Safa Brahmia¹, Zouhaier Brahmia², Fabio Grandi³, Rafik Bouaziz⁴

^{1,2,4} Department of Computer Science, Faculty of Economics and Management
University of Sfax
Sfax, 3018, Tunisia

³ Dipartimento di Informatica – Scienza e Ingegneria, Alma Mater Studiorum – Università di Bologna
Bologna, I-40136, Italy

safa.brahmia@gmail.com, zouhaier.brahmia@fsegs.rnu.tn, fabio.grandi@unibo.it, rafik.bouaziz@usf.tn

Journal of Digital
Information Management

ABSTRACT: *JSchema is a framework for the management of temporal documents stored in JSON format in a NoSQL database, similar to the τ XSchema framework proposed for XML. In this work, we extend Schema to temporal schema versioning support. The proposed approach provides a systematic solution to the challenging task of evolving a JSON schema while maintaining all schema versions with their corresponding instances and, thus, guarantees the bookkeeping of a full history of both data and schemas. All components of a temporal JSON schema (i.e., conventional schema, temporal logical and physical characteristics) are allowed to vary over time to reflect changes in the real world and, thus, in application requirements. To this end, four sets of primitives for changing JSON schema components and their related JSON instance documents are introduced. The feasibility of our approach and the usage of the proposed primitives have been shown through a detailed and illustrative example.*

Categories and Subject Descriptors

H.2.1 [Database Management]: Logical Design—Schema and subschema; **H.2.7 [Database Management]:** Database Administration—Data dictionary/directory

General Terms: Design, Management

Keywords: NoSQL; JSON; JSON Schema; τ XSchema; Schema change; Change propagation; Schema versioning

Received: 19 March 2017, Revised 28 April 2017, Accepted 25 May 2017

DOI: 10.6025/jdim/2017/15/4/179-202

1. Introduction

Nowadays, NoSQL databases [1-3] are being used in several emerging application fields (including social network management, e-health and e-government systems, big-science projects) to store data with unconventional types, and specifically big data [4-6]. Furthermore, those are live application domains where the evolution of the stored NoSQL data and of their structure has to keep the pace of the continually and rapidly changing application requirements. Hence, most of these advanced applications should keep a full history of both NoSQL data and NoSQL schemas [7], in order to satisfy variegated requirements (e.g., to support historical big data analytics, to help decision makers with protean perspectives, to maintain conformance to regulations in force, to interoperate new software platforms, etc.).

On the other hand, most of mainstream NoSQL DBMSs (e.g., Cassandra, Riak, Couchbase, CouchDB, MongoDB, HBase, Dynamo, and HyperTable) only provide support for data versioning and such support is inherently limited [8]. In fact, some of them allow complete histories of temporal data versions to be kept only if non-conservative operations (e.g., removing a schema component like a property of an object or an item of an array, changing the type of a property) are not performed on the data structure

and none of them provide (temporal) schema versioning support. Therefore, those systems do not allow the maintenance of schema histories and, consequently, they do not allow the bookkeeping of complete data histories under schema changes.

Moreover, most research works on schema changes in NoSQL databases, like [9-18] have dealt with schema evolution, consisting in keeping only the last schema version with its associated instances, and, to the best of our knowledge, there is no work that has studied schema versioning [19-20] in (temporal) NoSQL databases.

In this paper, we aim at filling this gap by proposing an approach for temporal schema versioning in a temporal NoSQL database. More specifically, we extend to schema versioning the τ JSchema framework that we have introduced in [21]. Such a framework allows developers and users to manage the definition and evolution of temporal JSON documents within a temporal JSON-based document-oriented NoSQL database, via the use of a temporal JSON schema. This latter consists of a conventional JSON schema (i.e., a JSON Schema [22] file), which is annotated with a set of temporal logical characteristics and a set of temporal physical characteristics [23]. Temporal logical characteristics allow the specification of which components of a JSON document can vary over time, whereas temporal physical characteristics allow the specification of how the time-varying aspects are represented in the temporal JSON document. Temporal JSON instances are obtained by applying both temporal logical and temporal physical characteristics to conventional JSON instances which are valid to the conventional JSON Schema.

In its initial formulation [21], τ JSchema only supports temporal versioning of temporal JSON instance documents and, thus, provides for the maintenance of a complete data history and yet it does not support schema changes. We aim at enhancing our framework so that not only is

changing temporal JSON instances allowed but also changing the temporal JSON schema (i.e., changing the conventional JSON schema and/or the temporal logical/physical characteristics), in order to guarantee the bookkeeping of a complete history of both temporal JSON instances and temporal JSON schema. Indeed, we propose a complete set of primitives that allows a NoSQL Database Administrator (NSDBA) to apply changes to a conventional JSON schema. Changing a conventional JSON schema (i) creates a new version of this schema, (ii) updates the corresponding temporal JSON schema, in order to take into account this new version, (iii) generates a new conventional JSON document version, conforming to the new conventional JSON schema version, from each conventional JSON document version that was valid with respect to the previous conventional JSON schema version, and (iv) updates the corresponding temporal JSON document, in order to take into account all new conventional JSON document versions. To this aim, we also propose a set of primitives for updating a temporal JSON schema, a set of primitives for updating a temporal JSON document, and a set of primitives for changing a conventional JSON document.

The rest of the paper is organized as follows. Section 2 gives an overview of the JSON Schema language on which our τ JSchema framework is based. Section 3 describes our technique for temporal schema versioning in τ JSchema. Section 4 proposes the sets of primitives to apply changes to conventional JSON schemas, temporal JSON schemas, conventional JSON documents, and temporal JSON documents. Section 5 illustrates our approach through an application example. Section 6 discusses related work. Section 7, finally, summarizes our work and provides some remarks about our future work.

2. JSON Schema Language

JSON Schema [22] is a language for defining JSON [24] data structures. Although it is non-standard, JSON Schema is the only existing schema language associated

	Keyword	Description
Keywords for a numeric instance	multipleOf	A positive number. The instance value must be multiple of this keyword value and the rest of the division of the value instance by this keyword value MUST be 0.
	maximum	It MUST be a JSON number. It is the maximum value that the instance can have.
	exclusiveMaximum	It MUST be boolean. If “exclusiveMaximum” is present, then “maximum” must also be present. If “exclusiveMaximum” has the value “false”, then the instance MUST be lower than, or equal to, the value of “maximum”. If it has the value “true”, then the instance must be strictly lower than the value of the “maximum”. If it is absent, then it may be considered as being present with the value “false”.
	minimum	It MUST be JSON number. It is the minimum value that the instance can have.
	exclusiveMinimum	It MUST be boolean. If the “exclusiveMinimum” is present, then “minimum” must be present. If “exclusiveMinimum” has the value “false”, then the instance MUST be greater than or

		equal to the value of “minimum”. If it has the value “true”, then the instance MUST be strictly greater than the value of “minimum”. By default, if “exclusiveMinimum” is absent it may be considered as being present with the value “false”.
Keywords for a string instance	maxLength	It MUST be an integer greater than, or equal to, 0. It represents the maximum length that the string instance can have.
	minLength	It MUST be an integer greater than, or equal to, 0. It represents the minimum length of the string value. If it is absent, then it may be considered as being present with the value 0.
	pattern	It MUST be a string. It SHOULD be a valid regular expression according to the ECMA 262 regular expression dialect. If it is specified, the instance MUST be valid according to this pattern.
Keywords for an array instance	items	It MUST be either an object, or an array. If it is array, then items of this array MUST be objects.
	additionalItems	It MUST be either a boolean, or an object.
	maxItems	It MUST be an integer greater than, or equal to, 0. It represents the maximum size of the array instance.
	minItems	It MUST be an integer greater than, or equal to, 0. It represents the minimum size that the array can have. If it is absent, it may be considered as being present with the value 0.
	uniqueItems	It MUST be boolean. It expresses if the array items are unique or no. If it is absent, it MAY be considered as being present with the value “false”.
Keywords for an object instance	maxProperties	It MUST be an integer greater than, or equal to, 0. It represents the maximum number of the properties of the object.
	minProperties	It MUST be an integer greater than, or equal to, 0. It represents the minimum number of the properties of the object. If it is absent, it MAY be considered as being present with the value 0.
	required	It MUST be an array that MUST have at least one item. Its items MUST be string and MUST be unique. It represents the properties that MUST always be present in the object.
	properties	It MUST be an object.
	additional Properties	It MUST be a boolean or an object. Each value of this object MUST be an object.
	patternProperties	It MUST be an object. Each property name of this object SHOULD be a valid regular expression (according to the ECMA 262 regular expression dialect) and each property value MUST be an object.
	dependencies	It MUST be an object. Each value of this object MUST be either an object (called “Schema dependency”, since the object MUST be a valid JSON Schema), or an array (called “Property dependency”). If the value is an array, then it MUST have at least one item. The items of the array MUST be a string and unique.
Keywords for any instance type	enum	It MUST be an array that has at least one item. Each item of this array MUST be unique and MAY be of any type including “null”. It represents the list of the values that the instance can have.
	type	It MUST be either a string or an array. If it is an array, then its items MUST be strings (for each item, the type of value specified between double quotes MUST be one of the seven primitive types proposed by JSON Schema in its specification v4) and unique. It defines the type of the instance.
	allOf	It MUST be an array that MUST have at least one item. Each item of this array MUST be an object (an object is a valid JSON Schema). An instance MUST be valid against all schemas defined by this keyword’s value.

	anyOf	It MUST be an array that can have at least one item. Each item of this array MUST be an object (an object is a valid JSON Schema). An instance MUST be valid against at least one schema defined by this keyword's value.
	one of	It MUST be an array that can have at least one item. Each item of this array MUST be an object (an object is a valid JSON Schema). An instance MUST be valid against exactly one object defined by this keyword's value.
	not	It MUST be an object. An instance MUST fail to validate successfully against the object defined by this keyword.
Metadata keywords	title	They MUST be of string type. They can be used to decorate a user interface with information about the data produced by this user interface.
	description	"title" should be a short text and "description" should provide more details and explanations about the instance.
	default	No restriction on the value of this keyword.
Other keywords	id	It MUST be a string and MUST be a valid URI.
	\$schema	It MUST be an URI and a valid JSON reference [27].

Table 1. List of JSON Schema keywords according to the JSON Schema specification ver. 4

to JSON instances and several professional tools [25], e.g., Liquid XML Studio 2016, Visual Studio 2013, JSONBuddy, and ReSharper 2016.1, are supporting it. An attempt to provide a formal definition of the syntax and semantics of JSON Schema has been done by [26], but such a definition does not cover all aspects of the language.

Since our TJSchema framework is based on the JSON Schema language, which is used for defining conventional JSON schemas, our study is based on the latest complete specification of this language, that is the fourth IETF Internet-Draft [22] providing the JSON Schema core definitions and terminology. Such a specification allows us to determine the list of components (objects, arrays, properties, etc.) that can be specified using JSON Schema. Such list is summarized in Table 1, where all JSON Schema components are presented with their allowed keywords and descriptions.

Notice that the components or keywords mentioned in Table 1 have helped us in defining primitives that could act on all these components (i.e., by creating, modifying, or dropping them). These primitives will be presented in detail in Section 4.

3. The Temporal Schema Versioning Process in TJSchema

In this section, we describe the process of temporal schema versioning in the TJSchema framework, which is summarized in Fig. 1. Indeed, the process starts when the NSDBA defines a new conventional JSON schema (which becomes the first version of the conventional schema and is a JSON Schema file) and annotates it with a set of temporal logical and of temporal physical characteristics (which becomes the first version of the

temporal characteristic document and is a JSON file). Thus, the system generates a temporal JSON schema (which is a JSON file) that ties together the conventional JSON schema version and the temporal characteristic document version.

Instances conformant to conventional JSON schema versions are stored in conventional JSON document versions (which are JSON files). In our TJSchema approach, the change of a conventional JSON document gives rise to (i) a new version of this document, and (ii) the update of the temporal JSON document in order to take into account this new version.

After that, the NSDBA can independently change the conventional JSON schema and/or the temporal characteristic document, when necessary. Each change to the conventional JSON schema (to the temporal characteristic document, respectively) leads to (i) the creation of a new version of this schema (of this document, respectively), (ii) the update of the temporal JSON schema in order to take into account this new schema version (this document version, respectively), (iii) the generation of a new conventional JSON document version, which is valid with respect to the new conventional JSON schema version, from each conventional JSON document version that was valid with respect to the previous conventional JSON schema version, and (iv) the update of the temporal JSON document in order to take into account the new conventional JSON document versions.

It is worth mentioning that, when the NSDBA changes the conventional JSON schema only (without changing the temporal characteristic document), the system could apply one of the following three strategies, according to the configuration parameters that are set by the NSDBA when he/she started exploiting the TJSchema framework:

Figure 1. Our technique for managing schema versioning in τ JSchema

A) Generate a new temporal characteristic document version associated to the new conventional JSON schema version, from the previous temporal characteristic document version, and ask the NSDBA to possibly update it, before saving it;

B) Assign default temporal logical characteristics (i.e., "anything can change") and default temporal physical characteristics (i.e., "a timestamp is located at the root") to the new conventional JSON schema version and ask the NSDBA to possibly update them, before saving them in a new temporal characteristic document version;

C) Inform the NSDBA that a new version of the conventional JSON schema will be added to the τ JSchema repository,

without an explicit definition of a new temporal characteristic document version, and ask him/her if he/she would like to choose A), B), or to (temporarily) keep this new conventional JSON schema version without an associated new temporal characteristic document version.

Notice also that, with the aim of making τ JSchema a flexible framework, some changes to a conventional JSON document version, like renaming a property or changing the value type of a property, also imply changes to the corresponding conventional JSON schema version.

In this paper, we are interested only in the changes of conventional JSON schemas and in their effects on the

rest of the related TJSchema documents, in a multiversion context.

4. Primitives for Changing Conventional JSON Schema, Temporal JSON Schema, Conventional JSON Documents and Temporal JSON Documents

In this section, we define primitives (i.e., low-level operations) for changing conventional JSON schemas (in subsection 4.2), temporal JSON schemas (in subsection 4.3), conventional JSON documents (in subsection 4.4), and temporal JSON documents (in subsection 4.5). For each one of these primitives, we provide a description of its arguments and its operational semantics. The definition of all these primitives is based on some common design principles and conventions that are presented in the subsection 4.1.

4.1 Design principles and conventions

The definition of the primitives will obey the following principles and conventions:

- All primitives must have a valid conventional JSON schema (CJS) (temporal JSON schema (TJS), conventional JSON document (CJD), or temporal JSON document (TJD), respectively) as input and must produce a valid CJS (TJS, CJD, or TJD, respectively) as output.
- All primitives need to work on a JSON schema file (JSON document file, respectively) storing the CJS (TJS, CJD, or TJD, respectively), whose name must be supplied as the first argument.
- For all primitives, arguments which are used to identify the object on which the primitive works are in the first place of the argument list.
- Components which are just containers for other components (e.g., conventionalJSONSchema, sliceSequence, and temporalCharacteristicSet, in the temporal JSON schema) can be managed by the primitives concerning the components, without specific primitives acting on them (i.e., the container is created when the first sub-component is created and is deleted when the last sub-component is deleted).

4.2 Primitives for changing a conventional JSON schema

Based on the complete JSON Schema specification, we define a set of thirteen primitives for changing a conventional JSON schema. Each schema change primitive acts on a JSON Schema component by creating, modifying or dropping it. For each primitive, we describe its arguments and explain its functioning.

a) CreateConventionalJSONSchema(CJS.json)

It creates a valid conventional JSON schema (CJS) file that contains an object with the two keywords "\$schema" and "id" having the values "http://json-schema.org/draft-04/schema#" and "http://jsonschema.net", respectively. In fact, these two keywords are required and must be

present in the root of the JSON Schema file. The argument of this primitive is the name of the JSON Schema file where the new CJS is stored.

The result of the application of this primitive is as follow:

CJS.json:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object"
}
```

b) DropConventionalJSONSchema(CJS.json)

It removes the conventional JSON Schema file "CJS.json" from the database.

c) AddPropertyToConventionalJSONSchema(CJS.json, targetComponentPath, position, propertyName, propertyType)

It adds a new property, with its propertyName and propertyType components, at a specified "position" (i.e., before, after, first, or last) with regard to a target schema component located at targetComponentPath. In fact, this latter can be either (i) the JSONPath [28] of the container where the property has to be added (i.e., the "properties" or the "additionalProperties" container), and in this case "position" must have the value "first" or "last" to express that this property will be the first or the last one in this container, respectively, or (ii) the JSONPath of another property in the same container, and in this case "position" must have the value "before" or "after" to express that this new property will be added before or after the property located at targetComponentPath, respectively. As for "propertyType", it must have one of the following values: string, number, boolean, object, array, or null.

Notice that in this work we have adopted the JSONPath language [28] to specify the paths of the JSON schema or JSON document components. Thus, in the rest of the paper, the "path" of a component means its JSONPath.

Example. If the NSDBA wants to add, as a first property,

CJS.json:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "prop1": {
      "type": "number"
    }
    ...
  }
}
```

a new property having the name “prop1” and the type “number” to the “properties” container, he/she can use the following primitive:

AddPropertyToConventionalJSONSchema(CJS.json, \$.properties, first, prop1, number). Its effect will be as follows (we mark with red color the portion modified with respect to the previous specification):

d) RenamePropertyInConventionalJSONSchema (CJS.json, propertyPath, newPropertyName)

It changes to newPropName the name of the property located at propertyPath.

Example. If the NSDBA wants to change the name of the property “prop1” to be “prop01”, he/she can use the following primitive:

RenamePropertyInConventionalJSONSchema (CJS.json, “\$.properties.prop1”, “prop01”). The effect of such a primitive is as follows:

effect of such a primitive is as follows:

```
CJS.json:
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "prop01":
    {
      "type": "number",
      "minimum": 20
    }
    ...
  }
}
```

Example 2. If the NSDBA wants to express that there are no additional properties, he/she can use the following primitive:

<pre>CJS.json: { "\$schema": "http://jsonschema.org/draft-04/schema#", "id": "http://jsonschema.net", "type": "object", "properties": { "prop1": {...}... } } Before change</pre>	<pre>CJS.json: { "\$schema": "http://json-schema.org/draft04/schema#", "id": "http://jsonschema .net", "type": "object", "properties": { "prop01": {...}... } } After change</pre>

e) DropPropertyFromConventionalJSONSchema (CJS.json, propertyPath)

It removes the property located at propertyPath, and all its components if they exist, from the JSON schema file CJS.json.

f) AddSimpleTypeKeywordToConventionalJSONSchema (CJS.json, keywordContainerPath, keywordName, keywordValue)

It adds a new keyword of simple type (i.e., string, number, or boolean), with its keywordName and keywordValue components, to a container located at keywordContainerPath.

Since the keyword that will be added is of simple type, keywordName must be one of the following JSON Schema keywords: title, description, type, multipleOf, maximum, exclusiveMaximum, minimum, exclusiveMinimum, maxLength, minLength, pettern, maxProperties, minProperties, additionalProperties, maxItems, minItems, uniqueItems, additionalItems.

Example 1. If the NSDBA wants to express that the prop01 property, which is of number type, must have a minimum value of 20, he/she can use the following primitive:

AddSimpleTypeKeywordToConventionalJSONSchema (CJS.json, “\$.properties.prop01”, minimum, 20). The

AddSimpleTypeKeywordToConventionalJSONSchema (CJS.json, “\$.”, additionalProperties, false). The effect

of such a primitive is as follows:

```
CJS.json
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": { ...
    "prop01": {...}
    ...
  }
  "additionalProperties": false }
```

g) AddObjectTypeKeywordToConventionalJSON Schema (CJS.json, keywordContainerPath, keyword Name)

It adds a new keyword of object type, named keywordName and having the value “{}” (i.e., an empty object), to the container located at keywordContainerPath.

Since the keyword that will be added is of object type, keywordName must be one of the following JSON Schema keywords: items, properties, additionalProperties, patternProperties, dependencies, not.

Example. If, contrarily to the previous Example, the

NSDBA wants to specify some additional properties indeed, he/she can start the job by means of the following primitive:

AddObjectTypeKeywordToConventionalJSONSchema(CJS.json, "\$.", additionalProperties). Its effect is as follows:

```
CJS.json
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": { ...
    "prop01": { ...
      ...
    }
  }
  "additionalProperties": { } }
```

As shown above, such a primitive will create an empty "additionalProperties" object (i.e., just the container) which can then be filled with the desired additional properties. After that, the NSDBA can use the primitive **AddPropertyToConventionalJSONSchema()** to complete the job.

h) AddArrayTypeKeywordToConventionalJSONSchema (CJS.json, keywordContainerPath, keywordName)

It adds a new empty keyword of array type, with its keywordName, to a container located at keywordContainerPath.

Since the keyword that will be added is of array type, keywordName must be one of the following JSON Schema keywords: type, required, enum, allOf, anyOf, oneOf.

Example. If the NSDBA wants to specify that some properties are required, he/she can start with the following primitive:

AddArrayTypeKeywordToConventionalJSONSchema (CJS.json, "\$.", required). Its effect is as follows:

```
CJS.json
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": { ...
    "prop01": { ...
      ...
    }
  }
  ...
  "required": [ ] }
```

As shown above, such a primitive will create an empty "required" array (i.e., just the container) which can then be filled with the desired required properties. To this aim, the NSDBA can use the primitive **AddItemToArrayInConventionalJSONSchema**.

Notice that an array can have one of the following three forms: an array of simple items (i.e., having simple types), an array of objects, or an array of arrays.

i) SetSimpleTypeKeywordInConventionalJSONSchema (CJS.json, keywordPath, newKeywordValue)

It sets in the CJS.json file the value of the keyword located at keywordPath, and having a simple type (i.e., string, number, boolean), to newKeywordValue.

Example. If the NSDBA wants to change the minimum value of the property prop01 from 20 to 10, he/she can use the following primitive:

SetKeywordInConventionalJSONSchema(CJS.json, "\$.properties.prop01.minimum", 10). The effect of such a primitive is as follows:

```
CJS.json
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "prop01": {
      {
        "type": "number"
        "minimum": 10
      }
    }
    ...
  }
}
```

On the other hand, in order to fill an array, we propose the two primitives that follow.

j) AddItemToArrayTypeKeywordInConventionalJSONSchema (CJS.json, arrayPath, targetItem, position, newItemValue)

It adds a new item, with its value newItemValue, to the array type keyword that is located at arrayPath, at a specified "position" (i.e., before, after, first, or last) with regard to a target item "targetItem". The newItemValue argument could be a value of simple type (i.e., string, number, boolean, null), an empty object (i.e., {}), or an empty array (i.e., []).

Example. If the NSDBA wants to add the property prop2 to the list of required properties, as the last item, he/she can use the following primitive:

AddItemToArrayTypeKeywordInConventionalJSONSchema(CJS.json, "\$.required", , last, "prop2"). The effect of such a primitive is as follows:

```
CJS.json
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
```



```

...
"prop2": {...}
...
}
...
"required": [..., prop2]
...

```

k) SetSimpleTypeItemOfArrayInConventionalJSONSchema (CJS.json, arrayPath, oldItemValue, newItemValue)

It replaces, in the array located at arrayPath, the old value of an item with a new one. Notice that both the old value and the new value are of simple type (i.e., number, boolean, string).

Example. Suppose that the NSDBA wants to correct, in the "enum" array of the conferenceRank property, the value "A+" with the value "A*". To this aim, he/she can use the following primitive:

SetSimpleTypeItemOfArrayInConventionalJSONSchema(CJS.json, "\$.properties.conferenceRank.enum", A+, A*). The effect of such a primitive is as follows:

```

CJS.json:
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    ...
    "conferenceRank": { "type": "string",
      "enum": ["A*", "A", "B", "C"] }
    ...
  } ... }

```

l) DropItemFromArrayInConventionalJSONSchema (CJS.json, arrayPath, itemValue)

It removes, from the array located at arrayPath, the item having the value "itemValue".

Example. Suppose that the NSDBA wants to remove, from the "enum" array of the conferenceRank property, the value "A*". Thus, he/she can use the following primitive: **DropItemFromArrayInConventionalJSONSchema(CJS.json, "\$.properties.conferenceRank.enum", A*).** The effect of such a primitive is as follows:

```

CJS.json:
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    ...
    "conferenceRank": { "type": "string",
      "enum": ["A", "B", "C"] }
    ...
  } ... }

```

Notice that we could propose another version of this primitive by replacing the two arguments "arrayPath" and "itemValue" with "arrayItemPath", so that it would become as follows:

DropItemFromArrayInConventionalJSONSchema(CJS.json, arrayItemValue). The second argument of this primitive denotes the index of the item that has to be deleted, with the first array index being 0. For example, "\$.tab[1]" refers to the second item of the array named "tab".

In this case, if the NSDBA wants to employ it for the same purpose of the example presented above, he/she can use the following primitive:

DropItemFromArrayInConventionalJSONSchema(CJS.json, "\$.properties.conferenceRank.enum[0])

m) DropKeywordFromConventionalJSONSchema (CJS.json, keywordPath)

It removes the keyword, located at keywordPath, and all its components if they exist, from the JSON schema file CJS.json.

4.3 Primitives for changing a temporal JSON schema

Before defining the primitives acting on a temporal JSON schema [21], we need first to know the structure of such a schema. To that end, we have defined the schema (TJSSchema.json) for the temporal JSON Schema; it is provided in Fig. 2.

```

{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "temporalJSONSchema": {
      "id": "http://jsonschema.net/temporalJSONSchema",
      "type": "object",
      "items": {
        "type": "object",
        "properties": {
          "conventionalJSONSchema": {
            "type": "object",
            "properties": {
              "sliceSequence": {
                "type": "array",
                "items": [ {
                  "type": "object",
                  "properties": {
                    "slice": {
                      "type": "object",
                      "properties": {
                        "location": {
                          "type": "string",
                          "begin": {
                            "type": "string"
                          }
                        }
                      }
                    }
                  }
                }
              },
              "temporalCharacteristicSet": {
                "type": "object",

```

```

    "properties": {
      "sliceSequence": {
        "type": "array",
        "items": [ {
          "type": "object",
          "properties": {
            "slice": {
              "type": "object",
              "properties": {
                "location": {
                  "type": "string",
                  "begin": {
                    "type": "string"
                  }
                }
              }
            }
          }
        } ]
      }
    }
  }
}

```

Figure 2. "TJSSchema.json", the schema of the temporal JSON Schema

Based on the schema of Fig. 2, we propose in this subsection a complete set of four primitives for changing a temporal JSON schema. These primitives are as follows:

a) CreateTemporalJSONSchema(TJS.json)

It creates an empty TJS. The argument is the name of the JSON file where the new TJS is stored. The result of the application of this primitive is as follows:

```

TJS.json
{
  "temporalJSONSchema": {}
}

```

b) DropTemporalJSONSchema(TJS.json)

It removes the TJS.json file, which represents the temporal JSON schema, from the database.

c) AddSliceToTemporalJSONSchema(TJS.json, toWhat, sourceSlice, targetSlice)

It adds a new object containing the property "slice" to the toWhat (i.e., "conventionalJSONSchema" or "temporalCharacteristicSet") container. The "slice" property is an object characterized by two properties: "location" (i.e., the name of the JSON schema file that represents the new conventional JSON schema version, or the name of the JSON file that represents the new version of the temporal characteristics document) and "begin" (i.e., the time that represents the transaction start time of the new conventional JSON schema version or the temporal characteristics document version).

Notice here that the "location" property will receive the value of the "targetSlice" argument (i.e., the name or the URL of a non-already existing JSON Schema/JSON file), and the "begin" property will receive the value of the current transaction time.

As for the argument sourceSlice, it could be one of the following three values: "empty", "current", or a specified JSON Schema/JSON file name (an URL).

i) If sourceSlice="empty", then the resource denoted by targetSlice is initialized to an empty conventional JSON schema or temporal characteristics document according to the toWhat value.

ii) If sourceSlice="current", then the resource pointed by targetSlice is initialized with a copy of the current conventionalJSONSchema/temporalCharacteristicSet resource (according to toWhat), whose location is found in the TJS.json file by choosing the slice with the latest value of begin in the corresponding sliceSequence container. Notice that this is the normal case after the creation of the first conventional JSON schema version or the first temporal characteristics document version.

iii) If sourceSlice provides a name (or a URL) of a JSON Schema/JSON file, then a copy of the specified file is renamed as targetSlice and used as the new location. This case occurs when an NSDBA wants to create a new conventional JSON schema version (a new temporal characteristics document version, respectively) from an already existing JSON Schema file (JSON file, respectively). Such a case could be quite common when creating the first conventional JSON schema version (the first temporal characteristics document version, respectively) but can also occur later for reuse purposes and/or when integrating independently developed JSON schemas (JSON documents, respectively) into a TJSchema framework.

Example. Suppose that on March 13, 2017, the NSDBA wants to specify the first conventional JSON schema version, named "PublicationsJSONSchema_V1.json", for scientific publications, as a slice of the new temporal JSON schema, named "PublicationsTemporalSchema.json". Thus, he/she can start by using the following primitive:

AddSliceToTemporalJSONSchema(PublicationsTemporalSchema.json, conventionalJSONSchema, empty, PublicationsJSONSchema_V1.json). The result of the application of such primitive is as follows:

```

PublicationsTemporalSchema.json:
{
  "temporalJSONSchema": {
    "conventionalJSONSchema": {
      "sliceSequence": [
        { "slice": {
          "location": "Publications
JSONSchema_V1.json",
          "begin": "2017-03-13"
        }
        }
      ]
    }
  }
}
and
PublicationsJSONSchema_V1.json:
{
}

```

After that, suppose that on the same day, the NSDBA

wants to specify some temporal logical and physical characteristics (that should be stored in a JSON file named "PublicationsTemporalCharacteristics_V1.json"), of some components of the first version of the conventional JSON schema version of publications. Therefore, he/she can proceed by using the following primitive:

AddSliceToTemporalJSONSchema(PublicationsTemporalSchema.json, temporalCharacteristicSet, empty, PublicationsTemporalCharacteristics_V1.json). The result of the application of such primitive is as follows:

```
PublicationsTemporalSchema.json:
{
  "temporalJSONSchema": {
    "conventionalJSONSchema": {
      "sliceSequence": [
        { "slice": {
          "location": "PublicationsJSONSchema
            V1.json",
          "begin": "2017-03-13"
        }
      ]
    },
    "temporalCharacteristicSet": {
      "sliceSequence": [
        { "slice": {
          "location": "PublicationsTemporal
            Characteristics_V1.json",
          "begin": "2017-03-13"
        }
      ]
    }
  }
}
and
PublicationsTemporalCharacteristics
_V1.json
{ }
```

d) DropSliceFromTemporalJSONSchema(TJS.json, fromWhat, sliceLocation)

It removes the "slice" object whose "location" property has the value "sliceLocation", from the "sliceSequence" array of the fromWhat (i.e., "conventionalJSONSchema" or "temporalCharacteristicSet") container.

4.4 Primitives for changing a conventional JSON document

In this subsection, we propose a complete set of eleven primitives of changing a conventional JSON document. These primitives are as follows:

a) CreateConventionalJSONDocument(CJD.json)

It creates an empty conventional JSON document, named CJD. The result of the application of this primitive is as follows:

```
CJD.json:
{ }
```

b) DropConventionalJSONDocument (CJD.json)

It removes the JSON file CJD.json from the database.

c) AddSimpleTypePropertyToConventionalJSON Document (CJD.json, targetPropertyPath, position, propertyName, propertyValue)

It adds a new simple type (string, number, boolean, or null) property, with its name (propertyName) and its value (propertyValue), at a specified position (i.e., before, after) with regard to a target property located at targetPropertyPath.

Example. Suppose that the NSDBA wants to add the age (30) of the first customer, after its last name. Thus, he/she can use the following primitive:

AddSimpleTypePropertyToConventionalJSONDocument(Customers.json, "\$.Customers[0].lastName", after, age, 30). The effect of such a primitive is as follows:

```
Customers.json:
{ "Customers": [
  {
    ...
    "lastName": "Tounsiya",
    "age": 30
  }
  ...
]
}
```

d) AddObjectTypePropertyToConventionalJSON Document (CJD.json, targetPropertyPath, position, propertyName)

It adds a new object type property, named "propertyName" and having the value "{}" (i.e., an empty object), at a specified position (i.e., before, after) with regard to a target property located at targetPropertyPath.

Example. Suppose that the NSDBA wants to add the address of the first customer, as an object characterized by four properties (i.e., number, street, city, postal code, country), after its age. Thus, he/she can start the job by using the following primitive:

AddObjectTypePropertyToConventionalJSONDocument (Customers.json, "\$.Customers[0].age", after, address). The effect of such a primitive is as follows:

```
Customers.json:
{ "Customers": [
  {
    ...
    "lastName": "Tounsiya",
    "age": 30,
    "address": { }
  }
  ...
]
}
```

e) AddArrayTypePropertyToConventionalJSON Document (CJD.json, targetPropertyPath, position, propertyName)

It adds a new array type property, named “propertyName” and having the value “[]” (i.e., an empty array), at a specified position (i.e., before, after) with regard to a target property located at targetPropertyPath.

Example. Suppose that the NSDBA wants to add the phones of the first customer, as an array of <“phoneNumber”, “type” (e.g., home, fax, mobile)> pairs, after its address. Thus, he/she can start by using the following primitive:

AddArrayTypePropertyToConventionalJSONDocument (Customers.json, “\$.Customers[0].address”, after, phones). The effect of such a primitive is as follows:

```
Customers.json:
{
  "Customers": [
    {
      ...
      "lastName": "Tounsiya",
      "age": 30,
      "address": {}
      "phones": []
    }
    ...
  ]
}
```

f) SetSimpleTypePropertyInConventionalJSON Document (CJD.json, propertyPath, newPropertyValue)

It sets the current value of the property located at propertyPath to a new value “newPropertyValue”. Notice that this property must be of simple type (string, number, boolean or null).

Example. Suppose that the NSDBA wants to correct the age (27 instead of 30) of the first customer. Thus, he/she can use the following primitive:

SetSimpleTypePropertyInConventionalJSONDocument (Customers.json, “\$.Customers[0].age”, 27). The effect of such a primitive is as follows:

```
Customers.json:
{
  "Customers": [
    {
      ...
      "lastName": "Tounsiya",
      "age": 27,
      "address": {},
      "phones": []
    }
    ...
  ]
}
```

g) RenamePropertyInConventionalJSONdocument (CJD.json, propertyPath, newPropertyName)

It changes the name of the property located at propertyPath to a new name “newPropertyName” in the conventional JSON schema stored in the CJD.json file.

Such an operation is needed, for instance, when adapting

a copy of a previous conventional JSON document version in order to generate a new one that is valid with respect to a newly created conventional JSON schema version.

Example. Suppose that the NSDBA wants to rename the property “lastName” of the first customer to be “familyName”. Thus, he/she can use the following primitive: **RenamePropertyInConventionalJSONDocument (Customers.json, “\$.Customers[0].lastName”, “familyName”).** The effect of such a primitive is as follows:

```
Customers.json:
{
  "Customers": [
    {
      ...
      "familyName": "Tounsiya",
      "age": 27,
      "address": {},
      "phones": []
    }
    ...
  ]
}
```

h) DeletePropertyFromConventionalJSONDocument (CJD.json, propertyPath)

It removes the property located at propertyPath, and all its components, from the JSON file CJD.json.

Example. Suppose that the NSDBA wants to remove the age of the first customer. To this end, he/she can use the following primitive:

DeletePropertyFromConventionalJSONDocument (Customers.json, “\$.Customers[0].age”). The effect of such a primitive is as follows:

<pre>Customers.json: { "Customers": [{ ... "familyName": "Tounsiya", "age": 27, "address": {...}, "phones": [...] } ...] }</pre>	<pre>Customers.json: { "Customers": [{ ... "familyName": "Tounsiya", "address": {...}, "phones": [...] } ...] }</pre>
Before change	After change

i) AddValueToArrayInConventionalJSONDocument (CJD.json, targetPath, position, newValue)

It adds a new value (newValue) to an array at a specified position (i.e., before, after, first, or last) with respect to a target located at targetPath. Notice that this latter could be either the path of the array (if one would like to add a value as the first or the last one of this array; in this case, the array could be empty), or the path of a value in an array (in this case the array must not be empty). Further, the new value could be of simple type (i.e., a string,

number, or boolean value), an empty object (i.e., {}), or an empty array (i.e., []).

Example. If the NSDBA wants to add “schema” as the first keyword associated to the first paper in the “Publications.json” file, he/she can use the following primitive:

AddValueToArrayInConventionalJSONDocument (Publications.json, “\$.papers[0].keywords”, first, “schema”). The effect of such a primitive is as follows:

Publications.json:
{ "papers":
[
{
"title":"title1",
"year":2008,
...
"keywords": []
}
...
] }
Before change
	Publications.json:
	{ "papers":
	[
	{
	"title":"title1",
	"year":2008,
	...
	"keywords":
	["schema"]
	}
	...
	] }
	After change

j) SetSimpleTypeValueInArrayInConventionalJSON Document (CJD.json, valuePath, newValue)

It changes to newValue the value of an array located at valuePath, in the CJD.json file. Notice that the new value must be of simple type (i.e., a string, a number, a null, or a boolean value).

Example. If the NSDBA wants to change to “database schema” the first keyword associated to the first paper in the “Publications.json” file, he/she can use the following primitive:

SetSimpleTypeValueInArrayInConventionalJSON Document(Publications.json,“\$.papers[0].keywords[0]”, “database schema”). The effect of such a primitive is as follows:

Publications.json:
{ "papers":
[
{
"title":"title1",
"year":2008,
...
"keywords":
["schema"]
}
...
] }
Before change
	Publications.json:
	{ "papers":
	[
	{
	"title":"title1",
	"year":2008,
	...
	"keywords":
	["database schema"]
	}
	...
	] }
	After change

k) DeleteValueFromArrayInConventionalJSON Document (CJD.json, valuePath)

It removes, from an array, the value located at valuePath.

Notice that this latter could refer to a value of simple type (i.e., string, number, boolean), to an object, or to an array.

Example. Suppose that the NSDBA wants to remove the fax of the first customer; notice here that the fax of any customer, if it exists, is an item of the array phones. Thus, to this aim, he/she can use the following primitive: **DeleteValueFromArrayInConventionalJSONDocument(Customers.json, “\$.Customers[0].phones[?(@.type = fax)]”).** The effect of such a primitive is as follows:

Customers.json:
{ "Customers": [
{
...
"familyName":
"Tounsiya",
"age":27,
"address": {},
"phones": [
{
"type": "home",
"Number": "12345678"
},
{
"type": "fax",
"Number": "11223344"
}
] }
] }
Before change
	Customers.json:
	{ "Customers": [
	{
	...
	"familyName":
	"Tounsiya",
	"age":27,
	"address": {},
	"phones": [
	{
	"type": "home",
	"Number": "12345678"
	}
	] }
	] }
	After change

4.5 Primitives for changing a temporal JSON document

Similarly to the logics followed in the subsection 4.3, we need first to know the structure of a temporal JSON document [21] before defining the primitives acting on it. For this purpose, we have defined the schema (TJDSchema.json) for the temporal JSON document; it is provided in Fig. 3.

<pre>{ "\$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "temporalRoot": {
      "id": "http://jsonschema.net/temporalRoot",
      "type": "object",
      "items": {
        "type": "object",
        "properties": {
          "temporalJSONSchema": {
            "type": "object",
            "properties": {
              "location": { "type":
"string" }
            } },
          "sliceSequence": {
            "type": "array",
            "items": [{
              "type": "object",
              "properties": {</pre>

```
"slice": {
  "type": "object",
  "properties": {
    "location": {
      "type": "string" },
    "begin": {
      "type": "string" }
  } } } ] } } } } }
```

Figure 3. “TJDSchema.json”, the schema of the temporal JSON document

Based on the schema of Fig. 3, we propose in this subsection a complete set of four primitives for changing a temporal JSON document. These primitives are as follows:

a) **CreateTemporalJSONDocument(TJD.json, TJS.json)**

It creates an empty temporal JSON document TJD. The first argument is the name of the JSON file where the new TJD is stored and the second argument is the TJS associated to this TJD (i.e., the TJS associated to all temporal conventional JSON document versions that will be referenced by the TJD). The result of the application of this primitive is as follows:

```
TJD.json:
{
  "temporalRoot": {
    "temporalJSONSchema": {
      "location": "TJS.json"
    }
  }
}
```

b) DropTemporalJSONDocument(TJD.json)

It removes the TJD.json file, which represents the temporal JSON document, from the database.

c) **AddSliceToTemporalJSONDocument(TJD.json, sourceTJDSlice, targetTJDSlice)**

It adds a new object containing the property “slice” to the sliceSequence container. The “slice” property is an object characterized by two properties: “location” (i.e., the name of the JSON file that represents the new conventional JSON document version) and “begin” (i.e., the time that represents the transaction start time of the new conventional JSON document version).

Notice here that the “location” property will receive the value of the “targetTJDSlice” argument (i.e., the name or the URL of a non-already existing JSON file), and the “begin” property will receive the value of the current transaction time.

As far as the argument `sourceTJDSlice` is concerned, it can be one of the following three values: “empty”, “current”, or a specified JSON file name (or a URL).

i) If sourceTJDSlice="empty", then the resource pointed by targetTJDSlice is initialized to an empty conventional JSON document.

ii) If sourceTJDSlice="current", then the resource pointed by targetTJDSlice is initialized with a copy of the current conventional JSON schema version, whose location is found in the TJS.json file by choosing the slice with the latest value of "begin" in the sliceSequence container corresponding to the conventional schema. Notice that this is the normal case after the creation of the first conventional JSON document version.

iii) If `sourceTJDSlice` provides a name (or a URL) of a JSON file, then a copy of the specified file is renamed as `targetTJDSlice` and used as the new location. This option is used to create a new conventional JSON document version from an already existing JSON file, which could be quite common when creating the first conventional JSON document version but can also be used later for reuse purpose and/or when integrating independently developed JSON documents into a `TJSchema` framework.

Example. Suppose that on March 13, 2017, the NSDBA wants to specify the first conventional JSON document version, named “Publications_V1.json”, for scientific publications, as a slice of the temporal JSON document, named “PublicationsTemporalDocument.json”. Thus, he/she can start by using the following primitive:

AddSliceToTemporalJSONDocument(Publications
TemporalDocument.json, empty, Publications_V1.
json). The result of the application of such primitive is as follows:

```

PublicationsTemporalDocument.json:
{
  "temporalRoot": {
    "temporalJSONSchema": {
      "location": "PublicationsTemporalSchema.json"
    },
    "scliceSequence": [
      {
        "slice": {
          "location": "Publications_V1.json",
          "begin": "2017-03-13"
        }
      }
    ]
  }
}

and

Publications_V1.json
{

```

d) DropSliceFromTemporalJSONDocument(TJD.json, locationTJDSlice)

It removes the “slice” object whose “location” property has the value “locationTJDSlice”, from the “sliceSequence” container.

5. Application example

As a motivating application example, we consider the management of a temporal JSON-based NoSQL database storing data concerning Youtube channels we introduced

in [22]. We assume that on the 15th of March 2017, our TJSchema repository contains the following documents:

- The first version of the conventional JSON schema shown in Fig. 4. In this schema, each Youtube channel is described by its name, its owner, the number of its subscribers, and the list of its videos; each video is described by its name, its URL, the number of its likes, the number of its dislikes, and the number of its shares.
- The first version of the document that stores temporal logical and physical characteristics shown in Fig. 5, associated to some components of the conventional JSON schema (cf. Fig. 4). In fact, as to temporal logical characteristics, the NSDBA has decided to make the contents of the following properties varying in valid-time: the name and the owner of a Youtube channel, and the name of a video in a Youtube channel. As to temporal physical characteristics, the NSDBA has chosen to add a transaction-time physical timestamp to the array property "videos" (i.e., whenever any item in the array

"videos" changes, a new timestamped version of the entire array "videos" is created).

- The temporal JSON schema shown in Fig. 6, that ties together the conventional JSON schema (cf. Fig. 4) and the temporal characteristics document (cf. Fig. 5).

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "YouTubeChannels": {
      "id": "http://jsonschema.net/YouTubeChannels",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": { "type": "string" },
          "owner": { "type": "string" },
          "subscribedNumber": { "type": "number" },
          "videos": { "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "name": { "type": "string" },
                "URL": { "type": "string" },
                "likeNumber": { "type": "number" },
                "dislikeNumber": { "type": "number" },
                "shareNumber": { "type": "number" }
              }
            }
          }
        }
      },
      "required": [ "name", "owner", "subscribedNumber", "videos" ],
      "required": [ "0" ]
    }
  },
  "required": [ "YouTubeChannels" ] }
```

Figure 4. The first version of the conventional JSON schema (youtubeChannelsConventionalSchema_V1.json) on March 15, 2017

```
{ "temporalCharacteristicSet": {
  "logical": [
    {
      "target": "$.properties.YouTubeChannels.items.properties.name",
      "validTime": { "kind": "state",
        "content": "varying",
        "existence": "constant" }
    },
    {
      "target": "$.properties.YouTubeChannels.items.properties.owner",
      "validTime": { "kind": "state",
        "content": "constant",
        "existence": "constant" }
    },
    {
      "target": "$.properties.YouTubeChannels.items.properties.videos.items.properties.name",
      "validTime": { "kind": "state",
        "content": "varying",
        "existence": "constant" }
    }
  ],
  "physical": [
    {
      "target": "$.properties.YouTubeChannels.items.properties.videos",
      "dataInclusion": "expandedVersion",
      "stampKind": {
        "timeDimension": "transactionTime",
        "stampBounds": "extent"
      }
    }
  ]
}
```

Figure 5. The temporal characteristics document (youtubeChannelsTemporalCharacteristics_V1.json) on March 15, 2017

```
{ "temporalJSONSchema": {
  "convetionalJSONSchema": {
    "scliceSequence": [ {
      "slice": {
        "location": "youtubeChannelsConvetionalSchema_V1.json",
        "begin": "2016-11-10"
      }
    } ],
  "temporalCharacteristicSet": {
    "scliceSequence": [ {
      "slice": {
        "location": "youtubeChannelsTemporalCharacteristics_V1.json",
        "begin": "2016-11-10"
      }
    } ]
  }
}
```

Figure 6. The temporal JSON schema (youtubeChannelsTemporalSchema.json) on March 15, 2017

- The first and the second versions of the conventional

JSON document that stores Youtube channel data in Fig. 7 and Fig. 8, respectively. These versions are valid with respect to the conventional JSON schema of Fig. 4.

• The temporal JSON document shown in Fig. 9, that links each conventional JSON document version (cf. Figures 7 and 8), which is valid with respect to the conventional JSON schema (cf. Fig. 4), to its corresponding temporal JSON schema (cf. Fig. 6), and more precisely to its corresponding temporal logical and physical characteristics (which are referenced by the temporal JSON schema).

```
{ "YouTubeChannels": [
  { "name": "Big Data videos",
    "owner": "Amira",
    "subscribedNumber": 60000,
    "videos": [
      { "name": "Big Data Technologies",
        "URL": "https://www.youtube.com/watch?v=BDT",
        "likeNumber": 100,
        "dislikeNumber": 5,
        "shareNumber": 300 },
      { "name": "Big Data Phenomenon",
        "URL": "https://www.youtube.com/watch?v=BDPP",
        "likeNumber": 50,
        "dislikeNumber": 2,
        "shareNumber": 100 } ] ] }
```

Figure 7. The first version of the conventional JSON document (youtubeChannels_V1.json) on March 15, 2017

```
{ "YouTubeChannels": [
  { "name": "Big Data channel",
    "owner": "A. Tounsiya",
    "subscribedNumber": 60000,
    "videos": [
      { "name": "Big Data Management: Current Approaches and Future Trends",
        "URL": "https://www.youtube.com/watch?v=BDT",
        "likeNumber": 100,
        "dislikeNumber": 5,
        "shareNumber": 300 },
      { "name": "Big Data Phenomenon",
        "URL": "https://www.youtube.com/watch?v=BDPP",
        "likeNumber": 50,
        "dislikeNumber": 2,
        "shareNumber": 100 } ] ] }
```

Figure 8. The second version of the conventional JSON document (youtubeChannels_V2.json) on March 15, 2017

```
{ "temporalRoot": {
  "temporalJSONSchema": {
    "location": "youtubeChannelsTemporalSchema.json",
    "sliceSequence": [ { "slice": {
      "location": "youtubeChannels_V1.json",
      "begin": "2016-11-10" } },
      { "slice": {
        "location": "youtubeChannels_V2.json",
        "begin": "2017-01-20" } } ] }
```

Figure 9. The temporal JSON document (youtubeChannelsTemporalDocument.json) on March 15, 2017

In order to show how conventional JSON schemas are versioned in the TJSchema framework, from one hand, and to illustrate the use the proposed primitives to perform changes on a conventional JSON schema while showing their effects at both instance and schema levels, from the other hand, we assume now that the NSDBA decides to make some changes to the current version of the conventional JSON schema (i.e., the first version shown in Fig. 4), in order to cope with some changes in the code of the applications that exploit such a JSON schema. The list of schema changes to be applied is as follows:

1. rename the property "name" of a YouTube channel to "channelName";
2. rename the property "name" of a video, in a YouTube channel, to "videoName";
3. remove the property "dislikeNumber" of a video, in a YouTube channel;
4. remove the property "shareNumber" of a video, in a YouTube channel;
5. add a new property, named "viewsNumber" and having the "number" type, to the properties of a video of a YouTube channel;
6. add a new property, named "duration" and having the "number" type, to the properties of a video of a YouTube channel;
7. add a new property, named "creationDate" and having the "string" type, to the properties of a video of a YouTube channel;
8. add a new property, named "comments" and defined as an array of objects (where each object represents a distinct objects), to the properties of a video of a YouTube channel; each comment (i.e., an item of this array) is described by three properties with string type: "youtubeUser", "content", and "commentDate".

When the NSDBA has finished changing the first conventional JSON schema version and asks the system to commit his/her work, the following tasks are performed by the system in order to complete the schema change transaction:


```
{ "$schema": "http://json-schema.org/draft-
    04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "YouTubeChannels": {
      "id": "http://jsonschema.net/
        YouTubeChannels",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "channelName": { "type": "string" },
          "owner": { "type": "string" },
          "subscribedNumber": { "type":
            "number" },
          "videos": {
            "type": "array",
            "items": [ {
              "type": "object",
              "properties": {
                "videoName":
                  { "type": "string" },
                "URL":
                  { "type": "string" },
                "likeNumber":
                  { "type": "number" },
                "viewsNumber":
                  { "type": "number" },
                "duration":
                  { "type": "number" },
                "creationDate":
                  { "type": "string" },
                "comments": {
                  "type": "array",
                  "items": {
                    "type": "object",
                    "properties": {
                      "youtubeUser": {
                        "type": "string" },
                      "content": {
                        "type": "string" },
                      "commentDate": {
                        "type": "string" }
                    }
                  }
                }
              }
            }
          ],
          "required": [ "channelName",
            "owner",
            "subscribedNumber",
            "videos" ] },
          "required": [ "0" ] }
        },
        "required": [ "YouTubeChannels" ] }
```

Figure 10. The second version of the conventional JSON schema (youtubeChannelsConventionalSchema_V2.json) on March 15, 2017

- i. generating the second version conventional JSON schema shown in Fig. 10;
- ii. updating the temporal JSON Schema, as shown in Fig.

11, by adding a new slice related to the new conventional JSON schema version;

- iii. generating two new conventional JSON document versions, valid with respect to the second conventional JSON schema version, from the two conventional JSON document versions valid with respect to the first conventional JSON schema version: the third version shown in Fig. 12 from the first one shown in Fig. 7, and the fourth version shown in Fig. 13 from the second one shown in Fig. 8;

```
{ "temporalJSONSchema": {
  "conventionalJSONSchema": { "sliceSequence": [
    { "slice": {
      "location": "youtubeChannelsConventionalSch
        ema_V1.json",
      "begin": "2016-11-10" } },
    { "slice": {
      "location": "youtubeChannelsConventionalSch
        ema_V2.json",
      "begin": "2017-03-15" } } ] },
  "temporalCharacteristicSet": { "sliceSequence": [
    { "slice": {
      "location": "youtubeChannelsTemporalCharact
        eristics_V1.json",
      "begin": "2016-11-10" } } ] } }
```

Figure 11. The temporal JSON schema (youtubeChannelsTemporalSchema.json) on March 15, 2017

```
{ "YouTubeChannels": [
  { "channelName": "Big Data videos",
    "owner": "Amira",
    "subscribedNumber": 60000,
    "videos": [
      { "videoName": "Big Data Technologies",
        "URL": "https://www.youtube.com/
watch?v=BDT",
        "likeNumber": 100,
        "viewsNumber": 0,
        "duration": 0,
        "creationDate": "",
        "comments": [ { "youtubeUser": "",
          "content": "",
          "commentDate": "" } ] },
      { "videoName": "Big Data Phenomenon",
        "URL": "https://www.youtube.com/
watch?v=BDPP",
        "likeNumber": 50,
        "viewsNumber": 0,
        "duration": 0,
        "creationDate": "",
        "comments": [ { "youtubeUser": "",
          "content": "",
          "commentDate": "" } ] } ] } }
```

Figure 12. The third version of the conventional JSON document (youtubeChannels_V3.json) on March 15, 2017

iv. updating the temporal JSON document, as shown in Fig. 14, by adding two new slices related to the two new conventional JSON schema versions.

```
{ "YouTubeChannels": [ {
  "channelName": " Big Data channel ",
  "owner": " A. Tounsiya ",
  "subscribedNumber": 60000,
  "videos": [
    { "videoName": "Big Data Management:
Current
Approaches and Future
Trends",
  "URL": "https://www.youtube.com/
watch?v=BDT",
  "likeNumber": 100,
  "viewsNumber": 0,
  "duration": 0,
  "creationDate": "",
  "comments": [ { "youtubeUser": "",
  "content": "",
  "commentDate": "" } ] },
    { "videoName": "Big Data Phenomenon",
  "URL": "https://www.youtube.com/
watch?v=BDPP",
  "likeNumber": 50,
  "viewsNumber": 0,
  "duration": 0,
  "creationDate": "",
  "comments": [ { "youtubeUser": "",
  "content": "",
  "commentDate": "" } ] } ] ] }
```

Figure 13. The fourth version of the conventional JSON document (youtubeChannels_V4.json) on March 15, 2017

```
{ "temporalRoot": {
  "temporalJSONSchema": {
    location: "youtubeChannelsTemporalSchema.js
n" },
  "sliceSequence": [
    { "slice":
    { "location": "youtubeChannels_V1.json",
      "begin": "2016-11-10" } },
    { "slice":
    { "location": "youtubeChannels_V2.json",
      "begin": "2017-01-20" } },
    { "slice":
    { "location": "youtubeChannels_V3.json",
      "begin": "2017-03-15" } },
    { "slice":
    { "location": "youtubeChannels_V4.json",
      "begin": "2017-03-15" } } ] }
```

Figure 14. The temporal JSON document (youtubeChannelsTemporalDocument.json) on March 15, 2017

Notice that, in each one of the new conventional/temporal

JSON schema/document versions (shown in figures 10, 11, 12, 13, and 14), we present in red color the specifications which come out different with regard to the corresponding conventional/temporal JSON schema/document version that precedes it.

The sequence of primitives that have been applied to the temporal JSON schema (youtubeChannelsTemporalSchema.json), to the first version of the conventional JSON schema (youtubeChannelsConventionalSchema_V1.json), to the first version of the conventional JSON document (youtubeChannels_V1.json), to the second version of the conventional JSON document (youtubeChannels_V2.json), and to the temporal JSON document (youtubeChannelsTemporalDocument.json), in order to update the temporal JSON schema, to produce the second version of the conventional JSON schema (youtubeChannelsConventionalSchema_V2.json), to produce the third and the fourth versions of the conventional JSON document (youtubeChannels_V3.json and youtubeChannels_V4.json), and to update the temporal JSON document, can be embedded in the transaction that is provided in the Appendix as Fig. 15.

Appendix

In this appendix, we illustrate the use of the proposed JSON schema/document change primitives by presenting the schema change transaction that could be executed in order to satisfy the requirements of the designer in Section 5. Fig. 15 shows the details of this transaction.

```
Begin transaction
/* Changes to the temporal JSON schema in
order to generate and take into account
the second conventional JSON schema ver-
sion */
i) AddSliceToTemporalJSONSchema (
  youtubeChannelsTemporalSchema.json,
  conventionalSchema,
  youtubeChannelsConventionalSchema_V1.json,
  youtubeChannelsConventionalSchema_V2.json)
/* Changes to the file representing the
second conventional JSON schema version
*/
ii) RenamePropertyInConventionalJSONSchema (
  youtubeChannelsConventionalSchema_V2.json,
  $.properties.YouTubeChannels.items.properties.name,
  channelName)
iii) RenamePropertyInConventionalJSONSchema (
  youtubeChannelsConventionalSchema_V2.json,
  $.properties.YouTubeChannels.items.properties.
  videos.items.properties.name, videoName)
iv) DropPropertyFromConventionalJSONSchema (
  youtubeChannelsConventionalSchema_V2.json, $.properties.
  YouTubeChannels.items.properties.
  videos.items.properties.dislikeNumber)
v) DropPropertyFromConventionalJSONSchema (
  youtubeChannelsConventionalSchema_V2.json, $.
  properties.YouTubeChannels.items.properties.
  videos.items.properties.shareNumber)
```

```

vi) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $.properties.YouTubeChannels.items.
    properties.videos.items.properties, last,
    viewsNumber, number)
vii) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $.properties.YouTubeChannels.items.properties.
    videos.items.properties, last, duration,
    number)
viii) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $.properties.YouTubeChannels.items.properties.
    videos.items.properties, last,
    creationDate, string)
ix) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $..videos.items.properties, last, com
    ments, array)
x) AddObjectTypeKeywordToConventional
    JSON Schema (
    youtubeChannelsConventionalSchema_V2.json,
    $..videos.items.properties.comments,
    items)
xi) AddObjectTypeKeywordToConventional
    JSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $..videos.items.properties.comments.items,
    properties)
xii) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $..videos.items.properties.comments.items.properties,
    first, youtubeUser, string)
xiii) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $..videos.items.properties.comments.items.properties,
    last, content, string)
xiv) AddPropertyToConventionalJSONSchema (
    youtubeChannelsConventionalSchema_V2.json,
    $..videos.items.properties.comments.items.properties,
    last, commentDate, string)
/* Changes to the file representing the
temporal JSON document in order to gener-
ate and take into account the third con-
ventional JSON document version */
xv) AddSliceToTemporalJSONDocument (
    youtubeChannelsTemporalDocument.json,
    youtubeChannels_V1.json,
    youtubeChannels_V3.json)
/* Changes to the file representing the
third conventional JSON document version */
xvi) RenamePropertyInConventionalJSON
    Document (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].name, channelName)
/* changes to the first video ($.vid-
eos[0]) in youtubeChannels_V3.json */
xvii) RenamePropertyInConventionalJSON
    Document (

```

```

    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].name,
    videoName)
xviii) DeletePropertyFromConventionalJSON
    Document (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].dislikeNumber)
xix) DeletePropertyFromConventionalJSON
    Document (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].shareNumber)
xx) AddSimpleTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].likeNumber,
    after, viewsNumber, 0)
xxi) AddSimpleTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].viewsNumber,
    after, duration, 0)
xxii) AddSimpleTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].duration,
    after, creationDate, "")
xxiii) AddArrayTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].creationDate,
    after, comments)
xxiv) AddValueToArrayInConventionalJSON
    Document (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].comments,
    first, {})
xxv) AddSimpleTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].comments[0],
    first, youtubeUser, "")
xxvi) AddSimpleTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[0].comments[0].youtubeUser,
    after, content, "")
xxvii) AddSimpleTypePropertyToConventional
    JSONDocument (
    youtubeChannels_V3.json
    $.YouTubeChannels[0].videos[0].comments[0].content,
    after, commentDate, "")
/* changes to the second video ($.vid-
eos[1]) in youtubeChannels_V3.json */
xxviii) RenamePropertyInConventionalJSON
    Document (
    youtubeChannels_V3.json,
    $.YouTubeChannels[0].videos[1].name,
    videoName)
xxix) DeletePropertyFromConventionalJSON
    Document (

```

```

youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].dislikeNumber)
xxx) DeletePropertyFromConventionalJSON
Document (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].shareNumber)
xxxi) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].likeNumber,
after, viewsNumber, 0)
xxxii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].viewsNumber,
after, duration, 0)
xxxiii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].duration,
after, creationDate, "")
xxxiv) AddArrayTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].creationDate,
after, comments)
xxxv) AddValueToArrayInConventionalJSON
Document (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].comments,
first, {})
xxxvi) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].comments[0],
first, youtubeUser, "")
xxxvii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json,
$.YouTubeChannels[0].videos[1].comments[0].youtubeUser,
after, content, "")
xxxviii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V3.json
$.YouTubeChannels[0].videos[1].comments[0].content,
after, commentDate, "")
/* Changes to the file representing the
temporal JSON document in order to gener-
ate and take into account the fourth con-
ventional JSON document version */
xxxix) AddSliceToTemporalJSONDocument (
youtubeChannelsTemporalDocument.json,
youtubeChannels_V2.json,
youtubeChannels_V4.json)
/* Changes to the file representing the
fourth conventional JSON document version
*/
xI) RenamePropertyInConventionalJSON
Document (
youtubeChannels_V4.json,

```

```

$.YouTubeChannels[0].name, channelName)
/* Changes applied on the first video
($.videos[0]) in youtubeChannels_V4.json
*/
xIi) RenamePropertyInConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].name,
videoName)
xIii) DeletePropertyFromConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].dislikeNumber)
xIiii) DeletePropertyFromConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].shareNumber)
xIiv) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].likeNumber,
after, viewsNumber, 0)
xIv) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].viewsNumber,
after, duration, 0)
xIvi) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].duration,
after, creationDate, "")
xIvii) AddArrayTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].creationDate,
after, comments)
xIviii) AddValueToArrayInConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].comments,
first, {})
xIix) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].comments[0],
first, youtubeUser, "")
I) AddSimpleTypePropertyToConventional
AJSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[0].comments[0].youtubeUser,
after, content, "")
Ii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json
$.YouTubeChannels[0].videos[0].comments[0].content,
after, commentDate, "")
/* changes applied on the second video
($.videos[1]) in youtubeChannels_V4.json
*/

```



```

Iii) RenamePropertyInConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].name,
videoName)
Iiii) DeletePropertyFromConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].
dislikeNumber)
Iiv) DeletePropertyFromConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].
shareNumber)
Iv) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].likeNumber,
after, viewsNumber, 0)
Ivi) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].viewsNumber,
after, duration, 0)
Ivii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].duration,
after, creationDate, "")
Iviii) AddArrayTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].creationDate,
after, comments)
Iix) AddValueToArrayInConventionalJSON
Document (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].comments,
first, {})
Ix) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].comments[0],
first, youtubeUser, "")
Ixi) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json,
$.YouTubeChannels[0].videos[1].comments[0].
youtubeUser,after, content, "")
Ixii) AddSimpleTypePropertyToConventional
JSONDocument (
youtubeChannels_V4.json
$.YouTubeChannels[0].videos[1].comments[0].content,
after, commentDate, "")
Commit

```

Figure 15. The schema change transaction associated to the application example of Section 5

6. Related Work Discussion

In this section, we discuss the state-of-the-art of schema changes in NoSQL databases and try to clarify our contribution with regard to it.

After studying the state-of-the-art of schema change management in the NoSQL setting, we have to conclude that, to the best of our knowledge, there is no work that has dealt with schema versioning in NoSQL databases. All existing research works that considered schema changes in NoSQL databases have applied the schema evolution technique (i.e., only the last schema version with its underlying instances is kept). These works are described below.

Liu et al. [29] have proposed an approach for schema evolution in column-oriented databases that supports all schema modification operations (i.e., create, drop, copy, and rename a table, decompose a table into two tables, merge two tables into one table, and add and drop a column) previously proposed by Curino et al. [30] for managing relational schema evolution. The authors have focused on decomposing and merging operations by proposing corresponding algorithms and showing that these operations are more efficient in column-oriented stores than in traditional (i.e., tuple-oriented) stores.

Since 2013, Scherzinger and her colleagues have published interesting works on schema evolution in document-oriented NoSQL data stores. Their contributions ([9-10]; [14-17]; [31]) are presented in the following.

In [14], the authors proposed a set of five primitives (i.e., addition, deletion, rename, move, and copy of entities' properties) for changing structures of entities in a document-oriented NoSQL database. However, only one version of the structure of each entity is kept (i.e., the schema evolution technique is adopted). Contrarily to [14], we (i) keep all versions of a conventional JSON schema, with its associated conventional JSON instances, and (ii) provide a complete list of primitives for changing conventional JSON schema and conventional JSON instances.

Scherzinger et al. [15] presented a tool, named Cleager, that implements the set of primitives proposed in [14]; they are executed as MapReduce jobs on the Google Cloud Platform. Cleager propagates schema changes in an eager way, as each change to the structure of an entity is automatically and directly propagated to all instances of the entity. With respect to [15], our approach supports more schema change operations. Like in our proposal, their approach assume to automatically generate a new conventional JSON document version, valid with respect to the new conventional JSON schema version, from each conventional JSON document version that was valid with respect to the previous conventional JSON schema version.

In [30], the authors showed that existing Object-NoSQL Mappers (e.g., Hibernate OGM, DataNucleus, EclipseLink) support some basic schema evolution operations (e.g., adding or deleting an attribute of an entity). However, they do not allow the execution of complex schema changes or of combinations of simple schema changes and, thus, developers must proceed in an ad hoc manner in order to apply non trivial changes. Our approach allows developers to perform, from one hand, any number of schema changes, and, from the other hand, any type (i.e., simple or complex) of schema change on a conventional JSON schema, thanks to the proposed complete set of primitives which allow the expression of any high-level schema change operation via a suitable sequence of primitives.

Scherzinger et al. [16] and Cerqueus et al. [9-10] introduced an Eclipse plugin, named ControVol, which allows a lazy conversion of instances without problems (e.g., execution errors). In fact, it allows the detection of schema changes (like adding, deleting, and renaming properties of entities) in the application code that are not compatible with structures of stored entities, reporting warnings to application programmers, and proposing solutions to overcome such warnings. This plugin is implemented on top of the Google Cloud Datastore using the Object-NoSQL Mapper Objectify. Our approach does not deal with schema changes in application code sources, but considers that schema changes are done by the NSDBA through his/her interface as a response to changes in the real world (e.g., deriving from, application programmers, decision makers, or users), and requiring a schema change. Furthermore, our approach applies an eager instance conversion, since we consider that is more suitable for both a JSON-based NoSQL database (i.e., a document-oriented one) and a schema versioning approach.

Scherzinger et al. [17] presented a tool, named Datalution, that supports the operations previously proposed in [14]. It is used to demonstrate the difference between the two techniques of data migration/conversion, that is eager migration and lazy migration, in a JSON-based NoSQL data store, and to prove that lazy migration is more efficient than the other technique. Contrarily to [17], our approach automatically propagates schema changes to all involved instances.

Besides, in [13], the authors proposed a tool-supported approach, named KVMove, for managing schema evolution in a JSON-based key-value NoSQL database adopting a lazy migration strategy. KVMove is developed on top of the NoSQL system Redis. It allows application developers to specify (i) changes to structures of JSON instances (i.e., adding, dropping, or modifying a field) and (ii) updates that should be effected on stored JSON instances in order to convert them to their new structures. The authors show through the evaluation of the KVMove tool that their proposal reduces downtime when data are being migrated and application source code is being revised. Contrarily to

[13], our approach proposes a language for specifying changes to JSON schema and another language for changing JSON documents, in a multiversion and temporal setting, while applying an eager/immediate conversion technique.

Notice that in the NoSQL literature, versioning has only been studied at instance level, like in [32] for graph-based NoSQL databases, and in [8] for distributed key-value NoSQL databases. Therefore, we can say that we are the first to deal with versioning at schema level, in a JSON-based document-oriented NoSQL database.

7. Conclusion

In this paper, we have extended our TJSchema framework [21] to support temporal schema versioning, by;

- i) proposing an approach for creating and evolving temporal JSON schemas;
- ii) defining four complete sets of primitives for changing;
  - a. conventional JSON schemas,
  - b. temporal JSON schemas,
  - c. conventional JSON documents,
  - d. temporal JSON schemas;
- iii) illustrating such an approach and the usage of such primitives through an application example.

Our approach makes the specification and the evolution of a conventional JSON schema easy; the proposed primitives allow the NSDBA to express any schema change with the system propagating such a change to the underlying instances while updating all affected JSON documents and schema. It also guarantees the bookkeeping of a whole history of conventional/temporal JSON schemas and conventional/temporal JSON documents, which is a fundamental requirement for many JSON-based NoSQL database applications, since such a history allows the evaluation of temporal queries across schema versions, the recovering of past JSON schema and JSON document versions, and the tracking of schema and instance changes over time. More precisely, our approach could be useful for several real world applications where temporal versioning is required at both instance and schema levels, like online social networks, online reservations and banking transactions, application logs, clickstreams, online archiving, and electronic medical record evolution.

Currently, we are developing a prototype tool, named TJSchema-Manager, which implements our approach and shows its feasibility. It is being programmed in Java (JDK 1.7) as a stratum on top of MongoDB, a document-oriented NoSQL DBMS supporting both JSON and JSON Schema documents. Such a tool will serve as a testbed for experimental evaluation of the approach. Furthermore, we are also extending our current work by dealing with changes

to temporal logical and temporal physical characteristics; to this end, a complete set of change primitives is being defined.

It is worth mentioning that in this work we have dealt only with structural aspects of JSON, without aiming attention at the JSON data handling in NoSQL databases, as our present approach is orthogonal to this aspect. In fact, JSON is just a serialization technique and we focused on this property. Then, for our future work, we could have considered that JSON data can either be stored in native format in a NoSQL document store, like MongoDB, or converted into another format, e.g., to be stored even in a relational database (like in [33]). It is the first scenario that we have chosen for handling JSON instances as we are implementing our schema versioning approach on top of MongoDB.

Moreover, we notice that JSON schema versioning can be a way to manage also semistructured JSON data. Assume we receive new instances of a JSON file that do not fit into the current schema, e.g., because they have a new property P, then we can create a new schema version by adding the property P to the current schema (as optional, if we want both the old and the new data to fit into the new schema version and be merged into the same JSON file). In general, unstructured JSON data can be formatted according to a sort of completed schema [34] compatible with all the variants the data may assume. Following the example above, such schema can be incrementally constructed by applying schema changes when needed (i.e., to adapt the schema to an unforeseen format of new data).

Besides, as for scalability [1] [3] that is a fundamental feature of NoSQL databases which typically should scale horizontally (i.e., their performances increase when adding/connecting new hardware or software entities), future scalability tests, using our ω Schema-Manager prototype, will take into account the versioning as a further source of complexity with respect to a standard (without versioning) approach.

As a part of our future work, we will generalize our present approach, which could be qualified as a schema-enforced NoSQL database approach, by defining a schema-less approach for managing evolution and versioning of temporal JSON documents. After that, we will try to propose a generic temporal and multiversion NoSQL database approach that supports both the first and the second approaches.

Last but not least, since accessing and manipulating both conventional and temporal JSON instances, in an environment that supports temporal schema versioning, should be allowed to both software applications and human users, we also plan to study manipulation and querying of JSON instances in ω Schema, under schema versioning, by extending, for example, the JSONiq query language [35] to support temporal and multi-schema

queries and updates.

References

- [1] Cattell, R. (2010). Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record* 39(4) 2-27.
- [2] Tiwari, S. (2011). *Professional NoSQL*, John Wiley & Sons, Inc., Indianapolis, Indiana, USA.
- [3] Pokorný, J. (2013). NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems* 9 (1) 69-82.
- [4] Gudivada, V.N., Rao, D., Raghavan, V.V. (2014). NoSQL Systems for Big Data Management. In: *Proc. of the 2014 IEEE World Congress on Services (SERVICES'2014)*, Anchorage, AK, USA, 27 June – 2 July, p. 190-197.
- [5] Sharma, S., Tim, U.S., Gadia, S.K., et al. (2015). Classification and comparison of NoSQL big data models. *International Journal of Big Data Intelligence* 2 (3) 201-221.
- [6] Corbellini, A., Mateos, C., Zunino, A., et al. (2017). Persisting big-data: The NoSQL landscape. *Information Systems* 63 1-23.
- [7] Cuzzocrea, A. (2015). Temporal Aspects of Big Data Management: State-of-the-Art Analysis and Future Research Directions. In: *Proc. of the 22nd International Symposium on Temporal Representation and Reasoning (TIME'2015)*, Kassel, Germany, 23-25 September, p. 180-185.
- [8] Felber, P., Pasin, M., Riviere, E., et al. (2014). On the Support of Versioning in Distributed Key-Value Stores. In: *Proc. of the 33rd IEEE International Symposium on Reliable Distributed Systems (SRDS'2014)*, Nara, Japan, 6-9 October, p. 95-104.
- [9] Cerqueus, T., Cunha de Almeida, E., Scherzinger, S. (2015). ControVol: Let yesterday's data catch up with today's application code. In: *Proc. of the 24th International Conference on World Wide Web Companion (WWW'2015)*, Florence, Italy, Companion Volume, 18-22 May, p. 15-16.
- [10] Cerqueus, T., Cunha de Almeida, E., Scherzinger, S. (2015). Safely Managing Data Variety in Big Data Software Development. In: *Proc. of the 1st IEEE/ACM International Workshop on Big Data Software Engineering (BIGDSE'2015)*, Florence, Italy, 23 May, p. 4-10.
- [11] Haubold, F., Schildgren, J., Scherzinger, S., et al. (2017). ControVol Flex: Flexible Schema Evolution for NoSQL Application Development. In: *Proc. of the 17th Conference on Database Systems for Business, Technology, and Web (BTW'2017)*, University of Stuttgart, Stuttgart, Germany, 6-10 March, p. 601-604.
- [12] Klettke, M., Störl, U., Shenavai, M., et al. (2016). NoSQL Schema Evolution and Big Data Migration at Scale. In: *Proc. of the 2016 IEEE International Conference on Big Data (BigData'2016)*, Washington DC, USA, 5-8 December, p. 2764-2774.

- [13] Saur, K., Dumitras, T., Hicks, M.W (2016). Evolving NoSQL Databases Without Downtime. *In: Proc. of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME'2016)*, Raleigh, North Carolina, USA, 2-7 October, p. 166-176.
- [14] Scherzinger, S., Klettke, M., Störl, U (2013). Managing Schema Evolution in NoSQL Data Stores. *In: Proc. of the 14th International Symposium on Database Programming Languages (DBPL'2013)*, Riva del Garda, Trento, Italy, 30 August.
- [15] Scherzinger, S., Klettke, M., Störl, U (2015a). Cleager: Eager Schema Evolution in NoSQL Document Stores. *In: Proc. of the 16th Conference on Database Systems for Business, Technology, and Web (BTW'2015)*, University of Hamburg, Hamburg, Germany, 2-6 March, p. 659-662.
- [16] Scherzinger, S., Cerqueusy, T., Cunha de Almeida, E. (2015). ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development. *In: Proc. of the 31st IEEE International Conference on Data Engineering (ICDE'2015)*, Seoul, South Korea, 13-17 April, p. 1464-1467.
- [17] Scherzinger, S., Sombach, S., Wiech, K., *et al.* (2016). Datalution: a tool for continuous schema evolution in NoSQL-backed web applications. *In: Proc. of the 2nd International Workshop on Quality-Aware DevOps (QUDOS@ISSTA'2016)*, Saarbrücken, Germany, 21 July, p. 38-39.
- [18] Störl, U., Müller, D., Klettke, M., *et al.* (2017). Enabling Efficient Agile Software Development of NoSQL-backed Applications. *In: Proc. of the 17th Conference on Database Systems for Business, Technology, and Web (BTW'2017)*, University of Stuttgart, Stuttgart, Germany, 6-10 March, p. 611-614.
- [19] Roddick, J.F. (2009). Schema Versioning, *In: L. Liu, M. T. Özsu (Eds.), Encyclopedia of Database Systems*, Springer-Verlag, Berlin, Germany, p. 2499-2502.
- [20] Brahmia, Z., Grandi, F., Oliboni, B., *et al.* (2015). Schema Versioning, *In: M. Khosrow-Pour (Ed.), Encyclopedia of Information Science and Technology (3rd edition)*, IGI Global, Hershey, Pennsylvania, USA, pages 7651-7661.
- [21] Brahmia, S., Brahmia, Z., Grandi, F., *et al.* (2016). Schema: A Framework for Managing Temporal JSON-Based NoSQL Databases. *In: Proc. of the 27th International Conference on Database and Expert Systems Applications (DEXA'2016)*, Porto, Portugal, Part 2, 5-8 September, p. 167-181.
- [22] IETF (Internet Engineering Task Force) (2013). JSON Schema: core definition and terminology, Internet-Draft, 4 August 2013, available at: <<https://tools.ietf.org/pdf/draft-zyp-json-schema-04.pdf>> (accessed: May 05, 2017)
- [23] Currim, F., Currim, S., Dyreson, C.E., *et al.* (2004). A Tale of Two Schemas: Creating a Temporal XML Schema from a Snapshot Schema with δ XSchema. *In: Proc. of the 9th International Conference on Extending Database Technology (EDBT'2004)*, Heraklion, Crete, Greece, 14-18 March, pages 348-365.
- [24] IETF (Internet Engineering Task Force) (2014). The JavaScript Object Notation (JSON) Data Interchange Format, Internet Standards Track document, March 2014, available at: <<https://tools.ietf.org/html/rfc7159>> (accessed: May 05, 2017)
- [25] JSON Schema. (2017). Software, available at: <<http://json-schema.org/implementations>> (accessed: May 05, 2017)
- [26] Pezoa, F., Reutter, J.L., Suarez, F., *et al.* (2016). Foundations of JSON Schema. *In: Proc. of the 25th International World Wide Web Conference (WWW'2016)*, Montréal, Québec, Canada, 11-15 April, pages 263-273.
- [27] IETF (Internet Engineering Task Force) (2012). JSON Reference, Internet-Draft, 16 September 2012, available at: <<https://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03>> (accessed: May 05, 2017)
- [28] Gössner, S (2007). JSONPath – Xpath for JSON, 21 February, available at: <<http://goessner.net/articles/JsonPath/>> (accessed: May 05, 2017)
- [29] Liu, Z., He, B., Hsiao, H.-I., *et al.* (2011). Efficient and scalable data evolution with column oriented databases. *In: Proc. of the 14th International Conference on Extending Database Technology (EDBT'2011)*, Uppsala, Sweden, 21-24 March, p. 105-116.
- [30] Curino, C., Moon, H.J., Zaniolo, C (2008). Graceful database schema evolution: the PRISM workbench. *Proceedings of the VLDB Endowment (PVLDB 2008) 1(1)* 761-772.
- [31] Störl, U., Hauf, T., Klettke, M., *et al.* (2015). Schemaless NoSQL Data Stores - Object-NoSQL Mappers to the Rescue?. *In: Proc. of the 16th Conference on Database Systems for Business, Technology, and Web (BTW'2015)*, University of Hamburg, Hamburg, Germany, 2-6 March, p. 579-599.
- [32] Castelltort, A., Laurent, A. (2013). Representing history in graph-oriented NoSQL databases: A versioning system. *In: Proc. of the 8th International Conference on Digital Information Management (ICDIM'2013)*, Islamabad, Pakistan, 10-12 September, p. 228-234.
- [33] Chasseur, C., Li, Y., Patel, J.M. (2013). Enabling JSON Document Stores in Relational Systems. *In: Proc. of the 16th International Workshop on the Web and Databases (WebDB 2013)*, New York, NY, USA, 23 June, p. 1-6.
- [34] Roddick, J.F. (1991). Dynamically changing schemas within database models. *Australian Computer Journal* 23 (3) 105-109.
- [35] Florescu, D., Fourny, G. (2013). JSONiq: The History of a Query Language. *IEEE Internet Computing* 17(5) 86-90.