Journal of Information and Data Management, 2026, 17:1, doi: 10.5753/jidm.2026.5700

Incremental Schema Evolution in Document-oriented Databases

Eleonília Monteiro Rodrigues [**Universidade Federal de Campina Grande (UFCG)** | ***eleoniliamonteiro@copin.ufcg.edu.br***]
Carlos Eduardo S. Pires [**Universidade Federal de Campina Grande (UFCG)** | ***cesp@dsc.ufcg.edu.br***]
Dimas Cassimiro do N. Filho [**Universidade Federal do Agreste de Pernambuco (UFAPE)** | ***dimas.cassimiro@ufape.edu.br***]

Computer Science Department, Federal University of Campina Grande (UFCG)

Received: 22 March 2025 • **Published:** 13 March 2026

Abstract Document-oriented databases offer high flexibility for storing structured, semi-structured, and unstructured data. However, the absence or obsolescence of predefined schemas can hinder the interpretation and analysis of the stored information. This study proposes an approach for the incremental evolution of schemas in document-oriented databases, aiming to continuously update the schema as new documents are inserted. The proposed solution is fully automated, requiring no manual intervention, and eliminates the need to reprocess the entire dataset. Experimental results, obtained from metadata collections derived from books and Twitter posts, show that the proposed approach achieves performance gains ranging from 1.93 to 2.85 times compared to a traditional batch-based approach.

Keywords:
Document-oriented databases, Incremental schema evolution, NoSQL, JSON schema.

1 Introduction

In recent years, the accelerated growth of data production has increased the demand for robust and scalable solutions capable of efficiently storing, processing, and analyzing massive datasets. Big Data technologies have become essential tools for organizations seeking to extract actionable insights that support strategic decision-making. Within this context, NoSQL databases stand out for their flexibility in managing structured, semi-structured, and unstructured data without imposing rigid, predefined schemas. However, this flexibility comes at a cost: the absence or obsolescence of an explicit schema introduces critical challenges, such as hindering real-time data validation, degrading query and read performance, complicating the integration of external tools and frameworks, and increasing the complexity of database maintenance [Baazizi *et al*., 2019; Atmatzides *et al*., 2022].

The problem of inferring schemas from semi-structured datasets has been extensively studied [Sevilla Ruiz *et al*., 2015; Cánovas Izquierdo and Cabot, 2013; Klettke *et al*., 2017; Baazizi *et al*., 2019]. Among the notable contributions, [Frozza *et al*., 2018]. [Frozza *et al*., 2018] proposed the JSON Schema Discovery (JSD) approarch, which infers JSON (JavaScript Object Notation) schemas from collections of JSON and extended JSON documents. While effective in generating schemas, JSD (batch) lacks support for dynamic schema evolution. As new documents are inserted into the collection, JSD requires recomputing the schema over the entire dataset to ensure its accuracy, which incurs significant computational overhead and limits its applicability in environments with continuous data ingestion.

Dynamic schema evolution is critical for ensuring data consistency and enabling informed decision-making across various domains. In e-commerce, platforms such as Amazon, Lojas Americanas, and eBay continually collect transaction records, product reviews, browsing histories, and customer preferences [Purnomo, 2023]. Keeping schemas up to date facilitates data integration and analysis, allowing organizations to optimize marketing strategies, improve fraud detection mechanisms, manage inventories, and personalize product recommendations based on accurate and current information. Similarly, social media platforms such as Twitter, Facebook, VK, and Instagram process diverse and dynamic data types, including posts, comments, likes, and user demographic information. Continuous schema updates are essential for identifying trends, analyzing user behavior, and enhancing user experiences [Karim and Boulmakoul, 2021; Li *et al*., 2023].

The proposed solution addresses the challenge of dynamic schema evolution in document-oriented collections by introducing JSD Evolution, an incremental approach to update schemas as new documents are added. Building upon the batch-based JSD approach of [Frozza *et al*., 2018], the proposed solution assumes the existence of a JSON schema that initially represents the document collection. As new documents are inserted, the schema incrementally evolves to accommodate structural changes, thereby reducing the computational cost and operational effort typically required for data management in dynamic environments, as it eliminates the need to process the entire collection. Accordingly, this study hypothesizes that an incremental approach can significantly reduce processing time while preserving the correctness and consistency of the resulting schema.

An earlier version of this work introduced the foundational concepts of JSD Evolution (incremental JSD) and reported preliminary experiments that demonstrated its efficiency [Rodrigues *et al*., 2024]. In this extended study, we present a more comprehensive analysis of the performance and scalability of JSD Evolution by examining the effects of document structural complexity, attribute density, and file size on processing times. Furthermore, we conduct a com-

parative evaluation of execution times between the batch approach (JSD) and the incremental strategy (JSD Evolution), with particular emphasis on identifying the algorithmic stages that most significantly impact overall execution time.

The remainder of this document is organized as follows: Section 2 presents the theoretical background; Section 3 discusses related works; Section 4 details the proposed incremental approach for schema evolution; Section 5 presents the experimental methodology; Section 6 discusses the obtained results; finally, Section 7 presents the conclusions of the work and future directions.

2 JSON, Extended JSON and JSON Schema

JSON[1] is a data transfer standard that enables a simple and concise representation of data structures in a text-based format. It is widely used in distributed systems and *Application Programming Interfaces (APIs)* as it allows different systems to exchange data in a standardized and unambiguous manner. In addition to standard JSON, which supports the data types Object, Array, String, Number, Boolean, and Null, Extended JSON introduces additional data types not present in the original specification. These include Date, Timestamp, Binary, ObjectID, RegExp, Long, Undefined, DBRef, Code, MinKey, and MaxKey.

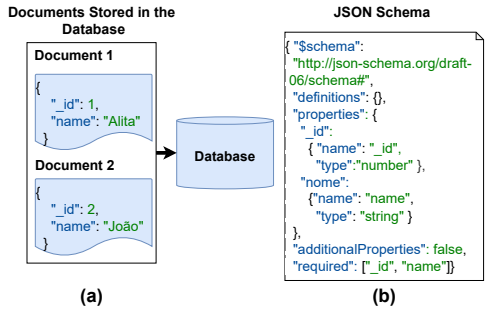


Figure 1. Examples of JSON documents and JSON schema

The JSON standard is also adopted by NoSQL database systems such as MongoDB[2]. In this system, data is manipulated in JSON documents and stored in BSON (binary JSON). Each document consists of key-value pairs and is considered a unique object within a collection of documents. The flexibility of the JSON format allows modifications to document structures without affecting the operation of applications using MongoDB [Frozza *et al*., 2018; Atmatzides *et al*., 2022].

A JSON Schema[3] is a standard that enables the description and validation of the structure, constraints, and data types in JSON documents. This standard provides a declarative schema-based language that defines the structure of a JSON document, specifies the data types it must contain, and establishes validation rules for each attribute. Consequently, it ensures that received data conforms to the expected format and structure. Figure 1 presents two JSON documents along with the corresponding JSON Schema that validates them. The schema specifies that the attributes *_id* and *name* are associated with the data types $Number$ and $String$, respectively, and that they are mandatory, meaning they must be present in all documents.

[1] https://www.json.org/
[2] https://www.mongodb.com/pt-br
[3] https://json-schema.org/

3 Related Work

In [Frozza *et al*., 2018], an approach is proposed for extracting schemas from collections of JSON (JavaScript Object Notation) and extended JSON documents. The approach consists of four stages. In the first stage, a raw schema is generated for each document in the collection. A raw schema preserves the hierarchical structure of attributes, nested objects, and arrays (or matrices), maintaining the organization of the data within the document. Primitive values in the document are replaced by JSON or extended JSON data types. In the second stage, an initial aggregation is performed to extract a collection of unique JSON objects. Then, the attributes of the raw schemas obtained in the first aggregation are sorted in alphabetical order (this sorting is applied to reduce the number of unique raw schemas). After sorting the attributes, a second aggregation is applied. The third stage involves constructing a data structure (tree) to store information about the hierarchical structure of the raw schemas, referred to as the Raw Schema Unified Structure (RSUS). The first raw schema serves as the foundation, and the others are used to update the tree with new information. In the final stage, the tree structure is transformed into a JSON schema.

Although the approach proposed by [Frozza *et al*., 2018] generates schemas, it does not support schema evolution when new documents are added to the collection. To obtain an updated schema, the entire approach must be reapplied to the entire collection (both new and existing documents), which can significantly increase processing time compared to the solution proposed in this work. In contrast, our proposed solution extends the approach of [Frozza *et al*., 2018] by incorporating incremental schema evolution, where only newly inserted documents trigger schema updates.

The studies of [Sevilla Ruiz *et al*., 2015] and [Abdelhédi *et al*., 2022] focus on schema generation to represent datasets, while the research of [Baazizi *et al*., 2019], [Cánovas Izquierdo and Cabot, 2013], and [Klettke *et al*., 2017] addresses schema updates. It is important to note that, although [Baazizi *et al*., 2019] and [Cánovas Izquierdo and Cabot, 2013] perform updates, they do not do so incrementally. In contrast, to evolve the existing schema, [Baazizi *et al*., 2019] proposes generating a new schema representing the newly inserted documents, which is then merged with the pre-existing schema. The schema evolution proposed by [Cánovas Izquierdo and Cabot, 2013] occurs continuously. As new documents are added to the collection, they are used to evolve the existing schema, ensuring adaptation to changes in the documents.

The approach described by [Klettke *et al*., 2017], which aims to reconstruct the historical changes of a database schema over time. This method relies on the use of temporal metadata (timestamps) and requires that data be organized in chronological order. A key component of the approach is the need for human intervention to resolve ambiguities (e.g.,

Table 1. Comparison of Related Works

	Input Format	Objective	Main Steps	Incremental Schema Evolution	Automation
[Baazizi *et al*., 2019]	JSON	Generate schemas for massive databases	1. Map 2. Reduce (Type and Label)	no	yes
[Cánovas Izquierdo and Cabot, 2013]	JSON Web	Generate schema for web API services	1. Pre-discovery 2. Single Service Discovery 3. Multi-service Discovery	no	yes
[Klettke *et al*., 2017]	JSON	Tool for schema visualization and validation	1. Extraction of Structures 2. Derivation of Schema Evolution Operations 3. Resolution of Ambiguities	yes	no (manual ambiguity resolution)
Proposed Approach	JSON and Extended JSON	JSON schema evolution	1. Generate raw schemas (new documents) 2. Group raw schemas (both new and existing) 3. Update the tree (new raw schemas) 4. Generate JSON Schema	yes	yes

when it is unclear whether a new attribute was added directly or copied from another entity), which is achieved through the construction of decision tables based on information about the entities involved, their interrelationships, and the types of transformations identified (e.g., attribute insertion or renaming). In contrast, the JSD Evolution approach adopts an incremental evolution model focused on automation and the direct update of a single schema. As new documents are incorporated, the schema is incrementally adjusted without requiring full reprocessing or the use of temporal markers. This strategy supports processing documents in any order and eliminates the need for manual validation. Due to its computational efficiency and its ability to preserve the structural coherence of the resulting schema, the approach is well-suited for scenarios in which data is inserted continuously and without temporal ordering.

Table 1 summarizes the objectives and main steps of the compared methods while highlighting limitations regarding incremental evolution and automation. While [Baazizi *et al*., 2019] and [Cánovas Izquierdo and Cabot, 2013] address large-scale schema extraction and schema generation for web APIs, respectively, both lack support for incremental updates. Although [Klettke *et al*., 2017] adopts an incremental strategy, its reliance on chronological ordering and manual intervention imposes constraints on scalability. The proposed approach overcomes these limitations by eliminating temporal prerequisites and enabling automated updates, making it more suitable for scenarios involving unordered and continuous data insertions. While [Baazizi *et al*., 2019] and [Cánovas Izquierdo and Cabot, 2013] address large-scale schema extraction and schema generation for web APIs, respectively, both lack support for incremental updates.

4 Proposed Solution

This section introduces the proposed solution for the incremental evolution of JSON (JavaScript Object Notation) schemas, addressing the formalization of the problem, providing an overview of the approach, and detailing the methodology employed in the JSD Evolution approach.

4.1 Formalization of the Problem

This section addresses the issue of the outdated nature of the JSON schema within a collection of documents and emphasizes that the insertion of new documents can cause the original schema to no longer accurately reflect the content of the collection.

Let D represent the initial set of JSON documents, where each document has a unique identifier id_i. Let S denote the JSON schema representing the set of documents D. Assume that f is the batch schema generation method that derives the JSON schema S from the set of documents D, as expressed in the following equation:

$$f_{\text{inc}} : Documents \rightarrow Schema \quad \text{s.t.} \quad f(D) = S \tag{1}$$

In the incremental evolution of the JSON schema S, a specific operation is considered: the insertion of new documents into D. These insertions are referred to as increments, denoted as Dn_i (with *i* varying from 1 to *m*), where Dn_i contains the newly added documents. The outcome of applying these insertions to D is defined by $D \cup Dn_i$.

In this study, we assume that each increment Dn_i contains only insertion operations. Furthermore, the documents in the increments Dn_i did not exist in the original document set D.

Definition 4.1 [Incremental Evolution of JSON Schemas]. Let D be a set of documents and Dn_i be an increment for D. Let S be the JSON schema associated with D. he incremental evolution of the JSON schema is performed on S based on Dn_i. The incremental evolution method is denoted as f_{inc} and is defined as:

$$\begin{aligned} f_{\text{inc}} : Schema \times Documents \rightarrow Schema \\ \text{s.t.} \quad f_{\text{inc}}(S, D_{n_i}) = S' \end{aligned} \tag{2}$$

The incremental evolution of schemas has two objectives:

- **Processing time efficiency**: the evolution process should execute faster than batch schema evolution, especially when the number of documents in an increment is small. It follows that $f_{inc}(S, Dn_i)$ should be faster than $f(D \cup Dn_i)$ when $|D| > |Dn_i|$;
- **Schema equivalence**: the results produced should be comparable to batch evolution, i.e.,

$$f_{inc}(S, Dn_i) \simeq f(D \cup Dn_i), \tag{3}$$

where $\simeq$ denotes the equivalence relation between the schema produced by incremental evolution S' and the one obtained through the batch process S. Two schemas S' and S are considered equivalent if, and only if, they share the same logical structure, disregarding the ordering of attributes and types.

4.2 Overview of the JSD Evolution Approach

This section presents the proposed solution for the incremental evolution of schemas in JSON document sets. The solution assumes that a JSON schema already exists to represent a

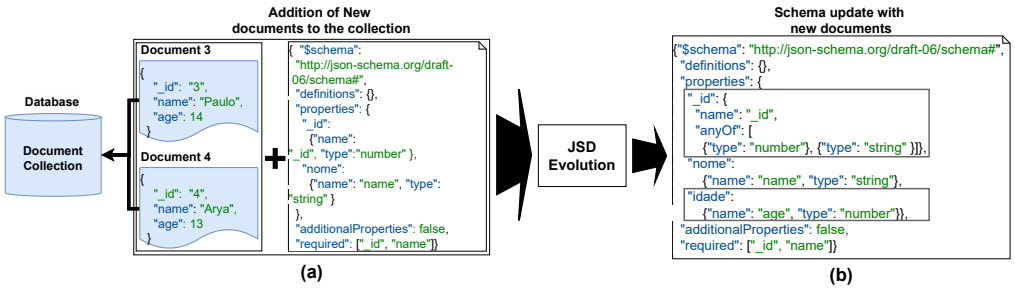


Figure 2. Insertion of new JSON documents and subsequent evolution of the JSON schema

collection of documents, and only the new documents added to the collection (i.e., documents not used in the creation of the original JSON schema) are considered for schema evolution. By considering only the new documents, there is no need to recreate the schema from scratch, which would require reprocessing all documents (both newly added and previously stored documents).

To illustrate, consider the example in Figure 2. The insertion of two new documents (Documents 3 and 4) into the database (Figure 2 (a)) causes the initial JSON schema (Figure 1(b)) to no longer reflect the content of the document set, as the schema does not yet include the *age* attribute introduced by the new documents. To ensure that this JSON schema accurately represents the data set (both new and existing documents), our solution, named JSD Evolution, is applied only to the new documents (Figure 2 (a)), resulting in an evolved schema (Figure 2 (b)). As a result, the *_id* attribute may be of type $String$ or $Number$ (in Figure 1 (b), *_id* is of type $Number$), and a new attribute, *age*, has been added (this attribute is optional, as it is only present in the new documents).

4.3 JSD Evolution Approach

This section describes how incremental evolution of JSON schemas occurs, as illustrated in Figure 3. Although the proposed approach is based on the work of [Frozza *et al.*, 2018], modifications were made to make it incremental in the steps of generating raw schemas (new documents), aggregating raw schemas (both new and existing), and updating the RSUS (Raw Schema Unified Structure) Tree. Therefore, the yellow-highlighted parts indicate the modifications made, while the gray-highlighted parts represent the unchanged components.

The proposed approach consists of four stages. In Step 1, raw schemas are generated exclusively for the new documents inserted into the database. In contrast to the approach proposed by [Frozza *et al.*, 2018], which generates raw schemas for the entire set of documents (both new and existing). The raw schemas are stored with a unique identifier, the JSON schema ID, and assigned a new version. The assignment of a new version to the generated raw schema is performed automatically by the solution proposed in this study. For example, if this is the second evolution of the JSON schema, the version of the newly generated raw schemas will be version two. Version numbering is essential for Step 3, as only raw schemas with the most recent version are used to update the RSUS Tree. In Step 2, raw schemas from previous versions and the current version are aggregated to retain only unique raw schemas. In Step 3, the RSUS Tree generation process was modified to use the most recent RSUS Tree as its foundation, whereas the approach in [Frozza *et al.*, 2018] uses the first raw schema as the starting point. The newly generated raw schemas are applied to update the RSUS Tree based on changes that occurred. For example, in the case of the second evolution of the JSON schema, only raw schemas with version 2 will be used to update the RSUS Tree. Finally, Step 4 is responsible for transforming the consolidated RSUS Tree into a JSON schema. This stage remains unchanged from the work of [Frozza *et al.*, 2018].

Regarding intermediate data repositories, the JSD Evolution approach introduces a temporary document repository (highlighted in blue) to store the documents required for schema evolution. In this repository, an increment refers to the new documents inserted into the database that will be used to evolve the schema. The size of each increment corresponds to the number of documents contained within it and is determined by a user-defined threshold. The repositories for raw schemas, the RSUS Tree, and the JSON schema (highlighted in gray) were reused from the approach proposed by [Frozza *et al.*, 2018]. The schema evolution process is described in more detail below.

Initially, let R be the set of raw schemas generated from the document set D. The subset $R_n \in R$ represents the raw schemas derived from Dn_i, where Dn_i denotes the increment containing the newly inserted documents in D.

The incremental schema evolution process consists of four steps:

Step 1 - Generate Raw Schemas (new documents):

at this step, a raw schema R_n is generated for each new document $d \in Dn_i$. Each new raw schema R_n generated is added to R, along with the corresponding update version.

Step 2 - Aggregate Raw Schemas (both new and existing): at this step, two aggregation processes are applied to both new and existing raw schemas to retain only unique schemas. Two sets are involved: $R = r_1, r_2, ..., r_n$, which represents the set of all raw schemas, where r is a raw schema; and $A = a_1, a_2, ..., a_n$, which is the set of attributes belonging to a raw schema $r \in R$.

The first aggregation is applied to R, retaining only unique raw schemas. Then, the attribute set A of each raw schema $r \in R$ is sorted to further reduce the number of distinct raw schemas. Finally, a second aggregation is performed on the ordered raw schemas in R, eliminating duplicate information appearing in different parts of the schema.

Step 3 - Update the RSUS Tree: at this step, the RSUS

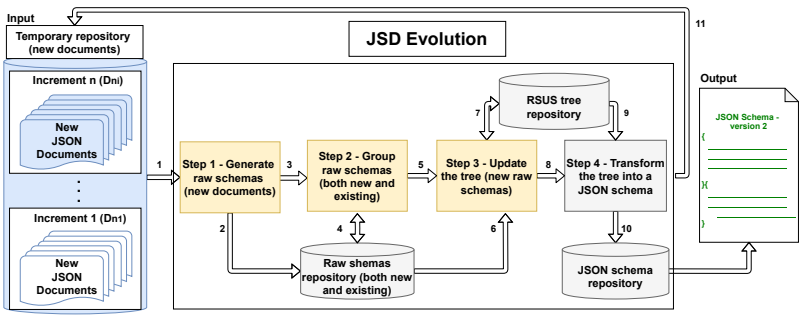


Figure 3. Flow of incremental schema evolution

Tree of the schema being updated is used as the base (T_{base}), and new raw schemas (R_n) are incorporated to refine it. The RSUS Tree is a hierarchical JSON-based structure whose root node represents the unified schema. It contains five field types: Field Type, Object Type, Primitive Type, Extended Type and Array Type. Each type has attributes such as `name`, `path`, `count` and `types`, depending on the field category. These field types collectively describe the occurrences, hierarchy and data type information of attributes present in the document collection.

Algorithm 1 (Figure 4) takes as input the base RSUS Tree (T_{base}), which will be updated, and the set of new unique raw schemas (R_n) obtained from Step 2, returning the updated tree ($T_{updated}$). The T_{base} structure has two main properties: `count`, which indicates the total number of field occurrences, and `fields`, which stores the list of attributes or subfields, each with its path (`path`), associated types, and count. In the pseudocode, `rootSchema` (line 13) represents the reference to the root node of the RSUS being updated, and `buildRawSchema` (line 31) is responsible for inserting new fields into the structure while preserving the hierarchy. The algorithm validates the inputs (lines 4–10), checking whether R_n is a valid array and whether T_{base} is defined and contains fields; otherwise, it initializes a new RSUS Tree. It then references the base structure (lines 12–13), updates existing fields (lines 15–24), and inserts the new fields from R_n (lines 26–31), finally returning the updated structure ($T_{updated}$).

Step 4 - Generate JSON Schema: in this step, the RSUS Tree is transformed into a JSON schema that represents the database.

$E_{\text{final}} = \text{TransformTreeIntoJSON}(\text{T}_{\text{updated}})$

In this expression, E_{final} is the updated JSON schema that represents the monitored collection. The function TransformTreeIntoJSON converts the updated RSUS Tree into a JSON schema, while T_{updated} is the RSUS Tree updated with the new raw schemas.

5 Experimental Evaluation

This section describes the experimental methodology employed to assess the efficiency of incremental JSON (JavaScript Object Notation) schema evolution and the batch approach proposed by [Frozza *et al.*, 2018]. Furthermore, the similarity between the JSON schemas generated and updated by the JSD (batch) and JSD Evolution approaches, respectively, was analyzed. To perform this verification, the

```
1.function UpdateRSUSWithNewRawSchemas(Rn, Tbase):
    Input: Tbase (base RSUS tree), Rn (new raw schemas)
    Output: Tupdated (updated RSUS tree)
2.  Begin
3.  // Input validation
4.  if (Rn is not an array) then
5.      Throw an error 'Rn is not an array'
6.  end if
7.  if (Tbase is undefined or
8.     does not contain valid fields) then
9.      // Create a new empty tree
10      Tbase = { fields: [], count: 0 }
11.  end if
12. // Keep a reference to the current root schema
13. Tupdated = rootSchema
14. Tupdated.count += Tbase.count
15. // Update the existing tree with the fields from Tbase
16. for each (matchingField in Tbase.fields) do
17.     existingField = find field in Tupdated.fields
18.          where field.path equals matchingField.path
19.     if (existingField exists) then
20.         // update its count
21.         existingField.count += matchingField.count
22.     else
23.         // add it to the existing tree
24.         Tupdated.fields.push(matchingField)
25.     end if
26. end for
27.     // Add the new raw Rn to the existing tree
28. for each (r in Rn) do
29.     parsedID = parse r._id
30.     parsedValue = parse r.value
31.     buildRawSchema(parsedID, parsedValue,Tupdated.fields)
32.     Tupdated.count += parsedValue
33. end for
34. return Tupdated
35.end function
```

Figure 4. RSUS tree update algorithm

DeepDiff[4] library in Python was utilized, which accounts for variations in attribute positions and types. The experiments were conducted on a computer equipped with an Intel Core i7-6700 processor and 32 GB of RAM.

5.1 Datasets

To evaluate our approach, three datasets were used: Books, Twitter, and VK, all obtained from the Kaggle repository. The latter two were also used in [Baazizi *et al.*, 2019]. The Twitter dataset[5] contains tweets related to the 2018 Russian elections. The VK dataset[6] includes documents describing user interactions on the Russian social network during the 2018 Russian elections. Finally, the Books dataset[7] comprises documents containing book metadata. A summary of the datasets is presented in Table 2.

[4]https://pypi.org/project/deepdiff/

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

[6]https://www.kaggle.com/datasets/borisch/russian-election-2018-vkcom-user-activity

[7]https://www.kaggle.com/datasets/opalskies/large-books-metadata-dataset-50-mill-entries

Table 2. Description of datasets

Feature/Dataset	Twitter	VK	Books
Size	11.5 GB	5.6 GB	82.2 GB
Number of objects	1,945,365	3,036,654	47,015,693
Avg Text Length per Document	5.67 KB	1.80 KB	1.76 KB
Number of Required Attributes	885	159	32
Number of Optional Attributes	234	97	1
Number of Attributes	high	medium	low

5.2 Evaluation Metrics

The metric used to assess the performance of the incremental schema evolution approach (JSD Evolution) and the batch approach (JSD) is processing time. The execution time required by JSD to generate a JSON schema representing the entire dataset (including both new and existing documents) was recorded. Additionally, the execution time of JSD Evolution was measured to update the existing JSON schema while considering only the newly inserted documents in the collection.

In this context, the execution times of both approaches are used to address the research question: "What is the impact of incremental schema evolution compared to the batch approach [Frozza *et al*., 2018] in terms of efficiency?"

5.3 Preparation of Data Sets

To conduct the tests for both incremental and batch schema evolution, distinct data preparation approaches were developed. The goal was to evaluate the performance of these approaches in evolving JSON schemas under different conditions, where experiments may vary in terms of the dataset used, the size of increments (number of new documents), and the number of attributes per document (e.g., low, medium, or high attribute count). The following sections detail the steps involved in these preparations.

5.3.1 Data Preparation for the Incremental Evolution Approach

For incremental evolution testing, a Geometric Progression (GP) was used to partition the data, representing the new documents used to evolve the schema. The first partition was used to create the initial schema, while subsequent partitions were employed to update the current schema. These partitions reflect different stages of document growth. For instance, this setup allows the observation of how the initial schema evolves as more data is added and how this evolution impacts processing time. The use of a GP facilitates a gradual distribution of data into partitions, enabling an effective evaluation of the impact of incremental updates under different data growth scenarios.

To determine the first term (Dn_1) and the common ratio (r) of the geometric progression (GP) used for data partitioning, only integer values were adopted. This constraint ensured that each partition contained an exact, whole number of documents. In the case of the Twitter dataset, the progression was configured with a first term of 450 ($Dn_1 = 450$) and a common ratio of 2, simulating a scenario of gradual document growth. The partitioning strategy considered both the total number of documents in the dataset and the average number of attributes per document. The cumulative sum of the first 12 terms in this progression, represented by Dn_i where i ranges from 1 to m (with $m = 12$), resulted in 1,872,500 documents (see Table 2), which closely approximates the actual size of the dataset. For the VK dataset, the same GP values as those used for partitioning the Twitter dataset were adopted ($Dn_1 = 450$ and $r = 2$). These partitioning strategies enable an analysis of how incremental and batch schema evolution approaches behave when handling datasets with the same number of documents but varying attribute densities — high for Twitter and medium for VK.

In the case of the Books dataset, a GP with a first term of 10,000 ($Dn_1 = 10,000$) and a common ratio of two was used, reflecting a progressive increase in document count. The GP values were selected based on the nature and size of each dataset, aiming to ensure a representative simulation of document growth in each collection.

5.3.2 Data Preparation for the Batch Approach

A dataset merging strategy was adopted to reflect the nature of batch processing, in which updates consider all accumulated documents up to that point (both new and existing documents). This dataset merging strategy was applied to the collections used in the experiments for the incremental schema evolution approach.

Let P_i be the partition set i, where i varies from 1 to m, with m being the total number of partitions (GP partitions). Then, for each partition P_i, we have:

- P_1: Initial partition containing Dn_1, where Dn_1 is the first term of the GP (representing the first partitioned dataset used for incremental evolution testing);
- P_2: $P_1 \cup Dn_2$, where Dn_2 is the second term of the GP (representing the second partitioned dataset used for incremental evolution testing);
- P_{i+1}: $P_i \cup Dn_{i+1}$, where Dn_{i+1} is the next term of the GP.

Each partition represents the union of the documents from previous partitions, simulating the gradual and cumulative growth of the dataset in batch evolution scenarios.

5.4 The JSD Evolution Controller and Input Configuration in the Proposed Approach

The JSD Evolution Controller and input configuration play crucial roles in conducting the experiments, as they are essential for generating the workload required for our approach. The Controller monitors the insertion of new documents into the collection and verifies whether the number of newly added documents has reached or exceeded the predefined user-defined threshold. This threshold, set based on the number of new documents in the collection, determines whether a schema update is necessary.

Figure 5 illustrates the input configuration and the operation of the JSD Evolution Controller. This process involves identifying the essential data required to initiate monitoring

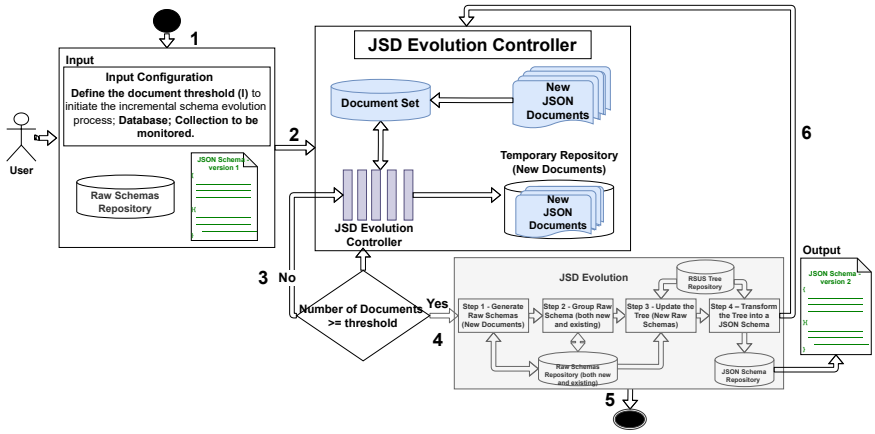


Figure 5. JSD Evolution Controller: used to monitor the insertion of documents into the document collection

and updating, including the input of new documents, the latest version of the JSON schema, the raw schemas generated in previous versions, and the document threshold that triggers the update (defined by the user). The new documents are temporarily stored in a repository and utilized during the schema evolution process (JSD Evolution). The source code for JSD Evolution, along with the controller and test scripts, is available on GitHub: https://github.com/Eleonilia/JSD-Evolution.

6 Experimental Results

A total of 36 performance experiments were conducted, varying the dataset, the size of increments, and the number of attributes per document, as described in Sections 5.3.1 and 5.3.2. Each experiment was repeated 10 times, and the average results were considered. Figure 6 illustrates the behavior of each approach in experiments conducted on the Books, Twitter, and VK datasets. The x and y axes represent the average execution time and the number of documents used in the experiments, respectively. Figure 6 (a) presents the performance of the approaches in experiments conducted on the Books metadata dataset, where documents contain a small number of attributes. The results indicate that the JSD Evolution solution is, on average, 1.93 times faster than the batch approach proposed by [Frozza *et al.*, 2018]. Similarly, experiments conducted on the Twitter dataset (Figure 6 (b)), which contains a high number of attributes per document, also confirm that our solution is more efficient than the batch approach, achieving an average speedup of 2.85 times.

In the experiments involving a moderate number of attributes (Figure 6 (c)), conducted on the VK dataset, it was observed that for experiments with up to 6,750 documents (experiments 1 to 4), the batch approach achieved a lower execution time compared to our solution, being on average 1.71 times faster. However, as the number of documents increased (experiments 5 to 12), our solution demonstrated an advantage in execution time, evolving the schema 1.77 times faster.

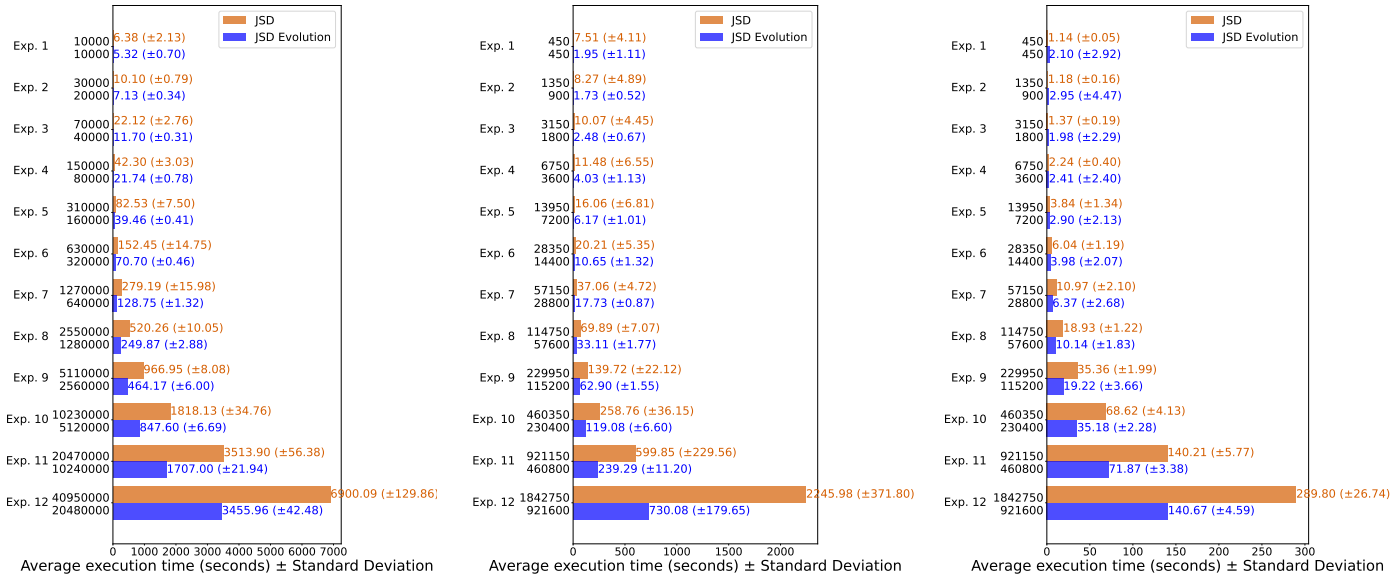
In addition to the efficiency analysis, a functional analysis was performed to assess the ability of the JSD (batch) and JSD Evolution approaches to correctly identify various features present in JSON (JavaScript Object Notation) schemas. Eight datasets created by [Latták and Koupil, 2022] were used, each named to reflect its content and focusing on a specific JSON schema feature, including *PrimitiveTypes, SimpleArrays, SimpleObjects, ComplexArrays, ComplexObjects, Optional, Union*, and *References*. The experiments showed that both approaches produced similar results in the generation and updating of JSON schemas, differing only in the position of certain attributes and types. This difference can be attributed to the fact that, in step 3, the JSD Evolution approach uses the RSUS (Raw Schema Unified Structure) Tree as a base, evolving it with newly generated raw schemas. In contrast, in step 3 of the JSD approach, the first generated raw schema serves as the starting point for RSUS Tree construction, and all subsequent raw schemas (both new and existing) are used for its update. When analyzing the functional behavior of the approaches, it was observed that, despite the adaptations required to enable incremental schema evolution, the JSD Evolution approach was able to correctly identify all tested JSON schema features.

The performance differences can be attributed to the way each approach manages database insertions. The batch approach, which requires reapplying the entire JSON schema generation process to the full set of documents (including both newly added and previously stored documents), shows a significant increase in execution time as the document count increases. This occurs because the batch approach does not differentiate between newly added and previously stored documents, necessitating the reprocessing of the entire document set during each schema update.

In contrast, the proposed solution proves to be more efficient in this regard. By processing only newly inserted documents, it can evolve the schema efficiently, resulting in significantly lower execution times, especially in experiments with a large number of documents. These results highlight the advantage of our solution for schema evolution in large datasets, reducing processing time by focusing exclusively on new documents.

6.1 Discussion of Results

This section presents the conclusions drawn from the analysis of the experimental results, emphasizing the impact of incremental schema evolution (JSD Evolution) in comparison



(a) Books **(b) Twitter** **(c) VK**

Figure 6. Comparison of the average execution time (in seconds) ± standard deviation of the JSD Evolution and JSD (batch) approaches for JSON schema evolution.

with the batch (JSD) approach proposed by [Frozza *et al.*, 2018]. The analysis encompasses a descriptive assessment of the execution times of the JSD and JSD Evolution approaches across the Books, Twitter, and VK datasets, the influence of the number of attributes per document on processing time, and the relationship between file size and execution time. Furthermore, the stages of the JSD Evolution approach that demanded the greatest processing effort were identified, and a comparison was conducted between the schemas generated by the batch approach (JSD) and those updated by the incremental approach, with the aim of assessing their structural and semantic similarity. Finally, the main threats to the validity of the results are discussed, taking into account factors that may limit their generalization and applicability.

Next, the results for each of these aspects are presented, providing a detailed analysis of the performance and scalability of the proposed approach.

Descriptive analysis of the execution time of the JSD and JSD Evolution approaches for the Books, Twitter, and VK datasets.

This descriptive analysis focuses on the standard deviation of execution times for the JSD and JSD Evolution approaches across the Books, Twitter, and VK datasets. In the Books dataset, the standard deviations ranged from 0.79 to 129.86 seconds for JSD, whereas JSD Evolution exhibited lower values, ranging from 0.31 to 42.48 seconds Figure 6 (a)). For the Twitter dataset, JSD produced higher standard deviations (4.11 to 371.80 seconds) compared to JSD Evolution (0.52 to 179.65 seconds), indicating greater consistency in the evolutionary approach (Figure 6 (b)). Figure 6 (c) presents the standard deviations observed in the VK dataset, where the JSD approach exhibited considerable variation, ranging from 0.05 to 26.74 seconds. In contrast, the JSD Evolution approach generated standard deviations between 1.83 and 4.59 seconds, demonstrating lower variability in execution times. These results highlight the greater consistency of the incremental approach (JSD Evolution) compared to the batch-based JSD approach.

The results indicate that, in the experiments conducted with the VK dataset, the standard deviation exhibited significantly lower variation, demonstrating greater stability compared to the Books and Twitter datasets. When analyzing the results of the experiments with the Books and Twitter datasets, we observed that the standard deviation was higher in the Twitter dataset. These findings suggest that the stability of the analyzed metric varies depending on the dataset, with the VK dataset exhibiting the highest consistency.

The Influence of the number of attributes per pocument on processing Time.

In this study, we evaluated the efficiency of the JSD Evolution approach using two datasets: Twitter, characterized by a high number of attributes per document (885 mandatory and 234 optional), and VK, with a moderate number of attributes (159 mandatory and 97 optional). The objective was to compare the execution times of JSD Evolution in both collections, analyzing how the number of attributes per document affects performance as the number of documents increases. The same analysis was conducted for the JSD approach applied to the Twitter and VK datasets.

The results presented in Figure 7 indicate that, in experiments with up to 900 documents (Exp. 1 and Exp. 2), the number of attributes per document did not significantly impact the execution time of JSD Evolution. Within this range, the Twitter dataset was, on average, 1.36 times faster than the experiments conducted with the VK dataset. However, as the number of documents increased (from Exp. 3 to Exp. 12), the impact became evident: the execution time for experiments conducted with the VK dataset was, on average, 2.89 times faster than for the Twitter dataset. Conversely, the results of the JSD approach, illustrated in Figure 8, show that in all experiments, the execution time of JSD applied to the VK dataset was, on average, 5.03 times faster than JSD applied to the Twitter dataset. These results highlight that the number of attributes per document significantly influences

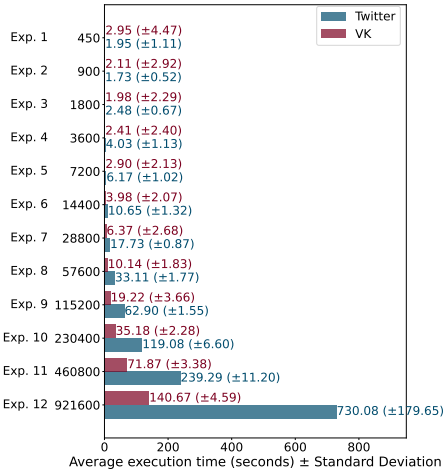


Figure 7. Comparison of the average execution time (in seconds) ± standard deviation of the JSD Evolution approach for the Twitter and VK datasets.

processing time, especially in scenarios with large volumes of documents.

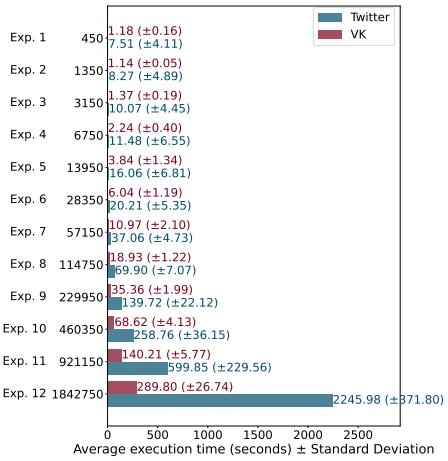


Figure 8. Comparison of the average execution time (in seconds) ± standard deviation of the JSD approach for the Twitter and VK datasets.

Impact of file size on processing time.

When examining the descriptive metrics of the execution time for the JSD and JSD Evolution approaches across the Books, Twitter, and VK datasets, a consistent trend is observed: as the file size increases, the average execution time also grows. This relationship was observed in all experiments conducted on the three datasets for both approaches (JSD and JSD Evolution), suggesting that the dataset file size has a significant impact on processing time (Table 3).

Identification of the steps of the JSD Evolution approach that require the most Time to complete.

To identify the stages that consume the most time in the JSD Evolution approach, an execution time analysis was conducted for each stage across different datasets (VK, Twitter, and Books). The analysis revealed that Stage 1, responsible for generating raw schemas from new documents, was the most time-consuming, accounting for an average of 96.39% of the total execution time. This was followed by the schema aggregation stage (new and existing schemas), whose purpose is to ensure that only unique schemas result from the aggregation process (Stage 2). On the other hand, Stage 3, which updates the RSUS tree with the new raw schemas, and Stage 4, which converts the updated RSUS tree into a JSON schema, had significantly lower execution times. The high time demand in Stage 1 is likely due to the fact that, in this stage, a raw schema is generated for each new document inserted into the collection. This process involves maintaining the original structure of the JSON documents and replacing the primitive values of each attribute with the corresponding data types in JSON and extended JSON, making it a detailed and computationally intensive process.

Comparison Between the Schemas Generated and Updated by the JSD and JSD Evolution Approaches.

A complementary analysis was conducted to assess the similarity between the schemas generated or updated by the JSD and JSD Evolution approaches, applied to the Books, Twitter, and VK datasets. For this comparison, the DeepDiff library from the Python language was employed, as it enables the identification of structural differences between JSON objects while disregarding aspects such as attribute or type ordering. The results showed that, although the schema construction procedures differ — JSD performs full schema generation based on the entire collection, whereas JSD Evolution incrementally updates the schema using only newly inserted documents — the final schemas exhibited a high degree of similarity. The only differences observed concerned the ordering of certain attributes and types within the JSON schema, which do not affect its representational capacity or validity.

Threats to Validity.

Although the experiments were conducted using real datasets from Kaggle (Books, Twitter, and VK), the generalization of the results to other domains or data sources cannot be assured. The Twitter and VK datasets exhibit high structural diversity and representativeness, including deeply nested documents, polymorphic structures, optional attributes, and fields that may assume multiple types (e.g., the description attribute may be either null or a string). However, the specific influence of these structural features, as well as scenarios involving document versioning, was not evaluated in isolation.

Furthermore, the datasets were partitioned using a geometric progression to simulate the insertion of new documents. While this strategy enables controlled experiments, it may not fully reflect real-world insertion patterns, particularly in dynamic domains such as social media, where data growth often follows irregular and bursty trends. Other factors, such as the potential overfitting to specific JSON structures and sensitivity to parameter settings (e.g., thresholds), may also influence the results. These aspects should be considered when applying the proposed approach in different scenarios and may indicate directions for future research.

7 Conclusion and Future Work

In this paper, we proposed an approach for the incremental evolution of JSON schemas, leveraging only newly inserted documents in the collection. The primary motivation is to contribute to areas characterized by continuous change and high data flow (e.g., Social Networks and E-Commerce), ensuring that schemas remain updated while reducing execution time, without the need to apply the approach to all

Table 3. Correlation Between File Size and Mean Execution Time (MET) – JSD Evolution and JSD

Experiments (Exp.)	JSD Evolution						JSD					
	book		Twitter		VK		book		Twitter		VK	
	File size	MET	File size	MET	File size	MET	File size	MET	File size	MET	File size	MET
1	14.04 MB	5.32	741 MB	1.95	397.31 KB	2.95	14.04 MB	6.38	741 MB	7.51	397.31 kB	1.18
2	27.76 MB	7.14	1.44 MB	1.73	761 kB	2.11	42.22 MB	10.10	2.17 MB	8.27	1.13 MB	1.14
3	53.53 MB	11.70	2.91 MB	2.48	1.6 MB	1.98	94.31 MB	22.12	5.07 MB	10.07	2.70 MB	1.37
4	101.91 MB	21.74	5.74 MB	4.03	3.24 MB	2.41	195/20 MB	42.30	10.76 MB	11.48	6.28 MB	2.24
5	187.45 MB	39.46	11.55 MB	6.17	7.29 MB	2.90	382/77 MB	82.53	22.18 MB	16.06	13.67 MB	3.84
6	346.42 MB	70.70	22.91 MB	10.65	15.04 MB	3.98	732.41 MB	152.45	44.94 MB	20.21	26.77 MB	6.04
7	634.82MB	128.75	45.01 MB	17.73	32.50 MB	6.37	1.37 GB	279.19	89.60 MB	37.06	61.72 MB	10.97
8	1.2 GB	249.87	90.13 MB	33.11	68.63 MB	10.14	2.58 GB	520.26	179.17 MB	69.90	129.39 MB	18.93
9	2.16 GB	464.17	186.14 MB	62.90	132.15 MB	19.22	4.73 GB	966.95	365.04 MB	139.72	259.87 MB	35.36
10	3.77 GB	847.60	379.73 MB	119.08	247.09 MB	35.18	8.50 GB	1,818.13	743.88 MB	258.76	506.55 MB	68.62
11	7.96 GB	1,707.00	757.28 MB	239.29	506.07 MB	71.87	16.56 GB	3,513.90	1.50 GB	599.85	1.00 GB	140.21
12	19.84 GB	3,455.96	1.49 GB	730.08	991.04 MB	140.67	33.38 GB	6,900.09	2.99 GB	2,245.98	1.97 GB	289.80

MET: Mean Execution Time, measured in seconds.

documents (both new and previously stored). Experimental results show that the proposed solution was, on average, 1.93 and 2.85 times faster than the approach of Frozza et al. [Frozza *et al*., 2018] for the book metadata and Twitter datasets, respectively. For the VK dataset, the proposed solution demonstrated lower execution times in experiments 5 to 12, as the number of documents increased. It is important to highlight that Step 1, which is responsible for generating raw schemas, was the most computationally expensive, accounting for 96% of the total execution time. Furthermore, the analysis revealed that the number of attributes per document significantly influences execution time, with this effect becoming more significant in larger datasets.

Future work includes extending the proposed approach to support the removal, update, and reorganization of documents or attributes. To this end, the JSD Evolution controller developed in this study will be employed to detect such changes, which will require update mechanisms that ensure schema consistency and compliance with the JSON Schema specification. Additionally, we aim to enhance the efficiency of the first step of the approach, identified as the main performance bottleneck, through the application of parallel and distributed processing techniques. This strategy is expected to reduce execution time and improve the scalability of the solution, particularly in scenarios involving large volumes of data and high structural heterogeneity.

References

Abdelhédi, F., Rajhi, H., and Zurfluh, G. (2022). Extraction process of the logical schema of a document-oriented nosql database. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2022)*, pages 61–71. DOI: 10.5220/0010899000003119.

Atmatzides, N., Bedo, M., and de Oliveira, D. (2022). Adoção de sgbds nosql em empresas brasileiras: um levantamento preliminar. In *Anais do XXXVII Simpósio Brasileiro de Bancos de Dados*, pages 385–390, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/sbbd.2022.226015.

Baazizi, M.-A., Colazzo, D., Ghelli, G., and Sartiani, C. (2019). Parametric schema inference for massive json datasets. *The VLDB Journal*, 28:497–521. DOI: 10.1007/s00778-018-0532-7.

Cánovas Izquierdo, J. L. and Cabot, J. (2013). Discovering implicit schemas in json data. In *International Conference on Web Engineering, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings 13*, pages 68–83. Springer. DOI: 10.1007/978-3-642-39200-9_8.

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 *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 356–363. DOI: 10.1109/IRI.2018.00060.

Karim, L. and Boulmakoul, A. (2021). Trajectory-based modeling for fraud detection and analytics: Foundation and design. In *2021 IEEE/ACS 18th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–7. IEEE. DOI: 10.1109/AICCSA53542.2021.9686920.

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)*, pages 2462–2471. IEEE. DOI: 10.1109/BigData.2017.8258204.

Latták, I. V. and Koupil, P. (2022). A comparative analysis of json schema inference algorithms. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 379–386. DOI: 10.5220/0011046000003176.

Li, X., Sun, L., Ling, M., and Peng, Y. (2023). A survey of graph neural network based recommendation in social networks. *Neurocomputing*, 549:126441. DOI: 10.1016/j.neucom.2023.126441.

Purnomo, Y. J. (2023). Digital marketing strategy to increase sales conversion on e-commerce platforms. *Journal of Contemporary Administration and Management (ADMAN)*, 1(2):54–62. DOI: 10.61100/adman.v1i2.23.

Rodrigues, E., Pires, C. E., and Filho, D. N. (2024). Evolução incremental de esquemas de banco de dados orientado a documentos. In *Anais do XXXIX Simpósio Brasileiro de Bancos de Dados*, pages 260–273, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/sbbd.2024.240265.

Sevilla Ruiz, D., Morales, S. F., and García Molina, J. (2015). Inferring versioned schemas from nosql databases and its applications. In *Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings 34*, pages 467–480. Springer. DOI: 10.1007/978-3-319-25264-3_35.