Schema Management for Document Stores

Lanjun Wang[†], Oktie Hassanzadeh[§], Shuo Zhang[†], Juwei Shi[†],
Limei Jiao[†], Jia Zou, and Chen Wang[‡] [*]

{wangljbj, shuozh, jwshi, jiaolm}@cn.ibm.com, hassanzadeh@us.ibm.com,
Jia.Zou.US@ieee.org, wang_chen@tsinghua.edu.cn

[†]IBM Research - China [§]IBM T.J. Watson Research Center [‡]Tsinghua University

ABSTRACT

Document stores that provide the efficiency of a schema-less interface are widely used by developers in mobile and cloud applications. However, the simplicity developers achieved controversially leads to complexity for data management due to lack of a schema. In this paper, we present a schema management framework for document stores. This framework discovers and persists schemas of JSON records in a repository, and also supports queries and schema summarization. The major technical challenge comes from varied structures of records caused by the schema-less data model and schema evolution. In the discovery phase, we apply a canonical form based method and propose an algorithm based on equivalent sub-trees to group equivalent schemas efficiently. Together with the algorithm, we propose a new data structure, eSiBu-Tree, to store schemas and support queries. In order to present a single summarized representation for heterogenous schemas in records, we introduce the concept of "skeleton", and propose to use it as a relaxed form of the schema, which captures a small set of core attributes. Finally, extensive experiments based on real data sets demonstrate the efficiency of our proposed schema discovery algorithms, and practical use cases in real-world data exploration and integration scenarios are presented to illustrate the effectiveness of using skeletons in these applications.

1. INTRODUCTION

In the era of cloud computing, application developers are deviating from data-centric application development paradigms relying on relational data models and moving to agile and highly iterative development approaches embracing numerous popular NoSQL data stores [26]. Among various NoSQL data stores, document stores (also referred to as document-oriented data store) such as MongoDB [22] and Couchbase [7] are among the most popular options. These data stores support scalable storage and retrieval of data encoded in JSON (JavaScript Object Notation) or its variants [19], in a hierarchical data structure as illustrated in Fig. 1.

[*]This work has been done when Jia Zou and Chen Wang were with IBM Research-China.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 9
Copyright 2015 VLDB Endowment 2150-8097/15/05.

Using JSON is particularly attractive to developers as a natural representation of object structures defined in application codes in object-oriented programming (OOP) languages. The flexibility of its structure also allows users to work with data without having to define a rigid schema in prior, as well as to manage ad hoc and changing data with evolving schemas [3, 20]. These features significantly simplify the interaction between the application and document stores, resulting in less code as well as ease of debugging and code maintenance [27]. JSON is also widely adopted as a data exchange format, which is used by major Web APIs such as Twitter [16], Facebook [17] and many Google services [18].

Conversely, the simplicity developers achieved from using JSON and document stores leads to difficulties in certain data management tasks, which are probably beyond their duties. Let's consider following scenarios: a data scientist wants to explore an application's data for analytic purposes; or an application is required to share its data with another application; or a database administrator wants to enable fine-grained access control; or data are required to be integrated into a data warehouse. All of these tasks are usually facilitated by schemas. However, since the design and specification of data structures are tightly coupled with data in JSON, and unlike other semi-structured data (e.g. XML) that are usually associated with an explicit schema, these tasks have to request developers' assistance or study design documents or even read source codes to understand the schema. Unfortunately, neither of these solutions are practical in real world. As a result, there is a need for a schema management function for document stores, like an RDBMS's data dictionary which extracts schema definitions from DDL, stores them in a repository, and enables exploration and search through query interfaces (e.g. "select * from user_tables" in ORACLE) [2] or commands/tools (e.g. "DESCRIBE table" function in DB2) [1].

Given that the schema-less nature and abandoning of schema-related APIs are main reasons that developers use document stores, it is not feasible to enable schema management through a change of interfaces and APIs or enforcing a data model. Therefore, what we need is a new schema management framework that can work on top of any document store, along with new schema retrieval and query mechanisms different from those implemented in RDBMSs.

The first challenge in designing a schema management framework for document stores is in discovery of the schema from stored data. Without an explicit definition of the schema, the need for schema retrieval from a document store results in a complex discovery process instead of a simple lookup. Due to lack of constraints to guarantee only one object type in a single collection[1],

[1]Different document stores use different terms referring to the equivalent of a "table" in RDBMSs. To simplify the presentation, in this paper, we use MongoDB's "collection" to refer to such "table"-like units of data in document stores.

a collection may contain records corresponding to more than one object type. Moreover, schemas of the same object type in a collection might also vary because of attribute sparseness in NoSQL as well as data model evolution caused by highly interactive adoption and removal of features. For example, in a real-world scenario using DBpedia [8] (a knowledge graph retrieved in JSON from its Web API as described in Sec. 7), the 24,367 records describing objects of type "company" have 21,302 different schemas. Note that a simple sampling of the records to examine their schemas would fail to capture the full schema because almost every record may have a distinct schema. Hence, the first problem we study is how to efficiently discover all distinct schemas appearing in all records. The importance for the efficiency of the schema discovery is more evident for online inserts where schemas of records are to be identified incrementally and the schema repository is to be updated in real time. To tackle this challenge, we propose a new data structure for both schema discovery and storage, eSiBu-Tree, and an equivalent sub-tree based algorithm to discover schemas from existing data as well as an online method for new inserts.

The second challenge we face is implementation of a query interface over the schema repository, similar to that of RDBMSs, which is essential for understanding the schema of a given data source. In this paper, we start with two basic queries on checking the existence of a given schema and the existence of a specified sub-structure on a finer granularity (e.g., an attribute "*root* $\rightarrow$ *author* $\rightarrow$ *name*" in Fig. 1). In the example as shown in Fig. 1, suppose developer "A" has created a collection named "article" for blog data. Despite of the flexibility of the schema-less system, developer "B" is still expected to conduct a pre-checking on the given schema to determine the right place to persist such type of data rather than creating a new collection "blog", and also persist data in a consistent way in a single collection but not separate "article" and its nested body "author". As mentioned above, nowadays these checking tasks primarily rely on developers' familiarity with the structure of data, or reading design documents or even codes, but this task could be simplified with the enabling of query functions. In this paper, we study how to support these two types of queries over eSiBu-Tree efficiently.

Finally, because of the heterogeneity in schemas, the querying functionality is not enough for more advanced data exploration scenarios over document stores. Considering the above example of "company" in DBpedia, if a data scientist wants to explore the company data's structure for feature selection, to display a single schema is a more preferable solution than to show all the 21,302 schemas at the same time. Therefore, the challenge comes to how to present varied schemas in a data model. Simple solutions such as using the intersection or union of all schemas do not work well in practice. In the "company" example, the intersection of all schemas is only one attribute (*root* $\rightarrow$ *uri*) whereas there are 1,648 attributes in the union set, among which more than half appear only once in records. What we need here is a balanced solution in the middle of the intersection which may miss prominent attributes and the union which may provide too many attributes that are less informative. In this case, we introduce a new concept called "skeleton", and propose to use it as a relaxed form of the schema. Moreover, we design an upper bound algorithm for efficient skeleton construction.

In summary, this paper makes the following contributions:

- We propose a framework for schema management over document stores. To the best of our knowledge, this is the first schema management framework for document stores.
- We propose a new data structure, eSiBu-Tree, to retrieve and store schemas and support queries, as well as an equivalent sub-tree based algorithm to discover all distinct record schemas in both batch and incremental manners.
- A concept "skeleton" is introduced to summarize record schemas. For efficient skeleton construction, an upper bound algorithm is designed to reduce the scale of candidates.
- We evaluate the performance of eSiBu-Tree and algorithms for discovery and query, which outperforms a baseline method based on the notion of "canonical forms". We also provide two real case studies on data exploration and integration to evaluate the effectiveness of using skeletons in these applications.

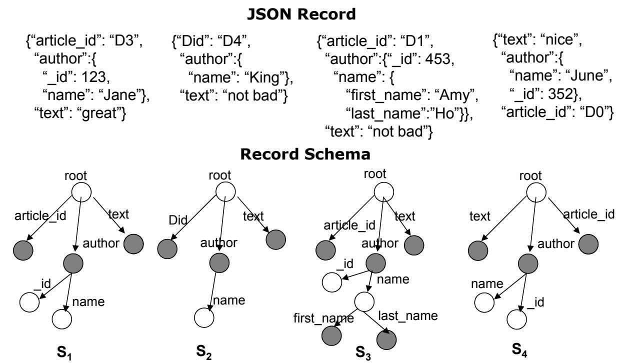


Figure 1: JSON Records from the collection "article" and their Record Schemas

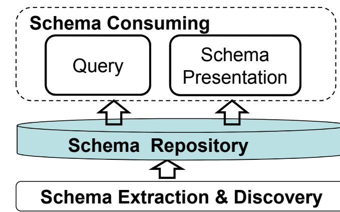


Figure 2: Schema Management Framework

The rest of this paper is organized as follows. The schema management framework is described in Sec. 2, and related preliminaries are presented in Sec. 3. Sec. 4, 5 and 6 present technical details of our solution to address the above mentioned challenges. We present experimental evaluations in Sec. 7. Sec. 8 discusses related work and Sec. 9 concludes this paper.

2. SCHEMA MANAGEMENT FRAMEWORK

Our framework of schema management is designed for document stores, which includes three components as shown in Fig. 2, schema extraction and discovery component, repository component, and schema consuming component with two functions of query and presentation.

Schema Extraction and Discovery This component provides a transparent way to discover all schemas in records. The first function this component provides is extraction of the schema of each record from input JSON records. Fig. 1 shows four JSON records and their corresponding schemas which we call *record schemas*. For existing data, this component discovers all distinct record schemas by grouping the equivalent ones into categories. For a new record, its record schema is compared with the current existed record schemas, and persisted immediately if it is a new structure. In this study, we apply a method based on the notion of canonical forms [6], and propose a novel hierarchical algorithm for schema discovery.

Table 1: Notations

r	record	$\mathbf{R}$	record set
S, S_i	record schema	$\mathbf{S}$	record schema set
v	node	V	node set
e	edge	E	edge set
$V(l)$	set of nodes in l-th level	L_{max}	maximum level of S
x, y	attribute	X, Y	attribute set
M	attribute universal set		
K	skeleton	K_c	candidate skeleton

Schema Repository This component is responsible for schema persistence of document stores, and also supports the efficient extraction process and schema consuming. A new data structure, eSiBu-Tree, is proposed for the repository in this study.

Query The schema consumption component provides the functionality to find exact answer to certain types of queries on schemas. Our implementation includes two types of existence queries, namely schema existence and attribute existence.

Schema Presentation Given the variety of record schemas in a collection, this functionality aims to provide a summarized representation of schemas. Our implementation is based a concept "skeleton" for JSON records, which is a parameter-free method to display core attributes in a concise format.

3. PRELIMINARIES

We first list notations used in this paper as shown in Table 1.

According to the JSON grammar [19], a JSON object is built on a collection of *name/value* pairs. A *value* can be an atomic value (e.g., string or number), another object, or an array (ordered list) of *value*s. Following previous work on semi-structured schemas [12, 13, 23], we represent the structure of a JSON document as a tree with its root node labelled as $root$.

In this study, a **record schema** $S = (V, E)$ consists a node set of V and an edge set of E. A node $v \in V$ is labelled with a *name* that appears in a *name/value* pair somewhere in the document. An edge $e = root \rightarrow v \in E$ is from *name/value* pair in the root of document. Together with it, an edge $e = v_1 \rightarrow v_2 \in E$ if and only if v_2 appears as a *name* in an object associated with v_1 directly or inside an array. For an edge $e = v_1 \rightarrow v_2$, v_1 is the parent node of v_2, and v_2 is a child node of v_1. Fig. 1 shows four example record schemas. All of them are extracted from a data set describing the object type of "article" for a blog.

Two record schemas $S = (V, E)$ and $S' = (V', E')$ are called **equivalent** if and only if $V = V'$ and $E = E'$. For example, S_1 and S_4 in Fig. 1 are equivalent record schemas.

We set the level of the root node in a record schema as Level 1, and for the other nodes, its level is one more than its parent's level. The set of nodes in the l-th level is denoted as $V(l)$. The maximum level of a record schema is the largest level among leaf nodes, denoted as L_{max}. For example, the maximum level of S_1 in Fig. 1 is 3.

In a record schema, each path from the root node to a leaf node is called an **attribute**. For example, S_1 in Fig. 1 contains the following four attributes: {*root→article_id*, *root→author→_id*, *root→author→name*, *root→text*}.

As stated earlier, record schemas in a collection can vary despite describing the same object type. Besides attribute sparsity (such as S_2 in Fig. 1 does not contain *root→author→_id*), another major reason for record schema variations is attribute evolution, which refers to semantically equivalent but different formats in describing a property of the object type. In this study, we use a matching relation, denoted as $X \cong Y$, to represent such semantic equivalence of two sets of attributes. In particular, there are two kinds of attribute evolution. The first kind is the naming convention (i.e., semantic equivalence but different labels), such as {*root→Did*} ≅ {*root→article_id*}, and the second kind is the structural variation (i.e., semantic equivalence but different granularity), such as {*root → author → name*} ≅ {*root → author → name → first_name*, *root → author →name → last_name*}.

Some applications (e.g., data exploration for analytic purposes) require a single view of the data model of a collection, but record schema variations make it a non-trivial task. In order to present the data model, one can return the union of all attributes, or a ranked list of the attributes based on their occurrence frequencies, or the intersection of record schema sets across all records (i.e., those with 100% occurrence). However, as described in Sec. 1, these approaches have drawbacks (e.g., missing prominent attributes, less informative, etc.) in practice because extensive heterogeneity often presents in record schemas. We therefore define the concept of "skeleton" to approximate the essential attributes of an object type, which is loosely related to the schema definition. **Skeleton** K is the *smallest* attribute set to capture *core* attributes of the record schema set for a specific object type.

DEFINITION 1. (CANDIDATE SKELETON) *A candidate skeleton K_c of a record schema set $\mathbf{S} = \{S_1, \ldots, S_N\}$ whose attributes compose $M = \cup_{i=1}^N S_i$ meets the following three criteria:*

- *(Existence) $K_c \subseteq M$;*
- *(Uniqueness) $\forall x, y \in K_c$, then for all $X \subseteq M$ with $x \in X$ and for all $Y \subseteq M$ with $y \in Y$, there is $X \ncong Y$;*
- *(Denseness) $\forall X \cong Y$ and $freq(X) > freq(Y)$ (where $freq(X)$ is the number of records containing X), then for all $y \in Y$, there is $y \notin K_c$.*

The aim of setting these criteria for candidates is to meet the requirement of "smallest" in the skeleton by avoiding noises from attribute evolution. Furthermore, in order to select the skeleton from candidates to meet the "core" requirement, we propose a quality measure based on the trade-off between significance and redundance of attributes in record schemas.

DEFINITION 2. (QUALITY) *For a record schema set $\mathbf{S} = \{S_1, \ldots, S_N\}$, the quality of an attribute set K_c is defined as:*

$$q(K_c) = \sum_{i=1}^{N} \alpha_i G(S_i, K_c) - \sum_{i=1}^{N} \beta_i C(S_i, K_c) \quad (1)$$

where $G(S_i, K_c)$ is the gain of K_c in retrieving S_i defined as:

$$G(S_i, K_c) = \frac{|S_i \cap K_c|}{|S_i|} \quad (2)$$

and $C(S_i, K_c)$ is the cost of K_c in retrieving S_i defined as:

$$C(S_i, K_c) = 1 - \frac{|S_i \cap K_c|}{|K_c|} \quad (3)$$

Two weights α_i and β_i reflect the importance of each S_i in gain and cost respectively, which have $\sum_i \alpha_i = \sum_i \beta_i = 1$.

Eq.(2) describes the percentage of S_i's attributes existed in K_c, and Eq.(3) describes the percentage of K_c's attributes which is useless in retrieving S_i. The total quality is the weighted average on all record schemas in $\mathbf{S}$. Finally, the skeleton is defined as:

DEFINITION 3. (SKELETON) *For a record schema set $\mathbf{S} = \{S_1, \ldots, S_N\}$ whose attributes compose $M = \cup_{i=1}^N S_i$, the skeleton $K \subseteq M$ is the candidate skeleton that has the highest quality among all candidate skeletons.*

4. SCHEMA DISCOVERY

This section focuses on the schema discovery function in the framework. Our goal is to discover all distinct record schemas. The output is a specific data structure to persist record schemas in the repository.

This section offers two methods to group equivalent record schemas. The canonical form (CF-)based method (Sec. 4.1) is to group record schemas based on the same canonical form, and the method for Depth-First Canonical Form generation [6] is applied. Since the number of sorts depends on the number of non-leaf nodes, this algorithm is not efficient for grouping records constituted by multiple embedded objects. In order to speed it up, we propose a hierarchical method to assign record schemas into a category with equivalent sub-trees level by level top to down, so we call this method as equivalent sub-tree (EST-)based method. In Sec. 4.2, together with the algorithm, a hierarchical data structure called eSiBu-Tree, is proposed as the data structure for record schema persistence and query. Furthermore, Sec. 4.3 introduces how to apply above algorithms in online schema identification for new inserts. In addition, we analyze time and space complexities of them respectively in Sec. 4.4.

4.1 CF-Based Record Schema Grouping

In this study, we apply the method for generating Depth-First Canonical Form [6] to group equivalent record schemas. Since the canonical form specifies a unique representation of a labelled rooted unordered tree, equivalent record schemas can be grouped together based on the same canonical form.

Input: Record set: $\mathbf{R}$
Output: Array of code maps: $[CM_1, CM_2, \ldots]$
1: **for all** $r \in \mathbf{R}$ **do**
2: Construct the record schema S of r, and obtain L_{max} of S
3: $l \leftarrow L_{max}$
4: **while** $l > 0$ **do**
5: Encode each $v \in V(l)$ with a code and persist label-code pair in CM_l
6: Update label of each $v' \in V(l-1)$ by appending ordered codes from children of v'
7: $l - -$
8: **end while**
9: **end for**

Algorithm 1: CF-Based Record Schema Grouping

Alg. 1 processes nodes from the record schema level by level bottom up. For each node from $V(l)$, we encode it with a code based on its label (Line 5). Such label is a sequence constituted by its original label and ordered codes from its children (Line 6).

These label-code pairs compose the code map of Level l (denoted as CM_l) as shown in Fig. 3. The detailed logic of encoding nodes is as: if the label of a node exists in the CM_l, we use the corresponding code to replace the label; otherwise, we assign a new code to the label and update the CM_l by adding this new label-code pair. The purpose of mapping a label to a code is to save the space for appending children's information to their parent. In the implementation, we use the natural number in sequence as codes, and so the code map is a hash map with a sequence as key and an integer as value.

In order to ensure that nodes with the same label and the same descendants are assigned with the same code in Line 5, this algorithm updates the label of each node by combining its ordered child codes in Line 6. In our implementation, since a code is an integer, we append the ascending order of codes from children on the original parent's label. For example, for S_1 in Fig. 1, *_id* and *name* in Level 3 are encoded as 1 and 2 respectively, and then the node labelled *author* in Level 2 is updated to *author,1,2* by combining ordered codes of its children *_id* and *name* (our implementation uses

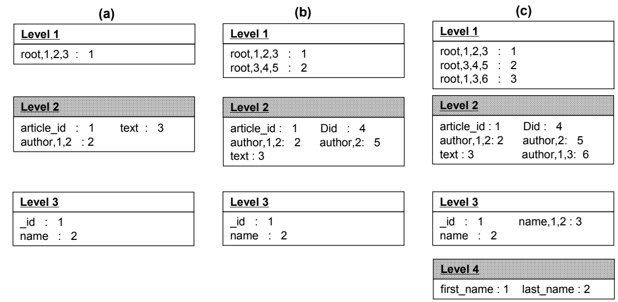


Figure 3: Examples of Generating Canonical Form

the comma as the divider). In addition, we leverage radix sort to ensure scan once on each node for sorting.

As a result, equivalent record schemas are assigned with the same code which is persisted in the code map of the root level (Level 1). Take S_1 in Fig. 1 as an example, its root node is updated as *root,1,2,3* and the whole record schema is encoded as 1. When a record with S_4 comes, the same procedure will be executed, and S_4 is also encoded as 1 because it is equivalent to S_1.

In this algorithm, with encoding nodes level by level, there is a byproduct which we call code map array, denoted as $\vec{CM} = [CM_1, CM_2, \ldots]$. From this code map array, all categories of record schemas in the collection can be retrieved. As a result, we keep it as the data structure to persist schemas of the collection.

Fig. 3 presents how the code map array is generated/updated by processing records from Fig. 1 one by one. All of these four records contain a node with a label *author* in Level 2. In S_1, it has two children *_id* and *name*, but in S_2, the attribute *_id* is absent. Thus, the code map of Level 2 is updated by adding a new label-code pair as shown in Fig. 3(b). Moreover, in S_3, the child node *name* has been evolved to *name→first_name* and *name→last_name*, so the code map of Level 2 is updated by adding another new label-code pair as shown in Fig. 3(c), together with a code map created in Level 4. For the S_4, since it is equivalent to S_1, there is no expansion on any code map. Finally, there are three codes in the root level, which implies these four records have three distinct record schemas.

In addition, the most time-consuming part of Alg. 1 is sorting children codes for updating the label of parent (Line 6). The number of sorts depends on the number of non-leaf nodes. Thus, the performance decreases when a record is constituted by a lot of embedded objects. Moreover, in our implementation, we leverage radix sort whose time complexity is $O(|CM_l|)$, where $|CM_l|$ is the size of the code map (i.e., radix size). When $|CM_l| \approx |v|$ (where $|v|$ is the number of child nodes to sort), the radix sort is faster than quick sort. When the $|CM_l| \gg |v|$, the radix sort becomes useless. In the schema discovery, with consuming more and more different record schemas, the size of a code map is increasing, so the radix sort is limited to be used to improve the sorting performance. Therefore, we have to consider a method to reduce the number of sorts as well as the size of code maps in order to make the grouping more efficiently.

4.2 eSiBu-Tree & EST-Based Record Schema Grouping

In this study, we propose an algorithm following a divide-and-conquer idea, which reduces the number of sorts from the number of non-leaf nodes to the maximal level, and generates a local code map instead of the global code map for reducing the radix size in the sorting.

The detailed procedure of this method is shown in Alg. 2.

```
Input: Record set: R
Output: eSiBu-Tree, each bucket contains: id,
    a code map CM_b,
    a category flag flag (defaulted as false), and
    a sub-bucket list
1:  for all r ∈ R do
2:      Construct the record schema S
3:      l ← 2, bucket ← root_bucket
4:      while l <= L_max do
5:          Encode each v ∈ V(l) with a code and persist label-code pair in CM_b
6:          Update label of each v' ∈ V(l + 1) by appending its parent's code
7:          codes ← Sort(V(l))
8:          Assign S to a sub_bucket in the sub-bucket list with id = codes
9:          l++, bucket ← sub_bucket,
10:     end while
11:     flag ← true
12: end for
```

Algorithm 2: EST-Based Equivalent Record Schema Grouping

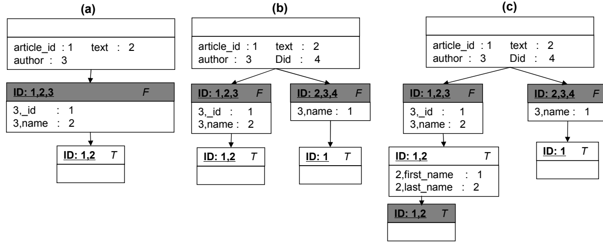


Figure 4: Examples of Generating eSiBu-Tree

The output is a hierarchical data structure, eSiBu-Tree (encoded Schema in Bucket Tree), whose paths persist equivalent record schema categories. Fig. 4 provides an example of eSiBu-Tree based on records in Fig. 1. Each $bucket$ has four variables. The first one is id, the identifier of a class, which is an ordered sequence. The second one is a code map CM_b, the same as Alg. 1, which is constituted by label-code pairs. But it only works for record schemas assigned to this $bucket$, so this code map is a subset of CM_l in the corresponding level from Alg. 1. The third one is a $flag$ to show whether the path from this bucket to the root represents a category of equivalent record schemas. The last one is a sub-bucket list, where $sub_buckets$ of a $bucket$ must have different ids.

In this algorithm, we begin the dividing task from the second level of record schemas (Line 3) because all of record schemas belong to the same category by nature of the same root node. The procedure is as follows. In Line 5, nodes in $V(l)$ are encoded based on the CM_b of the corresponding $bucket$. In this step, we assign each node in $V(l)$ with a code according to its label. Details for node encoding and code map updating are the same as Alg. 1.

The coding of a node in $V(l)$ has two purposes. One is to update its children's codes in Line 6. Different from Alg. 1, the label for encoding is constructed by appending parent's code on its original label. Since a node has one and only one parent in a record schema, there is no sorting effort in the node encoding step. The other purpose of these codes is to generate the id of corresponding sub_bucket by sorting(Lines 7-8). In Line 8, the record schema is assigned into to a sub_bucket based on the id. If there is no such sub_bucket existed, we append a new one with the id on the sub-bucket list of $bucket$. After Line 8, a $bucket$ in the l-th level of eSiBu-Tree represents a category of record schemas whose top-l level subtrees are equivalent.

Besides above steps, Line 11 is executed after processed the maximum level. In this step, we mark the $bucket$ to indicate the path from this bucket to the root representing a category of equivalent record schemas. The necessary of this step is to handle the impact of structural variations. As shown in Fig. 1, S_3 is similar as S_1 except $\{root \rightarrow author \rightarrow name\}$ is evolved as $\{root \rightarrow author \rightarrow name \rightarrow first_name, root \rightarrow author \rightarrow name \rightarrow last_name\}$. Thus, two buckets' flags are set as $true$ in the left branch of the bucket tree shown in Fig. 4(c), which means that the path from the leaf bucket to the root represents a category whose representative is S_3, and the path from the bucket next to the leaf to the root represents another category whose representative is S_1.

To sum up, the EST-based method assigns a record schema to the corresponding $bucket$ by equivalence identification level by level top down. The output of this algorithm is a bucket tree which compresses a category of equivalent record schemas into a path.

Fig. 4 shows how an eSiBu-Tree is generated/updated by processing records from Fig. 1. Comparing with code maps in Fig. 3, the code map in the bucket for encoding the nodes from Level 2 is not expanding with the attribute sparsity and evolution on the node *author*, which is benefit for the performance of the radix sort.

4.3 Online Record Schema Identification

Sec. 4.1 and 4.2 present how to group equivalent record schemas from existing data sets, which is suitable to discover distinct schemas in the batch manner. Furthermore, according to Alg. 1 and 2, both of them just scan a record once in the grouping. This property indicates that both of these algorithms are capable to support online equivalent record schema identification when records are coming incrementally.

Take the EST-based method as an example, for a newly inserted record, operations on each record in the batch manner (i.e., Alg. 2 Lines 2-11) are executed. If the schema of this record has not been persisted, a new path on the eSiBu-Tree will be nominated to represent this new category of record schemas (either the flag of a bucket is turned to $true$ or a new path is added); otherwise, eSiBu-Tree will not change and some statistics (e.g., the frequency of the hit record schema category) will be updated if needed.

4.4 Complexity Analysis

4.4.1 Time Complexity

As introduced in Sec. 4.3, for Alg. 1 and 2, their time complexities are both linear with the number of records $|\mathbf{R}|$. However, they are different in the step for sort. In Alg. 1, since every non-leaf node needs to be updated by combining its ordered children, the number of sorts depends on the number of non-leaf nodes. Meanwhile, Alg. 2 needs one sort in each level to generate id of each $bucket$, so the number of sorts depends on the maximum level of a record schema.

Furthermore, the EST-based method also reduces the size of the code map. The code map of a bucket is generated by record schemas assigned into it, meanwhile, the code map array considers the whole data set. As a result, the code map of eSiBu-Tree is part of the code map in the corresponding level from the code map array, which is of benefit to radix sort.

To summarize, the complexity for processing a record in Alg. 1 is O($|v^0| \times |CM_l|$), where v^0 is the number of non-leaf nodes, and $|CM_l|$ is the size of the global code map. For Alg. 2, the time complexity is $O(L_{max} \times |CM_b|)$, where $|CM_b|$ is the average size of the code map in a bucket. For data sets in document stores where attributes are diversity and have multiple levels (i.e., JSON records with embedded objects), Alg. 2 is much faster than Alg. 1, because the number of nodes is larger than the maximum level ($|v^0| > L_{max}$) and the size of the global code map is greater than the (local) bucket code map's size ($|CM_l| > |CM_b|$).

4.4.2 Space Complexity

Since the EST-based method splits the global code map into bucket code maps, it leads to duplications. For example, in Fig. 4(c), the label *3,author* appears twice. At the same time, for the code map array, the sizes of code maps expand, because each code map includes labels with common parts, such as *author,1,2* and *author,2* in the Level 2 of Fig. 3. Therefore, the worst cases of space complexities in these algorithms are roughly the same, which are linear with four factors: the number of equivalent record schema categories N, the average attribute size in a record schema $\bar{m}$, the maximum level L_{max}, and the average length of each LABEL len.

In details, they are the same in boundary cases. One of them is all record schemas in the collection are equivalent, so there is only one path in the eSiBu-Tree which is the same as the code map array. Another case is there is no common attribute for any two distinct record schemas. In this case, the sum of code maps in a level of the eSiBu-Tree is the code map in the same level of the code map array, as a result, their space consumptions are the same.

5. QUERY

This section presents the Query function of the schema management framework. In this section, we present two kinds of existence queries, which are schema existence query and attribute existence query. In each query, we first introduce its motivation, then propose a SQL-like API, and at last present algorithms to implement it based on the code map array and the eSiBu-Tree.

5.1 Query 1: Schema Existence

Schema Existence Query aims to check whether a specified record schema has been persisted. Similar to RDBMSs that have a function to allow users to check the existing table list and table definitions, this query mechanism could be broadly used by developers to find the right collection to persist a defined object. For example, suppose the developer is a newer in a project, to execute this query as a pre-checking helps him to decide whether to insert a record to the collection which persists records with the same schema, or to create a new collection to insert. The SQL-like API is as:

SELECT S^* *from METADATA where* $S^* = S(r)$

where $S(r)$ is the record schema of the given record r, and *METADATA* represents the repository persists all record schemas. As no duplicate record schema is persisted, the return of this query is a boolean, where $true$ represents the existence.

Alg. 3 shows the detailed procedure of executing Query 1 on the code map array. Besides filtering record schemas higher than all persisted ones in Lines 2-4, the major task is to check whether labels updated by combining their ordered children's codes have been contained by the corresponding code map (Lines 5-16).

The implementation of Query 1 on the eSiBu-Tree is shown as Alg. 4. In the eSiBu-Tree, there are three conditions to determine the existence. The first one is the label has to be contained by the code map of corresponding bucket (Lines 4-9). The second one is the bucket has a sub-bucket with an id the same as ordered code sequence, which indicates such combination of nodes has appeared (Lines 11-17). The last one is the $flag$ of the final bucket is $true$, which means the equivalent record schema category presented by the bucket path from this bucket to the root has been persisted (Lines 19-23).

Both of these algorithms are similar as procedure on each record in grouping equivalent record schemas respectively, except from generating the code map or creating a new sub-bucket to checking existences. As shown in Sec. 4.4.1, these two algorithms are comparable in the performance. The difference is also triggered by the sorting step. When schemas are from a data set where records are constituted by a lot of embedded objects, Query 1 implemented on the eSiBu-Tree runs faster than on the code map array.

```
Input: A record r, and code map array [CM_1, CM_2, ..., CM_D]
Output: true/false (true is by default)
1: Construct the record schema S of r
2: if L_max > D then
3:     return false
4: end if
5: l ← L_max
6: while l > 0 do
7:     for all v ∈ V(l) do
8:         if v is not in CM_l then
9:             return false
10:        else
11:            code ← Encode(v, CM_l)
12:        end if
13:    end for
14:    Update(V(l − 1)) with ordered codes from children
15:    l − −
16: end while
```

Algorithm 3: Query 1 based on Code Map Array

```
Input: A record r, and an eSiBu-Tree
Output: true/false (true is by default)
1: Construct the record schema S of r
2: l ← 2
3: while l <= L_max do
4:     for all v ∈ V(l) do
5:         if v is in CM then
6:             code ← Encode(v, CM)
7:         else
8:             return false
9:         end if
10:    end for
11:    codes ← Sort(codes)
12:    if existed a sub_bucket with id = codes then
13:        Update(V(l + 1))
14:        l + +, bucket ← sub_bucket,
15:    else
16:        return false
17:    end if
18: end while
19: if flag ≠ true then
20:     return false
21: end if
```

Algorithm 4: Query 1 based on eSiBu-Tree

5.2 Query 2: Attribute Existence

Attribute Existence Query aims to determine record schemas containing a specified attribute, which provides a finer granularity pre-checking by locating the attribute. Moreover, this query could be used to identify different object types due to lack of schema names (multiple objects in one collection), or the version of an object schema for the sake of the evolving data model caused by highly iterative developments, with a specific attribute they contain. The SQL-like API is as:

SELECT S from METADATA where $attr \in S$

where $attr$ is the given attribute. Because of the record schema variation, a given attribute may exist in more than one record schema, therefore the result of Query 2 is a record schema set. If the given attribute does not appear, the set is empty.

We implement this attribute existence query on the code map array and the eSiBu-Tree in Alg. 5 and Alg. 7 respectively. In the code map array, a record schema is represented by a code in the code map of root level, so Alg. 5 returns a set of codes. Similarly, Alg. 7 returns a set of bucket paths. A record schema can be retrieved based on the code and the bucket path respectively.

Alg. 5 shows procedure of checking the attribute existence in the code map array. Beside determining the existence by the maximal level (Line 1), the core operations are to check the existence of

Input: An attribute $attr = v_1 \rightarrow \ldots \rightarrow v_{D'}$,
and code map array $[CM_1, CM_2, \ldots, CM_D]$
Output: A set of codes: $\mathbf{C}$

```
1: if D' <= D and v_{D'} is in CM_{D'} then
2:     code_{D'} ← Encode(v_{D'}, CM_{D'})
3:     C ← FindCodebyDepth(C, CM_{D'-1}, v_{D'-1}, code_{D'})
4: end if
```

Algorithm 5: Query 2 based on Code Map Array

each label level by level bottom up iteratively in Alg. 6. Take *root* → *author* → *name* as an example, and the corresponding code map array is as Fig. 3(c). The label *name* is in the code map of Level 3, whose code is 2. Next, we implement Alg. 6, the label *author* and the code 2 appear simultaneously in two labels in code map of Level 2, which are *author,1,2* encoded as 2 and *author,2* encoded as 5. Then, continuing on Alg. 6, the label *root* with the code 2 is in *root,1,2,3*, and *root* with the code 5 is in *root,3,4,5*. As a result, the final canonical code set has two items.

Input: A set of codes: $\mathbf{C}$, a code map: CM_l, a node: v_l, and a code: $code_{l+1}$
Output: A set of codes: $\mathbf{C}$

```
1: for all label in CM_l do
2:     if label contains both v_l and code_{l+1} then
3:         code_l ← Encode(label, CM_l)
4:         if l − 1 == 0 then
5:             add code_l to C
6:         else
7:             C ← FindCodebyDepth(C, CM_{l−1}, v_{l−1}, code_l)
8:         end if
9:     end if
10: end for
```

Algorithm 6: FindCodebyDepth($\mathbf{C}$, CM_l, v_l, $code_{l+1}$)

Alg. 7 presents the attribute existence query on the eSiBu-Tree. This algorithm has two major steps. The first one is to determine the bucket (together with its ancestors) which contains the given attribute level by level recursively, as Alg. 8. In this step, we focus on two points: 1) whether the code of v_{l-1} is in an id of sub_bucket; and 2) whether a label combined by the label of v_l and the code of v_{l-1} is in the code map of corresponding sub_bucket. Still take the attribute *root*→*author*→*name* as an example, and the eSiBu-Tree is in Fig. 4(c). Following Alg. 8 level by level top down iteratively, we obtain that two buckets in the third depth of this eSiBu-Tree contain the given attribute.

Input: An attribute $attr = root \rightarrow v_2 \ldots \rightarrow v_{D'}$,
and the eSiBu-Tree
Output: A set of bucket path: $\mathbf{P}$

```
1: if v_2 is in CM of root_bucket then
2:     code_2 ← Encode(v_2, CM)
3:     B ← FindContainer(B, v_3, code_2, root_bucket)
4:     P ← RetrieveBucketPath(B)
5: end if
```

Algorithm 7: Query 2 based on eSiBu-Tree

After obtained the bucket containing the last node of the attribute, the second major step is to build up the bucket path representing a record schema as the output. In this example, since the $flag$s of these two buckets we have found out are both $true$, the final result set contains bucket paths from these buckets to the root. Besides this, there are other two cases to generate the bucket path which represents a category of record schemas.

One case is the $flag$ of the bucket containing the last node of attribute is $false$. For example, suppose the attribute is *root* → *text*, and the bucket containing its last node is in the 2-level with $id = 1, 2, 3$. In this case, all full paths from its descendants (whose $flag$ is $true$) constitute the output because each one of them contains

Input: A set of bucket: $\mathbf{B}$, a node: v_l, a code: $code_{l-1}$, and a bucket: $bucket$
Output: A set of bucket: $\mathbf{B}$

```
1: for all sub_bucket of bucket do
2:     if code_{l−1} exists in the id of sub_bucket and
       Update(v_l) exists CM of sub_bucket then
3:         code_l ← Encode(Update(v_l), CM)
4:         if l + 1 > D' then
5:             add sub_bucket in B
6:         else
7:             B ← FindContainer(B, v_{l+1}, code_l, sub_bucket)
8:         end if
9:     end if
10: end for
```

Algorithm 8: FindContainer($\mathbf{B}$, v_l, $code_{l-1}$, $bucket$)

information of the given attribute. As a result, we can determine the attribute *root* → *text* is in three record schemas.

The other one appears in querying *root* → *author* → *_id*. The bucket on the eSiBu-Tree containing its last node is in the 3-depth with $id = 1, 2$. The $flag$ of this bucket is $true$, so the path from it is included in the output. Besides, its sub_bucket does not have any attribute expansion from the given attribute. This is presented as no label in sub_bucket's code map containing the code of the last node in the given attribute. As a result, this sub_bucket and its descendants whose $flag$ is $true$ are all outputs. Thus, we can determine two distinct record schemas containing *root*→*author*→*_id*.

Considering the performance of implementations on the code map array and the eSiBu-Tree, it seems that procedure of searching the final bucket paths on the eSiBu-Tree is more complicated, however it is just comparable with the recursive searching level by level as Alg. 6. Sometimes, it is even faster since we only need to check the $flag$. The most time consuming part of them are both related to code existence checking. For the query on the code map array, we have to scan all labels in CM_l (as Alg. 6). For the query on the eSiBu-Tree, the scan region is the sub-bucket list (as Alg. 8). Since the number of sub-buckets is always fewer than the size of the code map, the query implemented on the eSiBu-Tree runs faster than that on the code map array.

6. SCHEMA PRESENTATION & SKELETON CONSTRUCTION

This section presents the skeleton construction process performed in the Schema Presentation function of the framework. In Sec. 6.1, we describe how to process a collection of records containing multiple object types. Then, Sec. 6.2 presents details of the skeleton construction for a specific object type.

6.1 Skeleton Construction for a Collection

As discussed in Sec. 1, a collection of records in a document store may persist more than one object type. Hence, the data model of this collection can be presented as a set of skeletons, each of which describing an object type. The workflow of skeleton construction for a collection are shown in Fig. 5. The inputs are record schemas parsed from the schema repository. The output includes skeletons of all object types. The detailed steps are as follows.

Schema Parser This step is to parse the specific data structure into distinct record schemas for the following study. In this study, we have presented two data structures to persist record schemas of document store, which are code map array and eSiBu-Tree. In the code map array, a category of equivalent record schemas is represented by a code in the code map of the root level (Level 1) as shown in Fig. 3. The retrieval process starts from the corresponding label of this code, and leverages code maps in each level to append subtrees on a record schema iteratively. In the eSiBu-Tree, the path

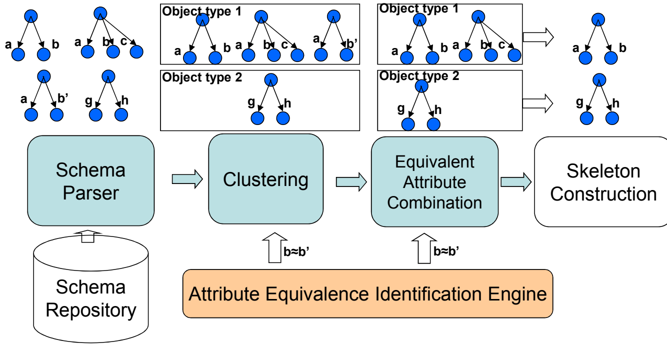


Figure 5: Schema Presentation Workflow

from a bucket with $flag$ as $true$ to the root-bucket represents a category of equivalent record schemas as Fig. 4. For such a path, the retrieval process starts from the root bucket, and leverages id and CM of each bucket to append nodes level by level iteratively.

Attribute Equivalence Identification Engine is the backend to identify attribute equivalences as defined in Sec. 3. *Clustering* aims to differentiate object types. As there are many methods and solid studies related to these parts, we will not discuss them elaborately. For readers interested, here are surveys on schema matching [25] and clustering [30] respectively.

Equivalent Attribute Combination This step is to ensure uniqueness and denseness of skeleton candidates, as listed in Def. 1. This pre-processing step is based on the result from the backend engine, which is to combine attributes which are equivalent semantically but have different names and/or different granularity. The detailed procedure is: for each $X \cong Y$, when the frequency of X is higher than Y, Y in record schemas is replaced by X.

Skeleton Construction This step constructs the skeleton of each object type based on record schemas, details of which are described in the following sub-section. As the above pre-processing step is designed to guarantee uniqueness and denseness of skeleton candidates, the updated record schema set is used as the input of the following skeleton construction.

6.2 Skeleton Construction for an Object Type

This section considers the problem of constructing the skeleton describing a specific object type. Recall the definition in Sec. 3, we formulate the skeleton construction as finding out the highest qualified attribute set. The quality of an attribute set K_c is as follows:

$$q(K_c) = \sum_{i=1}^{N} \alpha_i \frac{|S_i \cap K_c|}{|S_i|} - \sum_{i=1}^{N} \beta_i (1 - \frac{|S_i \cap K_c|}{|K_c|}) \tag{4}$$

Eq. (4) shows the total quality is the weighted average on all record schemas in $\mathbf{S} = \{S_1, S_2, \ldots, S_N\}$. In this study, we set weights for gain function and cost function respectively as:

$$\alpha_i = \frac{n_i}{\sum_{i=1}^{N} n_i} \tag{5}$$

$$\beta_i = \frac{\frac{1}{n_i}}{\sum_{i=1}^{N} \frac{1}{n_i}} \tag{6}$$

where n_i is the number of records whose record schemas are equivalent to S_i. Eq. (5) shows α_i for the gain function is proportional to its frequency, which means the skeleton tries to retrieve more information of the record schema which appears in the data set frequently. Eq. (6) shows β_i for the cost function is inversely proportional to its frequency, which implies the skeleton tolerates the redundancy in the highly frequent record schemas. To sum up, this heuristic idea for designing weights is to make the skeleton be inclined to frequent record schemas. The assumption here is that the frequency of a record schema represents its importance and significance in the data set.

As we have no prior knowledge of the data set, any subset of the universal attribute set M can be a candidate set, so there are $2^{|M|}$ candidates . In order to calculate their qualities, we need a computational complexity as $O(N \times 2^{|M|})$. Therefore, the most critical task for us is to reduce the scale of candidates.

Let K_m refer to the candidate attribute set with highest quality among attribute sets with size m. It is clear that there exists an m such that $K = K_m$, i.e., if the skeleton has m attributes, it is the highest quality candidate skeleton among attribute sets of size m. Therefore, we need to find the highest quality candidate for $m = 1 \cdots |M|$ first. The next strategy we use is obtaining an upper bound for the quality of candidate skeletons with a given size.

THEOREM 1. *The upper bound set K_m is composed by the top-m attributes in M with the feature $fea_m(p_k)$ defined as:*

$$fea_m(p_k) = \sum_{i|p_k \in S_i} (\frac{\alpha_i}{|S_i|} + \frac{\beta_i}{m}) \tag{7}$$

PROOF. Suppose K_m is the upper bound set of size m, $\tilde{K}_m$ is the attribute set with top-m $fea_m(p_k)$, and $K_m \neq \tilde{K}_m$, i.e., $K_m = \hat{K} \cup \{p_s\}$, where $\hat{K} = K_m \cap \tilde{K}_m$, and $\{p_s\} \neq \emptyset$. Since $\{p_s\} \not\subseteq \tilde{K}_m$, there is

$$\sum_{p_s \in \{p_s\}} \sum_{i|p_s \in S_i} (\frac{\alpha_i}{|S_i|} + \frac{\beta_i}{m}) < \sum_{p_t \in \tilde{K}_m \setminus \hat{K}} \sum_{i|p_t \in S_i} (\frac{\alpha_i}{|S_i|} + \frac{\beta_i}{m}) \tag{8}$$

Thus, $q(\tilde{K}_m) > q(K_m)$, so any attribute outside top-m $fea_m(p_k)$ is not the upper bound set K_m. □

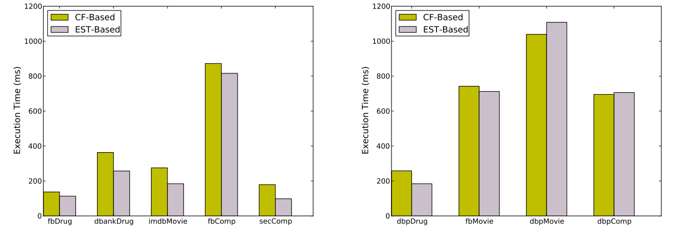
```
Input: Record schema set: S = {S_1, ..., S_N};
    Attribute set: M = ∪_{i=1}^N S_i;
    Weights: {α_i} and {β_i}
Output: Skeleton K
1:  for all p_k ∈ M do
2:    for all S_i ∈ S do
3:      if p_k ∈ S_i then
4:        γ(p_k) ← γ(p_k) + α_i/|S_i|
5:        φ(p_k) ← φ(p_k) + β_i
6:      end if
7:    end for
8:  end for
9:  for all m = 1 : |M| do
10:   for all p_k ∈ M do
11:     fea_m(p_k) = γ(p_k) + φ(p_k)/m
12:   end for
13:   pick top-m fea_m(p_k) as K_m
14:   q(K_m) ← Σ_{K_m} fea_m(p_k)
15:   if q(K_m) > q_max then
16:     K ← K_m; q_max ← q(K_m)
17:   end if
18: end for
```

Algorithm 9: Skeleton Construction

Alg. 9 shows details of constructing the skeleton K. Lines 1-8 generate two temporary variables $\gamma(p_k)$ and $\phi(p_k)$ to avoid duplicate computations in calculating the feature pointed by Theorem 1. After Line 8, $\gamma(p_k) = \sum_{i|p_k \in S_i} \frac{\alpha_i}{|S_i|}$ and $\phi(p_k) = \sum_{i|p_k \in S_i} \beta_i$ are ready, so the feature in Eq.(7) which equals to $\gamma(p_k) + \frac{\phi(p_k)}{m}$, is easy to be calculated in Line 11. Lines 13-14 obtain the upper bound set for a size m. Finally, Lines 15-17 are to find the highest quality candidate which is returned as the skeleton.

Table 2: Data Set Statistic & Schema Discovery Results

Object	Source	DataSet	Rec#	Attr#	$\|S\|_{avg}$	L_{max}	Sch#
Drug	Freebase	`fbDrug`	3,888	42	13	4	147
	DBpedia	`dbpDrug`	3,662	340	33	2	2,818
	DrugBank	`dbankDrug`	4,774	144	103	3	13
Movie	Freebase	`fbMovie`	84,530	48	14	2	13,914
	DBpedia	`dbpMovie`	30,332	1,513	42	2	25,137
	IMDb	`imdbMovie`	7,435	29	10	3	2,992
Company	Freebase	`fbComp`	74,970	110	10	6	6,847
	DBpedia	`dbpComp`	24,367	1,738	39	2	21,302
	SEC	`secComp`	1,981	60	29	5	180



(a) $L_{max} > 2$ (b) $L_{max} = 2$

Figure 6: Performance of Schema Discovery

In addition, the overall computational complexity for the skeleton construction is $O(N \times |M| + |M|^2 \log |M|)$. Since attribute numbers are always less than record schema numbers (as listed in Table 2), to scan record schemas for $fea_m(p_k)$ with $O(N \times |M|)$ is usually the dominant in constructing the skeleton in practice. Besides, applications supported by the Schema Presentation (such as data exploration) is without a real time requirement, i.e., such workload is just for off-line executions.

7. EVALUATION

In this section, we present results of evaluating the schema discovery from real-word data sets, as well as the performance of query implementations. In addition, we also evaluate the effectiveness of using skeletons in the schema presentation with two practical cases.

7.1 Data Sets

Table 2 shows the statistics of data sets used in our experiments. These data sets consist of three object types, and each of them has three data sources. Thus, there are nine data sets to evaluate. Similar to our motivating examples described in Sec. 1, record schemas in these data sets have several variations.

The data sets are all in the JSON format and stored in a document store. All of these data sets are publicly available and have been used in the past for evaluating linkage discovery algorithms [13]. The Freebase data [10] is downloaded in JSON using a query written in Metaweb Query Language (MQL). DBpedia data [8] is fetched from the DBpedia's SPARQL endpoint in the RDF/N-Triples format and converted into JSON. Company scenario uses data extracted from the U.S. Securities and Exchange Commission (SEC) online form files using IBM's SystemT [5] with outputs in JSON. The drug scenario uses information about drugs extracted from DrugBank [9], which is an online open repository of drug and drug target data. Movie scenario uses movie data from the online movie database, IMDb [15].

7.2 Performance of Schema Discovery

In this section, we first evaluate the performance of equivalent record schema grouping algorithms for the schema extraction. The experiments were run on a workstation (Intel Core i5 Processor with 2.67GHz and 4GB of RAM). The category numbers shown in the Table 2 with the column titled "Sch#" confirm our analysis in Sec. 1. The records describing one object are in various schemas because of variations as illustrated by motivating examples. This phenomenon is most notable in data sets from DBpedia, where equivalent record schema category numbers are very close to the number of records. In another saying, there exist large portions of record schemas only used in one of the records.

The experimental results in Fig. 6 show the performance of canonical form based and EST-based methods. For the data sets whose $L_{max} > 2$ (i.e JSON records are constructed by multiple embedded objects) as shown in Fig. 6a, the EST-based method outperforms the CF-based method in all data sets. This advantage comes from reductions on the number of sorts and the radix size as the analysis in Sec. 4.4. Take the data set `secComp` as an example, the average number of non-leaf nodes in a record schema is 9.25, while it has at most 5 levels (among which the root level does not need to sort in the EST-based algorithm), thus the number of sorts in the CF-based is more than double of that in the EST-based algorithm. In addition, the largest code map size in the code map array is 53 which is also larger than the largest one in the eSiBu-Tree with 33 label-code pairs. As a result, the EST-based method has an advantage in grouping/identifying equivalent record schemas when records are constituted by embedded objects.

In addition, when schemas of all records are flat, i.e., $L_{max} = 2$, both CF-based and EST-based methods only need to sort once for each record, and the code map size from the array and the eSiBu-Tree are also the same. As a result, executive times of two algorithms on flat schemas are comparable as shown in Fig. 6b.

7.3 Performance of Queries

7.3.1 *Performance of Schema Existence Query*

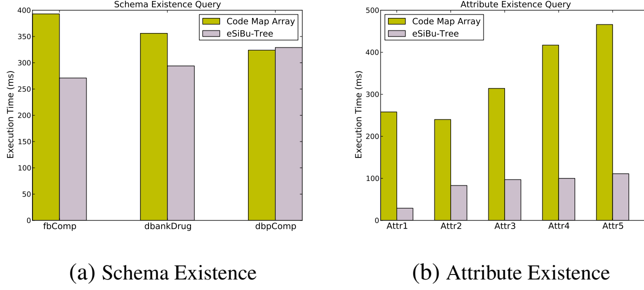
This section focuses on the performance of the schema existence query. In the experiment, we leverage 1000 records to generate specific data structures for record schema persistence first, and then check whether the schema of a given record has existed. Since the execution time of processing one record is very short, we present the cumulative execution time on 4000 records in Fig. 7a. Furthermore, in each experiment, 1000 records for data structure generation and 4000 records for checking are both randomly selected from the data set, and Fig. 7a displays the average execution time of five tests on each data set.

As analyzed in Sec. 5.1, the procedure on each record is similar to the method of grouping records with the equivalent record schema. Thus, the trend of performance on the schema existence query is the same as in the schema extraction (Fig. 6). When the maximal level is large (such as `fbComp` and `dbankDrug`), implementations on the eSiBu-Tree runs faster than that on the code map array. However, in the case that record schemas are flat (such as `dbpComp`), these two methods are comparable.

7.3.2 *Performance of Attribute Existence Query*

This section focuses on the performance of the attribute existence query. In order to evaluate the performance of this query under attributes with different lengths, we leverage `fbComp` as the data set because its maximum level is large and its attributes have various lengths. Table 3 lists attributes used in this experiment, and the corresponding results are shown in Fig. 7b.

As the analysis shown in Sec. 5.2, the difference of them is related to checking regions. For the query implemented on the code map array, we have to scan all labels in CM_l, and there are 7020



(a) Schema Existence

(b) Attribute Existence

Figure 7: Performance of Queries

Table 3: Attributes for Attribute Existence Query

NO.	Attribute	Sch#
Attr1	root$\rightarrow$ type	3,731
Attr2	root$\rightarrow$ advisors $\rightarrow$ name	117
Attr3	root$\rightarrow$ locations $\rightarrow$ address $\rightarrow$ street_address	187
Attr4	root$\rightarrow$ locations $\rightarrow$ address $\rightarrow$ citytown $\rightarrow$ id	187
Attr5	root$\rightarrow$locations$\rightarrow$address$\rightarrow$state_region$\rightarrow$country$\rightarrow$id	144

label-code pairs in the code map array for `fbComp`. For the query implemented on the eSiBu-Tree, the scan region is the sub-bucket list of a bucket. In the eSiBu-Tree of `fbComp`, there are 2376 sub-buckets in total, but for a specific attribute (especially longer ones), it does not need to check all these sub-buckets by filtering out buckets without upper level nodes. Therefore, the attribute existence query on the eSiBu-Tree overall outperforms that on the code map array. In addition, since both of them have to check all nodes in the given attributes, the performance also depends on the length of the attribute, as a result, execution times are increasing with the attribute lengths increasing.

7.4 Effectiveness

In this section, we evaluate the effectiveness of using the skeleton in schema presentation with two practical cases. The first one is designed for the scenario that data analyzers and scientists want to explore data from document stores. The second one shows advantages of the skeleton in reducing the linkage point searching space when integrating a document store with other data sources.

In the data extraction procedure, we persist data with the same object type from a data source into a data set. In other words, we will not consider the difference in record schemas caused by different object types, so the clustering step is not evaluated in this study. Moreover, in this study, we leverage an existing highly efficient schema matching engine to identify equivalent attributes [13]. After the equivalent attribute combination, the attribute size of data sets are: 303 for the `dbpDrug`, 1,472 for the `dbpMovie`, 1,648 for the `dbpComp`, and 109 for `fbComp`. For other data sets, their attribute sizes are the same as shown in Table 2.

7.4.1 Exploration Scenario

As introduced in Sec. 1, the schema-less nature makes document stores hard to be explored. Furthermore, some intuitive approaches such as the universal attribute set (union all distinct record schemas $\cup_{i=1}^{N} S_i$) and the common attribute set (intersect all distinct record schemas $\cap_{i=1}^{N} S_i$), are not suitable in the data exploration.

Take the collection describing `dbpComp` as an example. This data set has over twenty thousands record schemas, and in total 1,648 attributes. Fig. 8a shows the frequency distribution of the attributes in the ranking order. We observe that the frequency distribution is with a long tail, which means there are large portions of attributes with very low frequencies. In fact, according to our

Table 4: Five attributes in the Skeleton

root$\rightarrow$ http://dbpedia.org/property/homepage
root$\rightarrow$ http://dbpedia.org/property/companyName
root$\rightarrow$ http://dbpedia.org/property/products
root$\rightarrow$ http://dbpedia.org/property/location
root$\rightarrow$ http://dbpedia.org/property/keyPeople

Table 5: Five attributes out of the Skeleton

root$\rightarrow$ http://dbpedia.org/property/bankrupt
root$\rightarrow$ http://dbpedia.org/property/suspension
root$\rightarrow$ http://dbpedia.org/property/totalCarbohydrates
root$\rightarrow$ http://dbpedia.org/property/topSpeed
root$\rightarrow$ http://dbpedia.org/property/jurisdiction

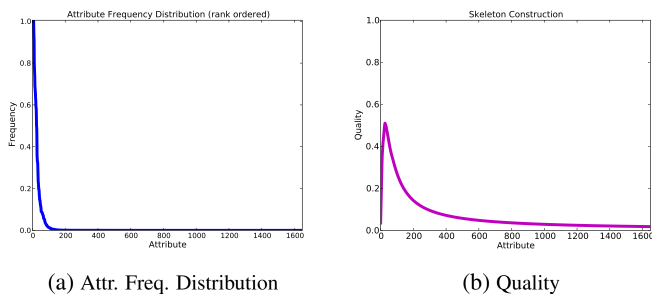
statistics, 954 attributes appear just once in this data set, and 211 attributes appear twice. In the preview scenario, if we presented all of these attributes to users, they are easy to mixed-up core attributes and others. Meanwhile, records in this data set have just one common attribute, which is *root* $\rightarrow$ *uri*. Obviously, if we only showed this attribute as the preview, users would lost a lot of important information.

The skeleton proposed in this study provides a single view of the data set. Fig. 8b shows how qualities as Eq. (4) vary with the size increasing of upper bound sets. The highest qualified, skeleton, is the attribute set generating the peak value of this plot. For the data set `dbpComp`, the attribute size of the skeleton is 28. Thus, it has a concise formation in comparison with the whole attribute set. The most of important is to capture the core attribute of the specific object, such as company in this example. In order to demonstrate the significance of the skeleton, we randomly select five attributes from and out of the skeleton, which are shown in Table 4 and 5 respectively. Comparing attributes in these two tables, it is easy to recognize attributes in the skeleton, such as *name*, *products*, *location*, are general to every company. However, attributes out of the skeleton are particular. For example, *bankrupt* is a possible property of unhealthy companies, and *carbohydrates* might be a noise in the data related to a set of food manufacturing companies' products instead of the company entity itself.

In order to evaluate the effectiveness of using the skeleton for summarizing the schema set, we propose two metrics to measure the summarization performance of the skeleton. The first index is to show the gain of this skeleton, named as retrieval rate (RR). As the definition in Sec. 3, the retrieval rate is defined as the object retrieved by the skeleton, denoted as: $RR = \sum_{S_i} \alpha_i |S_i \cap K| / |S_i|$. The second index is to assess the size of the attribute set for achieving such gain. We use the relative size (RS) here, which is defined as the percentage of the size of the skeleton over the universal attribute set, denoted as: $RS = |K|/|M|$.

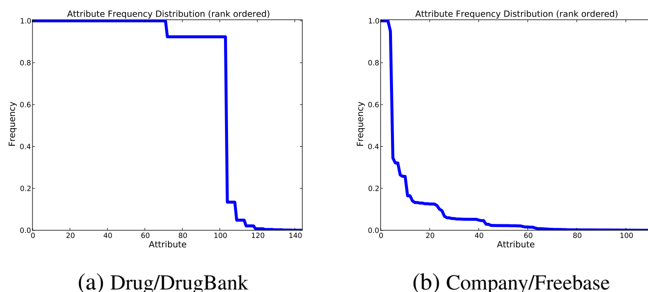
Table 6 shows the performance of the skeleton with retrieval rate and relative size comparing with the maximal common set and the universal set. For the universal set, both retrieval rate and relative size is 1. We also annotate differences between two baseline methods with the skeleton. For example, for the `dbpComp` we discussed, the skeleton leverage 2% of attributes to represent 76% information of the data set. Comparing with the maximal common set, it uses 0.14% extra attributes, but retrieves more than 70% information. Together with it, comparing with the universal set, the skeleton saves 98% attributes with only a loss of 24% retrieval rate.

Table 6 also shows us the retrieval rates and relative sizes vary in different data sets. For example, the skeleton for `dBankDrug` retrieves almost all of record schemas. However, in `fbComp`, the skeleton only covers 67%. These variances come from their attribute frequency distributions. Fig. 9a and 9b show the rank ordered frequency distribution of attributes from `dBankDrug` and



(a) Attr. Freq. Distribution (b) Quality

Figure 8: Company from DBpedia



(a) Drug/DrugBank (b) Company/Freebase

Figure 9: Attribute Frequency Distribution

`fbComp` respectively. These two figures present us one of them (`dBankDrug`) is with a short tail, while the other one is (`fbComp`) has a long tail. The short tail means there are large portions of attributes appearing frequently. If the skeleton excluded one of them, that will bring a lot of losses in retrieving the object. Thus, because of containing such large portions of high frequency attributes, the retrieval rate of the corresponding skeleton is near to 1. Meanwhile, for the distribution with a long tail, the proportion of infrequent attributes is large. As the gain of adding an extra attribute to the skeleton cannot offset the redundancy accompany with it, we use a concise attribute set to summarize the data set. As a conclusion, the number of attributes in the skeleton depends on the attribute frequency distribution.

Besides the common set and the universal set, there are some other intuitive schema presentation approaches, such as top-k most frequent attributes, attributes' frequency over a $threshold$, etc. In these approaches, how to choose the parameter is the most critical issue. For example, when we use top-50 most frequent attributes to present the schema, for `fbComp`, one out of five of the selected attributes appear less than 1% in records, while for `dbankDrug`, since over 60 attributes appear in every record, the top-50 approach cannot even cover the common set. To sum up, comparing with these parameterized approaches, the skeleton is a novel parameter-free schema presentation method, which is helpful in case users have no prior knowledge on the data set.

7.4.2 Integration Scenario

To further investigate the effectiveness of using the skeleton in schema presentation, we use an integration scenario from previous work [13] where the goal is discovering linkage points between data sources. Briefly, a linkage point is a pair of attributes from two sources that can be used to effectively perform *record linkage* (or *entity resolution*), i.e., to link records between the sources that refer to the same real-world entity. The final goal is to integrate all sources into one duplicate-free (clean) source that contains data from all sources.

Table 6: Performance of Skeleton in Summarization

Data Set	Skeleton		Common		Universal	
	RR	RS	RR	RS	RR=1	RS=1
`fbDrug`	0.91	0.29	0.81(-0.10)	0.24(-0.05)	(+0.09)	(+0.71)
`dbpDrug`	0.85	0.11	0.13(-0.72)	0.01(-0.10)	(+0.15)	(+0.89)
`dbankDrug`	0.9997	0.86	0.71(-0.29)	0.50(-0.36)	(+0.0003)	(+0.14)
`fbMovie`	0.88	0.29	0.53(-0.35)	0.15(-0.15)	(+0.12)	(+0.71)
`dbpMovie`	0.92	0.04	0.03(-0.89)	0.001(-0.04)	(+0.08)	(+0.96)
`imdbMovie`	0.88	0.45	0.35(-0.53)	0.10(-0.34)	(+0.12)	(+0.55)
`fbComp`	0.67	0.05	0.41(-0.26)	0.03(-0.02)	(+0.33)	(+0.95)
`dbpComp`	0.76	0.02	0.04(-0.72)	0.001(-0.02)	(+0.24)	(+0.98)
`secComp`	0.98	0.82	0.10(-0.88)	0.05(-0.77)	(+0.02)	(+0.18)

A challenge in linkage point discovery for schema-less data such as those used in our experiments is the large search space in terms of the number of attributes. For example, in the `dBankDrug` and `dbpComp` case, investigating all the possible pairs of attributes with the goal of finding linkage points means investigating 179,632 attribute pairs, each of which could contain a large number of values of different data types and characteristics. Previous work has proposed efficient methods of investigating and pruning the search space using extensive pre-processing and highly efficient indexes. Here, we would like to examine how using the skeleton instead of all search space can be effective in finding linkage points. In other words, we would like to study the correlation between attributes that appear in linkage points and those that appear in the skeleton.

Table 7 shows the search space for each of the nine integration scenarios in terms of the number of pairs of attributes that need to be investigated, along with the number of linkage points found in the search space comparing the use of the skeleton with the use of maximal common set and the universal set. The results show that using the skeleton to prune the search space for linkage point discovery is very effective overall. For example, for the `dbpDrug`/`dBankDrug` scenario, using the skeleton reduces the search space (Pair#) to only 8% of the universal set search space whereas 19 out of the 21 linkage points can be found in this space. Another example is `dbpComp`/`secComp` scenario, the search space is reduced to 1% by using the skeletons whereas 7 out of the 8 linkage points are found. Overall in these nine scenarios, using skeletons reduce search spaces to 6% of the universal set on average, and find out average 58% of linkage points. It is important to note that based on the manual verification of linkage results, those linkage points that include attributes from the skeleton are also very effective linkage points. Recall that the ultimate goal of discovering linkage points is at generating high quality record linkages, so we assess the effectiveness of the discovered linkage points by the quality of the record linkage generated by using these linkage points. For example, in the `dbpComp`/`fbComp` scenario, there are two linkage points among the 13 linkage points found in the search space of skeletons that can achieve perfect accuracy in record linkage (i.e., link all the records that can be linked, with 100% precision). This also shows that the use of the skeleton can improve state-of-the-art in linkage point discovery and ranking.

8. RELATED WORK

The early studies on schema extraction over semi-structured data are based on the Object Exchange Model (OEM) [23, 24, 28, 29]. The general approach in schema discovery based on OEM is relied on a labelled graph formulated by the objects and their relationships in the data set, and then finds the exact or approximate object typing via clustering or a translated optimization problem. However, this class of work does not consider efficiency of schema discovery, and a summarized presentation for a single object type.

Table 7: Linkage Point Discovery Search Space & Effectiveness

Scenario		Skeleton		Common		Universal	
ds1	ds2	Pair#	L.P#	Pair#	L.P#	Pair#	L.P#
fbDrug	dbpDrug	396	5	40	3	12,768	26
fbDrug	dbankDrug	1,488	12	720	3	6,048	17
dbpDrug	dbankDrug	4,092	19	288	5	43,776	21
fbMovie	dbpMovie	756	13	7	0	70,656	22
fbMovie	imdbMovie	182	2	21	2	1,392	2
dbpMovie	imdbMovie	702	5	3	0	42,688	8
fbComp	dbpComp	140	13	3	0	179,632	76
fbComp	secComp	245	6	9	4	6,540	30
dbpComp	secComp	1,372	7	3	0	98,880	8

Another group of studies related to schema discovery out of XML documents [11, 14, 21] address the issue of XML documents on the Web not having an accompanying schema, or not adhering to any given schema. Typically, a schema for XML data, e.g. DTD and XML Schema, specifies for every element a regular expression pattern that sub-element sequences of the element need to conform to. In accordance with this definition, the main focus of previous work [11, 14, 21] is to infer such regular expressions for a given XML data set. In contrast, our proposed notion of skeleton attempts to find core and representative attributions of a given collection in an imprecise manner. Our focus is to generate such a structure to represent heterogeneous schemas in the given collection.

There are also several studies on frequent substructures mining [4, 32]. However, under the context of document stores, due to the flexibility of developers ingesting data and the heterogeneity in data itself, it is hard for users to specify any parameters to pre-define the frequency on the data they want to explore. The skeleton construction method proposed in this paper aims to design a parameter-free approach which automatically provides a balance between the frequency of attributes and the size.

In addition, previous work has studied schema summarization based on quality measures [31]. This work addresses the problem of summarizing the schemas of multiple tables based on their linkages, but the goal in our schema presentation function is to find a single data model for every object type in one collection. Furthermore, their quality measures for summarization are defined based on connectivity of tables, whereas the skeleton focuses on finding a balance between the significance and the size of the summary.

9. CONCLUSION

In this paper, we presented a framework for schema management for document stores that deals with the schema-less data model and fast-evolving nature of document stores. We proposed a new data structure, eSiBu-Tree, to persist record schemas as well as support queries, and an equivalent sub-tree (EST) based method with a linear computational complexity to group equivalent record schemas. We compared the effectiveness of this data structure with a baseline approach using canonical forms. Extensive experiments demonstrated the efficiency of the overall framework, and also showed that the EST-based method outperformed the CF-based method in both discovery and query tasks. Furthermore, we proposed a new concept "skeleton" to describe a schema summary structure suitable for document stores. Practical use cases were presented to demonstrate the effectiveness of the skeleton in real data exploration and integration scenarios.

In future, we are planning to devise algorithms to retrieve the skeleton for other NoSQL applications, such as when some priori but uncertain knowledge of the data set is available. Furthermore, we plan to study the evolution of data models of a specific data set persisted in data stores based on skeletons corresponding to different versions of applications.

10. REFERENCES

[1] DB2 V10.5 Manual. http://www-01.ibm.com/support/docview.wss?uid=swg27038855. [Accessed April 3, 2015]

[2] Oracle Database 12c. https://docs.oracle.com/en/database/database.html. [Accessed April 3, 2015]

[3] Why NoSQL? Technical report, CouchBase, 2013.

[4] T. Asai, K. Abe, et al. Efficient substructure discovery from large semi-structured data. In *SDM*, 158–174, 2002.

[5] D. Burdick, M. A. Hernández, et al. Extracting, linking and integrating data from public sources: A financial case study. In *IEEE Data Eng. Bull.*, 34(3):60–67, 2011.

[6] Y. Chi, Y. Yang, and R. R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. In *Knowledge and Information Systems*, 8(2):203–234, 2005.

[7] Couchbase. http://www.couchbase.com. [Accessed April 3, 2015]

[8] DBpedia. http://dbpedia.org. [Accessed April 3, 2015]

[9] DrugBank. http://drugbank.ca. [Accessed April 3, 2015]

[10] Freebase. http://freebase.com. [Accessed April 3, 2015]

[11] M. Garofalakis, A. Gionis, et al. Xtract: a system for extracting document type descriptors from XML documents. In *ACM SIGMOD Record*, 29: 165–176, 2000.

[12] O. Hassanzadeh, S. Hassas Yeganeh, and R. J. Miller. Linking semistructured data on the web. In *WebDB*, 2011.

[13] O. Hassanzadeh, K. Q. Pu, et al. Discovering linkage points over web data. In *VLDB*, 6(6):444-456, 2013.

[14] J. Hegewald, F. Naumann, and M. Weis. Xstruct: efficient schema extraction from multiple and large XML documents. In *ICDE Workshop*, 81, 2006.

[15] IMDb. http://www.imdb.com/. [Accessed April 3, 2015]

[16] Twitter Inc. Twitter developers documentation. https://dev.twitter.com/docs/api/1.1/overview. [Accessed April 3, 2015]

[17] Facebook Inc. Facebook developers documentation. https://developers.facebook.com/docs/. [Accessed April 3, 2015]

[18] Google Inc. Using JSON in the google data protocol. https://developers.google.com/gdata/docs/json. [Accessed April 3, 2015]

[19] JSON. http://www.json.org/. [Accessed April 3, 2015]

[20] Z. H. Liu, B. Hammerschmidt, and D. McMahon. JSON data management: supporting schema-less development in RDBMs. In *SIGMOD*, 1247–1258, 2014.

[21] J. K. Min, J. Y. Ahn, and C. W. Chung. Efficient extraction of schemas for XML documents. In *Information Processing Letters*, 85(1):7–12, 2003.

[22] MongoDB. http://www.mongodb.org/. [Accessed April 3, 2015]

[23] S. Nestorov, J. Ullman, et al. Representative objects: Concise representations of semistructured, hierarchical data. In *ICDE*, 79–90, 1997.

[24] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ACM SIGMOD Record*,27: 295–306, 1998.

[25] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. In *VLDB* , 10(4):334–350, 2001.

[26] List of NoSQL Databases. http://nosql-database.org/. [Accessed April 3, 2015]

[27] F. Özcan, N. Tatbul, et al. Are we experiencing a big data bubble? In *SIGMOD*, 1407–1408, 2014.

[28] K. Wang and H. Liu. Schema discovery for semistructured data. In *KDD*, 97:271–274, 1997.

[29] Q. Y. Wang, J. X. Yu, and K. F. Wong. Approximate graph schema extraction for semi-structured data. In *EDBT* , 302–316, 2000.

[30] R. Xu, D. Wunsch, et al. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[31] C. Yu and H. Jagadish. Schema summarization. In *VLDB*, 319–330, 2006.

[32] M. Zaki. Efficiently mining frequent trees in a forest. In *SIGKDD*, 71–80, 2002.