.1 for publication scenario) for our experiments. In addition, the proposed indexes are evaluated with the Movies and Companies scenario as well. The Movie scenario is considered from freebase and IMDb databases, and the companies scenario is from freebase. Freebase Movies (FM) (Hassanzadeh et al. 2013) contains 84, 530 documents with 60 attributes, and IMDb (Hassanzadeh et al. 2013) contains 1, 37, 978 documents with 41 attributes. Freebase Companies (FC) (Hassanzadeh et al. 2013) contains 74,971 documents with 127 attributes.

Table 5.3: Reduction in number of search keys in *JIndex* using JSONPath index

Dataset	*JIndex*	Existing Approaches		Improvement in Reduction
DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017)	25	SIV	29	+13.8%
		AzureDB	29	+13.8%
SD [4]	194	SIV	195	-
		AzureDB	195	-
FM (Hassanzadeh et al. 2013)	52	SIV	60	+13.3%
		AzureDB	60	+13.3%
IMDb (Hassanzadeh et al. 2013)	39	SIV	41	+4%
		AzureDB	3208	82x
FC (Hassanzadeh et al. 2013)	123	SIV	127	+3%
		AzureDB	3623	29x

[4]https://github.com/umagourish/Synthetic-Datasets.

Table 5.4: Improvement in index size using JSONPath Index

Dataset	*JIndex* (MB)	Existing Approaches (MB)		Performance Improvement
DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017)	139.8	SIV	211.7	+33.9%
		UQP	211.7	+33.9%
SD [4]	13.8	SIV	14.3	+3%
		UQP	14.3	+3%
FM (Hassanzadeh et al. 2013)	58	SIV	72.4	+20.4%
		UQP	72.4	+20.4%
IMDb (Hassanzadeh et al. 2013)	57.6	SIV	72.3	+20.3%
		UQP	85.7	+32.7%
FC (Hassanzadeh et al. 2013)	21.9	SIV	33.1	+33.8%
		UQP	35.7	+38.6%

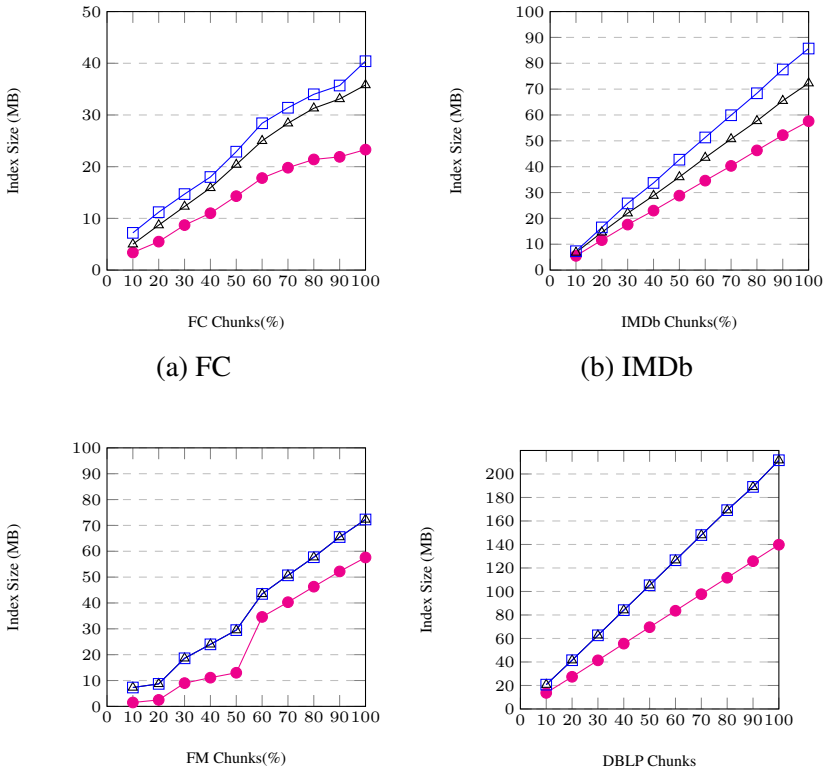
5.5.2 Baseline and Existing Approaches for Comparison

The proposed approach is evaluated by comparing its efficiency and effectiveness with the following approaches:

1. AzureDB (Shukla et al. 2015): The author designed the CAS index for Microsoft Azure's DocumentDB. The path and value are concatenated, and the result is stored in Bw-tree. All the paths in a JSON document collection act as an index key. Hence, AzureDB is considered to compare the number of index keys with the proposed work.

2. UQP (Budiu et al. 2014): UQP designed a summarized index tree for JSON documents. Each node in the index tree maintains an inverted index with the corresponding documents. For simplicity, in this work, UQP constructs a single inverted index by employing the merging techniques followed in Budiu et al. (2014). This assumption is based on the fact that a sum of all inverted indexes is equivalent to the size of a single index if the merging technique is valid. UQP is compared with the proposed approach to measure the index size.

3. Standard Inverted Index (SIV): The proposed *JIndex* without index pruning is



(a) FC (b) IMDb

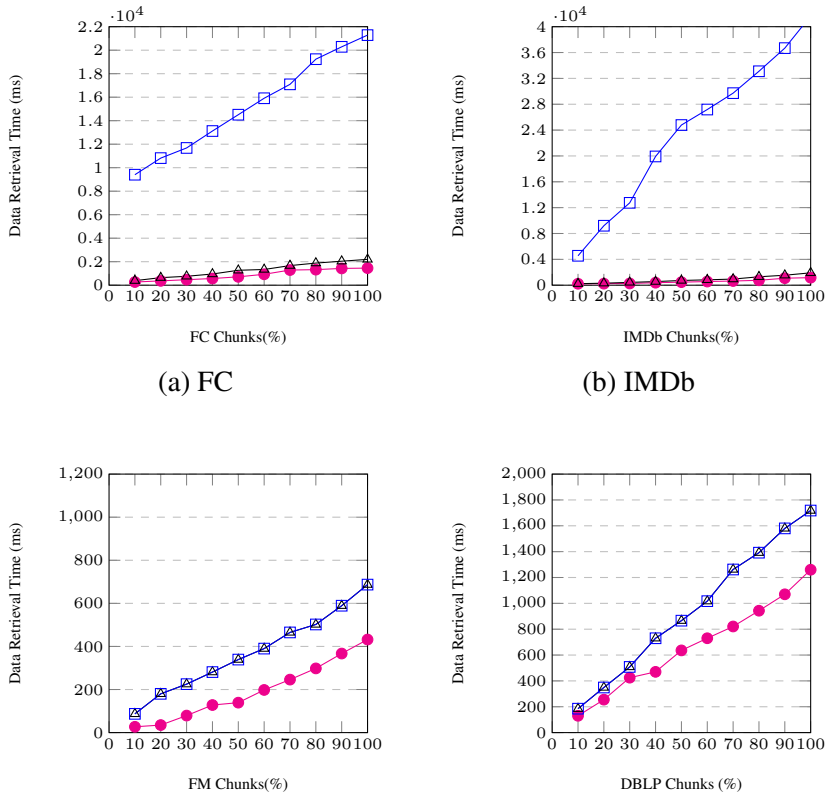
(c) FM (d) DBLP

Figure 5.6: Comparison of *JIndex* (—●—), SIV (—△—), and UQP (—□—) with respect to index size for various dataset chunk sizes

denoted as SIV. SIV is compared with the proposed approach for measuring index size and data retrieval time.

4. SemIndex+ (Tekli et al. 2019): SemIndex+ built an index for structured, unstructured, and partly structured data. This work focuses on SemIndex+ for partially structured data. SemIndex+ is compared with the proposed work to measure semantic relevance.

Efficiency refers to the ability of the approach to improve its performance by means of index size and data retrieval time. On the other hand, effectiveness determines the quality of relevant results for a specific query.



(a) FC (b) IMDb

(c) FM (d) DBLP

Figure 5.7: Comparison of *JIndex* (—●—), SIV (—△—), and UQP (—□—) with respect to data retrieval time for various dataset chunk sizes

5.5.3 Results

5.5.3.1 Evaluation of Efficiency

In this section, we have evaluated the index size required for *JIndex* as well as the time required to process the queries related to each dataset. Tables 5.3, 5.4, and 5.5 and Figures 5.6, and 5.7 illustrate the respective results and are explained briefly in the following sections.

Index Size: Table 5.4 shows the size of *JIndex* and its comparison with the existing approaches such as AzureDB (Shukla et al. 2015) and UQP (Budiu et al. 2014) for the datasets. Figure 5.6 illustrates the evaluation of the relative scalability of *JIndex* with existing approaches. The index structure of AzureDB is (term, posting list), where the term represents the full path of an attribute with value and the posting list contains the *DocIDs* of the term. This maximizes the number of index keys while minimizing the posting list size for each index key. *JIndex's* structure differs because it has fewer index

keys and a larger posting list. AzureDB also encodes path data as bytes and posting lists as bitmaps. To have a fair comparison, the number of index keys in AzureDB is examined. In particular, the path of an attribute without a value predicate is considered.

According to Table 5.4, *JIndex* uses less size than existing approaches and achieves an average performance improvement over existing approaches. The size of the index grows linearly with the size of the dataset, as shown in Figure 5.6. The difference between the size of *JIndex* and UQP increases with the size of the dataset. The experiment is also carried out to determine the reduction in the number of *JIndex* keys using the JSONPath index. Table 5.3 demonstrates the overall progress in reducing the number of index keys. Compared to AzureDB, *JIndex* performed better in datasets with a high number of array occurrences and nesting levels. FC and IMDb have achieved 29 times and 82 times improvement over the existing work. However, other datasets have minor differences in the number of entries comparatively. Although the number of keys differs less between *JIndex* and UQP, the index pruning approach has a significant impact on index size, as seen in Table 5.4. This variation is due to the nature of the datasets. According to Table 5.4, *JIndex* reduces index size by a maximum of 38.6% on diverse datasets.

Data Retrieval Time: To test the performance of *JIndex*, we formulated different query types as shown in Table 5.2. Since the index stores the root-to-leaf path as a search key, it efficiently returns the results in a single lookup. Figure 5.7 plots *JIndex* and existing work data retrieval time-averaged over all queries, considering different chunk sizes of all the datasets. Results show that query execution is linear in all dataset chunk sizes.

The query types mentioned in Table 5.2 has been evaluated for all datasets, and the results are shown in Table 5.5. The table also highlights the performance improvement with SIV. It is noted that the simple queries such as QT_1, QT_2, QT_3, and QT_4 had taken considerably less time than recursive queries such as QT_5 and QT_6. This is due to the fact that recursive queries are converted to a set of simple queries (absolute paths matching the path of the recursive query), and the results of all simple queries must be

Table 5.5: Improvement in data retrieval time for various kinds of queries

Dataset	Query Type	SIV(ms)	JIndex(ms)	Performance Improvement	Average Performance Improvement
DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017)	QT_1	356	200	+44%	+38.3%
	QT_2	370	265	+44%	
	QT_3	314	194	+39%	
	QT_4	398	259	+35%	
	QT_5	642	410	+37%	
	QT_6	723	502	+31%	
SD [4]	QT_1	466	309	+34%	+28%
	QT_2	216	174	+31%	
	QT_3	379	308	+20%	
	QT_4	276	203	+27%	
	QT_5	1756	1539	+23%	
	QT_6	259	176	+33%	
FM (Hassanzadeh et al. 2013)	QT_1	413	159	+72%	+33.75%
	QT_2	214	180	+16%	
	QT_3	326	283	+14%	
	QT_4	199	154	+33%	
	QT_5	NA	NA	NA	
	QT_6	NA	NA	NA	
IMDb (Hassanzadeh et al. 2013)	QT_1	226	147	+35%	+30.3%
	QT_2	276	180	+35%	
	QT_3	213	144	+33%	
	QT_4	294	207	+30%	
	QT_5	854	724	+16%	
	QT_6	246	190	+33%	
FC (Hassanzadeh et al. 2013)	QT_1	218	179	+18%	+34%
	QT_2	214	131	+39%	
	QT_3	212	134	+37%	
	QT_4	245	125	+50%	
	QT_5	1286	1133	+12%	
	QT_6	279	141	+49%	

[NA] Not Applicable

retrieved. Hence, the data retrieval time of recursive queries depends on the number of absolute paths present in an index. A recursive query is not applicable in FM dataset since FM contains arrays at the first level of a document and has a maximum of two nesting levels. The maximum performance on data retrieval time is achieved on DBLP because the number of absolute paths is less for recursive queries through which Q_5 and Q_6 have better performance. Although FC has six levels of depth, the number of index keys is 123, wherein the array of objects is compressed based on its P-C relationship, as in Figure 5.3. This reduces the number of hits in the index, which automatically boosts

the performance of data retrieval time. Similarly, *JIndex* has shown 30% improvement over SIV in the IMDb dataset.

In summary, *JIndex* outperformed the existing approaches in terms of index size and data retrieval time, as shown in Tables 5.4 and 5.5. The increased efficiency of our approach is linear on all datasets; thus, it exhibits robust performance irrespective of the dataset characteristics, size, and query set. The improved efficiency of *JIndex* is due to the fact that the approach does not enforce that all attributes must act as an index key.

5.5.3.2 Evaluation of Effectiveness

The real-time datasets are made up of a certain number of attributes with varying structures. As a result, they are inapplicable for semantic search. *EJIndex* is evaluated for DBLP and SD because they are part of the publication scenario. To perform the nearest neighbor search on our latent embedding space, we employed Faiss (Johnson et al. 2019) for vector quantization and integrated it with an inverted index.

Table 5.6: No. of relevant documents retrieved by JIndex (lexical analysis), EJIndex (semantic analysis), and SemIndex+

Query	JIndex	k=1		k=2		k=3	
		SemIndex+ (Tekli et al. 2019)	EJIndex	SemIndex+ (Tekli et al. 2019)	EJIndex	SemIndex+ (Tekli et al. 2019)	EJIndex
conference name	-	2,576	2,576	6,141	6,141	326	6,467
author	4,04,471	3,565	3,565	6,141	6,141	8,739	1,70,010
year	2,49,187	1,23,240	3,565	1,23,711	2,576	2,598	326

It is important to explore whether the improvement in index size provided by our approach results in poor/better performance when searching for relevant documents. Several researchers in the field of information retrieval have proposed that effectiveness is measured by means of Mean Average Precision (MAP). However, MAP combines precision and recall for ranked retrieval results (Zhang and Zhang 2009). While the indexing approach is responsible for guaranteeing that relevant documents are available for retrieval, this work evaluates the effectiveness by means of the percentage of relevance.

Table 5.6 presents the number of relevant documents retrieved for the sample queries. Since the proposed work focuses on both lexical and semantic analysis, Table 5.6 provides a separate list of documents using *JIndex* and *EJIndex*. The ratio of the posting list, say k, is constant for *EJIndex* and SemIndex+. Furthermore, in order to show the impact of k, the approach is evaluated for different values of k.

EJIndex and SemIndex+ obtain the relevant schemas associated with each query. It is noted from Table 5.6 that *JIndex* does not contain the query attribute *conferencename*, and hence the result is null. The query attribute, on the other hand, has a different structure in the dataset, as shown in Table 5.7. Because *author* and *year* is simple keyword queries in the DBLP dataset, *JIndex* matches exactly and return 4,04,471 and 2,49,187 documents, respectively. When $k = 2$, *EJIndex* and SemIndex+ return the identical documents for *conferencename* and *author*. Whereas when $k = 3$, *EJIndex* returns more documents than SemIndex+. As the proposed retrieval method is based on top-k most similar schemas, the size of the retrieved set is determined by the documents following the specified schemas.

To better understand the behavior of *EJIndex* and SemIndex+, it is vital to analyze not only the number of relevant documents but also the percentage of relevance. For example, for *year*, SemIndex+ retrieves more documents than *EJIndex*. However, it is not determined whether the quantity of documents retrieved is significant. As a result, the percentage of relevance is evaluated by inspecting the context of queries and schemas, as shown in Table 5.7. The top three semantically related schemas for *EJIndex* and SemIndex+ are shown in Table 5.7.

The SD has been populated with 29 schema variants and 13 classes. Thus, for each query, the relevant schemas are mapped with the classes to measure the relevance. For instance, the query *conferencename* belongs to the context *Conference*, so the retrieved schemas must match the same context, which SemIndex+ fails to do for $k = 3$. However, the queries *author* and *year* are simple keyword queries that belong to all contexts. In this case, the approaches are evaluated based on the presence of contextually similar attributes in each relevant schema. For *author*, SemIndex+ returns an exact match rather

Table 5.7: Representative top-3 relevant schemas retrieved by *EJIndex* and SemIndex+

Query	SemIndex+ (Tekli et al. 2019)		*EJIndex*	
	Schema	Context	Schema	Context
conferencename	1. article/title article/authors/author article/conference/name article/conference/year	Conference	1. article/title article/author article/conferencename article/year	Conference
	2. article/title article/author article/conferencename article/year	Conference	2. article/title article/authors/author article/conference/name article/conference/year	Conference
	3. misc/author misc/title misc/howpublished misc/month misc/year misc/note	Miscella-neous	3. proceedings/editor# proceedings/title proceedings/year proceedings/publisher proceedings/address proceedings/isbn proceedings/abstract proceedings/location	Conference
author	1. article/title article/author article/conferencename article/year	Conference	1. article/title article/author article/conferencename article/year	Conference
	2. _id/$oid mdate author _key title type	General	2. article/title article/authors/author article/conference/name article/conference/year	Conference
	3. paper/headline paper/reporter paper/company paper/date	News Article	3. _id/$oid mdate author _key title type	General
year	1. _id/$oid mdate author ee booktitle title pages url year type _key crossref#	Conference	1. article/title article/author article/conferencename article/year	Conference
	2. _id/$oid ee number year mdate author# type journal volume pages url title _key	Journal	2. article/title article/authors/author article/conference/name article/conference/year	Conference
	3. _id/$oid author booktitle cdrom cite crossref editor ee isbn journal mdate month note number pages publisher publtype school series title type url volume year	Journal	3. misc/author misc/title misc/howpublished misc/month misc/year misc/note	Miscella-neous

than a semantic match. However, *EJIndex* returns contextually relevant schemas. For the *year* query, SemIndex+ returns all exact matches, which *JIndex* already does. Therefore, *EJIndex* retrieves relevant schemas irrespective of the query structure and schemas.

5.5.4 Discussion

5.5.4.1 JIndex

Considering the scalability of the index size for JSON datasets of varying sizes, the results demonstrate that the current approaches utilized more space and had lower scalability than *JIndex*. Since JSON documents are varied by structure in real-time, we analyzed the proposed work based on the nature of the datasets, as below:

1. **JSON Documents with $\leq$2 level:** DBLP, SD, and FM datasets have two levels of attributes. In particular, all the attributes in the FM dataset belong to an array of strings or numbers data types. Hence, both SIV and UQP have an equal number of index keys. However, *JIndex* has less number of index keys compared to existing approaches. In addition, Table 5.4 demonstrates that even a slight variance in index keys results in a big difference in index size. This is due to the fact that the attributes pruned by *JIndex* are core attributes that often take up a significant amount of space in all the datasets. Similarly, the approaches yield a good result on index size for DBLP and SD datasets. The impact of index size is reflected in data retrieval time as well. The results from Table 5.5 show that the data retrieval time for three datasets yielded a maximum of 38% improvement in performance.

2. **JSON Documents with $>$2 level:** *JIndex* outperformed all other existing approaches for datasets with more than two nesting levels (IMDb and FC). This is owing to the fact that large levels have many arrays of object data types which introduce redundant paths. From Table 5.3, the results of IMDB and FC datasets for *JIndex* and AzureDB show the impact of the index pruning technique and array compression followed by *JIndex*. *JIndex* achieved a maximum of 82 times AzureDB's index keys. The redundant paths are compressed efficiently by *JIndex*, and specifically, the redundant core attributes are pruned from *JIndex*. As a result, *JIndex* can have fewer index keys and a smaller index size without sacrific-

ing any data from datasets. AzureDB consumed an ample size compared to other approaches. This is because it stores all occurrences of an array as a separate index key, which has redundant paths in the index key. For instance, in Figure 5.2, even though the 0^{th} occurrence of attribute a in two documents has the same structure, AzureDB has four index keys due to the presence of different values. This increases the redundancy in paths which increases the index size. However, *JIndex* has two index keys holding the values of two documents and hence achieved better compression in the index. Regarding the data retrieval time, the performance improvement of *JIndex* is similar to the above case. One may think that the fewer index keys and index size may improve data retrieval time compared to the above case. This case is predicted based on the nature of the dataset, i.e., the large number of array of objects has a high posting list for an index key that needs to be retrieved for all the queries. Hence, the data retrieval time is based on the nature of the dataset.

Although there is a trade-off between data retrieval time and index size, *JIndex* achieves at least a 28% improvement in query response time for all kinds of datasets compared to existing approaches. By analyzing the index size from Table 5.4, it is observed that *JIndex* works exceptionally well on datasets with more than two nesting levels.

5.5.4.2 EJIndex

The data in Table 5.7 allow for numerous observations. The first observation reveals that both proposed and existing works obtain a sufficient number of relevant documents for all five document collections. Another observation is that even though both works yield a similar percentage of relevant documents, their effectiveness varies depending on the context. For instance, consider the *conferencename* query; *EJIndex* retrieves documents related to the conference irrespective of its exact term. However, the schema belonging to *Miscellaneous* context returned by SemIndex+ is not relevant to the query. While both works return a similar number of relevant documents for each query, when k grows larger, the received documents are not always overlapping and become complementary.

In summary, results demonstrate that *JIndex* and *EJIndex* serve the purpose of efficient JSON data retrieval by leveraging both lexical and semantic matching of queries with reduced index size and data retrieval time. The proposed indexing scheme can be used in NoSQL distributed environment where the common and schema-specific attributes in each shard help to reduce the index size at each shard. The only problem in the proposed scheme is that when the query is CSQ and a core attribute, then the approach does a sequential scan to retrieve the respective documents. However, this kind of query type is not frequent in real-time, and the number of core attributes is usually less. Therefore, the proposed indexing scheme would take less time for retrieval. However, this is the primary limitation of the approach, which will be addressed in the future.

5.6 SUMMARY

This chapter proposed a compact schema-aware indexing scheme for JSON document collection that supports both lexical and semantic matching of JSON path-based queries. To the best of our knowledge, this is the first work of its kind to capture contextual information in JSON schemas and design an embeddings-based index for JSON documents. In addition, we introduced the *JDewey* labeling scheme to preserve the structural relationship of the attributes. The experimental results of the proposed work confirm that the index size and data retrieval time are linear with respect to the document collection size and more effective in retrieving contextually relevant documents.

CHAPTER 6

INCREMENTAL APPROACH FOR HANDLING DYNAMIC JSON DATA

6.1 INTRODUCTION

The widespread use of IoT systems and modern real-time applications in today's world generates massive JSON data dynamically. As the data changes, the implicit schemas associated with the data also change. The process of updating the schema of the document in a collection from one version to another, often by adding, removing, or modifying the attributes, is referred to as schema evolution. Any change in the JSON documents can be addressed in the following two ways: either by schema versions of the documents or by updating the old schema variants of the modified documents.

Most literature focuses on extracting the schema versions from a collection using schema class types (entities) manually embedded in the documents. These approaches maintain schema versions in data lakes efficiently for the updated document versions (Klettke et al. 2017, 2016; Scherzinger et al. 2013). Due to the dynamic nature and sheer size of JSON documents, the manual embedding of class types in each document is not feasible in real-time. Therefore, there is a need to automatically identify the class types of new schema variants with the help of the previous class knowledge. This work employs an *Incremental embedding-based clustering* approach to handle the dynamic data.

The modified documents are clustered based on the contextual similarity of the respective schemas. The *Incremental SchemaEmbed* model computes the contextual sim-

ilarity of the modified JSON schemas based on the class-incremental scenario where the classes for the new and modified documents are identified by preserving the old classes, i.e., new documents are efficiently organized into the clusters based on their similarity with the old documents. At each cluster, the *SVTree* is updated incrementally by adding new schema variants and modifying existing schema variants.

When documents evolve, it is necessary to provide the latest documents for user queries. To address this issue, incrementally updating the indexes is essential. Most existing works of JSON indexing techniques are well-suited for the static environment (Budiu et al. 2014; Shang et al. 2021; Shukla et al. 2015), and they lack in providing dynamic support. Although few studies exist in XML indexing techniques (Hsu and Liao 2020; Wellenzohn et al. 2020), they suffer from huge index sizes and high data retrieval time. Hence, in this work, an updatable compact indexing scheme is developed based on *SVTree* for efficient updates of JSON documents.

The major contributions of this work include the following:

1. Proposing an *Incremental SchemaEmbed* model to generate the schema embeddings for new data using the pre-trained *SchemaEmbed* model.

2. Proposing an *Incremental embedding-based clustering* approach to cluster the new documents based on the previous clusters.

3. Designing an approach for updating the *SVTree, JIndex*, and *EJIndex* to reflect the updates in the data.

4. Evaluating the proposed approach with baseline approaches on real and synthetic datasets.

The rest of this chapter is structured as follows. Section 6.2 describes the problem statement. Section 6.3 presents the incremental schema variants extraction, and Section 6.4 presents the incremental indexing in detail. Section 6.5 presents the experimental study and performance analysis. Finally, Section 6.6 presents a summary of the chapter.

6.2 PROBLEM DESCRIPTION

This section presents the design of our approach to update schema variants and indexes incrementally with respect to updated document collection. Consider an updated JSON document collection $G' = \{D'_i\}_{i=1}^{x}$ and $D'_i = \{A'_{j,s}\}_{s=1}^{t_j}$ where $A'_{j,s}$ represents the attribute A'_s in document j, t_j represents the number of attributes in a document j and x denotes the size of the collection, the proposed work aims to develop an *Incremental embedding-based clustering* approach that extracts different JSON schemas in the collection G' and analyses the contextual similarity among them to group the similar JSON documents based on the old K clusters i.e., the contextual similarity of G' is compared with the K clusters. The output is an updated *SVTree* that represents the summarization of updated schema variants available at each cluster C_i along with its core and schema-specific attributes. In addition, *JIndex* and *EJIndex* are updated incrementally to support the retrieval of the latest documents for user queries.

The proposed approach is divided into two phases such as (i) Incremental Schema Variants Identification and (ii) Incremental Schema-Aware Indexing. The description of each phase is given in the following sections.

6.3 INCREMENTAL SCHEMA VARIANTS EXTRACTION

The process of schema variants identification works in two steps: (i) *Incremental SchemaEmbed* model and (iii) *Incremental SVTree* construction.

6.3.1 Incremental SchemaEmbed model

Numerous deep learning algorithms succeed based on the following two assumptions: i) training data is identical and independently distributed, and ii) training data labels are available. The former is uncommon if new data arrives sequentially over time. Therefore, continuous learning has been offered as a solution to the former issue of incrementally learning new tasks without forgetting previous information. The latter requires additional human annotation and can be noisy. Therefore, unsupervised learning addresses the latter by directly learning the latent representations for downstream tasks from unlabeled data. However, unsupervised continuous learning, which is sup-

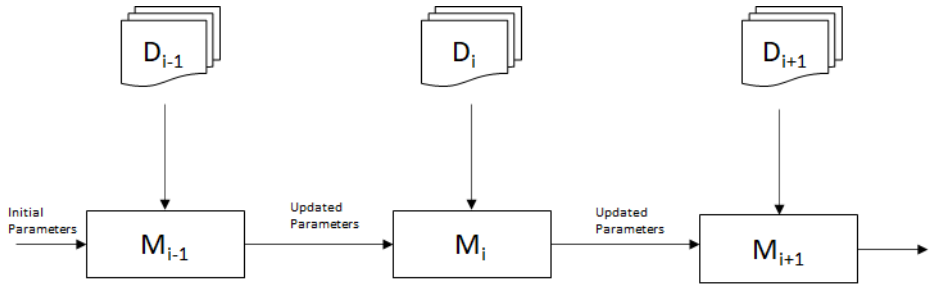


Figure 6.1: *Incremental SchemaEmbed* Learning Architecture

posed to address both of the aforementioned difficulties, has received limited research. Hence, in this work, we examine a novel challenge for continuous learning that involves pseudo-label fusion at a later stage rather than human annotations, which is crucial for real-world applications.

As documents evolve, the model must learn them to acquire new knowledge. However, the model forgets what it has learned before. This is called catastrophic forgetting. To alleviate this issue, the *Incremental SchemaEmbed* model M_i in this work uses the old parameters of a *SchemaEmbed* model M_{i-1} to preserve the old knowledge. Figure 6.1 shows the flow description of the proposed incremental model.

To define formally, the incremental learning problem T can be expressed as learning a sequence of N tasks $T^1, T^2, ..., T^N$ corresponding to $N-1$ incremental steps. Each task $T^i \in T$ contains P non-overlapped classes to learn. In this work, we explore class-incremental learning in an unsupervised environment starting from task T^2 for incremental steps, assuming that the initial model was trained on T^1. Let $D_1, ..., D_N$ be the training data associated with N tasks, and D_i represent the data associated with T^i. P^i refers to the number of newly added classes for task $T^i \in T$, which is also defined as incremental step size.

6.3.1.1 Pre-Training Phase

The pre-training phase allows us to learn the contextual similarity of JSON schemas for a given JSON collection G. The initial non-incremental task T^1 is trained with data G_1 on model M_1. The documents belonging to initial schema embeddings are clustered with the spherical clustering algorithm, and the initial classes P^1 for G_1 were identified

in Section 4.3.

6.3.1.2 Incremental Training Phase

As an initial step, the schemas for all the new documents are extracted. Given a collection of JSON documents represented as $G' = \{D'_1, D'_2, ..., D'_x\}$ where D_i comprises of a set of flattened JSON attributes $\{A'_1, A'_2, ..., A'_y\}$, the trigrams $t'_i = \{t'_{i_1}, t'_{i_2}, ..., t'_{i_x}\}$ for each D'_i where $t'_{i_i} = (A'_p, A'_q, A'_r)$ are extracted. The trigram set T' comprises all the trigrams extracted from all new schemas.

As a second step, the trigrams T' are given as input to the *Incremental SchemaEmbed* model. At each task T^i, where $i \in \{2, 3, ..., N\}$, the objective is to learn a model M_i at each incremental step which recognizes $N^t = P^2 + P^3 + ... + P^t$ classes. The model M_i is trained using the previous model M_{i-1} using the pre-trained weights of M_{i-1} except the last layer of the fully connected deep autoencoder. The model M_{i-1} is trained with dataset G and M_i is trained with G'. Algorithm 6.1 shows the flow of the proposed incremental learning of schemas for the new documents.

Algorithm 6.1: Incremental embedding-based clustering algorithm

Input: A sequence of N tasks $T^1, T^2, ..., T^N$, Initial Model M_1, JSON Document Collection $G_1, G_2, ..., G_N$
Output: Updated Model M_N

$P^1 = \mid T^1 \mid_{class}$;
for $i \leftarrow 2$ **to** N **do**
  $G_i \leftarrow \{x_1, x_2, ..., x_{n_i}\}$;
  $p_{ae} \leftarrow p_{i-1}$;
  $M_i, E_i = Incremental\_Learning(G_i, p_{ae})$;
  $\{c_1, c_2, ..., c_k\} = \Theta(E_i, P^{i-1})$;
  $P^i = \mid T^i \mid_{class}$;
end

The fully connected and sigmoid layers are used as they were used in *SchemaEmbed* model. The weights of *SchemaEmbed* model are carried to the *Incremental SchemaEmbed* model. The binary cross entropy is used as a loss function. The proposed network is trained using a backpropagation algorithm based on the values of the loss function obtained through successive iterations.

6.3.2 Incremental Embedding-based Clustering

We apply a clustering algorithm based on schema embeddings for new documents to generate cluster assignments. Firstly, this algorithm classifies the new documents by comparing the similarity of old and new embeddings instead of relying on new embeddings alone. Secondly, the labels of old and new data are fused and passed to a new set of data.

The *Incremental SchemaEmbed* model produces a set of schema embeddings E_i for every batch of new documents which are clustered with the Mini-batch K-means clustering algorithm (Sculley 2010). This clustering algorithm is used because it utilizes small random batches of fixed-size vectors to store them easily in memory. Whenever new documents arrive or the existing document changes, a fresh sample of schema embeddings is obtained from the model. Then, the nearest cluster center for the new documents is calculated with the objective function f(C,d). The objective function determines the distance between each point in 'N' samples and 'K' centers, and then the document is assigned to the closest center. The count for each center is updated with the vectors, and the learning rate is determined as the inverse of the number of samples assigned to a cluster during the process. The applied learning rate decreases with the number of iterations. The clusters are updated, and the process is repeated until the clusters converge. An increase in the count of iterations reduces the outcome of recent examples so that convergence can be noticed in the existing condition without any modifications in the clusters. Moreover, the perfect balance of accuracy and computation time is taken care of, which makes real-time clustering efficient.

6.3.3 Incremental SVTree

SVTree SVT for document collection G is constructed with the help of core and schema-specific attributes A_c and A_s identified in the schema restructuring phase of Section 4.4.1. When the set of new documents G' arrives with schemas S', the old schemas S and new schemas S' may share some attributes if they belong to the same dataset scenario.

The description of adding the updated schemas S' in an already existing *SVTree* is

explained below:

1. **Insertion of new schemas:** When a set of new schemas are to be inserted, the existing *SVTree* is updated by accommodating the updated node information of new schemas. The inclusion of schemas to *SVTree* involves the following steps, and its pseudocode is given in Algorithm 6.2:

   (a) The core attributes A'_c = $A_c \cap S$ and the schema-specific attributes $A'_s = S' \setminus A'_c$.

   (b) If $A'_c = A_c$, then sort A'_s according to its IDF value and check if the same sequence is already present in the *SVTree*. If so, increment the $SVNum$ of the specific branch. Otherwise, insert them as a new branch in *SVTree*.

   (c) If $A'_c < A_c$ then change the elements of radix node from A_c to A'_c. Then the respective A_s are changed to A'_s, which should be reflected in the *SVTree* as well.

   (d) If $A'_c > A_c$ then $A'_c = A_c$ and $A'_s = S' \setminus A'_c$ and then follow the step (b).

2. **Deletion of existing schemas:** To delete a set of schemas, the user is required to provide the list of document identities so that the respective schemas will be deleted from the *SVTree*. The given *DocIDs* is removed from *DocIDList* of the leaf node. If the size of *DocIDList* becomes zero after deleting the respective *DocIDs*, then the nodes containing the specific schema variant are deleted from *SVTree*. However, this operation does not affect the radix node containing core attributes. Therefore, deletion of a schema variant from a tree reduces the number of nodes describing the schema-specific attributes.

3. **Modification of existing schemas:** While performing modifying operations on any existing document, it is highly desirable to incorporate only the updated part of the respective schemas in an existing *SVTree* instead of regenerating the entire tree. For instance, when a document is modified with new attributes, then it is required to incorporate the new updates without reconstructing the *SVTree*. The process of *SVTree* updation involves deleting the old information and adding the

Algorithm 6.2: SVTree Updation

Input: SVTree SVT, JSON Documents $D'_1, D'_2, ..., D'_x$, Schema Variants $S'\}$
Output: Updated SVTree SVT'

```
1  A'_c = A_c ∩ S';
2  foreach S_i ∈ S do
3  |  A'_{s_i} = S_i \ A'_c;
4  end
5  if A'_c < A_c then
6  |  SVT.radixNode := A'_c;
7  |  newNode := SVT.addNode(A_c \ A'_c);
8  |  V := V ∪ newNode;
9  |  newEdge := SVT.addEdge(SVT.radixNode, newNode);
10 |  E := E ∪ newEdge;
11 |  foreach childnode in SVT.radixNode do
12 |  |  newEdge := SVT.addEdge(childnode, SVT.newNode);
13 |  |  E := E ∪ newEdge;
14 |  end
15 else
16 |  foreach S_i ∈ S' do
17 |  |  traverse SVT till SVT.radixNode;
18 |  |  SVT.currentNode := SVT.radixNode;
19 |  |  SVT.currentNode := SVT.nextNode;
20 |  |  SVT' :=
   |  |   InsertSS(A'_{s_i}, S_i, SVT, SVT.currentNode, SVT.radixNode);
21 |  end
22 end
```

new information in the respective nodes of *SVTree*. Deleting any attributes in a given schema variant S_a of a document D_a is handled in the following fashion.

(a) If S_a becomes S_b after deleting some attributes where $S_b \in S$, then the respective nodes in *SVTree* is updated, i.e., the *DocIDList* of a leaf node containing S_b is added with D_a. Similarly, the *DocIDList* of a leaf node containing S_a is updated by deleting D_a. This situation arises because S_b is a subset of S_a.

(b) If $S_b \notin S$, then S_b is a new schema variant for *SVTree*. Therefore, the *DocIDList* of a leaf node containing S_b is updated by deleting D_a. Similarly, the new schema variant is added to *SVTree* as in 1.

6.4 INCREMENTAL JINDEX AND EJINDEX

The research community has paid less attention to supporting dynamic operations on JSON indexes. When the original document is modified, expensive index rebuilding or identifier reassigning is required. Thus, there is a need for an efficient indexing scheme to provide concurrent updates of JSON documents. The proposed *JIndex* and *EJIndex* are constructed based on the schema-specific attributes in a cluster. The indexes are updated based on the changes in *SVTree*. The new branch in *SVTree* corresponds to adding new index keys or updating the posting list of existing index keys. *JIndex* and *EJIndex* require the same number of keys as the number of updated schema-specific attributes in a cluster. Thus, the proposed *JIndex* and *EJIndex* require less number of index keys, resulting in a reduction in index size as well as reduced data retrieval time while supporting dynamic operations efficiently. All update operations are directly executed without index rebuilding. The insertion and deletion of documents are handled by updating the posting list of the respective attributes. The detailed approaches are explained as follows:

The structure of *JIndex*, *EJIndex* and insertion of keys have been described in Section 5.4. This section describes dynamic insertion, deletion, and modifications to *JIndex* and *EJIndex* in different cases as below:

1. **Insertion of new documents:**

   (a) *Inserting documents with already existing keys:* If the new document D_a arrives with already existing keys say case (b) of insertion in Section 6.3.3, then D_a is appended to the posting list of the attributes A_a (index keys) in the specific schema variant.

   (b) *Inserting documents with new keys:* If the new document D_a arrives with new keys, then the respective keys are added to the JSONPath index. In addition, the keys are created in *JIndex*, and *EJIndex*. The respective posting lists are created with D_a.

2. **Document Deletion:** The user provides the *DocID* to be deleted from the index. The A_c and A_s are obtained from *SVTree*. Therefore, *JIndex* searches for the

index keys (A_s), and then the entry *DocID* and its value will be removed from the respective posting list. Similarly, in *EJIndex*, the *DocID* will be removed from the respective posting list. Algorithm 6.3 shows the procedure of deletion operation in *JIndex* and *EJIndex*.

3. **Modification of existing documents:** Updating the index for the existing documents involves deletion of old attributes and insertion of new attributes. Therefore, *JIndex* searches for the attributes to be deleted using A_s, and then the given *DocID* and its value will be removed from the respective posting list. As a next step, the new attributes to be added follow the step 1. Similarly, in *EJIndex*, the *DocID* will be updated in the respective posting lists.

Algorithm 6.3: JIndex and EJIndex Deletion

Input: JIndex, EJIndex, D_a
Output: Updated JIndex and EJIndex

```
1 foreach D_{a_a} ∈ D_a do
2     foreach K ∈ D_{a_a} do
3         parse the flattened D_{a_a};
4         if K.A_p ∈ A_s then
5             JIndex[K.A_p].postingList := JIndex[K.A_p].postingList \ {D_a, K.A_pv};
6             EJIndex[K.A_p].postingList := EJIndex[K.A_p].postingList \ D_a;
7         end
8     end
9 end
```

6.5 EXPERIMENTAL EVALUATION

This section presents the experimental results of the proposed approach with existing approaches.

6.5.1 Datasets

We use real and synthetic datasets for conducting our experiments as described in Section 4.5.1. In order to evaluate the proposed approach, the original dataset has been updated as follows:

- As per Section 4.5.1, there are 39 schema variants in SD [1] with 77 unique at-

[1] https://github.com/umagourish/Synthetic-Datasets

tributes in a collection. Five schema variants have been newly added, with ten new attributes. Therefore, 15,045 documents have been added to evaluate the incremental approach.

- For real datasets such as DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017), IMDb (Hassanzadeh et al. 2013), FC (Hassanzadeh et al. 2013), and FM (Hassanzadeh et al. 2013), new documents (10% of the old dataset) are added to the old dataset.

6.5.2 Baseline and Existing Approaches for Comparison

The proposed model is compared with the following two baseline models. The baselines and proposed model adapt the same architecture f. However, the training procedures of f differ among the baselines.

1. **Weight Initialization (Init) model:** The *Init* model uses the parameters of the previous model and trains the new data. For instance, Let M_{i-1} and M_i be the models trained sequentially using the deep autoencoder architecture f. The parameters of f for training on M_i are initialized with the parameters of f trained on M_{i-1}.

2. **Combined Training (Comb) model:** In this baseline, the data from old and new classes are combined to train the autoencoder f. The last hidden layer of the encoder is dropped from model M_{i-1}, and the mean of parameters used in the previous layers are calculated for the new additional layer in model M_i.

6.5.3 Results and Discussion

This section presents the experimental results of the proposed work using real and synthetic datasets.

6.5.3.1 Incremental Embedding-based Clustering

The quality of clustering new documents based on their schema embeddings is evaluated using NMI, AMI, and ARI.

Table 6.1: Efficiency of incremental clustering approach

Evaluation Metrics	Comb	Init	Proposed Work
SC	0.61	0.56	0.67
NMI	0.7	0.68	0.74
AMI	0.68	0.66	0.70
ARI	0.41	0.39	0.59

Experimental Setup: Let the schema variant S is denoted by a sequence of attributes $S = \{A_1, A_2, ..., A_n\}$ where n is the number of schema variants. Attributes present in the vocabulary W are initialized to the corresponding attribute embeddings. However, the attributes present in new documents may not be present in the vocabulary list, and hence the vocabulary list is updated for each and every new document. These attributes are updated to generate the final attribute embeddings.

The proposed model focuses on updating the parameters of decoding layers. In this work, the last layer of the autoencoder is dropped from model M_{i-1}, and the mean of parameters used in the previous layers are calculated for the new layer in model M_i. After updating the model with pre-trained weights, the final decoder layer with its input size is added at the end of the deep autoencoder to reduce the reconstruction loss.

Results: Table 6.1 presents the experimental results on the updated datasets. It is observed from Table 6.1 that the proposed model has shown better results than baseline models, and the values have an upward trend. The proposed approach achieves an NMI score equal to 74%, which is an improvement of an absolute 5.7% and 8.8% compared to the baseline models *Comb* and *Init*, respectively. Furthermore, the AMI and ARI scores have shown similar improvement over the baselines. The SC value of 0.67 has shown that the inter-similarity between the clusters is less, and hence the clusters are well formed.

The performance of the *Init* model is less because it only works on static datasets and is not capable of learning new knowledge incrementally. Furthermore, it suffers

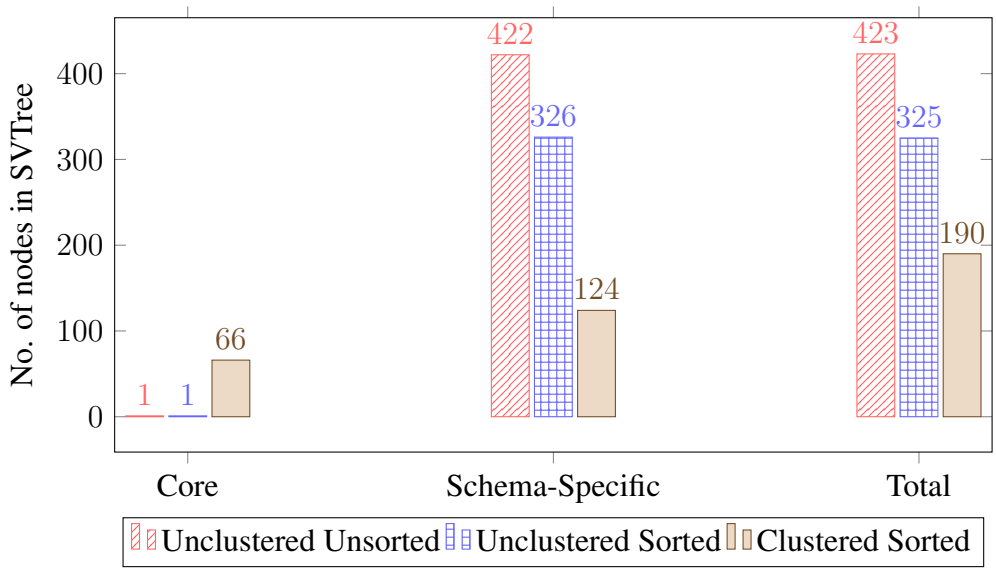


Figure 6.2: Efficiency of *Incremental SVTree*

from catastrophic forgetting, i.e., the knowledge about previous data is not carried to train the new data. The *Comb* model directly deals with the encoder layer, which lacks focus on controlling the whole autoencoder. Although it fine-tunes the encoder layer, it fails to consider the fully connected autoencoder. The proposed work focuses on controlling the whole autoencoder by fine-tuning the decoder rather than the encoder layers.

While comparing the results of the proposed *Incremental embedding-based clustering* approach with the baseline *embedding-based clustering* approach (ref. Chapter 4), both approaches yield similar results for all the evaluation measures we have considered. However, it is observed that the incremental clustering approach has updated the clusters for new and modified documents with reasonable SC values and is able to achieve good NMI, AMI, and ARI scores effectively.

6.5.3.2 Incremental SVTree

Figure 6.2 depicts the efficiency of the *Incremental SVTree* after updating the old *SVTree* as discussed in Section 4.5.4.3. Based on the new schema variants at each cluster, it is observed from the results that the number of common attributes is increased from 65 to 66, and the respective schema-specific attributes have increased from 186 to 190. However, there is a large difference in the unclustered data, i.e., the number of schema-

specific attributes has been increased from 297 to 326. Similarly, the unclustered and unsorted tree yields a difference of 30 nodes. By comparing the results, the number of nodes in both baselines is almost the same, whereas the proposed work shows a drastic difference in the number of nodes. Therefore, *Incremental SVTree* contributes to providing the summarization of schema variants efficiently even after updating the database.

6.5.3.3 Incremental JIndex

Since the research on JSON dynamic indexing techniques is still in the preliminary stage, the proposed indexes are compared with the baseline inverted index rebuild from scratch. Each update process begins by searching for the target attributes and then inserting or deleting them. The proposed approach is evaluated by index update time and index size.

Table 6.2: Improvement in Index Size using JSONPath index

Dataset	JIndex (MB)	SIV (MB)	Performance Improvement
DBLP (Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal 2017)	153.5	253.2	+39.3%
SD [1]	16.2	17.3	+6%
FM (Hassanzadeh et al. 2013)	59.1	90.9	+34.9%
IMDb (Hassanzadeh et al. 2013)	63.1	87.1	+27.5%
FC (Hassanzadeh et al. 2013)	26.7	44.5	+40%

Index Size: Table 6.2 shows the updated index size of *JIndex* and SIV after adding new documents to the old *JIndex* and SIV. It is observed from the results that although the number of core attributes is less in the new documents, *JIndex* achieved a minimum of 6% over SIV for SD and a maximum of 40% improvement for the FC dataset over SIV. The major difference between the minimum and maximum improvement of *JIndex* occurs due to the nature of the datasets. SD has only one core attribute, and hence *JIndex*

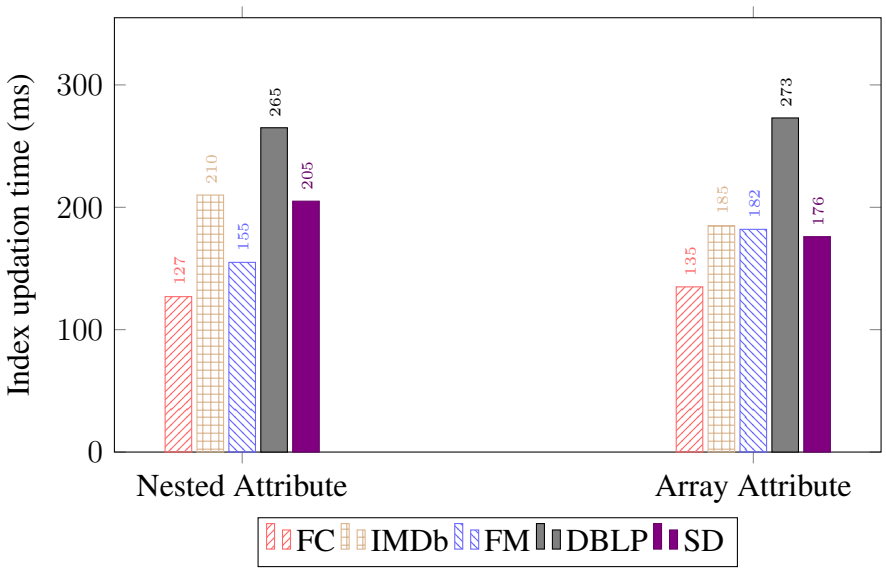


Figure 6.3: Index updation time

achieved less improvement over SIV. However, FC has a nesting depth of 7, and hence the number of core attributes is comparatively high. Therefore, the proposed approach was able to show better improvement.

Index Updation Time: To evaluate the update performance, the varying number of core and schema-specific attributes are fed into the indexes. The proposed indexes involve modification of the existing posting list for already existing attributes and adding a new posting list for new attributes. While index rebuild creates new keys for all core and schema-specific attributes, the index rebuild time and data retrieval time are high for each update operation. In contrast, *JIndex* and *EJIndex* focus on updating only schema-specific attributes through which the index update time is reduced.

The results of index updation time are analyzed for different data types such as array attribute and nesting objects as illustrated in Figure 6.3. It is noted from the results that the array attribute considerably consumes more time than the nested attribute. This is because the array attribute has more than one value, and all the values must be removed before inserting the new values. The performance of the proposed work also depends on the nature of the datasets, where long documents consume more time than small documents in a large collection.

Table 6.3: Representative top-3 relevant schemas retrieved by *EJIndex* and *Incremental EJIndex*

Query	EJIndex		*Incremental EJIndex*	
	Schema	Context	Schema	Context
headline	1. paper/headline paper/reporter paper/company paper/date	News Article	1. paper/headline paper/reporter paper/company paper/date	News Article
	2. article/title article/author article/conferencename article/year	Conference	2. paper/author paper/article_type paper/article_name paper/date_of_publication paper/pages	News Article
	3. article/title article/authors/author article/conference/name article/conference/year	Conference	3. article/title article/author article/conferencename article/year	Conference
monthfiled	1. patent author title language assignee address nationality type number day dayfiled month monthfiled year_or_year_filed note url	Patent	1. patent author title language assignee address nationality type number day dayfiled month monthfiled year_or_year_filed note url	Patent
	2. patent author title language assignee address nationality type number day_filed month_filed year_filed note url	Patent	2. patent author title language assignee address nationality type number day_filed month_filed year_filed note url	Patent
	3. paper headline reporter company date	News Article	3. patent title assignee number datefiled date_issued	Patent

6.5.3.4 Incremental EJIndex

The research on dynamic semantic indexing techniques for JSON documents is still in the preliminary stage. Therefore, the proposed *Incremental EJIndex* is compared with *EJIndex* to evaluate whether the incremental approach is able to identify the new schemas for the given queries. The top three semantically related schemas identified before (*EJIndex*) and after (*Incremental EJIndex*) adding new documents are tabulated in Table 6.3. For the query *headline*, *EJIndex* has identified the *news article* context correctly, and the other two schemas are related by the meaning. The schema *paper/author paper/article_type paper/article_name paper/date_of_publication paper/pages* belongs to the context *News Article* was added to the collection. Similarly, for the query *month-filed*, the *Incremental EJIndex* is able to identify the newly added schema for the context *Patent*. The respective relevant documents will be retrieved for the given queries in the same way as in Section 5.5.3.2. It is evident from the results that *Incremental EJIndex* identifies the new schema correctly. Experiments reveal that the proposed *Incremental EJIndex* is able to identify the context during schema evolution.

6.6 SUMMARY

In this chapter, we have presented an approach for updating the new documents to the respective clusters using the *Incremental SchemaEmbed model* and *Incremental embedding-based clustering approach*. The model used a late fusion of classes to determine the cluster centers for new documents. The core and schema-specific attributes in the *SVTree* are updated to reflect the new schema variants efficiently. To provide latest documents to user queries, the index structures such as *JIndex* and *EJIndex* are updated incrementally. Experimental results show that the proposed model determines the clusters efficiently, and the proposed indexes outperform baseline indexes in reducing the index size and index updation time. It is also evident from the results that the proposed approach incorporates the updates efficiently in already existing indexes.

CHAPTER 7

CONCLUSIONS AND FUTURE SCOPE

This dissertation has focused on extracting schema variants from JSON collections and constructing compact indexes for efficient retrieval of dynamic JSON data. This is achieved in three facets. Firstly, we proposed *Embedding-based clustering* approach using *SchemaEmbed* model to cluster the contextually relevant JSON documents and *SVTree* to represent the schema variants. As the schema variants capture the different sets of attributes present in a collection, it supports various applications such as big data analytics, distributed query decomposition, query optimization, etc. The proposed clustering approach has been tested with real and synthetic datasets. The experimental results substantiate the effectiveness of the proposed approach compared to existing approaches. The results of *SVTree* confirm that the proposed work is able to identify the core and schema-specific attributes, which will eventually be used for efficient data retrieval. While the proposed work concentrated on generating the schema variants and cluster-friendly data representation, such as schema embeddings, there is a scope for optimizing the clustering algorithm through which we can achieve better cluster assignment and schema embeddings.

Secondly, in order to support efficient retrieval of JSON data, we proposed compact indexes such as *JIndex* and *EJIndex* to answer JSON path-based queries. Using the schema variants extracted, the proposed indexes facilitate efficient data retrieval for structure-based and structure and content-based queries. An in-depth experimental analysis demonstrated the efficiency of the proposed indexes by achieving less index

size and data retrieval time. The results also demonstrate that simple path-based queries take less retrieval time than recursive queries because all the child nodes of the query attribute must be traversed to retrieve results for recursive queries.

Thirdly, in order to handle the dynamic nature of JSON data, we proposed *Incremental embedding-based clustering* approach using *Incremental SchemaEmbed* model to cluster the new and modified documents incrementally. The *SVTree, JIndex*, and *EJIndex* are simultaneously updated without rebuilding from scratch. Through the discussion and comparison of experimental results, it is evident that the clustering quality of the proposed work is better than baseline models. Our analysis of update operations on the proposed index shows that it significantly reduces the index size, resulting in faster data retrieval.

7.1 FUTURE SCOPE

This section discusses the future research directions of the proposed work.

- In the present work, the schema embeddings generated by *SchemaEmbed* model are fed into a clustering algorithm to group the contextually relevant documents. In the future, the quality of the clustering algorithm can be further improved by jointly optimizing the *SchemaEmbed* model parameters and cluster centers in the same vector space.

- *SVTree* is an effort to design a compact data structure for summarizing schema variants. The number of nodes in the tree structure can be further minimized by identifying the frequent attribute set in the schema-specific attributes while preserving all the features of JSON data format.

- The proposed *JIndex* uses the path of an attribute as an index key to answering path-based queries. It is observed that recursive queries take a long query processing time by searching for all the child nodes of the query attribute. Assigning a single index key for all the child nodes of the query attribute helps reduce the query processing time. The single index key can be designed by combining the path of the sibling keys of an attribute, which eventually reduces the index size.

- While the preliminary experiments of the incremental approach for handling dynamic data show significant performance in updating the schema variants and indexes, we plan to examine the same in a real-time scenario.

BIBLIOGRAPHY

Abelló, A., de Palol, X. and Hacid, M.-S. (2018). "Approximating the Schema of a Set of Documents by Means of Resemblance." *Journal on Data Semantics*, 7(2), 87–105.

Aftab, Z., Iqbal, W., Almustafa, K. M., Bukhari, F. and Abdullah, M. (2020). "Automatic NoSQL to Relational Database Transformation with Dynamic Schema Mapping." *Scientific Programming*, 2020.

Agarwal, M. K., Ramamritham, K. and Agarwal, P. (2016). "Generic Keyword Search over XML Data.." In *EDBT*, 149–160.

Aïtelhadj, A., Boughanem, M., Mezghiche, M. and Souam, F. (2012). "Using structural similarity for clustering XML documents." *Knowledge and Information Systems*, 32(1), 109–139.

Alghamdi, N. S., Rahayu, W. and Pardede, E. (2014). "Semantic-based Structural and Content indexing for the efficient retrieval of queries over large XML data repositories." *Future Generation Computer Systems*, 37, 212–231. Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications.

Allan, J., Croft, B., Moffat, A. and Sanderson, M. (2012). "Frontiers, challenges, and opportunities for information retrieval: Report from SWIRL 2012 the second strategic workshop on information retrieval in Lorne." In *ACM SIGIR Forum*, volume 46, ACM New York, NY, USA, 2–32.

Amghar, S., Cherdal, S. and Mouline, S. (2019). "Data Integration and NoSQL Systems: A State of the Art." In *Proceedings of the 4th International Conference on Big Data and Internet of Things*, BDIoT'19, Association for Computing Machinery, New York, NY, USA.

Apache Drill (2019) `http://drill.apache.org/`,Last accessed on 01-08-2019).

Apache Spark (2019). "Latex — Wikipedia, the free encyclopedia." `https://spark.apache.org/docs/latest/sql-programming-guide.html`). [Last accessed on 21-09-2019].

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

Baazizi, M.-A., Colazzo, D., Ghelli, G. and Sartiani, C. (2019). "Parametric schema inference for massive JSON datasets." *The VLDB Journal*.

Baazizi, Mohamed-Amine and Colazzo, Dario and Ghelli, Giorgio and Sartiani, Carlo (2019). "Schemas and Types for JSON Data: From Theory to Practice." In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, Association for Computing Machinery, New York, NY, USA, 2060–2063.

Baeza-Yates, R., Ribeiro-Neto, B. et al. (1999). *Modern information retrieval*, volume 463, ACM press New York.

Bawakid, F. (2019). *A schema exploration approach for document-oriented data using unsupervised techniques*. PhD thesis, University of Southampton.

Bex, G. J., Neven, F., Schwentick, T. and Tuyls, K. (2006). "Inference of Concise DTDs from XML Data." In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, VLDB Endowment, 115–126.

Blaselbauer, V. M. and Josko, J. M. B. (2020). "JSONGlue: A hybrid matcher for JSON schema matching." In *Proceedings of the Brazilian Symposium on Databases*.

Bourhis, P., Reutter, J. L., Suárez, F. and Vrgoč, D. (2017). "JSON: Data Model, Query Languages and Schema Specification." In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, ACM, New York, NY, USA, 123–135.

Budiu, M., Yu, Y. and Zhang, L. (2014). "Unified Query Processing for JSON Documents and Indexes." Technical Report MSR-TR-2014-129, Microsoft Research.

Cánovas Izquierdo, J. L. and Cabot, J. (2013). "Discovering Implicit Schemas in JSON Data." In Daniel, F., Dolog, P. and Li, Q., editors, *Web Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, 68–83.

Carpineto, C. and Romano, G. (2012). "A survey of automatic query expansion in information retrieval." *Acm Computing Surveys (CSUR)*, 44(1), 1–50.

Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C. et al. (2018). "Universal Sentence Encoder for English." In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 169–174.

Chouder, M. L., Rizzi, S. and Chalal, R. (2017). "Enabling Self-Service BI on Document Stores.." In *EDBT/ICDT Workshops*.

Chouder, M. L., Rizzi, S. and Chalal, R. (2019). "EXODuS: Exploratory OLAP over Document Stores." *Information Systems*, 79, 44–57. Special issue on DOLAP 2017: Design, Optimization, Languages and Analytical Processing of Big Data.

Ciucanu, R. and Staworko, S. (2013). "Learning schemas for unordered XML." *arXiv preprint arXiv:1307.6348*.

Conneau, A., Kiela, D., Schwenk, H., Barrault, L. and Bordes, A. (2017). "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data." In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Copenhagen, Denmark, 670–680.

Costa, G. and Ortale, R. (2013). "A latent semantic approach to XML clustering by content and structure based on non-negative matrix factorization." In *2013 12th International Conference on Machine Learning and Applications*, volume 1, IEEE, 179–184.

Costa, G. and Ortale, R. (2017). "XML clustering by structure-constrained phrases: A fully-automatic approach using contextualized n-grams." *International Journal on Artificial Intelligence Tools*, 26(01), 1760002.

Couch Spark Connector (2019))`https://github.com/couchbase/couchbase-spark-connector/wiki/Spark-SQL`,Last accessed on 23-09-2019.

Curé, O., Hecht, R., Duc, C. L. and Lamolle, M. (2011). "Data integration over NoSQL stores using access path based mappings." In *International Conference on Database and Expert Systems Applications*, Springer, 481–495.

D, U. P. and Santhi Thilagam, P. (2022). "ClustVariants: An Approach for Schema Variants Extraction from JSON Document Collections." In *2022 IEEE IAS Global Conference on Emerging Technologies (GlobConET)*, 515–520.

Dhanalekshmi, G. and Asawa, K. (2018). "A combined path index for efficient processing of XML queries." *International Journal of Metadata, Semantics and Ontologies*, 13(1), 20–32.

DiScala, M. and Abadi, D. J. (2016). "Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data." In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, ACM, New York, NY, USA, 295–310.

Elastic Search (2019). "Elastic search.")`http://www.elastic.co/guide/en/elasticsearch/reference/current/dynamic-mapping.html`,Last accessed on 12-08-2019.

Finger, A., Bruder, I., Heuer, A., Klemkow, M. and Konerow, S. (2014). "PageBeat-Zeitreihenanalyse und Datenbanken.." In *Grundlagen von Datenbanken*, 53–58.

Forresi, C., Gallinucci, E., Golfarelli, M. and Hamadou, H. B. (2021). "A dataspace-based framework for OLAP analyses in a high-variety multistore." *The VLDB Journal*, 30(6), 1017–1040.

Frozza, A. A., Defreyn, E. D. and dos Santos Mello, R. (2020a). "A Process for Inference of Columnar NoSQL Database Schemas." In *Anais do XXXV Simp{ó}sio Brasileiro de Bancos de Dados*, SBC, 175–180.

Frozza, A. A., Defreyn, E. D. and dos Santos Mello, R. (2021). "An Approach for Schema Extraction of NoSQL Columnar Databases: the HBase Case Study." *Journal of Information and Data Management*, 12(5).

Frozza, A. A., dos Santos Mello, R. and da Costa, F. d. S. (2018). "An approach for schema extraction of JSON and extended JSON document collections." In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, IEEE, 356–363.

Frozza, A. A., Jacinto, S. R. and dos Santos Mello, R. (2020b). "An approach for schema extraction of NoSQL graph databases." In *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*, IEEE, 271–278.

Frozza, A. A. and Mello, R. d. S. (2020). "JS4Geo: a canonical JSON Schema for geographic data suitable to NoSQL databases." *GeoInformatica*, 24(4), 987–1019.

Gallinucci, E., Golfarelli, M. and Rizzi, S. (2018). "Schema profiling of document-oriented databases." *Information Systems*, 75, 13–25.

Gallinucci, E., Golfarelli, M. and Rizzi, S. (2019). "Approximate OLAP of document-oriented databases: A variety-aware approach." *Information Systems*, 85, 114 – 130.

Goldman, R. and Widom, J. (1997). "DataGuides: Enabling query formulation and optimization in semistructured databases." VLDB, 436–445. cited By 861.

Gómez, P., Casallas, R. and Roncancio, C. (2016). "Data schema does matter, even in NoSQL systems!." In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, IEEE, 1–6.

Guo, J., Cai, Y., Fan, Y., Sun, F., Zhang, R. and Cheng, X. (2022). "Semantic Models for the First-Stage Retrieval: A Comprehensive Review." *ACM Trans. Inf. Syst.*, 40(4).

Habib, A., Shinnar, A., Hirzel, M. and Pradel, M. (2019). "Type safety with JSON subschema." *arXiv preprint arXiv:1911.12651*.

Hamadou, H. B., Ghozzi, F., Péninou, A. and Teste, O. (2019). "Schema-independent querying for heterogeneous collections in NoSQL document stores." *Information Systems*, 85, 48–67.

Hassanzadeh, O., Pu, K. Q., Yeganeh, S. H., Miller, R. J., Popa, L., Hernández, M. A. and Ho, H. (2013). "Discovering Linkage Points over Web Data." *Proceedings of the VLDB Endowment*, 6(6).

Hsu, W.-C. and Liao, I.-E. (2013). "CIS-X: A compacted indexing scheme for efficient query evaluation of XML documents." *Information Sciences*, 241, 195–211.

Hsu, W.-C. and Liao, I.-E. (2020). "UCIS-X: An Updatable Compact Indexing Scheme for Efficient Extensible Markup Language Document Updating and Query Evaluation." *IEEE Access*, 8, 176375–176392.

Huang, J.-T., Sharma, A., Sun, S., Xia, L., Zhang, D., Pronin, P., Padmanabhan, J., Ottaviano, G. and Yang, L. (2020). "Embedding-based retrieval in facebook search." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM, 2553–2561.

Irshad, L., Yan, L. and Ma, Z. (2019). "Schema-based JSON data stores in relational databases." *Journal of Database Management (JDM)*, 30(3), 38–70.

Istiqamah, A. N. and Wiharja, K. R. S. (2021). "A Schema Extraction of Document-Oriented Database for Data Warehouse." *International Journal on Information and Communication Technology (IJoICT)*, 7(2), 36–47.

Janga, P. and Davis, K. C. (2019). "A grammar-based approach for XML schema extraction and heterogeneous document integration." *International Journal of Data Mining, Modelling and Management*, 11(3), 235–258.

Jiang, L., Qiu, J. and Zhao, Z. (2020). "Scalable structural index construction for JSON analytics." *Proceedings of the VLDB Endowment*, 14(4), 694–707.

Johnson, J., Douze, M. and Jégou, H. (2019). "Billion-scale similarity search with gpus." *IEEE Transactions on Big Data*, 7(3), 535–547.

Kellou-Menouer, K., Kardoulakis, N., Troullinou, G., Kedad, Z., Plexousakis, D. and Kondylakis, H. (2021). "A survey on semantic schema discovery." *The VLDB Journal*, 1–36.

Kellou-Menouer, K. and Kedad, Z. (2017). "On-line Versioned Schema Inference for Large Semantic Web Data Sources." In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 1–12.

Klettke, M., Awolin, H., Störl, U., Müller, D. and Scherzinger, S. (2017). "Uncovering the evolution history of data lakes." In *2017 IEEE international conference on big data (Big Data)*, IEEE, 2462–2471.

Klettke, M., Störl, U. and Scherzinger, S. (2015). "Schema extraction and structural outlier detection for JSON-based NoSQL data stores." *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*.

Klettke, M., Störl, U., Shenavai, M. and Scherzinger, S. (2016). "NoSQL schema evolution and big data migration at scale." In *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, 2764–2774.

Koupil, P., Hricko, S. and Holubová, I. (2022). "MM-infer: A Tool for Inference of Multi-Model Schemas.." In *EDBT*, 2–566.

Kumar, S., Rana, R. K. and Singh, P. (2012). "Ontology based semantic indexing approach for information retrieval system." *International Journal of Computer Applications*, 49(12).

Lashkari, F., Bagheri, E. and Ghorbani, A. A. (2019). "Neural embedding-based indices for semantic search." *Information Processing & Management*, 56(3), 733–755.

Le, Q. and Mikolov, T. (2014). "Distributed representations of sentences and documents." In *International conference on machine learning*, PMLR, 1188–1196.

Li, Y., Cao, J., Chen, H., Ge, T., Xu, Z. and Peng, Q. (2020). "FlashSchema: Achieving High Quality XML Schemas with Powerful Inference Algorithms and Large-scale Schema Data." In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, IEEE, 1962–1965.

Li, Y., Zhang, X., Cao, J., Chen, H. and Gao, C. (2019). "Learning k-Occurrence Regular Expressions with Interleaving." In *International Conference on Database Systems for Advanced Applications*, Springer, 70–85.

Li, Y., Zhang, X., Xu, H., Mou, X. and Chen, H. (2018). "Learning restricted regular expressions with interleaving from XML data." In *International Conference on Conceptual Modeling*, Springer, 586–593.

Liu, P., Wang, S., Wang, X., Ye, W. and Zhang, S. (2021a). "QuadrupletBERT: An Efficient Model For Embedding-Based Large-Scale Retrieval." In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 3734–3739.

Liu, P., Wang, X., Wang, S., Ye, W., Xi, X. and Zhang, S. (2021b). "Improving Embedding-based Large-scale Retrieval via Label Enhancement." In *Findings of the Association for Computational Linguistics: EMNLP 2021*, Punta Cana, Dominican Republic, Association for Computational Linguistics, 133–142.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. and Stoyanov, V. (2019). "RoBERTa: A robustly optimized BERT pretraining approach." *arXiv preprint arXiv:1907.11692*.

Liu, Z. H. and Gawlick, D. (2015). "Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -." In *CIDR*, CIDR.

Liu, Z. H., Hammerschmidt, B. and McMahon, D. (2014). "JSON data management: supporting schema-less development in RDBMS." In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 1247–1258.

Liu, Z. H., Hammerschmidt, B., McMahon, D., Liu, Y. and Chang, H. J. (2016). "Closing the Functional and Performance Gap between SQL and NoSQL." In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, Association for Computing Machinery, New York, NY, USA, sigmod, 227–238.

Maatuk, A. M., Ali, M. A. and Aljawarneh, S. (2015). "An algorithm for constructing XML Schema documents from relational databases." In *Proceedings of the The International Conference on Engineering & MIS 2015*, ACM, 12.

Mikolov, T., Chen, K., Corrado, G. and Dean, J. (2013). "Efficient estimation of word representations in vector space." *arXiv preprint arXiv:1301.3781*.

Miller, G. A. (1995). "WordNet: a lexical database for English." *Communications of the ACM*, 38(11), 39–41.

Miller, G. A. (1998). *WordNet: An electronic lexical database*, MIT press.

Min, J.-K., Lee, J. and Chung, C.-W. (2009). "An efficient XML encoding and labeling method for query processing and updating on dynamic XML data." *Journal of Systems and Software*, 82(3), 503–515.

Mlỳnková, I. (2008). "XML Schema Inference: A Study." *Technická zpráva, Charles University, Prague, Czech Republic*.

Mlýnková, I. and Nečaský, M. (2009). "Towards Inference of More Realistic XSDs." In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, ACM, New York, NY, USA, 639–646.

Mohamed L. Chouder and Stefano Rizzi and Rachid Chalal (2017). "JSON Datasets for Exploratory OLAP." `doi:10.17632/ct8f9skv97.1`). [Last accessed on 21-12-2020].

Möller, M. L., Scharlau, N. and Klettke, M. (2021). "An Empirical Study of Open Data JSON Files.." In *DOLAP*, 121–125.

mongodb schema (2019). "mongodb-schema." `https://github.com/mongodb-js/mongodb-schema`,Last accessed on 23-09-2019).

NAMBA, J. (2021). "Enhancing JSON Schema Discovery by Uncovering Hidden Data." In *Proceedings of the VLDB 2021 PhD Workshop*, Copenhagen, Denmark, VLDB.

Navigli, R. (2009). "Word Sense Disambiguation: A Survey." *ACM Comput. Surv.*, 41(2).

Park, J., Park, C., Kim, J., Cho, M. and Park, S. (2019). "ADC: Advanced document clustering using contextualized representations." *Expert Systems with Applications*, 137, 157–166.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K. and Zettlemoyer, L. (2018). "Deep contextualized word representations." In *Proceedings of NAACL-HLT*, 2227–2237.

Piernik, M., Brzezinski, D. and Morzy, T. (2016). "Clustering XML documents by patterns." *Knowledge and Information Systems*, 46(1), 185–212.

Piernik, M., Brzezinski, D., Morzy, T. and Lesniewska, A. (2015). "XML clustering: a review of structural approaches." *The Knowledge Engineering Review*, 30(3), 297–323.

Priya, D. U. and Thilagam, P. S. (2022). "JSON document clustering based on schema embeddings." *Journal of Information Science*, 01655515221116522.

Qadah, G. Z. (2017). "Indexing techniques for processing generalized XML documents." *Computer Standards and Interfaces*, 49, 34–43.

Qtaish, A. and Ahmad, K. (2016). "XAncestor: An efficient mapping approach for storing and querying XML documents in relational database using path-based technique." *Knowledge-Based Systems*, 114, 167–192.

Quilitz, B. and Leser, U. (2008). "Querying distributed RDF data sources with SPARQL." In *European semantic web conference*, Springer, 524–538.

Sasaki, Y., Fletcher, G. and Onizuka, M. (2020). "Structural indexing for conjunctive path queries." *arXiv preprint arXiv:2003.03079*.

schema.js (2019))https://www.npmjs.com/package/schema-js,Last accessed on 10-07-2019.

Scherzinger, S., Klettke, M. and Störl, U. (2013). "Managing schema evolution in NoSQL data stores." *arXiv preprint arXiv:1308.0514*.

Sculley, D. (2010). "Web-scale k-means clustering." In *Proceedings of the 19th international conference on World wide web*, 1177–1178.

SeedScientific (2021). "How much data is created every day?." https://seedscientific.com/how-much-data-is-created-every-day/). [Last accessed on 21-12-2021].

Sevilla Ruiz, D., Morales, S. F. and García Molina, J. (2015). "Inferring versioned schemas from NoSQL databases and its applications." In Johannesson, P., Lee, M. L., Liddle, S. W., Opdahl, A. L. and Pastor López, Ó., editors, *Conceptual Modeling*, Springer International Publishing, Cham, 467–480.

Shang, S., Wu, Q., Wang, T. and Shao, Z. (2021). "LiteIndex: Memory-Efficient Schema-Agnostic Indexing for JSON documents in SQLite." In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 435–440.

Shukla, D., Thota, S., Raman, K., Gajendran, M., Shah, A., Ziuzin, S., Sundaram, K., Guajardo, M. G., Wawrzyniak, A., Boshra, S. et al. (2015). "Schema-agnostic indexing with Azure DocumentDB." *Proceedings of the VLDB Endowment*, 8(12), 1668–1679.

Song, C., Huang, Y., Liu, F., Wang, Z. and Wang, L. (2014). "Deep Auto-Encoder Based Clustering." *Intell. Data Anal.*, 18(6S), S65–S76.

Spoth, W., Arab, B. S., Chan, E. S., Gawlick, D., Ghoneimy, A., Glavic, B., Hammerschmidt, B., Kennedy, O., Lee, S., Liu, Z. H. et al. (2017). "Adaptive schema databases." In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

Spoth, W., Kennedy, O., Lu, Y., Hammerschmidt, B. and Liu, Z. H. (2021). "Reducing ambiguity in JSON schema discovery." In *Proceedings of the 2021 International Conference on Management of Data*, 1732–1744.

Spoth, W., Xie, T., Kennedy, O., Yang, Y., Hammerschmidt, B., Liu, Z. H. and Gawlick, D. (2018). "SchemaDrill: Interactive Semi-Structured Schema Design." In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA'18, ACM, New York, NY, USA, 11:1–11:7.

Spring, A., Lewerentz, M., Bluhm, T., Heimann, P., Hennig, C., Kühner, G., Kroiss, H., Krom, J. G., Laqua, H., Maier, J. et al. (2012). "A W7-X experiment program editor—-A usage driven development." *Fusion Engineering and Design*, 87(12), 1954–1957.

Subramaniam, S., Haw, S.-C. and Soon, L.-K. (2019). "Performance evaluation of XML query processing in centralized and distributed environment." In *Proceedings of the 2019 2nd International Conference on Computational Intelligence and Intelligent Systems*, 129–133.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C. (2002). "Storing and querying ordered XML using a relational database system." In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, 204–215.

Tekli, J. (2016). "An overview on XML semantic disambiguation from unstructured text to semi-structured data: Background, applications, and ongoing challenges." *IEEE Transactions on Knowledge and Data Engineering*, 28(6), 1383–1407.

Tekli, J., Chbeir, R., Traina, A. J. and Traina Jr, C. (2019). "SemIndex+: A semantic indexing scheme for structured, unstructured, and partly structured data." *Knowledge-Based Systems*, 164, 378–403.

Tonellotto, N. and Macdonald, C. (2021). "Query Embedding Pruning for Dense Retrieval." In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, IEEE, 3453–3457.

Uma Priya, D. and Santhi Thilagam, P. (2020). "Dynamic Data Retrieval Using Incremental Clustering and Indexing." *International Journal of Information Retrieval Research (IJIRR)*, 10(3), 74–91.

Valeri, K. (2014). "Crunching 30 years of NBA data with MongoDB aggregation.")`https://thecodebarbarian.wordpress.com/`.

variety.js (2019))`https://github.com/variety/variety`,Last accessed on 01-08-2019.

Vinh, N. X., Epps, J. and Bailey, J. (2010). "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance." *J. Mach. Learn. Res.*, 11, 2837–2854.

Von Luxburg, U. (2007). "A tutorial on spectral clustering." *Statistics and computing*, 17(4), 395–416.

Wahyudi, E., Sfenrianto, S., Hakim, M. J., Subandi, R., Sulaeman, O. R. and Setiyawan, R. (2019). "Information retrieval system for searching JSON files with vector space model method." In *2019 International Conference of Artificial Intelligence and Information Technology (ICAIIT)*, IEEE, 260–265.

Wang, L., Zhang, S., Shi, J., Jiao, L., Hassanzadeh, O., Zou, J. and Wangz, C. (2015). "Schema Management for Document Stores." *Proc. VLDB Endow.*, 8(9), 922–933.

Wang, X. and Chen, H. (2019). "Learning Restricted Deterministic Regular Expressions with Counting." In *International Conference on Web Information Systems Engineering*, Springer, 98–114.

Wellenzohn, K., Böhlen, M. H. and Helmer, S. (2020). "Dynamic interleaving of content and structure for robust indexing of semi-structured hierarchical data (extended version)." *arXiv preprint arXiv:2006.05134*.

Xie, J., Girshick, R. and Farhadi, A. (2016). "Unsupervised deep embedding for clustering analysis." In *International conference on machine learning*, PMLR, 478–487.

Yang, B., Fu, X., Sidiropoulos, N. D. and Hong, M. (2017). "Towards k-means-friendly spaces: Simultaneous deep learning and clustering." In *international conference on machine learning*, PMLR, 3861–3870.

Zhan, J., Mao, J., Liu, Y., Guo, J., Zhang, M. and Ma, S. (2021). "Jointly optimizing query encoder and product quantization to improve retrieval performance." In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2487–2496.

Zhan, J., Mao, J., Liu, Y., Zhang, M. and Ma, S. (2020). "Learning To Retrieve: How to Train a Dense Retrieval Model Effectively and Efficiently." *arXiv preprint arXiv:2010.10469*.

Zhang, E. and Zhang, Y. (2009). *Average Precision*, 192–193. Springer US, Boston, MA.

Zhang, W., Byna, S., Tang, H., Williams, B. and Chen, Y. (2019). "MIQS: Metadata Indexing and Querying Service for Self-Describing File Formats." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, Association for Computing Machinery, New York, NY, USA.

Zhang, X., Li, Y., Cui, F., Dong, C. and Chen, H. (2018). "Inference of a Concise Regular Expression Considering Interleaving from XML Documents." In Phung, D., Tseng, V. S., Webb, G. I., Ho, B., Ganji, M. and Rashidi, L., editors, *Advances in Knowledge Discovery and Data Mining*, Springer International Publishing, Cham, 389–401.

Zhou, X., Gururajan, R., Li, Y., Venkataraman, R., Tao, X., Bargshady, G., Barua, P. D. and Kondalsamy-Chennakesavan, S. (2020). "A survey on text classification and its applications." In *Web Intelligence*, number Preprint, IOS Press, 1–12.

PUBLICATIONS

JOURNAL PUBLICATIONS

1. Uma Priya D. & P. Santhi Thilagam. (2020). Dynamic Data Retrieval using Incremental Clustering and Indexing. International Journal of Information Retrieval Research (IJIRR), 10(3), 661-685. DOI: https://doi.org/10.4018/IJIRR.2020070105.

2. Uma Priya D. & P. Santhi Thilagam. (2022). JSON Document Clustering based on Schema Embeddings. Journal of Information Science, SAGE DOI: https://doi.org/10.1177/01655515221116522.

3. Uma Priya D. & P. Santhi Thilagam. (2022). Leveraging Structural and Semantic Measures for JSON Document Clustering. Journal of Universal Computer Science DOI: https://doi.org/10.3897/jucs.86563

4. Uma Priya D. & P. Santhi Thilagam. (2022). JSON Schema Discovery for NoSQL Databases: Research Trends and Applications. Computer Science Review, Elsevier (status: "Under Review")

CONFERENCE PUBLICATIONS

1. Uma Priya D. & P. Santhi Thilagam. (2022). ClustVariants: An Approach for Schema Variants Extraction from JSON Document Collections. *2022 IEEE IAS Global Conference on Emerging Technologies (GlobConET)*, May 20-22, 2022, 515-520. DOI: `https://doi.org/10.1109/GlobConET53749.2022.9872382`.

2. Uma Priya D. & P. Santhi Thilagam. (2022). JSON Documents Clustering based on Structural Similarity and Semantic Fusion. 5^{th} *International Conference on Computational Intelligence and Data Engineering (ICCIDE-2022)*, August 12-13, 2022, VIT-AP University, India (Received BEST PAPER Award, Awaiting for Publication).

3. Uma Priya D. & P. Santhi Thilagam. (2022). Extracting Schema Variants from JSON Collections using JSVTree, *CODS-COMAD 2023*, January 4-7, 2023, IIT Bombay, India. DOI: `https://doi.org/10.1145/3570991.3571032`