



## ARCHIVIO ISTITUZIONALE DELLA RICERCA

### Alma Mater Studiorum Università di Bologna Archivio istituzionale della ricerca

tauJUpdate: A Temporal Update Language for JSON Data

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

*Published Version:*

tauJUpdate: A Temporal Update Language for JSON Data / Brahmia, Zouhaier; Grandi, Fabio; Brahmia, Safa; Bouaziz, Rafik. - STAMPA. - 13761:(2023), pp. 250-263. (Intervento presentato al convegno 11th International Conference on Model and Data Engineering (MEDI 2022) tenutosi a Cairo, Egitto nel 21-24 Novembre 2022) [10.1007/978-3-031-21595-7\_18].

This version is available at: <https://hdl.handle.net/11585/910321> since: 2024-02-28

*Published:*

DOI: [http://doi.org/10.1007/978-3-031-21595-7\\_18](http://doi.org/10.1007/978-3-031-21595-7_18)


*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.

(Article begins on next page)

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

This is the final peer-reviewed accepted manuscript of:

Brahmia, Z., Grandi, F., Brahmia, S., Bouaziz, R. (2023). JUpdate: A Temporal Update Language for JSON Data. In: Fournier-Viger, P., Hassan, A., Bellatreche, L. (eds) Model and Data Engineering. MEDI 2022. Lecture Notes in Computer Science, vol 13761. Springer, Cham.

The final published version is available online at: [https://doi.org/10.1007/978-3-031-21595-7\\_18](https://doi.org/10.1007/978-3-031-21595-7_18)

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.***

# $\tau$ JUpdate: A Temporal Update Language for JSON Data

Zouhaier Brahmia<sup>1</sup>[0000-0003-0577-1763], Fabio Grandi<sup>2</sup>[0000-0002-5780-8794],  
Safa Brahmia<sup>3</sup>[0000-0001-9304-4819], and Rafik Bouaziz<sup>4</sup>[0000-0001-5398-462X]

<sup>1</sup> University of Sfax, Tunisia, [zouhaier.brahmia@fsegs.rnu.tn](mailto:zouhaier.brahmia@fsegs.rnu.tn)

<sup>2</sup> University of Bologna, Italy, [fabio.grandi@unibo.it](mailto:fabio.grandi@unibo.it)

<sup>3</sup> University of Sfax, Tunisia, [safa.brahmia@gmail.com](mailto:safa.brahmia@gmail.com)

<sup>4</sup> University of Sfax, Tunisia, [rafik.bouaziz@usf.tn](mailto:rafik.bouaziz@usf.tn)

**Abstract.** Time-varying JSON data are being used and exchanged in a lot of today's application frameworks like IoT platforms, Web services, cloud computing, online social networks, and mobile systems. However, in the state-of-the-art of JSON data management, there is neither a consensual nor a standard language for updating (i.e., inserting, modifying, and deleting) temporal JSON data, like the TSQL2 or SQL:2016 language for temporal relational data. Moreover, existing JSON-based NoSQL DBMSs (e.g., MongoDB, Couchbase, CouchDB, OrientDB, and Riak) and both commercial DBMSs (e.g., IBM DB2 12, Oracle 19c, and MS SQL Server 2019) and open-source ones (e.g., PostgreSQL 15, and MySQL 8.0) do not provide any support for maintaining temporal JSON data. Also in our previously proposed temporal JSON framework, called  $\tau$ JSchema, there was no feature for temporal JSON instance update. For these reasons, we propose in this paper a temporal update language, named  $\tau$ JUpdate (Temporal JUpdate), for JSON data in the  $\tau$ JSchema environment. We define it as a temporal extension of our previously introduced non-temporal JSON update language, named JUpdate (JSON Update). Both the syntax and the semantics of the data modification operations of JUpdate have been extended to support temporal aspects.  $\tau$ JUpdate allows (i) to specify temporal JSON updates in a user-friendly manner, and (ii) to efficiently execute them.

**Keywords:** JSON · Temporal JSON · JUpdate · Temporal JSON data manipulation · JSON update operation ·  $\tau$ JSchema · Conventional JSON instance · Temporal JSON instance

## 1 Introduction

The lightweight format JavaScript Object Notation (JSON) [15], which is endorsed by the Internet Engineering Task Force (IETF), is currently being used by a lot of networked applications to store and exchange data. Moreover, many of these applications running in IoT, cloud-based and mobile environments, like Web services, online social networks, e-health, smart-city and smart-grid applications, require bookkeeping of the full history of JSON data updates so that

they can handle temporal JSON data, audit and recover past JSON document versions, track JSON document changes over time, and answer temporal queries.

However, in the state-of-the-art of JSON data management [21, 17, 20, 1, 27, 22, 6], there is neither a consensual nor a standard language for updating (i.e., inserting, modifying, and deleting) temporal JSON data, like the TSQL2 (Temporal SQL2) [28] or SQL:2016 [24] language for temporal relational data. It is worth mentioning here that the extension of the SQL language, named SQL/JSON [23, 29, 18] and standardized by ANSI to empower SQL to manage queries and updates on JSON data, has no built-in support for updating time-varying JSON data. In fact, even for non-temporal data, SQL/JSON is limited since it does not support the update of a portion of a JSON document through the SQL UPDATE statement [26].

Moreover, existing JSON-based NoSQL database management systems (DBMSs) (e.g., MongoDB, Couchbase, CouchDB, DocumentDB, MarkLogic, OrientDB, RethinkDB, and Riak) and both commercial DBMSs (e.g., IBM DB2 12, Oracle 19c, and Microsoft SQL Server 2019) and open-source ones (e.g., PostgreSQL 15, and MySQL 8.0) do not provide any support for maintaining temporal JSON data [3, 11, 13].

In this context, with the aim of having an infrastructure that allows efficiently creating and validating temporal JSON instance documents and inspired by the  $\tau$ XSchema design principles [9], we have proposed in [2] a comprehensive framework, named  $\tau$ JSchema (Temporal JSON Schema). In this environment, temporal JSON data are produced from conventional (i.e., non temporal) JSON data, by applying a set of temporal logical and physical characteristics that have been already specified by the designer on the conventional JSON schema, that is a JSON Schema [14] file that defines the structure of the conventional JSON data:

- the temporal logical characteristics [2] allow designers to specify which components (e.g., objects, object members, arrays, . . . ) of the conventional JSON schema can vary over valid and/or transaction time;
- the temporal physical characteristics [2] allow designers to specify where timestamps should be placed and how the temporal aspects should be represented.

A temporal JSON schema is generated from a conventional JSON schema and the set of temporal logical and physical characteristics that have been specified for this non-temporal JSON schema. Thus, by using temporal JSON schemas and temporal characteristics and by making a separation between conventional JSON data and temporal JSON data, from one hand, and between conventional JSON schema and temporal JSON schema, from the other hand,  $\tau$ JSchema offers the following advantages: (i) it extends the traditional JSON world to temporal aspects in a systematic way; (ii) it guarantees logical and physical data independence [8] for temporal JSON data (i.e., a temporal JSON document, having some physical representation, could be automatically transformed into a different temporal document with a different physical representation while

conserving the semantics of the temporal JSON data, that is keeping the same temporal logical characteristics); (iii) it does not require changes to existing JSON instance/schema files nor revisions of the JSON technologies (e.g., the IETF specification of the JSON format [15], the IETF specification of the JSON Schema language [14], JSON-based NoSQL DBMSs, JSON editors/validators, JSON Schema editors/generators/validators, etc.). However, there is no feature for temporal JSON instance update in  $\tau$ JSchema.

With the aim of overcoming the lack of an IETF standard or recommendation for updating JSON data, we have recently proposed a powerful SQL-like language, named JUpdate (JSON Update) [6], to allow users to perform updates on (non-temporal) JSON data. It provides fourteen user-friendly high-level operations (HLOs) to fulfill the different JSON update requirements of users and applications; not only simple/atomic values but also full portions (or chunks) of JSON documents can be manipulated (i.e., inserted, modified, deleted, copied or moved) The semantics of JUpdate is based on a minimal and complete set of six primitives (i.e., low-level operations, which can be easily implemented) for updating JSON documents. The data model behind JUpdate is the IETF standard JSON data model [15]. Thus, from one hand, it is independent from any underlying DBMSs, which simplifies its use and implementation, and, from the other hand, it can be used to maintain generic JSON documents.

Taking into account the requirements mentioned above, we considered very interesting to fill the evidenced gap and to propose a temporal JSON update language that would help users in the non-trivial task of updating temporal JSON data. Moreover, based on our previous work, we think that (i) the JUpdate language [6] can be a good starting point for deriving such a temporal JSON update language, and (ii) the  $\tau$ JSchema framework can be used as a suitable environment for defining the syntax and semantics of a user-friendly temporal update language, mainly due to its support of logical and physical data independence.

For all these reasons, we propose in this paper a temporal update language for JSON data named  $\tau$ JUpdate (Temporal JUpdate) and define it as a temporal extension of our JUpdate language, to allow users to update (i.e., insert, modify, and delete) JSON data in the  $\tau$ JSchema environment. To this purpose, both the syntax and the semantics of the JUpdate statements have been extended to support temporal aspects. The  $\tau$ JUpdate design allows users to specify in a friendly manner and efficiently execute temporal JSON updates. In order to motivate  $\tau$ JUpdate and illustrate its use, we will provide a running example.

The rest of the paper is structured as follows. The next section presents the environment of our work and motivates our proposal. Section 3 proposes  $\tau$ JUpdate, the temporal JSON instance update language for the  $\tau$ JSchema framework. Section 4 illustrates the use of some operations of  $\tau$ JUpdate, by means of a short example. Section 5 provides a summary of the paper and some remarks about our future work.

## 2 Background and Motivation

In this section, first we briefly describe the  $\tau$ JSchema framework (more details can be found in [2]), and then we present a motivating example that (i) recalls how temporal JSON data are represented under  $\tau$ JSchema, (ii) presents problems and difficulties of dealing with temporal data management using a JUpdate-like language, and (iii) introduces our contributions.

### 2.1 The $\tau$ JSchema Framework

$\tau$ JSchema allows a NoSQL database administrator (NSDBA) to create a temporal JSON schema for temporal JSON instances, from a conventional JSON schema, some temporal logical characteristics, and some temporal physical characteristics. It uses the following two principles: (i) separation between the conventional JSON schema and the temporal JSON schema, and also between the conventional JSON instances and the temporal JSON instances; (ii) use of temporal logical and physical characteristics to specify temporal logical and physical aspects, respectively, at schema level.

Since there are many techniques to make a (non-temporal) JSON document temporal, the logical and physical independence supported by the  $\tau$ JSchema framework represents a real breakthrough in temporal JSON data management, as it separates temporal JSON data design (specified via temporal logical characteristics) from implementation details (specified via temporal physical characteristics). Notice that this aspect is emphasized when dealing with updating, through a JUpdate-like language, temporal JSON documents (in the next subsection). In fact, in JSON documents, some JSON structuring conforming to the conventional JSON schema is devoted to modeling the non-temporal structure of data, whereas some additional JSON structuring is needed to encode the temporal aspects of the data modeling, actually based on some timestamped multi-version representation. Hence, by adopting a  $\tau$ JSchema-based approach, we want to also separate temporal data update specification from implementation details. We want to enable users to manipulate (i.e., insert, modify, and delete) temporal JSON data by reasoning at the level of their conventional JSON schema, abstracting from the knowledge of additional JSON structuring needed to encode low-level data versioning and timestamping details. In practice, we want the users to express their JUpdate updates exactly as if their JSON data were not temporal. The only thing they have to add to their update high-level statements, when dealing with valid-time data, is a VALID clause to specify the “applicability period” of the update in case they want to explicitly manage it. It should be mentioned that our approach in this paper is similar to that proposed in our previous work [7], where an “XQuery Update Facility”-like language is used to support update operations on temporal data that are recorded in XML format, abstracting from their implementation details.

## 2.2 Motivating Example

We assume that a company uses a JSON repository for the storage of the information about the devices that manufactures and sells, where each device is described by its name and cost price. For simplicity, let us consider a temporal granularity of one day for representing the data change events (and, therefore, for temporal data timestamping). We assume that the initial state of the device repository, valid from February 1, 2022, can be represented as shown in Fig. 1: it contains, in a JSON file named `device1.json`, data about one device called `CameraABC` costing €35.

```
{ "devices": [
  { "device": {
    "name": "CameraABC",
    "costPrice": 35 } } ] }
```

**Fig. 1.** The initial state of the device repository (file `device1.json`, on February 01, 2022).

Then, we assume that, effective from April 15, 2022, the company starts producing a new device named `CameraXYZ` with a cost price of €42 and `CameraABC`'s cost price is raised by 8%. The new state of the device repository can be represented in a JSON file named `device2.json` as shown in Fig. 2. Changed parts are presented in red color.

```
{ "devices": [
  { "device": {
    "name": "CameraABC",
    "costPrice": 37.8 } },
  { "device": {
    "name": "CameraXYZ",
    "costPrice": 42 } } ] }
```

**Fig. 2.** A new state of the device repository (file `device2.json`, on April 15, 2022).

Consequently, we consider that the device repository is implemented in the  $\tau$ JSchema framework and that the conventional JSON schema for our JSON device data has been annotated so that "device" is a time-varying object for representing the history of devices along valid time. As a result, the entire history of the device repository can be represented in the temporal JSON document shown in Fig. 3, composed of two slices corresponding to the repository states of Fig. 1 and Fig. 2.

```
{ "temporalJSONDocument": {
  "conventionalJSONDocument": {
    "sliceSequence": [
      { "slice": {
        "location": "device1.json",
        "begin": "2022-02-01" } },
      { "slice": {
        "location": "device2.json",
        "begin": "2022-04-15" } } ] } } }
```

**Fig. 3.** Fig. 3. The temporal JSON document representing the entire history of the device repository (file `deviceTJD.json`, on April 15, 2022).

The temporal JSON document can also be "squashed" to obtain a self-contained temporal JSON document, conformant to the temporal JSON schema

that can be derived from both the conventional JSON schema and the temporal logical and physical characteristics, representing the whole devices' history, as shown in Fig. 4. The valid-time timestamps are presented in blue color.

```

{ "devices": [
  { "device": [
    { "name": "CameraABC",
      "costPrice": 35,
      "VTbegin": "2022-02-01",
      "VTend": "2022-04-14" },
    { "name": "CameraABC",
      "costPrice": 37.8,
      "VTbegin": "2022-04-15",
      "VTend": "Forever" } ] },
  { "device": [
    { "name": "CameraXYZ",
      "costPrice": 42,
      "VTbegin": "2022-04-15",
      "VTend": "Forever" } ] } ] }

```

**Fig. 4.** The squashed JSON document corresponding to the entire history of the device repository (file deviceSJD.json, on April 15, 2022).

Notice that the squashed JSON document deviceSJD.json in Fig. 4 also corresponds to one of the manifold possible representations of our temporal JSON [3] data without the  $\tau$ JSchema approach.

After that, let us consider that we have to record in the device repository that the company has stopped manufacturing the device CameraABC effective from May 25, 2022. At the state-of-the-art of JSON technology, we could use JUpdate HLOs to directly perform the required updates on the deviceSJD.json file in Fig. 4. A skilled developer, expert in both temporal databases and JUpdate, and aware of the precise structure of the squashed document, will satisfy such requirements via the following JUpdate statement:

```

UPDATE deviceSJD.json
PATH $.devices[0.device[0.name="CameraABC"
    && 0.VTend="Forever"].VTend]
VALUE "2022-05-24"

```

(S1)

In practice, deleting the device CameraABC effective from 2022-05-25 means to close to 2022-05-24 the valid timestamp of its last version, assuming for simplicity (and without checking) that the one valid at 2022-05-25 is the last CameraABC's version and, therefore, there are no future versions to delete. Anyway, we think that this is a complex solution for what it is a simple problem (for example, in temporal relational databases).

Thus, our **first contribution** is a temporal extension of the JUpdate language. JUpdate statements will be equipped with a new VALID clause to specify the so-called “applicability period” of the update, that is the time period in which the update has to be in effect (e.g., from 2022-05-25 on, in our example). This solution will allow the developer to formulate the required update as a JUpdate deletion valid from 2022-05-25 of CameraABC's data, relying on the temporal semantics of the language for its correct execution, including version and timestamp management. Nevertheless, working on the temporal JSON document in Fig. 4, this will mean to specify the following DeleteValue operation:



```

DELETE FROM deviceSJD.json
PATH $.devices[@.device[@.name="CameraABC"
&& @.VTend="Forever"]]
VALID from "2022-05-24"

```

(S2)

Although this solution is simpler than solution (S1), it requires from the developer a detailed knowledge of the specific temporal structuring of the JSON file including version organization and timestamping. Another consequence is that such solution template would not be portable to another setting in which a different temporal structuring of JSON data is adopted.

Moreover, our **second contribution** is to integrate the temporal JUpdate extension into the  $\tau$ JSchema framework, in order to enjoy the logical and physical independence property. In this framework, the required update will be specified via the following  $\tau$ JUpdate DeleteValue statement:

```

DELETE FROM deviceSJD.json
PATH $.devices[@.device[@.name="CameraABC"]]
VALID from "2022-05-24"

```

(S3)

The update could be applied either to the temporal JSON document (i.e., deviceTJD.json) or to its squashed version (i.e., deviceSJD.json); the system using the temporal logical and physical characteristics can manage both ways correctly. Notice that, ignoring the VALID clause, the solution (S3) represents exactly the same way we would specify the deletion of the device CameraABC’s data in a non-temporal environment (e.g., executing it on the device2.json file in Fig. 2). In practice, we want to allow the developer to focus on the structuring of data simply as defined in the conventional JSON schema and not on the temporal JSON schema, leaving the implementation details and their transparent management to the system (e.g., the mapping to a squashed JSON document, being aware of the temporal characteristics). This means, for example, that in order to specify a cost price update, we want  $\tau$ JUpdate users be able to deal with updates to the “device.costPrice” value instead of dealing with updates to the “device.costPrice” array of objects, where each object represents a version of a cost price and has three properties: “VTbegin” (the beginning of the valid-time timestamp of the version), “VTend” (the end of the valid-time timestamp of the version), and “value” (the value of the version).

Notice that such a way in which temporal updates of JSON data will be specified with our  $\tau$ JUpdate language, corresponds exactly to the way updates of temporal relational data can be specified using a temporal query language like TSQL2 [28] or SQL:2016 [24], that is using the same update operations that are used in a non-temporal context augmented with a VALID clause to specify the applicability period of each update operation.

In sum, the motivation of our approach is twofold: from one hand, (i) leveraging the logical/physical independence supported by the  $\tau$ JSchema framework to the JUpdate language and, from the other hand, (ii) equipping  $\tau$ JSchema with a user-friendly update language, which is consistent with its design philosophy.

### 3 The $\tau$ JUpdate Language

In this section, we propose the  $\tau$ JUpdate language, by showing how the JUpdate specification [6] has to be extended. More precisely, in Sec. 3.1, we start by presenting the syntax of  $\tau$ JUpdate high-level operations (HLOs) before defining their semantics while considering temporal JSON documents in unsquashed form.

#### 3.1 Syntax and Semantics of $\tau$ JUpdate Update HLOs

The management of transaction time does not require any syntactic extension to the JUpdate language: owing to the transaction time semantics, only current data can be updated and the “applicability period” of the update is always [Now, UntilChanged], which is implied and cannot be overridden by users. On the contrary, the management of valid time is under the user’s responsibility. Hence, syntactic extensions of the JUpdate language are required to allow users to specify a valid time period representing the “applicability period” of the update. To this purpose, the JUpdate update HLOs [6] are augmented with a VALID clause as shown in Fig. 5.

```

 $\tau$ JUpdateHLO ::= JUpdateHLO "VALID" validTimePeriod
JUpdateHLO  ::= ValueChangeHLO | MemberChangeHLO
              | ObjectChangeHLO
ValueChangeHLO ::= InsertValue | DeleteValue | UpdateValue
                | CopyValue | MoveValue
MemberChangeHLO ::= InsertMember | DeleteMember
                 | RenameMember | ReplaceMember
                 | CopyMember | MoveMember
ObjectChangeHLO ::= UpdateObject
validTimePeriod ::= "in [" validTimeBegin "," validTimeEnd "]"
                 | "from" validTimeBegin
                 | "to" validTimeEnd
validTimeBegin ::= "Beginning" | "Now" | temporalValue
validTimeEnd  ::= "Forever" | "Now" | temporalValue

```

Fig. 5. The syntax of  $\tau$ JUpdate HLOs.

Due to space limitations, we do not consider here other JUpdate HLOs (e.g., InsertMember, ReplaceMember, UpdateObjects) as they are used for specifying complex updates; they will be investigated in a future work. Temporal expressions “from T” and “to T”, while T is a temporal value, are used as syntactic sugar for the temporal expressions “in [T, Forever]” and “in [Beginning, T]”, respectively.

As far as the semantics of  $\tau$ JUpdate is concerned, we can define it, for the sake of simplicity, by considering JSON update operations on the temporal JSON document in its unsquashed form. Based on the well-known theory developed in the temporal database field [12, 19], the operational semantics of a  $\tau$ JUpdateHLO, equal to a JUpdateHLO augmented with the VALID clause, can be defined as follows:

- validTimePeriod is evaluated. The result must be a valid period specification; otherwise a type error is raised. Let [vts, vte] be the period resulting from the evaluation.

- Let `jdoc` be the temporal JSON document involved in the update; find in `jdoc` all the temporal slices `jdoc_vers` having a timestamp `VTimestamp` which overlaps `[vts, vte]`.
- For each such slice `jdoc_vers`:
  - let `jdoc_vers'` the result of the evaluation of `JUpdateHLO` on `jdoc_vers`;
  - if `VTimestamp  $\subset$  [vts, vte]` then remove the whole slice `jdoc_vers` from the temporal JSON document `jdoc` (and delete the corresponding JSON file) else restrict to `VTimestamp  $\setminus$  [vts, vte]` the timestamp of `jdoc_vers` in the temporal JSON document `jdoc`;
  - add `jdoc_vers'` to the temporal JSON document `jdoc` as a new slice with timestamp `VTimestamp  $\cap$  [vts, vte]`.
- Coalesce the resulting slices in the temporal JSON document.

The last step aims at limiting the unnecessary proliferation of slices, giving rise to redundant JSON files in the unsquashed setting. Two slices, `jdoc_vers1` and `jdoc_vers2` with timestamps `VTimestamp1` and `VTimestamp2`, respectively, can be coalesced when `jdoc_vers1` and `jdoc_vers2` are equal and `VTimestamp1` meets `VTimestamp2` [28]. In this case, coalescing produces one slice `jdoc_vers1` with timestamp `VTimestamp1  $\cup$  VTimestamp2`.

This definition of the  $\tau$ JUpdate HLO semantics, which can be easily extended to the transaction-time or bitemporal case, is in line with the  $\tau$ JSchema principles, considering a temporal JSON document as representing a sequence of conventional JSON documents, and reuses the standard (non-temporal) JUpdate HLOs.

Even if the temporal JSON document `jdoc` is physically stored in squashed form, the above semantics can still be used to evaluate a  $\tau$ JUpdate HLO after the document has been explicitly unsquashed. The results of the evaluation can then be squashed back to finally produce an updated temporal JSON document. Although correct from a theoretical point of view, such a procedure could be inefficient in practice, in particular when the temporal JSON document is composed of several slices. To resolve this problem, a different method can be applied for updating temporal JSON documents that are stored in squashed form. To this end, the semantics of  $\tau$ JUpdate HLOs can be defined in an alternative way, as shown in the next subsection (the solution is inspired from our previous work on updates to temporal XML data [7]).

## 4 Running Example Reprise

In this section, we resume the motivating example introduced in Sec. 2.2 to illustrate some of the functionalities of  $\tau$ JUpdate.

First of all, starting from the initial state of the device repository containing only the slice in Fig. 1, the second slice in Fig. 2 can be added via the execution of the following sequence of  $\tau$ JUpdate HLOs:

```
INSERT INTO deviceTJD.json
PATH $.devices[last]
```

```

VALUE { "device":{ "name":"CameraXYZ", "costPrice":42 } }
VALID from "2022-04-15";
UPDATE deviceTJD.json
PATH $.devices[device.name="CameraABC"].costPrice
VALUE $.devices[device.name="CameraABC"].costPrice * 1.08
VALID from "2022-04-15"

```

The first one is an example of InsertValue HLO that inserts CameraXYZ's data, while the second one is an example of UpdateValue HLO that increases CameraABC's cost price. The result of this HLO sequence corresponds to the temporal JSON document in Fig. 3 completed by the slices in Fig. 1 and Fig. 2, and which has been shown in squashed form in Fig. 4.

As an example of DeleteValue HLO, we can consider the  $\tau$ JUpdate HLO (S3) in Sec. 2.2, deleting CameraABC's data effective from 2022-05-25. As an example of RenameMember HLO, we can consider changing the name of the "devices" object to "products", also valid from 2022-05-25. Notice that such an operation could be more properly considered as a conventional JSON schema change, as it acts on metadata rather than on data and, thus, could be better effected using the high-level JSON schema change operation RenameProperty, acting on the conventional JSON schema, we previously defined in [5], which is automatically propagated to extant conventional JSON data. However, as part of  $\tau$ JUpdate, we can also consider it a JSON data update that propagates indeed to the JSON schema by means of the implicit JSON schema change mechanism that we have proposed in [4]. The global effects in the  $\tau$ JSchema framework, anyway, are exactly the same. Such updates can be performed via the following  $\tau$ JUpdate HLOs:

```

DELETE FROM deviceTJD.json
PATH $.devices[device.name="CameraABC"]
VALID from "2022-05-25";
ALTER DOCUMENT deviceTJD.json
OBJECT $.devices
RENAME MEMEBER devices TO products
VALID from "2022-05-25"

```

The result of this HLO sequence is the new temporal JSON document shown in Fig. 6 with the new slice shown in Fig. 7.

```

{ "temporalJSONDocument":{
  "temporalJSONSchema":{
    "conventionalJSONSchema":{
      "sliceSequence":[
        {"slice":{
          "location":"deviceCJS1.json",
          "begin":"2022-02-01" } },
        {"slice":{
          "location":"deviceCJS2.json",
          "begin":"2022-05-25" } }
      ] } },
  "conventionalJSONDocument":{
    "sliceSequence":[
      {"slice":{
        "location":"device1.json",
        "begin":"2022-02-01" } },
      {"slice":{
        "location":"device2.json",
        "begin":"2022-04-15" } },

```

```

    {"slice":{
      "location":"device3.json",
      "begin":"2022-05-25" } } ] }
  } }

```

**Fig. 6.** The new temporal JSON document representing the whole history of the device repository (file deviceTJD.json).

```

{ "products":[
  { "device":{
    "name":"CameraXYZ",
    "costPrice":42 } } ] }

```

**Fig. 7.** The final state of the device repository (file device3.json).

As a side effect of the RenameMember HLO requiring an implicit JSON schema change, two conventional JSON schema versions are included in the new temporal JSON document (without entering into the whole details, deviceCJS1.json is the conventional JSON schema version having "devices" as its root object, whereas deviceCJS2.json is the conventional JSON schema version having "products" as its root object). As a consequence, squashing of the temporal JSON document in Fig. 6 produces two squashed JSON documents: deviceSJD1.json shown in Fig. 8, which is conformant to the first conventional JSON schema version deviceCJS1.json, and deviceSJD2.json shown in Fig. 9, which is conformant to the second conventional JSON schema version deviceCJS2.json. Changes are evidenced with red color.

```

{ "devices":[
  { "device":[
    { "name":"CameraABC",
      "costPrice":35,
      "VTbegin":"2022-02-01",
      "VTend":"2022-04-14" },
    { "name":"CameraABC",
      "costPrice":37.8,
      "VTbegin":"2022-04-15",
      "VTend":"2022-05-24" } ] },
  { "device":[
    { "name":"CameraXYZ",
      "costPrice":42,
      "VTbegin":"2022-04-15",
      "VTend":"2022-05-24" } ] } ] }

```

**Fig. 8.** The squashed JSON document (file deviceSJD1.json) corresponding to the first conventional JSON schema version deviceCJS1.json.

```

{ "products":[
  { "device":[
    { "name":"CameraXYZ",
      "costPrice":42,
      "VTbegin":"2022-05-25",
      "VTend":"Forever" } ] } ] }

```

**Fig. 9.** The squashed JSON document (file deviceSJD2.json) corresponding to the second conventional JSON schema version deviceCJS2.json.

## 5 Conclusion

In this paper, we have proposed  $\tau$ JUpdate, a temporal extension of the JUpdate language by equipping JUpdate update HLOs with a VALID clause to specify the applicability period of the update operations, in the  $\tau$ JSchema framework. Ignoring the VALID clause, any  $\tau$ JUpdate HLO is exactly the same as the corresponding JUpdate HLO to be executed in a non-temporal environment. Indeed, by taking advantage of the  $\tau$ JSchema logical and physical independence feature, our goal was to help the users by allowing them to focus only on the data structure as defined in the conventional JSON schema, and ignore how data are structured in the temporal JSON schema. Hence, implementation details and their transparent management are left to the system. Moreover, any  $\tau$ JUpdate HLO could be specified either on the temporal JSON document or on its squashed version; the system is able to correctly manage both ways, via the use of temporal (logical and physical) characteristics. We have also shown in Sec. 3.1 how the  $\tau$ JUpdate semantics can be defined to correctly deal with temporal JSON documents physically stored according to both forms (i.e., unsquashed and squashed forms).

Moreover, since JSON databases [16] are document-oriented NoSQL databases [10, 25], which are in general schemaless, a JSON instance document could be, at the end of an update operation, not conformant to its initial JSON schema. To cover this aspect, we have also dealt with JSON data updates that require implicit JSON schema changes (exemplified with the RenameMember HLO). Hence, in such a situation,  $\tau$ JUpdate executes implicit changes to conventional JSON schema, in a way transparent to the user, before performing temporal updates on conventional JSON data.

In the future, we envisage to extend  $\tau$ JUpdate to also support updating transaction-time and bitemporal JSON data, in  $\tau$ JSchema, as in the present work we have dealt only with valid-time JSON data. Finally, we plan to develop a tool that supports  $\tau$ JUpdate, in order to show the feasibility of our proposal and to use it in the experimental evaluation of our language (e.g., involving usability, user-friendliness and performance).

## References

1. Bourhis, P., Reutter, J., Vrgoč, D.: JSON: data model and query languages. *Information Systems* **89**, 101478 (2020)
2. Brahmia, S., Brahmia, Z., Grandi, F., Bouaziz, R.:  $\tau$ 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, 5-8 September 2016, Part 2. pp. 167–181 (2016)
3. Brahmia, S., Brahmia, Z., Grandi, F., Bouaziz, R.: A disciplined approach to temporal evolution and versioning support in JSON data stores. In: *Emerging Technologies and Applications in Data Processing and Management*, pp. 114–133. IGI Global (2019)

4. Brahmia, Z., Brahmia, S., Grandi, F., Bouaziz, R.: Implicit JSON schema versioning driven by big data evolution in the  $\tau$ JSchema framework. In: Proceedings of the International Conference on Big Data and Networks Technologies (BDNT 2019), Lecture Notes in Networks and Systems, Vol. 81. pp. 23–35 (2020)
5. Brahmia, Z., Brahmia, S., Grandi, F., Bouaziz, R.: Versioning schemas of JSON-based conventional and temporal big data through high-level operations in the  $\tau$ JSchema framework. *International Journal of Cloud Computing* **10**(5-6), 442–479 (2021)
6. Brahmia, Z., Brahmia, S., Grandi, F., Bouaziz, R.: JUpdate: A JSON update language. *Electronics* **11**(4), 508 (2022)
7. Brahmia, Z., Grandi, F., Bouaziz, R.:  $\tau$ XUF: a temporal extension of the XQuery update facility language for the  $\tau$ XSchema framework. In: Proc. of the 23rd International Symposium on Temporal Representation and Reasoning (TIME 2016), Technical University of Denmark, Copenhagen, Denmark, 17-19 October 2016. pp. 140–148 (2016)
8. Burns, T., Fong, E., Jefferson, D., Knox, R., Mark, L., Reedy, C., Reich, L., Roussopoulos, N., Truszkowski, W.: Reference model for DBMS standardization, database architecture framework task group (DAFTG) of the ANSI/X3/SPARC database system study group. *SIGMOD Record* **15**(1), 19–58 (1986)
9. Currim, F., Currim, S., Dyreson, C., Snodgrass, R.: A tale of two schemas: Creating a temporal XML schema from a snapshot schema with  $\tau$ XSchema. In: Proceedings of the International Conference on Extending Data Base Technology (EDBT 2004), Crete, Greece, 14-18 March 2004. pp. 348–365 (2004)
10. Davoudian, A., Chen, L., Liu, M.: A survey on NoSQL stores. *ACM Computing Surveys (CSUR)* **51**(2), 1–43 (2018)
11. Goyal, A., Dyreson, C.: Temporal JSON. In: 2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC 2019). pp. 135–144 (2019)
12. Grandi, F.: Temporal databases. In: *Encyclopedia of Information Science and Technology*, Third Edition, pp. 1914–1922. IGI Global (2015)
13. Hu, Z., Yan, L.: Modeling temporal information with JSON. In: *Emerging Technologies and Applications in Data Processing and Management*, pp. 134–153. IGI Global (2019)
14. Internet Engineering Task Force: JSON Schema: A Media Type for Describing JSON Documents, Internet-Draft, 19 March 2018. <https://json-schema.org/latest/json-schema-core.html>
15. Internet Engineering Task Force: The JavaScript Object Notation (JSON) Data Interchange Format, Internet Standards Track document, December 2017. <https://tools.ietf.org/html/rfc8259>
16. Irshad, L., Ma, Z., Yan, L.: A survey on JSON data stores. In: *Emerging Technologies and Applications in Data Processing and Management*, pp. 45–69. IGI Global (2019)
17. Irshad, L., Yan, L., Ma, Z.: Schema-based JSON data stores in relational databases. *Journal of Database Management (JDM)* **30**(3), 38–70 (2019)
18. ISO/IEC, Information technology Database languages SQL Technical Reports – Part 6: SQL support for JavaScript Object Notation (JSON), 1st Edition, Technical Report ISO/IEC TR 19075-6:2017(E), March 2017. [http://standards.iso.org/itf/PubliclyAvailableStandards/c067367\\_ISO\\_IEC\\_TR\\_19075-6\\_2017.zip](http://standards.iso.org/itf/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip)
19. Jensen, C., Snodgrass, R.: Temporal database. In: *Encyclopedia of Database Systems*, Second Edition, pp. 3945–3949. Springer (2018)

20. Liu, Z.: JSON data management in RDBMS. In: *Emerging Technologies and Applications in Data Processing and Management*, pp. 20–44. IGI Global (2019)
21. Liu, Z., Hammerschmidt, B., McMahon, D.: JSON data management: supporting schema-less development in RDBMS. In: *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014)*, Snowbird, UT, USA, 22-27 June 2014. pp. 1247–1258 (2014)
22. Lv, T., Yan, P., Yuan, H., He, W.: Linked lists storage for JSON data. In: *2021 International Conference on Intelligent Computing, Automation and Applications (ICAA 2021)*. pp. 402–405 (2021)
23. Melton, J., Zemke, F., Hammerschmidt, B., Kulkarni, K., Liu, Z., Michels, J., McMahon, D., Özcan, F., Pirahesh, H.: SQL/JSON part 1, DM32.2-2014-00024R1, 6 March 2014. [https://www.wiscorp.com/pub/DM32.2-2014-00024R1\\_JSON-SQL-Proposal-1.pdf](https://www.wiscorp.com/pub/DM32.2-2014-00024R1_JSON-SQL-Proposal-1.pdf)
24. Michels, J., Hare, K., Kulkarni, K., Zuzarte, C., Liu, Z., Hammerschmidt, B., Zemke, F.: The new and improved SQL: 2016 standard. *ACM SIGMOD Record* **47**(2), 51–60 (2018)
25. NoSQL Databases List by Hosting Data Updated 2020. <https://hostingdata.co.uk/nosql-database/>
26. Petković, D.: SQL/JSON standard: Properties and deficiencies. *Datenbank-Spektrum* **17**(3), 277–287 (2017)
27. Petković, D.: Implementation of JSON update framework in RDBMSs. *International Journal of Computer Applications* **177**, 35–39 (2020)
28. Snodgrass, R. T. (ed.), Ahn, I., Ariav, G., Batory, D., Clifford, J., Dyreson, C., Elmasri, R., Grandi, F., Jensen, C., Käfer, W., Kline, N., Kulkarni, K., Cliff Leung, T., Lorentzos, N., Roddick, J., Segev, A., Soo, M., Sripada, S.: *The TSQL2 Temporal Query Language*. Kluwer Academic Publishing, New York (1995)
29. Zemke, F., Hammerschmidt, B., Kulkarni, K., Liu, Z., McMahon, D., Melton, J., Michels, J., Özcan, F., Pirahesh, H.: SQL/JSON part 2 Querying JSON, ANSI INCITS DM32.2-2014-00025r1, 4 March 2014. <https://www.wiscorp.com/pub/DM32.2-2014-00025r1-sql-json-part-2.pdf>