Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 184 (2021) 823–828

Procedia Computer Science

www.elsevier.com/locate/procedia

The 3rd International Workshop on Big Data and Business Intelligence (BDBI'2021)
March 23-26, 2021, Warsaw, Poland

A Graphical Conceptual Model for Conventional and Time-varying JSON Data

Zouhaier Brahmia[a,*], Fabio Grandi[b], Safa Brahmia[a], Rafik Bouaziz[a]

[a]*University of Sfax, Road of the Aerodrome - Km 4.5 - P.O. Box 1088, 3018 Sfax, Tunisia*
[b]*Università di Bologna, Viale Risorgimento 2, I-40136 Bologna, Italy*

Abstract

Today, although there is an increasing interest in temporal JSON instance documents, since they allow tracking data changes, recovering past data versions, and executing temporal queries, there is no support (data model, modelling language, method, or tool) for conceptual modelling of temporal JSON data. Moreover, even though there are some graphical editors to build JSON Schemata (like JSON Schema Editor of Altova), they do not provide any built-in support for modelling temporal aspects of JSON data. Therefore, designers of JSON-based NoSQL data stores are proceeding in an ad hoc manner when they have to model some temporal requirements. To fill this theoretical and practical gap, we propose in this paper a graphical conceptual model for time-varying JSON data, named Temporal JSON Conceptual Model (TempoJCM). To this purpose, first we define a graphical conceptual model for conventional (i.e., non temporal) JSON data, called JSON Conceptual Model (JCM), and then we extend it to support modelling of temporal aspects of JSON data. TempoJCM facilitates conceptual modelling of both conventional and temporal JSON data, in a graphical and user-friendly manner. An editor supporting TempoJCM is planned to become the user interface for temporal JSON schema design in the τJSchema framework.

© 2021 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.

Keywords: Conceptual modelling; Graphical conceptual data model; JSON; Temporal JSON; Temporal data model; NoSQL data stores

* Corresponding author. Tel.: +216-99 809 402; fax: +216-74 279 139.
E-mail address: zouhaier.brahmia@fsegs.rnu.tn

1877-0509 © 2021 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2021.03.102

1. Introduction

Nowadays, time-varying JSON instance documents [1,2] are being widely used in the development of applications that are backed by NoSQL databases [3], like IoT, blockchain, and social networks applications. They provide several advantages like keeping a complete data history, tracking data changes, recovering past data versions, and executing temporal queries. However, the state of the art does not provide any support for conceptual modelling of temporal JSON data: any conceptual data model (like TEER [4]), modelling language (like [5]), modelling method (like TUML [6]), or modelling tool (like [7]), has been proposed to allow a JSON-based NoSQL database designer or administrator (NSDBA) to conceptually model such data.

Furthermore, although there are some commercial (like Altova XMLSpy's JSON Schema Editor and Generator [8], or Liquid Studio's JSON Schema Editor [9]) and free (like JSON Schema Editor [10], or Visual JSON Editor [11]) tools to help building JSON schemas in a textual or in graphical manner, they do not provide any built-in support to model temporal aspects [12] of JSON data, e.g., transaction time, valid time. It is worth mentioning that the JSON Schema language (its last version can be found in [13]) does not provide built-in temporal data types like "date" or "time"; only the seven following data types are provided: "null", "boolean", "object", "array", "number", "string", and "integer". Instead, it considers temporal data types as "formats" of "string" data and, therefore, it provides four defined formats to that end: "date", "time", "date-time", and "duration". For example, to define a "birthdate" property, we should define it as of "string" data type, with a "date" format, as shown below. Moreover, we could also specify a pattern (i.e., a regular expression) to which the value of the birthdate must be valid.

```
"birthdate": { "type": "string", "format": "date", "pattern": "\d\d\d\d-\d\d-\d\d" }
```

Thus, designers or administrators of JSON-based NoSQL data stores are proceeding in an ad hoc manner when they have to satisfy some temporal requirements and to model temporal aspects of JSON data. To fill this gap at both theoretical and practical levels, we propose in this paper a graphical conceptual model for time-varying JSON data, named Temporal JSON Conceptual Model (TempoJCM). To this purpose, first we introduce a graphical conceptual model for conventional (i.e., non temporal) JSON data, called JSON Conceptual Model (JCM), and then we extend it to support modelling of temporal aspects of JSON data by allowing the NSDBA to mark any component in the model (i.e., an object, an object member, an array, or an array element) as evolving along transaction time, valid time, or both. So, TempoJCM allows conceptual modelling of both conventional and temporal JSON data, in a graphical and user-friendly manner. Further, to the best of our knowledge, it is the first graphical conceptual data model for temporal JSON NoSQL databases.

A graphical editor supporting TempoJCM is also planned to become a user-friendly CASE tool for temporal JSON schema design in the τJSchema (Temporal JSON Schema) framework [1].

The rest of the paper is organized as follows. Section 2 provides JCM, our (snapshot) graphical model for conceptual modelling of conventional JSON data, and illustrates its use through an example. Section 3 proposes TempoJCM, our temporal conceptual JSON data model defined as a temporal extension of JCM, and illustrates its use through an example. Section 4 provides a summary of the paper and some remarks about our future work.

2. JCM: A Snapshot Graphical Model for Conceptual Modelling of JSON Data

In this section, we propose our graphical conceptual model for JSON data, called JCM. It allows NSDBAs to graphically build a JSON Schema document for a JSON document-oriented NoSQL database, similarly to the use of the ER model or the UML class diagram for conceptual modelling of a relational database schema or an object-oriented database schema, respectively. We are currently developing a JCM graphical editor to be integrated in our τJSchema framework [1].

Based on the specification of the JSON format [14] and the JSON Schema language [13], we have defined a minimal and complete set of six primitives for the creation of JCM schema components: Object, Array, Member(string), Value(type), Combinator(keyword), and Subschema.

- **Object**

In place of a null-type value x, it draws an empty rectangular box (as shown in Fig. 1) that represents an empty JSON object.

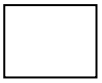

Fig. 1. The graphical representation of an empty JSON object.

- **Array**

In place of a null-type value x, it draws: [x], where the edges of the square brackets are joined with two horizontal dashed lines (as shown in Fig. 2), to represent an empty JSON array.

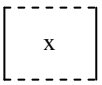

Fig. 2. The graphical representation of an empty JSON array.

Notice that an array of objects, as shown by Fig. 3, is obtained by applying the Object primitive to the empty value x inside the array.

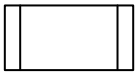

Fig. 3. The result of the application of the Object primitive to the empty value x inside the array of Fig. 2.

- **Member(string)**

Inside an object box, it draws: string——————x, where the line goes out of the box (as shown in Fig. 4), to represent an object member with a null-type value x.

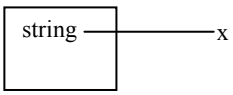

Fig. 4. The graphical representation of an "object member" of null type.

- **Value(type)**

In place of a null-type value x, it draws: o (type), where type is string, number, boolean, or date, to represent either the value of an object member or an array element, which is of simple and not null type. Fig. 5 shows two examples of application of this primitive.

Notice that although the "date" is not provided by JSON Schema [13] as a built-in data type (but as a format of the "string" type), we add it to our model since such a type is widely used (by designers) at conceptual level (and also to represent timestamps at physical level).

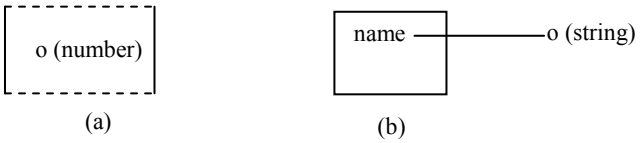

Fig. 5. The result of the application of (a) Value(number) on an empty array, and (b) a Value(string) on an object member called "name".

- **Combinator(keyword)**

In place of a null-type value x, it draws: a triangle with label keyword and a single null subschema, where keyword is "allOf", "anyOf", "oneOf", or "not", to represent a combination of schemas.

Fig. 6 shows an example of application of the "Combinator(anyOf)" primitive on an object member.

Notice that since the "keyword" stands for a boolean combination, we propose, as alternative notation, that the NSDBA can also use logical operator names (or symbols) inside the combinator triangle: AND (or $\wedge$) for "allOf", OR (or $\vee$) for "anyOf", XOR (or $\oplus$) for "oneOf", and NOT (or $\neg$) for "not".

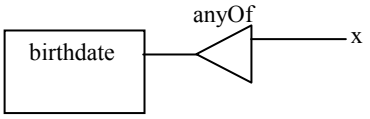

Fig. 6. The effect of the application of the Combinator(anyOf) primitive on the null-type value x of an object member named "birthdate".

- **Subschema**

It adds a single null subschema to an existing schema combinator. For example, if the designer wants to define a "birthdate" object member whose value could be of date type or of string type, he/she starts by applying the "Combinator(anyOf)" primitive on the null-type value x of "birthdate", as shown in Fig. 6. Then, he/she should apply Subschema on the "anyOf" combinator. After that, he/she should apply Value(date) on one of the two null-type values x and Value(string) on the other.

Moreover, we propose six deletion primitives as the inverse operations of those presented above: DelObject, DelArray, DelMember, DelValue, DelCombinator, and DelSubschema.

Notice also that our primitives (unless they are used in a graphical environment) have JSONPath [15] arguments to denote the schema components to which they have to be applied.

3. TempoJCM: A Temporal Extension of JCM

Since our proposal is a temporal data model, we start this section by briefly recalling some temporal database concepts that we consider necessary for understanding the reminder of this paper. In fact, a temporal database is a database that provides built-in support for managing time-varying data [12]. Two main temporal dimensions have been proposed for timestamping time-varying data: transaction time [16], which denotes when some datum is current in the database, and valid time [17], which denotes when some datum is valid in the modelled reality. Therefore, we talk about transaction-time databases, which support only transaction time, and valid-time databases, which support only valid time. As for databases that support both transaction time and valid time, they are called bitemporal databases. Temporal data can be of type state (i.e., with a continuous persistence over an interval) or event (i.e., with an instantaneous occurrence). Besides, a temporal data model [18] is a data model for representing time-varying data (e.g., BCDM, XBiT).

In the rest of this section, we propose TempoJCM, a temporal conceptual data model defined as a temporal extension of our JCM model presented in the previous section. The temporal extension consists in enhancing JCM to support the modelling of the transaction and/or valid time aspects [12] of JSON components (in the rest of this paper, we use the term "JSON component" to denote a JSON object, a JSON array, an object member, or an array element).. Hence, for specifying temporal aspects of a JSON component in a TempoJCM model, we introduce a primitive operation TemporalFormat(format), where the format argument can be TT (for transaction time), VT (for valid time) or BT (for bitemporal).

- TemporalFormat(TT): the declaration of a JSON component as of transaction-time format means that each version of this latter will have a distinct transaction-time timestamp (i.e., a transaction-time interval or a transaction-time point) automatically assigned by the system.
  Graphically, the transaction time format is represented via a circle with "TT" written inside, to be placed near the corresponding JSON component: (TT).

- TemporalFormat(VT): a JSON component that is declared as valid-time means that each version of this latter will have a distinct valid-time timestamp (as a temporal interval or a temporal point) provided by the user.
  The graphical representation of the valid time format is a circle with "VT" written inside, to be placed near the corresponding JSON component: (VT).

- TemporalFormat(BT): when a JSON component is declared as of bitemporal format this means that it is evolving along both transaction time and valid time. Graphically, the bitemporal format is represented via a circle with

"BT" written inside, to be put near the corresponding JSON component: (BT).

Besides, it is worth mentioning that timestamping is inherited by nested components. For example, when an object is declared to be in transaction time format, each one of its object members is automatically considered to be in transaction-time format and, if some of its members are also declared to be in valid-time format, such members acquire therefore a bitemporal format.

We would also remark that a TempoJCM schema is actually a sort of meta-schema, which can be translated into a plain JSON Schema [13] code, by expliciting the versioning structures and timestamp value representations.

3.1. Illustrative Example

To illustrate the use of our TempoJCM model, let us consider the example of an application for research laboratory management, using time-varying JSON data modelled as shown in Fig. 7. We assume that the NSDBA has defined first a JCM model in which the researchers are modelled as an array of objects. Each researcher (defined as an object) has a name (string), a birthdate (date), an address (string), a phone (string), a salary (number), a department (an object characterized by a name (string), and a director (string)), a list of research interests (an array of strings), a list of journal articles (an array of objects each one is characterized by a list of authors (an array of strings), a title (string), a journal (string), a volume (integer), an issue (an integer, or a string like "5-6", combined via a schema combinator "anyOf" (∨)), the start and end pages (string), the number of citations (integer)), and a ResearchGate score (number).

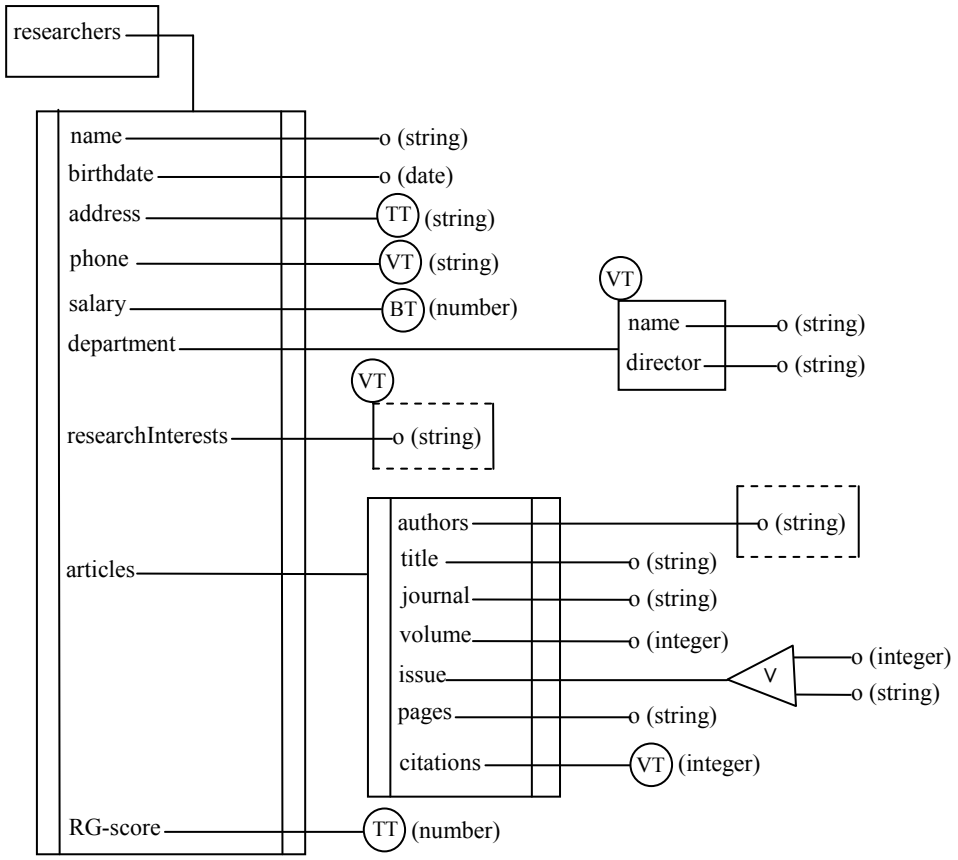

Fig. 7. A TempoJCM model for research laboratory management.

Then, we assume that, in order to satisfy a set of temporal requirements coming from the users who exploit the application, the NSDBA has annotated the above described JCM model with some temporal aspects to obtain a TempoJCM model. In fact, he/she has declared (i) transaction-time the address and the ResearchGate score of a researcher, (ii) bitemporal his/her salary, and (iii) valid-time his/her phone number, department, and research interests as well as the citations of each one of his/her articles. Fig. 7 shows the resulting TempoJCM diagram.

4. Conclusion and Future Work

In this paper, we have proposed TempoJCM, a graphical conceptual data model for temporal JSON data. It has been defined as a temporal extension of JCM, i.e., another graphical conceptual model that we have also proposed for conventional (i.e., non temporal) JSON data. TempoJCM allows NSDBA to build conceptual models for time-varying JSON data, in a graphical and a user-friendly manner. Moreover, it is conceived to also allow modelling conventional JSON data (i.e., it is backward compatible with JCM). The temporal aspects of data, which are supported by TempoJCM, are the transaction time and/or the valid time of a JSON component. In sum, and to the best of our knowledge, TempoJCM is the first graphical conceptual model for temporal JSON data.

To show the feasibility of our proposal, we are developing a graphical prototype editor, named TempoJCM-Editor, which supports TempoJCM (and, consequently, JCM). Such an editor will be integrated in our τJSchema framework [1] to produce as output conventional JSON schema and temporal characteristics documents, to be used to manage the modelled temporal JSON documents in this framework: the plain JCM editor commands are used to produce the conventional JSON schema, and the TempoJCM timestamping commands are then used to produce the temporal characteristics documents (see [1] for details). More precisely, TempoJCM-Editor is aimed at becoming the graphical design interface of temporal JSON documents in τJSchema. On the other hand, such an editor will help us to test the usability and the efficiency of our TempoJCM conceptual model.

References

[1] Brahmia, S., Brahmia, Z., Grandi, F., and Bouaziz, R. (2016). τJSchema: A Framework for Managing Temporal JSON-Based NoSQL Databases. In: *Proc. of DEXA'2016*, pp. 167-181.

[2] Goyal, A., and Dyreson, C. (2019). Temporal JSON. In: *Proc. of the IEEE CIC'2019*, pp. 135-144.

[3] Davoudian, A., Chen, L., and Liu, M. (2018). A Survey on NoSQL Stores. *ACM Computing Surveys* **51**(**2**): Article 40.

[4] Elmasri, R., and Wuu, G.T.J. (1990). A temporal model and query language for ER databases. In: *Proc. of ICDE'1990*, pp. 76-83.

[5] Cabot, J., Olivé, A., and Teniente, E. (2003). Representing temporal information in UML. In: *Proc. of UML'2003*, pp. 44-59.

[6] Svinterikou, M., and Theodoulidis, B. (1999). TUML: A method for modelling temporal information systems. In: *Proc. of CAiSE'1999*, pp. 456-461.

[7] Detienne, V., and Hainaut, J.L. (2001). CASE tool support for temporal database design. In: *Proc. of ER'2001*, pp. 208-224.

[8] Altova XMLSpy's JSON Schema Editor and Generator. https://www.altova.com/xmlspy-xml-editor/json_schema_editor (accessed: 2021-01-11)

[9] Liquid Studio's JSON Schema Editor. https://www.liquid-technologies.com/json-schema-editor (accessed: 2021-01-11)

[10] JSON Schema Editor. https://dashjoin.github.io/#/schema (accessed: 2021-01-11)

[11] Visual JSON Editor. https://github.com/RicoSuter/VisualJsonEditor (accessed: 2021-01-11)

[12] Grandi, F. (2015). Temporal Databases. In: Khosrow-Pour, M. (Ed.) *Encyclopedia of Information Science and Technology (3rd edition)*, pp. 1914-1922. IGI Global, Hershey, PA, USA

[13] IETF. (2018). JSON Schema: A Media Type for Describing JSON Documents. Internet-Draft, 19 March 2018. https://json-schema.org/latest/json-schema-core.html (accessed: 2021-01-11)

[14] IETF. (2017). The JavaScript Object Notation (JSON) Data Interchange Format. Internet Standards Track document, December 2017. https://tools.ietf.org/html/rfc8259 (accessed: 2021-01-11)

[15] Gössner, S. (2007). JSONPath – XPath for JSON, 21 February 2007. http://goessner.net/articles/JsonPath/ (accessed: 2021-01-11)

[16] Jensen, C.S., and Snodgrass, R.T. (2018). Transaction Time. In: Liu, L., and Özsu, M.T. (Eds.) *Encyclopedia of Database Systems (2nd edition)*, doi: 10.1007/978-1-4614-8265-9_1064. Springer-Verlag, New York, USA

[17] Jensen, C.S., and Snodgrass, R.T. (2018). Valid Time. In: Liu, L., and Özsu, M.T. (Eds.) *Encyclopedia of Database Systems (2nd edition)*, doi: 10.1007/978-1-4614-8265-9_1066. Springer-Verlag, New York, USA

[18] Jensen, C.S., and Snodgrass, R.T. (2018). Temporal Data Models. In: Liu, L., and Özsu, M.T. (Eds.) *Encyclopedia of Database Systems (2nd edition)*, doi: 10.1007/978-1-4614-8265-9_394. Springer-Verlag, New York, USA