

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the TJSchema Framework

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the TJSchema Framework / Brahmia, Zouhaier; Brahmia, Safa; Grandi, Fabio; Bouaziz, Rafik. - In: INTERNATIONAL JOURNAL OF CLOUD COMPUTING. - ISSN 2043-9989. - STAMPA. - 10:5-6(2021), pp. 442-479. [10.1504/IJCC.2021.120386]

Availability:

This version is available at: <https://hdl.handle.net/11585/859449> since: 2023-12-07

Published:

DOI: <http://doi.org/10.1504/IJCC.2021.120386>

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>).
When citing, please refer to the published version.

(Article begins on next page)

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework

Zouhaier Brahmia*, Safa Brahmia

Department of Computer Science, Faculty of Economics and Management, University of Sfax,
Road of the Aerodrome, Km 4.5, P.O.Box 1088, 3018 Sfax, Tunisia
E-mail: zouhaier.brahmia@fsegs.rnu.tn
E-mail: safa.brahmia@gmail.com
*Corresponding author

Fabio Grandi

DISI - Department of Computer Science and Engineering, University of Bologna
Viale Risorgimento, 2, I-40136 Bologna BO, Italy
E-mail: fabio.grandi@unibo.it

Rafik Bouaziz

Department of Computer Science, Faculty of Economics and Management, University of Sfax,
Road of the Aerodrome, Km 4.5, P.O.Box 1088, 3018 Sfax, Tunisia
E-mail: rafik.bouaziz@usf.tn

Abstract: τ JSchema is a framework for managing time-varying JSON-based Big Data, in temporal JSON NoSQL databases, through the use of a temporal JSON schema. This latter ties together a conventional JSON schema and its corresponding temporal logical and temporal physical characteristics set. In our previous work, we have proposed low-level operations for changing the components of a τ JSchema schema, which are not very friendly for users (database administrators) as they are too primitive. In this paper, we propose three sets of high-level operations for changing the temporal JSON schema, the conventional JSON schema, and the temporal characteristics. These high-level operations are based on our previously proposed low-level operations. They are also consistency-preserving and more user-friendly than the low-level ones. To improve the readability of their definitions, we have divided these new operations into two classes: basic high-level operations, which cannot be defined through other basic high-level operations, and complex ones.

Keywords: Big Data, NoSQL, JSON, JSON Schema, τ JSchema, Conventional JSON schema, Temporal JSON schema, Temporal logical characteristic, Temporal physical characteristic, Schema change operation, Schema versioning, temporal databases

Reference to this paper should be made as follows: Brahmia, Z., Brahmia, S., Grandi, F. and Bouaziz, R. (xxxx) 'Versioning Schemas of JSON-based

Z. Brahmia et al.

Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework', *Int. J. Cloud Computing*, Vol. , No. , pp. .

Biographical notes: Zouhaier Brahmia is currently an Associate Professor of Computer Science in the Department of Computer Science at the Faculty of Economics and Management of the University of Sfax, Tunisia. He is a member of the Multimedia, InfoRmation systems, and Advanced Computing Laboratory (MIRACL). His scientific interests include temporal databases, schema versioning, and temporal, evolution and versioning aspects in emerging databases (XML, and NoSQL), Big Data, and Semantic Web ontologies. He received an MSc degree in Computer Science, in July 2005, and a PhD in Computer Science, in December 2011, from the Faculty of Economics and Management of the University of Sfax.

Safa Brahmia is currently a PhD student in Computer Science at the Faculty of Economics and Management of the University of Sfax, Tunisia. She is working on temporal and multi-schema-version JSON-based NoSQL databases. She is a member of the Multimedia, InfoRmation systems, and Advanced Computing Laboratory (MIRACL) since 2015. She received her MSc degree in Computer Science, in December 2014, from the Faculty of Economics and Management of the University of Sfax.

Fabio Grandi has been an Associate Professor at the University of Bologna, Italy, since 1998, currently in the Department of Computer Science and Engineering. From 1989 to 2012 he has worked at the CSITE center of the Italian National Research Council (CNR) in Bologna in the field of neural networks and temporal databases, initially supported by a CNR fellowship. In 1993 and 1994 he was an Adjunct Professor at the Universities of Ferrara, Italy, and Bologna. He was appointed as Research Associate at the University of Bologna in 1994. His scientific interests include temporal, evolution and versioning aspects in data management, WWW and Semantic Web, knowledge representation, storage structures and access cost models. He received a Laurea degree cum Laude in Electronics Engineering and a PhD in Electronics Engineering and Computer Science from the University of Bologna.

Rafik Bouaziz is currently a Full Professor of Computer Science at the Faculty of Economics and Management of the University of Sfax, Tunisia. He was the president of the University of Sfax between 2014 and 2017, and the director of the "Economy, Management, and Computer Science" doctoral school, in the same university, between 2011 and 2014. In his PhD thesis (defended in 1988), he has dealt with temporal data management and historical record of data in Information Systems. The subject of his habilitation (obtained in 2007) was "A contribution for the management of data and schema versioning in advanced information systems". His current research interests are temporal databases, real-time databases, information systems engineering, ontologies, data warehousing and workflows.

1 Introduction

Nowadays, modern applications that are running on the Cloud (e.g., Smart Cities, Internet of Things, online social networks, Secure Mobile Cloud Computing, e-CRMs, and ERPs) are storing, manipulating and exchanging Big Data (Al_Janabi and Hussein, 2020; Chen and Zhang, 2014; IRMA, 2016; Khosla and Kaur, 2018). Obviously, both schemas and

values/instances of these Big Data evolve over time, in general in a frequent manner, as a consequence of changes in the modeled reality and in the application requirements. Furthermore, a lot of these applications are requiring an efficient management of the full history of all updates that have been done on Big Data (Cuzzocrea, 2015), at both schema and instance levels. In this way, advanced requirements involving time-varying Big Data, like tracking changes over time, recovering past Big Data versions, supporting time-slice queries (Snodgrass *et al.*, 1995), can be satisfied.

In the database community, schema versioning (Brahmia *et al.*, 2015; Roddick, 2018) is considered the best solution for providing a complete history of both data and schema changes, by producing a new version of the schema each time such a schema is changed, while keeping old schema versions with their underlying data untouched. However, NoSQL database systems¹ (Cattell, 2010; Tiwari, 2011; Pokorný, 2013; Sharma *et al.*, 2014; Gudivada *et al.*, 2014; Sharma *et al.*, 2015; Corbellini *et al.*, 2017; Davoudian *et al.*, 2018), which are built to be the most efficient tools for managing Big Data, do not provide any support for handling temporal and versioning aspects of Big Data. Thus, NoSQL database administrators (NSDBAs) and Big Data application developers are proceeding in an ad hoc manner when they have to model, manipulate or query temporal and multi-schema-versioned Big Data.

We are convinced that, if we want an efficient and systematic management of time-varying Big Data in the presence of multiple schema versions, we need a temporal NoSQL database system that supports temporal schema versioning. Moreover, a data format commonly used for Big Data management is the JSON standard (IETF, 2014), which encompasses either in-memory representation, network exchange and disk storage requirements (Wang, 2017). For these reasons, we have proposed, in a previous work (Brahmia *et al.*, 2016), a disciplined approach, named τ JSchema (Temporal JSON Schema), for specifying and managing temporal JSON-based Big Data in NoSQL databases. It consists of a data model and a suite of tools that allow a NSDBA to create and validate temporal JSON documents, which store time-varying Big Data, through the use of a temporal JSON schema that defines the structure of such temporal Big Data and to which the temporal JSON documents conform. This temporal schema ties together a conventional JSON schema, that is a standard JSON Schema document (IETF, 2013a; Pezoa *et al.*, 2016), and a temporal characteristic document, that is a standard JSON document that contains temporal logical and temporal physical characteristics corresponding to components of the conventional schema. Temporal logical characteristics identify whether a component (e.g., property, object) of the conventional JSON schema is timestamped with valid and/or transaction time, whether its lifetime is described as a continuous state or a single event, whether the item itself may appear at certain times (and not at others), and whether its content changes. We recall that valid time is the period when some fact is true in the real world, whereas transaction time is the period when some fact is stored in the database. Temporal physical characteristics specify the concrete representation options chosen by the NSDBA to implement the timestamps.

1.1 Problems

¹ <http://www.nosql-database.org/>

Each component of a τ JSchema schema (conventional JSON schema, temporal logical characteristics and temporal physical characteristics) can individually evolve over time to respond to new application requirements received from all stakeholders (e.g., application developers, decision makers or final users). In a τ JSchema environment that supports schema versioning (Brahmia et al., 2015, 2018a; Roddick, 2018), changing a conventional JSON schema leads to a new version of it and automatically updates the corresponding temporal JSON schema in order to take into account the new conventional schema version. Similarly, changing temporal logical and/or temporal physical characteristics gives rise to a new version of the whole temporal characteristic document and also to the update of the temporal JSON schema so that the new temporal characteristic document is taken into account.

Notice that, whereas instance data can be versioned along either valid and/or transaction time (according to the application requirements), schemas are implicitly versioned along transaction time: when the NSDBA applies a schema change, the system records the time the change is applied which is stored as end time of the modified version and as begin time of the newly created version.

In our previous work (Brahmia et al., 2017) and (Brahmia et al., 2018b, 2019a), we have dealt with schema versioning in the τ JSchema framework. In fact, in (Brahmia et al., 2017), we have shown how a conventional JSON schema could be versioned and have proposed two complete and sound sets of low-level operations: one for changing conventional JSON schemas and the other for updating temporal JSON schemas. In (Brahmia et al., 2018b, 2019a), we have enhanced the framework by incorporating the versioning of temporal logical and physical characteristics: we have shown how such characteristics can be versioned and proposed in this direction a complete and sound set of change primitives for their manipulation. In each of these works, we have also illustrated the use of the introduced low-level operations through an application example.

However, since these schema change operations are low-level and not very user-friendly, they are not designated to be directly used by τ JSchema NSDBAs. A production τ JSchema-based system should rather allow NSDBAs to change schemas, for example, through an easy-to-use tool which provides friendly high-level schema change operations. Notice that a high-level schema change operation could be defined as a valid sequence of low-level schema change operations. Moreover, it corresponds to the fulfillment of a frequently occurring schema evolution requirement and allows expressing a complex change to the schema in a more compact way than the equivalent sequence of primitives (Guerrini et al., 2005).

1.2 Contributions

Based on our previous definition of primitives, in this work, we aim at introducing high-level and more user-friendly schema change operations. Therefore, in order to facilitate the NSDBA's task of managing schema versioning in τ JSchema and, thus, to make our approach even more useful, we propose in this paper three sets of high-level schema operations for consistently defining and changing conventional JSON schemas, temporal logical and physical characteristics, and temporal JSON schemas.

τ JSchema NSDBAs could use these high-level operations to create a τ JSchema schema or to perform any change on it, by composing them into valid sequences and executing them on the desired schema component (i.e., the temporal JSON schema, the

conventional JSON schema, or the temporal characteristics), within the same schema change transaction. Notice that, similarly to the traditional transaction notion, we define in our work a schema change transaction as a sequence of valid (low-level or high-level) schema change operations, which would be executed on a τ JSchema schema component and which would be either all successfully completed or all cancelled.

The proposed high-level operations are based on the primitives previously provided in (Brahmia et al., 2017, 2018b, 2019a). Since each primitive is consistency preserving (i.e., each primitive applied to a consistent τ JSchema schema component generates a consistent τ JSchema schema component) and high-level operation will be defined as a sequence of such primitives, our proposed high-level operations will also result consistency preserving. Besides, the introduced high-level operations are also bound to result more user-friendly, since they consider complex schema component structures (e.g., sub-schemas) that have often to be changed together rather than simple schema components (e.g., a single property). In addition, in order to improve the readability of their definitions, we have divided them into two classes: (i) basic high-level operations, which cannot be defined through other basic high-level operations, and (ii) complex high-level operations, which are defined through the composition of other basic and/or complex high-level operations.

Notice that, with respect to our previous works (Brahmia et al., 2017, 2018b, 2019a), the present paper deals with high-level operations that could be exploited by NSDBAs for creating and changing entire schemas in τ JSchema-based NoSQL data stores.

1.3 Organization

The remainder of this paper is organized as follows. Section 2 briefly describes our τ JSchema framework, and presents our approach for versioning of conventional JSON schemas, temporal (logical and physical) characteristics, and temporal JSON schemas. Section 3 proposes the set of high-level operations for changing the temporal JSON schema in τ JSchema. Section 4 introduces the set of high-level operations for changing conventional JSON schemas. Section 5 provides the set of high-level operations for changing temporal logical and temporal physical characteristics. Section 6 illustrates the use of the proposed operations through an application example. Section 7 discusses the contribution of our work with respect to the related ones. Section 8 summarizes the paper and gives some remarks about our future work.

2 Versioning of Conventional JSON Schemas, Temporal Logical and Physical Characteristics, and Temporal JSON Schemas

In this section, first we briefly describe the organization of the τ JSchema framework, then we explain how schema evolution and versioning can be supported in our approach, and finally we present the design choices on which the definition of our high-level schema change operations is based.

2.1 The τ JSchema Framework

In this subsection, we provide a succinct description of our τ JSchema framework (Brahmia et al., 2016) for handling temporal JSON documents. Its definition was inspired by the well-known τ XSchema framework proposed for XML documents in (Currim et al., 2004; Snodgrass et al., 2008) and further developed in (Brahmia et al., 2014, 2018a), with which it shares the same organization. A more detailed and comprehensive description can be found in (Brahmia et al., 2019a: Sec. 2).

In order to create a temporal JSON schema for the management of temporal JSON data instances, a NSDBA must supply three specifications: a conventional (i.e., non-temporal) JSON schema, temporal logical characteristics and temporal physical characteristics. Although the two sets of temporal characteristics are orthogonal and can evolve independently, they are stored together in a single JSON document associated to the conventional schema, which is a standard JSON document named the temporal characteristics document. From these specifications, the system automatically creates the temporal JSON schema, which is a standard JSON document, providing the linkage between the conventional schema and its corresponding logical and physical characteristics. In the τ JSchema framework, the temporal JSON schema is the logical equivalent of the conventional JSON schema in a non-temporal environment. A validator tool (named Temporal JSON Schema Validator) can also be used to report whether a temporal JSON schema document is valid or invalid.

At the instance level, the system creates a temporal JSON document, which is a standard JSON document that represents the complete evolution of a non-temporal JSON document over time, by linking all the individual versions of the document (conformant to a conventional schema) to their corresponding timestamps and by specifying the temporal schema associated to these versions. The temporal JSON document facilitates the support of temporal queries retrieving specific JSON document versions (i.e., time-slice queries) or dealing with changes between JSON document versions.

A validated temporal JSON schema can also be used by the JSON Schema Mapper tool to create the so-called representational JSON schema, which allows the individual versions of the JSON document to be stored together in a single temporal document, called temporal JSON data instances, which is a JSON document containing timestamped data and conformant to the representational JSON schema. Such assembly of non-temporal JSON instances connected to a temporal JSON document into a temporal JSON instances document can be performed using the JSON Instances Squasher tool. A validator tool (named Temporal JSON Instances Validator) is also available to report whether temporal JSON instances are valid or not with respect to the representational JSON schema.

In our approach, the four mentioned tools are currently under development. For example, the temporal JSON instances validator tool is being implemented as a temporal extension of an existing conventional JSON instances validator (IETF, 2013b).

2.2 Schema versioning

The first step of a versioned schema management process is the creation of the first temporal JSON schema version: the NSDBA creates a conventional JSON Schema document (i.e., a classical JSON Schema file) annotated with some logical and physical characteristics in an independent document (which is also stored as a JSON file). Consequently, the system generates the temporal JSON schema (also stored as a JSON

file) that ties together the conventional schema and the temporal characteristics. In further steps of the versioning process, when necessary, the NSDBA can independently change the conventional schema, the temporal logical characteristics or the temporal physical characteristics.

Executing a transaction that applies changes to the conventional schema leads to a new version of it. Similarly, applying changes to the temporal logical and/or temporal physical characteristics leads to a new version of the whole temporal characteristics document. Hence, the temporal JSON schema is automatically updated after each transaction involving change to the conventional JSON schema or to the temporal characteristics document, in order to take into account the new version of the corresponding changed component.

Notice that, in contrast to the conventional JSON schema and the temporal characteristics document, the temporal schema is not “explicitly” versioned: for each conventional schema (i.e., all the versions of this schema) and its associated temporal characteristic document (i.e., all the versions of this document), there is always one JSON document that represents the temporal schema, which is updated when the conventional JSON schema and/or the temporal characteristic document are changed. In fact, in the τ JSchema framework, the temporal JSON schema is instrumental to support versioning of anything can change in the managed JSON document repository. As a consequence, by its nature, the temporal JSON schema is “implicitly” versioned (i.e., all versions of a temporal JSON schema document are timestamped and assembled together within the same container document (and a specific version of the temporal JSON schema, valid at any given time, can be extracted from that schema by executing a simple timeslice query). For this reason, other kinds of versioning operations involving the temporal schema are neither necessary nor could be explicitly put at NSDBA’s disposal in a meaningful way (i.e., without allowing the NSDBA to potentially harm the internal coherence of the τ JSchema framework).

Schema change operations performed by the NSDBA are assumed to be high-level, since they are usually conceived having in mind to target high-level real-world object properties. Each of these high-level schema change operations is then mapped onto a sequence of low-level schema change operations (or schema change primitives). The mapping has to be performed by the schema change processor (currently under development) and allows the high-level operations to be implemented and made available in the administration user-interface.

2.3 Design choices

The definition of the high-level operations obeys the following design choices:

- All operations must have a valid conventional JSON schema (CJS) (temporal characteristics document (TCD) or temporal JSON schema (TJS), respectively) as input and must produce a valid CJS (TCD or TJS, respectively) as output.
- All operations need to work on a JSON Schema (JSON document, respectively) file storing the CJS (TCD or TJS, respectively), whose name must be supplied as first argument.
- For all operations, arguments which are used to identify the object on which the operation works are always in the first place of the argument list.

- Components which are just containers for other components can be implicitly managed by the operations concerning the components, without specific operations 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).
- Operations adding objects with possibly optional properties have the values for all the properties as arguments; empty places in the argument list stand for unspecified optional properties.

Taking into account the design choices presented above, we have defined a set of fifty-three (53) high-level operations. For each operation, we describe in the following its arguments and its operational semantics. Moreover, due to space limitations, we do not present in this paper the effects of all proposed operations but only of some selected ones.

3 High-level Operations for Changing Temporal JSON Schemas

In (Brahmia et al., 2017), we defined four low-level change operations that act on temporal JSON schemas: `CreateTemporalJSONSchema`, `RemoveTemporalJSONSchema`, `AddSliceToTemporalJSONSchema`, and `RemoveSliceFromTemporalJSONSchema`. Based on these primitives, we proposed four high-level operations for changing temporal JSON schemas in the τ JSchema framework: `DefineTemporalJSONSchema`, `UpdateTemporalJSONSchema`, `DropTemporalJSONSchema`, and `RenameTemporalJSONSchema`. They are presented below. With respect to their low-level counterparts, such operations hide the details connected to the management of slices.

```
DefineTemporalJSONSchema(TJS.json, sourceFirstVersionCJS,  
    targetFirstVersionCJS, sourceFirstVersionTCD,  
    targetFirstVersionTCD)
```

It creates a new temporal JSON schema document (TJS.json) that includes a first version of a conventional JSON schema (located at targetFirstVersionCJS and its content is obtained from the sourceFirstVersionCJS) and a first version of the corresponding temporal characteristic document (located at targetFirstVersionTCD and its content is obtained from the sourceFirstVersionTCD).

Here, the sourceFirstVersionCJS (sourceFirstVersionTCD, respectively) argument could be:

1) The keyword empty; in this case the resource pointed by targetFirstVersionCJS (targetFirstVersionTCD, respectively) is initialized to an empty conventional JSON schema (temporal characteristic document, respectively).

2) The keyword current; in this case the resource pointed by targetFirstVersionCJS (targetFirstVersionTCD, respectively) is initialized with a copy of the current conventional JSON schema (temporal characteristic document, respectively) resource, whose location is found in the TJS.json temporal JSON schema file by choosing the “slice” property with the maximum value of begin in the “sliceSequence” property of the “conventionalJSONSchema” (“temporalCharacteristicSet”, respectively) container. Notice that this is the normal case after the creation of the first temporal JSON schema version (obviously with a first conventional JSON schema version and a first temporal characteristic document version).

3) A specified file name (URL): in this case, a copy of the specified resource is renamed as `targetFirstVersionCJS` (`targetFirstVersionTCD`, respectively) and used as the new location (e.g., this case is used to create a new conventional JSON schema (temporal characteristic document, respectively) version from an already existing JSON Schema (JSON document, respectively) file, which could be quite common when creating the first version but can be used also later for reuse purpose and/or integrating independently developed JSON schemata (JSON documents, respectively) into a τ JSchema framework).

As for the `targetFirstVersionCJS` (`targetFirstVersionTCD`, respectively) argument, it corresponds to the location of the new conventional JSON schema (temporal characteristic document, respectively) version; it must not correspond to the URL of any already existing JSON Schema (JSON document, respectively) file/resource.

The semantics of this operation is defined by mapping it onto a sequence of primitives that have been already proposed in our previous work (Brahmia et al., 2017), as shown in Algorithm 1.

```
Begin
CreateTemporalJSONSchema (TJS.json)
AddSliceToTemporalJSONSchema (TJS.json, conventionalJSONSchema,
    sourceFirstVersionCJS, targetFirstVersionCJS)
AddSliceToTemporalJSONSchema (TJS.json, temporalCharacteristicSet,
    sourceFirstVersionTCD, targetFirstVersionTCD)
End
```

Algorithm 1 Semantics of the **DefineTemporalJSONSchema** operation

Example: Suppose that the NSDBA wants to define a new temporal JSON schema for an inventory, based on a JSON Schema file “Inventory.json” and a JSON document “InventoryTemporalCharacteristics.json” that includes temporal logical and temporal physical characteristics associated to “Inventory.json”. To do this, he/she calls the **DefineTemporalJSONSchema** high-level operation as follows:

```
DefineTemporalJSONSchema ("InventoryTemporalJSONSchema.json",
    "Inventory.json", "Inventory_V1.json",
    "InventoryTemporalCharacteristics.json",
    "InventoryTemporalCharacteristics_V1.json")
```

```
UpdateTemporalJSONSchema (TJS.json, sourceNewVersionCJS,
    targetNewVersionCJS, sourceNewVersionTCD,
    targetNewVersionTCD)
```

It updates a temporal JSON schema by including a new conventional JSON schema version (i.e., `sourceNewVersionCJS`), or a new temporal characteristic document version, (i.e., `sourceNewVersionTCD`). Notice that only one of these two arguments can be omitted.

The semantics of this operation is defined by mapping it onto a sequence of low-level operations that have been already proposed in our previous work (Brahmia et al., 2017), as shown in Algorithm 2.

```
Begin
  If (sourceNewVersionCJS is not null) Then
    AddSliceToTemporalJSONSchema (TJS.json,
      conventionalJSONSchema, sourceNewVersionCJS,
      targetNewVersionCJS)
  End If
  If (sourceNewVersionTCD is not null) Then
    AddSliceToTemporalJSONSchema (TJS.json,
      temporalCharacteristicSet, sourceNewVersionTCD,
      targetNewVersionTCD)
  End If
End
```

Algorithm 2 Semantics of the **UpdateTemporalJSONSchema** operation

DropTemporalJSONSchema (TJS.json)

It allows the NSDBA to drop a temporal JSON schema, if necessary.

The semantics of this operation is defined by mapping it onto a sequence of low-level operations that have been already proposed in our previous work (Brahmia et al., 2017), as shown in Algorithm 3.

```
Begin
  For each sourceLocation := "slice" property in
    "conventionalJSONSchema" container of "TJS.json" do:
    RemoveSliceFromTemporalJSONSchema (TJS.json,
      conventionalJSONSchema, sourceLocation)
  For each sourceLocation := "slice" property in
    "temporalCharacteristicSet" container of "TJS.json"
    do:
    RemoveSliceFromTemporalJSONSchema (TJS.json,
      temporalCharacteristicSet, sourceLocation)
  RemoveTemporalJSONSchema (TJS.json)
End
```

Algorithm 3 Semantics of the **DropTemporalJSONSchema** operation

RenameTemporalJSONSchema (TJS.json, newName)

It changes the name of an existing temporal JSON schema ("TJS.json") to "newName".

4 High-level Operations for Changing Conventional JSON Schemas

We have defined twenty-three high-level operations: thirteen basic high-level operations, acting on JSON schema properties and keywords, and ten complex high-level operations, dealing with entire conventional JSON schemas and portions of conventional JSON schemas (or subschemas).

Due to space limitations, we do not present in this paper the total list of these operations, which can be found in the online appendix (Brahmia et al., 2019b: Sec. A1).

4.1 Basic High-level Operations

4.1.1 Basic High-level Operations Dealing With Properties

We have defined eight basic high-level operations acting on properties: `MoveProperty`, `CopyProperty`, `RenameProperty`, `ReplacePropertyWithNewProperty`, `ExchangeProperties`, `SplitPropertyIntoProperties`, `AddProperty`, and `DropProperty`.

Due to space limitation, we present here only the `SplitProperty` operation. The full list of basic high-level operations acting on properties can be found in the online appendix (Brahmia et al., 2019b: Sec. A1.1.1).

`SplitPropertyIntoProperties`(CJS.json, PropertyPath)

It splits a non-empty object property (located at PropertyPath) that contains only simple subproperties (i.e., having string, number, boolean or null type), into a sequence of its subproperties.

Moreover, this operation must update:

- all other components of this conventional JSON schema that are using (or referring to) the split property (e.g., “required” and “dependencies” components);
- all “stamps” components, in the current version of the temporal characteristic document corresponding to this conventional JSON schema, that are referring to the split property.

Example: Suppose that the NSDBA wants to split the property “authorContact” which contains five simple sub-properties (“address”, “phone”, “cell”, “fax”, and “email”) into five new properties (as shown below). He/She calls the `SplitPropertyIntoProperties` operation as follows:

`SplitPropertyIntoProperties`(CJS.json, "\$..authorContact")

Before applying the <code>SplitPropertyIntoProperties</code> operation	After applying the <code>SplitPropertyIntoProperties</code> operation
<pre> ... "authorContact":{ "type": "object", "properties": { "address":{"type":"string"}, "phone":{"type":"string"}, "cell":{"type":"string"}, "fax":{"type":"string"}, "email":{"type":"string"} } } ... </pre>	<pre> ... "address":{"type":"string"}, "phone":{"type":"string"}, "cell":{"type":"string"}, "fax":{"type":"string"}, "email":{"type":"string"} ... </pre>

4.1.2 Basic High-level Operations Dealing With Keywords

We have defined five basic high-level operations acting on keywords: `AddKeywordToConventionalJSONSchema`, `MoveKeyword`, `CopyKeyword`, `ReplaceKeywordWithNewKeyword`, and `RemoveKeyword`.

Due to space limitation, we present here only the `AddKeywordToConventionalJSONSchema` operation. The full list of basic high-level operations acting on keywords can be found in the online appendix (Brahmia et al., 2019b: Sec. A1.1.2).

AddKeywordToConventionalJSONSchema (`CJS.json`, `keywordContainerPath`, `keywordName`, `keywordType`, `keywordValue`)

It adds a new keyword having a `keywordType` (i.e., string, number, boolean, null, object or array), with its `keywordName` and `keywordValue` components, to a container located at `keywordContainerPath`, in the conventional JSON schema “CJS.json”.

The semantics of this operation is defined by mapping it onto a sequence of low-level operations that have been already proposed in our previous work (Brahmia et al., 2017), as shown in Algorithm 4.

```
Begin
  If (keywordType is "string", "number", "boolean" or "null")
    Then
      AddSimpleTypeKeywordToConventionalJSONSchema (CJS.json,
        keywordContainerPath, keywordName, keywordValue)
    Else
      If (keywordType is "object") Then
        AddObjectTypeKeywordToConventionalJSONSchema (CJS.json,
          keywordContainerPath, keywordName)
      Else
        If (keywordType is "array") Then
          AddArrayTypeKeywordToConventionalJSONSchema (CJS.json,
            keywordContainerPath, keywordName)
        End If
      End
    End
  End
End
```

Algorithm 4 Semantics of the `AddKeywordToConventionalJSONSchema` operation

4.2 Complex High-level Operations

In this subsection, we present complex high-level schema change operations (i.e., high-level operations that are defined by using other basic and/or complex high-level operations). More precisely, we propose ten schema change operations: five operations acting on the whole conventional JSON schema, in the subsection 4.2.1, and five operations acting on portions of conventional JSON schemas (i.e., subschemas), in the subsection 4.2.2.

4.2.1 Complex High-level Operations Dealing With the Whole Conventional JSON Schema

In this subsection, we propose five complex high-level operations for changing the entire conventional JSON schema: `CreateConventionalJSONSchemaByExtraction`, `MergeConventionalJSONSchema`, `ReplaceConventionalJSONSchema`, `DropConventionalJSONSchema`, and `RenameConventionalJSONSchema`.

Due to space limitation, we present here only the `CreateConventionalJSONSchemaByExtraction` operation. The full list of complex high-level operations dealing with the whole conventional schema can be found in the online appendix (Brahmia et al., 2019b: Sec. A1.2.1).

CreateConventionalJSONSchemaByExtraction(sourceCJS, selectionBeginningPath, selectionEndPath, targetCJS, option)

It extracts some JSON Schema code (which starts at “selectionBeginningPath” and ends at “selectionEndPath”) from a source conventional JSON schema (“sourceCJS”) and saves it as a new target conventional JSON schema (“targetCJS”), with specified option (leave or delete). The option argument values are detailed as follows:

1) leave: the sourceCJS conventional JSON schema is left unchanged after the extraction (i.e. the operation simply saves a copy of the selected subschema into the new targetCJS);

2) delete: the selected subschema is deleted from the sourceCJS conventional JSON schema after the extraction (this would potentially require propagation of heavy modifications to the JSON data instances);

Furthermore, we should notice that the extracted subschema has to be completed with the required headers (which may include the outermost “\$schema” property) in order to be saved as a valid independent conventional JSON schema.

Example: Suppose that the NSDBA wants to create a conventional JSON schema for customers that are organizations, by physically extracting the JSON Schema code corresponding to such customers from an existing conventional JSON schema for customers (see Figure 1), with option “delete”. To do this, he/she calls the `CreateConventionalJSONSchemaByExtraction` operation as follows:

```
CreateConventionalJSONSchemaByExtraction(customers.json,
    "$.properties.customer-organizations",
    "$.properties.customer-organizations",
    customerOrganizations.json, delete)
```

After the execution of this operation, the desired conventional JSON schema will be created (see Figure 2) and the initial conventional JSON schema will become as shown in Figure 3.

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "customer-persons": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
```

```
    "name": {"type": "string"},
    "address": {"type": "string"},
    "phone": {"type": "number"},
    "turnover": {"type": "number"},
    "birthdate": {"type": "string"} },
    "required": ["name", "address", "phone", "turnover",
                 "birthdate"] } }},
  "customer-organizations": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
        "address": {"type": "string"},
        "phone": {"type": "number"},
        "turnover": {"type": "number"},
        "activityDomain": {"type": "string"} },
      "required": ["name", "address", "phone", "turnover",
                  "activityDomain"] } } },
  "required": ["customer-persons", "customer-organizations"] }
```

Figure 1 Conventional JSON schema for customers (customers_V1.json) before change.

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "customer-organizations": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {"type": "string"},
          "address": {"type": "string"},
          "phone": {"type": "number"},
          "turnover": {"type": "number"},
          "activityDomain": {"type": "string"} },
          "required": ["name", "address", "phone", "turnover",
                      "activityDomain"] } } },
    "required": ["customer-organizations"] }
```

Figure 2 Conventional JSON schema for customers which are organizations (customerOrganizations_V1.json).

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "customer-persons": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {"type": "string"},
          "address": {"type": "string"},
          "phone": {"type": "number"},
          "turnover": {"type": "number"},
          "activityDomain": {"type": "string"} },
          "required": ["name", "address", "phone", "turnover",
                      "activityDomain"] } } },
    "customer-organizations": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {"type": "string"},
          "address": {"type": "string"},
          "phone": {"type": "number"},
          "turnover": {"type": "number"},
          "activityDomain": {"type": "string"} },
          "required": ["name", "address", "phone", "turnover",
                      "activityDomain"] } } },
    "required": ["customer-persons", "customer-organizations"] }
```

```
"phone": {"type": "number"},
"turnover": {"type": "number"},
"birthdate": {"type": "string"} },
"required": ["name", "address", "phone", "turnover",
            "birthdate"] } } },
"required": ["customer-persons"] }
```

Figure 3 Conventional JSON schema for customers (customers_V2.json) after change.

4.2.2 Complex High-level Operations Dealing With Portions of Conventional JSON Schemas

In this subsection, we propose five complex high-level operations for changing portions of a conventional JSON schema: `InsertSubJSONSchema`, `RemoveSubJSONSchema`, `ReplaceSubJSONSchema`, `MoveSubJSONSchema`, and `CopySubJSONSchema`.

Due to space limitation, we present here only the `InsertSubJSONSchema` operation. The full list of complex high-level operations dealing with portions of conventional schemas can be found in the online appendix (Brahmia et al., 2019b: Sec. A1.2.2).

InsertSubJSONSchema(CJS.json, targetPropertyPath, position, subJSONSchema)

It inserts a new subschema (“subJSONSchema”) at a specified position (i.e., before or after) with regard to a target property (located at “targetPropertyPath”) in the conventional JSON schema “CJS.json”.

The new subschema to be inserted can be provided by the NSDBA either as a string explicitly containing the subschema text, or as a file name corresponding to a source conventional JSON schema to be used for insertion (after removal of headers).

Example: Suppose that the NSDBA wants to add a new subschema that describes foreign students at the end (i.e., after the property “students”) of the current conventional JSON schema which describes only local students “students_V1.json” (see Figure 4). To do this, he/she calls the `InsertSubJSONSchema` operation as follows:

```
InsertSubJSONSchema(students_V1.json, "$.properties.students",
after,
"foreign-students": {
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "name": {"type": "string"},
      "address": {"type": "string"},
      "phone": {"type": "number"},
      "passport": {"type": "string"},
      "country": {"type": "string"} },
    "required": ["name", "address", "phone",
                "passport", "country"] } } )
```



```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "students": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {"type": "string"},
          "address": {"type": "string"},
          "phone": {"type": "number"} },
        "required": ["name", "address", "phone"]
      } } },
  "required": ["students"] }
```

Figure 4 Conventional JSON schema for only local students (students_V1.json).

Figure 5 shows the new version “students_V2.json” of the updated conventional JSON schema.

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "students": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {"type": "string"},
          "address": {"type": "string"},
          "phone": {"type": "number"} },
        "required": ["name", "address", "phone"] } },
    "foreign-students": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {"type": "string"},
          "address": {"type": "string"},
          "phone": {"type": "number"},
          "passport": {"type": "string"},
          "country": {"type": "string"} },
        "required": ["name", "address", "phone", "passport",
                     "country"] } } },
  "required": ["students"] }
```

Figure 5 New conventional JSON schema for local and foreign students (students_V2.json).

5 High-level Operations for Changing Temporal Logical and Temporal Physical Characteristics

We have defined twenty-six high-level schema change operations which act on the temporal characteristic document. We organize them into three categories: (i) operations

acting on the whole temporal characteristic document, presented in the subsection 5.1, (ii) operations that are specific to the temporal logical characteristics, described in the subsection 5.2, and (iii) operations that are specific to the temporal physical characteristics, presented in the subsection 5.3.

5.1 High-level Operations Acting on the Whole Temporal Characteristic Document

We have defined six high-level change operations that act on the whole temporal characteristic document. These operations are as follows:

- SpecifyDefaultTimeFormatUsedInTemporalCharacteristicDocument,
- RemoveTemporalCharacteristicDocument,
- RenameTemporalCharacteristicDocument,
- ReplaceTemporalCharacteristicDocument,
- MergeTemporalCharacteristicDocuments, and
- SplitTemporalCharacteristicDocuments.

Due to space limitation, we present here only the SpecifyDefaultTimeFormatUsedInTemporalCharacteristicDocument operation. The full list of high-level operations dealing with the whole temporal characteristic document can be found in the online appendix (Brahmia et al., 2019b: Sec. A2.1).

```
SpecifyDefaultTimeFormatUsedInTemporalCharacteristicDocument(  
    TCD.json, temporalCharacteristicType, usedPlugin,  
    granularityOfTimeFormat, usedCalendricSystem,  
    dateFormatProperties, valueSchemaUsedForDate)
```

It specifies the default time format used in the temporal characteristic document “TCD.json” for temporal logical or for temporal physical characteristics (according to the temporalCharacteristicType argument that should have the value logical or physical). This default time format has the following properties: usedPlugin, granularityOfTimeFormat, usedCalendricSystem, dateFormatProperties, and valueSchemaUsedForDate.

The semantics of this operation is defined by mapping it onto a sequence of low-level operations that have been already proposed in our previous work (Brahmia et al., 2018b), as shown in Algorithm 5.

```
Begin  
  If (temporalCharacteristicType = logical) Then  
    AddDefaultTimeFormat(TCD.json, logical, usedPlugin,  
      granularityOfTimeFormat, usedCalendricSystem,  
      dateFormatProperties, valueSchemaUsedForDate)  
  Else  
    AddDefaultTimeFormat(TCD.json, physical, usedPlugin,  
      granularityOfTimeFormat, usedCalendricSystem,  
      dateFormatProperties, valueSchemaUsedForDate)  
  End If  
End
```

Algorithm 5 Semantics of the
SpecifyDefaultTimeFormatUsedInTemporalCharacteristicDocument
operation

5.2 High-level Operations Specific to the Temporal Logical Characteristics

We have defined ten high-level operations for changing temporal logical characteristics. We organized them into three categories: (i) operations that act on the whole temporal logical characteristic set, presented in the subsection 5.2.1, (ii) operations that act on a portion of a temporal logical characteristic set, described in the subsection 5.2.2, and (iii) operations that act on a time-varying logical item, presented in the subsection 5.2.3.

5.2.1 High-level Operations dealing with the whole Temporal Logical Characteristic Set

We have defined three high-level operations that act on the whole temporal logical characteristic set. They are as follows:

- InsertTemporalLogicalCharacteristicSet,
- RemoveTemporalLogicalCharacteristicSet, and
- ReplaceTemporalLogicalCharacteristicSet.

Due to space limitation, we present here only the InsertTemporalLogicalCharacteristicSet operation. The full list of high-level operations dealing with the whole temporal logical characteristic set can be found in the online appendix (Brahmia et al., 2019b: Sec. A2.2.1).

InsertTemporalLogicalCharacteristicSet(TCD.json,
temporalLogicalCharacteristicSet)

It adds, within a temporal characteristic document “TCD.json”, a set of temporal logical characteristics (“temporalLogicalCharacteristicSet”); this latter is either a string explicitly containing the text of the new temporal logical characteristic set, or a file name corresponding to a JSON document that stores the new temporal logical characteristic set. Besides, since the ordering of “logicalItems” items in the “logical” container is unimportant, this operation adds the new set of temporal logical characteristics (i.e., the new items of “logicalItems” array) at the end of the existing set of temporal logical characteristics, within the “logical” container. If this latter does not exist in “TCD.json”, this operation first creates it and then inserts the new set of temporal logical characteristics.

Example: Suppose that the NSDBA wants to add a set of temporal logical characteristics in the temporal characteristic document “EnterpriseTemporalCharacteristics.json”. To do this, he/she calls the InsertTemporalLogicalCharacteristicSet operation as follows:

```
InsertTemporalLogicalCharacteristicSet(  
  EnterpriseTemporalCharacteristics.json,  
  "{  
    \"logical\":  
      {  
        \"logicalItems\": [  
          {  
            \"target\": \"$.properties.enterprise\",
```

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework

```
"transactionTime": {},
"itemIdentifier": {
  "name": "enterpriseId1",
  "timeDimension": "transactionTime"
} },
{ "target": "$.properties.enterprise.properties.customers",
"validTime": {
  "kind": "state",
  "content": "varying",
  "existence": "varyingWithGaps"
},
"transactionTime": {},
"itemIdentifier": {
  "name": "customerId1",
  "timeDimension": "bitemporal",
  "field": {
    "path": "$.properties.enterprise.properties.
      customers.items.properties.customerNo" } } },
{ "target": "$.properties.enterprise.properties.equipments.
  items.properties.price",
"validTime": {
  "kind": "state",
  "content": "varying"
},
"transactionTime": {},
"itemIdentifier": {
  "name": "priceId1",
  "timeDimension": "bitemporal" } } ] } }")
```

5.2.2 High-level Operations dealing with a Subset of the Temporal Logical Characteristics

We have defined three high-level operations that act on a subset of the temporal logical characteristics. They are as follows:

- InsertTemporalLogicalCharacteristicSubSet,
- RemoveTemporalLogicalCharacteristicSubSet, and
- ReplaceTemporalLogicalCharacteristicSubSet.

Due to space limitation, we present here only the ReplaceTemporalLogicalCharacteristicSubSet operation. The full list of high-level operations dealing with the whole temporal logical characteristic set can be found in the online appendix (Brahmia et al., 2019b: Sec. A2.2.2).

```
ReplaceTemporalLogicalCharacteristicSubSet(TCD.json,
beginningIndexPath, endingIndexPath,
newTemporalLogicalCharacteristicSubSet)
```

It replaces, within a temporal characteristic document “TCD.json”, a subset of temporal

logical characteristics, delimited by a beginning item (located at “beginningIndexPath”, specifying the index of this item in the “logicalItems” array) and an ending item (located at “endingIndexPath”, specifying the index of this item in the “logicalItems” array), with a new subset of temporal logical characteristics (“newTemporalLogicalCharacteristicSubSet”); this latter is either a string explicitly containing the text of the new temporal logical characteristic subset, or a file name corresponding to a JSON document that stores the new temporal logical characteristic subset.

5.2.3 High-level Operations dealing with Time-varying Logical Items

Since a single temporal logical characteristic is described by a time-varying item (i.e., an item of the array “logicalItems” in the “logical” container), high-level operations for changing temporal logical characteristics must include operations acting on such a time-varying item. But the single item in the “logicalItems” array is a complex one, which includes four properties (Brahmia et al., 2018b; Sec. A2): “target”, “validTime”, “transactionTime” and “itemIdentifier”. Notice that each one of the last three properties has also sub-properties. Therefore, while taking into account all the information presented above, we propose the following list of four high-level operations for defining, removing and changing time-varying logical items:

- DefineTimeVaryingLogicalItem;
- RemoveTimeVaryingLogicalItem;
- ChangeTimeVaryingLogicalItem;
- ReplaceTimeVaryingLogicalItem.

Due to space limitation, we present here only the DefineTimeVaryingLogicalItem operation. The full list of high-level operations dealing with time-varying logical items can be found in the online appendix (Brahmia et al., 2019b; Sec. A2.2.3).

```
DefineTimeVaryingLogicalItem(TCD.json, logicalItemTarget,  
    validTimeKind, validTimeContent, validTimeExistence,  
    validTimeContentVaryingApplicabilityBegin,  
    validTimeContentVaryingApplicabilityEnd,  
    validTimeMaximalExistenceBegin, validTimeMaximalExistenceEnd,  
    validTimeFrequency, transactionTimeKind,  
    transactionTimeContent, transactionTimeExistence,  
    transactionTimeFrequency, itemIdentifierName,  
    itemIdentifierTimeDimension, itemIdentifierKeyRefName,  
    itemIdentifierKeyRefType, itemIdentifierPathField)
```

It defines, in a temporal characteristic document “TCD.json”, a new time-varying logical item for a property (located at “logicalItemTarget” in the conventional JSON schema corresponding to “TCD.json”) and having the following properties: validTimeKind, validTimeContent, validTimeExistence, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeMaximalExistenceBegin, validTimeMaximalExistenceEnd, validTimeFrequency, transactionTimeKind, transactionTimeContent, transactionTimeExistence, transactionTimeFrequency, itemIdentifierName, itemIdentifierTimeDimension, itemIdentifierKeyRefName,

itemIdentifierKeyRefType, and itemIdentifierPathField. Precisely, this operation inserts a new non-empty object in the “logicalItems” array in the “logical” container of “TCD.json”.

The semantics of this operation is defined by mapping it onto a sequence of low-level operations that have been already proposed in our previous work (Brahmia et al., 2018b), as shown in Algorithm 6.

```
Begin
  AddLogicalItem(TCD.json, logicalItemTarget)
  AddValidTimeToLogicalItem(TCD.json, logicalItemTarget,
    validTimeKind, validTimeContent, validTimeExistence,
    validTimeFrequency)
  AddContentVaryingApplicabilityToValidTimeInLogicalItem(TCD.json,
    logicalItemTarget, contentVaryingApplicabilityBegin,
    contentVaryingApplicabilityEnd)
  SetMaximalExistenceInValidTimeInLogicalItem(TCD.json,
    logicalItemTarget, maximalExistenceBegin,
    maximalExistenceEnd)
  AddTransactionTimeToLogicalItem(TCD.json, logicalItemTarget,
    transactionTimeKind, transactionTimeContent,
    transactionTimeExistence, transactionTimeFrequency)
  AddItemIdentifierToLogicalItem(TCD.json, logicalItemTarget,
    itemIdentifierName, itemIdentifierTimeDimension)
  AddKeyRefToItemIdentifier(TCD.json, logicalItemTarget,
    itemIdentifierName, keyRefName, keyRefType)
  AddFieldToItemIdentifier(TCD.json, logicalItemTarget,
    itemIdentifierName, pathField)
End
```

Algorithm 6 Semantics of the **DefineTimeVaryingLogicalItem** operation

5.3 High-level Operations Specific to the Temporal Physical Characteristics

We have defined ten high-level operations for changing temporal logical characteristics. We organized them into three categories: (i) operations that act on the whole temporal physical characteristic set, presented in the subsection 5.3.1, (ii) operations that act on a portion of a temporal physical characteristic set, described in the subsection 5.3.2, and (iii) operations that act on a physical timestamp item, presented in the subsection 5.3.3.

5.3.1 High-level Operations dealing with the whole Temporal Physical Characteristic Set

We have defined three high-level operations that act on the whole temporal physical characteristic set. They are as follows:

- **InsertTemporalPhysicalCharacteristicSet**,

Z. Brahmia et al.

- RemoveTemporalPhysicalCharacteristicSet, and
- ReplaceTemporalPhysicalCharacteristicSet.

Due to space limitation, we present here only the InsertTemporalPhysicalCharacteristicSet operation. The full list of high-level operations dealing with the whole temporal physical characteristic set can be found in the online appendix (Brahmia et al., 2019b: Sec. A2.3.1).

```
InsertTemporalPhysicalCharacteristicSet(TCD.json,  
temporalPhysicalCharacteristicSet)
```

It adds, in a temporal characteristic document “TCD.json”, a set of temporal physical characteristics (“temporalPhysicalCharacteristicSet”); this latter is either a string explicitly containing the text of the new temporal physical characteristic set, or a file name corresponding to a JSON document that stores the new temporal physical characteristic set. Since the ordering of items in the “stamps” array, in the “physical” container, is unimportant, this operation adds the new set of temporal physical characteristics (i.e., the new items of the “stamps” array) at the end of the existing set of temporal physical characteristics, in the “stamps” array within the “physical” container. If these do not already exist in “TCD.json”, this operation first creates them and then inserts the new set of temporal physical characteristics.

Example: Suppose that the NSDBA wants to add a set of temporal physical characteristics in the temporal characteristic document “EnterpriseTemporalCharacteristics.json”. To do this, he/she calls the InsertTemporalPhysicalCharacteristicSet operation as follows:

```
InsertTemporalPhysicalCharacteristicSet(  
EnterpriseTemporalCharacteristics.json,  
{"physical":  
  "stamps": [  
    { "target": "$.properties.enterprise",  
      "dataInclusion": "expandedVersion",  
      "stampKind": {  
        "timeDimension": "transactionTime",  
        "stampBounds": "step" } },  
    { "target": "$..customers",  
      "dataInclusion": "expandedVersion",  
      "stampKind": {  
        "timeDimension": "bitemporal",  
        "stampBounds": "extent" } },  
    { "target": "$..price",  
      "dataInclusion": "expandedVersion",  
      "stampKind": {  
        "timeDimension": "bitemporal",  
        "stampBounds": "extent" } } ] })
```

5.3.2 High-level Operations dealing with a Subset of the Temporal Physical Characteristics

We have defined three high-level operations that act on a subset of the temporal physical

characteristics. They are as follows:

- InsertTemporalPhysicalCharacteristicSubSet,
- RemoveTemporalPhysicalCharacteristicSubSet, and
- ReplaceTemporalPhysicalCharacteristicSubSet.

Due to space limitation, we present here only the ReplaceTemporalPhysicalCharacteristicSubSet operation. The full list of high-level operations dealing with a subset of temporal physical characteristics can be found in the online appendix (Brahmia et al., 2019b: Sec. A2.3.2).

```
ReplaceTemporalPhysicalCharacteristicSubSet(TCD.json,  
beginningIndexStampPath, endingIndexStampPath,  
newTemporalPhysicalCharacteristicSubSet)
```

It replaces, in a temporal characteristic document “TCD.json”, a subset of the temporal physical characteristics, delimited by a beginning physical stamp (located at “beginningIndexStampPath” which specifies its index in the “stamps” array) and an ending physical stamp (located at “endingIndexStampPath”, specifying its index in the “stamps” array), with a new subset of temporal physical characteristics (“newTemporalPhysicalCharacteristicSubSet”); this latter is either a string explicitly containing the text of the new temporal physical characteristic subset, or a file name corresponding to a JSON document that stores the new temporal physical characteristic subset.

5.3.3 High-level Operations dealing with Physical Timestamps

Since a single temporal physical characteristic is described by a physical timestamp (i.e., an item in the “stamps” array in the “physical” container), high-level operations for changing temporal physical characteristics must include operations acting on such a physical timestamp (i.e., on an item of the “stamps” array). According to the schema of the temporal characteristic document presented in (Brahmia et al., 2018b: Sec. 3), an item of the “stamps” array is a complex item with five properties (“target”, “dataInclusion”, “stampKind”, “defaultTimeFormat”, and “orderBy”). Since the maximal occurrence of each one of the properties of such an item (in the “stamps” array) is one, we were able to define high-level operations for managing it without any problem.

Therefore, based on these premises, we have proposed the following list of four high-level operations for defining, removing and changing physical timestamps:

- SpecifyPhysicalTimeStamp;
- RemovePhysicalTimeStamp;
- ChangePhysicalTimeStamp;
- ReplacePhysicalTimeStamp.

Due to space limitation, we present here only the SpecifyPhysicalTimeStamp operation. The full list of high-level operations dealing with physical timestamps can be found in the online appendix (Brahmia et al., 2019b: Sec. A2.3.3).

```
SpecifyPhysicalTimeStamp(TCD.json, stampTarget,  
stampDataInclusion, timeDimensionStampKind,  
stampBoundsStampKind, pluginStampKindFormat,
```



```
granularityStampKindFormat, calendarStampKindFormat,
propertiesStampKindFormat, valueSchemaStampKindFormat,
targetFieldOrderBy, timeDimensionFieldOrderBy)
```

It defines, within a temporal characteristic document “TCD.json”, a new physical timestamp for a property (located at “stampTarget” in the conventional JSON schema corresponding to “TCD.json”) having the following properties: stampDataInclusion, timeDimensionStampKind, stampBoundsStampKind, pluginStampKindFormat, granularityStampKindFormat, calendarStampKindFormat, propertiesStampKindFormat, valueSchemaStampKindFormat, targetFieldOrderBy, and timeDimensionFieldOrderBy. This operation inserts a new non-empty object in the “stamps” array, in the “physical” container of “TCD.json”.

The semantics of this operation is defined by mapping it onto a sequence of low-level operations that have been already proposed in our previous work (Brahmia et al., 2018b), as shown in Algorithm 7.

Begin

```
AddStamp(TCD.json, stampTarget, stampDataInclusion,
           timeDimensionStampKind, stampBoundsStampKind)
SetFormatInStampKind(TCD.json, stampTarget,
                       pluginStampKindFormat, granularityStampKindFormat,
                       calendarStampKindFormat, propertiesStampKindFormat,
                       valueSchemaStampKindFormat)
AddOrderByFieldToStamp(TCD.json, stampTarget,
                          targetFieldOrderBy, timeDimensionFieldOrderBy)
```

End

Algorithm 7 Semantics of the **SpecifyPhysicalTimeStamp** operation

Example: Suppose that the NSDBA wants to annotate the element “equipments” with a new physical timestamp (i.e., by means of a new item in the “stamps” array) in the temporal characteristic document “EnterpriseTemporalCharacteristics.json”. To do this, he/she calls the **SpecifyPhysicalTimeStamp** operation as follows:

```
SpecifyPhysicalTimeStamp(EnterpriseTemporalCharacteristics.json,
                          "$.properties.entreprise.properties.equipments",
                          "expandedVersion", "bitemporal", "extent", , , , , , )
```

The new item, which will be added to the “stamps” array in the temporal characteristic document “EnterpriseTemporalCharacteristics.json”, is as follows:

```
{ "target": "$.properties.entreprise.properties.equipments",
  "dataInclusion": "expandedVersion",
  "stampKind": {
    "timeDimension": "bitemporal",
    "stampBounds": "extent" } }
```

6 Application Example

As a motivating application example, let us consider a new international IT company that decided to use a temporal JSON NoSQL database with schema versioning for managing the details of its employees. Suppose that on March 10, 2018, the NSDBA created the first version of the temporal JSON schema of the employees (as shown in Figure 8), while using two files that have been already defined with some existing JSON tools (e.g., JSON Schema editor of Altova XMLSpy 2019, JSON Editor Online) or reused from another JSON-based project:

(i) a JSON Schema file (as shown in Figure 6) that represents the first version of the conventional JSON schema of the employees, in which each employee is described by an SSN, a name, a title, and a salary;

(ii) a JSON file (as shown in Figure 7) that represents the first version of the temporal logical and temporal physical characteristic document associated to the employees' conventional JSON schema. As to temporal logical characteristics, we suppose that it was specified that the content of the "salary" property is varying in valid-time, in order to keep the history along valid time of the changes the salary of each employee undergoes. As to temporal physical characteristics, we suppose that a transaction-time physical timestamp was added to the object "employee", which means that whenever any property of the object "employee" changes, the entire "employee" object is repeated to represent a new temporal version.

To reach his/her goal, the NSDBA could use a single high-level operation (CreateTemporalJSONSchema) that can be executed with the following schema change transaction:

Begin Transaction

```
(i) DefineTemporalJSONSchema("emp_TemporalJSONSchema.json",  
    "emp_Schema.json", "emp_ConventionalJSONSchema_V1.json",  
    "emp_TemporalAspects.json", "emp_TemporalCharacteristics_V1.json")
```

Commit

```
{ "$schema": "http://json-schema.org/draft-04/schema#",  
  "id": "http://jsonschema.net",  
  "type": "object",  
  "properties": {  
    "employees": {  
      "id": "http://jsonschema.net/employees",  
      "type": "array",  
      "items": {  
        "type": "object",  
        "properties": {  
          "employee": {  
            "type": "object",  
            "properties": {  
              "SSN": { "type": "string" },  
              "name": { "type": "string" },  
              "title": { "type": "string" },
```

```
        "salary":{"type":"number"} },
        "required":["SSN", "name", "title", "salary"] } },
        "required":["employee"] } } },
        "required": ["employees"] }
```

Figure 6 The first version of the conventional JSON schema of the employees (emp_ConventionalJSONSchema_V1.json) created on March 10, 2018.

```
{ "temporalCharacteristicSet":{
  "logical":{
    "logicalItems":[
      { "target":"$.properties.employees..employee.properties.salary",
        "validTime":{
          "kind":"state",
          "content":"varying",
          "existence":"constant" } } ] },
    "physical":{
      "stamps":[
        { "target":"$.properties.employees.items.properties.employee",
          "dataInclusion":"expandedVersion",
          "stampKind":{
            "timeDimension":"transactionTime",
            "stampBounds":"extent" } } ] } } }
```

Figure 7 The first version of the temporal characteristic document (emp_TemporalCharacteristics_V1.json) created on March 10, 2018.

```
{ "temporalJSONSchema":{
  "conventionalJSONSchema":{
    "sliceSequence":[
      { "slice":{
          "location":"emp_ConventionalJSONSchema_V1.json",
          "begin":"2018-03-10" } } ] },
    "temporalCharacteristicSet":{
      "sliceSequence":[
        { "slice":{
            "location":"emp_TemporalCharacteristics_V1.json",
            "begin":"2018-03-10" } } ] } } }
```

Figure 8 The temporal JSON schema of the employees (emp_TemporalJSONSchema.json) created on March 10, 2018.

After that, assume that on July 28, 2018, the NSDBA realized that he/she needed to add, to the conventional schema of employees, information on temporary employees, reusing to this purpose an existing JSON schema (as shown in Figure 9) that describes them. Moreover, he/she wants to rename the old “employee” property to become “permanent_employee”. As for temporal characteristics, the NSDBA wants to declare a temporal logical characteristic corresponding to the “salary” property of a

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework

“temporary_employee”, and to define a temporal physical characteristic on the property “temporary_employee”.

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "temporary_employee": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "firstName": {"type": "string"},
          "lastName": {"type": "string"},
          "hireDate": {"type": "string"},
          "period": {"type": "integer"},
          "hourlyRate": {"type": "number"} },
        "required": ["firstName", "lastName", "hireDate", "period",
                    "hourlyRate"] } } },
    "required": ["temporary_employee"] }
```

Figure 9 A JSON schema for temporary employees (temporary-employees.json).

The resulting second version of the conventional JSON schema is shown in Figure 10 and that of the temporal characteristic document is shown in Figure 11. Consequently, the temporal JSON schema is also updated by adding a new slice related to the new version of the conventional JSON schema and a new slice related to the new version of the temporal characteristic document, as shown in Figure 12. Changes are evidenced in purple bold type.

```
{ "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://jsonschema.net",
  "type": "object",
  "properties": {
    "employees": {
      "id": "http://jsonschema.net/employees",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "permanent_employee": {
            "type": "object",
            "properties": {
              "SSN": {"type": "string"},
              "name": {"type": "string"},
              "title": {"type": "string"},
              "salary": {"type": "number"} },
            "required": ["SSN", "name", "title", "salary"] },
          }
        }
      }
    }
  }
```

```
      "temporary_employee":{
        "type":"object",
        "properties":{
          "firstName":{"type":"string"},
          "lastName":{"type":"string"},
          "hireDate":{"type":"string"},
          "salary":{"type":"number" } },
          "required":["firstName", "lastName", "hireDate",
            "salary" ] } },
      "required":["permanent_employee" ] } } },
    "required": ["employees" ] }
```

Figure 10 The second version of the conventional JSON schema of the employees (emp_ConventionalJSONSchema_V2.json) created on July 28, 2018.

```
{ "temporalCharacteristicSet":{
  "logical":{
    "logicalItems":[
      { "target":"$.properties.employees..permanent_employee.
        properties.salary",
      "validTime":{
        "kind":"state",
        "content":"varying",
        "existence":"constant" } },
      { "target":"$.properties.employees..temporary_employee.
        properties.salary",
      "validTime":{
        "kind":"state",
        "content":"varying",
        "existence":"constant" } } ] },
  "physical":{
    "stamps":[
      { "target":"$.properties.employees.items.properties.
        permanent_employee",
      "dataInclusion":"expandedVersion",
      "stampKind":{
        "timeDimension":"transactionTime",
        "stampBounds":"extent" } },
      { "target":"$.properties.employees.items.properties.
        temporary_employee",
      "dataInclusion":"expandedVersion",
      "stampKind":{
        "timeDimension":"transactionTime",
        "stampBounds":"extent" } } ] } } }
```

Figure 11 The second version of the temporal characteristic document (emp_TemporalCharacteristics_V2.json) created on July 28, 2018.

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework

```
{ "temporalJSONSchema":{
  "conventionalJSONSchema":{
    "sliceSequence":[
      { "slice":{
        "location":"emp_ConventionalJSONSchema_V1.json",
        "begin":"2018-03-10" } },
      { "slice":{
        "location":"emp_ConventionalJSONSchema_V2.json",
        "begin":"2018-07-28" } } ] },
  "temporalCharacteristicSet":{
    "sliceSequence":[
      { "slice":{
        "location":"emp_TemporalCharacteristics_V1.json",
        "begin":"2018-03-10" } },
      { "slice":{
        "location":"emp_TemporalCharacteristics_V2.json",
        "begin":"2018-07-28" } } ] } } }
```

Figure 12 The temporal JSON schema of the employees (emp_TemporalJSONSchema.json) created on July 28, 2018.

The sequence of high-level operations, that have been performed on the temporal JSON schema (emp_TemporalJSONSchema.json, Figure 8) to update it (as shown in Figure 12), on the first version of the conventional JSON schema (emp_ConventionalJSONSchema_V1.json, Figure 6) to produce the second one (emp_ConventionalJSONSchema_V2.json, Figure 10), and on the first version of the temporal characteristic document (emp_TemporalCharacteristics_V1.json, Figure 7) to produce the second one (emp_TemporalCharacteristics_V2.json, Figure 11), is listed in the schema change transaction which follows:

Begin Transaction

```
(i) UpdateTemporalJSONSchema("emp_TemporalJSONSchema.json",
    current, "emp_ConventionalJSONSchema_V2.json",
    current, "emp_TemporalCharacteristics_V2.json")

(ii) MergeConventionalJSONSchema("emp_ConventionalJSONSchema_V2.json",
    "temporary-employees.json",
    "$.properties.employees..employee", after)

(iii) RenameProperty("emp_ConventionalJSONSchema_V2.json",
    "$.properties.employees..employee", "permanent_employee")

(iv) DefineTimeVaryingLogicalItem("emp_TemporalCharacteristics_V2.json",
    "$.properties.employees..temporary_employee.properties.salary",
    "state", "varying", "constant", , , , , , , , , , , )

(v) SpecifyPhysicalTimeStamp("emp_TemporalCharacteristics_V2.json",
    "$.properties.employees..temporary_employee",
    "expandedVersion", "transactionTime", "extent", , , , , , , )
```

Commit

The transaction time associated to the execution of the transaction above is 2018-07-28, which is used as value of `begin` in the temporal JSON schema file.

7 Related Work Discussion

In this section, we briefly review studies on schema changes and schema versioning in NoSQL databases used for storing and managing temporal Big Data, while trying to clarify our contribution with respect to the state of the art.

First of all, we notice that, to the best of our knowledge, there is no work except our previous work (Brahmia et al., 2017, 2018b, 2019a) that has studied temporal schema versioning in temporal NoSQL databases. All existing related works have dealt with either schema evolution in NoSQL databases, by keeping only the last schema version with its instances, or instance versioning under a static schema. Moreover, all these works (including ours) have not dealt with high-level schema change operations; they have only provided low-level operations. We present and discuss these works in the paragraphs that follow.

The Scherzinger’s team has proposed several important contributions (Scherzinger et al., 2013, 2015a, 2015b, 2016; Cerqueus et al., 2015a, 2015b; Klettke et al., 2016; Haubold et al., 2017; Störl et al., 2015) in the field of schema evolution in JSON NoSQL databases. We survey them in the following.

In (Scherzinger et al., 2013), a set of five primitives (i.e., to add, delete, rename, move, and copy a property of an entity) have been proposed for managing changes to properties of entities in a JSON NoSQL database, while keeping only the last version of each modified JSON schema and adapting existing instances to it. In contrast to this approach, we have proposed a very large set of high-level operations for changing conventional JSON schema, temporal characteristics, and temporal JSON schemas, in an environment that supports temporal schema versioning.

A JSON schema management tool, named Cleager, has been proposed in (Scherzinger et al., 2015a). It supports the low-level operations proposed in (Scherzinger et al., 2013) and executes them as MapReduce jobs on the Google Cloud Platform. Schema change propagation is carried out eagerly, that is schema changes are systematically propagated to all concerned instances. Notice here that our work provides more operations for changing JSON schemas than the Cleager tool. As for the proposed schema change propagation strategy, our approach supports a similar strategy since it directly applies all changes that have been performed on a conventional JSON schema version (i.e., the current one) to each conventional JSON document version that was valid to the changed schema version, in order to produce a new conventional JSON document version that is valid to the new conventional schema version.

Störl et al. (2015) have studied a representative example of Object-NoSQL Mappers for Java development and have shown that these libraries provide some support for a set of basic schema change operations (e.g., to add, delete, rename, an attribute/property of an object/entity; to add, delete, and rename an object). Nevertheless, these mappers do not support complex schema change operations. Contrary to existing Object-NoSQL Mappers, our approach offers a very large set of high-level operations and, therefore,

allows NSDBAs and application developers to execute any desired basic or complex change on a JSON schema, through the use of a valid sequence of these operations.

Another system prototype for managing changes to JSON schemas, named ControVol, has been introduced in (Scherzinger et al., 2015b; Cerqueus et al., 2015a, 2015b) and supports a lazy propagation of schema changes. It is an Eclipse plugin that has been built on top of the Google Cloud Datastore using the Object-NoSQL Mapper Objectify. It detects changes to structures of JSON entities (e.g., to add, delete, or rename a property of an entity) in the Java application source code, which are not compatible with structures of corresponding JSON entities that are stored in NoSQL data store. Moreover, it reports warnings to application developers, and proposes solutions to resolve such problems. In contrast to the ControVol's technique, our approach does not study JSON schema changes in program source codes, written by application programmers, but deals with JSON schema changes that are performed by NSDBAs. In fact, we think that JSON schema changes have to be actually effected by NSDBAs via suitable interfaces, to respond to changes in the modelled reality (e.g., changes in the requirements of end users), which require changes to the JSON schemas. Besides, contrary to ControVol, our approach does not support lazy schema change propagation. It automatically propagates to the JSON instance level the effects of all changes that have been executed at the JSON schema level, in an eager manner. In fact, we think that an eager migration is the most appropriate propagation modality for a temporal JSON NoSQL database supporting schema versioning, for the following reasons: (1) it guarantees a continuous conformity of instances with regard to their schemas, which can be directly tested by validator tools at any time without the need to catch up still unapplied data changes and which translates into a permanent database consistency; (2) changing the schema of a NoSQL database is a quite infrequent task during the database lifetime, so that eager migration has a little impact on the average performance; (3) since the schema is versioned along transaction-time, the eager migration is the only correct choice in case data are also versioned along transaction-time, since a synchronous interaction between data instances and their schemas is required as shown in (De Castro et al., 1997). Notice that a synchronous management along transaction time is incompatible with lazy migration as it would require that changed data are assigned the transaction time of their schema creation even though the change is lazily applied later, when data are first accessed by an application, which would violate the semantics of transaction time.

The difference between the two strategies for JSON schema change propagation, i.e., eager and lazy migration, has been studied in (Scherzinger et al., 2016), while using a system prototype, named Datalution. Similarly to Cleager, Datalution implements the operations introduced in (Scherzinger et al., 2013). The authors focus on showing the benefits of lazy data migration, as it converts legacy entities on-the-fly, one at-a-time, when they are loaded by the application. Eager data migration is preferable when all legacy entities should be migrated in one go. Contrary to Datalution, our approach supports only eager conversion of JSON data. As mentioned in the previous paragraph, we think that eager migration is more suitable for a time-varying document-oriented JSON NoSQL data store in the presence of JSON schema versioning support.

Scalability of JSON NoSQL data stores with respect to both long-term JSON schema evolution (i.e., chains of pending JSON schema change operations that have to be applied together) and lazy migration of underlying JSON-based Big Data has been studied in

(Klettke et al., 2016). Precisely, a rule-based composition for chains of pending schema change operations has been proposed and an experimental comparison of four scalable lazy data migration strategies (i.e., lazy stepwise, lazy composed, predictive, and incremental data migration), implemented on top of the MongoDB NoSQL DBMS, has been carried out. It is worth mentioning that chains of pending schema change operations occur when legacy entities, which have been written by an application and then underwent multiple changes to their structures, are finally accessed by the application. Contrary to the approach of Klettke et al. (2016), our approach does not propagate JSON schema changes to underlying JSON data in a lazy manner, for the reasons cited above.

In (Haubold et al., 2017), the authors have extended the ControVol system (Scherzinger et al., 2015b; Cerqueus et al., 2015a, 2015b), which supports only lazy data migration, to a new system, named ControVol Flex, that supports both eager and lazy data migration. Indeed, this latter allows the application programmer to choose, according to the application requirements, either one data migration technique or a combination of the two techniques (i.e., to start an eager data migration in the background, while lazily migrating legacy entities, if the application requests access to them and eager migration has not reached them yet).

Furthermore, KVolve (Saur et al., 2016) is an approach and a tool that have been proposed, for applications developers, to allow them changing, from their programs, structures of JSON entities that are stored in a key-value NoSQL data store, while only the last schema version is kept (i.e., schema evolution only is supported). It allows using some basic schema change operations, like addition, deletion and modification of the property (or field) of an entity, and applies a lazy data migration. The authors have implemented their approach on top of the NoSQL DBMS Redis and have focused on experimentally showing that KVolve minimizes downtime when JSON schemas of NoSQL database evolve. With respect to the KVolve approach, our approach provides more high-level operations for changing both conventional and temporal JSON schemas, bookkeeps all JSON data along with their JSON schema versions (since schema versioning is being supported), and provides an immediate data migration.

The works that are more strictly related with our approach are our previous works (Brahmia et al., 2016, 2017, 2018b, 2019a). Brahmia et al. (2016) have introduced τ JSchema for the management of temporal JSON data without any support of schema versioning. In (Brahmia et al., 2017), we have dealt with versioning of conventional JSON schemas only; we have shown how such schemas can be versioned and proposed a complete and sound set of low-level operations (or primitives) for changing schema components and also another set of primitives for updating temporal JSON schemas. In (Brahmia et al., 2018b, 2019a), the picture has been enhanced by dealing with versioning of temporal logical and temporal physical characteristics; we have described how these characteristics could be versioned and provided a complete and sound set of low-level operations for changing them. In the present work, we complete the picture extending either (Brahmia et al., 2017) and (Brahmia et al., 2018b, 2019a), by proposing high-level operations for changing conventional JSON schemas, temporal characteristics, and temporal JSON schemas. The new high-level operations have been defined on the basis of the low-level operations proposed in the previous works.

As a recap, Table 1 provides a comparison of the approaches presented above, including our proposal, and shows the advantages of this latter. Notice that, in this table, LLOs, N/A, and HLOs mean “low-level operations”, “non applicable”, and “high-level operations”, respectively.

Table 1 Comparison of approaches dealing with schema changes in NoSQL databases

Approach	Instance versioning	Schema versioning	Changes to data structure	Changes to temporal structure	Change propagation
Scherzinger et al. (2013)	No	No	Partial with LLOs	N/A	Partial, lazy migration
Scherzinger et al. (2015a)	No	No	Partial with LLOs	N/A	Partial, eager migration
Störl et al. (2015)	No	No	Partial with LLOs	N/A	
Scherzinger et al. (2015b); Cerqueus et al. (2015a, 2015b)	No	No	Partial with LLOs	N/A	Partial, lazy migration
Scherzinger et al. (2016)	No	No	Partial with LLOs	N/A	Partial, lazy and eager migrations
Klettke et al. (2016)	No	No	Partial with LLOs	N/A	Partial, lazy migration
Haubold et al. (2017)	No	No	Partial with LLOs	N/A	Partial, lazy and eager migrations
Saur et al. (2016)	No	No	Partial with LLOs	N/A	Partial, lazy migration
Brahmia et al. (2016)	Yes, Temporal	No	No	No	No
Brahmia et al. (2017)	Yes, Temporal	Yes, Temporal	Yes with LLOs	Yes with LLOs	Yes, immediate propagation
Brahmia et al. (2018b, 2019a)	Yes, Temporal	Yes, Temporal	No	Yes with LLOs	Yes, immediate propagation
Our present approach	Yes, Temporal	Yes, Temporal	Yes with HLOs	Yes with HLOs	Yes, immediate propagation

8 Conclusion

In this paper, we have extended our previous work on schema versioning in the τ JSchema framework (Brahmia et al., 2017, 2018b, 2019a), by dealing with high-level schema change operations, as they are more useful to NSDBAs and allow them specifying desired changes in a more user-friendly and compact way. Indeed, we have proposed three sets of high-level schema change operations and defined their operational

semantics. The creation and update operations in the first set are for temporal JSON schemas, the ones in the second set are for conventional JSON schemas, and the ones in the third are for temporal logical and physical characteristics. Each one of these operations has been defined as a consistent sequence of low-level schema change operations that we have proposed in our previous works (Brahmia et al., 2017, 2018b, 2019a). We have also classified these operations into basic ones (i.e., high-level operations that cannot be defined via other basic high-level operations) and complex ones.

Our proposed high-level operations have the following advantages: (1) they facilitate the task of NSDBAs as they are user-friendly and allow them to express schema changes in a more compact and intuitive way, in a τ JSchema-based Big Data environment that supports temporal JSON schema versioning; (2) their support can easily be embedded in the implementation of tools for JSON Schema management, in general, and JSON Schema versioning, in particular; (3) they allow a NSDBA to build his/her own high-level operations through the composition of some high-level operations, in order to satisfy a specific requirement; (4) they are consistency preserving, since each operation applied to a consistent τ JSchema schema component (i.e., a conventional JSON schema, a temporal characteristic document, or a temporal JSON schema) produces a consistent new version of this τ JSchema schema component; (5) not only they act on small and medium components, like a property, a temporal logical item, or a physical timestamp, but also on large components, like a portion of a conventional JSON schema or a portion of a temporal characteristic document, or even on entire conventional/temporal JSON schemas and entire temporal characteristic documents.

Currently, we are developing a tool, named τ JSchema-Manager, whose aim is to support such high-level operations, based on the primitives previously introduced in (Brahmia et al., 2017, 2018b, 2019a), and which could allow NSDBAs to change τ JSchema schemas in a friendly and efficient manner.

As a part of our future work, we intend to deal with schema change propagation, by studying the effects of schema changes on the underlying conventional and temporal JSON-based Big Data in order to optimize their execution. Moreover, we also plan to study querying of temporal JSON schemas and temporal JSON Big Data in a fully fledged τ JSchema setting, by proposing a temporal extension of the JSONiq language (Florescu and Fourny, 2013), which is becoming well-known in the JSON NoSQL database community.

References

- Al_Janabi, S. and Hussein, N.Y. (2020) 'The Reality and Future of the Secure Mobile Cloud Computing (SMCC): Survey', in: Farhaoui, Y. (Ed.) *Big Data and Networks Technologies (BDNT 2019)*, Lecture Notes in Networks and Systems, vol 81, pp. 231-261. Springer, Cham. DOI: 10.1007/978-3-030-23672-4_18
- Brahmia, Z., Grandi, F., Oliboni, B. and Bouaziz, R. (2014) 'Schema Change Operations for Full Support of Schema Versioning in the τ XSchema Framework', *International Journal of Information Technology and Web Engineering*, Vol. 9, No. 2, pp. 20-46.
- Brahmia, Z., Grandi, F., Oliboni, B. and Bouaziz, R. (2015) 'Schema Versioning', in: Khosrow-Pour, M. (Ed.) *Encyclopedia of Information Science and Technology (3rd edition)*, IGI Global, Hershey, Pennsylvania, USA, pp. 7651-7661.
- Brahmia, S., Brahmia, Z., Grandi, F. and Bouaziz, R. (2016) ' τ JSchema: 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, pp. 167-181.
- Brahmia, S., Brahmia, Z., Grandi, F. and Bouaziz, R. (2017) 'Temporal JSON Schema Versioning in the τ JSchema Framework', *Journal of Digital Information Management*, Vol. 15, No. 4, pp. 179-202.
- Brahmia, Z., Grandi, F., Oliboni, B. and Bouaziz, R. (2018a) 'Supporting Structural Evolution of Data in Web-Based Systems via Schema Versioning in the τ XSchema Framework', in: Elçi, A. (Ed.), *Handbook of Research on Contemporary Perspectives on Web-Based Systems*, IGI Global, Hershey, PA, USA, pp. 271-307.
- Brahmia, S., Brahmia, Z., Grandi, F. and Bouaziz, R. (2018b) 'Managing Temporal and Versioning Aspects of JSON-based Big Data via the τ JSchema Framework', in: *Proc. of the International Conference on Big Data and Smart Digital Environment (ICBDSDE'2018)*, Casablanca, Morocco, Studies in Big Data, Vol. 53, Springer Nature AG, pp. 27-39.
- Brahmia, S., Brahmia, Z., Grandi, F. and Bouaziz, R. (2019a) 'Versioning Temporal Characteristics of JSON-based Big Data via the τ JSchema Framework', *International Journal of Cloud Computing*, in press.
- Brahmia, Z., Brahmia, S., Grandi, F. and Bouaziz, R. (2019b) 'Online Appendix of the Paper: Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework', 10 February 2019. <http://www-db.disi.unibo.it/~fgrandi/papers/IJCCpaperHLOAppendix.pdf> (accessed: November 20, 2019)
- Cattell, R. (2010) 'Scalable SQL and NoSQL Data Stores', *ACM SIGMOD Record*, Vol. 39, No. 4, pp. 2-27.
- Cerqueus, T., Cunha de Almeida, E. and Scherzinger, S. (2015a) '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, pp. 15-16.

- Cerqueus, T., Cunha de Almeida, E. and Scherzinger, S. (2015b) '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, pp. 4-10.
- Chen, C.P. and Zhang, C.Y. (2014) 'Data-intensive applications, challenges, techniques and technologies: A survey on Big Data', *Information Sciences*, Vol. 275, pp. 314-347.
- Corbellini, A., Mateos, C., Zunino, A., Godoy, D. and Schiaffino, S.N. (2017) 'Persisting big-data: The NoSQL landscape', *Information Systems*, Vol. 63, pp. 1-23.
- Currim, F., Currim, S., Dyreson, C.E. and Snodgrass, R.T. (2004) 'A Tale of Two Schemas: Creating a Temporal XML Schema from a Snapshot Schema with τ XSchema', in: *Proc. of the 9th International Conference on Extending Database Technology (EDBT 2004)*, Heraklion, Crete, Greece, pp. 348-365.
- 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, pp. 180-185.
- Davoudian, A., Chen, L. and Liu, M. (2018) 'A Survey on NoSQL Stores', *ACM Computing Surveys*, Vol. 51, No. 2, Article 40.
- De Castro, C., Grandi, F. and Scalas, M.R. (1997) 'Schema Versioning for Multitemporal Relational Databases', *Information Systems*, Vol. 22, No. 5, pp. 249-290.
- Florescu, D. and Fourny, G. (2013) 'JSONiq: The History of a Query Language', *IEEE Internet Computing*, Vol. 17, No. 5, pp. 86-90.
- Gudivada, V.N., Rao D. and 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, pp. 190-197.
- Guerrini, G., Mesiti, M. and Rossi, D. (2005) 'Impact of XML Schema Evolution on Valid Documents', in: *Proc. of the 7th ACM International Workshop on Web Information and Data Management (WIDM 2005)*, Bermen, Germany, pp. 39-44.
- Haubold, F., Schildgen, J., Scherzinger, S. and Deßloch, S. (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)*, Stuttgart, Germany, pp. 601-604.
- Information Resources Management Association (IRMA) (2016) *Big data: Concepts, methodologies, tools, and applications*, IGI Global, Hershey, PA, USA.
- Internet Engineering Task Force (IETF) (2013a) *JSON Schema: core definition and terminology*, Internet-Draft, 31 January 2013. <https://tools.ietf.org/html/draft-zyp-json-schema-04> (accessed: November 20, 2019)
- Internet Engineering Task Force (IETF) (2013b) *JSON Schema: interactive and non interactive validation*, Internet-Draft, 1 February 2013. <http://tools.ietf.org/html/draft->

Versioning Schemas of JSON-based Conventional and Temporal Big Data through High-level Operations in the τ JSchema Framework

fge-json-schema-validation-00 (accessed: November 20, 2019)

- Internet Engineering Task Force (IETF) (2014) *The JavaScript Object Notation (JSON) Data Interchange Format*, Internet Standards Track document, March 2014.
- Khosla, P.K. and Kaur, A. (2018) ‘Big Data Technologies’, in: Mittal, M., Balas, V.E., Hemanth, D.J. and Kumar, R. (Eds.) *Data Intensive Computing Applications for Big Data*, IOS Press, Amsterdam, The Netherlands, pp. 28-55.
- Klettke, M., Störl, U., Shenavai, M. and Scherzinger, S. (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, pp. 2764-2774.
- Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M. and Vrgoč, D. (2016) ‘Foundations of JSON schema’, in: *Proc. of the 25th International Conference on World Wide Web (WWW 2016)*, Montreal, Canada, pp. 263-273.
- Pokorný, J. (2013) ‘NoSQL databases: a step to database scalability in web environment’, *International Journal of Web Information Systems*, Vol. 9, No. 1, pp. 69-82.
- Roddick, J.F. (2018) ‘Schema Versioning’, in: Liu, L. and Özsu, M.T., (Eds.), *Encyclopedia of Database Systems (2nd edition)*, Springer-Verlag, New York, USA. DOI: 10.1007/978-1-4614-8265-9_323
- Saur, K., Dumitras, T. and 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, pp. 166-176.
- Scherzinger, S., Klettke, M. and 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.
- Scherzinger, S., Klettke, M. and 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, pp. 659-662.
- Scherzinger, S., Cerqueusy, T. and Cunha de Almeida, E. (2015b) ‘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, pp. 1464-1467.
- Scherzinger, S., Sombach, S., Wiech, K., Klettke, M. and Störl, U. (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, pp. 38-39.
- Sharma, S., Tim, U.S., Wong, J., Gadia, S.K. and Sharma, S. (2014) ‘A brief review on leading big data models’, *Data Science Journal*, Vol. 13, pp. 138-157.

Z. Brahmia et al.

- Sharma, S., Tim, U.S., Gadia, S.K., Wong, J., Shandilya, R. and Peddoju, S.K. (2015) 'Classification and comparison of NoSQL big data models', *International Journal of Big Data Intelligence*, Vol. 2, No. 3, pp. 201-221.
- Snodgrass, R.T. (ed.), Ahn, I., Ariav, G., Batory, D.S., Clifford, J., Dyreson, C.E., Elmasri, R., Grandi, F., Jensen, C.S., Käfer, W., Kline, N., Kulkarni, K., Cliff Leung, T.Y., Lorentzos, N., Roddick, J.F., Segev, A., Soo, M.D. and Sripada, S.M. (1995) *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers, Norwell, MA, USA.
- Snodgrass, R.T., Dyreson, C.E., Currim, F., Currim, S. and Joshi, S. (2008) 'Validating quicksand: Temporal schema versioning in τ XSchema', *Data and Knowledge Engineering*, Vol. 65, No. 2, pp. 223-242.
- Störl, U., Hauf, T., Klettke, M. and Scherzinger, S. (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, pp. 579-599.
- Tiwari, S. (2011) *Professional NoSQL*, John Wiley & Sons, Inc., Indianapolis, Indiana, USA.
- Wang, Y. (2017) *JSON and the Confusion of Formats in Big Data*. Credit Karma. <https://engineering.creditkarma.com/json-and-the-confusion-of-formats-in-big-data/> (accessed: November 20, 2019)