An Approach for Schema Extraction of NoSQL Columnar Databases: the HBase Case Study

Angelo Augusto Frozza[1,2], Eduardo Dias Defreyn[2] and Ronaldo dos Santos Mello[2]

[1] Instituto Federal Catarinense - IFC, Brazil
angelo.frozza@ifc.edu.br
[2] Programa de Pós-graduação em Ciência da Computação - PPGCC
Universidade Federal de Santa Catarina - UFSC, Brazil
eduardo_dududex@hotmail.com,r.mello@ufsc.br

Abstract. Although NoSQL databases do not require a schema a priori, being aware of the database schema is essential for activities like data integration, data validation, or data interoperability. This paper presents a process for the extraction of columnar NoSQL database schemas. We adopt JSON as a canonical format for data representation, and we validate the proposed process through a prototype tool that is able to extract schemas from the HBase columnar NoSQL database system. HBase was chosen as a case study because it is one of the most popular columnar NoSQL solutions. When compared to related work, we innovate by proposing a simple solution for the inference of column data types for columnar NoSQL databases that store only byte arrays as column values, and a resulting schema that follows the JSON Schema format.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design; E.1 [**Data Structures**]: Miscellaneous; H.2 [**Database Management**]: Miscellaneous

Keywords: Columnar, HBase, JSON Schema, NoSQL, Schema extraction

1. INTRODUCTION

In the current scenario of computer systems development, new applications have as a requirement the need for greater flexibility over the data representation, which may comprise complex nested structures and data structures *in memory*. In order to meet these needs, NoSQL databases (NoSQL DBs) have been proposed. They are classified into four data models: key-value, document, columnar, and graph [Sadalage and Fowler 2013].

Unlike traditional relational DBs, NoSQL DBs are usually *schemaless*, *i.e.*, they do not require a previous schema or support a flexible schema, which facilitates data storage and data integrity checking [Han et al. 2011; Sadalage and Fowler 2013]. Nevertheless, to be aware of the data schema is essential for processes, such as data integration, data interoperability, or data analysis. Although NoSQL DBs do not require an explicit schema, there is usually an implicit schema on each DB instance that is ruled by the application that accesses it [Ruiz et al. 2015]. However, to infer the data schema from the source code of the application is a complicated process. Another alternative is to analyse and get information from the DB management system (DBMS) catalog, which may also be a hard task.

Relational databases have a pre-defined and rigid schema, usually represented by the SQL/DDL standard [Elmasri and Navathe 2016]. In theory, all relational DBMSs follow the same schema representation pattern. On the other hand, columnar NoSQL databases may have pre-defined schemas

This research was partially financed by the *Instituto Federal Catarinense* (IFC) and by *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) - Brazil - Finance Code 001.

with less rigid structures that can change at runtime [Sadalage and Fowler 2013]. Each DBMS has its own DML language, and there is no standard in the form of representation and retrieval of the schemas.

Thus, this paper presents a process for columnar NoSQL DB schema extraction. Although we focus on the HBase DBMS, the process can be adapted to other columnar NoSQL DBMS. It also generates schemas in *JSON Schema* format[1]. HBase particularly stores data as byte arrays. It makes the column data types known only by the application that generated the data. So, a challenge is how to infer the correct data types from a byte array. Our proposed process is materialized as a prototype tool called *HBase Schema Inference (HBaSI).*

The literature presents few studies related to schema extraction from NoSQL DBs. Our contribution focuses on the extraction of schemas from columnar NoSQL DBs through a recursive and more straightforward approach than the ones available in related work [Kiran and Vijayakumar 2014; Ruiz et al. 2015]. We deal with the problem of data type inference from HBase data instances, and we justify the usage of JSON format as a canonical data model for NoSQL DBs. From that, we adopt *JSON Schema* as a schema representation format for the schema of NoSQL DBs.

This paper is an extended version of the work of [Frozza et al. 2020], presented as a short paper in the XXXV Brazilian Symposium on Databases (SBBD 2020). In this new version, we add formal definitions to the concepts of the columnar NoSQL data model, as well as the canonical representation of that data model in JSON format. We also extend the discussion of related works, and detail the schema extraction process as well as experimental evaluation. The remaining text is organized as follows. Section 2 introduces columnar NoSQL DBs and HBase. Section 3 presents our proposal to use JSON as a canonical format for NoSQL data. Section 4 details the proposed process and Section 5 shows the evaluation. Section 6 comments the related work and Section 7 presents the conclusion.

2. COLUMNAR NOSQL DATABASES AND HBASE

A NoSQL DB can be defined as a distributed and scalable DB that normally does not have a fixed schema, avoids join operations, does not always have an SQL access interface and tends to be open source [Tudorica and Bucur 2011]. NoSQL DBs can be categorized by the adopted data model: key-value, document-oriented, columnar, and graph-oriented. The first three are known as *key-based NoSQL models*, as they share the principle of retrieving data from an input key while differing in terms of access to the internal components of the data [Atzeni et al. 2014].

The *columnar* data model organizes data based on a model distributed in columns, similar to the relational model, but with a flexible scheme. Figure 1 shows the structure of a columnar NoSQL DB. Each column is associated with a key-value pair. A set of columns defines a row, which is accessed by an identifier (*Row-Key*). Different rows can hold different columns, being suitable for the representation of heterogeneous data [Hewitt 2010]. Some columnar NoSQL DBMSs also support the *supercolumn* concept, *i.e.*, columns composed of other columns. The most popular columnar DBMSs are HBase[2], Cassandra[3], and Hypertable[4].

As a contribution of this article, we formally define the concepts of a NoSQL columnar data model in the following.

Definition 1. ***(Keyspace).*** *A keyspace ks is a tuple $ks = (n_{ks}, CF)$, where n_{ks} is the name of the ks, CF is a set of column families, and ks is identified by n_{ks}.*

[1]JSON Schema, `https://json-schema.org` [Accessed 03-June-2021]
[2]HBase, `https://hbase.apache.org/` [Accessed 03-June-2021]
[3]Cassandra, `https://cassandra.apache.org/` [Accessed 03-June-2021]
[4]Hypertable, `https://hypertable.org/` [Accessed 03-June-2021]

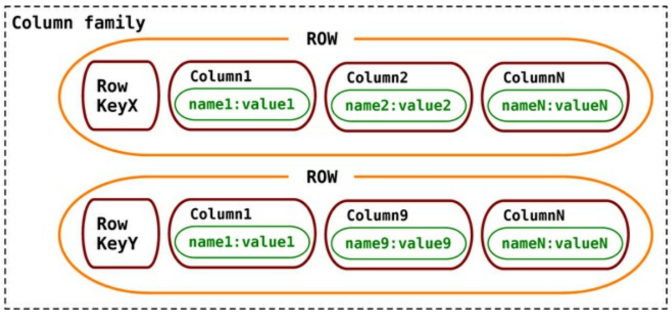


Fig. 1. Example of a columnar NoSQL DB structure **[Sadalage and Fowler 2013]**

Definition 2. ***(Column Family)****. A column family $cf \in ks.CF$ is a tuple $cf = (n_{cf}, CS)$, where n_{cf} is the name of the cf, CS is a set of column sets so that each $cs \in CS$ is a tuple $cs = (key_{cs}, C)$, being $cs.key_{cs}$ the key to access a column set and C a column set, and cf is identified by n_{cf}.*

Definition 3. ***(Column)****. A column $col \in cs.C$ is a tuple $col = (n_{col}, v_{col})$, where n_{col} is the column name and holds an atomic content, v_{col} is the column value and holds an atomic content or a nested column set C, as defined in Definition 2, and col is identified by n_{col}.*

In this paper, we consider *HBase* as a case study, as this is a widely used columnar NoSQL DBMS. It was developed by *Apache Foundation* and runs on the *Hadoop* distributed system[5]. It is a high performance and open-source DBMS with a flexible schema [Shriparv 2010]. Its data model can accommodate diverse semistructured data, which are converted into a *byte array* to facilitate data distribution. This physical representation makes hard to determine the data types of the columns without prior knowledge when accessing data.

The HBase data model, as well as other columnar NoSQL DBMSs, defines a hierarchical structure (see Figure 2)[6]. A DB instance is called *namespace* (that corresponds to the *keyspace* of definition 1), which contains a set of *tables*. Each table, in turn, holds a set of *rows*, where each row has an identifier (*RowKey*) and at least one *column* that is always linked to a *column family*. A *column family* holds a set of required or optional *columns*, and different column family instances may hold a different number of columns (a so-called *column-oriented* data model).

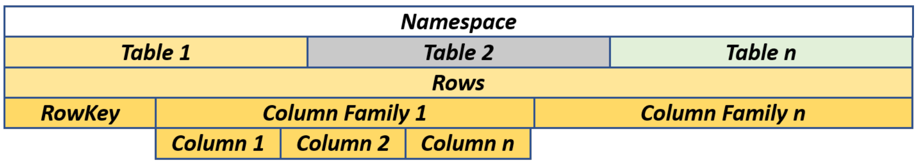


Fig. 2. HBase hierarchical structure.

3. CANONICAL REPRESENTATION OF A COLUMNAR NOSQL DATABASE

This paper proposes the usage of the JSON format as a canonical representation of the columnar NoSQL data model. It was chosen as a canonical representation because most NoSQL DBs, including columnar DBs, support the JSON format. JSON is able to represent the concepts of all NoSQL

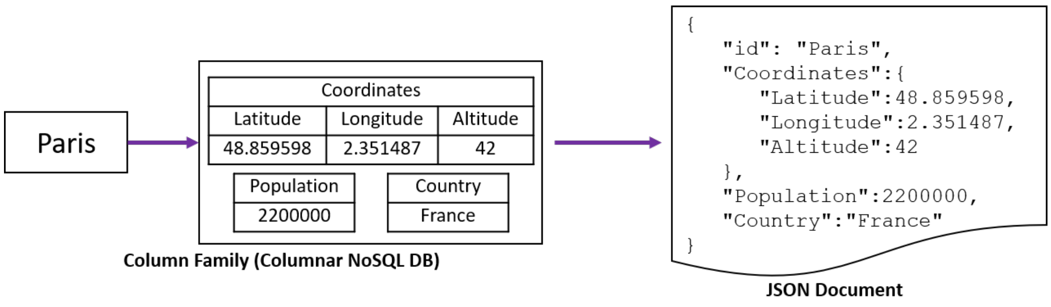
[5]Apache HBase ™ Reference Guide, `https://hbase.apache.org/book.html` [Accessed 03-June-2021]
[6]adapted from `http://www.informit.com/articles/article.aspx?p=2253412` [Accessed 03-June-2021]

data models (document-oriented, key-value, columnar, and graph-oriented), as shown in Table I[7]. It is noteworthy that JSON elements of type *array* only exist in document-oriented NoSQL DB. In contrast, the other NoSQL data models do not model ordered data.

Table I. Mapping NoSQL data models to JSON format.

Key-Value	**Columnar**	**Graph-oriented**	**JSON** (document-oriented)
key-value pair	column family	vertex	document
key	column family key	vertex key	element ID
non-serialized or serialized atomic value	atomic column	vertex atomic attribute	atomic element
serialized nested value	super column	vertex nested attribute or vertex edges	object type element
N/A	N/A	N/A	array element

Figure 3 shows data about the city of *Paris* instantiated in a columnar NoSQL DB, as well as its conversion to a JSON instance in a document-oriented DB. The mapping of the columnar data model to JSON is presented in Definition 4. In this case, each uniquely-identified column set from a column family is mapped to a JSON document due to their conceptual similarity.



```
{
   "id": "Paris",
   "Coordinates":{
      "Latitude":48.859598,
      "Longitude":2.351487,
      "Altitude":42
   },
   "Population":2200000,
   "Country":"France"
}
```

Fig. 3. JSON as a canonical format for columnar NoSQL DBs.

DEFINITION 4. ***(Columnar-to-JSON Mapping)***. *Given a column family cf from a NoSQL columnar DB, each column set $cs \in cf.CS$ generates a JSON document doc so that $doc.id_{doc} \leftarrow cf.n_{cf} + \text{“.”} + cs.key_{cs}$, and each column $col \in cs.C$ generates an (atomic/object) element $e_k \in doc.A$ so that $e_k.n_{att} \leftarrow col.n_{col}$ and $e_k.v_{att} \leftarrow col.v_{col}$.*

Moreover, a column value can be converted to an atomic or object element in a JSON document if its value is atomic or the column is a supercolumn, respectively. We also consider the document key as the concatenation of the column family name and the column set key because we cannot guarantee that column sets in different column families have distinct keys. In the example of Figure 3, the document key holds only the column set key because the example does not present the column family name.

[7]N/A means not applicable.

4. THE SCHEMA EXTRACTION PROCESS

As stated before, we consider HBase as our case study as it is a widely used columnar NoSQL DBMS. Our schema extraction process receives as input a *namespace* (see Section 2) and outputs a schema specification for the DB in *JSON Schema*, which is a common recommendation for data representation and exchange[8]. The basis of the schema extraction process is the analysis of the DB hierarchical structure. So, a new *namespace* is created, containing a copy of the input without the data, *i.e.*, only the nested structure *table/column family/column* is maintained. The process goes through each table and, for each one, it analyses its columns, making the inference of the data type (see Figure 4). It is necessary to scroll all the column values to check for variations in the values of the data types stored into a column. Algorithm 1 summarizes this process. The worst-case computational complexity of the approach is defined by $O(\#T+\#C+\#R)$, where $\#T$ is the number of tables, and $\#C$ and $\#R$ are the number of columns and records of all tables, respectively.

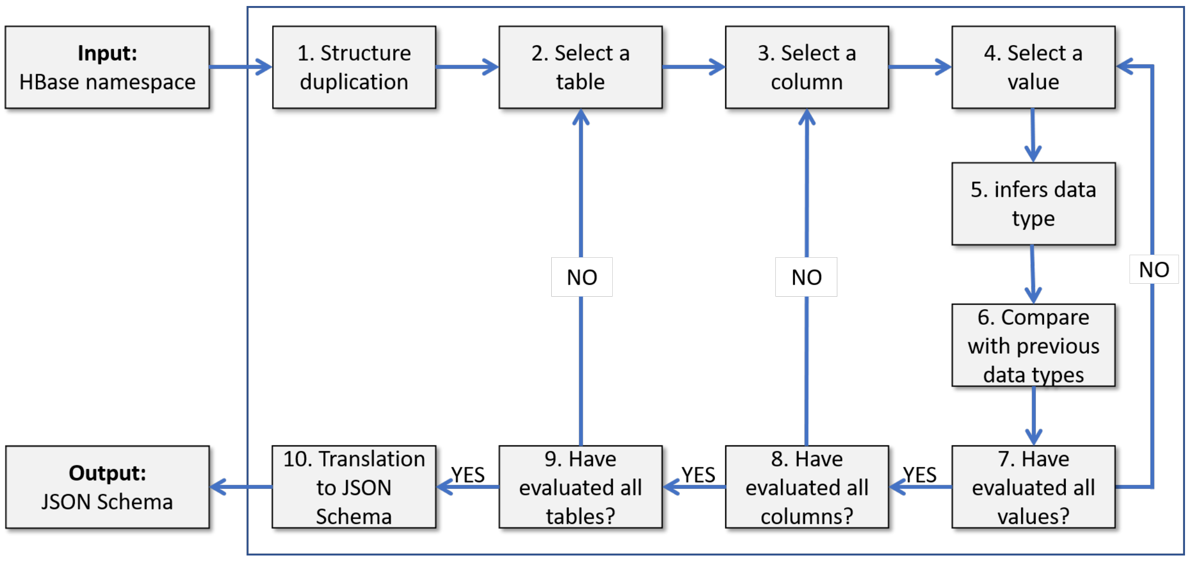
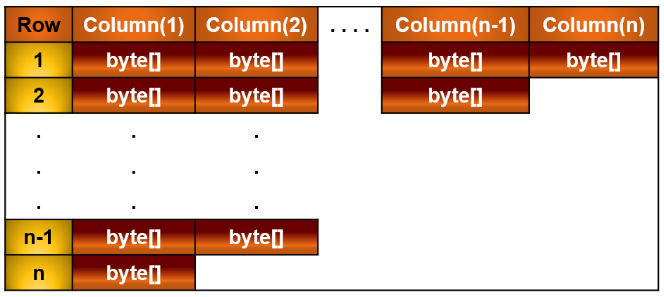

Input:
HBase namespace
1. Structure duplication
2. Select a table
3. Select a column
4. Select a value
5. infers data type
6. Compare with previous data types
7. Have evaluated all values?
8. Have evaluated all columns?
9. Have evaluated all tables?
10. Translation to JSON Schema
Output:
JSON Schema
YES
YES
YES
NO
NO
NO

Fig. 4. Organization of the JSON Schema document generated by the process.

As mentioned earlier, HBase stores data in terms of byte arrays. This particularity represents the most difficult task of our process, *i.e.*, to infer a data type from a byte array, as the data type is only known by the application that uses the data [Shriparv 2010].



Row	Column(1)	Column(2)	....	Column(n-1)	Column(n)
1	byte[]	byte[]		byte[]	byte[]
2	byte[]	byte[]		byte[]	
.	.	.			
.	.	.			
.	.	.			
n-1	byte[]	byte[]			
n	byte[]				

Fig. 5. HBase table example.

[8]JSON Schema, `https://json-schema.org` [Accessed 03-June-2021]

Algorithm 1: Schema extraction process

```
Input: HBase $namespace$
Output: JSON Schema document
begin
    $rawSchema \longleftarrow structureDuplicate(namespace)$
    foreach $table \in namespace$ do
        foreach $column \in table$ do
            $datatypeList \longleftarrow \{\}$
            foreach $value \in column$ do
                $datatypeList \longleftarrow datatypeList + datatypeInference(value, len)$
            end
            $rawSchema.table.column \longleftarrow datatypeComparison(datatypeList)$
        end
    end
    $JSONSChema \longleftarrow schemaMapping(rawSchema)$
    return $JSONSchema$
end
```

Figure 5 represents a table of an HBase namespace. It shows several instances of data (*Rows*), each one with a variable number of columns. Each column may have a different data type, and a byte array represents each value in a column.

Figure 6 shows the structure of a data item (key-value pair) stored by HBase as a byte array. The first eight bytes correspond to the *key* (4 bytes) and the *value* (4 bytes) size of the data. The next part, with variable length, presents metadata that identifies the *column* (column family, column, timestamp, and others). The last part corresponds to the *value* itself. Thus, on considering only the part of the array corresponding to the *value*, our strategy here was to develop a set of rules for the inference of the most common data types from the binary content of the data item as follows:

—*byte*: a byte array of size one, accepting any value;

—*boolean*: a byte array of size one, only with the values 0xFF or 0x00;

—*string*: a variable size byte array that follows the UTF8 binary standard;

—*short*: a byte array of size two, accepting any value;

—*char*: a byte array of size four, having its two least significant bytes represented in UNICODE;

—*float*: a byte array of size four, being limited by the values indicated in the IEEE 7541 standard (representation of binary numbers in floating-point);

—*integer*: a byte array of size four that can use up to 32 bits to represent a value;

—*double*: a byte array of size eight, being limited by the values indicated in the IEEE 7541 standard;

—*long*: a byte array of size eight that can use up to 64 bits to represent a value;

—*blob*: any entry that does not fit any of the previous rules.

After analyzing all the values in a column (Algorithm 2), the inferred data types are compared to each other to define a consensual data type (Algorithm 1, line 9), *i.e.*, the more general data type that best represents all the values in the column. Once analyzed the DB structure and the columns of all tables, a *JSON Schema* document is created.

Since most of the inferred data types are not the same as the native data types of JSON Schema (integer, numeric, string, boolean), the *definitions* section of the JSON Schema document is used to describe these data types, concerning extended data type definitions specified by [Frozza et al. 2018].

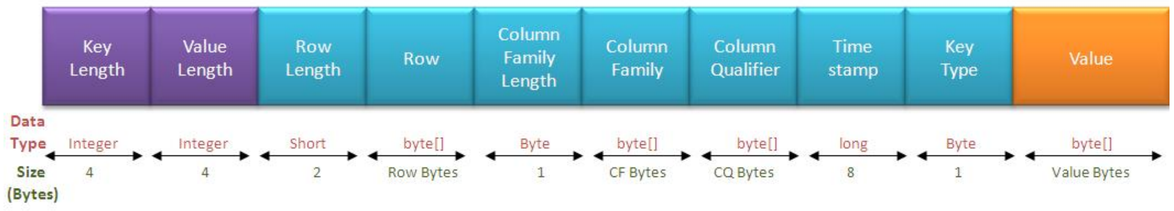
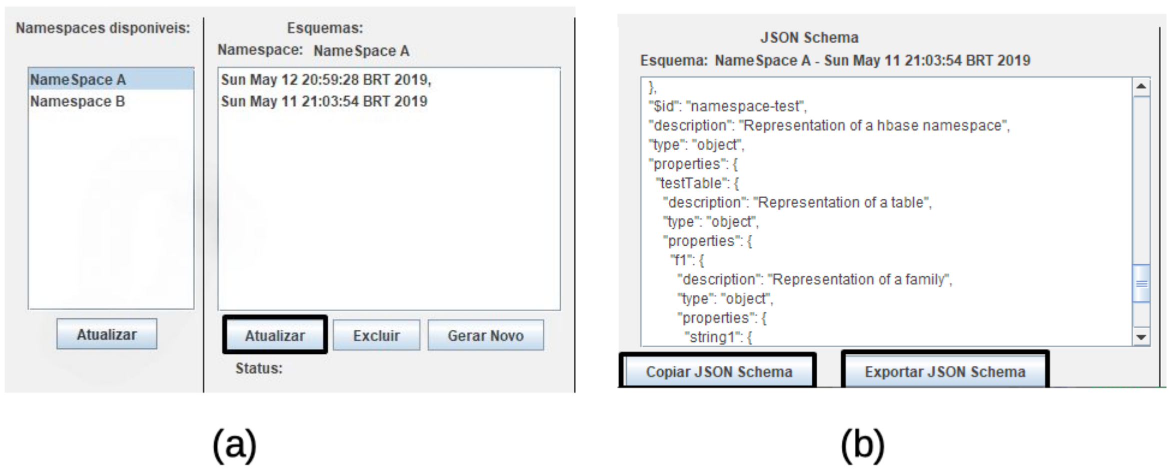


Fig. 6. Byte array structure of a key-value pair in HBase.
(http://prafull-blog.blogspot.com/2012/06/how-to-calculate-record-size-of-hbase.html [Accessed 03-June-2021])

In turn, the *properties* section of the JSON Schema describes the DB hierarchical structure. Each column is a property of an object that represents a column family, each column family is a property of an object that represents a table, and so on.

5. EXPERIMENTAL EVALUATION

A prototype tool called *HBaSI (HBase Schema Inference Tool)* was developed to validate our schema extraction process over HBase. The Java language was used for the implementation, and we adopt the MVC (Model, View, Controller) pattern with the aid of the Apache Maven (version 1.2.1), HBase API (version 1.4.6), and Gson (version 2.2.4) libraries. Figure 7 presents the HBaSI user interface (in Portuguese). As shown in Figure 7 (a), it is able to connect to HBase and list the available namespaces (left window), as well as the different schema versions generated for the selected *namespace* (right window). If the user decides to create or update a namespace schema, the tool generates a corresponding *JSON Schema* document that can be copied to the clipboard or exported as a JSON file (Figure 7 (b)).



Namespace: NameSpace A

Sun May 12 20:59:28 BRT 2019,
Sun May 11 21:03:54 BRT 2019

Esquema: NameSpace A - Sun May 11 21:03:54 BRT 2019

```
},
"$id": "namespace-test",
"description": "Representation of a hbase namespace",
"type": "object",
"properties": {
  "testTable": {
    "description": "Representation of a table",
    "type": "object",
    "properties": {
      "f1": {
        "description": "Representation of a family",
        "type": "object",
        "properties": {
          "string1": {
```

Fig. 7. User interface of the prototype tool

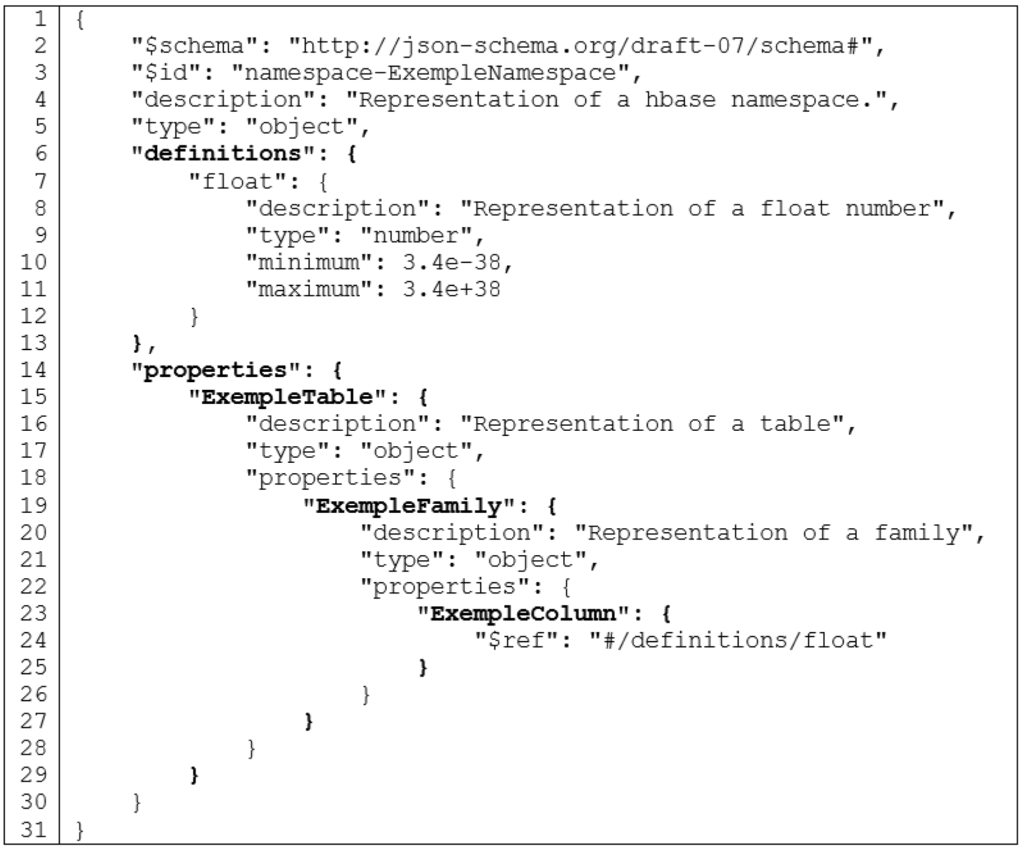
Figure 8 shows a JSON Schema fragment generated by the tool from a test DB. We highlight the definition of the *float* data type (lines 7-12) in the *definitions* section, the representation of a *table* (line 15), a *column family* as an embedded object in the table properties (line 19), and a *column* as an embedded object in the column family properties (line 23).

Algorithm 2: *datatypeInference(value,len)*

```
Input: value (byte array), len (array length)
Output: byte[2] with inferred data types
begin
    switch len do
        case 1 do
            if $isUTF8Valid(value)$ then
                $output \leftarrow string + byte$
            else if value is 0xFF or 0x00 then
                $output \leftarrow boolean + byte$
            else
                $output \leftarrow byte$
        case 2 do
            if $isUTF8Valid(value)$ then
                $output \leftarrow string + short$
            else
                $output \leftarrow short$
        case 4 do
            if $isUTF8Valid(value)$ then
                $output \leftarrow string$
            else if $isChar(value)$ then
                $output \leftarrow short$
            if $isFloat(value)$ then
                $output \leftarrow float$
            $output \leftarrow integer$
        case 8 do
            if $isUTF8Valid(value)$ then
                $output \leftarrow string$
            if $isDouble(value)$ then
                $output \leftarrow double$
            $output \leftarrow long$
        case outro do
            if $isUTF8Valid(value)$ then
                $output \leftarrow string$
            else
                $output \leftarrow blob$
    end
    return output
end
```

For the experimental evaluation, we also implemented a data generator that creates random DB tables with different amounts of rows (10, 100, and 1000). The data types supported by the data generator are the same mentioned in Section 4. In order to assess the quality of the schema extraction, three possible results were considered: *a)* the data type does not match as expected (*incorrect*); *b)* the data type is as expected (*correct*); *c)* more than one possible (equivalent) data type is returned and the expected type is one of them (*partial*). The *partial* option deals with cases where more than one data type is inferred to the same column value, which usually occurs for numeric types.

The extracted schemas from the generated HBase DBs always got a 100% accuracy for the structural hierarchy, as expected, since this information can be obtained in a straightforward way from the DB



```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "$id": "namespace-ExempleNamespace",
    "description": "Representation of a hbase namespace.",
    "type": "object",
    "definitions": {
        "float": {
            "description": "Representation of a float number",
            "type": "number",
            "minimum": 3.4e-38,
            "maximum": 3.4e+38
        }
    },
    "properties": {
        "ExempleTable": {
            "description": "Representation of a table",
            "type": "object",
            "properties": {
                "ExempleFamily": {
                    "description": "Representation of a family",
                    "type": "object",
                    "properties": {
                        "ExempleColumn": {
                            "$ref": "#/definitions/float"
                        }
                    }
                }
            }
        }
    }
}
```

Fig. 8. Fragment of a JSON Schema generated by HBaSI.

namespace. This is not the case for column data type inference, as shown in Table II. We see that *string*, *boolean*, *byte*, *short*, and *blob* types were correctly identified. It happens because the set of rules that defines them (see Section 4) is restricted and does not contain many ranges in common with other data types.

Table II. Result of the data type inference experiment.

Data type	Incorrect	Correct	Partial
string	0%	100%	0%
boolean	0%	100%	0%
byte	0%	99%	1%
short	0%	99%	1%
blob	0%	100%	0%
char	0%	0%	100%
integer	0%	17%	83%
float	0%	0%	100%
long	0%	11%	89%
double	0%	0%	100%

On the other hand, *char*, *integer*, *float*, *long*, and *double* data types, although it was possible to obtain the expected type, it was not possible to always define it accurately. It happens because of the limitations of the four bytes data types (*char*, *integer*, and *float*) which are not exclusive, as well as the eight bytes ones (*long* and *double*), which are also not. In fact, when we analyse data in binary format, the representation of one data type may be contained in the representation of another one.

For example, a *char* content always has a binary encoding similar to an *integer* or *float* content. Additionally, some *integer* contents do not have a binary encoding that also represents a *char* or a *float* content. Therefore, in some cases, it is possible to correctly infer an *integer* data type. The same holds for some *long* contents. For these cases, our schema extraction process considers the set of all possible data types.

We also analyse the spent time to generate the schemas. Although HBase can be used as a distributed DB, this experiment was limited to a monolithic DB instance, running on an Intel (R) Core (TM) i5-3337U CPU @ 1.80GHz machine, with 6 GB RAM. Table III shows the execution times for each input size. The last column shows how many Key-Value (K-V) pairs were analysed per second. It is possible to see that, with a small number of rows and K-V pairs, the time required to generate the schema (third column in Table III) impacts the final performance. When the number of K-V pairs increases, a more linear relationship between processing time versus K-V pairs number is maintained.

Table III. Result of schema extraction processing time experiment.

Rows	**K-V pairs**	**HBaSI processing time (seconds)**	**K-V pairs per second**
10	550	4	137
100	50500	203	248
1000	5005000	23037	229

As this is a preliminary assessment over a monolithic instance, it is not always possible to guarantee a good performance of our solution, as the processing time can be degraded in a distributed environment. However, this tendency for a linear complexity of our process w.r.t. the number of K-V pairs shows that it is a promising proposal. A possible optimization would be to define filters to limit the number of records loaded in the data entry, which are needed to define the data types. These filters could significantly decrease processing time in a Big Data context.

6. RELATED WORK

When migrating from a relational DB to a NoSQL DB, the biggest concern is how to represent the relationship data in the adopted NoSQL data model [Zhao et al. 2014; Schreiner et al. 2015; Lee and Zheng 2015]. This work, instead, aims to extract the most possible detailed schema from an HBase DB, respecting its columnar data model.

There are few papers in the literature related to schema extraction from NoSQL databases. Most of them refer to schema extraction from JSON documents or document-oriented NoSQL databases. Previous papers of our research group had presented proposals for schema extraction from NoSQL DBs that follow the document-oriented [Frozza et al. 2018], key-value [Frozza et al. 2019], and graph-oriented models [Frozza et al. 2020]. With respect to schema extraction from columnar NoSQL DBs, two approaches were found, as described in the following.

The work of [Ruiz et al. 2015] applies MDE (Model-Driven Engineering) techniques to create NoSQL DB schemas based on the aggregate-oriented data model [Sadalage and Fowler 2013], as is the case of the columnar NoSQL data model. They propose a generic NoSQL data model based on the JSON document hierarchical structure. This generic data model requires that the extracted data items have a unique identifier and a *type* property, and the considered data types must be the same as those available for the JSON format. The inference process is composed of three steps and implements model-to-model transformations: *(i)* to extract a collection of relevant objects used in the inference; *(ii)* to inject this collection into a model that conforms to a JSON metamodel. This is accomplished through the mapping of the elements of the JSON grammar to the elements of the metamodel; *(iii)* to transform the JSON model obtained in the previous step into a model that conforms to a NoSQL-Schema metamodel, which represents an aggregate-oriented NoSQL database. The proposed approach was evaluated for the MongoDB, CouchDB, and HBase DBMSs.

Another work introduces an approach for data integration in HBase, in which schemas are extracted and later converted to ontologies in the OWL (Ontology Web Language) format [Kiran and Vijayakumar 2014]. Genetic algorithms are used to identify which record best represents the column families of a table, *i.e.*, a schema that best represents all the data in the table. This schema is then converted into suitable OWL primitives that give rise to a local OWL ontology. Finally, the COMA++ 3.0 CE tool (Community Edition)[9] is used to align local ontologies and find out semantic similarities, thus creating a global ontology.

Different from the aforementioned works, our schema extraction proposal is less complex. It neither imposes constraints on the data items to be analysed nor requires the mapping to high-level abstraction models, like an ontology, and respects the concepts and structure defined in the NoSQL columnar data model. Additionally, our extracted schema follows the *JSON Schema* recommendation.

7. CONCLUSION

This paper presents a process for extracting column-oriented NoSQL database schemas that deals with the problem of determining data types for columns that are represented in a low level of abstraction as byte arrays. Our proposal differs from related work by introducing a simple extraction process that outputs an extracted schema in JSON Schema format.

We also developed a prototype tool that is able to generate DB schemas from the columnar NoSQL HBase DBMS. Preliminary experimental evaluation shows that our proposal is promising in terms of quality of the generated schemas as well as processing time. Although the developed tool does not present 100% accuracy for all inferred data types, it generates a JSON Schema that can be used as a reference in integration, interoperability, or data search processes. Besides, it allows a better understanding of the data stored into an HBase DB. The tool source code is available at *UFSC Database Group repository*[10].

Future work includes the improvement of the data type inference rules, the inference of optional and mandatory columns, performance analysis with large data volumes, and tool internationalization.

REFERENCES

Atzeni, P., Bugiotti, F., and Rossi, L. Uniform access to NoSQL systems. *Information Systems* vol. 43, pp. 117–133, 07, 2014.

Elmasri, R. and Navathe, S. B. *Fundamentals of Database Systems.* Pearson, Boston, 2016.

Frozza, A. A., Defreyn, E. D., and Mello, R. d. S. A Process for Inference of Columnar NoSQL Database Schemas. In *Anais Principais do Simpósio Brasileiro de Banco de Dados (SBBD).* SBC, Porto Alegre, pp. 175–180, 2020.

Frozza, A. A., Jacinto, S. R., and Mello, R. d. S. An Approach for Schema Extraction of NoSQL Graph Databases. In *Proceedings - 2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science, IRI 2020.* IEEE, Las Vegas, NV (USA), 2020.

Frozza, A. A., Mello, R. d. S., and da Costa, F. d. S. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *XIX Int. Conf. on Information Reuse and Integration.* IEEE, Salt Lake City, Utah (USA), pp. 356–363, 2018.

Frozza, A. A., Schreiner, G. A., Machado, B. R. L., and Mello, R. d. S. REx - NoSQL Redis Schema Extraction Module. In *Anais da Escola Regional de Banco de Dados (ERBD).* Sociedade Brasileira de Computacao - SB, Chapecó (SC), pp. 81–90, 2019.

[9]COMA 3.0 Community Edition, `https://dbs.uni-leipzig.de/en/research/projects/schema_and_ontology_matching/coma_3_0/coma_3_0_community_edition` [Accessed 04-June-2021
[10]HBaSI, `https://github.com/gbd-ufsc/HBaSI` [Accessed 03-June-2021]

Han, J., Haihong, E., Le, G., and Du, J. Survey on NoSQL database. In *VI International Conference on Pervasive Computing and Applications*. IEEE, Port Elizabeth, South Africa, pp. 363–366, 2011.

Hewitt, E. *Cassandra: The Definitive Guide*. O'Reilly Media, Sebastopol, CA, 2010.

Kiran, V. K. and Vijayakumar, R. Ontology-based data integration of NoSQL datastores. In *IX Int. Conf. on Industrial and Information Systems*. IEEE, Gwalior, India, pp. 1–6, 2014.

Lee, C.-H. and Zheng, Y.-L. SQL-To-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems. In *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015*. IEEE, Hong Kong, pp. 2022–2026, 2015.

Ruiz, D. S., Morales, S. F., and Molina, J. G. Inferring Versioned Schemas from NoSQL Databases and its Applications. *LNCS* vol. 9381, pp. 467–480, 2015.

Sadalage, P. J. and Fowler, M. *NoSQL Distilled : A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, New Jersey, USA, 2013.

Schreiner, G., Duarte, D., and Dos Santos Mello, R. SQLtoKeyNoSQL: A layer for relational to key-based NoSQL database mapping. In *17th International Conference on Information Integration and Web-Based Applications and Services, iiWAS 2015 - Proceedings*. ACM, Brussels, Belgium, 2015.

Shriparv, S. *Learning HBase*. Packt Publishing, Birmigham, UK, 2010.

Tudorica, B. G. and Bucur, C. A. A Comparison between Several NoSQL Databases with Comments and Notes. In *Proc. RoEduNet IEEE Intern. Conference*. IEEE, Iasi, Romania, 2011.

Zhao, G., Lin, Q., Li, L., and Li, Z. Schema conversion model of SQL database to NoSQL. In *Proc. 9th Intern. Conference 3PGCIC*. IEEE, Washington, DC (USA), pp. 355–362, 2014.