Temporal JSON

1st Aayush Goyal
Dept. of Computer Science
Utah State University
Logan, Utah, USA
aayushgoyal9507@gmail.com

2nd Curtis Dyreson
Dept. of Computer Science
Utah State University
Logan, Utah, USA
curtis.dyreson@usu.edu

***Abstract*—Web services are the primary suppliers of data on the web, such data is typically formatted using JavaScript Object Notation (JSON) and only the current JSON snapshot is available from a service. But data evolves over time as it is added to, modified, or reduced. Providing an historical view of data is important in many applications. This paper describes how to capture the evolving history of a JSON document and how to support temporal queries. Our approach is to model temporal JSON as a virtual document in which time metadata is mixed with JSON data. The time metadata records when the JSON data is alive. We also describe how to navigate to data within the virtual document. The primary technical contribution of the paper is to show how to represent the virtual model in JSON and how to map temporal path expressions to the representational model. Experiments show that our model is efficient.**

***Index Terms*—temporal databases, web services, versioning**

I. Introduction

Web applications increasingly rely on web services to read and write to a database. There are many tools for constructing and documenting web services, such as Swagger, but there are no technologies for *temporal web services*. A temporal web service is a service that provides a *temporal* view of data, that is, a view of not only the current data, but past data or how the data has changed over time.

There has been extensive previous research to supporting temporal data [1]–[3]. This research has fallen into two broad categories: versioning and timestamp-based support. Timestamp-based queries are common in temporal relational databases. A temporal relational database [4] stores data that is annotated with time metadata. The time metadata records when the data was alive in some time domain, *e.g.,* transaction time [5], valid time [6], or both. Such databases can be queried in various ways. For instance in TSQL2 [7] a query can be evaluated to retrieve the data's history *e.g.,* a timeslice query [8], or retrieve the data as of some time instant, *e.g.,* a snapshot query [9], or perform a query at every time instant in the data's history, *e.g.,* a sequenced query [10]. But TSQL2 does not support queries that ask for versions of data, *e.g.,* get the second version of an employment record or retrieve the changes to the employment record. Data versioning is more common in temporal object-oriented databases [11] or temporal documents where each edit or change creates a new version of an object or document. Users can navigate among the versions and restore old versions if necessary.

Semi-structured data formats such as JSON, XML, and YAML are used to represent both data and documents and thus need to support both timestamp and version histories [12]–[17]. Semi-structured data changes over time, sometimes frequently, as new data is inserted and existing data is edited and deleted [18]–[20]. Previous research in temporal XML and JSON called elements that maintain their identity over time *items* [21]–[23]. Items are timestamped with a lifetime. Each change to an item creates a version, which is also timestamped. Previous research showed how to represent, query, describe with a schema and validate temporal semi-structured data. Differences in XML and JSON spawned further research in schema validation and versioning for JSON data [24].

JSON differs from XML in some key aspects. First, XML is a document representation so the ordering of elements is important. JSON is undordered. Second, JSON has an array type that XML lacks. Third, XML allows multiple subelements with the same name, in a JSON key/value set only unique keys are allowed. Fourth, JSON is integrated into many scripting languages, such as JavaScript, whereas XML has a separate query language, *i.e.,* XQuery and XPath. JSON is used to represent objects, swizzling and unswizzling of JSON is natively supported, and object path expressions, *i.e.,* "dot" notation, are used to navigate within a JSON document as well as an object hierarchy.

This paper proposes a new way to represent temporal JSON documents that better supports path expressions. We make the following contributions.

- We model JSON at three levels: snapshot, temporal, and representational. The snapshot model is a sequence of timestamped JSON documents. The temporal model glues the snapshots together to provide a temporal and version history for each value in a document. Finally, the representational model is how the temporal model is represented in JSON itself. JSON representation is important to providing backwards-compatibility with existing web service and JSON technology. If temporal JSON can be represented in JSON then it can be sent and received by web services, and it can be tightly integrated with scripting languages, just as JSON itself is. The representational model is an adaptation of the representational model for temporal XML developed by Currrim et. al [22].
- We describe operations for temporal JSON, namely, se-

quenced path lookup, time snapshot, version snapshot, time slice, and version slice.
- We implement temporal JSON in Python and extensively evaluate the implementation with experiments that empirically measure the cost of the temporal operations on the representational model.

In sum, we provide a complete description of temporal JSON and in future work describe potential improvements.

This paper is organized as follows. The next section is an example of temporal JSON. We then describe the three models. An evaluation of the implementation is followed by conclusions and a discussion of future work.

II. Example

Assume that data on the specimen Hawkweed (*Hieracium umbellatum*) is described in a JSON data collection called `specimen.json` as shown in Figure 1. The collection also has information about taxonomic authority, which is `unknown` in 2015.

```
{
 "specimen": {
  "name": "Hieracium umbellatum",
  "colloquial": "Hawkweed"
 },
 "taxaAuthority": {
  "author": "Unknown"
 }
}
```

Figure 1: The file `specimen.json` in 2015

In subsequent months, there is new scientific data about *Hieracium umbellatum*. In 2016 it was learned that *Hieracium umbellatum* was identified first by `Barkworth`. The value of the `author` field was updated creating a new version of the data, as shown in Figure 2. Additionally the `habitat` for the specimen was described.

```
{
 "specimen": {
  "name": "Hieracium umbellatum",
  "colloquial": "Hawkweed",
  "habitat": ["shoreline", "forest"]
 },
 "taxaAuthority": {
  "author": "Barkworth"
 }
}
```

Figure 2: *Hieracium umbellatum* identified by Barkworth, as of 2016

In 2018, the specimen description became more specific, relating `Hieracium umbellatum` to `Narrowleaf Hawkweed` so additional text was added to the `colloquial` field as shown in Figure 3, and the `habitat` was further clarified.

Researchers would like to learn of changes to the *Hieracium umbellatum* data over time. Hence it is important to capture the entire history of the data. Figure 4 shows the history of the *Hieracium umbellatum* data. The `specimenItem` is shown

```
{
 "specimen": {
  "name": "Hieracium umbellatum",
  "colloquial": "Hawkweed, Narrowleaf Hawkweed",
  "habitat": ["shoreline", "forest", "sand"]
 },
 "taxaAuthority": {
  "author": "Barkworth"
 }
}
```

Figure 3: *Hieracium umbellatum* is related to *Narrowleaf Hawkweed*, as of 2018

in Figure 5 while the `taxaAuthorityItem` is shown in Figure 6. The data lists `specimen` and `taxaAuthority` *items*. An item is a datum that retains its temporal identity through changes to the data. Each `specimenItem` has a `nameItem`, a `colloquialItem` and a `habitatItem`. Each item has an associated timestamp that indicates the version at each point in time. A new version of the item is created each time the item changes.

```
{
  "specimenItem": {...}
  "taxaAuthorityItem" : {...}
}
```

Figure 4: Temporal JSON data, the specimen item is shown in Figure 5

III. Models

Temporal JSON is modeled at three levels: *snapshot*, *temporal*, and *representational*. The snapshot model is the sequence of snapshots, or non-temporal JSON documents, that represent the document at a particular moment in time. The temporal model is how the document's history is made available to a user. The representational model is how the document is transported or stored. Importantly we believe that the representational model should be a JSON-based model to ensure compatibility with web services that provide data in JSON. So both the snapshot and representational models use JSON, while the temporal model uses temporal JSON. This section describes the three models, starting with the snapshot model.

```
{
  "timestamp" : "2015-2018",
  "specimenVersions": [{...specimen version 1...},
                       {...specimen version 2...}]
}
```

Figure 5: The `specimenItem`

```
{
  "timestamp" : "2015-2018",
  "taxaAuthorityVersions":
     [{...taxaAuthority version 1...},
      {...taxaAuthority version 2...}]
}
```

Figure 6: The `taxaAuthorityItem`

```
{
 "timestamp": "2015-2015",
 "data": {
  "specimen": {
   "nameItem": {
    "timestamp": "2015-2015",
    "nameVersions": [{
     "timestamp": "2015-2015",
     "data": {
      "name": "Hieracium umbellatum"
     }
    }]
   },
   "colloquialItem": {
    "timestamp": "2015-2015",
    "colloquialVersions": [{
     "timestamp": "2015-2015",
     "data": {
      "colloquial": "Hawkweed"
     }
    }]
   }
  }
 }
}
```


Figure 7: The first element in the `specimenVersions` list

A. Snapshot Model

The snapshot model is built from a sequence of JSON documents.

Definition III.1 (JSON document)**.** *A JSON document, D, is a value that could be either*

- *a literal, that is a string, number or boolean,*
- *a set of key/value pairs, $\{(k_1, v_1), \ldots, (k_n, v_n)\}$, where each key, k_i, is a string and v_i is a value,*
- *or an array, $[v_1, \ldots, v_m]$ where each v_i is a value.*

The snapshot model is a set of JSON documents, where each document is paired with a timestamp.

Definition III.2 (Snapshot Model)**.** *A snapshot model, $M_S(D)$, of an evolving document, D, is a set of documents from time 0 to time n: $\{(D_0, 0), \ldots, (D_n, n)\}$, where document D_i represents the JSON document at time i.*

A snapshot model support two operations: snapshot and path lookup within a snapshot.

Definition III.3 (Snapshot)**.** *Let $\mathcal{S}$ be the snapshot operator.*

$$\mathcal{S}(M_S(D), t) = (D_t, t) \in M_S(D)$$

Path lookup uses a *path expression*.

Definition III.4 (Path Lookup)**.** *Let $\mathcal{P}$ be the lookup operator, p be a path expression, and D be a JSON document. A path expression consists of three forms.*

- *$k.\alpha$ where α is a path expression and k is a string then $\mathcal{P}(D, k.\alpha) = \mathcal{P}(v, \alpha)$ if $(k, v) \in D$ else* **nil**.
- *ϵ where ϵ is the empty string, then $\mathcal{P}(D, k) = D$.*
- *$k[i].\alpha$ where the value of k is a list, then $\mathcal{P}(D, k[i].\alpha) = \mathcal{P}(v, \alpha)$ if $(k, a) \in D \ \wedge \ v = a[i]$ else* **nil**.


```
{
 "timestamp": "2016-2018",
 "data": {
  "specimen": {
   "nameItem": {
    "timestamp": "2016-2018",
    "nameVersions": [{
     "timestamp": "2016-2018",
     "data": {
      "name": "Hieracium umbellatum"
     }
    }]
   },
   "colloquialItem": {
    "timestamp": "2016-2018",
    "colloquialVersions": [{
      "timestamp": "2016-2017",
      "data": {
       "colloquial": "Hawkweed"
      }
     },
     {
      "timestamp": "2018-2018",
      "data": {
       "colloquial": "Hawkweed, Narrowleaf Hawkweed"
      }
     }
    ]
   },
   "habitatItem": {
    "timestamp": "2016-2018",
    "habitatVersions": [{
      "timestamp": "2016-2017",
      "data": {
       "habitat": ["shoreline", "forest"]
      }
     },
     {
      "timestamp": "2018-2018",
      "data": {
       "habitat": ["shoreline", "forest", "sand"]
      }
     }
    ]
   }
  }
 }
}
```


Figure 8: Second element in the `specimenVersions` list

B. Temporal Model

A temporal model captures the evolving history of a JSON document as a sequence of snapshots and versions. An important part of the temporal model is time metadata that records when each part of the document is live is some temporal dimension. The time metadata is a timestamp with times taken from several kinds of clocks.

There are two widely-accepted kinds of time in a temporal data collection: *valid time* and *transaction time*. The valid time represents real-world time, while the transaction time is when the data was current in the database, *i.e.,* the time between being inserted and deleted. As an example the valid time of a person's birth might be January 1, 2019, while the transaction time would be when that fact was entered into a data collection on March 17, 2019 until the time it is deleted (if it has not been deleted it is still current). In the paper we record only

the transaction time.

To record versions we introduce *version clocks*. Each change to a value in a JSON document creates a new version of the value. A version clock records when (in transaction time) the version was current. As an example suppose that a version clock records two versions, the first started at time 4 and ended at time 7, and the second was current from time 8 to time 10, then the version clock would be $[4, 7, 8, 10]$.

The clocks form a timestamp.

Definition III.5 (Timestamp)**.** *A timestamp is a pair, (i, t), such that i is the clock identifier and t is a clock measurement.*

We will use two clocks, a version clock, with identifier *vc*, and a parent's version clock, with identifier *pc*.

Temporal JSON glues information from a sequence of snapshots into a meaningful history of times and versions. Each change to a value produces a new version of that value. The kinds of changes depend on the type of the value.

- If the value is a literal, that is a string, number or Boolean, then there is only one version of the value (literals are not versioned).
- If the value is a set of key/value pairs, $\{(k_1, v_1), \ldots, (k_n, v_n)\}$, where each key, k_i, is a string and v_i is a value, then any change to the set (insertion or deletion of a pair) creates a new version of the set and any change to a v_i creates a new version of the pair.
- If the value is an array, $[v_1, \ldots, v_m]$ where each v_i is a value then the array is modeled as a set of key/value pairs by using the array index as the key, *e.g.,* key "0" is paired with the first array value, and versioning is applied as if the array were a set of key/value pairs.

A temporal JSON model is based on a snapshot model.

Definition III.6 (Temporal model)**.** *For a snapshot model, D^S, let*

- *P be the set of valid path expressions across all of the snapshots, that is,*
  $$P = \{(p, k) \mid \exists k, p\ [D_k \in D^S\ \wedge\ \mathcal{P}(D_k, p)\ \neq\ \text{nil}]\},$$
- *L be the set of valid literals prefixed with path expressions,*
  $$L = \{(p.z, k) \mid \exists k, p, z\ [D_k \in D^S\ \wedge\ \mathcal{P}(D_k, p)\ =\ z \wedge\ z \text{ is a literal}]\},$$
- *Z be the union of L and P, $Z = P \bigcup L$,*
- *$\mathcal{T}_v$ be the transaction time function that determines a transaction time timestamp,*
  $$\mathcal{T}_v(p, Z) = \{[\ldots, [t_i, t_{i+1}], \ldots] \mid (p, t_i - 1) \notin Z \wedge (p, t_{i+1} + 1) \notin Z \wedge \forall t_i \leq k \leq t_{i+1}[(p, k)\ \in\ Z]\},$$
- *$F(p)$ be the longest prefix function, $F(a_1 \ldots . a_{n-1}, a_n = a_1 \ldots . a_{n-1}$*
- *$C(p, t)$ be the set of children of p at time t,*
  $$C(p, t) = \{x \mid F(x) = p\ \wedge\ (x, t)\ \in\ Z\},$$
- *and $\mathcal{T}_c$ be the version clock function that determines a child version clock timestamp.*

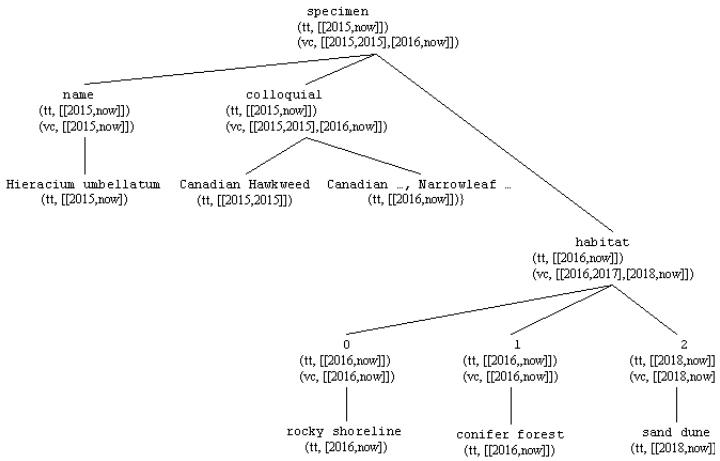

Figure 9: The history of the `specimen`

$$\mathcal{T}_c(p, Z) = \{[\ldots, [t_i, t_{i+1}], \ldots] \mid C(p, t_i - 1) \neq C(p, t_i) \wedge C(p, t_{i+1} + 1) \neq C(p, t_{i+1}) \wedge \forall t_i \leq k < t_{i+1}[C(p, k)\ =\ C(p, k + 1)]\}.$$

Then a temporal model, D^T, is a graph, (V, E), where

- $V = \{(p, \mathcal{T}_v(p, Z), \mathcal{T}_c(p, z)) \mid \exists t[(p, t) \in Z]\}$
- $E = \{(v, w) \mid v, w \in V \wedge v = (p, \_, \_) \wedge w = (c, \_, \_) \wedge F(c) = p\}$

As an example consider the representation of the temporal model shown in Figure 9. The model is the history of the snapshots shown in Figures 1 through 3. To save space in the figure just the `specimen` value is shown. In the figure the timestamps are shown below each value. Each timestamp has a transaction time (clock id "tt") or a version time (clock id "vc"). The version clock for a value represents its versions. For instance, consider the `habitat` value. It was added to the `specimen` key/value set in 2016, so the version clock for `specimen` has two timestamps, indicating two versions, and the transaction time of `habitat` places it in the second version. Since literals have only one version, we show only the transaction time, which is the same as the version time.

There are three things to note about the temporal model. First, it is implicitly coalesced [25]. A model would not be coalesced if there existed siblings in the graph that represented the same value. But this is not possible since each node has a unique identity, specified by its path. Second, each timestamp is a temporal element [26], *i.e.,* a set of temporal intervals. The timestamp functions in the temporal model definition stipulate that the timestamps must be maximal (periods in the set cannot be temporally adjacent). Third, the model is not JSON. A key in a key/value pair may be connected to more than one value (over time). For instance, the `colloquial` key has two children (each of which is a literal).

A temporal model has several operators.

Temporal path lookup uses a path expression to navigate within a temporal model to a specified value. The path

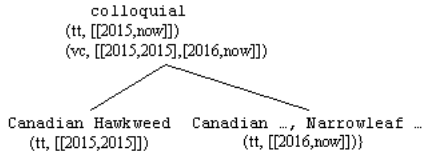

Figure 10: Path lookup for `specimen.colloquial`

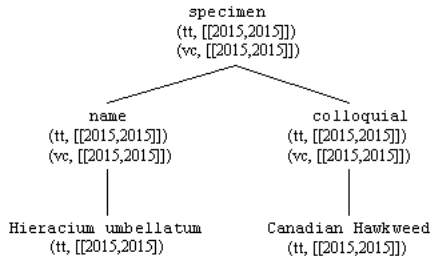

Figure 11: Time slice of 2015 on the model in Figure 9

expression is sequenced [10], that is, it navigates paths in all the snapshots simultaneously.

Definition III.7 (Temporal path Lookup)**.** *Let $\mathcal{P}^T$ be the temporal lookup operator, p be a path expression, and D^T be a temporal model. Then $\mathcal{P}^T(D^T, p) = (V_T, E_T)$ where*

- $V_T = \{(s, x, y) \mid s = p \ \vee \ p \text{ is a prefix of } s\}$*, and*
- $E_t = \{(v, w) \mid v, w \in V_t \ \wedge \ (v, w) \in E\}$

As an example, if D^T is the temporal model shown in Figure 9 then $\mathcal{P}^T(D^T, \texttt{specimen.colloquial})$ yields the model shown in Figure 10.

The time slice operator slices the model at a specific time returning a temporal model restricted to the give time.

Definition III.8 (Time Slice)**.** *Let $\mathcal{T}^T$ be the temporal time slice operator, $D^T = (V, E)$ be a temporal model, and t be a timestamp, then $\mathcal{T}^T(D_T, t) = (V_t, E_t)$ where*

- $V_t = \{(p, x \bigcap t, y \bigcap t) \mid (p, x, y) \in V \ \wedge \ x \bigcap t \neq \emptyset \ \wedge \ y \bigcap t \neq \emptyset\}$*, and*
- $E_t = \{(v, w) \mid v, w \in V_t \ \wedge \ (v, w) \in E\}$

As an example, if D^T is the temporal model shown in Figure 9 then $\mathcal{T}^T(D^T, 2015)$ yields the model shown in Figure 11.

A version slice uses a path expression to navigate to a value, and then uses time slice on the times for that version.

Definition III.9 (Version slice)**.** *Let $\mathcal{V}^T$ be the version slice operator, $D^T = (V, E)$ be a temporal model, $E^T(D^T, n)$ be the n^{th} timestamp in the version clock time of the root value in the temporal model, and n be a version number, then $\mathcal{V}^T(D_T, n) = \mathcal{T}^T(D^T, E(D^T, n))$.*

As an example, if D^T is the temporal model shown in Figure 9 then $\mathcal{V}^T(D^T, 1)$ yields the model shown in Figure 12 (note that the version numbers start at 0).

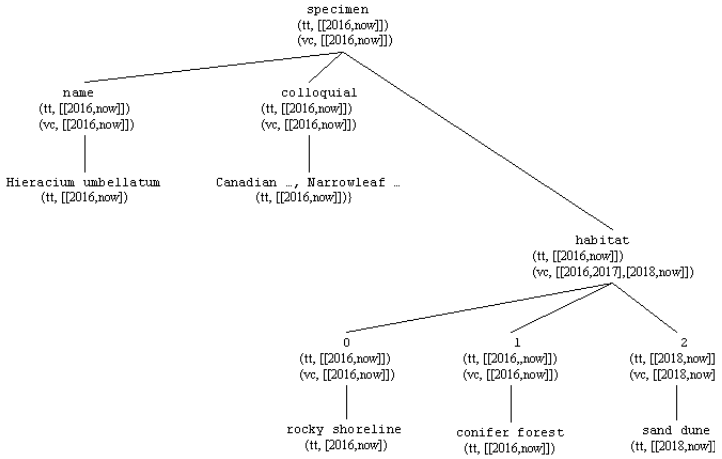

Figure 12: The second version of `specimen`

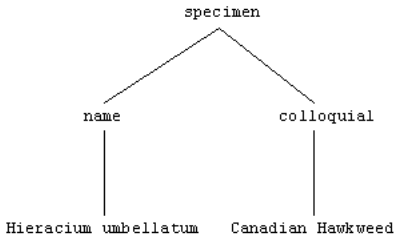

Figure 13: Result of a time snapshot of 2015 on the model in Figure 9, which corresponds to the `specimen` in the JSON of Figure 1

The snapshot operators produce a snapshot JSON document.

Definition III.10 (Time snapshot)**.** *Let $\mathcal{T}^S$ be the time snapshot operator, D^T be a temporal model, t be a time, and $\mathcal{S}(\mathcal{F}^T)$ be a function that removes the timestamps from each node in a temporal model. Then $\mathcal{T}^S(D^t, t) = \mathcal{S}(\mathcal{T}^T(D^t, t))$.*

Note that the operator selects a temporal model as of specific time. The model represents a snapshot, but has extra timestamps that are stripped to get the snapshot. As an example, if D^T is the temporal model shown in Figure 9 then $\mathcal{T}^S(D^T, 2015)$ yields the model shown in Figure 13, which corresponds to the JSON of Figure 1.

Version snapshot is similar to time snapshot.

Definition III.11 (Version snapshot)**.** *Let $\mathcal{V}^S$ be the version snapshot operator, D^T be a temporal model, $E^T(D^T, n)$ be the n^{th} timestamp in the version clock time of the root value in the temporal model, and n be a version number, then $\mathcal{V}^T(D_T, n) = \mathcal{T}^S(D^T, E(D^T, n))$.*

As an example, the first version snapshot of `specimen`, $\mathcal{V}^S(D^T, 0)$, gives the same result as the time snapshot at 2015 as shown in Figure 13.

Path expressions in the temporal model can be extended

to support the temporal operators. We describe a denotational semantics for temporal path expressions. Let $[\![\ ]\!]^T[D^t]$ denote the temporal semantic function applied to temporal model D^T and $[\![\ ]\!][D]$ denote the snapshot semantic function applied to JSON document D. We assume that the snapshot semantics of path expressions is given.

Dot notation
$[\![\texttt{x}.\alpha]\!]^T[D^T] \equiv [\![\alpha]\!]^T[\mathcal{P}^T(D^T, \texttt{x})]$
Time slice
$[\![\texttt{timeSlice}(t).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!]^T[\mathcal{T}^T(D^T, t)]$
Version slice
$[\![\texttt{versionSlice}(n).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!]^T[\mathcal{V}^T(D^T, n)]$
Time snapshot
$[\![\texttt{timeSnapshot}(t).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!][\mathcal{T}^S(D^T, t)]$
The current snapshot has special syntax
$[\![\texttt{current}().\alpha]\!]^T[D^T] \equiv [\![\alpha]\!][\mathcal{T}^S(D^T, \texttt{now})]$
Version snapshot
$[\![\texttt{versionSnapshot}(n).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!][\mathcal{V}^S(D^T, t)]$

For example, `specimen.name.versionSlice[0]` would be translated as follows.

$[\![\texttt{specimen.name.versionSlice(0)}]\!]^T[D^T] \equiv$
$[\![\texttt{name.versionSlice(0)}]\!]^T[\mathcal{P}^T(D^T, \texttt{specimen})]$
$[\![\texttt{versionSlice(0)}]\!]^T[\mathcal{P}^T(\mathcal{P}^T(D^T, \texttt{specimen}), \texttt{name})]$
$V^T(\mathcal{P}^T(\mathcal{P}^T(D^T, \texttt{specimen}), \texttt{name}), 0)$

C. Representational Model

The temporal model is the model for querying and managing a temporal JSON data collection. Though the model can be implemented in a straightforward manner using a graph data structure, the goal of our research is to use JSON to represent the temporal model. The primary reason why we want to use JSON is for compatibility with existing technologies that utilize JSON. Web services send and receive JSON data. JSON is also tightly integrated in scripting languages such as JavaScript and Python. These languages have special libraries and syntax for parsing and navigating paths within a JSON document. Hence it is important to use JSON for the representational model in order to send temporal JSON through the web and integrate it into scripting languages.

The representational model is based on prior research in temporal XML [22]. That work referred to elements that retain their identity over time as *items*. Each change to an item produces a new *version* of the item.

We utilize the framework developed by Currim et. al to represent items and versions but adapt their framework from XML to JSON. In our representational model each key in a key/value set represents an item. Recall that we model arrays as key/value sets with the array index as the key. For instance, suppose we had key/value set $\{(k_1, v_1), \ldots, (k_n, v_n)\}$ then the items in the set would be represented as shown in Figure 14. The timestamp in the representation of an item is the transaction time timestamp.

Changes to a value (a key/value set, a literal, or an array) creates a new version of an item. The version representation

```
{
  k1Item: {
    timestamp:
    versions:[...]
  }
  k2Item: {...}
  ...
  knItem: {...}
}
```

Figure 14: Format of an key/value set as a set of itmes

```
{
  timestamp:
  data: {
    ki:...
  }
}
```

Figure 15: Format of an key/value set as a set of items

captures the version as shown in Figure 15. The timestamp in a version is a version clock timestamp.

The representational model has operations similar to the temporal model.

Path lookup has two changes from the snapshot model. First, a path expression must navigate through `Item` and `versions`. Consider for instance the expression `specimen.name`. This expression must first find the versions of the specimen item.

```
specimenItem.versions
```

Next, the array of versions must be traversed to access the corresponding `nameItems` as shown in the pseudo-code of Algorithm 1. The second change is that the resulting array of items must be *coalesced*. Coalescing merges versions that are the same. For example, when a new version of a new version of `specimen` was created because `habitat` was added, `name` did not change so same version is copied in the second version of `specimenItem`.

The snapshot function shows a non-temporal view of the document at a particular time or a particular version as requested by the user. As an example, `specimen.TimeSnapshot(2015)` would yield the snapshot shown in Figure 16. Note that an array of snapshots is returned with the time of each snapshot given by the `timestamp` key and the snapshot in the `data` key.

Version snapshot retrieves all the snapshots of particular version of the specimen. A version changes if a new item has been added or an older one has been deleted or modified to previous version. In our example, there is no `habitat` item in the first version of `specimen`, but in 2016 `habitat` is added which creates a second version of `specimen`. For example, `specimen.versionSnapshot(0)`, outputs all of the snapshots in the first version of `specimen`.

Algorithm 1: Path lookup in the representational model

Output: List K of Keys, Representational Document D^R

Output: List R of Items

Procedure pathLookup(Keys K, Doc D^R)

$h \leftarrow D^R.K$.pop().`Item`

if K is empty **then**
  return h
end

$R \leftarrow [\,]$

for $v \in h$.`versions` **do**
  R.append(pathLookup(K, h)
end

return R

```
[{
 "time": "2015-2015",
 "data": {
  "specimen": {
   "name": "Hieracium umbellatum",
   "colloquial": "Hawkweed"
  }
 }
}]
```

Figure 16: Time snapshot as of 2015

```
{
 "specimenItem": {
  "timestamp": "2015-2015",
  "specimenVersions": [{
   "timestamp": "2015-2015",
   "data": {
    "specimen": {
     "nameItem": {
      "timestamp": "2015-2015",
      "nameVersions": [{
       "timestamp": "2015-2015",
       "data": {
        "name": "Hieracium umbellatum"
       }
      }]
     },
     "colloquialItem": {
      "timestamp": "2015-2015",
      "colloquialVersions": [{
       "timestamp": "2015-2015",
       "data": {
        "colloquial": "Hawkweed"
       }
      }]
     }
    }
   }
  }]
 }

}
```

Figure 17: Version(0) Slice of `Specimen`

Slicing gives a temporal view of the document at a specific time or version. Unlike snapshot, which retrieves just the actual data at particular of particular time or version, slicing retrieves a temporal (representational) document. There are two types of slice operations namely *time slice* and *version slice*.

A version slice is used to obtain slice of a version with its temporal details. Unlike version snapshot which gives the version's data content without any temporal information, slicing gives all the data along with time information. There can be different versions of an item, a new version is created when an attribute is added, deleted or changed from previous version, *e.g.,* `specimen.versionSlice(0)` would output the JSON in Figure 17.

IV. Evaluation

This section presents the results of an empirical evaluation of the representational model. We perform several experiments to measure the cost of creating the representational model from a sequence of snapshot documents and to measure the cost of temporal operations, such as time snapshot.

A. Experimental Environment

We implemented our functions in Python 2.7. The implementation uses Python's inbuilt library for parsing JSON. The experiments were run on a MacBook Pro, 2.7GHz quad-core 8^{th}-generation Intel Core i7 processor, 16GB of RAM. Each experiment was performed in an machine dedicated to testing with background processes minimized.

B. Experiment: Representational Model Creation

The first experiment measures the impact of increasing data size on time taken to create the representational model. The experiment uses an online JSON generator to generate snapshots that are then combined to form a temporal document. There are mainly two types of temporal document that we created, one where we change key-value pairs for the outermost key (parent) which in turn creates different versions of that parent , and another where we change values of inner elements (child) which creates different versions of those children nodes and only single version of parent. We show the creation cost for both types of temporal documents.

1) Representational Model Creation as Parent Version Changes: The creation takes up to 30 seconds when new versions of parent are made as it leads to a larger document (up to 4Mb) depending on the number of key-value pairs. We change up to 30 key-values of the parent, with each new version of parent, all of the children are copied into another version which leads to higher complexity while reversing the snapshots. The results are plotted in Figure 18, which represents the cost of creation.

2) Representational Model Creation as Child Version Changes: The creation takes up to a few milliseconds when new versions of children are made and leads to smaller document size (up to 2Mb) and less complex JSON, depending on the number of key-value pairs. We change between 100 to 1000 key-values of children. The results are plotted in Figure 19, which represents the cost of document creation.

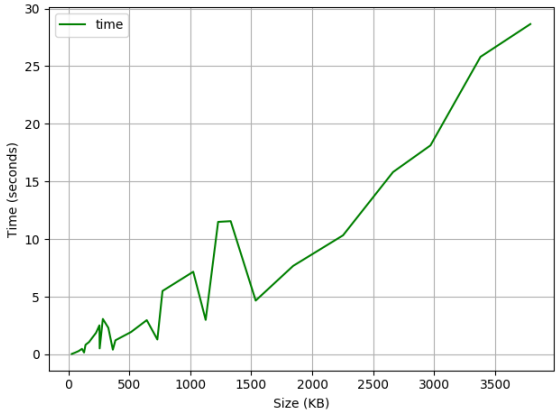

Figure 18: Representational model creation time with parent version changes

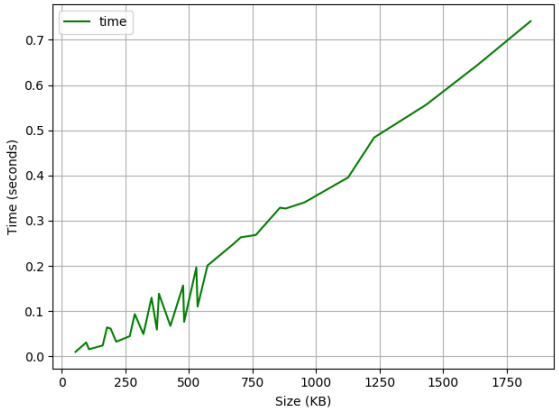

Figure 19: Representational model creation time with child version changes

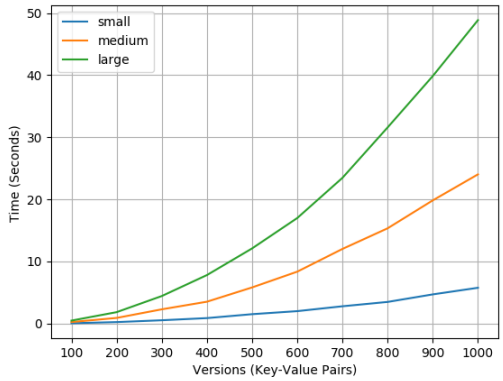

Figure 20: Time snapshot with new version of children

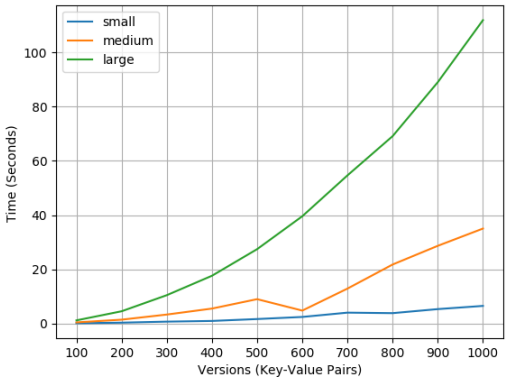

Figure 21: Time snapshot with new version of parent

C. Experiment: Time Snapshot

In this section, we study the cost of creating time snapshots with varying number of changes in key-value pairs (100 to 1000). The cost can vary on a number of factors such as levels of nesting, deeper the nesting, the higher time it will take to get that item recursively. We experiment with three item sizes: small, medium and large. Small items have two key-value pairs, medium have ten key-value pairs and large have thirty. We perform changes between 100 to 1000 in the increments of 100.

1) Time Snapshot with New Children Versions: We change the values of children keys which in turn creates new versions of children, *e.g.,* if we perform 100 changes then 100 new versions of each children key will be created. Figure 20 shows cost of creating time snapshots with respect to the number of changes/versions in children. This is less expensive operation as new versions of only children are created and there will only be one version of parent.

2) Time Snapshot with New Parent Versions: We change the values of parent which creates new versions of parent and also values of children which creates new versions of children as well, *e.g.,* if we perform 100 changes to parent values, then 100 new versions of it will be created and at the same time we also change values of children, which creates new versions of them as well. This is comparatively costly operation. Figure 21 shows how cost varies with changes.

D. Experiment: Time Slice

In this section we study the cost of the time slice operation. As this operation only needs to retrieve the slice of the document at a point in time, it simply iterates through the `JSON` and checks for the overlap with input timestamp. It does not need to go through all the child versions, only if the parent timestamp overlaps will it dig into its children versions. However, with changes in children versions, it has to go through all the child versions until it finds the overlap. The cost also depends upon the input timestamp itself, as a slice with an earlier timestamp with be retrieved faster than a later one. The performance is shown by Figures 22 and 23.

E. Experiment: Version Snapshot

Version snapshots are relatively fast.

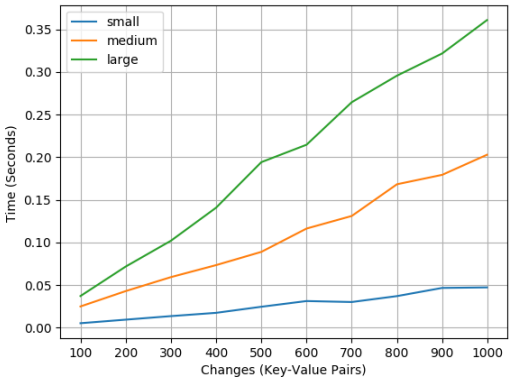

Figure 22: Time slice with new versions of children

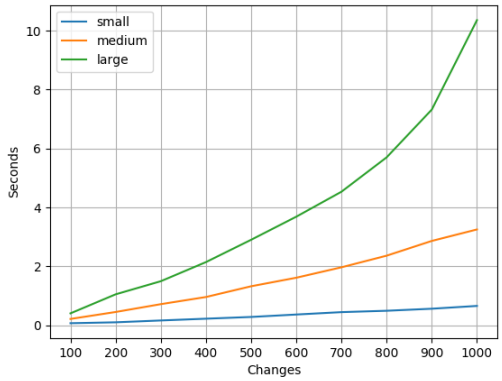

Figure 24: Version snapshot with new versions of children

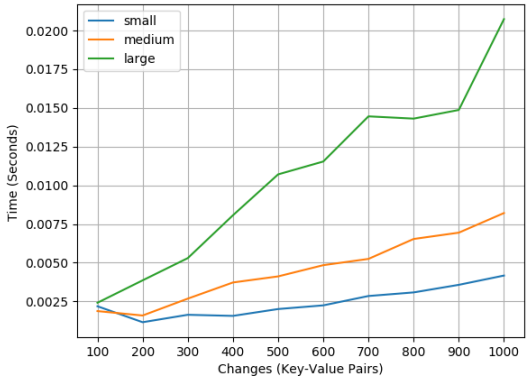

Figure 23: Time slice with new versions of parent

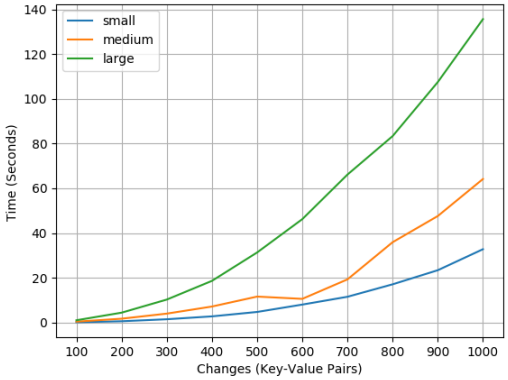

Figure 25: Version snapshot with new versions of parent

1) Version Snapshot with New Children Versions: We change the values of children keys which in turn creates new versions of children, *e.g.,* if we perform 100 changes then 100 new versions of each children key will be created. Figure 24 shows the cost of creating version snapshots with respect to number of changes/versions in children. Version Snapshot's run time depends on factors like levels of nesting and number of versions the requested item has. Grabbing the *Nth* version will be straightforward and requires no coalescing.

2) Version Snapshot with New Parent Versions: Here we change the values of parent which creates new versions of parent and also values of children which creates new versions of children as well, *e.g.,* if we perform 100 changes to parent values, then 100 new versions of it will be created and at the same time we also change values of children, which leads in new versions of them as well. This is comparatively costly operation as all the versions of children will be copied in next versions of parent, which results in higher coalescing time. Figure 25 shows how cost varies with changes.

F. Experiment: Version Slice

In this section we study the cost of creating version slices by varying number of changes in key-value pairs (100 to 1000). There are two types of changes: change in key-values of parent and change in values of children. Similar to version snapshot, version slice works slightly faster with changes in children version because of the coalescing. When parent version changes, same versions of children are copied across newer parent versions, which results in duplication, which adds up the coalescing time. The performance is shown by Figure 26 and Figure 27.

V. Conclusion and Future Work

JavaScript Object Notation (JSON) is a format for representing data. In this paper we show how to capture the history of changes to a JSON document. Capturing the history is important in many applications, where not only the current version of a document is required, but all the previous versions. Conceptually the history can be thought of as a sequence of non-temporal JSON documents, one for each instant of time. Each document in the sequence is called a snapshot. Since changes to a document are few and infrequent, the sequence of snapshots largely duplicates a document across many time instants, so the snapshot model is (wildly) inefficient in terms of space needed to represent the history and time taken to navigate within it. A more efficient representation can be

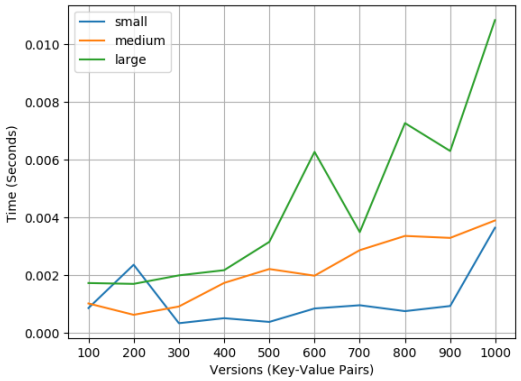

Figure 26: Version slice with new versions of children

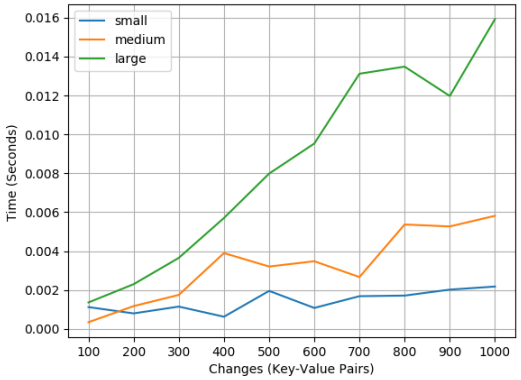

Figure 27: Version slice with new versions of parent

achieved by "gluing" the snapshots together to form a temporal model. Data that remains unchanged across snapshots is represented only once in a temporal model. But we show that the temporal model is not a JSON document, and it is important to represent a history as JSON to ensure compatibility with web services and scripting languages that use JSON. So we describe a representational model that captures the information in a temporal model. We implement the representational model in Python and extensively experiment with the model.

This paper makes the following contributions.

- We adapt the representational model for temporal XML developed by Currrim et. al [22] to JSON.
- We describe operations for the representation model for temporal JSON, namely, time snapshot, version snapshot, time slice, and version slice.
- We implement temporal JSON in Python and extensively evaluate the implementation with experiments that empirically measure the cost of the temporal operations on the representational model.

Future work consists of adding more temporal operations such as `delta()`, which gets the difference between two documents whether it be two snapshots or temporal documents, others being `next()` and `previous()`, which can used to retrieve next and previous versions of current version.

References

[1] O. Etzion, S. Jajodia, and S. M. Sripada, Eds., *Temporal Databases: Research and Practice*, ser. Lecture Notes in Computer Science, vol. 1399. Springer, 1998.

[2] V. Radhakrishna, P. V. Kumar, and V. Janaki, "A survey on temporal databases and data mining," in *ICEMIS*, 2015, pp. 52:1–52:6.

[3] A. Artale, R. Kontchakov, A. Kovtunova, V. Ryzhikov, F. Wolter, and M. Zakharyaschev, "Ontology-mediated query answering over temporal data: A survey (invited talk)," in *TIME*, 2017, pp. 1:1–1:37.

[4] C. S. Jensen and R. T. Snodgrass, "Temporal database," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[5] ——, "Transaction time," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[6] ——, "Valid time," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[7] R. T. Snodgrass, Ed., *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[8] C. S. Jensen and R. T. Snodgrass, "Timeslice operator," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[9] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner, "Transitioning temporal support in TSQL2 to SQL3," in *Temporal Databases: Research and Practice*, 1997, pp. 150–194.

[10] M. H. Böhlen and C. S. Jensen, "Sequenced semantics," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[11] C. Combi, "Temporal object-oriented databases," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[12] T. Amagasa, M. Yoshikawa, and S. Uemura, "A Data Model for Temporal XML Documents," in *DEXA*, 2000, pp. 334–344.

[13] S. S. Chawathe, S. Abiteboul, and J. Widom, "Representing and Querying Changes in Semistructured Data," in *ICDE*, 1998, pp. 4–13.

[14] S. Chien, V. J. Tsotras, and C. Zaniolo, "Efficient Schemes for Managing Multiversion XML Documents," *VLDB J.*, vol. 11, no. 4, pp. 332–353, 2002.

[15] F. Rizzolo and A. A. Vaisman, "Temporal XML: Modeling, Indexing, and Query Processing," *VLDB J.*, vol. 17, no. 5, pp. 1179–1212, 2008.

[16] C. E. Dyreson and F. Grandi, "Temporal XML," in *Encyclopedia of Database Systems*, 2009, pp. 3032–3035.

[17] F. Wang and C. Zaniolo, "An XML-Based Approach to Publishing and Querying the History of Databases," *World Wide Web*, vol. 8, no. 3, pp. 233–259, 2005.

[18] J. Cho and H. Garcia-Molina, "The Evolution of the Web and Implications for an Incremental Crawler," in *VLDB*, 2000, pp. 200–209.

[19] V. N. Padmanabhan and L. Qiu, "The Content and Access Dynamics of a Busy Web Site: Findings and Implications," in *SIGCOMM*, 2000, pp. 111–123.

[20] P. G. Ipeirotis, A. Ntoulas, J. Cho, and L. Gravano, "Modeling and Managing Content Changes in Text Databases," in *ICDE*, 2005, pp. 606–617.

[21] R. T. Snodgrass, C. E. Dyreson, F. Currim, S. Currim, and S. Joshi, "Validating Quicksand: Temporal Schema Versioning in tauXSchema," *Data Knowl. Eng.*, vol. 65, no. 2, pp. 223–242, 2008.

[22] F. Currim, S. Currim, C. E. Dyreson, R. T. Snodgrass, S. W. Thomas, and R. Zhang, "Adding Temporal Constraints to XML Schema," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 8, pp. 1361–1377, 2012.

[23] C. E. Dyreson and K. G. Mekala, "Prefix-Based Node Numbering for Temporal XML," in *WISE*, 2011, pp. 172–184.

[24] S. Brahmia, Z. Brahmia, F. Grandi, and R. Bouaziz, "τjschema: A framework for managing temporal json-based nosql databases," in *DEXA*, 2016, pp. 167–181.

[25] M. H. Böhlen, "Temporal coalescing," in *Encyclopedia of Database Systems, Second Edition*, 2018.

[26] C. S. Jensen and R. T. Snodgrass, "Temporal element," in *Encyclopedia of Database Systems, Second Edition*, 2018.