TEMPORAL JSON

by

Aayush Goyal

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Haitao Wang, Ph.D. Committee Member
Vladimir Kulyukin, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

ABSTRACT

Temporal JSON

by

Aayush Goyal, Master of Science

Utah State University, 2019

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

Web services are the primary suppliers of data on the web. Data supplied by a service 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 thesis describes how to capture the evolving history of a JSON document and how to support temporal queries on the history. 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 thesis 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.

(73 pages)

PUBLIC ABSTRACT

Temporal JSON

Aayush Goyal

JavaScript Object Notation (JSON) is a format for representing data. In this thesis 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 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. Our experiments show that the model is efficient.

To my family

ACKNOWLEDGMENTS

Foremost, I would like to express my gratitude to my advisor Dr. Curtis Dyreson, for the continuous support of my Master's study and research, for his patience, motivation, enthusiasm, and immense knowledge. The door to Dr. Dyreson's office was always open whenever I ran into a trouble spot or had a question about my research or writing. His guidance helped in all the time of research and writing of this thesis. I could not have imagined a better advisor and mentor for my Master's study.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Vladimir Kulyukin and Dr. Haitao Wang for their encouragement, insightful comments, and hard questions.

My sincere thanks also go to Genie Hanson, Vicki Anderson and Cora Price for providing constant help and support whenever I ran into departmental issues, these are the people who bind the Computer Science department at Utah State University together.

Last but not the least, I am extremely grateful to my family, my parents Manish Goyal and Jyoti Goyal and my brother Itish Goyal, for their love, prayers, and sacrifices for educating me and preparing me for my future.

Aayush Goyal

CONTENTS

LIST OF FIGURES

CHAPTER 1

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 representations 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 and as an element can be moved within a document. 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 thesis 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, time snapshot, version snapshot, time slice, and version slice, and we give algorithms for implementing the operations.

- 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 thesis is organized as follows. The next section is an example of temporal JSON. We then describe the three models and how they are implemented. The evaluation of the implementation is followed by conclusions and a discussion of future work.

CHAPTER 2
Example

Assume that data on the specimen Canadian Hawkweed (*Hieracium umbellatum*) is described in a JSON data collection called `specimen.json` as shown in Figure 2.1. The collection also has information about taxonomic authority, which is `unknown` in 2015.

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

Fig. 2.1: 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 2.2. Additionally the `habitat` for the specimen was described.

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

Fig. 2.2: *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 2.3, and the `habitat` was further clarified.

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

Fig. 2.3: *Hieracium umbellatum* is related to *Narrowleaf Hawkweed*, as of 2018

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 2.4 shows the history of the *Hieracium umbellatum* data. The `specimenItem` is shown in Figure 2.5 while the `taxaAuthorityItem` is shown in Figure 2.6. 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" : {...}
}
```

Fig. 2.4: Temporal JSON data, the specimen item is shown in Figure 2.5

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

Fig. 2.5: The `specimenItem`

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

Fig. 2.6: 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": "Canadian Hawkweed"
          }
        }]
      }
    }
  }
}
```

Fig. 2.7: The first element in the `specimenVersions` list

```
{
  "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": "Canadian Hawkweed"
            }
          },
          {
            "timestamp": "2018-2018",
            "data": {
              "colloquial": "Canadian Hawkweed, Narrowleaf Hawkweed"
            }
          }
        ]
      },
      "habitatItem": {
        "timestamp": "2016-2018",
        "habitatVersions": [{
            "timestamp": "2016-2017",
            "data": {
              "habitat": ["rocky shoreline", "conifer forest"]
            }
          },
          {
            "timestamp": "2018-2018",
            "data": {
              "habitat": ["rocky shoreline", "conifer forest", "sand dune"]
            }
          }
        ]
      }
    }
  }
}
```

Fig. 2.8: The second element in the `specimenVersions` list

CHAPTER 3
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.

3.1 Snapshot Model

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

Definition 3.1.1 (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, $\{(k_1, v_1), \dots, (k_n, v_n)\}$, where each key, k_i, is a string and v_i is a value,*
- *or an array, $[v_1, \dots, v_m]$ 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 3.1.2 (Snapshot Model) *A snapshot model, $M_S(D)$, of an evolving JSON document, D, is a set of JSON documents from time 0 to time n: $\{(D_0, 0), \dots, (D_n, n)\}$, 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 3.1.3 (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 3.1.4 (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**.

3.2 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 thesis 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 3.2.1 (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 3.2.2 (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,*

$$\begin{aligned}\mathcal{T}_v(p,Z) = \{[\ldots,[t_i,t_{i+1}],\ldots] \mid\ & (p,t_i-1) \notin Z \\ & \wedge\ (p,t_{i+1}+1) \notin Z \\ & \wedge\ \forall t_i \leq k \leq t_{i+1}[(p,k)\ \in\ Z]\},\end{aligned}$$

- *$F(p)$ 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.*

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

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 3.1. The model is the history of the snapshots shown in Figures 2.1 through 2.3. To save space in the figure just the `specimen` value is shown. In the figure the timestamps are shown below

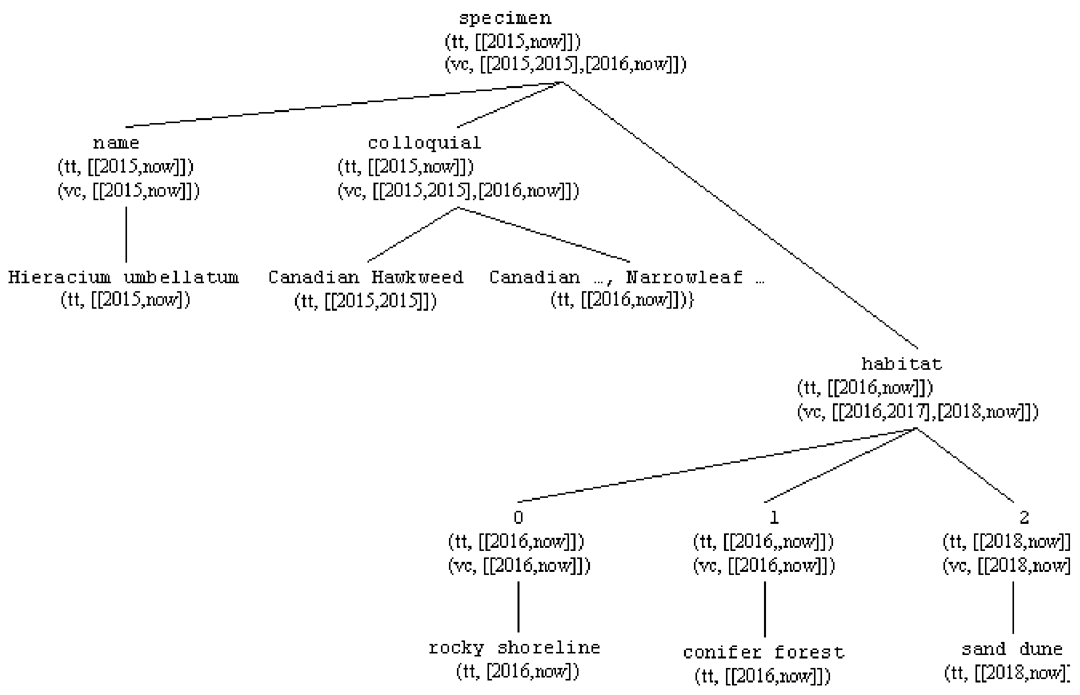


Fig. 3.1: The history of the `specimen`

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. Note that literals have only one version, hence we show only the transaction time since it 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 periods (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

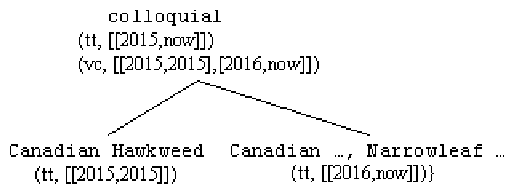


Fig. 3.2: Result of path lookup for `specimen.colloquial`

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

Definition 3.2.3 (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 3.1 then $\mathcal{P}^T(D^T, \texttt{specimen.colloquial})$ yields the model shown in Figure 3.2.

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

Definition 3.2.4 (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 3.1 then $\mathcal{T}^T(D^T, 2015)$ yields the model shown in Figure 3.3.

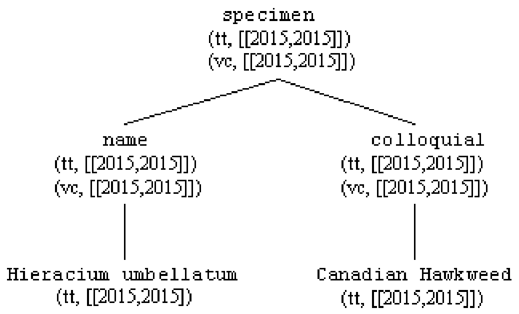


Fig. 3.3: Result of a time slice of 2015 on the model in Figure 3.1

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

Definition 3.2.5 (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 3.1 then $\mathcal{V}^T(D^T, 1)$ yields the model shown in Figure 3.4 (note that the version numbers start at 0).

The snapshot operators produce a snapshot JSON document.

Definition 3.2.6 (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 3.1 then $\mathcal{T}^S(D^T, 2015)$ yields the model shown in Figure 3.5, which corresponds to the JSON of Figure 2.1.

Version snapshot is similar to time snapshot.

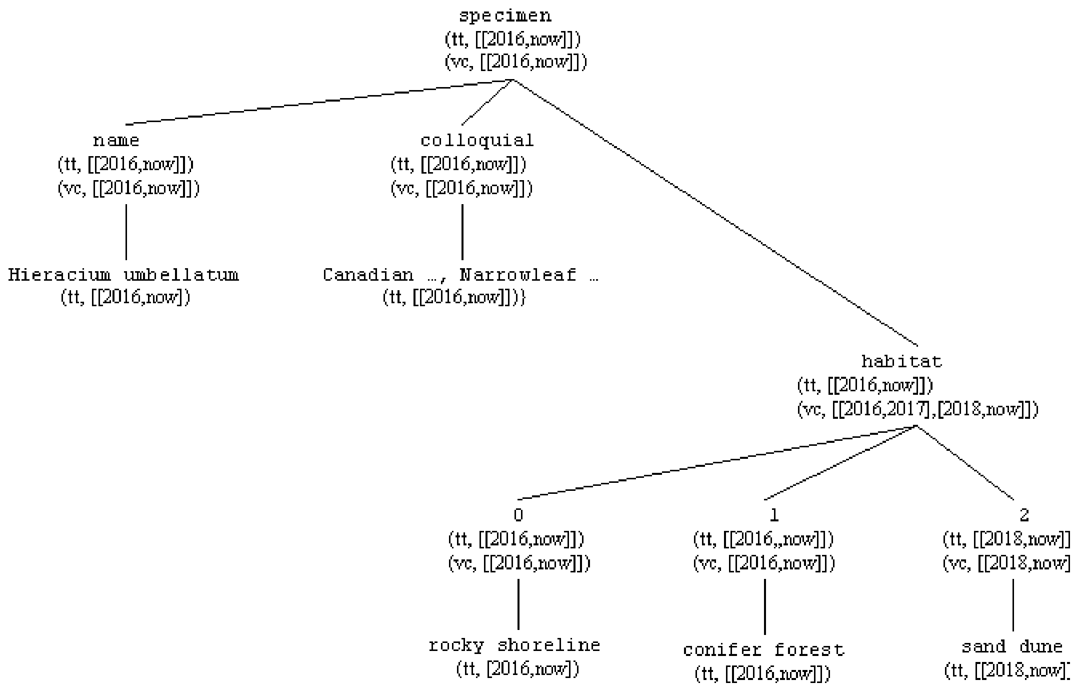


Fig. 3.4: The second version of `specimen`

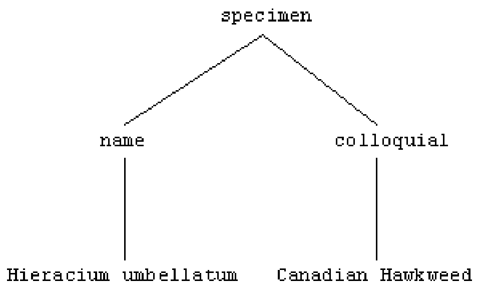


Fig. 3.5: Result of a time snapshot of 2015 on the model in Figure 3.5, which corresponds to the JSON of Figure 2.1

Definition 3.2.7 (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 3.5.

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

$[\![\mathtt{x}.\alpha]\!]^T[D^T] \equiv [\![\alpha]\!]^T[\mathcal{P}^T(D^T, \mathtt{x})]$

Time slice

$[\![\mathtt{timeSlice}(t).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!]^T[\mathcal{T}^T(D^T, t)]$

Version slice

$[\![\mathtt{versionSlice}(n).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!]^T[\mathcal{V}^T(D^T, n)]$

Time snapshot

$[\![\mathtt{timeSnapshot}(t).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!][\mathcal{T}^S(D^T, t)]$

The current snapshot, that is, the snapshot as of now, has special syntax

$[\![\mathtt{current}().\alpha]\!]^T[D^T] \equiv [\![\alpha]\!][\mathcal{T}^S(D^T, \mathtt{now})]$

Version snapshot

$[\![\mathtt{versionSnapshot}(n).\alpha]\!]^T[D^T] \equiv [\![\alpha]\!][\mathcal{V}^S(D^T, t)]$

As an example the expression `specimen.colloquial.versionSlice[1]` would be translated as follows.

$[\![\mathtt{specimen.colloquial.versionSlice(1)}]\!]^T[D^T] \equiv$

$\quad [\![\mathtt{colloquial.versionSlice(1)}]\!]^T[\mathcal{P}^T(D^T, \mathtt{specimen})]$

$[\![\texttt{versionSlice(1)}]\!]^T[\mathcal{P}^T(\mathcal{P}^T(D^T, \texttt{specimen}), \texttt{colloquial})]$

$V^T(\mathcal{P}^T(\mathcal{P}^T(D^T, \texttt{specimen}), \texttt{colloquial}), 1)$

3.3 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 3.6. 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 captures the version as shown in Figure 3.7. 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`.

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

Fig. 3.6: Format of a key/value set as a set of items

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

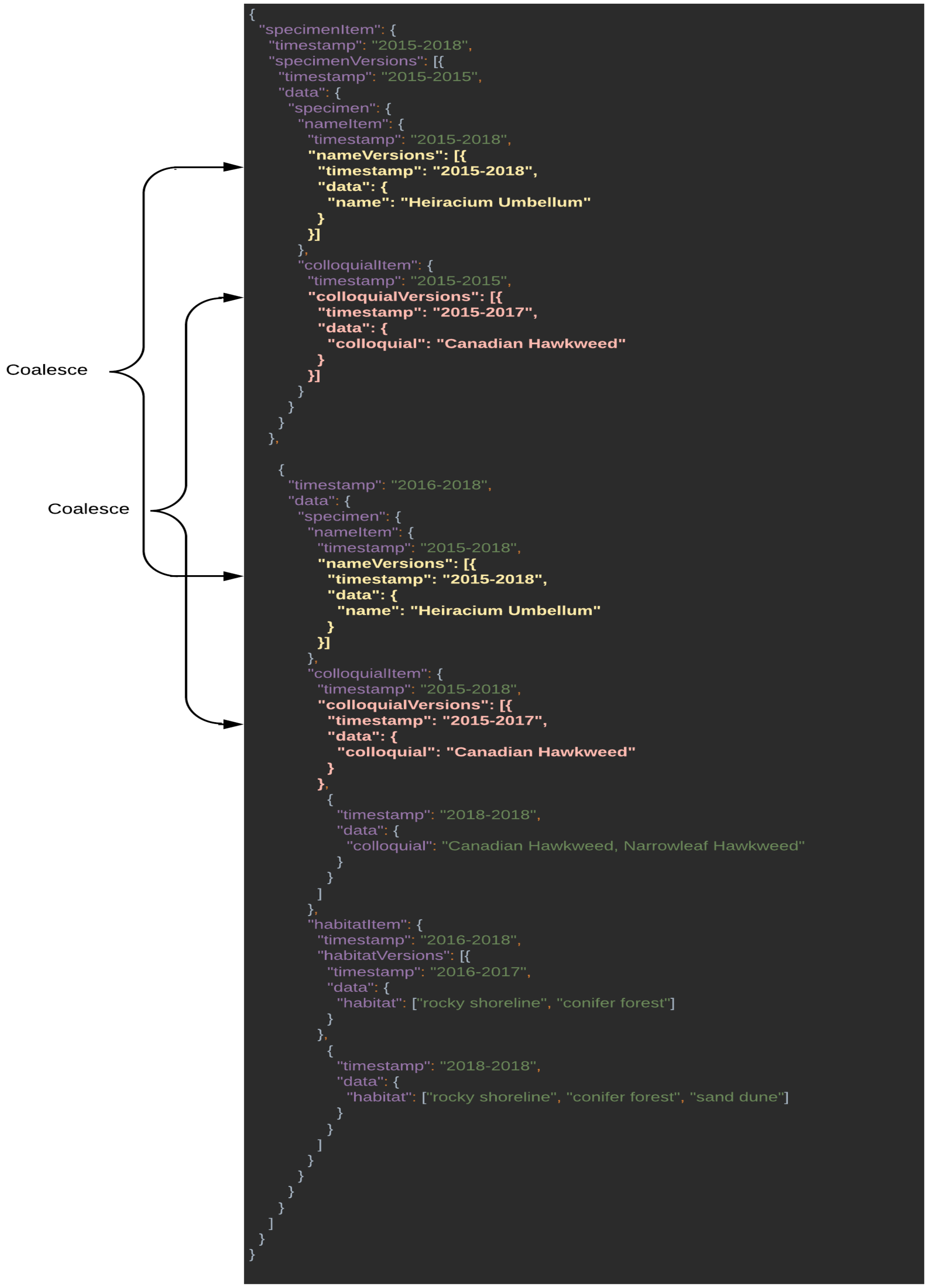
Fig. 3.7: Format of a key/value set as a set of items with Version representation

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. Figure 3.8 shows `name` and `colloquial` items across two versions of their parent(`specimen`). Because a new key-value of `habitat` was added, a new version of `specimen` was created. But `name` did not change so same version is copied in the second version of `specimenItem`. Similarly, there is a version of `colloquial` which is present in both versions of `specimenItem`. These version of children need to be coalesced to represent a single version as their values did not change.

The snapshot function shows a non-temporal view of the document at a particular time or a particular version as requested by the user. Examples of path expressions using the snapshot function are given below.



```
{
  "specimenItem": {
    "timestamp": "2015-2018",
    "specimenVersions": [{
      "timestamp": "2015-2015",
      "data": {
        "specimen": {
          "nameItem": {
            "timestamp": "2015-2018",
            "nameVersions": [{
              "timestamp": "2015-2018",
              "data": {
                "name": "Heiracium Umbellum"
              }
            }]
          },
          "colloquialItem": {
            "timestamp": "2015-2015",
            "colloquialVersions": [{
              "timestamp": "2015-2017",
              "data": {
                "colloquial": "Canadian Hawkweed"
              }
            }]
          }
        }
      }
    },

      {
        "timestamp": "2016-2018",
        "data": {
          "specimen": {
            "nameItem": {
              "timestamp": "2015-2018",
              "nameVersions": [{
                "timestamp": "2015-2018",
                "data": {
                  "name": "Heiracium Umbellum"
                }
              }]
            },
            "colloquialItem": {
              "timestamp": "2015-2018",
              "colloquialVersions": [{
                "timestamp": "2015-2017",
                "data": {
                  "colloquial": "Canadian Hawkweed"
                }
              },
                {
                  "timestamp": "2018-2018",
                  "data": {
                    "colloquial": "Canadian Hawkweed, Narrowleaf Hawkweed"
                  }
                }
              ]
            },
            "habitatItem": {
              "timestamp": "2016-2018",
              "habitatVersions": [{
                "timestamp": "2016-2017",
                "data": {
                  "habitat": ["rocky shoreline", "conifer forest"]
                }
              },
                {
                  "timestamp": "2018-2018",
                  "data": {
                    "habitat": ["rocky shoreline", "conifer forest", "sand dune"]
                  }
                }
              ]
            }
          }
        }
      }
    ]
  }
}
```

Fig. 3.8: Coalescing

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": "Canadian Hawkweed"
    }
  }
}]
```

Fig. 3.9: Time snapshot as of 2015

- `specimen.TimeSnapshot(2015)` Evaluating this expression should return the snapshot shown in Figure 3.9. 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.
- `specimen.TimeSnapshot(2016-2018)` would output Figure 3.10.
- `habitat.TimeSnapshot(2016-2018)`, time snapshots of habitat item would look like Figure 3.11.

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`.

```
[{
    "time": "2016-2017",

    "data": {
      "specimen": {
        "name": "Hieracium umbellatum",
        "colloquial": "Canadian Hawkweed",
        "habitat": ["rocky shoreline", "conifer forest"]
      }
    }
  },
  {
    "time": "2018-2018",

    "data": {
      "specimen": {
        "name": "Hieracium umbellatum",
        "colloquial": "Canadian Hawkweed, Narrowleaf Hawkweed",
        "habitat": ["rocky shoreline", "conifer forest", "sand dune"]
      }
    }
  }
]
```

Fig. 3.10: Time Snapshots of `Specimen` as of 2016-2018

```
[{
    "time": "2016-2017",

    "data": {
      "habitat": ["rocky shoreline", "conifer forest"]
    }
  },
  {
    "time": "2018-2018",

    "data": {
      "habitat": ["rocky shoreline", "conifer forest", "sand dune"]
    }
  }
]
```

Fig. 3.11: Time Snapshots of `habitat` 2016-2018

```
[{
  "time": "2015-2015",

  "data": {
    "specimen": {
      "name": "Hieracium umbellatum",
      "colloquial": "Canadian Hawkweed"
    }
  }
}]
```

Fig. 3.12: Version one snapshot of `specimen`

- `specimen.versionSnapshot(0)` should output all the snapshots in the first version of `specimen`.
- Similarly `specimen.versionSnapshot(1)` should output all the snapshots in second version of specimen.

Slicing is an operation performed on temporal JSON which gives a temporal view of the document at a particular slice of 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 3.14.

```
[{
    "timestamp": "2016-2017",

    "data": {
      "specimen": {
        "name": "Hieracium umbellatum",
        "colloquial": "Canadian Hawkweed",
        "habitat": ["rocky shoreline", "conifer forest"]
      }
    }
  },
  {
    "timestamp": "2018-2018",

    "data": {
      "specimen": {
        "name": "Hieracium umbellatum",
        "colloquial": "Canadian Hawkweed, Narrowleaf Hawkweed",
        "habitat": ["rocky shoreline", "conifer forest", "sand dune"]
      }
    }
  }
]
```

Fig. 3.13: Snapshots of `specimen` Version 2

```
{
  "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": "Canadian Hawkweed"
              }
            }]
          }
        }
      }
    }]
  }

}
```

Fig. 3.14: Version(0) Slice of `Specimen`

```
{
  "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": "Canadian Hawkweed"
              }
            }]
          }
        }
      }
    }]
  }

}
```

Fig. 3.15: Time Slice of `Specimen` as of 2015

CHAPTER 4
Implementation

This chapter discusses the implementations of operations we defined. Section 4.1 explains the flow of recursive function, Section 4.2 provides the technical information about reducing overlapping timestamps, Section 4.3 outlines the architecture and algorithm of `TimeSnapshot`, Section 4.4 describes the algorithm for `VersionSnapshot`, which follows the same architecture as `TimeSnapshot`. Section 4.5 and 4.6 describes the algorithm implementation for `TimeSlice` and `VersionSlice` operations respectively.

4.1 Recursive Function

To create a snapshot or a slice from the *JSON* data, first we need to find the requested `item` for which snapshot/slice needs to be created from the path expression. To address this, we created a recursive function which creates a list of `items` from the input path expression and deep dives into the *JSON* until that `item` is found. Once the `item` is found it will act as new *JSON* data for later functions. This flow is common to all the operations, except a small difference. For time related operations, overlapping of input timestamp is checked and for version operations, coalescing is performed.
Sample Path Expressions using Dot notations:

1. `specimen.timeSnapshot[2015-2015]`
2. `specimen.name.subname.versionSnapshot[3]`
3. `specimen.name.subname.versionSlice[1]`
4. `specimen.colloquial.timeSlice[2016-2018]`

Expression. 2 as a list of `items`:

$$\texttt{items} = [\texttt{'specimen','name','subname'}] \tag{4.1}$$

Function Call - `get_recursive_items(items, JSON)`

```
{
    "specimenItem": {
        "nameItem":{
            "subnameItem": {...}
        }
    }
}
```

Fig. 4.1: *Initial JSON*

```
{
    "nameItem": {
        "subnameItem": {...}
    }
}
```

Fig. 4.2: *JSON* after 2nd recursive call

This function will recursively deep dive into `specimen` item until it finds `subname` item. After each recursive call JSON is updated to the inner item which itself is a dictionary.

1st recursive call -

$$\texttt{items} = [\textbf{'specimen'}, \texttt{'name'}, \texttt{'subname'}] \tag{4.2}$$

2nd recursive call -

$$\texttt{items} = [\texttt{'specimen'}, \textbf{'name'}, \texttt{'subname'}] \tag{4.3}$$

3rd recursive call -

$$\texttt{items} = [\texttt{'specimen'}, \texttt{'name'}, \textbf{'subname'}] \tag{4.4}$$

```
{
    "subnameItem": {...}
}
```

Fig. 4.3: *JSON* after last recursive call

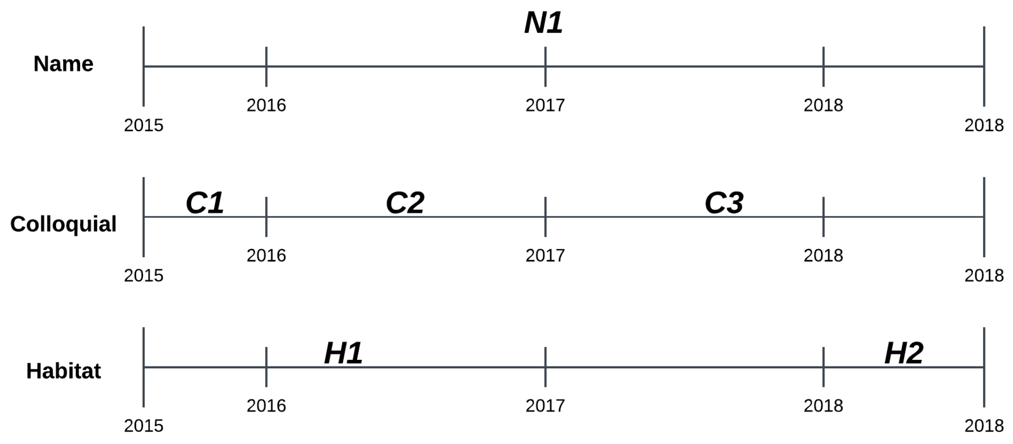


Fig. 4.4: Timeline Example

4.2 Timestamp Reduction

Reducing the timestamps is a very crucial part of creating `TimeSnapshot` or `Version Snapshot`. There can be multiple overlapping timestamps which need to be reduced to a set of timestamps which can incorporate the proper timeline of snapshots. Figure 4.4 shows three attributes with their timeline, *i.e.,* life of the attribute's value between the respective years. The value of `name` is $N1$ from 2015-2018, `colloquial` and `habitat` have different values between 2015-2018. Algorithm 2 defines the implementation.

4.3 Time Snapshot

A `TimeSnapshot` architecture has to transform the temporal JSON into an array of snapshots on the basis of input time. These snapshots comprises of data represented in key/value pairs for a particular point in time. To implement this operation, we broke it down into a couple of functions where output from one function is consumed by another function. The algorithm begins with a recursive function which deep dives into *JSON* and retrieves the requested `item`, after which this new *JSON* is fed to a `getTimestamps()` function which extracts all the item's timestamps and outputs an array. Many of these timestamps overlap, which are then reduced by a `preprocess()` function and a snapshot

Algorithm 2: Timestamp Reduction Algorithm

```
Input: List $T1$ of Timestamps
Output: List $T2$ of Reduced timestamps
Procedure reduceTimestamps(Timestamps $T1$)
$startTimes$ := [ ]
$endTimes$ := [ ]
$start$ := ' '
$end$ := ' '
for $t$ in $T1$ do
    // Split the timestamp
    $start$, $end$ = $t$.split("-")
    // Append to respective lists
    $startTimes$.append($start$)
    $endTimes$.append($end$)
end
$smallestStart$ = min($startTimes$)
$maxEnd$ = max($endTimes$)
while $smallestStart <= maxEnd$ do
    $smallestend$ = min($endTimes$)
    Pop $smallestend$ from $endTimes$
    Push " $smallestStart$ + '-' + $smallestend$ " in $T2$
    $smallestStart = smallestend + 1$
end
return $T2$
```

skeleton is created with reduced timestamps. The last function, `populateData()` iterates through the new *JSON* and the skeleton and populates the skeleton only if `checkOverlap()` return *True*. This `checkOverlap()` is used to check if timestamps of items in the new *JSON* and skeleton overlap. The architecture is represented by Figure 4.5.

4.4 Version Snapshot

A `VersionSnapshot` architecture has to transform the temporal JSON into an array of snapshots on the basis of input version *N*. It follows the same architecture as `TimeSnapshot`. The key difference in this operation is, while populating the skeleton we don't use the `checkOverlap` function to check for the overlapping timestamps between the skeleton and new *JSON* produced by recursive function, rather we populate the skeleton till *Nth* version of requested `item` by coalescing which makes sure there are no duplicate versions and return the requested version from the array. The architecture is represented in Figure 4.5.

4.5 Time Slice

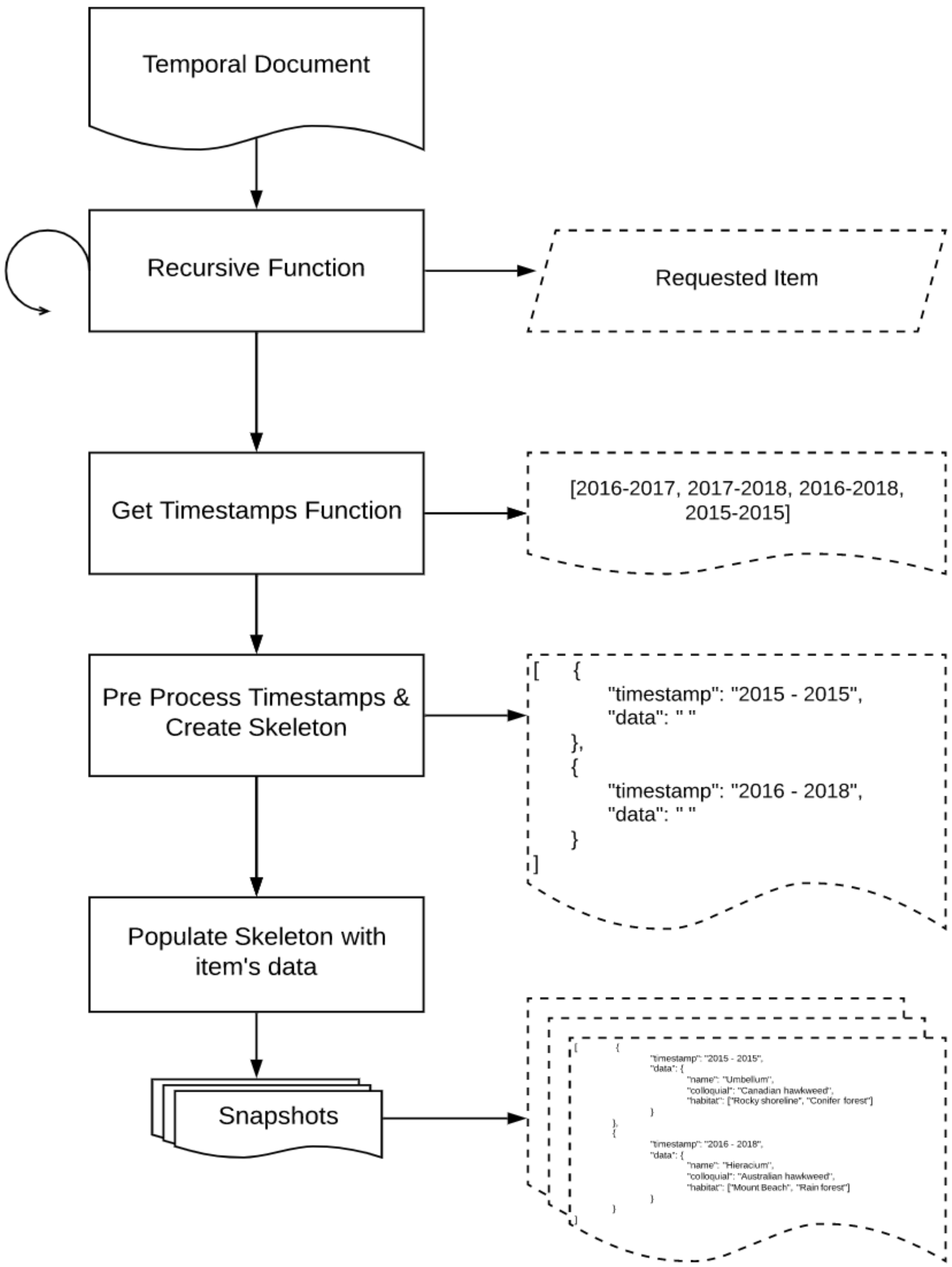
`Time slice` is a simple operation which takes a path notation of items and time as an input. It is a recursive function which deep dives into *JSON*, until `item` is found. It checks for the overlapping timestamps and return the `item` with its temporal attributes. Algorithm 5 describes the implementation of this operation.

4.6 Version Slice

`Version slice` is similar to `Time Slice` in the way both extracts the slice of `item` from *JSON* with its temporal attributes intact, the difference is, `Version Slice` does not check for overlapping timestamps. Here we use coalescing to make sure duplicate versions across items are counted as a single version. Coalescing is implemented by using a `Flag` variable which marks a version once it has been extracted. Algorithm 6 describes the implementation of this operation.

```
Algorithm 3: Time Snapshot Algorithm
  Input: List $A$ of Items, Timestamp $T$
  $A$ consists of dot notation path to item for which Time Snapshot is created.
  Last item is the final item.
  $T$ is used to populate data with overlapping timestamps.
  Output: List $O$
  $O$ contains JSON objects of Time snapshots
  Data: Parent JSON
  Procedure TimeSnapshot(Array $A$, Timestamp $T$, Parent JSON)
  $newdata$:= Null
  $data$ := Parent JSON
  $timeArray$ := [ ]
  $processedArray$ := [ ]
  $snapshotArray$ := [ ]
  // Getting the last item of input array from data
  for $i \leftarrow 0$ to $len(data)$ do
      if $i+1 == len(A)$ then break
      else
      // Call function recursively
      $newdata \leftarrow$ getItems($data, T, A$)
  end
  // Extract timestamps from $newdata$ which overlap with input timestamp $T$
  for $i \leftarrow 0$ to $len(newdata)$ do
      for $k,v$ in $newdata[i].iteritems()$ do
          if $k$ == 'timestamp' and checkOverlap($v,T$) is True then

          timeArray.append(v)
      end
  end
  // Preprocess timestamps and generate a snapshot skeleton
  $preprocessedArray \leftarrow$ preProcess($timeArray$)
  // Populate skeletons with relevant data
  for $i \leftarrow 0$ to $len(preprocessedArray)$ do
      for $j \leftarrow 0$ to $len(newdata)$ do
          if checkOverlap($i$['timestamp'], $j$['timestamp']) is True then
          $snapshotArray \leftarrow$ populateData(i['data'],j['data'])
      end
  end
  return $snapshotArray$
```



```
[    {
          "timestamp": "2015 - 2015",
          "data": " "
     },
     {
          "timestamp": "2016 - 2018",
          "data": " "
     }
]
```

```
[         {
               "timestamp": "2015 - 2015",
               "data": {
                         "name": "Umbellum",
                         "colloquial": "Canadian hawkweed",
                         "habitat": ["Rocky shoreline", "Conifer forest"]
               }
          },
          {
               "timestamp": "2016 - 2018",
               "data": {
                         "name": "Hieracium",
                         "colloquial": "Australian hawkweed",
                         "habitat": ["Mount Beach", "Rain forest"]
               }
          }
]
```

Fig. 4.5: Method Architecture

Algorithm 4: Version Snapshot Algorithm

```
Input: List $I$ of Items, Version Number $N$
$I$ consists of dot notation path to item for which Version Snapshot is created.
Last item is the final item.
$N$ is used to retrieve the Nth version of item ($I[-1]$ from $I$).
Output: List $O$
$O$ contains JSON objects of Time snapshots
Data: Parent JSON
Procedure VersionSnapshot(List $I$, Version $N$, Parent JSON $P$)
$newdata$:= Null
$data$ := P
$timeArray$ := [ ]
$processedArray$ := [ ]
$versionsArray$ := [ ]
// Getting the last item of input array from data
for $i \leftarrow 0$ to len(data) do
    if $i$ == len(I) then break
    else
    // Call function recursively
    $newdata \leftarrow$ getItems($data, T, I$)
end
// Extract timestamps from $newdata$
for $i \leftarrow 0$ to $len(newdata)$ do
    for k,v in newdata[i].iteritems() do
        if k == 'timestamp' and v not in timeArray then timeArray.append(v)
    end
end
// Preprocess timestamps and generate a snapshot skeleton
$preprocessedArray \leftarrow$ preProcess($timeArray$)
// Populate skeletons with relevant values
for $i \leftarrow 0$ to $len(preprocessedArray)$ do
    for $j \leftarrow 0$ to $len(newdata)$ do
        if checkOverlap(i['timestamp'], j['timestamp']) is True then
          $versionsArray \leftarrow$ populateData(i['data'],j['data'])
    end
end
return $versionsArray$[N]
```

Algorithm 5: Time Slice Algorithm

```
Input: List I of Items, Timestamp T
T is used to populate data with overlapping timestamps.
Last item is the final item.
Output: JSON object O
O contains JSON object or Time Slice
Data: Parent JSON
Procedure TimeSlice(List I, Timestamp T, Parent JSON P, i)
data := P
sliceArray := [ ]
sliceDict := {}
// Getting the last item of input array from data
for i ←0 to len(data) do
    if i+1 == len(I) then
        if checkOverlap(data[i]['timestamp'], T) is True then
            sliceArray.append(data[i]['data'])
        end
    else
        // Call function recursively
        TimeSlice(I, T, newdata, i+1)
    end
end
// Create skeleton for timeslice and append slice
for i ←0 to len(sliceArray) do
    for j in sliceArray do
        if checkOverlap(j['timestamp'], T) is True then
            slice ← j
            t ← j['timestamp']
        end
    end
end
sliceDict.update(slice)
sliceDict.update(t)
return sliceDict
```

Algorithm 6: Version Slice Algorithm

```
Input: List $I$ of Items, Version $N$
$N$ is used to retrieve the Nth version of item ($I[-1]$ from $I$).
Last item is the final item.
Output: JSON object $O$
$O$ contains JSON object or Time Slice
Data: Parent JSON
Procedure VersionSlice(List $I$, Parent JSON $P$, $i$)
$data$ := P
//To keep track of already added versions ( Coalescing ) $Flag$ := {}
$versionsArray$ := [ ]
$sliceArray$ := [ ]
$sliceDict$ := {}
// Getting the last item of input array from data
for $i$ ←0 to len(data) do
    if $i + 1 == len(I)$ then
        //Coalescing
        if (data[i]['timestamp']) not in Flag then
            $sliceArray.append(data[i]['data'])$
            $true \leftarrow Flag[(data[i]['timestamp'])]$
        end
    else
        // Call function recursively
        VersionSlice(I, newdata[i], i+1)
    end
end
// Create skeleton for versionslice and append slice
for $i$ ←0 to len(sliceArray) do
    for j in sliceArray[i][versions] do
        if j['timestamp']) not in Flag or Flag[j]['timestamp']] is false then
            $versionsArray$.append(j)
            $true \leftarrow Flag[(j['timestamp'])]$
        end
    end
end
$t \leftarrow versionsArray$[N]['timestamp']
$slice \leftarrow versionsArray$[N]
$sliceDict$.update($slice$)
$sliceDict$.update($t$)
return $sliceDict$
```

CHAPTER 5
Evaluation

In this thesis we empirically evaluate 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.

5.1 Experimental Environment

We implemented our functions in Python 2.7. The implementation uses Python's inbuilt library for parsing JSON. To quantify the cost of primary functions like Time Snapshot, Version Snapshot, Time Slice & Version Slice we performed several experiments on varying sizes and types of temporal data. The experiments were run on a MacBook Pro, 2.7GHz quad-core 8th-generation Intel Core i7 processor, 16GB of RAM. Each experiment was performed in an isolated machine.

5.2 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, these time snapshots are then reversed by an auxiliary `reverse_function()` to form a temporal document which will be used to run our primary functions. 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 as shown in Figure 5.1, and another where we change values of inner elements (child) which creates different versions of those children nodes and only single version of parent as shown in Figure 5.2. We show the creation cost for both types of temporal documents.

```
{
  "specimenItem": {
    "timestamp": "2015-2017",
    "specimenVersions": [{
        "timestamp": "2015-2016",
        "data": {
          "specimen": {
            "colloquialItem": {},
            "nameItem": {}
          }
        }
      },
      {
        "timestamp": "2016-2017",
        "data": {
          "specimen": {
            "colloquialItem": {},
            "nameItem": {},
            "habitatItem": {}
          }
        }
      }
    ]
  }
}
```

Fig. 5.1: Temporal JSON with multiple versions of parent `specimen`

```
{
  "specimenItem": {
    "timestamp": "2015-2017",
    "specimenVersions": [{
      "timestamp": "2015-2017",
      "data": {
        "specimen": {
          "colloquialItem": {
            "timestamp": "2015-2016",
            "colloquialVersions": [{
              "timestamp": "1201-1201",
              "data": {
                "colloquial": "accruex"
              }
            }, {
              "timestamp": "2016-2017",
              "data": {
                "colloquial": "Hawkweed"
              }
            }]
          },
          "nameItem": {
            "timestamp": "2015-2017",
            "nameVersions": [{
              "timestamp": "2015-2016",
              "data": {
                "name": "Ramos"
              }
            }, {
              "timestamp": "2016-2017",
              "data": {
                "name": "Narrowleaf"
              }
            }]
          }
        }
      }
    }]
  }
}
```

Fig. 5.2: Temporal JSON with multiple versions of children (`colloquial` and `name`)

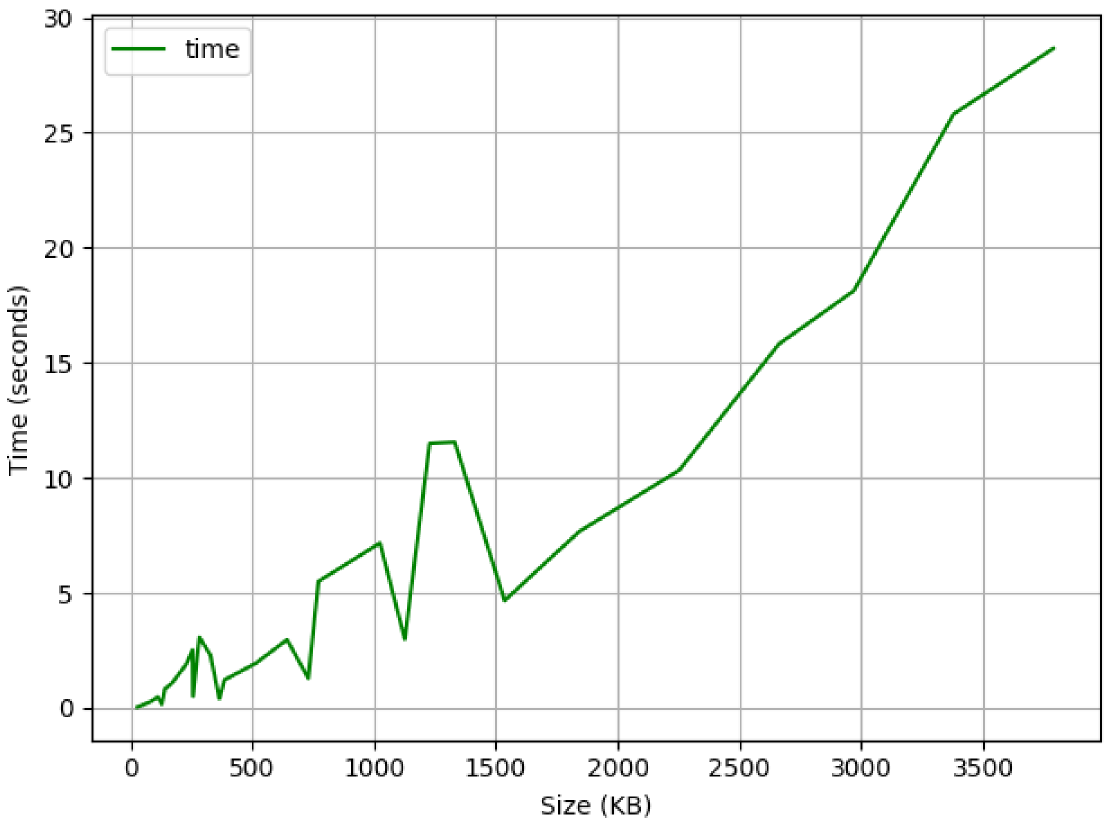


Fig. 5.3: Representational model creation time with parent version changes

5.2.1 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 higher size of document (up to 4Mb) depending on the number of key-value pairs. We change up to 30 key-values of parent, with each new version of parent, all of subsequent children values are copied into another version which leads to higher complexity while reversing the snapshots. The results are plotted in Figure 5.3, which represents the cost of creation.

5.2.2 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 5.4, which represents the cost of document creation.

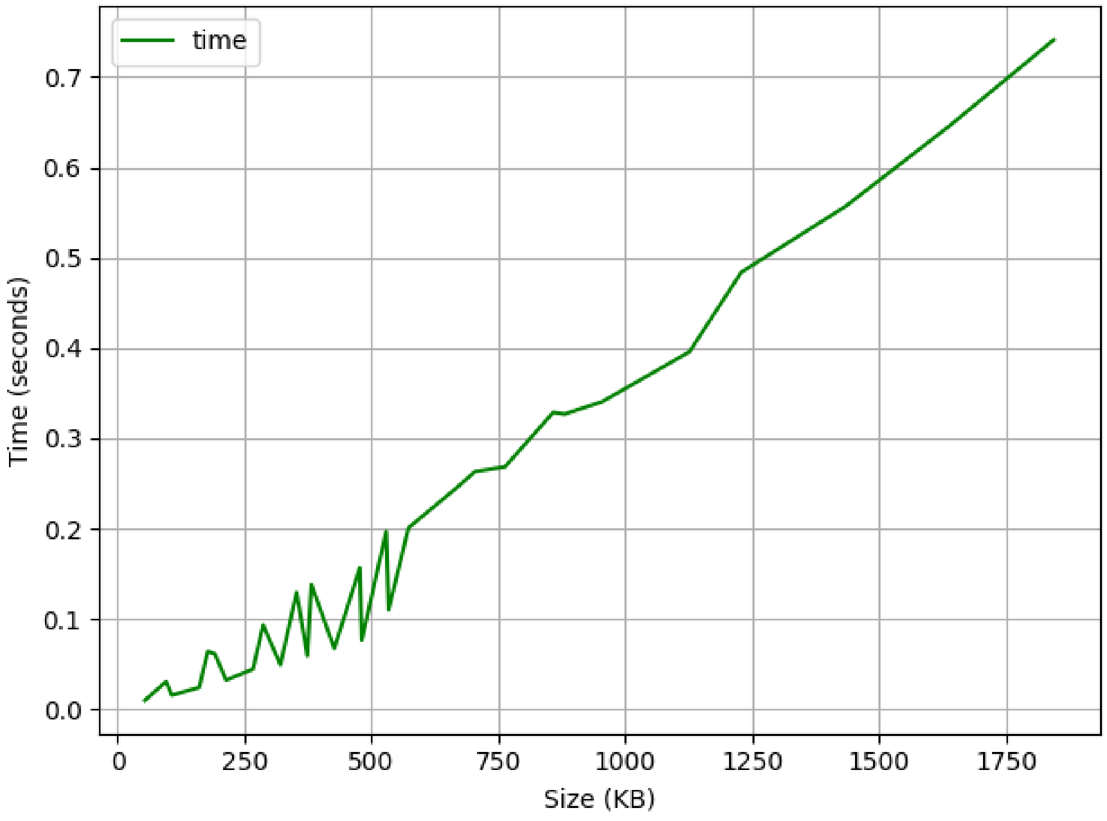


Fig. 5.4: Representational model creation time with child version changes

5.3 Experiment: Time Snapshot

In this section, we study the cost of creating time snapshots with two kinds of experiments: Cost with varying number of changes in key-value pairs (100 to 1000) and Cost with varying size of the temporal document. 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. There are two types of changes: change in key-values of parent and change in key-values of children.

We describe number of children key-value pairs in terms of three sizes: small, medium and large. Small documents have two key-value pairs, medium has ten key-value pairs and large has 30 different key-value pairs of children. We perform changes between 100 to 1000 in the increments of 100.

5.3.1 Time Snapshot with New Children Versions

Here 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 5.5 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.

5.3.2 Time 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. Figure 5.6 shows how cost varies with changes.

5.3.3 Time Snapshot Varying the Document Size

In this experiment we measure the time for time snapshot with respect to the document size. We perform this experiment for two different types of files: one with multiple versions

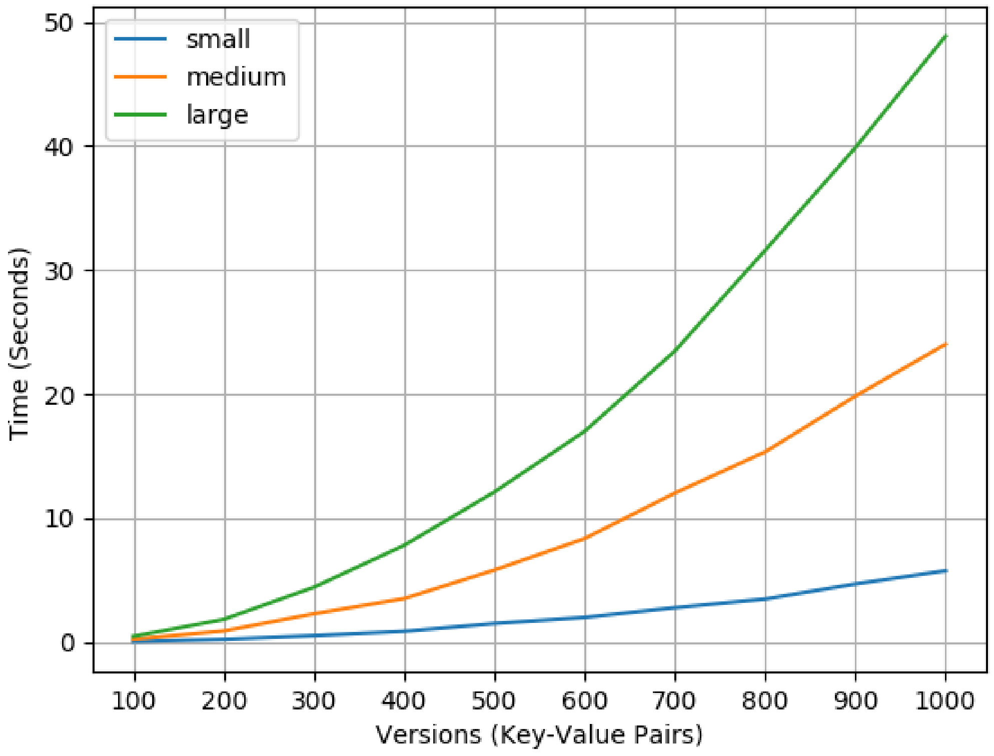


Fig. 5.5: Time snapshot with new version of children

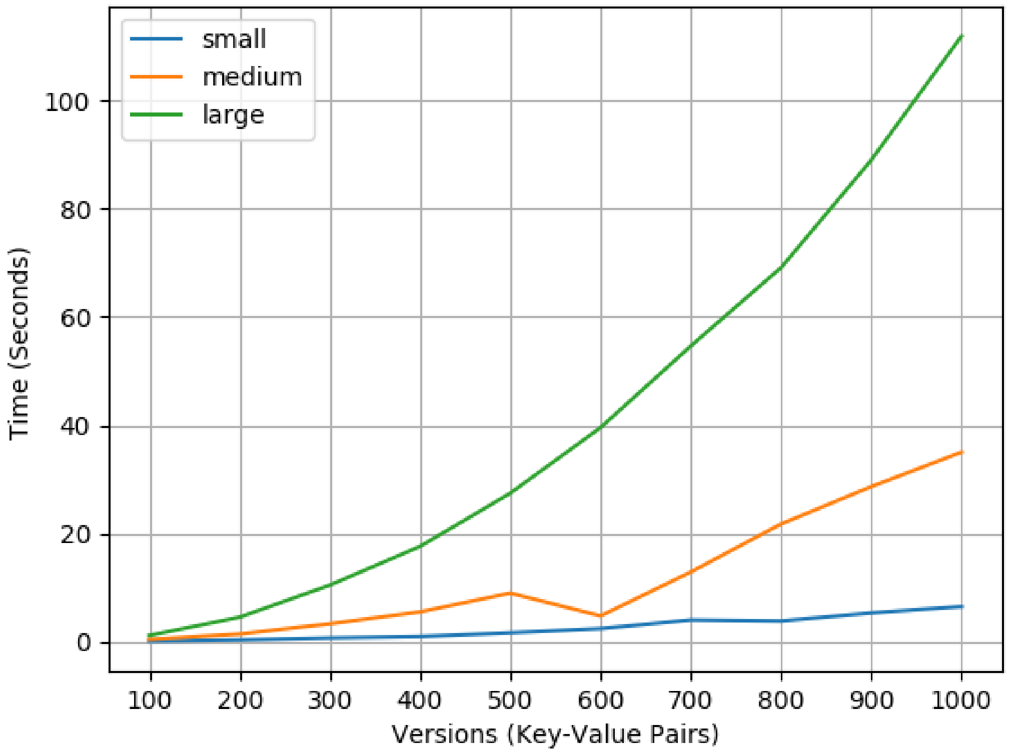


Fig. 5.6: Time snapshot with new version of parent

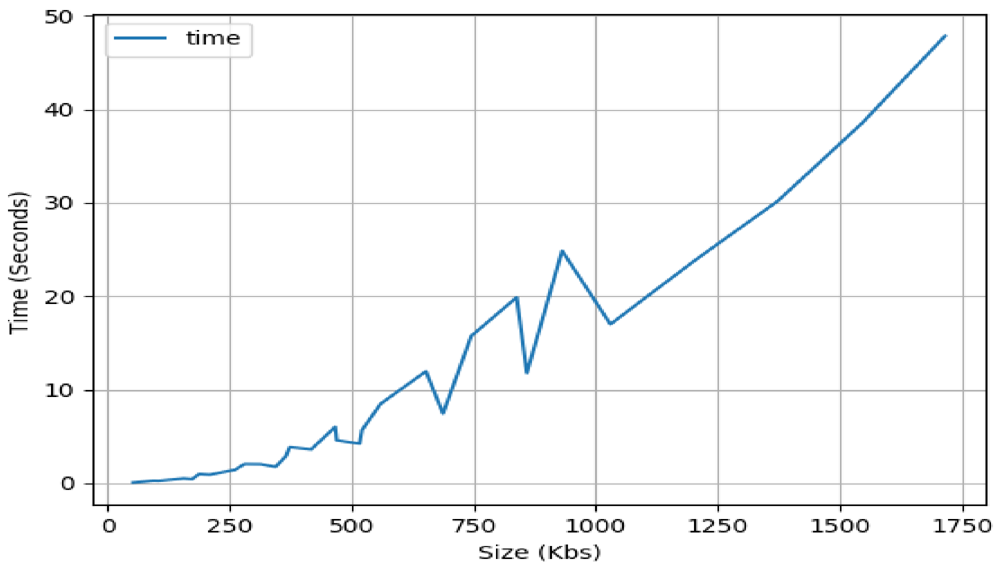


Fig. 5.7: Time snapshot varying the document size and new child versions

of parent and another with multiple versions of both. We observed that the latter takes more time as its size is almost double compared to former. The size for latter is up to 4Mb's (around 300,000 lines of JSON).

5.4 Experiment: Time Slice

In this section we study the cost of creating time slices (slice consists of temporal information) with two kinds of experiments: Cost with varying number of changes in key-value pairs(100 to 1000) and Cost with varying size of temporal document. There are two types of changes: change in key-values of Parent and change in values of Children.

5.4.1 Time Slice with New Children Versions and Parent Versions

`Time Slice` is slightly faster with new versions of the parent as compared with new versions of children. 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, if the parent timestamp overlaps then only it will dig into its children versions. However, with changes in children

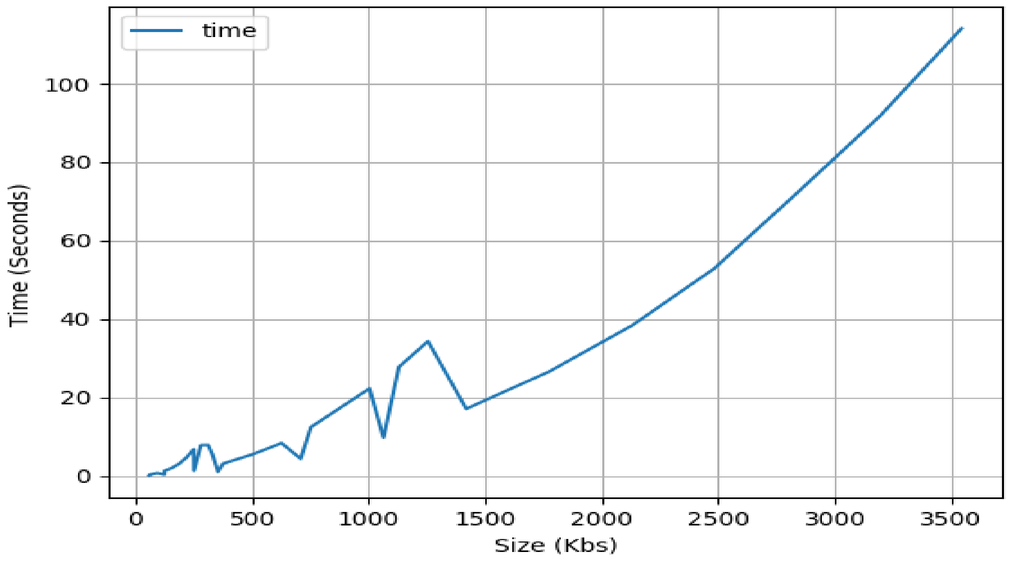


Fig. 5.8: Time snapshot varying the document size and new versions of child and parent

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 5.9 and 5.10.

5.5 Experiment: Version Snapshot

In this section we study the cost of creating Version-Snapshots with two kinds of experiments: Cost with varying number of changes in key-value pairs (100 to 1000) and Cost with varying size of temporal document. There are two types of changes: change in key-values of parent and change in key-values of children.

We describe number of children key-value pairs in terms of three sizes: small, medium and large. Small documents have two key-value pairs, medium has ten key-value pairs and large has 30 different key-value pairs of children. We perform changes between 100 to 1000 in the increments of 100.

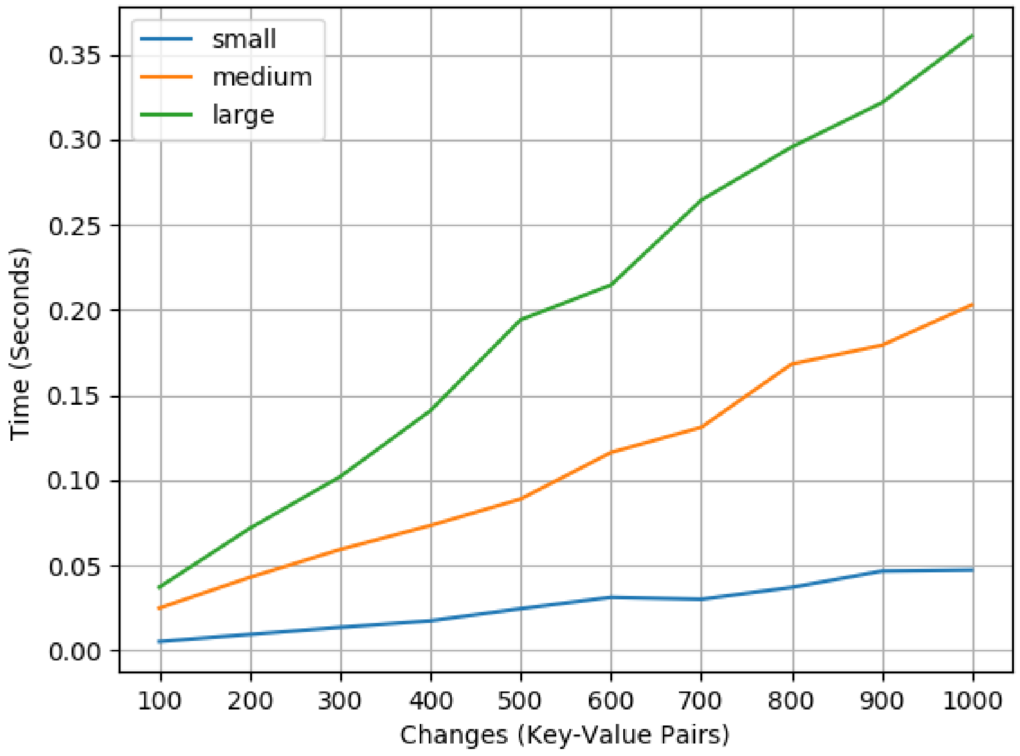


Fig. 5.9: Time slice with new versions of children

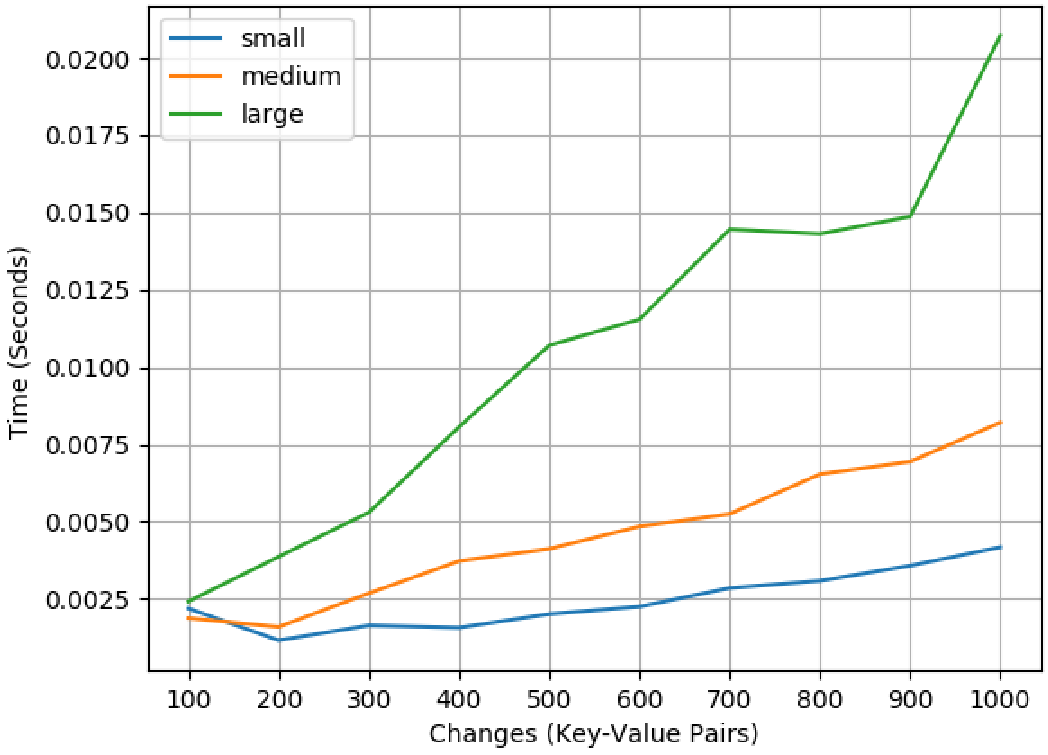


Fig. 5.10: Time slice with new versions of parent

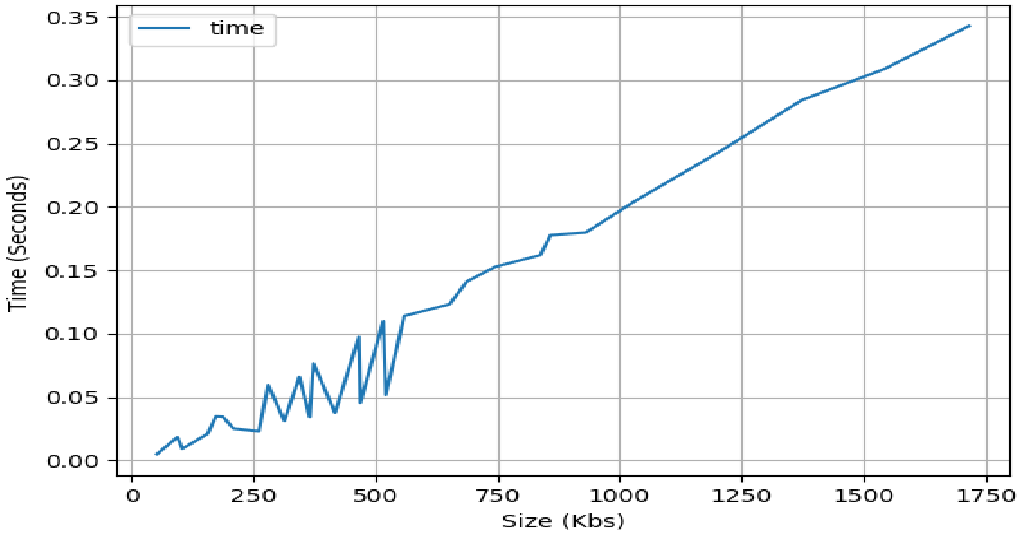


Fig. 5.11: Time slice with varying document size and new versions of child

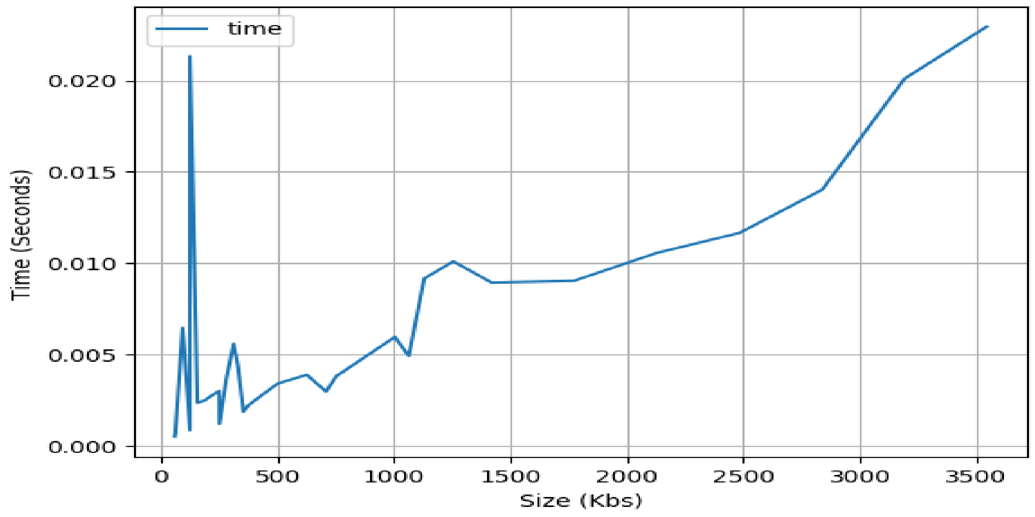


Fig. 5.12: Time slice with varying document size and new versions of parent

5.5.1 Version Snapshot with New Children Versions

Here 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 5.13 shows cost of creating version snapshots with respect to 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. Version Snapshot's run time depends on factors like levels of nesting and number of versions the requested item has. Grabbing *Nth* version will be straightforward and would require no coalescing.

5.5.2 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 5.14 shows how cost varies with changes.

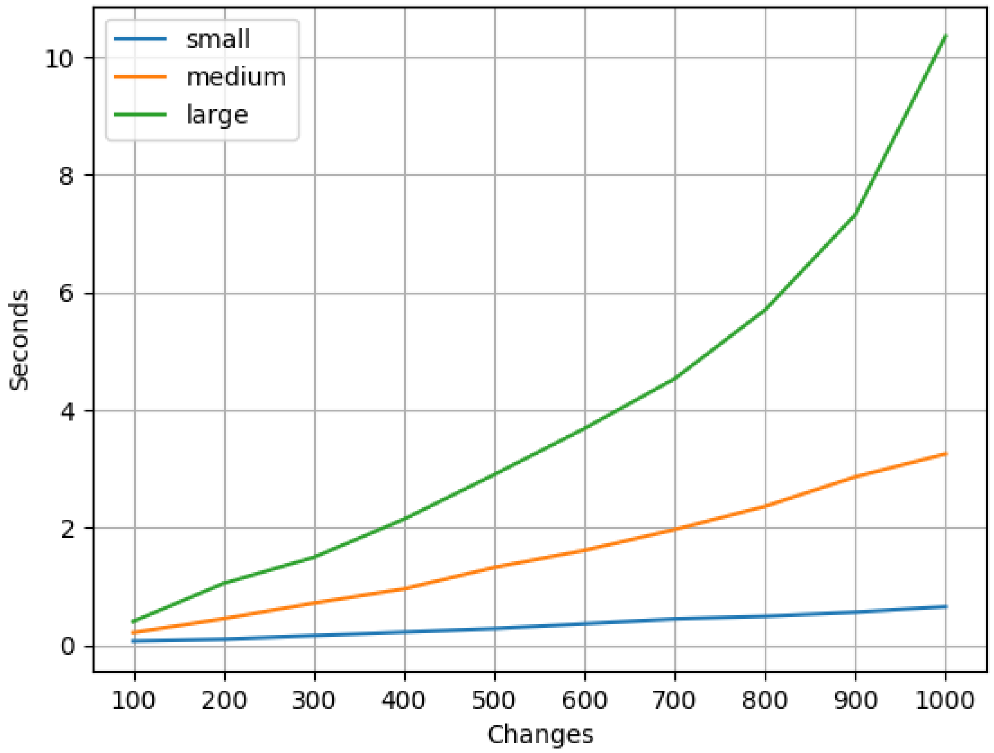


Fig. 5.13: Version snapshot with new versions of children

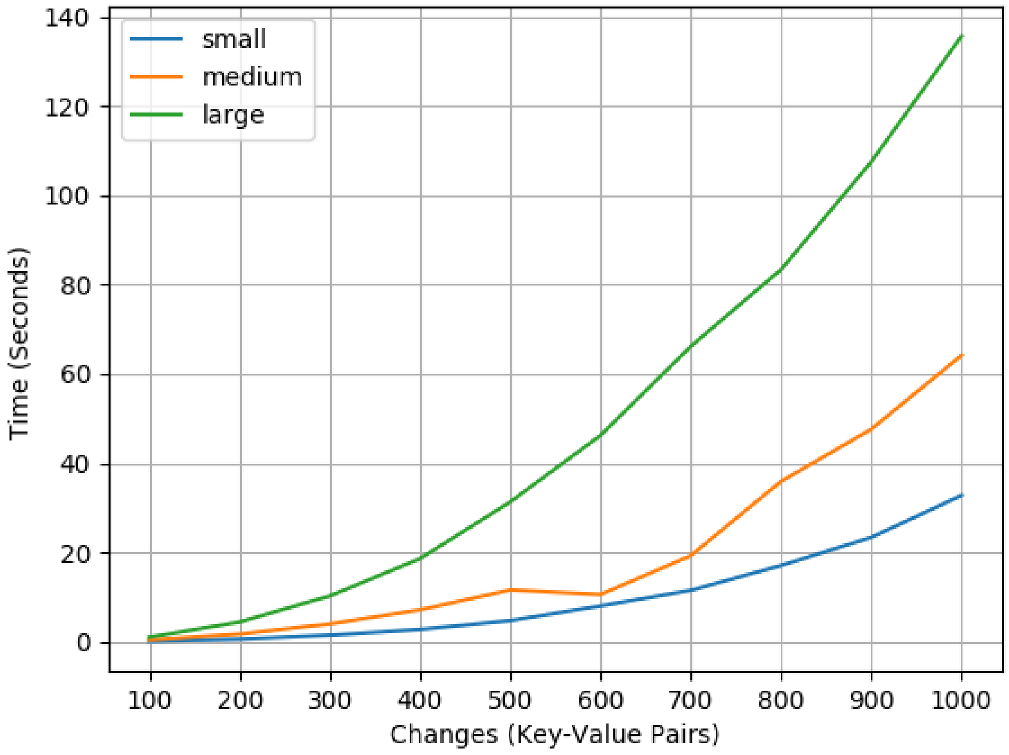


Fig. 5.14: Version snapshot with new versions of parent

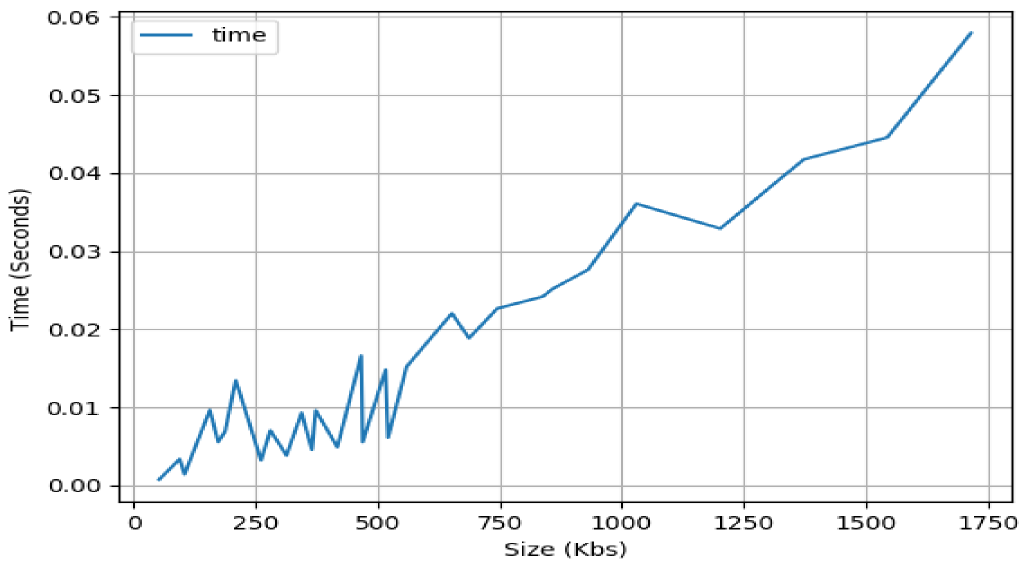


Fig. 5.15: Time with new versions of children

5.5.3 Version Snapshot Varying the Document Size

In this experiment we measure the time with respect to the document size. We perform this experiment for two different types of files: one with multiple versions of parent and another with multiple versions of both. We observed that the latter takes more time as its size is almost double compared to former. The size for latter is upto 4Mb's (around 300,000 lines of JSON.

5.6 Experiment: Version Slice

In this section we study the cost of creating version slices with two kinds of experiments: Cost with varying number of changes in key-value pairs (100 to 1000) and cost with varying size of temporal document. There are two types of changes: change in key-values of parent and change in values of children.

5.6.1 Version Slice with New Children Versions and Parent Versions

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

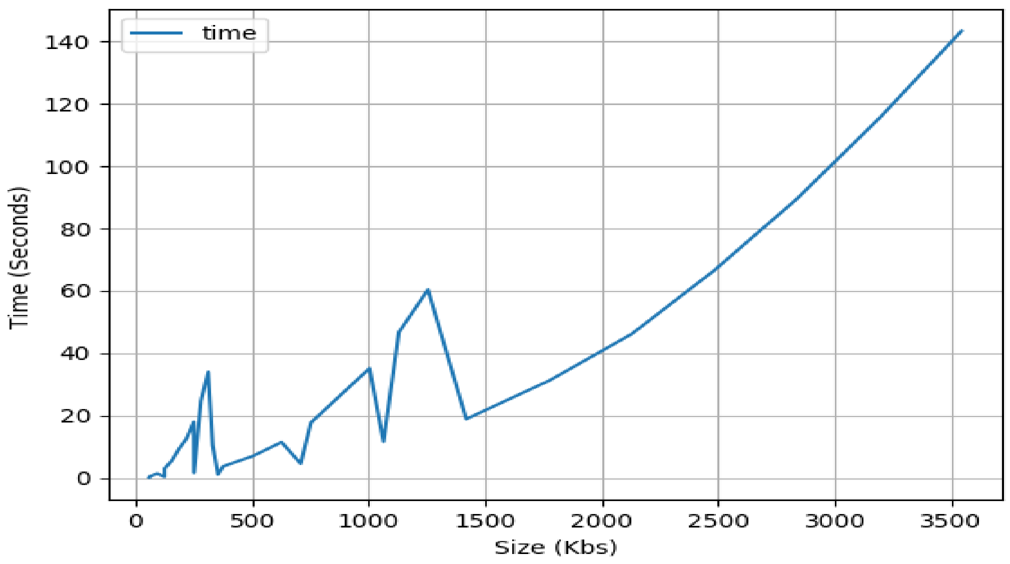


Fig. 5.16: Time with new versions of parent and children

children are copied across newer parent versions, which results in duplication, which adds up the coalescing time. The performance is shown by Figure 5.17 and Figure 5.18.

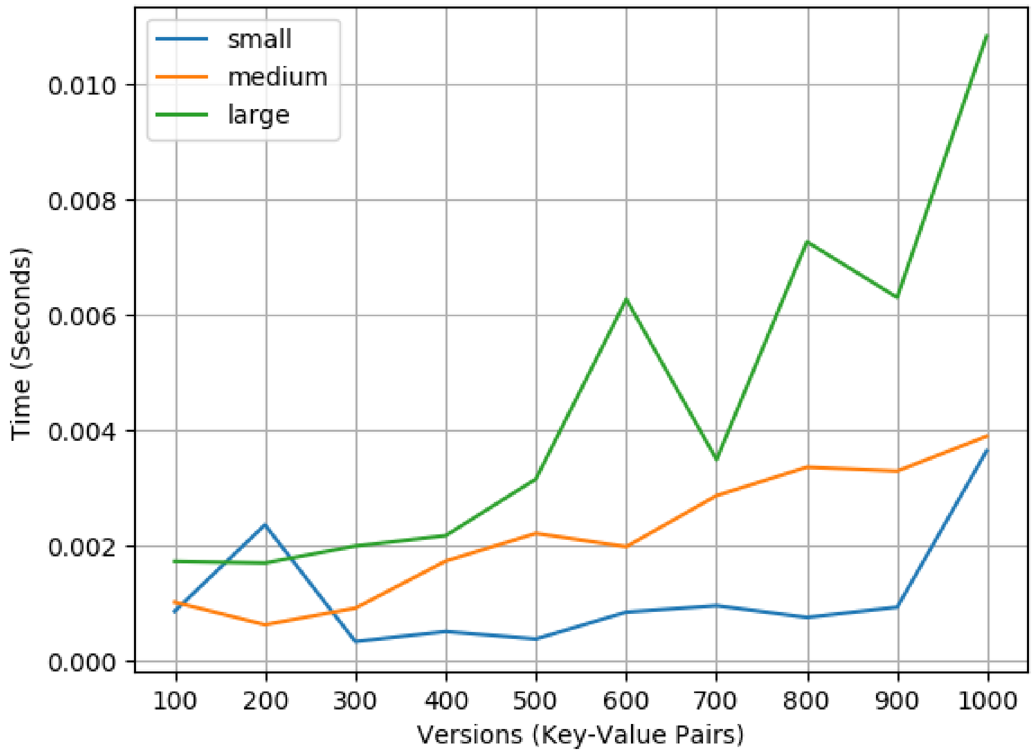


Fig. 5.17: Version slice with new versions of children

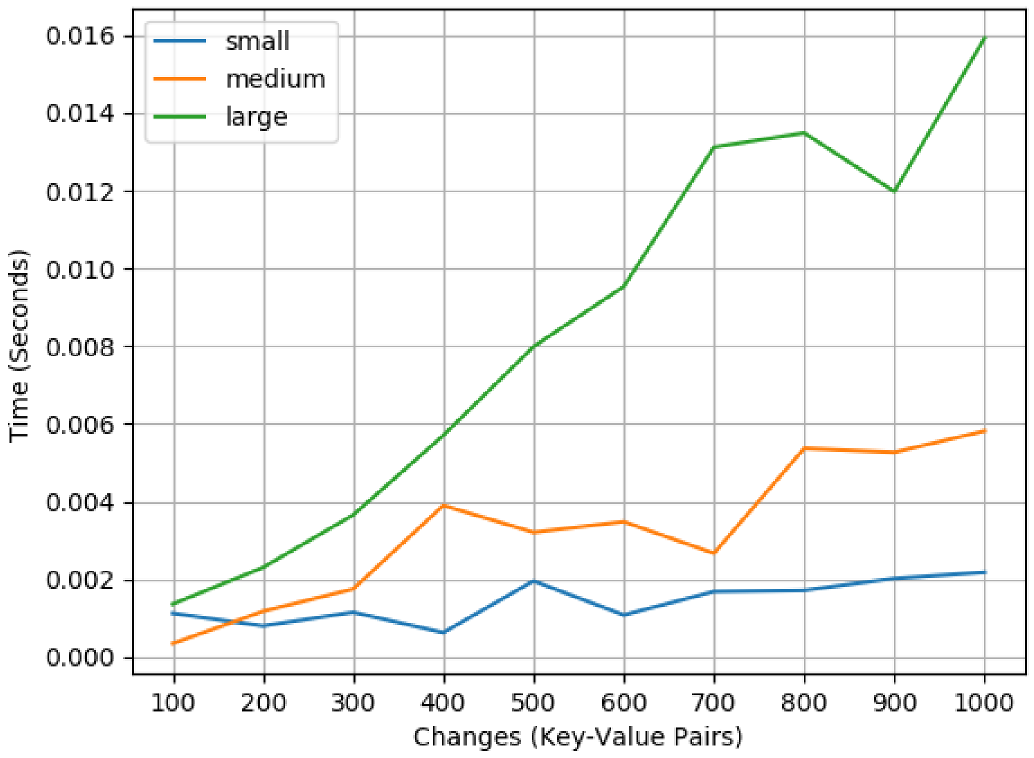


Fig. 5.18: Version slice with new versions of parent

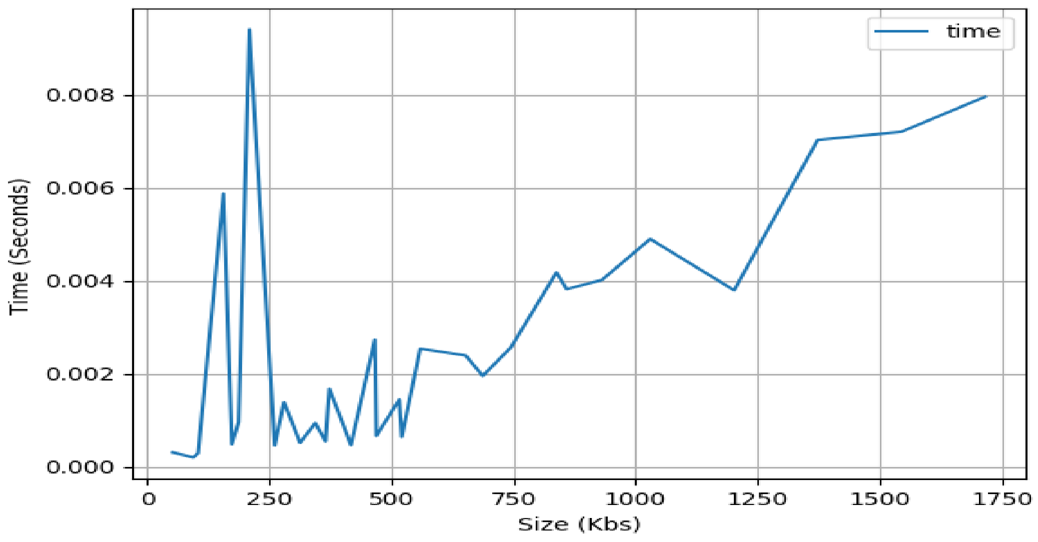


Fig. 5.19: Version slice with varying document size and new versions of child

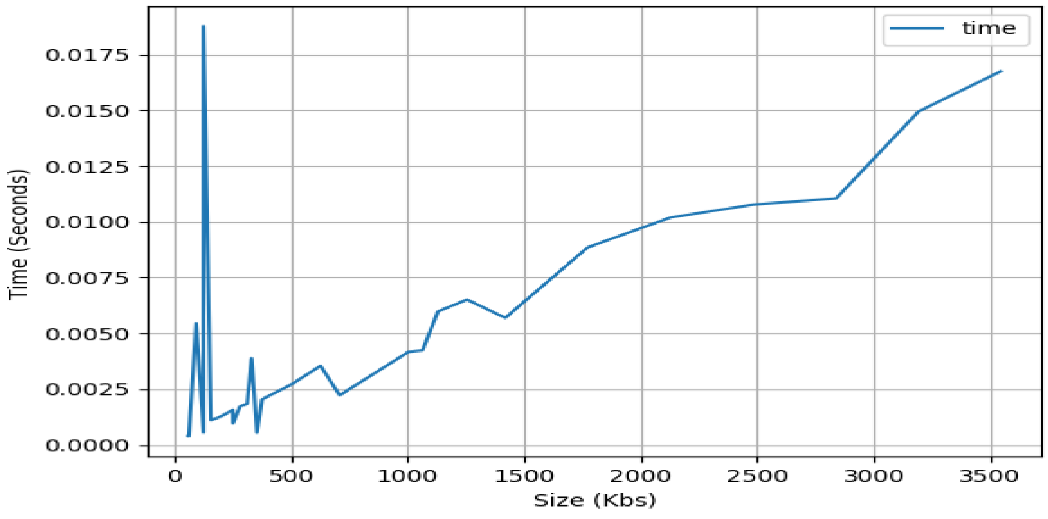


Fig. 5.20: Version slice with varying document size and new versions of parent

CHAPTER 6
Conclusions and Future Work

JavaScript Object Notation (JSON) is a format for representing data. In this thesis 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 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 thesis 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, and we give algorithms for implementing the operations.

- 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 more functions which could be implemented, like `delta()`, which gets the difference between two documents, whether it be two snapshots or temporal documents, others being `next()` & `previous()` functionality which can used to retrieve next and previous versions of current version.

Finally, future work considers an alternative, representational model to the item-based model presented previously. The primary drawback of the item-based model is that simple path expressions devolve to loops over versions of items. The alternative model utilizes the idea that versions of data should be pushed to values, *i.e.,* in a tree-based model of data, to the leaves of the tree, when possible.

An alternative representational model is shown in Figure 6.1. The model is similar to the temporal model, but explicitly represents versions of values in an array.

Considering the alternative representational model, let's consider the following operations.

TimeSlice -

```
specimen.name.timeslice[2015]
timeslice(specimen.data.name, 2015)

specimen.timeslice[2015]
timeslice(specimen, 2015)
If we convert specimen to timestamp and list,
interior nodes have just one object in list.
```

VersionSlice -

```
specimen.name.versionslice[2]
versionslice(specimen.data.name, 2) just slice metadata
```

```
{
  "specimen": {
    "versions": [
      ["name", "colloquial"],
      ["name", "colloquial", "habitat"]
    ],
    "timestamps": ["2015", "2017"],
    "types": ["object", "object"],
    "data": {
      "name": {
        "versions": ["Hieracium umbellatum"],
        "timestamps": ["2015"],
        "types": ["string"]
      },
      "colloquial": {
        "versions": ["Canadian Hawkweed", "Canadian Hawkweed, Narrowleaf Hawkweed"],
        "timestamp": ["2015", "2017"],
        "types": ["string", "string"]
      },
      "habitat": {
        "versions": ["2", "3"],
        "timestamp": ["2015", "2017"],
        "types": ["list", "list"],
        "data": {
          "1": ["rocky shoreline"],
          "2": ["conifer forest"],
          "3": ["sand dune"]
        }
      },
      "taxaAuthority": {
        "author": "Barkworth"
      }
    }
  }
}
```

Fig. 6.1: *Hieracium umbellatum* is related to *Narrowleaf Hawkweed*, as of 2018

```
specimen.version[2]

versionslice(specimen, 2) Must do a "deep dive" on specimen
```

TimeSnapshot -

```
specimen.name.timesnapshot[2015]

snapshot(specimen.data.name, 2015)


specimen.timesnapshot[2015]

snapshot(specimen, 2015) Must do a "deep dive" on specimen
```

The value of doing it this way is that other kinds of metadata can be supported.

Bibliography

[1] O. Etzion, S. Jajodia, and S. M. Sripada, Eds., *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, ser. Lecture Notes in Computer Science, vol. 1399. Springer, 1998. [Online]. Available: https://doi.org/10.1007/BFb0053695

[2] V. Radhakrishna, P. V. Kumar, and V. Janaki, "A survey on temporal databases and data mining," in *Proceedings of the The International Conference on Engineering & MIS 2015*, ser. ICEMIS '15. New York, NY, USA: ACM, 2015, pp. 52:1–52:6. [Online]. Available: http://doi.acm.org/10.1145/2832987.2833064

[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 *24th International Symposium on Temporal Representation and Reasoning, TIME 2017, October 16-18, 2017, Mons, Belgium*, 2017, pp. 1:1–1:37. [Online]. Available: https://doi.org/10.4230/LIPIcs.TIME.2017.1

[4] C. S. Jensen and R. T. Snodgrass, "Temporal database," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_395

[5] ——, "Transaction time," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1064

[6] ——, "Valid time," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1066

[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. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1426

[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. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, 1997, pp. 150–194. [Online]. Available: https://doi.org/10.1007/BFb0053702

[10] M. H. Böhlen and C. S. Jensen, "Sequenced semantics," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1053

[11] C. Combi, "Temporal object-oriented databases," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_404

[12] T. Amagasa, M. Yoshikawa, and S. Uemura, "A Data Model for Temporal XML Documents," in *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000, Proceedings*, 2000, pp. 334–344. [Online]. Available: http://dx.doi.org/10.1007/3-540-44469-6_31

[13] S. S. Chawathe, S. Abiteboul, and J. Widom, "Representing and Querying Changes in Semistructured Data," in *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, 1998, pp. 4–13. [Online]. Available: http://dx.doi.org/10.1109/ICDE.1998.655752

[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. [Online]. Available: http://dx.doi.org/10.1007/s00778-002-0079-4

[15] F. Rizzolo and A. A. Vaisman, "Temporal XML: Modeling, Indexing, and Query Processing," *VLDB J.*, vol. 17, no. 5, pp. 1179–1212, 2008. [Online]. Available: http://dx.doi.org/10.1007/s00778-007-0058-x

[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. [Online]. Available: http://dx.doi.org/10.1007/s11280-005-1317-7

[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. [Online]. Available: http://dx.doi.org/10.1016/j.datak.2007.09.003

[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. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.74

[23] C. E. Dyreson and K. G. Mekala, "Prefix-Based Node Numbering for Temporal XML," in *Web Information System Engineering - WISE 2011 - 12th International Conference, Sydney, Australia, October 13-14, 2011. Proceedings*, 2011, pp. 172–184. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24434-6_13

[24] S. Brahmia, Z. Brahmia, F. Grandi, and R. Bouaziz, "τjschema: A framework for managing temporal json-based nosql databases," in *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II*, 2016, pp. 167–181. [Online]. Available: https://doi.org/10.1007/978-3-319-44406-2_13

[25] M. H. Böhlen, "Temporal coalescing," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_388

[26] C. S. Jensen and R. T. Snodgrass, "Temporal element," in *Encyclopedia of Database Systems, Second Edition*, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1419