Article

JUpdate: A JSON Update Language

Zouhaier Brahmia [1], Safa Brahmia [1], Fabio Grandi [2,*] and Rafik Bouaziz [1]

1 Department of Computer Science of the Faculty of Economics & Management, University of Sfax, Sfax 3018, Tunisia; zouhaier.brahmia@fsegs.usf.tn (Z.B.); safa.brahmia@gmail.com (S.B.); rafik.bouaziz@usf.tn (R.B.)
2 Department of Computer Science and Engineering (DISI), University of Bologna, 40136 Bologna, Italy
* Correspondence: fabio.grandi@unibo.it (F.G.); Tel.: +39-051-20-9-3555

Abstract: Although JSON documents are being used in several emerging applications (e.g., Big Data applications, IoT, mobile computing, smart cities, and online social networks), there is no consensual or standard language for updating JSON documents (i.e., creating, deleting or changing such documents, where changing means inserting, deleting, replacing, copying, moving, etc., portions of data in such documents). To fill this gap, we propose in this paper an SQL-like language, named JUpdate, for updating JSON documents. JUpdate is based on a set of six primitive update operations, which is proven complete and minimal, and it provides a set of fourteen user-friendly high-level operations with a well-founded semantics defined on the basis of the primitive update operations.

Keywords: JSON; JSON document; JSON update; primitive update operation; high-level update operation; completeness

Citation: Brahmia, Z.; Brahmia, S.; Grandi, F.; Bouaziz, R. JUpdate: A JSON Update Language. *Electronics* **2022**, *11*, 508. https://doi.org/10.3390/electronics11040508

Academic Editors: Amin Karami, Fahimeh Jafari and Manel Guerrero-Zapata

Received: 1 January 2022
Accepted: 4 February 2022
Published: 9 February 2022

1. Introduction

Nowadays, the JavaScript Object Notation (JSON) data format [1], which is endorsed by the Internet Engineering Task Force (IETF), is widely used in the context of several emerging applications, such as Big Data applications, Internet of Things (IoT), mobile computing, streaming applications, smart cities, and online social networks, since it is a lightweight serialization data format that makes storage, interchange, and querying of data easy. Such application environments are highly dynamic by nature and, thus, require a continuous update of the managed data repositories. To fulfill this requirement, application developers need to rely on a suitable low-level proprietary Application Programming Interface (API) exposed by the data management system or on the existence and support of a high-level data update language. The latter only would also bring about desirable properties of reusability, portability, and interoperability and, thus, is the option of choice from a software engineering standpoint.

However, by studying the state-of-the-art of JSON data management (e.g., [2–12]), we have not found a recommended, consensual, and/or standard language for updating (native) JSON data (such as SQL for relational data, XQuery Update Facility for XML data, or SPARQL 1.1 Update for RDF data). Several JSON query languages have been proposed, such as Jaql [13], JSONiq [14], JPQ [15], SQL++ [16,17], JNL [18], JSON-λ [19], JSONata, GROQ, and JMESPath, which all lack modification statements making up a JSON update language. Notice that with the locution "JSON update language", we mean a language that allows users to insert, delete, and modify valid data portions in a JSON document; hence, the term "update" is used in this work in such a broader sense.

After having surveyed the state-of-the-practice of JSON data management, we have concluded that JSON-based NoSQL DBMSs [20] can be divided into two categories: (i) those that provide their own (proprietary) query languages, such as the MongoDB language and several languages that are SQL-like, e.g., N1QL of Couchbase, CQL of Cassandra, and AQL of AsterixDB, and (ii) those that only provide low-level programmatic APIs for accessing, querying, and updating JSON data, such as Redis and Amazon Dynamo. Moreover, an extension of SQL, named SQL/JSON [21–23], has been standardized by ANSI to allow the SQL language to also support queries on JSON data. However, according to the discussion of the SQL/JSON standard presented in [24], this new language does not support the modification of portions of a JSON document through the SQL UPDATE statement.

In this context, it is our conviction that a JSON data update requires more attention and that the definition of a JSON update language is necessary. Moreover, such a language should be powerful and user-friendly but also defined with a well-founded and DBMS-independent semantics in order to facilitate its use and implementation. For these reasons, we propose in this paper a language, named JUpdate, for updating JSON data. JUpdate is composed of a set of fourteen user-friendly high-level operations with a SQL-like syntax for the insertion, deletion, and various modifications of valid portions of JSON documents. Its operational semantics is based on the implementation of a set of low-level, primitive update operations, which will be also introduced in the paper. The completeness and the minimality of the proposed set of primitives, on which also the completeness and consistency of JUpdate is founded, will be proven. In sum, our contributions could be summarized as follows:

- We propose JUpdate, an SQL-like update language for JSON. The objective of defining JUpdate similarly to the SQL language is to facilitate the use/acceptance of this new language to users who are familiar with SQL and developers who have been used to write SQL code for decades.
- JUpdate provides fourteen user-friendly high-level operations to satisfy the different update requirements, either simple or complex, of JSON data users and JSON-based application developers.
- The semantics of JUpdate is founded on a minimal and complete set of six primitives (i.e., low-level operations) for updating JSON documents.
- The JUpdate definition has been based on the IETF standard JSON data model. Therefore, it is DBMS-independent, on one hand, and it can be used for the maintenance of generic JSON documents, on the other hand.

The rest of this paper is structured as follows. Section 2 presents the JUpdate language by introducing the fourteen update high-level operations (HLOs) it is composed of. Section 3 presents a short example that illustrates the use of JUpdate. Section 4 introduces the set of primitive update operations underlying the definition of JUpdate. Section 5 provides a completeness and minimality proof for this proposed set and proves the completeness of the JUpdate language. Section 6 deals with the definability of the JUpdate HLOs. Section 7 discusses related work. Section 8 summarizes our contributions and gives some remarks on our future research directions.

2. High-Level Update Operations on JSON Documents: The JUpdate Language

In this section, we present the set of powerful HLOs for updating JSON documents, which make up the JUpdate language. JUpdate is composed of fourteen HLOs with an SQL-like syntax, so that it could be considered, for instance, as a complement of SQL++ to make up a complete JSON data manipulation language (DML). An equivalent alternative specification of JUpdate is also provided as an API composed of fourteen procedure interfaces, where the names of the procedures correspond to the names of the JUpdate HLOs.

The proposed set of HLOs making up JUpdate includes fourteen operations: JUpdate = { `CreateDocument`, `DropDocument`, `InsertValue`, `DeleteValue`, `UpdateValue`, `CopyValue`, `MoveValue`, `InsertMember`, `DeleteMember`, `RenameMember`, `ReplaceMember`, `CopyMember`, `MoveMember`, `UpdateObject` }. We have organized them into four subsets of HLOs, each one of which is inspired by a subset of the SQL statements, as shown in Table 1.

Table 1. The four subsets of HLOs of JUpdate and their inspiration by the SQL statements.

HLOs Subset	Objective
`CreateDocument`, `DropDocument`	They are used to manage JSON documents as a whole, with a syntax based on the SQL`CREATE`/`DROPTABLE` statements.
`InsertValue`, `DeleteValue`, `UpdateValue`, `CopyValue`, `MoveValue`	They are used to manage (possibly complex) values in JSON documents, with a syntax based on the SQL`INSERT`/`DELETE`/`UPDATE` statements.
`InsertMember`, `DeleteMember`, `RenameMember`, `ReplaceMember`, `CopyMember`, `MoveMember`	They are used to manage the structure of JSON objects, with a syntax based on the `SQL ALTER TABLE` statement. Notice that JSON objects have to be used to represent (nested) relational tables.
`UpdateObject`	It is used to change at once the value of more than one members of a JSON object, with a syntax based on the SQL `UPDATE` statement using the `SET` clause.

Path expressions based on JSONPath [25] are used in JUpdate statements to locate the target position for updates in JSON documents. Moreover, filters embedded in JSONPath expressions are used as selection conditions for document portions to be modified, without the need of a `WHERE` clause in JUpdate statements. The JUpdate operations will be individually presented in the following, whereas a full syntax listing of the JUpdate language is provided in Appendix A. Then, we will show in Section 6 how each one of these operations can be defined (and, thus, also implemented) on the basis of a complete and minimal set of primitive update operations.

2.1. *CreateDocument*

The `CreateDocument` HLO is defined by the statement:

`CREATE DOCUMENT` *doc*
`VALUE` *value*

and can be used to create a new JSON instance document named "doc", using "value" as its initial contents. The corresponding API procedure is `CreateDocument(doc, value)`.

An error exception is raised if the JSON instance document "doc" already exists or if "value" is not a correct JSON value.

2.2. *DropDocument*

The `DropDocument` HLO is defined by the statement:

`DROP DOCUMENT` *doc*

and can be used to delete a JSON instance document named "doc", regardless of its contents. The corresponding API procedure is `DropDocument(doc)`.

An error exception is raised if the doc.json JSON instance document does not exist.

2.3. *InsertValue*

The `InsertValue` HLO is defined by the statement:

`INSERT INTO` *doc*
`PATH` *path* `VALUE` *value*

and is used to insert a possibly complex value "value" as an object member value or new array element to the JSON document "doc", where "path" denotes the point of insertion. The corresponding API procedure is `InsertValue(doc, path, value)`.

Since "path" (i.e., the point of insertion) is a JSONPath path, it can denote zero, one, or many JSON components, where a JSON component is an object, an object member, an array, or an array element.

If "path" has the form "p[index]", then the path prefix "p" must denote an array, and a "value" is inserted before the index-th element of the array (so that it becomes itself the index-th element after the update). If the literal "last" is used instead of the index (i.e., path has the form "p[last]"), then "value" is inserted as the last element of the array.

An error exception is raised if "value" is not a correct JSON value or if "path" does not denote a null object member or a valid array location (i.e., when path has the form "p[index]", where "p" does not denote an array or it denotes an array but the index is either a number greater than the array length or index is not the literal last).

2.4. *DeleteValue*

The `DeleteValue` HLO is defined by the statement:

`DELETE FROM` *doc*
`PATH` *path*

and is used to delete a possibly complex value designated by "path", which either can denote an object member or an array element location in the "doc" document.

If "path" denotes an object member, its value is set to "null" after the execution of the deletion operation. Notice that this operation preserves the structure of the object (i.e., although with a null value, the member denoted by "path" is still present in the object after its execution). The corresponding API procedure is `DeleteValue(doc, path)`.

An error exception is raised if "path" does not denote an object member or a valid array element location (i.e., when "path" has the form "p[index]", where "p" does not denote an array or it denotes an array but the index is greater than the array length).

2.5. *UpdateValue*

The `UpdateValue` HLO is defined by the statement:

`UPDATE` *doc*
`PATH` *path* `VALUE` *value*

and is used to replace the value denoted by "path" with a new (possibly complex) value "value" in the JSON document "doc". The corresponding API procedure is `UpdateValue(doc, path, value)`.

An error exception is raised if "value" is not a correct JSON value or if "path" does not denote an object member or a valid array element location (i.e., when "path" has the form "p[index]", where "p" does not denote an array or it denotes an array but the index is neither a number less than the array length nor is the literal last).

2.6. *CopyValue*

The `CopyValue` HLO is defined by the statement:

`UPDATE` *doc*
`COPY FROM` *path* `TO` *newpath*

and is used to copy the value designated by "path" to a new location denoted by "newpath" in the JSON document "doc". The corresponding API procedure is `CopyValue(doc, path, newpath)`.

An error exception is raised if "path" or "newpath" do not denote an object member or a valid array element location.

2.7. *MoveValue*

The `MoveValue` HLO is defined by the statement:

`UPDATE` *doc*
`MOVE FROM` *path* `TO` *newpath*

and is used to move the value designated by "path" to a new location denoted by "newpath" in the JSON document "doc". In fact, it is equivalent to the execution of the two statements:

`UPDATE` *doc*
`COPY FROM` *path* `TO` *newpath;*
`DELETE FROM` *doc*

```
PATH path.
```

The corresponding API procedure is `MoveValue(doc, path, newpath)`.

An error exception is raised if "path" or "newpath" do not denote an object member or a valid array element location or if "path" is a prefix of the "newpath" location (i.e., a value cannot be moved into itself).

Whereas the statements dealing with (possibly complex) values are modeled on the SQL DML `INSERT/DELETE/UPDATE` statements (as shown in the five HLOs above), the JUpdate statements acting on JSON objects are modeled on the SQL `ALTER` DDL statement (as shown in the six HLOs below), where members of an object in a JSON document are assimilated to columns in a relational table. In fact, modifications that involve members deal with the structure of objects and, thus, can be considered for all intents (implicit) schema changes [26,27].

Notice that the enforcement at the language level of the distinction between the management of JSON values and JSON objects is also a precise design choice that prevents from the "interchangeability" of arrays and objects to store data collections. JSON allows indeed for a sort of equivalence between an array `[‘‘foo’’, ‘‘bar’’]` and an object `{‘‘0’’:’’foo’’, ‘‘1’’:’’bar’’}` (deriving from the equivalence of dot and bracket notations in JSONPath), which we consider rather a severe flaw in its design and which should be fixed by disallowing numbers to be used as object member names. In fact, from a conceptual modeling perspective, arrays and objects are clearly different constructs with a different semantics and serving different data design purposes (e.g., such as Lists and Dictionaries used as data collections in Python [28]). In this respect, JSON is considered somewhat superior to XML as data modeling language precisely because of its native support of arrays. Moreover, admitting the above equivalence would make one of the two constructs actually redundant and the commonly used JSON syntax (see Section 5) in some sense ambiguous, as two different syntactic trees would correspond to equivalent data expressions.

2.8. InsertMember

The `InsertMember` HLO is defined by the statement:

```
ALTER DOCUMENT doc OBJECT path
ADD MEMBER name VALUE value
```

and is used to insert an object member with name "name" and a possibly complex value "value" into the JSON document "doc", where "path" denotes the point of insertion. The `VALUE` clause is optional; if omitted, a null value is assigned to the new member. The corresponding API procedure is `InsertMember(doc, path, name, value)`.

An error exception is raised if "path" does not denote an object.

Notice that there is no need to have in JUpdate also an `InsertArray` or `DeleteArray` HLO, since the `InsertValue` and `DeleteValue` HLO, respectively, can be used with an array-type value. For instance, the whole array [0,5] can be deleted from the JSON instance (1), as shown below:

```
doc.json =
{‘‘X’’:[{‘‘A’’:null,’’B’’:10},{},{‘‘A’’:true,’’Z’’:[0,5]},{‘‘C’’:’’xy’’}]}(1)
```

via the statement:

```
DELETE FROM doc.json
PATH $.X[2].Z
```

producing the JSON instance (2) which follows:

```
doc.json =
{‘‘X’’:[{‘‘A’’:null,’’B’’:10},{},{‘‘A’’:true,’’Z’’:null},{‘‘C’’:’’xy’’}]}.(2)
```

2.9. DeleteMember

The `DeleteMember` HLO is defined by the statement:

```
ALTER DOCUMENT doc OBJECT path
DROP MEMBER name
```

and is used to delete the member "name" from an object denoted by "path" (with a possibly complex value) from the JSON document "doc". The corresponding API procedure is `DeleteMember(doc, path.name)`, where "path.name" is the path denoting the member to be deleted.

An error exception is raised if "path.name" does not denote an object member.

2.10. RenameMember

The `RenameMember` HLO is defined by the statement:

> `ALTER DOCUMENT` *doc* `OBJECT` *path*
> `RENAME MEMBER` *name* `TO` *newname*

and is used to rename the member "name" of an object denoted by "path" to "newname" in the JSON document "doc". The corresponding API procedure is `RenameMember(doc, path.name, newname)`.

An error exception is raised if "path.name" does not denote an object member.

2.11. ReplaceMember

The `ReplaceMember` HLO is defined by the statement:

> `ALTER DOCUMENT` *doc* `OBJECT` *path*
> `REPLACE MEMBER` *name* `WITH` *newname* `VALUE` *value*

and is used to replace the name and the value of the object member denoted by "path.name" in the JSON document "doc" with "newname" and "value", respectively. The `VALUE` clause is optional; if omitted, a null value is assigned to the replacing member. The corresponding API procedure is `ReplaceMember(doc, path.name, newname, value).`

An error exception is raised if "path.name" does not denote an object member.

2.12. CopyMember

The `CopyMember` HLO is defined by the statement:

> `ALTER DOCUMENT` *doc* `OBJECT` *path*
> `COPY MEMBER` *name* `TO` *newpath*

and is used to copy the member "name" of an object denoted by "path" to a new location denoted by "newpath" in the JSON document "doc". The corresponding API procedure is `CopyMember(doc, path.name, newpath).`

An error exception is raised if "path.name" does not denote an object member or if "newpath" does not denote an object.

2.13. MoveMember

The `MoveMember` HLO is defined by the statement:

> `ALTER DOCUMENT` *doc* `OBJECT` *path*
> `MOVE MEMBER` *name* `TO` *newpath*

and is used to move the member "name" of an object denoted by "path" to a new location denoted by "newpath" in the JSON document "doc". In fact, it is equivalent to the execution of the two statements:

> `ALTER DOCUMENT` *doc* `OBJECT` *path*
> `COPY MEMBER` *name* `TO` *newpath;*
> `ALTER DOCUMENT` *doc* `OBJECT` *path*
> `DROP MEMBER` *name*.

The corresponding API procedure is `MoveMember(doc, path.name, newpath).`

An error exception is raised if "path.name" does not denote an object member or if "newpath" does not denote an object or if "path" is a prefix of the "newpath" location (i.e., an object member cannot be moved to a location inside its value).

2.14. UpdateObject

Finally, we add an `UpdateObject` HLO defined by the statement:

`UPDATE` *doc* `OBJECT` *path*
`SET` *name1=value1, name2=value2, . . . , nameN=valueN*

as an SQL-like operation for updating several members of an object at the same time. In fact, it can be used to replace with "value1", "value2", . . . , "valueN" the values of the members "name1", "name2", . . . , "nameN", respectively, of the JSON object denoted by "path" in the JSON document "doc". The corresponding API procedure is `UpdateObject(doc, path, name1, value1, name2, value2, . . . , nameN, valueN)`.

An error exception is raised if "value1", "value2", . . . , "valueN", are not correct JSON values or if "path" does not denote a JSON object.

Notice that the `UpdateObject` operation above is just *syntactic sugar* for the JUpdate statement sequence:

`UPDATE` *doc*
`PATH` *path.name1* `VALUE` *value1;*
`UPDATE` *doc*
`PATH` *path.name2* `VALUE` *value2;*
. . .
`UPDATE` *doc*
`PATH` *path.nameN* `VALUE` *valueN*.

Similarly, the `UpdateObject` API procedure can be considered as a macro defined as an equivalent sequence of `UpdateValue` procedure calls.

In all the statements listed above are equipped with a `VALUE` clause supporting a constant value, we assume it can also be replaced, as it happens with `INSERT` in SQL, by a subquery dynamically returning a computed JSON value. Such a subquery can be expressed via a JSONPath expression (possibly defined on a JSON document different from that under modification) or via a `SELECT` statement expressed in an SQL-like JSON query language such as SQL++. An error exception is raised if this subquery, specified as a JSONPath expression or as a SQL-like SELECT statement, returns a collection of values and not a single value. On the other hand, if the subquery result is empty, a "null" value is assumed.

3. Illustrative Example

To illustrate the use of our JUpdate language, we assume to have a JSON NoSQL database in use in a research laboratory, including a JSON instance document, named "scientificProduction.json", which stores information about the publications of the members of the laboratory. The initial state of such a document is displayed in Figure 1.

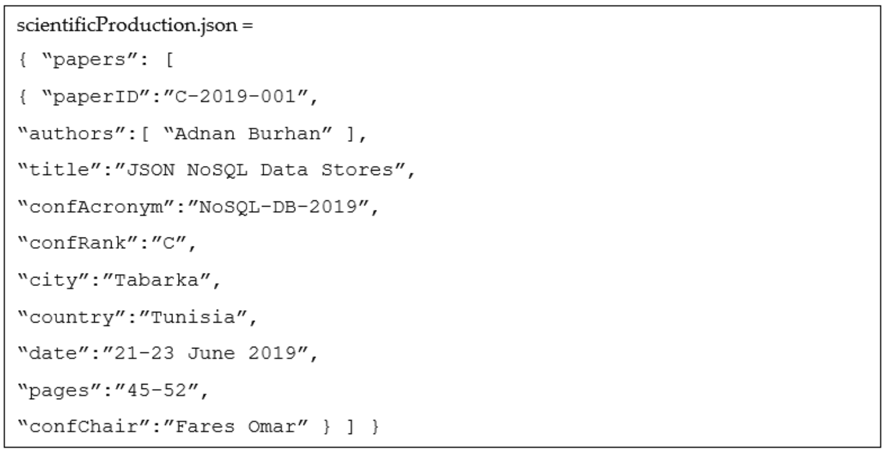

scientificProduction.json =
```
{ "papers": [
{ "paperID":"C-2019-001",
"authors":[ "Adnan Burhan" ],
"title":"JSON NoSQL Data Stores",
"confAcronym":"NoSQL-DB-2019",
"confRank":"C",
"city":"Tabarka",
"country":"Tunisia",
"date":"21-23 June 2019",
"pages":"45-52",
"confChair":"Fares Omar" } ] }
```

Figure 1. Initial state of the "`scientificProduction.json`" JSON instance file.

Suppose an end user of this database wants to add the information about a new journal paper (with its details: ID, title, journal name, volume, issue, year of publication, and pages). To this purpose, as papers are stored as objects in the array “papers” (as shown in Figure 1), he/she could use the `InsertValue` JUpdate operation as follows:

```
INSERT INTO scientificProduction.json
PATH $.papers[last]
VALUE { "paperID":"J-2019-001",
        "authors":["Layla Ahmad","Mario Rossi"],
        "title":"JSON Query Languages",
        "journalName":"Emerging Databases",
        "volume":5,
        "issue":1,
        "year":2019,
        "pages":null }.
```

The result of the execution of this operation is shown in Figure 2, where added parts are evidenced in red bold type.

```
scientificProduction.json =
{ “papers”:
[ { “paperID”:”C-2019-001”,
“authors”:[ “Adnan Burhan” ],
“title”:”JSON NoSQL Data Stores”,
“confAcronym”:”NoSQL-DB-2019”,
“confRank”:”C”,
“city”:”Tabarka”,
“country”:”Tunisia”,
“date”:”21-23 June 2019”,
“pages”:”45-52”,
“confChair”:”Fares Omar” },
{ “paperID”:”J-2019-001”,
“authors”:[ “Layla Ahmad”, “Mario Rossi” ],
“title”:”JSON Query Languages”,
“journalName”:”Emerging Databases”,
“volume”:5,
“issue”:1,
“year”:2019,
“pages”:null } ] }
```

Figure 2. State of the “`scientificProduction.json`” JSON file after the execution of the `InsertObject` HLO that adds the new (journal) paper.

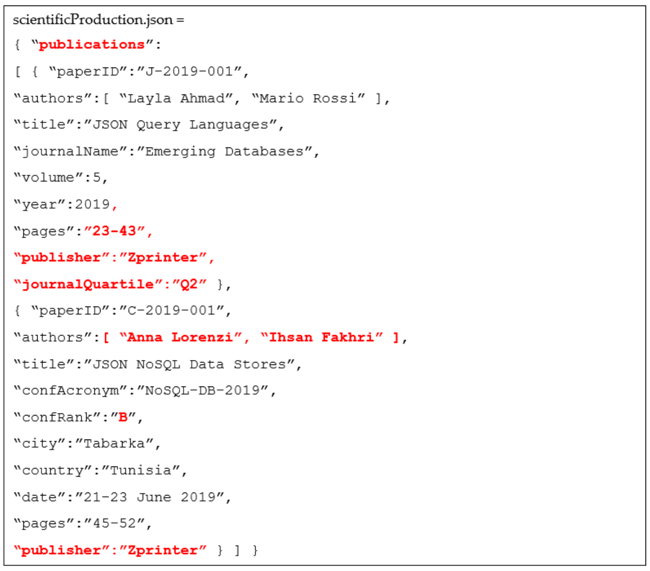
After that, suppose that the end user wants to perform the following tasks:

- To change the name of the array from “papers” to “publications”;
- To fix an error concerning the authors of the paper with ID “C-2019-001”. In fact, the user acknowledged that this paper has not been written by “Adnan Burhan”, but actually by “Anna Lorenzi” and “Ihsan Fakhri”. In order to apply the correction, the user wants to have the current wrong value of the “authors” array replaced by a new value including the two proper authors;
- To replace the “confChair” property of the conference having the acronym “NoSQL-DB-2019”, with the publisher “Zprinter” of the conference proceedings;
- To update from “C” to “B” the rank of the conference with acronym “NoSQL-DB-2019”;
- To add the missing page numbers “23–43” to the paper with ID “J-2019-001”;
- To add journal quartile information, with value “Q2”, to the paper with ID “J-2019-001”;
- To copy the publisher information from the paper with ID “C-2019-001” to the paper with ID “J-2019-001”, as the publisher is the same;
- To delete the issue property of the journal with name “Emerging Databases”, since it has no longer had issue numbers since its fifth volume;
- To move the paper with ID “J-2019-001” at the beginning of the list of papers.

Hence, to this aim, he/she can use the following sequence of JUpdate statements:

```
ALTER DOCUMENT scientificProduction.json OBJECT $
RENAME MEMBER papers TO publications ;
UPDATE scientificProduction.json
PATH $.publications[?(@.paperID=='C-2019-001')].authors
VALUE ["Anna Lorenzi","Ihsan Fakhri"] ;
ALTER DOCUMENT scientificProduction.json
OBJECT $.publications[?(@.confAcronym=='NoSQL-DB-2019')]
REPLACE MEMBER confChair WITH publisher VALUE "Zprinter" ;
UPDATE scientificProduction.json
OBJECT $.publications[?(@.confAcronym=='NoSQL-DB-2019')]
SET confRank="B" ;
INSERT scientificProduction.json
PATH $.publications[?(@.paperID=='J-2019-001')].pages
VALUE "23-43" ;
ALTER DOCUMENT scientificProduction.json
OBJECT $.publications[?(@.paperID=='C-2019-001')]
COPY MEMBER publisher
          TO $.publications[?(@.paperID=='J-2019-001')] ;
ALTER DOCUMENT scientificProduction.json
OBJECT $.publications[?(@.paperID=='J-2019-001')]
ADD MEMBER journalQuartile VALUE "Q2" ;
ALTER DOCUMENT scientificProduction.json
OBJECT $.publications[?(@.journalName=='Emerging Databases'),
                                            ?(@.volume>=5)]
DROP MEMBER issue ;
UPDATE scientificProduction.json
MOVE FROM $.publications[?(@.paperID=='J-2019-001')]
          TO $.publications[0].
```

Notice that for the update of an object member value, in order to exemplify both options, we used either the `UPDATE ... VALUE ...` form (i.e., the `UpdateValue` HLO) and the `UPDATE ... SET ...` syntactic variant (i.e., the `UpdateObject` HLO). The result of the execution of this sequence of operations is shown in Figure 3, where changes are evidenced in red bold type.



scientificProduction.json =

```
{ "publications":
[ { "paperID":"J-2019-001",
"authors":[ "Layla Ahmad", "Mario Rossi" ],
"title":"JSON Query Languages",
"journalName":"Emerging Databases",
"volume":5,
"year":2019,
"pages":"23-43",
"publisher":"Zprinter",
"journalQuartile":"Q2" },
{ "paperID":"C-2019-001",
"authors":[ "Anna Lorenzi", "Ihsan Fakhri" ],
"title":"JSON NoSQL Data Stores",
"confAcronym":"NoSQL-DB-2019",
"confRank":"B",
"city":"Tabarka",
"country":"Tunisia",
"date":"21-23 June 2019",
"pages":"45-52",
"publisher":"Zprinter" } ] }
```

Figure 3. Final state of the "`scientificProduction.json`" JSON instance document after the execution of the sequence of JUpdate operations described in the text.

4. Primitive Update Operations

In this section, we propose a set of six primitive update operations PO = { `CreaDoc`, `DropDoc`, `InsMemb`, `InsVal`, `DelMemb`, `DelVal` }, on which the definition and implementation of the JUpdate language can be based. As it will be shown in the next section (and precisely in Section 5.1), the proposed primitive operation set PO is complete and minimal, and it is given two arbitrary (and possibly empty) JSON instance documents doc1.json and doc2.json; doc1.json can always be transformed into doc2.json via the application of a suitable sequence of primitive operations taken from PO. The chosen names of the proposed operations are derived from corresponding JUpdate HLOs: whereas high-level operations have been assigned full names (e.g., `InsertValue`), the strictly related PO operations have shortened names (e.g., `InsVal`). The semantics of the PO operations is defined as described in the following.

4.1. CreaDoc

The `CreaDoc(doc)`primitive creates a JSON instance document, named "doc", initialized with a null content. An error exception is raised if an instance file named "doc" already exists.

4.2. DropDoc

The `DropDoc(doc)`primitive removes the JSON file named "doc" from the database with the constraint that the argument represents a JSON file with a null content. Any other contents must have been removed before. An error exception is raised if the instance file named "doc" does not exist or it does not have a null content.

4.3. InsMemb

The `InsMemb(doc, path, name)` primitive adds an initially empty (i.e., with a "null" value) member named "name" to an object (possibly empty) denoted by the path "path" of the JSON instance document "doc". An error exception is raised if "path" does not denote an object.

4.4. InsVal

The `InsVal(doc, path, name)` primitive assigns a value to an empty object member (i.e., whose value is "null") located at path "path" in the JSON instance document "doc". The supplied value "value" can be either a valid number, string, or Boolean (i.e., a true or false literal) value, or the literal "object" or "array". If "value" is "object" or "array", the corresponding structure is created initially empty (i.e., "{}" or "[]", resp.). If path has the form "p[index]", then the path prefix "p" must denote an array and "value" is inserted before the index-th element of the array (so that it becomes itself the index-th element after the update). If the literal "last" is used instead of index (i.e., the path has the form "p[last]"), then a value is inserted as the last element of the array. An error exception is raised if the path does not denote an empty object member or if the path has been specified as p[index] but p does not denote an array or denotes an array with fewer index components.

The `InsVal` primitive is also used to initialize an empty document (i.e., with a "null" content) with a non-null value (e.g., "object" or "array").

After having been created empty with the `CreaDoc` operation, any complex JSON file can be built with a suitable sequence of `InsMemb` and `InsVal` operations.

4.5. DelVal

The `DelVal(doc, path)` primitive removes from the JSON instance document "doc" the value denoted by "path", which must be a simple-type (i.e., string, number, or Boolean) value or an empty object or array. An error exception is raised if "path" does not denote either a simple-type value or an empty object or array.

4.6. DelMemb

The `DelMemb(doc, path)` primitive removes from the JSON instance document “doc” the empty (i.e., with a null value) object member located at the path “path”. An error exception is raised if “path” does not denote an empty object member.

5. Completeness and Minimality

In this section, we first show that the proposed set of primitive updates PO is complete and minimal. Then, we use this result to prove the completeness of the JUpdate language.

5.1. Completeness and Minimality of the Proposed Primitive Update Operations Set PO

Let us consider the complete JSON grammar as composed of the six production rules in BNF format listed in Figure 4.

```
(P0) <json>       ::= <value>
(P1) <object>     ::= { } | { <members> }
(P2) <members>    ::= <members> , <pair> | <pair>
(P3) <pair>       ::= <string> : <value>
(P4) <array>      ::= [ ] | [ <elements> ]
(P5) <elements>   ::= <elements> , <value> | <value>
(P6) <value>      ::= <object> | <array> |
                      <string> | <number> | <boolean> | null
```

Figure 4. The BNF syntax of the JSON language.

For the sake of brevity, in the grammar, we do not expand further the `<string>`, `<number>`, and `<boolean>` non-terminals, but we assume instead they can produce the desired terminal data value through a call to a function eval() (e.g., when the `<number>` non-terminal has to produce the value 123, we assume the function eval(`<number>`) returns 123). The non-terminal `<boolean>` produces the `true` or `false` literal names. However, other than that (and non-explicitation of “insignificant whitespace” specifications), the grammar in Figure 4 is perfectly equivalent to the standard JSON grammar presented in EBNF format in [1].

Furthermore, if PR is a variable denoting a production rule, we employ a dotted notation such that PR.type, PR.H, and PR.B refer to the type, head, and body of the rule, respectively. For example, if PR is the P5-type production rule `<elements>::=<elements>,<value>` then PR.type is P5, PR.H is `<elements>`, and PR.B is `<elements>,<value>`.

Then, we consider a consistent set of production rule applications, giving rise to the syntax tree corresponding to a valid JSON instance. For example, Figure 5 shows the syntax tree corresponding to the JSON instance (3) shown below.

```
doc.json =
{‘‘X’’:[{‘‘A’’:null,’’B’’:10},{},{‘‘C’’:’’xy’’}]} (3)
```

In particular, intermediate nodes of the syntax tree contain the head of a production rule, and their (one or two) children node(s) contain the non-terminal(s) in the body of such a rule. The root node always contains the start non-terminal `<json>`, which is the head of the only P0-type production rule used in the tree. The leaves of the tree are terminal values: strings, numbers, Boolean, or null values. Obviously, for any given valid JSON instance, there is such a set of production rule applications. We further consider a linear order of the production rule applications corresponding to the depth-first traversal of the syntax tree, which represents the order in which the production rules are applied. We will use a function next(PR) that returns the production rule that follows PR in such an order (the superscripts preceding the non-terminal in each node of the tree in Figure 5 correspond to the position in such an order). For example, if PR is ^{6}`<value>::=<array>`and is the 6th production rule applied in the syntax tree of Figure 5, next(PR) is the 7th production rule applied, that is ^{7}`<array>::=[<elements>]`.

Hence, we can prove that the set of primitive change operations introduced in Section 4 is minimal and complete. In particular, any given JSON instance can be constructed (or destroyed) via the execution of a sequence of such operations.

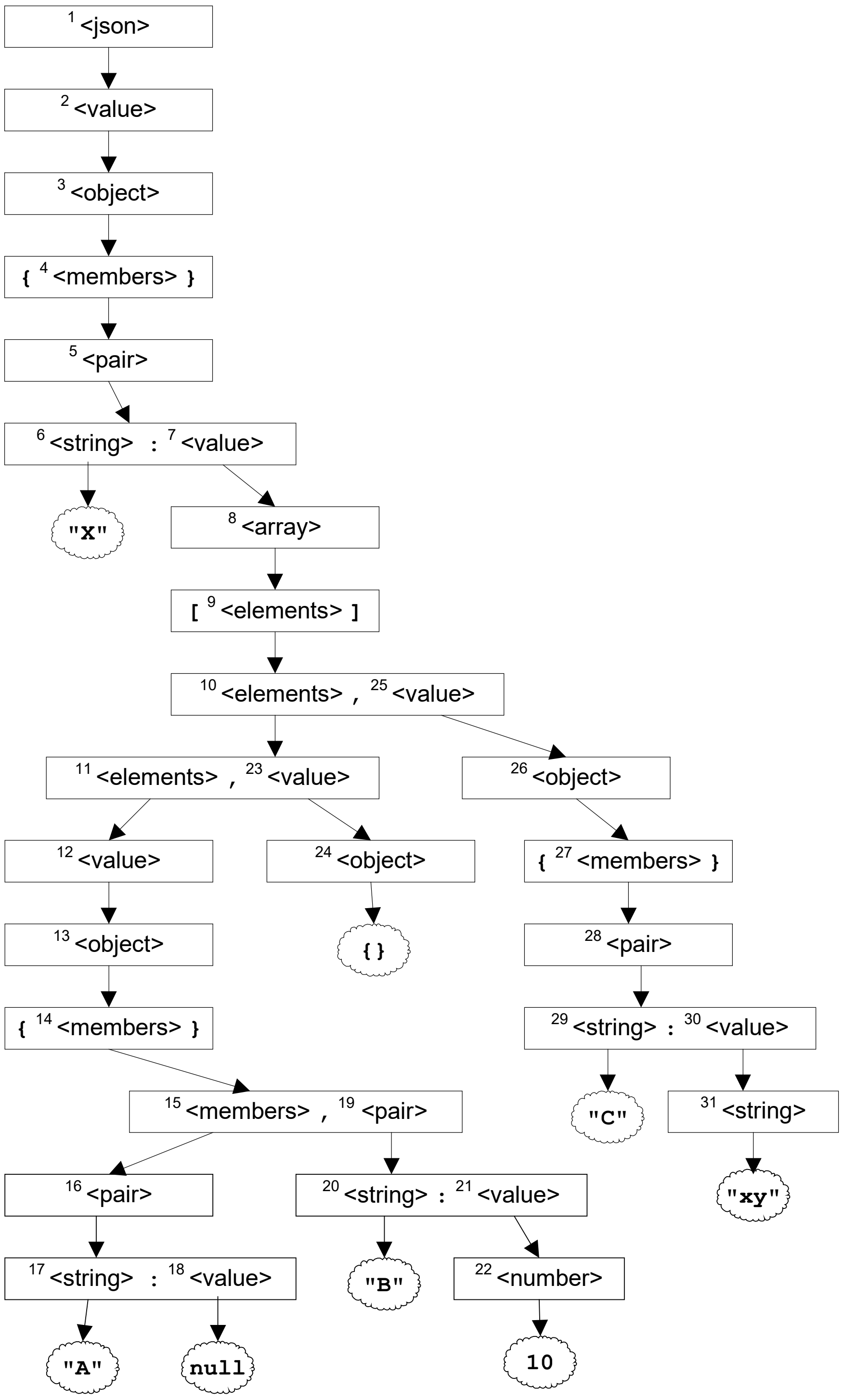


Figure 5. The syntax tree corresponding to the JSON instance (3).

We start by proving the completeness of the construction primitive operation set `{ CreaDoc, InsVal, InsMemb }` by showing how any set of production rule applications corresponding to the syntax tree of a valid JSON instance can be translated into a sequence of primitive operations, whose execution produces the same JSON instance from scratch. To this end, we introduce a recursive procedure applyPR(PR, doc, path), which applies the production rule PR to the document "doc" starting at path "path". The applyPR procedure is defined as follows:

```
procedure applyPR(PR, doc, path)
case PR.type of
P0: // PR.H = <json>
      output CreaDoc(doc);
      applyPR(next(PR), doc, $); // next(PR) must be P6-type
P1: // PR.H = <object>
      output InsVal(doc, path, object);
      if (PR.B ≠ {}) then applyPR(next(PR), doc, path) endif;
                                             // next(PR) must be P2-type
P2: // PR.H = <members>
      if (PR.B ≠ <pair>) then applyPR(next(PR), doc, path) endif;
                    // next(PR) must be P2-type
      applyPR(next(PR), doc, path); // next(PR) must be P3-type
P3:     // PR.H = <pair>
      Name := eval(next(PR)); // next(PR) must be <string>
      output InsMemb(doc, path, Name);
      applyPR(next(PR), doc, path.Name); // next(PR) must be P6-type
P4:     // PR.H=<array>
      output InsVal(doc, path, array);
      if (PR.B ≠ []) then applyPR(next(PR), doc, path) endif;
                                             // next(PR) must be P5-type
P5:     // PR.H = <elements>
      if (PR.B≠<value>) then applyPR(next(PR), doc, path) endif;
                                             // next(PR) must be P5-type
      applyPR(next(PR), doc, path[last]); // next(PR) must be P6-type
P6:     // PR.H = <value>
      if (PR.B = <object>) then applyPR(next(PR), doc, path)
                                             // next(PR) must be P1-type
      else if (PR.B = <array>) then applyPR(next(PR), doc, path)
                                             // next(PR) must be P4-type
      else if (PR.B ≠ null) then Value := eval(next(PR)) ;
                                   // next(PR) must be <string>, <number> or <boolean>
                              output InsVal(doc, path, Value)
      end if;
   end case;
end procedure.
```

The full translation involving the construction of the doc.json JSON instance can be done by invoking applyPR(PR, doc.json, null), where PR is the first production rule to be applied (i.e., ^{1}`<json>::=<value>`), that is the P0-type production rule whose head is the root of the syntax tree of the JSON instance. It easy to see that each case in the body of the procedure translates the semantics of a production rule of the grammar into the corresponding action(s) performed via change primitives. For example, the applyPR procedure translates the set of production rule applications depicted in Figure 5 and corresponding to the JSON instance (3) into the sequence of operations that follows:

```
CreaDoc(doc.json);
            → doc.json = null
InsVal(doc.json, $, object);
            → doc.json = {}
InsMemb(doc.json, $, X);
            → doc.json = {"X":null}
InsVal(doc.json, $.X, array);
            → doc.json = {"X":[]}
InsVal(doc.json, $.X[last], object);
            → doc.json = {"X":[{}]}
InsMemb(doc.json, $.X[last], A);
            → doc.json = {"X":[{"A":null}]}
InsMemb(doc.json, $.X[last], B);
            → doc.json = {"X":[{"A":null,"B":null}]}
InsVal(doc.json, $.X[last].B, 10);
            → doc.json = {"X":[{"A":null,"B":10}]}
InsVal(doc.json, $.X[last], object);
            → doc.json = {"X":[{"A":null,"B":10},{}]}
InsVal(doc.json, $.X[last], object);
            → doc.json = {"X":[{"A":null,"B":10},{},{}]}
InsMemb(doc.json, $.X[last], C);
            → doc.json = {"X":[{"A":null,"B":10},{},{"C":null}]}
InsVal(doc.json, $.X[last].C, xy);
            → doc.json = {"X":[{"A":null,"B":10},{},{"C":"xy"}]}
```

which clearly produce the same JSON instance (3) from scratch (the step-by-step construction of the contents of the doc.json instance is also shown above). A full trace of the execution of the applyPR procedure applied to this example can be found in Figure A1 of Appendix B.

In addition, to be complete, the proposed set of construction primitive changes `{ CreaDoc, InsVal, InsMemb }` is also minimal, as all of them are needed in the definition of the translation procedure applyPR.

As far as the destruction primitives are concerned, we observe that `DropDoc(doc)` is the inverse operation of `CreaDoc(doc)`, `DelVal(doc, path)` is the inverse operation of `InsVal(doc, path, val)`, and `DelMemb(doc, path)` is the inverse operation of `InsMemb (doc, path, name)`, as each of the former undoes the effects of the latter. For example, it can be easily seen that the JSON instance (3) can be completely destroyed via the application of the sequence of primitive changes that follows:

```
DelVal(doc.json, $.X[last].C);
DelMemb(doc.json, $.X[last], C);
DelVal(doc.json, $.X[last]);
DelVal(doc.json, $.X[last]);
DelVal(doc.json, $.X[last].B);
DelMemb(doc.json, $.X[last], B);
DelMemb(doc.json, $.X[last], A);
DelVal(doc.json, $.X[last]);
DelVal(doc.json, $.X);
DelMemb(doc.json, $,X);
DelVal(doc.json, $);
DropDoc(doc.json).
```

In general, given any JSON instance doc.json defined by a production rule application set corresponding to its syntax tree, in order to find the sequence of primitive changes that completely destroys it, we can:

1. Use the applyPR procedure in order to derive from the production rule application set the sequence of creation primitives that could have been used to construct the doc.json instance;
2. Scan the sequence of creation primitives in reverse order and replace each of them by its inverse destruction primitive.

Hence, also the set of destruction primitives `{ DropDoc, DelVal, DelMemb }` comes out as complete and minimal, as all of them and only them are needed to destroy any arbitrary JSON instance.

Therefore, the full set PO = `{ CreaDoc, DropDoc, InsVal, DelVal, InsMemb, DelMemb }` of change primitives presented in Section 4 is complete and minimal.

5.2. Completeness of JUpdate

In this section, we prove that the JUpdate language is complete by showing that a suitable use of a subset of its operations is equivalent to the set of primitive operations PO presented in Section 4, which is complete. In fact, starting from the JUpdate API procedures, the primitive operations can be defined as follows:

```
procedure CreaDoc(doc)
    CreateDocument(doc, null);
end procedure;
procedure DropDoc(doc)
    DeleteDocument(doc);
end procedure;
procedure InsVal(doc, path, value)
    if value=object then InsertValue(doc, path, {});
    else if value=array then InsertValue(doc, path, []);
    else InsertValue(doc, path, value);
    end if;
end procedure;
procedure InsMemb(doc, path, name)
    InsertMember(doc, path, name, null);
end procedure;
procedure DelVal(doc, path)
    DeleteValue(doc, path);
end procedure;
procedure DelMemb(doc, path)
    DeleteMember(doc, path);
end procedure.
```

Owing to the equivalence with the primitives in PO, the subset of JUpdate operations used in this proof constitutes a complete and minimal subset of HLOs necessary for applying any given transformation to a JSON instance document. The other proposed HLOs, together with the other possible uses of the HLOs in this subset, provide *syntactic sugar* for putting powerful and friendly operations at the user's disposal.

6. Definability of the HLOs Making up JUpdate

In this section, we show how the HLOs presented in Section 2 and making up the proposed JUpdate language can be defined starting from the primitive operations presented in Section 4. In particular, the operational semantics of JUpdate API procedures is given in terms of operations in PO.

To this purpose, we distinguish between two basic HLOs, `InsertValue` and `DeleteValue`, which we will show how can be defined from primitive operations only, and the rest of the HLOs, which can be more conveniently defined from primitive operations and previously defined HLOs.

We assume that the applicability errors that have been described in the definitions of Section 2 are dealt with separately and, thus, the procedures presented in the following are executed when errors are absent and, thus, exceptions have not been raised.

6.1. Basic HLOs: InsertValue and DeleteValue

The operational semantics of the `InsertValue` HLO can be defined in terms of primitive operations as follows:

```
procedure InsertValue(doc, path, value)
    ST := syntaxCheck(value, jsonSyntax);
    PR := ST.root;
    if PR.type = P6 then
        call applyPR(PR, doc, path);
        execute the resulting sequence of primitive operations;
    else alert("Syntax error in value");
    end if;
end procedure;
```

where jsonSyntax corresponds to the encoding of the JSON grammar in Figure 4 and the syntaxCheck() procedure, which can be implemented via a simply adapted standard JSON parser, returns the syntactic tree of the "value" expression. Hence, the syntactic check on the correctness of "value" is successful if the root PR of ST is a P6-type production rule.

For instance, consider the JSON instance (3) introduced in Section 5.1 and an object-type value `{‘‘A’’:true,’’Z’’:[0,5]}`, which we want to insert between the last two elements of the array in doc.json. This can be done by executing the HLO `InsertValue(doc.json, $.X[2], {‘‘A’’:true,’’Z’’:[0,5]})`. Then, the syntactic check on "value" is successful and produces the syntactic tree ST depicted in Figure 6. Then, the effect of the applyPR procedure is the generation of the primitive operation sequence:

```
InsVal(doc.json, $.X[2], object);
InsMemb(doc.json, $.X[2], A);
InsVal(doc.json, $.X[2].A, true);
InsMemb(doc.json, $.X[2], Z);
InsVal(doc.json, $.X[2].Z, array);
InsVal(doc.json, $.X[2].Z[last], 0);
InsVal(doc.json, $.X[2].Z[last], 5);
```

whose execution eventually updates the JSON instance (3) as follows:

```
doc.json =
{‘‘X’’:[{‘‘A’’:null,’’B’’:10},{},{‘‘A’’:true,’’Z’’:[0,5]},{‘‘C’’:’’xy’’}]}.(4)
```

A full trace of the execution of the applyPR procedure applied to this example can be found in Figure A2 of Appendix B.

In practice, there is no need for applying the above definition in order to execute an `InsertValue` *HLO, as it is sufficient (after the syntactic checks on "value" and "path") to directly insert the JSON text "value" as a whole at the desired point "path" in the document.*

The operational semantics of the `DeleteValue` HLO can be defined in terms of primitive operations as follows:

```
procedure DeleteValue(doc, path)
    ST := syntax Check(doc.path, jsonSyntax);
    PR := ST.root;
    if PR.type=P6 then
        call applyPR(PR, doc, path);
        scan the resulting sequence of primitive operations in reverse order
          and replace each one by its inverse one;
        execute the resulting sequence of primitive operations;
    else alert("Syntax error in path");
    end if;
end procedure;
```

where jsonSyntax corresponds to the encoding of the JSON grammar in Figure 4 and the syntaxCheck() procedure, which can be implemented via a simply adapted standard JSON parser, returns the syntactic tree of value. The syntactic check on the correctness of "path" is successful if the root PR of ST is a P6-type production rule.

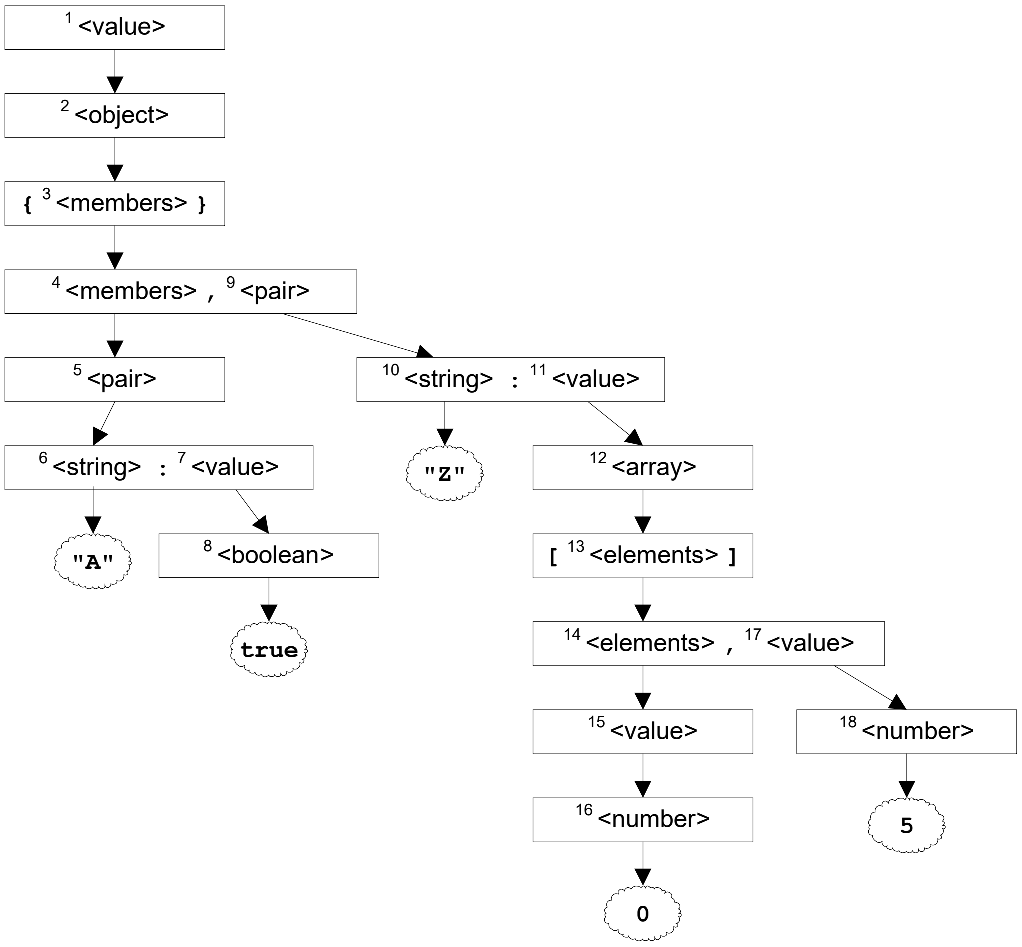


Figure 6. The syntax tree of the JSON object-type value {"A":true, "Z":[0,5]}.

For instance, consider the JSON instance (4) presented above and the deletion of the array-type value of the object member with label Z. This can be done by executing the HLO `DeleteValue(doc.json, $.X[2].Z)`. Then, the syntactic check on "value" is successful (the path `$.X[2].Z` denotes an array-type value), and the effect of the applyPR procedure is the generation of the primitive operation sequence:

```
InsVal(doc.json, $.X[2].Z, array);
InsVal(doc.json, $.X[2].Z[last], 0);
InsVal(doc.json, $.X[2].Z[last], 5).
```

Then, the scan in reverse order and substitution with inverse primitive operations produces the sequence:

```
DelVal(doc.json, $.X[2].Z[last]);
DelVal(doc.json, $.X[2].Z[last]);
DelVal(doc.json, $.X[2].Z, array);
```

whose execution eventually updates the JSON instance (4) as follows:

```
doc.json =
{"X":[{"A":null,"B":10},{},{"A":true,"Z":null},{"C":"xy"}]}.
```
(5)

In practice, also in this case, there is no need for applying the above definition in order to execute a `DeleteValue` *HLO, as it is sufficient (after the syntactic checks on "value" and "path") to directly delete the JSON text "value" as a whole from the desired point "path" in the document.*

6.2. Derived HLOs

In this subsection, we show how all the other HLOs making up JUpdate can be defined starting from primitive operations presented in Section 4 and the two basic HLOs `InsertValue` and `DeleteValue` described above.

The `CreateDocument` HLO can be simply defined from the `CreaDoc` primitive operation and the `InsertValue` basic HLO as follows:

```
procedure CreateDocument(doc, value)
    CreaDoc(doc);
    InsertValue(doc, $, value);
end procedure.
```

The `DropDocument` HLO can be simply defined from the `DropDoc` primitive operation and the `DeleteValue` basic HLO as follows:

```
procedure DropDocument(doc)
    DeleteValue(doc, $)
    DropDoc(doc);
end procedure.
```

The `UpdateValue` HLO can be simply defined from the basic `InsertValue` and `DeleteValue` HLOs as follows:

```
procedure UpdateValue(doc, path, value)
    DeleteValue(doc, path);
    InsertValue(doc, path, value);
end procedure.
```

The `CopyValue` HLO can be simply defined using the basic `InsertValue` HLO as follows:

```
procedure CopyValue(doc, path, newpath)
    InsertValue(doc, newpath, doc:path); /* inserts at location newpath in doc
                                           the value of doc:path
                                           which is the value of path in doc */
end procedure.
```

The `MoveValue` HLO can be simply defined from the `CopyValue` and `DeleteValue` HLOs as follows:

```
procedure MoveValue(doc, path, newpath)
    CopyValue(doc, path, newpath);
    DeleteValue(doc, path);
end procedure.
```

The `InsertMember` HLO can be simply defined from the `InsMemb` primitive operation and the basic `InsertValue` HLO as follows:

```
procedure InsertMember(doc, path, name, value)
    InsMemb(doc, path, name);
    InsertValue(doc, path.name, value);
end procedure.
```

The `DeleteMember` HLO can be simply defined from the `DelMemb` primitive operation and the basic `DeleteValue` HLO as follows:

```
procedure DeleteMember(doc, path)
    DeleteValue(doc, path);
    Let name be such that path=p.name;
    DelMemb(doc, p, name);
end procedure.
```

The `RenameMember` HLO can be simply defined from the `InsertMember` and the `DeleteMember` HLOs as follows:

```
procedure RenameMember(doc, path, name)
    Let oldname be such that path=p.oldname;
    InsertMember(doc, p, name, doc:path); /* creates a new member p.name
                                            with the value doc:path which is
                                            the value of p.oldname in doc */
```

```
        DeleteMember(doc, path);
    end procedure.
```

In practice, the `RenameMember` *HLO can be more efficiently implemented by directly replacing with "name" the string representing the name of the member denoted by "path".*

The `ReplaceMember` HLO can be simply defined using the `RenameMember` and `UpdateValue` HLOs as follows:

```
procedure ReplaceMember(doc, path, name, value)
    RenameMember(doc, path, name);
    UpdateValue(doc, path, value);
end procedure.
```

The `CopyMember` HLO can be simply defined using the `InsertMember` HLO as follows:

```
procedure CopyMember(doc, path, newpath)
    Let path be such that path=p.name;
    InsertMember(doc, newpath, name, doc:path);
                                /* creates a new member newpath.name
                                   with the value doc:path
                                   which is the value of p.name in doc */
end procedure.
```

The `MoveMember` HLO can be simply defined from the `CopyMember` and `DeleteMember` HLOs as follows:

```
procedure MoveMember(doc, path, newpath)
    CopyMember(doc, path, newpath);
    DeleteMember(doc, path);
end procedure.
```

As already observed in Section 2, the `UpdateObject` HLO is equivalent to a sequence of `UpdateValue` operations.

7. Related Work Discussion

The problem of updating JSON data/documents has not been previously studied in the literature, whereas many works have dealt with JSON data querying. Indeed, several JSON query languages have been proposed such as JSONiq [14], which is an XQuery-like language, SQL++ [16,17], which is a SQL-like language, and JSONPath [25], which is an XPath-like language. Notice that since our JUpdate language is SQL-like, it could be combined with SQL++ to produce a full JSON DML.

As far as mainstream NoSQL data stores are concerned, each one of them provides, for managing JSON data, either a proprietary query language, such as Couchbase whose language is N1QL, Cassandra whose language is CQL, and AsterixDB whose language is AQL, or some low-level programmatic APIs, such as Amazon Dynamo and Redis. In the following, we try to compare our JUpdate language to the three languages N1QL, CQL, and AQL, which also provide for update operations.

- N1QL, CQL, and AQL, similarly to JUpdate, are SQL-like. However, contrary to them, JUpdate is DBMS-independent. Moreover, such languages are based on predefined precise data models based on collections of homogeneous JSON objects (e.g., key-value/keyspaces for N1QL, tables for CQL, and ADM/dataspaces for AQL), to be used as relational tables (with objects playing the role of table rows). CQL also supports collections (i.e., lists/sets/maps) to store structured multiple values in a table. JUpdate is not based on any predefined data model and, thus, it best suits the JSON philosophy and can be used for the maintenance of generic JSON documents.
- N1QL and AQL support an `UPSERT` operation that behaves as an `UPDATE` if corresponding data (which are JSON object members) are found or as an `INSERT` if corresponding data are not found, such as the `RECORD` operation proposed in [29]. Such operations are based on the precise data models based on collections of JSON objects recalled above and cannot be used on generic JSON documents. On object members, the JUpdate `UPDATE` operation works as an `UPSERT`, since it behaves as `INSERT` if the value

to be updated has a null value. On a JSON array, the JUpdate `UPDATE` operation still works as an `UPSERT` if used on the "[last]" component of the array (such component is replaced if present or a new component is inserted if the array is empty). Hence, JUpdate does not need an `UPSERT`, as the `UPDATE` operation can be used in a more general and powerful way (i.e., to make "upsertions" in both objects and arrays placed at any nesting level of the structure of a generic JSON document).

- Contrary to CQL, which supports a `BATCH` operation (i.e., a sequence of multiple `INSERT`, `UPDATE`, and `DELETE` operations that are grouped together in a single operation so that they can be executed together), JUpdate does not support it. In fact, JUpdate allows its users to specify any valid sequence of operations among the set of fourteen operations of JUpdate and not only among a set of three operations.

As for the well-known NoSQL DBMS, MongoDB, it provides also a proprietary data manipulation language for performing CRUD (https://docs.mongodb.com/manual/crud/ (accessed on 31 December 2021)) (create, read, update, and delete) operations on JSON documents. However, its syntax is neither SQL- nor XQuery-like.

Furthermore, several authors have dealt with the integration of JSON data in relational DBMSs, such as [11,30–33], focusing only on JSON data querying in such a context. In the present work, we consider JSON data stored in a native JSON (NoSQL) environment and focus on JSON data updating in such a framework. Moreover, an ANSI SQL/JSON standard [21–23] has been created to extend the specifications of SQL to support the management of JSON documents. However, in [24], the author shows that this standard does not allow users to change single parts of JSON documents and proposes to improve it in this direction via an extension of the SQL `UPDATE` statement syntax and semantics.

Thus, the investigation of the state-of-the-art and the state-of-the-practice shows that there is no consensual or standard language for updating native JSON data.

Considering formal aspects, logical foundations of JSON structures and JSON querying have been studied in [3,5], respectively. Our paper deals with an operational formalization of JSON updates.

JSON Patch [34] is a standard for specifying a sequence of changes to a JSON document through the definition of another JSON document, which is suitable to be used with the HTTP PATCH method, called a JSON Patch document,. This latter consists of an array of objects where each object describes a change operation on the corresponding JSON document through three members: (i) "op", which represents the desired operation; notice that JSON Patch provides six operations: "add", "remove", "replace", "move", "copy", and "test"; (ii) "path", which represents the path of the target node; (iii) "value", which represents a new value if needed, or "from", which represents a source path if needed. The specified operations have to be executed in their defined order. Similarly to the concept of "transaction" in relational DBMSs, the sequence of these operations is either completely executed or not executed at all if one of its operations fails. In Appendix C, we show how a JSON Patch processor can be set up using JUpdate operations. It is also shown how JSON Patch can be considered less powerful and less user-friendly than JUpdate, owing to the overload of its supplied operators.

As far as high-level update languages are concerned, no other proposals can be found in the literature on JSON data management. Even existing JSON NoSQL DBMSs do not provide any support for this type of operations. Notice that in a previous work [35], we have dealt with high-level operations but for changing XML schemas in a temporal XML environment that supports schema versioning [36–38].

Furthermore, it is worth mentioning that the well-known W3C XML language [39] has many similarities with JSON. Indeed, they are both based on tree/hierarchical data models to be used for the storage and exchange of structured, semi-structured, and unstructured data, including all emerging and NoSQL data. However, although there are two languages that are recommended by the W3C to query and to update XML data (i.e., XQuery [40] and XQuery Update Facility (XUF) 1.0 [41], respectively), there are no similar recommendations by the IETF to query and to update JSON data. Notice that several

XML update languages have been proposed before the recommendation of XUF 1.0, such as XUpdate [42], SiXDML [43], UpdateX [44], XML-RL update language [45], GLASS^{U} [46], and FLUX [47]. XUF 1.0 (its 3.0 release is yet under development [48]) provides a set of update operations, which includes update primitives that are the components of pending update lists, and update routines that are used in defining XQuery semantics but do not appear on pending update lists. The update primitives whose number is twelve are as follows: insertBefore, insertAfter, insertInto, insertIntoAsFirst, insertIntoAsLast, insertAttributes, delete, replaceNode, replaceValue, replaceElementContent, rename, and put. The update routines whose number is six are as follows: mergeUpdates, applyUpdates, revalidate, removeType, setToUntyped, and propagateNamespace. As for the pending update list, it is defined in [41] as an unordered collection of update primitives, which represent node state changes that have not yet been applied. Notice also that recently, Brahmia et al. [49] have provided a survey of the proposals (approaches, languages, tools, etc.) that have dealt with XML data update, in a conventional (i.e., non-temporal) or a temporal environments. Furthermore, the authors of [50] have extended the XUF 1.0 language, with a VALID clause that allows the specification of an applicability period of an update (i.e., insert, replace, delete, or rename) operation, to support the manipulation of *temporal* XML data.

In [51], we have used the high-level JSON update operations of JUpdate to define the operational semantics and to provide a basis for the implementation of the high-level JSON Schema change operations of our JSON Schema change language previously proposed in [52], based on the fact that a JSON Schema [3,53] file is also a plain JSON [1] file.

8. Conclusions

In this paper, we have proposed JUpdate, a JSON Update language, which includes a complete set of fourteen user-friendly high-level operations. The definition of the semantics of the proposed JUpdate operations has been based on a set of six primitive update operations, which has been proved complete and minimal. This theoretical underpinning, which was not provided before for other JSON update (sub)languages, shows that JUpdate is actually all that is needed for the maintenance of generic JSON documents.

To the best of our knowledge, we are the first to propose (i) a user-friendly SQL-like JUpdate language composed of high-level operations for manipulating valid portions of JSON instance documents and (ii) a set of primitive operations for updating native JSON data, on which the semantics of JUpdate is based and for which completeness and minimality properties have been proved. Moreover, our primitives and high-level operations are so general, well-founded, and DBMS-independent that on one hand, they can easily be implemented to set up a stand-alone JUpdate processor, and on the other hand, they could also be exploited to support high-level declarative (and not low-level programmatic) JSON update APIs.

Since several JSON-based emerging advanced applications (e.g., e-government and e-health applications) require bookkeeping of the history of data changes and, therefore, need to manage the evolution over time of JSON data, it will be interesting to extend JUpdate to also support updating of temporal JSON data. For this purpose, we intend in our future work to propose a temporal extension of JUpdate, named τJUpdate (Temporal JUpdate), to equip our temporal JSON-based NoSQL framework τJSchema (Temporal JSON Schema) [7,54].

Author Contributions: Conceptualization, Z.B., S.B., F.G. and R.B.; methodology, Z.B., S.B., F.G. and R.B.; software, Z.B., S.B. and F.G; validation, Z.B., S.B., F.G. and R.B.; formal analysis, Z.B., S.B. and F.G.; investigation, Z.B., S.B. and F.G.; resources, S.B.; data curation, S.B.; writing—original draft preparation, Z.B., S.B. and F.G.; writing—review and editing, Z.B. and F.G.; visualization, Z.B., S.B. and F.G.; supervision, F.G. and R.B.; project administration, Z.B.; funding acquisition, F.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. JUpdate Syntax

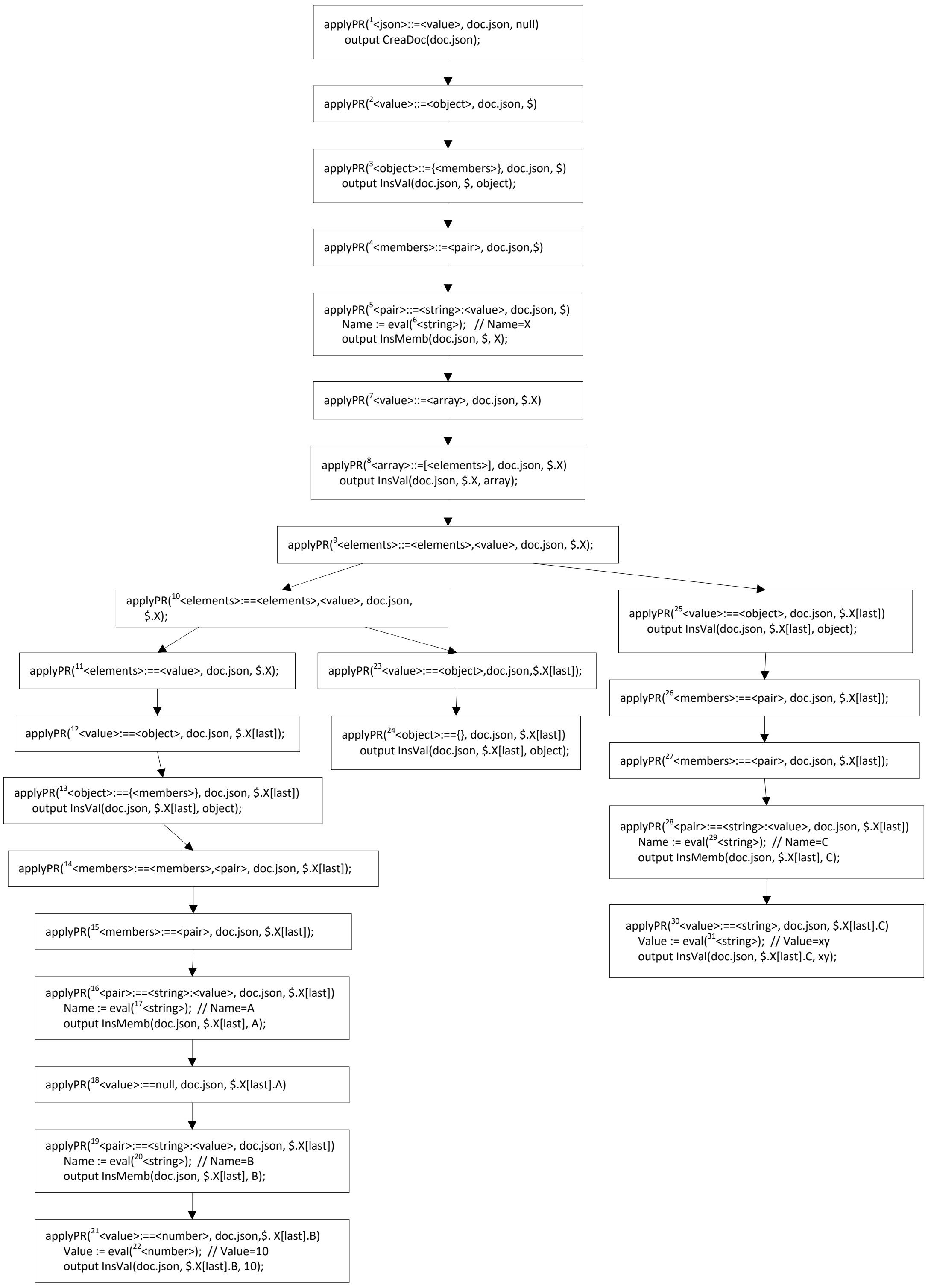
In this Appendix, we provide a full syntax of the JUpdate language, which was specified in BNF format.

```
<jUpdate>          ::= <docUpdateStmt> | <valUpdateStmt> |
           <objUpdateStmt>
<docUpdateStmt>    ::= <creaDocStmt> | <dropDocStmt>
<valUpdateStmt>    ::= <insertStmt> | <deleteStmt> | <updValStmt> |
                       <updCopyStmt> | <updMoveStmt>
<objUpdateStmt>    ::= <addMembStmt> | <dropMembStmt> |
                       <renameMembStmt> | <replaceMembStmt> |
                       <copyMembStmt> | <moveMembStmt> | <updObjStmt>
<creaDocStmt>      ::= CREATE <docName> <jsonValue>
<dropDocStmt>      ::= DROP <docName>
<insertStmt>        ::= INSERT INTO <name> <jsonPath> <jsonValue>
<deleteStmt>        ::= DELETE FROM <name> <jsonPath>
<updValStmt>        ::= <updateDoc> <jsonPath> <jsonValue>
<updCopyStmt>      ::= <updateDoc> COPY <fromToPath>
<updMoveStmt>      ::= <updateDoc> MOVE <fromToPath>
<addMembStmt>      ::= <alterObj> ADD <membName> <jsonValue>
<dropMembStmt>    ::= <alterObj> DROP <membName>
<renameMembStmt>    ::= <alterObj> RENAME <membName> TO <string>
<replaceMembStmt>  ::= <alterObj> REPLACE <membName>
                       WITH <string> <jsonValue>
<copyMembStmt>        ::= <alterObj> COPY <membName> <toPath>
<moveMembStmt>        ::= <alterObj> MOVE <membName> <toPath>
<updObjStmt>        ::= <updateDoc> <objPath> SET <assignments>
<assignments>      ::= <assignment> | <assignment>,<assignments>
<assignment>        ::= <string> = <json>
<docName>        ::= DOCUMENT <name>
<jsonValue>        ::= VALUE <json> | <jPath> | <selectStmt>
<jsonPath>        ::= PATH <jPath>
<updateDoc>        ::= UPDATE <name>
<fromToPath>        ::= FROM <jPath> <toPath>
<toPath>        ::= TO <jPath>
<alterObj>        ::= ALTER <docName> <objPath>
<objPath>        ::= OBJECT <jPath>
<membName>        ::= MEMBER <string>
```

For the sake of simplicity, as for the JSON grammar in Figure 4, also in this case, we did not explicit "insignificant whitespace" specifications. The non-terminal `<json>`, producing a valid JSON value, actually corresponds to the one in Figure 4 (also `<string>` is the same as appearing in Figure 4). The non-terminals `<name>` and `<jPath>` are assumed to produce a valid JSON document name and a valid JSON Path expression (It can be expanded as shown, for instance, in "https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/diagrams-basic-sql-json-path-expression-syntax.html" (accessed on 31 December 2021)), respectively. In the production rule for `<jsonValue>`, the `<jPath>` and `<selectStmt>` non-terminals represent the possibility to use a subquery formulated as a JSONPath expression or as a `SELECT` statement in some SQL-like JSON query language (e.g., SQL++), respectively, to compute the required JSON value rather than providing it as a literal constant.

Appendix B. Execution Traces of the `applyPR` Procedure

In this Appendix, we show in Figures A1 and A2 the execution trace of the **applyPR** procedure for the cases presented in Sections 5.1 and 6.1, respectively. The order in which primitive operation calls are output corresponds to the depth-first traversal of the syntax trees displayed in Figures 5 and 6, respectively, which represents the order in which the production rules are applied (and is visualized in Figures A1 and A2 via the superscripts preceding the first argument in each **applyPR** call).



```
applyPR($^{1}$<json>::=<value>, doc.json, null)
    output CreaDoc(doc.json);

applyPR($^{2}$<value>::=<object>, doc.json, $)

applyPR($^{3}$<object>::={<members>}, doc.json, $)
    output InsVal(doc.json, $, object);

applyPR($^{4}$<members>::=<pair>, doc.json,$)

applyPR($^{5}$<pair>::=<string>:<value>, doc.json, $)
    Name := eval($^{6}$<string>);   // Name=X
    output InsMemb(doc.json, $, X);

applyPR($^{7}$<value>::=<array>, doc.json, $.X)

applyPR($^{8}$<array>::=[<elements>], doc.json, $.X)
    output InsVal(doc.json, $.X, array);

applyPR($^{9}$<elements>::=<elements>,<value>, doc.json, $.X);

applyPR($^{10}$<elements>:==<elements>,<value>, doc.json,
    $.X);

applyPR($^{11}$<elements>:==<value>, doc.json, $.X);

applyPR($^{12}$<value>:==<object>, doc.json, $.X[last]);

applyPR($^{13}$<object>:=={<members>}, doc.json, $.X[last])
    output InsVal(doc.json, $.X[last], object);

applyPR($^{14}$<members>:==<members>,<pair>, doc.json, $.X[last]);

applyPR($^{15}$<members>:==<pair>, doc.json, $.X[last]);

applyPR($^{16}$<pair>:==<string>:<value>, doc.json, $.X[last])
    Name := eval($^{17}$<string>);  // Name=A
    output InsMemb(doc.json, $.X[last], A);

applyPR($^{18}$<value>:==null, doc.json, $.X[last].A)

applyPR($^{19}$<pair>:==<string>:<value>, doc.json, $.X[last])
    Name := eval($^{20}$<string>);  // Name=B
    output InsMemb(doc.json, $.X[last], B);

applyPR($^{21}$<value>:==<number>, doc.json,$. X[last].B)
    Value := eval($^{22}$<number>);  // Value=10
    output InsVal(doc.json, $.X[last].B, 10);

applyPR($^{23}$<value>:==<object>,doc.json,$.X[last]);

applyPR($^{24}$<object>:=={}, doc.json, $.X[last])
    output InsVal(doc.json, $.X[last], object);

applyPR($^{25}$<value>:==<object>, doc.json, $.X[last])
    output InsVal(doc.json, $.X[last], object);

applyPR($^{26}$<members>:==<pair>, doc.json, $.X[last]);

applyPR($^{27}$<members>:==<pair>, doc.json, $.X[last]);

applyPR($^{28}$<pair>:==<string>:<value>, doc.json, $.X[last])
    Name := eval($^{29}$<string>);  // Name=C
    output InsMemb(doc.json, $.X[last], C);

applyPR($^{30}$<value>:==<string>, doc.json, $.X[last].C)
    Value := eval($^{31}$<string>);  // Value=xy
    output InsVal(doc.json, $.X[last].C, xy);
```

Figure A1. The execution trace of applyPR for the JSON instance (3) with syntax tree in Figure 5.

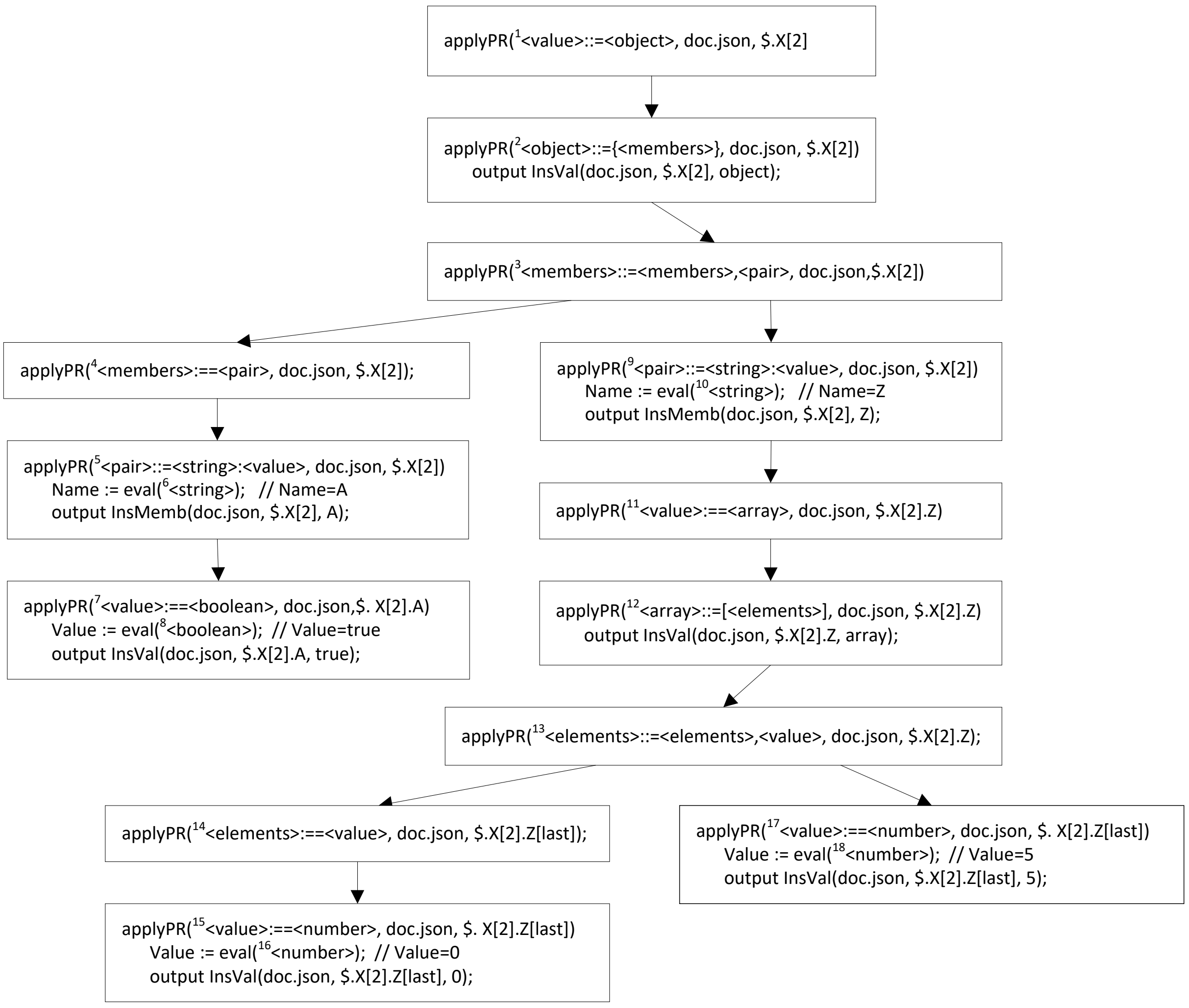

applyPR(1<value>::=<object>, doc.json, $.X[2]
applyPR(2<object>::={<members>}, doc.json, $.X[2]) output InsVal(doc.json, $.X[2], object);
applyPR(3<members>::=<members>,<pair>, doc.json,$.X[2])
applyPR(4<members>:==<pair>, doc.json, $.X[2]);
applyPR(5<pair>::=<string>:<value>, doc.json, $.X[2]) Name := eval(6<string>); // Name=A output InsMemb(doc.json, $.X[2], A);
applyPR(7<value>:==<boolean>, doc.json,$. X[2].A) Value := eval(8<boolean>); // Value=true output InsVal(doc.json, $.X[2].A, true);
applyPR(9<pair>::=<string>:<value>, doc.json, $.X[2]) Name := eval(10<string>); // Name=Z output InsMemb(doc.json, $.X[2], Z);
applyPR(11<value>:==<array>, doc.json, $.X[2].Z)
applyPR(12<array>::=[<elements>], doc.json, $.X[2].Z) output InsVal(doc.json, $.X[2].Z, array);
applyPR(13<elements>::=<elements>,<value>, doc.json, $.X[2].Z);
applyPR(14<elements>:==<value>, doc.json, $.X[2].Z[last]);
applyPR(15<value>:==<number>, doc.json, $. X[2].Z[last]) Value := eval(16<number>); // Value=0 output InsVal(doc.json, $.X[2].Z[last], 0);
applyPR(17<value>:==<number>, doc.json, $. X[2].Z[last]) Value := eval(18<number>); // Value=5 output InsVal(doc.json, $.X[2].Z[last], 5);

Figure A2. The execution trace of applyPR for the execution of the HLO `InsertValue(doc.json, $.X[2], {‘‘A’’:true,’’Z’’:[0,5]})` on the JSON instance (3).

Appendix C. Implementation of a JSON Patch Processor

In this appendix, we show how a JSON Patch [33] processor can be easily implemented using our proposed JUpdate HLOs.

JSON Patch is an Internet Engineering Task Force (IETF) standard that allows the specification of a sequence of update operations to be executed on a JSON document. A JSON Patch document is itself a JSON document, which is defined as an array of objects such that each object has two mandatory members, “op” and “path”, and one optional member, “value” or “from”.

- “op” denotes the operation to be applied; JSON Patch supports six operations: “add”, “remove”, “replace”, “move”, “copy” and “test”;
- “path” denotes the target path of the operation, that is the path of the JSON component affected by the operation;
- “value” denotes the (new) value provided by the user, to be used by the “add” and “replace” operation;
- “from” denotes the source path to be used by the “move” and “copy” operations.

For example, if we have a JSON document whose initial contents are as follows:

```
{ "researchLab":"DataLab", "URL":"http://www.mhesr.uni-sfax.datalab.tn/",
 "fax":"(+216)11111111" }
```

and the user specifies the following JSON Patch document:

```
[ { "op":"add", "path":"/country", "value":"Tunisia" },
  { "op":"replace", "path":"/URL", "value":"http://www.datalab.tn/" },
  { "op":"remove", "path":"/fax" } ],
```

after the application of this Patch, the contents of the JSON document become as follows:

```
{ "researchLab":"DataLab", "country":"Tunisia",
  "URL":"http://www.datalab.tn/"}.
```

Notice that operations contained in a JSON Patch document are sequentially executed in the order they are specified. Moreover, a JSON Patch document behaves as a transaction, as the sequence of operations it contains is atomic: either it completely succeeds, if all specified operations are successfully executed, or it must not leave traces of its partial execution, if one of the specified operations fails to complete.

The algorithm for processing a JSON Patch file PF using JUpdate operations is as follows (we do not consider the JSON Patch "test" operation as it is not a specification of updates):

```
procedure applyJPatch(PF, doc)
for (index := 0; index <PF.length; index++)
PO := PF[index];
Operation := PO.op; Path := PO.path; Value := PO.value; From := PO.from;
case Operation of
add: if Path = p.name
then if p.name exists in doc
then UpdateValue(doc, Path, Value);
else InsertMember(doc, p, name, Value);
end if;
else if Path = $
then UpdateValue(doc, Path, Value);
else InsertValue(doc, Path, Value);
end if;
remove: if Path = p.name
                  then DeleteMember(doc, Path);
                  else DeleteValue(doc, Path);
            end if:
          replace:   UpdateValue(doc, Path, Value);
          move:   if From = p.name
                  then MoveMember(doc, From, Path);
                  else MoveValue(doc, From, Path);
                end if;
          copy:   if From = p.name
                  then CopyMember(doc, From, Path);
                  else CopyValue(doc, From, Path);
                end if;
      end case;
  end for;
end procedure.
```

Although this is not a great difference in practical terms, notice that a reverse translation from JUpdate to JSON Patch is not always possible, as the JSON Patch update language is, strictly speaking, formally less expressive than JUpdate. In particular, JSON Patch is not complete in the sense of Section 5.1, as it lacks operations equivalent to the `CreateDocument/DropDocument` (or `CreaDoc/DropDoc`) pair.

Moreover, the semantics and usage of JSON Patch is somewhat less clear than JUpdate due to the overload of its operations, which are allowed to work either on values (e.g., array elements) and on members of objects. For instance, the "add" operation comes out particularly overloaded: if "path" denotes an existing object member, the value of such a member is replaced by the supplied "value" (being equivalent to a JUpdate `UpdateValue` operation); if "path" denotes a non-existing member of an existing object, such a member is added to the object with a value equal to the supplied "value" (being equivalent to a JUpdate `InsertMember` operation); if "path" denotes a location within an existing array, the supplied "value" is added as a new element of the array at the specified location (being equivalent to a JUpdate `InsertValue` operation); if "path" denotes the root of a document, the whole contents of the document are replaced by the supplied "value" (still being equivalent to a JUpdate `UpdateValue` operation). Although the resulting overloaded operation can be considered "very high level", since with a single tool we can actually produce a variety of effects, its correct usage may be quite unclear and error-prone to unskilled users.

References

1. Internet Engineering Task Force. The JavaScript Object Notation (JSON) Data Interchange Format, Internet Standards Track document, December 2017. Available online: https://tools.ietf.org/html/rfc8259 (accessed on 31 December 2021).
2. Jackson, W. *JSON Quick Syntax Reference*; Apress: Berkeley, CA, USA, 2016.
3. Pezoa, F.; Reutter, J.L.; Suarez, F.; Ugarte, M.; Vrgoč, D. Foundations of JSON Schema. In Proceedings of the 25th International Conference on World Wide Web (WWW 2016), Montreal, QC, Canada, 11–15 April 2016; pp. 263–273.
4. Bourhis, P.; Reutter, J.L.; Suárez, F.; Vrgoč, D. JSON: Data model, query languages and schema specification. In Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2017), Chicago, IL, USA, 14–19 May 2017; pp. 123–135.
5. Hidders, J.; Paredaens, J.; van den Bussche, J. J-Logic: Logical foundations for JSON querying. In Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2017), Chicago, IL, USA, 14–19 May 2017; pp. 137–149.
6. Baazizi, M.-A.; Colazzo, D.; Ghelli, G.; Sartiani, C. Schemas and Types for JSON Data: From Theory to Practice. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD'2019), Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 2060–2063.
7. Brahmia, S.; Brahmia, Z.; Grandi, F.; Bouaziz, R. A Disciplined Approach to Temporal Evolution and Versioning Support in JSON NoSQL Data Stores. In *Emerging Technologies and Applications in Data Modeling and Processing*; Ma, Z., Yan, L., Eds.; IGI Global: Hershey, PA, USA, 2019; pp. 114–133.
8. Friesen, J. Extracting JSON Values with JsonPath. In *Java XML and JSON—Document Processing for Java SE*, 2nd ed.; Apress: Berkeley, CA, USA, 2019; pp. 299–322.
9. Hu, Z.; Yan, L. Modeling Temporal Information With JSON. In *Emerging Technologies and Applications in Data Modeling and Processing*; Ma, Z., Yan, L., Eds.; IGI Global: Hershey, PA, USA, 2019; pp. 134–153.
10. Irshad, L.; Ma, Z.; Yan, L. A Survey on JSON Data Stores. In *Emerging Technologies and Applications in Data Modeling and Processing*; Ma, Z., Yan, L., Eds.; IGI Global: Hershey, PA, USA, 2019; pp. 45–69.
11. Liu, Z.H. JSON Data Management in RDBMS. In *Emerging Technologies and Applications in Data Modeling and Processing*; Ma, Z., Yan, L., Eds.; IGI Global: Hershey, PA, USA, 2019; pp. 20–44.
12. Möller, M.L.; Berton, N.; Klettke, M.; Scherzinger, S.; Störl, U. jHound: Large-Scale Profiling of Open JSON Data. In Proceedings of the 18th Conference on Database systems for Business, Technology and Web (BTW'2019), Rostock, Germany, 4–8 March 2019; pp. 555–558.
13. Beyer, K.S.; Ercegovac, V.; Gemulla, R.; Balmin, A.; Eltabakh, M.Y.; Kanne, C.-C.; Özcan, F.; Shekita, E.J. Jaql: A scripting language for large scale semistructured data analysi. *PVLDB* **2011**, *4*, 1272–1283. [CrossRef]
14. Florescu, D.; Fourny, G. JSONiq: The History of a Query Language. *IEEE Internet Computing* **2013**, *17*, 86–90. [CrossRef]
15. Li, X.; Liu, Z.; Liu, M.; Wu, X.; Zhu, S. Towards a Pattern-Based Query Language for Document Databases. In Proceedings of the 20th International Conference on Database Systems for Advanced Applications (DASFAA'2015), Hanoi, Vietnam, 20–23 April 2015; pp. 320–325.
16. Ong, K.W.; Papakonstantinou, Y.; Vernoux, R. The SQL++ Query Language: Configurable, Unifying and Semi-structured. CoRR abs/1405.3631. 2015. Available online: https://arxiv.org/pdf/1405.3631.pdf (accessed on 31 December 2021).
17. Chamberlin, D. *SQL++ for SQL Users: A Tutorial*; Couchbase Inc.: Santa Clara, CA, USA, 2018.
18. Bourhis, P.; Reutter, J.L.; Vrgoč, D. JSON: Data model and query languages. *Inf. Syst.* **2020**, *89*, 101478. [CrossRef]
19. Pokorný, J. JSON Functionally. In Proceedings of the 24th European Conference on Advances in Databases and Information Systems (ADBIS 2020), Lyon, France, 25–27 September 2020; LNCS 12245. pp. 139–153.
20. Davoudian, A.; Chen, L.; Liu, M. A Survey on NoSQL Stores. *ACM Computing Surv.* **2018**, *51*, 40. [CrossRef]
21. Melton, J.; Zemke, F.; Hammerschmidt, B.; Kulkarni, K.; Liu, Z.H.; Michels, J.-E.; McMahon, D.; Özcan, F.; Pirahesh, H. SQL/JSON Part 1, DM32.2-2014-00024R1, 6 March 2014. Available online: https://www.wiscorp.com/pub/DM32.2-2014-00024R1_JSON-SQL-Proposal-1.pdf (accessed on 31 December 2021).
22. Zemke, F.; Hammerschmidt, B.; Kulkarni, K.; Liu, Z.H.; McMahon, D.; Melton, J.; Michels, J.-E.; Özcan, F.; Pirahesh, H. SQL/JSON Part 2—Querying JSON, ANSI INCITS DM32.2-2014-00025r1, 4 March 2014. Available online: https://www.wiscorp.com/pub/DM32.2-2014-00025r1-sql-json-part-2.pdf (accessed on 31 December 2021).

23. 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. Available online: http://standards.iso.org/ittf/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip (accessed on 31 December 2021).
24. Petković, D. SQL/JSON Standard: Properties and Deficiencies. *Datenbank Spektrum* **2017**, *17*, 277–287. [CrossRef]
25. Gössner, S. JSONPath—Xpath for JSON, 21 February 2007. Available online: http://goessner.net/articles/JsonPath/ (accessed on 31 December 2021).
26. Brahmia, S.; Brahmia, Z.; Grandi, F.; Bouaziz, R. Temporal JSON Schema Versioning in the τJSchema Framework. *J. Digit. Inf. Manag.* **2017**, *15*, 179–202.
27. Brahmia, Z.; Brahmia, S.; Grandi, F.; Bouaziz, R. Implicit JSON Schema Versioning Driven by Big Data Evolution in the τJSchema Framework. In *Big Data and Networks Technologies—BDNT 2019*; Farhaoui, Y., Ed.; LNNS; Springer: Berlin/Heidelberg, Germany, 2020; Volume 81, pp. 22–35.
28. Van Rossum, G.; Drake, F.L., Jr. *Python Reference Manual*; Centrum voor Wiskunde en Informatica: Amsterdam, The Netherlands, 1995.
29. Grandi, F.; Scalas, M.R.; Tiberio, P. A History-oriented Temporal SQL Extension. In Proceedings of the 2nd International Workshop on Next Generation Information Technologies and Systems (NGITS'95), Naharia, Israel, 27–29 June 1995; pp. 42–48.
30. Chasseur, C.; Li, Y.; Patel, J.M. Enabling JSON Document Stores in Relational Systems. In Proceedings of the 16th International Workshop on the Web and Databases (WebDB 2013), New York, NY, USA, 23 June 2013; pp. 1–6.
31. Liu, Z.H.; Hammerschmidt, B.; McMahon, D. JSON data management: Supporting schema-less development in RDBMS. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014), Snowbird, UT, USA, 22–27 June 2014; pp. 1247–1258.
32. Liu, Z.H.; Hammerschmidt, B.; McMahon, D.; Liu, Y.; Chang, H.J. Closing the functional and Performance Gap between SQL and NoSQL. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD 2016), San Francisco, CA, USA, 26 June–1 July 2016; pp. 227–238.
33. Liu, Z.H.; Gawlick, D. Management of Flexible Schema Data in RDBMSs—Opportunities and Limitations for NoSQL. In Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR'2015), Asilomar, CA, USA, 4–7 January 2015. Paper 5.
34. Internet Engineering Task Force. JavaScript Object Notation (JSON) Patch, Internet Standards Track Document, April 2013. Available online: https://tools.ietf.org/html/rfc6902 (accessed on 31 December 2021).
35. Brahmia, Z.; Grandi, F.; Oliboni, B.; Bouaziz, R. High-level Operations for Creation and Maintenance of Temporal and Conventional Schema in the τXSchema Framework. In Proceedings of the 21st International Symposium on Temporal Representation and Reasoning (TIME'2014), Verona, Italy, 8–10 September 2014; pp. 101–110.
36. Snodgrass, R.T.; Dyreson, C.E.; Currim, F.; Currim, S.; Joshi, S. Validating Quicksand: Schema Versioning in τXSchema. *Data Knowl. Eng.* **2008**, *65*, 223–242. [CrossRef]
37. Brahmia, Z.; Grandi, F.; Oliboni, B.; Bouaziz, R. Schema Change Operations for Full Support of Schema Versioning in the τXSchema Framework. *Int. J. Inf. Technol. Web Eng.* **2014**, *9*, 20–46. [CrossRef]
38. Brahmia, Z.; Grandi, F.; Oliboni, B.; Bouaziz, R. Supporting Structural Evolution of Data in Web-Based Systems via Schema Versioning in the τXSchema Framework. In *Handbook of Research on Contemporary Perspectives on Web-Based Systems*; Elçi, A., Ed.; IGI Global: Hershey, PA, USA, 2018; pp. 271–307.
39. W3C, Extensible Markup Language (XML) 1.0 (5th Edition), W3C Recommendation, 26 November 2008. Available online: http://www.w3.org/TR/2008/REC-xml-20081126/ (accessed on 31 December 2021).
40. W3C, XQuery 1.0: An XML Query Language (2nd Edition), W3C Recommendation, 14 December 2010. Available online: http://www.w3.org/TR/2010/REC-xquery-20101214/ (accessed on 31 December 2021).
41. W3C, XQuery Update Facility 1.0, W3C Recommendation, 17 March 2011. Available online: http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/ (accessed on 31 December 2021).
42. Laux, A.; Martin, L. XUpdate—XML Update Language, XML:DB Working Draft, 14 September 2000. Available online: http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html (accessed on 31 December 2021).
43. Obasanjo, D.; Navathe, S.B. A Proposal for an XML Data Definition and Manipulation Language. In Proceedings of the 1st VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT'2002), Hong Kong, China, 19 August 2002; pp. 1–21.
44. Sur, G.M.; Hammer, J.; Siméon, J. An XQuery-Based Language for Processing Updates in XML. In Proceedings of the 2nd Workshop on Programming Language Technologies for XML (PLAN-X'2004), Venice, Italy, 13 January 2004; pp. 40–53. Available online: http://www.brics.dk/NS/03/4/BRICS-NS-03-4.pdf (accessed on 31 December 2021).
45. Wang, G.-R.; Zhang, X.-L. Declarative XML Update Language Based on a Higher Data Model. *J. Comput. Sci. Technol.* **2005**, *20*, 373–377. [CrossRef]
46. Ni, W.; Ling, T.W. Update XML data by using graphical languages. In Proceedings of the Tutorials, Posters, Panels and Industrial Contributions at the 26th International Conference on Concep-tual Modeling (ER'2007), Auckland, New Zealand, 5–9 November 2007; pp. 209–214.
47. Cheney, J. FLUX: Functional Updates for XML. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, BC, Canada, 22–24 September 2008; pp. 3–14.

48. W3C. XQuery Update Facility 3.0, W3C Working Group Note, 24 January 2017. Available online: https://www.w3.org/TR/2017/NOTE-xquery-update-30-20170124/ (accessed on 31 December 2021).
49. Brahmia, Z.; Hamrouni, H.; Bouaziz, R. XML Data Manipulation in Conventional and Temporal XML Databases: A Survey. *Comput. Sci. Rev.* **2020**, *36*, 100231. [CrossRef]
50. Brahmia, Z.; Grandi, F.; Bouaziz, R. τXUF: A Temporal Extension of the XQuery Update Facility Language for the τXSchema Framework. In Proceedings 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.
51. Brahmia, Z.; Brahmia, S.; Grandi, F.; Bouaziz, R. Implementation of High-Level JSON Schema Change Operations Using JUpdate. In *Marketing and Smart Technologies: Proceedings of ICMarkTech'2020 (SIST vol. 205)*; Rocha, Á., Reis, J.L., Peter, M.K., Cayolla, R., Loureiro, S., Bogdanović, Z., Eds.; Springer Nature: Singapore, 2021; pp. 255–265.
52. Brahmia, Z.; Brahmia, S.; Grandi, F.; Bouaziz, R. Versioning schemas of JSON-based conventional and temporal big data through high-level operations in the τJSchema framework. *Int. J. Cloud Comput.* **2021**, *10*, 442–479. [CrossRef]
53. Internet Engineering Task Force (IETF). JSON Schema: A Media Type for Describing JSON Documents. Internet-Draft, 8 December 2020. Available online: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00 (accessed on 31 December 2021).
54. Brahmia, S.; Brahmia, Z.; Grandi, F.; Bouaziz, R. τJSchema: A Framework for Managing Temporal JSON-Based NoSQL Databases. In Proceedings of the 27th International Conference on Database and Expert Systems Applications (DEXA'2016), Porto, Portugal, 5–8 September 2016; Part 2. pp. 167–181.