Temporal JSON Keyword Search

CURTIS DYRESON, Department of Computer Science Utah State University, USA
AMANI SHATNAWI, Department of Information Technology Yarmouk University, Jordan
SOURAV S. BHOWMICK, College of Computing & Data Sc. Nanyang Technological University, Singapore
VISHAL SHARMA, Department of Healthcare Admin. & Policy University of Nevada, Las Vegas, USA

JSON keyword search searches the current versions of documents in a collection. However, JSON documents change over time due to edits. Some applications, such as data forensics and auditing, need to search past versions of documents and for changes to documents. This paper introduces a system called *Temporal JSON Keyword Search* (TJKS) for search in a collection of JSON documents that vary over time. TJKS lets users control which *temporal slice*, or part of the history, can be searched using a *temporal search semantics*; we support both of the major temporal semantics: *sequenced* and *nonsequenced* search. This paper presents the semantics of temporal JSON keyword search, discusses an efficient implementation, and evaluates the implementation. Our extensions are largely orthogonal to specific keyword search techniques, so this research provides a blueprint for extending keyword search to include time and potentially other kinds of metadata.

CCS Concepts: • **Information systems → Database query processing**; *Temporal data*; Semi-structured data.

Additional Key Words and Phrases: Temporal, Keyword search, Sequenced, JSON

ACM Reference Format:
Curtis Dyreson, Amani Shatnawi, Sourav S. Bhowmick, and Vishal Sharma. 2024. Temporal JSON Keyword Search. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 177 (June 2024), 27 pages. https://doi.org/10.1145/3654980

1 INTRODUCTION

JavaScript Object Notation (JSON) has become one of the most widely-used format for representing and exchanging objects. Consequently, systems for storing and querying JSON data have also become increasingly popular. For instance, MongoDB currently ranks fifth in one ranking of popularity among all DBMSs [1].

Keyword search is one way to query a collection of JSON documents. Keyword search finds a fragment of a database that best matches the keywords in a query [31]. For JSON data, which is logically organized as a hierarchy, a branch of keyword search known as *hierarchical keyword search* [3, 8, 23, 30, 39, 43–47, 53, 60, 66, 69, 72] finds the objects that match the query. Keyword search on structured, semistructured, and unstructured data (*e.g.,* Google or Bing) has gained popularity primarily because it benefits lay users by relieving them from learning complex query languages (*e.g.,* JSONPath [2]). It is also helpful in exploring a data collection since it can be used without knowing the precise structure of the JSON data [42]. Keyword search is also used when

Authors' addresses: Curtis Dyreson, Department of Computer Science and Utah State University, Logan, Utah, USA, curtis.dyreson@usu.edu; Amani Shatnawi, Department of Information Technology and Yarmouk University, Irbid, Jordan, ashatnawi@yu.edu.jo; Sourav S. Bhowmick, College of Computing & Data Sc. and Nanyang Technological University, Singapore, Singapore, assourav@ntu.edu.sg; Vishal Sharma, Department of Healthcare Admin. & Policy and University of Nevada, Las Vegas, Las Vegas, Nevada, USA, vishal.sharma@unlv.edu.

ACM 2836-6573/2024/6-ART177
https://doi.org/10.1145/3654980

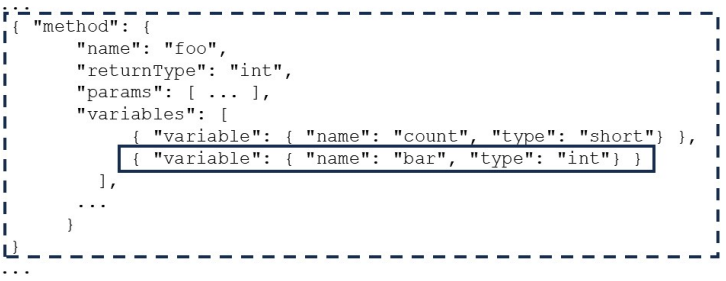

Fig. 1. A fragment of JSON representing objects in the text of a Java program

selecting objects for further processing and is more resilient to changes in the structure of the data than using path expressions to select objects. Last but not the least, it is also very flexible in the sense that it is able to provide good results with both highly-structured JSON as well as loosely-structured data.

As an example from the field of data forensics, consider a group of programmers trying to track down bugs in a source code repository (*e.g.,* a collection of GitHub projects). We will assume that the code is encoded in JSON to facilitate the search, with the JSON hierarchy corresponding to the nested, block structure of the programs. Figure 1 shows a small fragment of the JSON representation of some source code. The programmers want to find code with an integer variable named "bar." A keyword search query such as

```
variable int bar
```

is simply a list of keywords (keys or values) to look for in the data. Evaluation of the search query given above would find objects that contain a `variable`, `int`, and `bar`. In Figure 1 the object[1] marked by the dashed rectangle and the object marked by the solid rectangle match the keywords, but keyword search would choose the second as the "smaller" matching object [66].

Although there is a long stream of research on keyword search semantics and techniques, keyword search of a *time-varying* or *temporal* data collection has received scant attention from various research communities despite the importance of exploring the temporal aspects of data [10]. There has been extensive research in the management and querying of data annotated with time metadata *c.f.,* [58] and this research has impacted industry standards and practice with the addition of temporal support in SQL [40]. Canonical use cases for time include auditing, forensics, analysis of trends, and tracking the history of data changes. JSON data can change frequently over time as data is inserted, edited, and deleted [49, 62]. JSON stores like MongoDB and CouchDB support document versioning but the fine-grained history of data can be captured best in a *temporal JSON* document [16, 33, 67] or more generally in a temporal hierarchy [5, 18, 20, 28, 54, 64]. Temporal JSON has data annotated with time metadata that records when each object or value is alive.

Keyword search on *temporal JSON differs from non-temporal search* in both the front-end (functionality) and back-end (implementation) in several important ways that we outline below.

1.1 Front-end: New Kinds of Searches

Consider the example of a group of programmers trying to track down bugs in a source code repository. The programmers want to find `variables` of type `int` that changed to type `short`. The following keyword search, evaluated on an extant engine such as XSeek [46], ignores the time metadata and yields a superset of the desired information.

[1]In JSON, an object is a key/value pair set.

```
variable int short
```

It is a superset because it will include all temporal relationships between `int`s and `short`s when they are present at the same time, when `short` changed to `int`, or when separated in time by years. A tighter result can be achieved by evaluating a *temporal keyword search* such as the following (the temporal search syntax and semantics is described in Section 4).

```
variable @contains (int @meets short)
```

The two temporal constructors, `@meets` and `@contains`, are adapted from Allen's algebra [4]. They test the time metadata associated with the matched nodes. To qualify for the result an `int` node's lifetime must end exactly when a `short` node's lifetime begins. And the `int` and `short` lifetimes must be within the lifetime of a matching `variable`.

Next, the programmers want to find the `method`s that have the `int variable` `foo` in the code. Using XSeek, the following query will again produce a superset of the desired result.

```
method int variable foo
```

The result will be a superset because it cannot be ensured that the keywords are present in the data *at the same time* or in the same version. That is, there could be an `int variable` in some early version of a `method`, and only in a later version a `foo variable`. A temporal keyword search can achieve a tighter result.

```
@sequenced method int variable foo
```

The search is performed using `@sequenced` search semantics [12] which ensures that the keywords are present at the same time in the data, as described in detail in Section 4.2.

In addition to precisely controlling the temporal relationships between keywords and sequenced search, other new kinds of searches include the following.

- Retrieve the history of `int foo`s in 2015.

  ```
  @sequenced @slice(2015) variable int foo
  ```

- Find the earliest changes to `int foo`s.

  ```
  @earliest variable int foo
  ```

- Find how long `int foo`s lived in the data.

  ```
  @duration variable int foo
  ```

All of these searches must use the time metadata to constrain the search and find relevant data.

1.2 Back-end: New Kinds of Techniques

Existing keyword search techniques must be adapted when multiple versions of objects are in a temporal JSON document.

Many of the proposed keyword search algorithms model a document as a hierarchy and focus on using a common ancestor of the set of nodes that match the keywords to find and rank the best matches, *e.g.,* [44, 66]. In a time-varying data collection common ancestors may vary over time so previous techniques will not work. As an example consider the fragment of a temporal hierarchy created from a temporal JSON document shown in Figure 2. The figure shows a snapshot of the hierarchy at two instants in its history, t_1 and t_2. Each snapshot is a complete, non-temporal hierarchy. Suppose each snapshot performs a non-temporal keyword search for x and y independently. Assume that the search uses some technique that finds a relevant common ancestor (CA), such as SLCA [66], as the result. Observe that the CA w at time t_1 is different than the CA z at time t_2 because the objects in the snapshots are different. Figure 2 illustrates that the CA can vary over time, hence to find the CA a *temporal search* must, in effect, evaluate each snapshot independently, *i.e.,* for each time instant across the timeline. Of course it is impractical to compute

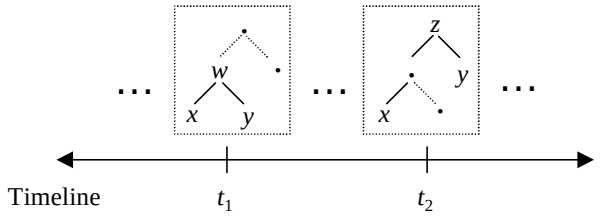

Fig. 2. The relevant common ancestor is w at time t_1, but z at time t_2

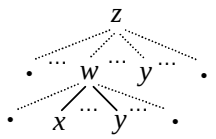

Fig. 3. The relevant common ancestor is w only, z is not lowest

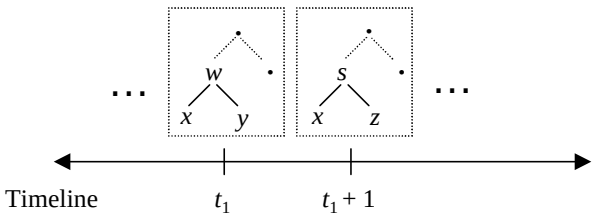

Fig. 4. Is w or s the relevant common ancestor?

the CA separately for each snapshot since there would be far too many such snapshots in a typical time-line, and such a computation would redundantly compute the same CA for many of the snapshots. Consider a different, more efficient representation of the temporal hierarchy that *merges* the common information across each snapshot as shown in Figure 3. We assume that each node in the hierarchy is annotated with metadata that records the lifetime of the node (the annotation is not shown in the figure). A non-temporal search in the merged representation will still be incorrect since it will compute the CA as node w only. SLCA will eliminate z from the result since w is a *lower* CA yielding a smaller result. Yet z is a CA at some times so z should be part of the result. A temporal search, unlike a non-temporal search, must be sensitive to the fact that CAs can vary over time in a temporal hierarchy.

A common ancestor may not be common. Even finding a CA can be challenging in temporal search. Consider a search for `@contains x (@meets y z)`, which is colloquially equivalent to "x and (y changes to z)", on the two snapshots shown in Figure 4. The search looks for a change to the document from y to z and keyword x remaining constant. Such a search can only be performed between snapshots since it requires y to be present in an earlier snapshot and absent in a later one, and vice-versa for z. Snapshots t_1 and $t_1 + 1$ fit the change criteria but should the CA be w, s, both, or neither? Note that neither w nor s are ancestors of both y and z. Since the "best" result for the search starts with finding the CA it is important for temporal searches to identify CAs across changing snapshots, and for temporal search to be able to find changes to matching nodes.

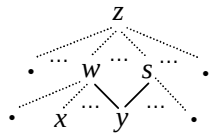

Fig. 5. The node y has moved so has two parents, w and s

The model for a temporal JSON document is a partial order. Keyword search techniques generally assume the hierarchy is a tree (or forest). But in a temporal hierarchy a node may move to a new parent, so even if each slice of the hierarchy is a tree, a node may have more than one parent. Figure 5 shows node y moved in the merged representation, with two parents, w and s. So techniques for non-temporal keyword search, which all assume a tree, need to be generalized to use in a temporal JSON document.

1.3 Contributions

In summary, hierarchical keyword search on time-varying data needs to address the following challenges.

- Common ancestors are critical to hierarchical keyword search and, unlike in a nontemporal document, in a temporal document an ancestor may be a common ancestor or not, at different times, due to several factors.
  - A node may switch common ancestors over time because its partner in determining the common ancestor is deleted or another partner is inserted.
  - A node may have multiple parents over time because it has been be moved in the hierarchy.
  - Nonsequenced search differs from sequenced search in determining a common ancestor, because in the latter case the common ancestor and all its relevant descendants must live in the document at the same times, but not in nonsequenced search.
- Search queries need to support both nonsequenced and sequenced semantics.

Recognizing the importance of exploring how data changes over time [10] this paper introduces *Temporal JSON Keyword Search* (*TJKS*), a keyword search system for time-varying JSON documents. TJKS is novel in that it is the first keyword search system that supports *sequenced semantics* [13, 59], an important branch of temporal semantics, and variants such as *earliest semantics* [29]. Examples of these kind of keyword searches that can be evaluated with TJKS were given previously. As described above supporting these semantics requires new back-end techniques, which this paper also describes. Additionally TJKS supports *nonsequenced semantics* [14, 15], which has been previously explored in the context of graph databases [48, 65]. TJKS works on hierarchical data in general, both JSON and XML are supported; for keyword search the kind of hierarchical data is of little importance. Finally, we believe that tracking versions of objects is important and so TJKS also is unique in supporting keyword search on "version clocks" as well as other kinds of clocks.

In summary, this paper makes the following contributions.

- We present a taxonomy that classifies the different kinds of keyword search (Section 2).
- We formalize a temporal data model to support search on a changing JSON document (Section 3).
- We describe the semantics of temporal search (Section 4).
- We present an algorithm to compute a temporal search efficiently (Section 5).
- We implement and experimentally evaluate the algorithm (Section 6).

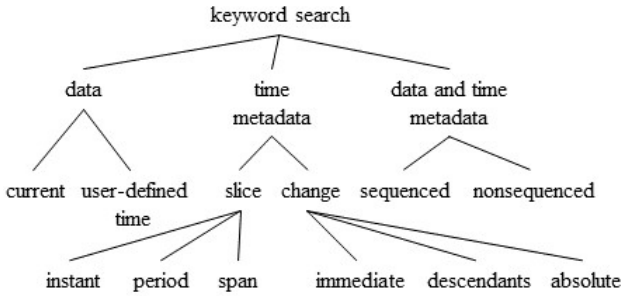

Fig. 6. A taxonomy of temporal search

2 RELATED WORK

Keyword search is popular because it relieves users from having to know a complex query language or the structure of the underlying data. Previous research focused on two areas. One research area developed techniques to connect nodes that match the search keywords. Specifically, this research involved finding parts of a document containing the nodes that match the search terms [23, 44, 60, 66]. The second research area investigated how to process nodes within the matching document fragments to produce relevant and coherent results [8, 39, 43, 46, 47]. But there has been little research on search in time-varying or temporal keyword search.

Time data and metadata are involved in searches in different ways. The taxonomy in Figure 6 is a classification of the kinds of keyword search. Below we give a description of each classification branch in the taxonomy.

data, current – A keyword search on a non-temporal data collection. It is the kind of search previously researched, *c.f.,* [46], *e.g.,* find `int foo variable`.

data, user-defined-time – A data search that uses a time present in the data, *e.g.,* find `int foo 1/1/2021` where `1/1/2021` is a time value stored in the (current) document collection. Non-temporal search techniques can process user-defined times, *e.g.,* [51].

time metadata, slice – Restrict the history to a *slice* of the data. The data can be sliced in different clocks (see Section 3 for a description of the clocks) and using either an instant (a single time), period (a pair of times), or span (a duration of time) [37]. For example, find `int foo variable` in the valid-time slice of `2021` would perform the search only within the data alive at time `2021` (within that year or with the time metadata interpreted in the granularity of years). Previous research in keyword search in temporal graph has considered slicing [48].

time metadata, change – Similar to a slice but uses only nodes that span a change to their immediate content, descendants, or anywhere in a document (absolute). For example, find changes to `int foo variable`. Keyword search in versioned text focuses on changes in the temporal metadata [6].

data and time metadata – Sequenced and nonsequenced are standard semantics for temporal queries and operations [11, 12, 14, 15]. We describe both kinds of search in more detail in Section 4.4, but, broadly, a sequenced search performs a search on the history of a temporal data collection, *e.g.,* find the history of `int foo variables` whereas a nonsequenced search can compare data alive at different times, *e.g.,* find `int short foo variable` where `int` must temporally precede `short`. Nonsequenced keyword search has been explored in the context of graph databases by imposing a hierarchy on the graph as part of the search [48, 65]. Issues related to versioning of the nodes in the graph were not addressed.

Our paper addresses the sequenced category in the taxonomy, as well as covering the other categories. We are not aware of any previous research in sequenced keyword search. Note that the taxonomy is not exhaustive, for instance, it excludes search for data that changes over time, *e.g.,* find parts that increased in price in 2022. It also does not consider how to include time in ranking search results *c.f.,* [38].

Keyword search is not the only kind of search on JSON or XML data, similarity search has also been popular [7, 36, 70]. The kinds of search are different, but the techniques presented in this paper could be adapted to similarity search. We are unaware of any work on temporal similarity search.

There has been extensive previous research in temporal databases. Most of this research has focused on query languages [19, 27, 34, 54, 57, 58, 61], indexing [9, 26, 52, 54, 55, 64, 71], and modeling [24, 35, 63, 64]. Temporal keyword search has not been previously research by the temporal database community.

3 A TEMPORAL JSON MODEL

Central to temporal search is a data model that captures the history of a JSON document. We first define a model for a JSON document, then extend it to a temporal model.

A non-temporal document can be modeled as an ordered, labeled tree.

Definition 1 (JSON data model instance). *A JSON document, D, is modeled as a tuple, $(V, E, \Sigma, \mathbb{L})$ where*

- *V is a set of nodes,*
- *$E \subseteq V \times V$ is a set of edges,*
- *V and E form a tree,*
- *Σ is an alphabet of labels and text values, and*
- *$\mathbb{L} : V \rightarrow \Sigma$ is a label/text function that maps a node to its label or text content.*

□

To capture the history of a changing document, we need to keep track of each node's lifetime. We define a *temporal JSON instance* as follows. In the instance, an *item* is a node that has continuity over time [25, 33]. Without loss of generality we assume a single time dimension (note that a time in the dimension could be composed of measurements from several clocks which are described below). We assume the time dimension is well-ordered, and could be discrete, dense, or continuous.

Definition 2 (Temporal JSON data model instance). *A temporal instance, I_T, is modeled as a tuple,*

$$(I, T, \mathbb{I}, S, \mathbb{M}, V_T, E_T, \Sigma_T),$$

constructed from a document, $D = (V_T, E_T, \Sigma_T, \mathbb{L}_T)$ where

- *I is a set of items,*
- *T is a set of times,*
- *$\mathbb{I} : V_T \rightarrow T \rightarrow I$ is an item identity function that maps each element node in V_T at a time in T to an item in I,*
- *$S \subseteq I \times I \times T$ is a set of super edges such that if $(v, w) \in E_T$ at time $t \Rightarrow (\mathbb{I}(v, t), \mathbb{I}(w, t), t) \in S$,*
- *$\mathbb{M} : V_T \rightarrow T \rightarrow \Sigma$ is a time-varying label/text function that maps a node in V_T at time t to its label or text content, and*
- *V_T, E_T, and Σ_T are the union of the corresponding sets in the document over time.*

□

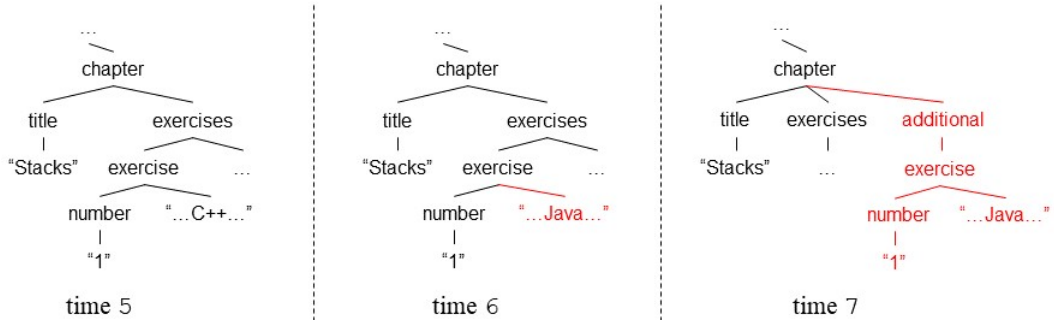

Fig. 7. JSON data model instances over time

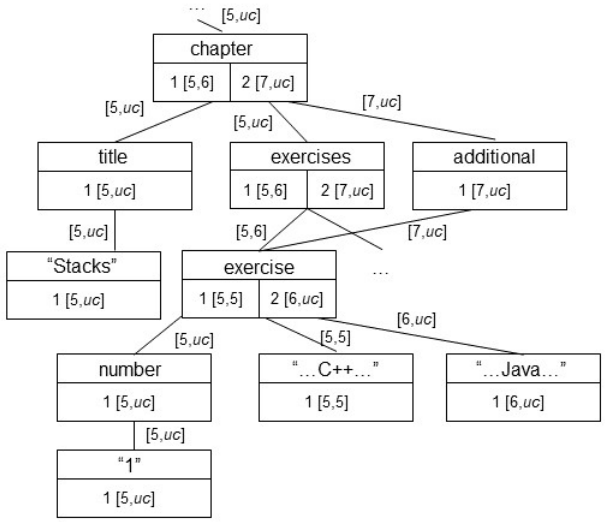

Fig. 8. Item-versioned temporal instance

As an example, consider the three JSON data instances depicted in Figure 7. Changes to the snapshots are depicted in red. At time 5, part of a book `chapter` on "`Stacks`" is shown. The first `exercise` is about "`C++`." At time 6, the `exercise` was edited to be about "`Java`." Then at time 7, the `exercise` was moved into a new section on `additional` materials. With respect to Definition 2, V_T is the union of the set of nodes in each instance. Note that each node in each instance is unique and has no connection to a node is another instance. E_T is the union of the set of edges. The instances are glued into a temporal instance shown in Figure 8. Each node is an item, depicted as a box in Figure 8. Super edges connect the items. Through items, nodes (and edges) have identity and connection over time, and lifetimes can be constructed for them. With respect to Definition 2, $\mathbb{I}$ connects each node in an instance to an item. So the `exercise` node in the instance at time 5 is mapped to the `exercise` item. Super edges, set S in Definition 2, are depicted in the Figure 8 as edges that connect the items. Finally $\mathbb{M}$ maps an item (over some lifetime) to the text (value or key name) associated with that item. So the `chapter` item over the lifetime of all the instances maps to the text "`chapter`." Observe that the `exercise` item has two parents: the `exercises` item and the `additional` item. The times (and *versions* of items) in the figure are described below.

The temporal database community has recognized three primary kinds of time: transaction, valid, and user-defined time [37]. *Transaction time* is the time for which a fact is "live" in a data collection, usually the database time between when it is inserted and deleted. *Valid time* is the real-world time of a fact. Both valid and transaction time are kinds of *metadata*, that is, they are data *about* a fact stored in a database. In contrast, *user-defined time* is a time value within a fact, *i.e.,* user-defined time is data that happens to be a time. For example consider the data that Sue was born in 1978, is employed at the ACME company from 2020 through 2021, and that this data was inserted into the data collection in 2022. One way to model Sue's employment is with a user-defined time for his date of birth (1978), a valid time period of [2020-2021] representing the real-world time of his employment, and a transaction time period of [2022-*uc*] (since the data is current and has not yet been deleted, it lives in the database "until changed (*uc*)" [22]).

There are several potential kinds of clocks that can provide times.

- **valid-time clock** - Valid time is the real-world time of the data. Valid times range from *timeMin* to *timeMax* representing the maximum valid time. A valid time period, $[b, e]$, is the time from b to e, inclusive [37].
- **transaction-time clock** - Transaction time represents the time in the data management system. It is the time during which the data was alive in the system. Transaction times range from *timeMin* to *uc*. We will represent a transaction time as a temporal period. In Figure 8 the times, *e.g.,* `[5,`*uc*`]` are from the transaction-time clock.
- **version clock** - Each change to the instance creates a new version of that instance. There is one version clock for the instance. We will represent a version time as a temporal period.
- **child-version clock** - Each change to the immediate content of a node creates a new version of that node. The child version clock records the version. It is a *relative* clock, relative to a given node. We will represent a child version time as version number. Child-versions are depicted inside each item in Figure 8. For example, the `chapter` item has two child versions, one at transaction time `[5,6]` and the second at time `[7,`*uc*`]`.
- **descendant-version clock** - Each change to any descendant advances the time on the descendant-version clock for that node. The descendant-version clock is also a relative clock.

We assume that a temporal instance could utilize some or all of the different clocks, that is, a time could be a list of time from different clocks. For example, Figure 8 has times from the transaction-time and version clocks only.

4 TEMPORAL SEARCH SEMANTICS

This section gives a semantics for *search on a temporal instance.* A straightforward implementation of the semantics would be inefficient so in Section 5 we describe how we implement temporal search. We first review the semantics of non-temporal search.

4.1 Data, Current Search

Given a search query $Q(w_1, \ldots, w_k)$, the keyword search process can be broadly divided into three steps:

(1) *match* the keywords, $w_1, \ldots, w_k$, to nodes in the data instance(s),
(2) compute *result nodes*, and
(3) *retrieve* results.

The first step processes the keywords in a query to find matching nodes. If a node is mapped by $\mathbb{L}(n)$, the label and text function for a data instance, to a string containing a keyword w_i then the node n is called a *match* of w_i. We denote a set of matches to w_i in instance D as $M(D, w_i)$. The second step, computing result nodes, is at the heart of keyword search. The step computes

result nodes, $R(Q, D)$, from instance D satisfying Q. Most result node computations find a common ancestor (CA) among the set of matching nodes that meet some condition [44, 60, 66]. This paper is agnostic about the specific CA-based approach used.

4.2 Temporal Search

So what makes a search *temporal*? Temporal database research has identified two distinct temporal semantics: *sequenced* and *nonsequenced* [13, 59]. In this section, we describe sequenced and nonsequenced search semantics for the three steps described above: match, compute result nodes, and retrieve results.

4.3 Temporal Matching

Temporal search uses the same matching process as non-temporal search. The temporal match operation takes a list of keywords and finds the nodes that match each keyword. Each node in the result also retains its times from the temporal instance.

Definition 3 (Temporal Match). *Given a temporal instance,*

$$I_T = (I, T, \mathbb{I}, S, \mathbb{M}, V_T, E_T, \Sigma_T),$$

and a list of keywords $w_1, \ldots, w_n$ the match is the Cartesian product of the nodes that are labelled with each keyword:

$$\textbf{match}(I_T, [w_1, \ldots, w_n]) = \\ \{(v_1, t_1) \mid (v_1, t_1, w_1) \in \mathbb{M}\} \times \ldots \times \{(v_n, t_n) \mid (v_n, t_n, w_n) \in \mathbb{M}\}.$$

□

For example,

match(D, [`title`]),

when applied to the instance in Figure 8 results in a set consisting of the single `title` node.

4.4 Temporal Result Nodes Computation

Non-temporal result nodes computation finds the relevant CA(s) for combinations of nodes produced by match. Temporal result nodes computation adds the ability to enforce various temporal semantics by utilizing the time metadata. We begin the discussion with sequenced semantics, which is the easiest semantics to understand.

4.4.1 Sequenced. Sequenced semantics, which we sketch in Figure 9 stipulates that the meaning of a temporal operation, such as a temporal result nodes computation, is given in terms of the previously-known semantics for the corresponding non-temporal operation. Imagine a temporal instance, I_T, *sliced* into *snapshots*, $I_1, \ldots, I_n$, for each instant, $1, \ldots, n$, in the instance's history (the *slice* operation is described below). Sequenced result nodes computation on a temporal instance, I_T, produces a result, R_T, that must contain the same slices as running the non-temporal result nodes computation on each snapshot, I_i, to produce result R_i. Hence, a sequenced computation is *snapshot-reducible* [13, 57] to the application of non-temporal computation on each snapshot.

The *slice* operation slices a temporal instance, which is a partial order, into an item hierarchy by following super edges. Slice keeps the items and super edges visited in a traversal of a temporal instance.

Definition 4 (Slice). *A slice is the items and edges visited in a traversal of a temporal instance, I_T, during a set of times, $\{t_1, \ldots, t_n\}$, together with the metadata from I_T following super-edges only*

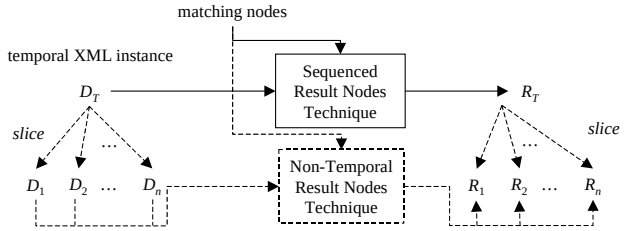

Fig. 9. Sequenced search semantics

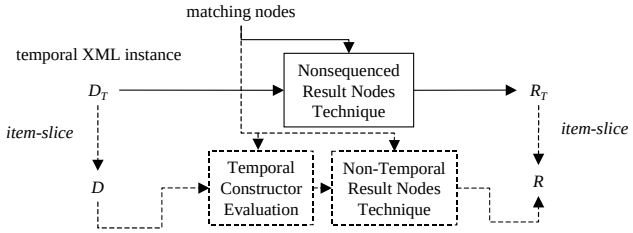

Fig. 10. Nonsequenced search semantics

alive during X and the parent's lifetime. Given

$$I_T = (I, T, \mathbb{I}, S, \mathbb{M}, V_T, E_T, \Sigma_T),$$

the slice of I_T *for times* $\{t_1, \ldots, t_n\}$ *is*

$$\textbf{slice}(I_T, \{t_1, \ldots, t_n\}) = (V_I, E_I, \Sigma_T, \mathbb{M}, \mathbb{T})$$

where

- $\mathbb{T} = \{(v,t) \mid (\mathbb{I}(p,t), \mathbb{I}(v,t), t) \in S \wedge$
  $t \in (\{t_1, \ldots, t_n\} \bigcap \{t_p \mid (p, t_p) \in \mathbb{T})\}$
  ($\mathbb{T}$ ensures that super edges are followed only during the set of times and the parent's lifetime),
- $V_I = \{v \mid (v,t) \in \mathbb{T}\}$
- $E_I = \{(w,v) \mid (\mathbb{I}(w,t), \mathbb{I}(v,t), t) \in S \ \wedge \ (v,t) \in \mathbb{T}\}$.

□

Note that the slice results in a hierarchy, since an item is split into different nodes if it can be reached by following different super edges (at different times), that is, $t \neq s \implies v_t \neq v_s$. Note also, that the slice creates a hierarchy in which the CA is well-defined. Hence, non-temporal keyword search techniques that utilize the CA can be applied to the slice (the times on the nodes are inert in the non-temporal search).

The slice of Figure 8 for $\{5, 6, 7\}$ is shown in Figure 11. Note that the items `exercise`, `number`, "`Java`" and "1" appear in two places since they moved. Removing the time metadata from the slice at each instant in Figure 8 produces the snapshots in Figure 7.

Consider the sequenced result nodes computation of `title` and "`Java`" on Figure 11. At time 5, "`Java`" is not matched. At times 6 and 7, the CA of the text node "`Java`" and the `title` node is the `chapter` node. So the *coalesced* result [37] is the `chapter` at times `[6,7]`. In effect, this is equivalent to first slicing the hierarchy using `[6,7]`, then finding the CA. The time `[6,7]` is constructed by taking the intersection of the lifetime of the "`Java`" and `title` nodes. We will call the constructed time the *slice time*. The CA is annotated with the slice time to represent the time(s) when it is a CA.

4.4.2 Nonsequenced. Sequenced can be generalized to *nonsequenced*, which compares data not necessarily alive at the same time, by allowing the user to construct any time of interest for the slice time yielding the following general definition.

Definition 5 (Temporal Result Nodes Computation). *Let C be a temporal constructor and $\mathbb{R}$ be a technique to compute the "best" CA, e.g., SLCA. Then the temporal result is*

$$\begin{aligned}&\textbf{temporalResult}(I_t, \mathbb{R}, C) = \\ &\quad \{\ (v,t) \mid [(v_1,t_1),\ldots,(v_n,t_n)] \in \textbf{match}(I_T, [w_1,\ldots,w_n]) \\ &\qquad \wedge\ t = C([t_1,\ldots,t_n])) - \\ &\qquad\quad \bigcup\{t_c \mid (v_c,t_c) \text{ is a better CA than } (v,t)\ \} \\ &\qquad \wedge\ v = \mathbb{R}([(v_1,t_1),\ldots,(v_n,t_n)], \textbf{slice}(I_t,t)) \\ &\quad \}\end{aligned}$$

□

For instance, Consider the nonsequenced search of "`C++`" and "`Java`" on the slice from `[5,7]` as shown in the hierarchy in Figure 11, corresponding to the snapshots of Figure 7. Suppose that the slice time is constructed as the *union* of the lifetimes of "`C++`" and "`Java`." Then the CA at times `[5,6]` is the `exercise` node, and at times `[5,7]` it is the `chapter` node. But many techniques, such as SLCA [66], further refine the CA. For instance SLCA chooses the *lowest* CA as the "best" CA. So for temporal SLCA, the slice time of a descendant CA should be removed from that of an ancestor CA. For instance, the time of the `chapter` node would be reduced to `[7,7]` since the lower `exercise` node is the SLCA at times `[5,6]`.

The temporal semantics is sketched in Figure 10. The temporal semantics is similar to the sequenced semantics, but adds an explicit temporal constructor and temporal pruning.

4.5 Temporal Retrieve Results

The final step in search is to retrieve the result. Though different techniques have been proposed, each technique retrieves a subset of the descendants of some result node identified by the previous step [60]. Among the descendants included in the result are (at least some of) the nodes that matched each keyword in the query.

Definition 6 (Non-temporal result). *Let*

$$M(D,w_1),\ldots,M(D,w_k)$$

be the sets of nodes that match the query keywords, $w_1,\ldots w_k$, and let $R(Q,D)$ be the set of result nodes for query Q on instance, D. Then for each $r \in R(Q,D)$, the result, $X(D,r) \subseteq \textbf{desc}(D,r)$ and

$$X(D,r)\ \cap\ (M(D,w_1)\ \cup\ \ldots\ \cup M(D,w_k)) \neq \emptyset$$

where $\textbf{desc}(D,r)$ is the set of descendants of r in D. The result is some subset of the descendants of r and must include at least some of the nodes that matched the keywords among those descendants. □

Temporal search changes the result in two ways. First, versions can be recomputed with respect to the result. Note that in retrieving the result (some subset of) the descendants of the result node are visited. Descendant-version and child-version clocks can be computed if desired during the visit. Second, descendants of the result nodes can be visited in the context of some time. The remainder of this section presents the second change in more detail.

In sequenced search only descendants with a lifetime that intersects with the result node are kept.

Definition 7 (Sequenced result). *Let $M_i(I_T,w_i,t_i)$ be the set of items or nodes that temporally matches the query keyword w_i at time t_i in the temporal instance I_T. Let $R_S(Q,I_t)$ be the set of*

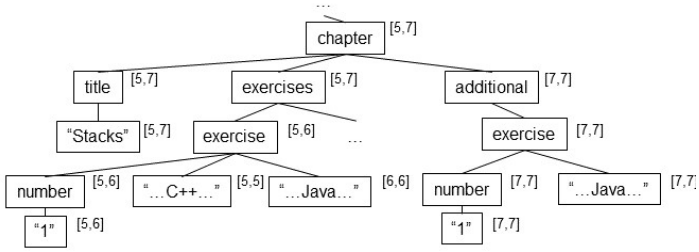

Fig. 11. The slice of Figure 8

result nodes for the sequenced search using query Q. For each $r \in R_S(Q, D)$, let y be the time of r and let D_y be the slice of I_T at time y, that is $D_y = \mathbf{slice}(I_T, y)$. Then the sequenced result, X_S, is $X_S(I_T, r) \subseteq \mathbf{desc}(D_y, r)$ where

$$X_S(I_T, r) \cap (M_T(I_T, w_1, t) \cup \ldots \cup M_T(I_T, w_k, t)) \neq \emptyset.$$

□

As an example, consider the sequenced result for a search for `additional` material on the temporal instance in Figure 8. The search would find the `additional` item. The sequenced result would slice the temporal instance at time `[7,`*uc*`]`, pruning the "`C++`" text node and modifying the timestamps to `[7,`*uc*`]` for the other descendants in version 1 of `additional`.

The nonsequenced result does not enforce the slicing at time t, and so constructs the result like non-temporal search.

Definition 8 (Nonsequenced result). *Let the set of items or nodes that temporally matches the query keyword w_i at time t_i in the temporal instance I_T be $M_i(I_T, w_i, t_i)$. Let $R_N(Q, I_t)$ be the set of result nodes for the nonsequenced search using query Q. Then for each $r \in R_N(Q, D)$ the nonsequenced result is $X_N(I_T, r) \subseteq \mathbf{desc}(I_t, r)$ such that*

$$X_N(I_T, r) \cap (M_T(I_T, w_1, t) \cup \ldots \cup M_T(I_T, w_k, t)) \neq \emptyset.$$

□

As an example, consider the nonsequenced result for a search for `additional` material on the temporal instance in Figure 8. The search would find the `additional` item. The nonsequenced result visits the descendants, potentially including each as is (without pruning using the time).

4.6 Search Functions

The three steps in the search can be expressed using the following functions.

search	→ *kind* `(` *temporalExpr* `)`
kind	→ `sequenced` | `nonsequenced`
temporalExpr	→ *term* [*temporalCons temporalExpr*]
temporalCons	→ `before` | `after` | `...` | `contains`
term	→ *slice* | *changes* | `keyword`
slice	→ `slice(` `keyword` `,` `time_literal` `)`
changes	→ `changes(` `keyword` `)`

The *slice* function performs temporal matching. The *change* function is also a temporal matching, but slices based on the times at which the matched `keyword` changes. For nonsequenced search, a temporal expression is built from *slice* and *change*

- `seqResults`$([r]) = [y]$ - Retrieve sequenced results from a list of result nodes, $[r]$.
- `nonseqResults`$([r]) = [y]$ - Retrieve nonsequenced results from a list of result nodes, $[r]$.
- `descVersions`$([r]) = [y]$ - Retrieve results from a list of result nodes, $[r]$, and compute versions using the descendant-version clock.
- `childVersions`$([r]) = [y]$ - Retrieve results from a list of result nodes, $[r]$, and compute versions using the child-version clock.
- `sequenced`$(P, [a_1], \ldots, [a_k]) = [r]$ - Perform sequenced search on the node lists $[a_1]$ to $[a_k]$ using predicate P on the times of nodes taken from the lists.
- `nonsequenced`$(P, [v_1, \ldots, v_k]) = [r]$ - Perform nonsequenced search on the node tuples v_1 to v_k, where each node tuple, $v_i = (n_1, \ldots, n_j, t)$ is a list of nodes, n_i, and a timestamp, t, and predicate P.
- *cons*$([a], [b]) = [n]$ - Apply the temporal constructor, *cons*, to the Cartesian product of $[a]$ and $[b]$,

$$[n] = [\ (v_a, v_b), cons(t_a, t_b)) \mid (v_a, t_a) \in [a]\ \wedge\ (v_b, t_b) \in [b]\ ]$$

- `slice`$([a], t) = [\ (v_a, \text{intersects}(t_a, t)) \mid (v_a, t_a) \in [a]\ ]$ - Slice the list of timestamped nodes at time t. Results in a list of timestamped nodes.
- `match`$(k) = [n]$ - Match keyword k at all times, returns a timestamped list of matching nodes, $[(n_1, t_1), \ldots, (n_k, t_k)]$.
- `temporalMatch`$(k, t) = [n]$ - Match keyword k at time t, returns a timestamped list of matching nodes.
- `descChanges`$(k) = [n]$ - Temporal matching of descendent changes to keyword k, *e.g.,* each descendant version (except the first) in a matched item/node is in the result timestamped with the time when it changed.
- `childChanges`$(k) = [n]$ - Same as `descChanges` but with child versions.
- *temporal predicate*$(k_1, k_2) = [n]$ - Apply temporal predicate, *e.g.,* meets, overlaps, contains, etc., to pairs of nodes from k_1 and k_2.

We now give some example queries.

- Find *current*, *i.e.,* those alive *now*, `exercises` about "`C++`". Note that `uc` represents *until changed* [37], that is, the current time.

```
seqResults(
  sequenced(overlaps,
    temporalMatch(<exercise>, uc),
    temporalMatch("C++", uc)
  )
)
```

  If executed on the temporal instance shown in Figure 8, "`C++`" is not current, so the answer would be empty.
- Find the history of "`C++`" `exercises`.

```
seqResults(
  sequenced( overlaps,
             match(<exercise>), match("C++")
           )
)
```

- Find `exercises` that changed from "`C++`" to "`Java`."

```
nonseqResults(
  nonsequenced(
```

```
      contains(
        match(<exercise>),
        meets( match("C++"), match("Java") )
      )
    )
  )
```

- Find changes to "C++" exercises in times [5,6] and reconstruct descendant versions.

```
descVersions(
  seqResults(
    sequenced(
      overlaps,
      descChanges(temporalMatch(<exercise>, [5,6])),
      temporalMatch(match("C++"), [5,6])
    )
  )
)
```

- Find chapter changes that happened after changes to "C++" exercise.

```
nonseqResults( nonsequenced(
    after(
      descChanges(<chapter>),
      intersects(
        descChanges(<exercise>),
        match("C++")
      )
    )
  )
)
```

5 AN EFFICIENT ALGORITHM

In this section we describe how to efficiently implement temporal search. There are two challenges. First, existing SLCA techniques do not support time-varying SLCAs. Unlike a nontemporal search, in a temporal search the SLCA for a set of matching keywords could vary over time. We show how to reuse a common node numbering scheme to efficiently find temporal SLCAs. The second issue is how to search the partial order.

A sketch of the pipeline for temporal search is given in Figure 12. JSON documents are read and merged into a temporal document offline. The temporal document is shredded and stored in a database, and some indexes to improve performance are created. When a user evaluates a keyword search, the search first matches the keywords to items in the document, then (as described in this section) determines the temporal SLCAs, *i.e.,* the smallest parts of the document that best fit the search parameters. Once the SLCAs are found, the results are constructed by walking (a part of) the subtree rooted at each SLCA.

5.1 Review of Prefix-Based Numbering

A popular node numbering scheme for hierarchical data instances is *prefix-based numbering* (PBN) (also called *Dynamic-level numbering*, *Dewey order*, *containment encoding*, and *Dewey numbering*) [32]. In PBN each node in an instance is numbered as follows: a node is numbered $p.k$, where p (the prefix) is the number of its parent and k represents that it is the k^{th} sibling in document

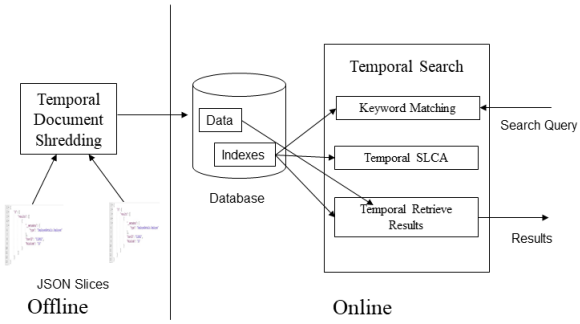

Fig. 12. An architecture for TJKS

Algorithm 1: SLCA by merging

Input: $V_1, \ldots, V_n$
Output: S: a list of SLCAs

```
1  List S = [] // Initialize the SLCA list
2  Node c // The candidate SLCA
3  // Create a priority queue for the next node
4  PriorityQueue Q = new Queue(V_1, ..., V_n)
5  // Process the keywords
6  while Q.hasNext() do
7      // Get a candidate SLCA, x, from Q
8      (v_1, ..., v_n) = Q.next()
9      x = LCA(v_1, ..., v_n)
10     // Figure out if x is an SLCA
11     if not(x.isAncestorOrDescendant(c) then
12         S.add(c)
13         c = x
14     else
15         if x.isDescendant(c) then
16             c = x
17         else
18             // x is an ancestor of c, so not a viable SLCA
19 S.add(c) // add the candidate
```

order. There are strategies for making PBN numbers relatively concise [41], able to be efficiently updated [68], and fast for LCA computation [56]. Given two PBN numbers we can quickly compute (by comparing the numbers) their LCA. The LCA is the common prefix. For instance, the LCA of $p.1.1.2$ and $p.1.2$ is $p.1$.

5.2 SLCA Computation

PBN numbers are used to quickly compute the SLCA of the nodes that match a keyword as follows. Let lists $V_1, \ldots, V_n$ be lists of nodes ordered by PBN that match keywords $k_1, \ldots, k_n$. Algorithm 1 presents the algorithm for computing the SLCAs by merging the lists of matching nodes. The algorithm sets up a priority queue, Q, that produces the next candidate tuple of nodes, $(v_1, \ldots, v_n)$,

Algorithm 2: Temporal SLCA by merging

```
Input: V_1, ..., V_n
Output: S: a list of SLCAs
List S = [] // Initialize the SLCA list
List C = [] // List of candidate SLCAs, one for each level
List L = [] // List of candidate SLCAs lifetimes
List X = [] // List of candidate SLCAs lifetime exclusions
PriorityQueue Q = new Queue(V_1, ..., V_n)
while Q.hasNext() do
    // Get a candidate temporal SLCA, x, from Q
    ((v_1, t_1), ..., (v_n, t_n)) = Q.next()
    (x, t_x) = temporalLCA((v_1, t_1), ..., (v_n, t_n))
    // Figure out if x is an SLCA at some level
    Integer i = C.size()
    while i > 1 do
        if not(x.isAncestorOrDescendant(C[i]) then
            // Calculate the time for the SLCA
            Time t_t= L[i] − X[i]
            // Add to result list
            S.add((C[i], t_t))
            C[i] = x
            L[i] = t_x
            // Add to the exclusion at higher levels
            Integer k = i
            while k > 1 do
                X[k−−].union(t_t)
        else
            if x.isDescendant(C[i]) then
                C[x.level()]=x
                L[x.level()]=x_t
            else
                // x is an ancestor of c, so not a viable SLCA
        i−−
// add the remaining candidates
∀i, 0 ≤ i ≤ C.size[S.add((C[i], L[i] − X[i]))]
```

one node drawn from each list, $V_1, \ldots, V_n$. Each time a candidate is chosen, the next node in some list is consumed. Suppose for instance that two keywords match the following lists of nodes, (1.1.2, 1.3.4), (1.1.4, 1.2.5, 1.3.7). The algorithm first chooses the first node from each list as a candidate tuple, (1.1.2, 1.1.4). The LCA of the tuple is 1.1, which becomes the candidate, c. The next node (in order) is then chosen from one of the lists, producing the tuple (1.1.2, 1.2.5) since 1.2.5 comes before 1.3.4 in PBN order. The LCA of that tuple is 1, which is an ancestor of 1.1, and so it is ignored. Next the algorithm chooses candidate tuple, (1.3.4, 1.2.5), which again produces the LCA 1, which is ignored. Finally, the algorithm chooses candidate tuple, (1.3.4, 1.3.7), which produces the LCA 1.3, which is neither an ancestor or descendant of the candidate, so 1.1 is added

to the list of SLCAs, S, and 1.3 becomes the new candidate, c. Finally, the loop exits since the first list has been exhausted, and the candidate, 1.3, is added to the list of SLCAs.

The complexity of the algorithm is $O(NM \log(N * M))$ where N is the length of the longest list and M is the number of lists. The loop iterates $O(NM)$ times, and each iteration costs $O(\log(N*M))$ to calculate the next candidate tuple in the priority queue.

5.3 Indexes in SLCA Computation

There are two indexes that can be used to improve the speed of SLCA computation. The first index, called the *keyword index*, maps a keyword to a list of nodes, ordered by the PBN number, which match the keyword in the document. The index is used to populate the priority queue with the list of nodes for each keyword in the search. Some keyword search systems, *e.g.,* MESSIAH [60], use the node's type rather than the keyword (the type is the list of keys on the path to a node), because the type is important to determining the specific kind of SLCA for the search. Such systems often have an additional index that maps a keyword to a list of types which match the keyword. A second index, called the *node index*, maps a PBN number to the string value for the node, *i.e.,* the name of the key for the node or its text value. This index is used when building the result by traversing the subtree rooted at the node.

5.4 Temporal SLCA

The temporal SLCA algorithm must account for SLCAs that vary over time, that is, an ancestor of an SLCA item could itself be an SLCA at a different time. For example, suppose 1.3.4.6 is an SLCA from time 2 to time 5. Normally, 1.3.4.6 being an SLCA precludes ancestors such as 1.3.4 and 1.3 from being SLCAs. But at time 1, it could be that 1.3.4 is an SLCA, and at time 6, 1.3 could be one.

The temporal SLCA algorithm, presented as Algorithm 2 keeps track of the time-varying ancestor SLCAs. It keeps a list, C, of candidate SLCAs, one for each level in the hierarchy, a list L, of each candidate's lifetime, and a list X, of each level's lifetime exclusion (the lifetime hidden by SLCAs at lower levels in the tree). The algorithm proceeds by finding a candidate SLCA. It adds the candidate to C at the appropriate level, and adds the candidate's lifetime to L, again at the appropriate level. An SLCA is computed by comparing a new candidate to the candidates at each level. If an SLCA is found, then the lifetime of the SLCA is the level's lifetime except the lifetimes of all lower levels. Note that the root is not an SLCA, so the root level (level 1) is not part of the computation.

The complexity of the algorithm is $O(NMTL \log(N * M))$ where N is the length of the longest list, M is the number of lists, L is the number of levels, and T is the size of a lifetime (the number of intervals in a lifetime). The loop iterates $O(NM)$ times, and each iteration costs $O(LT\log(N * M))$ to calculate the next candidate.

5.5 PBN to Support a Temporal Partial Order

To understand the scheme to support a temporal partial order, observe how a temporal JSON document potentially changes over time. Initially, a document is a hierarchy (a tree). Each insertion of an item adds to the hierarchy. Deletion is accomplished by ending the lifetime of an item in the document. It is only when an item is moved that it may potentially have more than one parent and, in effect, the document becomes a temporal partial order. Consider the example given in Figure 8. The `exercise` item was moved from `exercises` to `additional` at time 7, along with the descendant items, *e.g.,* the `number` item.

When an item is moved its lifetime continues in the new location. When the `exercise` item in Figure 8 is moved at time 7, its lifetime extends from time 5 (when it was created) to time *uc* (until changed). Also, when an item is moved it has more than location in the tree, *i.e.,* more than one PBN number. As an example, consider the move of the `exercise` item in Figure 8. The PBN numbers and

lifetimes before and after the move are depicted in Figure 13. In the figure the number $\boldsymbol{A}$ represents the PBN number of the root. The `exercise` item moves from location $\boldsymbol{A}$.1.2.1 to $\boldsymbol{A}$.1.3.1, so it has two different PBN numbers. The lifetime of each PBN number is shown in each node.

The key to supporting a temporal partial order is recording the lifetime and location of each moving item, and reconstructing the lifetime during keyword search evaluation. When an item is moved in the hierarchy several things happen. First, each moved item needs an item identifier to identify the item as it moves in the hierarchy. This identifier is separate from a PBN number. If the identifier is 0, then the item has not yet moved and is given a new identifier when it is moved. If it is non-zero then it already has an identifier and that identifier is used. In Figure 13 the item identifier is below the PBN number and to the left of the PBN number's lifetime. For example the `number` node has an item identifier of 2. Second, the tree of PBN numbers is updated. Each PBN number in the subtree rooted at the item is deleted in its current location by terminating the lifetime of each (still living) descendant PBN number. And the item is inserted in the new location by moving each (still living) item in the subtree rooted at the moved item, creating a new PBN number that starts its lifetime at the time of the move. In Figure 13 the `number` item is moved at time 7 from location $\boldsymbol{A}$.1.2.1.1 to $\boldsymbol{A}$.1.3.1.1. Its lifetime at location $\boldsymbol{A}$.1.2.1.1 is terminated at time 6 (logically deleting it from the hierarchy) and its lifetime at location $\boldsymbol{A}$.1.3.1.1 starts at time 7. Each descendant of `exercise` is similarly moved, with the exception of the previously deleted "`C++`" item. Third, the moved item's new location and lifetime in that location is added to a list for that item. Figure 14 shows the lists for the moved items in Figure 13.

The list of locations for an item is used to construct the lifetime of the item in a keyword search. In Algorithm 2 line 8 shows that the search identifies pairs $(v_1, t_1), \ldots, (v_n, t_n)$ to use in computing a candidate SLCA, each v_i is a PBN number corresponding to some keyword and t_i is its lifetime. Those are fed into the `temporalLCA` computation in step 9 that computes the lifetime of the LCA of $v_1, \ldots, v_n$. In a temporal partial order, that lifetime is computed from the item lifetimes as follows. Let v_i have item identifier i that corresponds to a list of pairs of PBN numbers and their lifetimes, where each pair for item i is (p_i, u_i) and x have item identifier x, also with a list of pairs. Then

$$t_x = \bigcup((u_1 \cap \ldots \cap u_n \cap u_x) \text{ where } p_x = \mathtt{LCA}(\mathtt{p_1},\ \ldots,\ \mathtt{p_n}).$$

The equation computes the lifetime, t_x, of an SLCA, x, as the union over the lifetime of each fragment of the item that has the item x as the LCA. As an example, consider a search for `chapter`, yielding node $\boldsymbol{A}$.1 and `number`, yielding nodes $\boldsymbol{A}$.1.2.1.1 and $\boldsymbol{A}$.1.3.1.1 (`number` has two different parents over time). The LCA of $\boldsymbol{A}$.1 and $\boldsymbol{A}$.1.2.1.1 is $\boldsymbol{A}$.1 with lifetime [5-6] and the LCA of $\boldsymbol{A}$.1 and $\boldsymbol{A}$.1.3.1.1 is also $\boldsymbol{A}$.1 but with lifetime [7-*uc*]. So the lifetime of the common ancestor, $\boldsymbol{A}$.1, is the union of the lifetimes of the two fragments: [7-*uc*]. In contrast if we search for `exercises` and `number` then the common ancestor $\boldsymbol{A}$.1.2 only has a computed lifetime of [5-6] since `exercises` was moved out of the subtree rooted at `exercises` at time 7.

Finally, we observe that if all of the keyword matches are to moved items, then the technique could produce duplicates. For example, a search for `exercise` and `number` will result in combining the lifetime of item 1 and item 2 for both $\boldsymbol{A}$.1.2.1 and $\boldsymbol{A}$.1.3.1 having the same lifetime of [7-*uc*]. These duplicates are eliminated by using a hash table to record which combinations of item identifiers have previously produced results, and skipping hash table hits.

5.6 Additional Indexes in Temporal SLCA Computation

In addition to the two indexes for SLCA computation, temporal partial order computation adds an additional index that maps an item identifier to a list of nodes for that identifier.

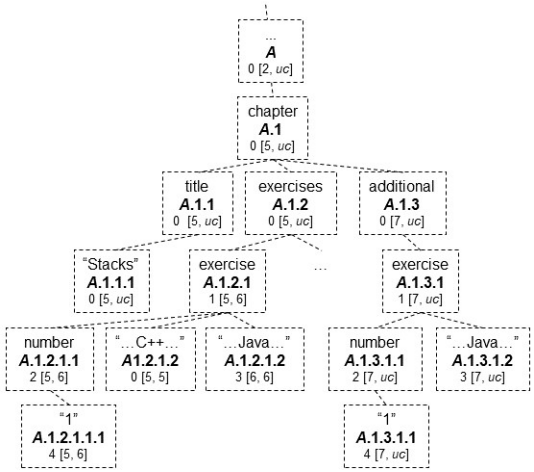

Fig. 13. The PBN tree for the temporal instance of Figure 8

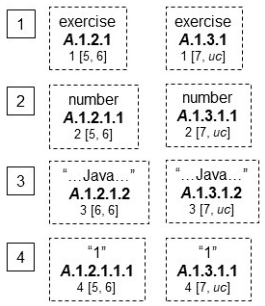

Fig. 14. PBNs and lifetimes for the moved items in Figure 13

6 EXPERIMENTAL STUDY

We performed several experiments to evaluate the performance of TJKS. The code for the system is available at https://github.com/cdyreson/temporalHierarchicalKeywordSearch. The first goal of the experiments is to evaluate the *additional cost of the temporal functionality*. We note that the paper does not claim a speed (or other metric) improvement over the state-of-the-art. Rather, the paper describes new functionality not previously supported, such as sequenced semantics. To measure the overhead we compare the cost of non-temporal with temporal search. A second goal is to determine whether *sequenced is more expensive than nonsequenced search*. Normal search measures, such as precision and recall, are irrelevant to the evaluation since we are not proposing a new non-temporal search technique, *e.g.*, an alternative to XSeek. The techniques in this paper work with any existing keyword search engine that uses some search technique based on a common ancestor.

6.1 Experiments Environment

Since the source code of the MESSIAH keyword search engine [60] is publicly-available, we chose to extend it to support our proposed algorithm. MESSIAH also offers good search performance (in terms of precision and recall) but this paper is orthogonal to and agnostic about the specific

keyword search technique used, as long as it is based on some notion of a common ancestor. MESSIAH is open source, written in Java, and has a keyword index, node index, and a type index. It uses a variant of SLCA, called FLSCA, that keeps track of “missing" nodes, in effect, it is better able to cope with irregular-structured, and missing data in a JSON document [60].

The experiments were run on an Oracle Cloud instance with 4 vCPUs, 32GB of RAM, and a 1TB, SSD disk. We implemented and testing using the Java 21, with a Jackson JSON parser, on a Linux system running Ubuntu 20.0.4. We used the default settings for Java (did not increase heap memory or change the behavior of the garbage collector). The data was parsed and stored persistently using BerkeleyDB Java edition version 7.5.11. A BerkeleyDB database is a collection of BerekelyDB tables, where each table is logically a key-value store, and physically a persistent B^+ tree. We used the default settings for BerkeleyDB, for instance, we did not modify the size of the DB cache. To mitigate JVM garbage collection effects we restarted the search engine for each query, so every query starts with an empty Java heap. We did not clear the operating system cache between runs.

6.2 Datasets

The experiments were performed with the following three datasets from the JEDI project; JEDI is a JSON similarity search engine [36].

(1) Trees - The tree dataset is a JSON representation of a phylogenetic tree from Influenza H3N2/A HA CDS between 2008 and 2015 [17]. It is 11.9MB but has a deep, complex structure with a tree depth of 379.
(2) DBLP - The DBLP dataset contains two million documents scraped from DBLP in XML and converted into JSON [21]. It is 792MB and has a relatively flat and regular structure, with a depth of 4.
(3) NBA - The NBA dataset was collected by Sports Reference LLC [21]. It contains around 32.000 nested documents representing NBA games in the period 1985-2013. Each document represents a game between two teams with at least 11 players each. It is 207MB and has a maximum depth of 8.

The datasets do not have temporal data so we manufactured the temporal data for the experiments by adding timestamps during parsing of the dataset.

6.3 Experiment 1 - Impact of Dataset Size

The first experiment is designed to measure the impact of dataset size for a constant query result size. We generated five slices of each dataset ranging from 20% to 100% of the dataset size. We seeded each slice with 100 equally spaced key-value pairs to ensure a constant result size. The seeded datasets were indexed by the search engine. For the temporal datasets we repeated the process, with the additional step of adding a timestamp to each node during the indexing (resulting database sizes are approximately the same). The timetsamp is the same for every node (ensuring that a sequenced search will produce the same result as a nontemporal search, but the sequenced search will process the timestamps and compute a sequenced SLCA). The experiment measures the time taken for a nontemporal search, and the same search (same keywords) is run as a sequenced search. The search is designed to return 100 search results (SLCA nodes).

The results are shown in Figures 15, 17, and 16. For each kind of search the cost is split into the time to compute the SLCAs and the time to extract the fragment for the search result. The different datasets have different sizes and structures, but because of how keyword search works the cost is constant with respect to the dataset size because the number and size of the search results is

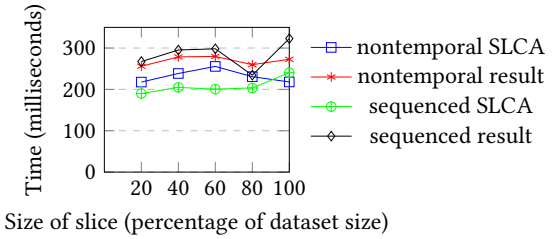

Fig. 15. Result constant Trees data

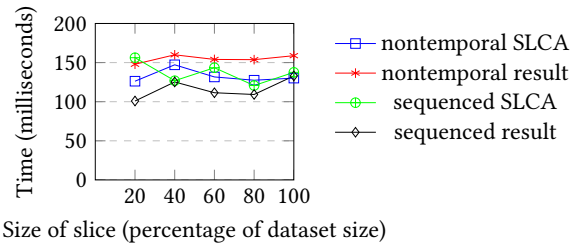

Fig. 16. Result constant DBLP data

constant across the datasets. Note that in both experiments the overhead of sequenced search is negligible compared to that of nontemporal search.

6.4 Experiment 2 - Impact of Result Size

The second experiment keeps the dataset size constant, but increases the size of the search result. We increase the size of the query result by randomly seeding the document with more matches, leading to a larger number of nodes returned by the same search query. We again compare nontemporal and sequenced searches. Figures 18, 19, and 20 shows the results. The cost increases linearly with respect to the search result size. Note also that the overhead of sequenced search is negligible compared to that of nontemporal search.

6.5 Experiment 3 - Impact of Time Distribution

In the previous experiments the timestamps were all the same, but in a real-world document as nodes are added and deleted, node lifetimes vary in duration. In the third experiment the duration of the lifetime of a node is randomly chosen to be at most a given portion of the timeline, and the portion is varied from 1% to 100%. We tested only the DBLP dataset as it is the largest. Note that the lifetime of each node is also constrained to be within than its parent's. So as the duration decreases fewer results are returned since nodes that match the keywords do not overlap in time. The result is shown in Figure 21.

6.6 Experiment 4 - Sequenced vs. Nonsequenced

The fourth experiment measures the cost of sequenced vs. nonsequenced SLCA computation. It compares a sequenced search with a nonsequenced search that computes the overlap of the timestamps in the search result (so the searches are the same except for the sequenced SLCA). We used the same setup as Experiment One for creating the DBLP dataset and tested on only DBLP. The result is shown in Figure 22. The costs are roughly equivalent.

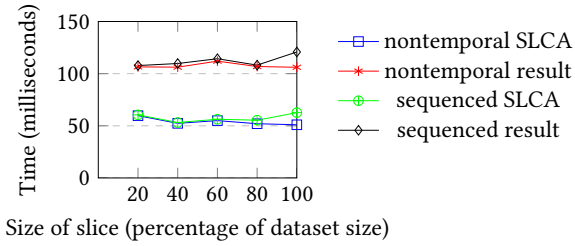

Fig. 17. Result constant NBA data

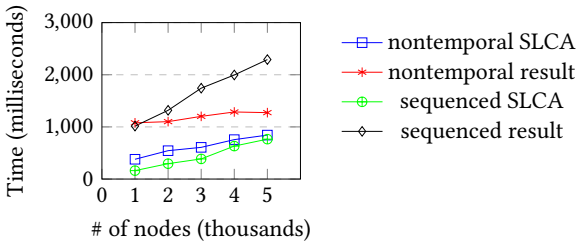

Fig. 18. Size of the result increases for the Trees dataset

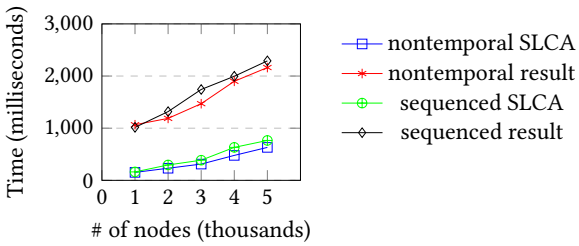

Fig. 19. Size of the result increases for the DBLP dataset

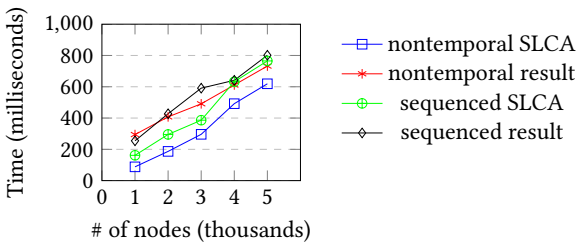

Fig. 20. Size of the result increases for the NBA dataset

6.7 Analysis

Overall the experiments show that overhead of temporal search is modest. Experiment 1 showed that dataset size has only a small impact on the cost. Keyword search heavily indexes the dataset so that the keywords can be quickly found. Experiment 2 showed that result size impacts the cost,

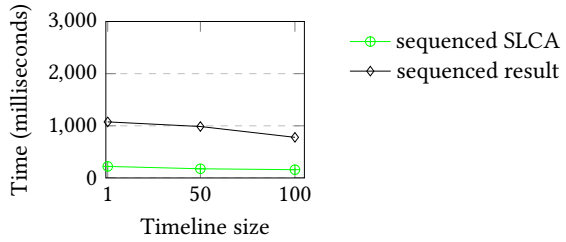

Fig. 21. Duration varies for the DBLP dataset

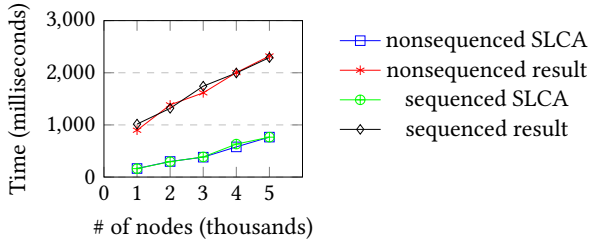

Fig. 22. Sequenced vs. nonsequenced for the DBLP dataset

more results yields a higher cost, but the cost increase is largely linear in the size of the result. Experiment 3 showed that the size of the time-line has little impact on cost. Finally Experiment 4 demonstrated that sequenced and nonsequenced have similar cost.

7 CONCLUSIONS AND FUTURE WORK

This paper presents Tjks, a system for temporal JSON keyword search. Keyword search can only search a static JSON document, or the current version of a changing document. But JSON documents are edited and change over time. Temporal keyword search expands keyword search to cover the history of and changes to a document. We gave a taxonomy that classified the different kinds of search and identified two, uniquely temporal, kinds of searches: sequenced and nonsequenced. Sequenced search, in effect, runs a keyword search simultaneously on slices of the document, while nonsequenced search can compare different slices to determine how a document has changed over time. We showed how sequenced and nonsequenced search can be implemented and experimentally explored the additional cost of supporting temporal search. We also showed how versions can be searched and reconstructed in search results.

In future, we plan to research *temporal similarity search*. Similarity search takes a fragment of a hierarchy and finds similar fragments. For instance a user could search using the edit history of a fragment to find similarly edited fragments over time. Another direction for future research is *other kinds of metadata*. (probabilities) could also be present. The same techniques presented in this paper could be used to support these other kinds of metadata, such as probability, data quality, or provenance, and meta-metadata such as versioned security (time annotating security metadata). For instance, keyword search on a document annotated with privacy metadata will have to observe private vs. public views of the data. We also plan to explore how a temporal index could be used to improve temporal slicing [50]. A temporal slice shrinks the data to only data alive during the slice interval. A temporal index could be employed to quickly isolate that data.

REFERENCES

[1] [n. d.]. DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed: 2023-10-10.

[2] [n. d.]. JSONPath. https://github.com/json-path/JsonPath. Accessed: 2023-10-10.

[3] Manoj K Agarwal and Krithi Ramamritham. 2015. Enabling generic keyword search over raw XML data. In *2015 IEEE 31st International Conference on Data Engineering*. 1496–1499. https://doi.org/10.1109/ICDE.2015.7113410

[4] James F. Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843. https://doi.org/10.1145/182.358434

[5] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. 2000. A Data Model for Temporal XML Documents. In *DEXA*. 334–344. https://doi.org/10.1007/3-540-44469-6_31

[6] Avishek Anand, Srikanta J. Bedathur, Klaus Berberich, and Ralf Schenkel. 2010. Efficient temporal keyword search over versioned text. In *CIKM*. ACM, 699–708. https://doi.org/10.1145/1871437.1871528

[7] Nikolaus Augsten, Michael H. Böhlen, Curtis E. Dyreson, and Johann Gamper. 2012. Windowed PQ-Grams for Approximate Joins of Data-centric XML. *VLDB J.* 21, 4 (2012), 463–488. https://doi.org/10.1007/s00778-011-0254-6

[8] Zhifeng Bao, Jiaheng Lu, Tok Wang Ling, and Bo Chen. 2010. Towards an Effective XML Keyword Search. *IEEE TKDE* 22, 8 (2010), 1077–1092.

[9] Rasha Bin-Thalab, Neamat El-Tazi, and Mohamed E. El-Sharkawi. [n. d.]. TMIX: Temporal Model for Indexing XML Documents. IEEE, 1–8. https://doi.org/10.1109/AICCSA.2013.6616483

[10] Tobias Bleifuß, Leon Bornemann, Theodore Johnson, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2018. Exploring Change: A New Dimension of Data Analytics. *Proc. VLDB Endow.* 12, 2 (2018), 85–98. https://doi.org/10.14778/3282495.3282496

[11] Michael H Bohlen, Renato Busatto, and Christian S Jensen. 1998. Point-versus Interval-based Temporal Data Models. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 192–200.

[12] Michael H. Böhlen and Christian S. Jensen. 2009. Sequenced Semantics. In *Encyclopedia of Database Systems*. 2619–2621. https://doi.org/10.1007/978-0-387-39940-9_1053

[13] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. 2000. Temporal Statement Modifiers. *ACM Trans. Database Syst.* 25, 4 (2000), 407–456. http://portal.acm.org/citation.cfm?id=377674.377665

[14] Michael H Böhlen, Christian S Jensen, and Richard Thomas Snodgrass. 2000. Temporal Statement Modifiers. *ACM Transactions on Database Systems (TODS)* 25, 4 (2000), 407–456.

[15] Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass. 2009. Nonsequenced Semantics. In *Encyclopedia of Database Systems*. 1913–1915. https://doi.org/10.1007/978-0-387-39940-9_1052

[16] Zouhaier Brahmia, Fabio Grandi, Safa Brahmia, and Rafik Bouaziz. 2021. A Graphical Conceptual Model for Conventional and Time-varying JSON Data. *Procedia Computer Science* 184 (2021), 823–828. https://doi.org/10.1016/j.procs.2021.03.102 The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops.

[17] Jan P Buchmann, Mathieu Fourment, and Edward C Holmes. 2018. The Biological Object Notation (BON): a structured file format for biological data. *Scientific reports* 8, 1 (2018), 1–8.

[18] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. 1998. Representing and Querying Changes in Semistructured Data. In *ICDE*. 4–13. https://doi.org/10.1109/ICDE.1998.655752

[19] Cindy Xinmin Chen and Carlo Zaniolo. 2000. SQLST: A Spatio-Temporal Data Model and Query Language. In *ER*. 96–111. https://doi.org/10.1007/3-540-45393-8_8

[20] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. 2002. Efficient Schemes for Managing Multiversion XML Documents. *VLDB J.* 11, 4 (2002), 332–353. https://doi.org/10.1007/s00778-002-0079-4

[21] Mohamed L Chouder, Stefano Rizzi, and Rachid Chalal. 2019. EXODuS: exploratory OLAP over document stores. *Information Systems* 79 (2019), 44–57.

[22] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. 1997. On the Semantics of "Now" in Databases. *ACM Transactions on Database Systems* 22, 2 (1997), 171–214.

[23] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. 2003. XSEarch: A Semantic Search Engine for XML. In *VLDB*. 45–56.

[24] Faiz Currim, Sabah Currim, Curtis Dyreson, and Richard T. Snodgrass. 2004. A Tale of Two Schemas: Creating a Temporal XML Schema from a Snapshot Schema with τXSchema. In *EDBT*. Vol. 2992. Springer Berlin Heidelberg, 348–365. http://link.springer.com/10.1007/978-3-540-24741-8_21

[25] Faiz Currim, Sabah Currim, Curtis E. Dyreson, Richard T. Snodgrass, Stephen W. Thomas, and Rui Zhang. 2012. Adding Temporal Constraints to XML Schema. *IEEE Trans. Knowl. Data Eng.* 24, 8 (2012), 1361–1377. https://doi.org/10.1109/TKDE.2011.74

[26] Gao Dandan, Wang Xinjun, and Deng Li. [n. d.]. Indexing Temporal XML Using Interval-Tree Index. IEEE, 689–691. https://doi.org/10.1109/CSSE.2008.1223

[27] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2012. Temporal Alignment. In *SIGMOD*. 433–444. https://doi.org/10.1145/2213836.2213886

[28] Curtis E. Dyreson and Fabio Grandi. 2009. Temporal XML. In *Encyclopedia of Database Systems*. 3032–3035.

[29] Curtis E. Dyreson, Venkata A. Rani, and Amani Shatnawi. 2015. Unifying Sequenced and Non-sequenced Semantics. In *TIME*. IEEE Computer Society, 38–46. https://doi.org/10.1109/TIME.2015.22

[30] Vahid Ghadakchi, Abtin Khodadadi, and Arash Termehchy. 2019. Less Data Delivers Higher Search Effectiveness for Keyword Queries. In *SSDBM*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/3335783.3335794

[31] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. 2008. Keyword proximity search in complex data graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. New York, NY, USA, 927–940. https://doi.org/10.1145/1376616.1376708

[32] Gang Gou and Rada Chirkova. 2007. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Trans. Knowl. Data Eng.* 19, 10 (2007), 1381–1403.

[33] Aayush Goyal and Curtis Dyreson. 2019. Temporal JSON. In *2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC)*. 135–144. https://doi.org/10.1109/CIC48465.2019.00025

[34] Fabio Grandi. 2010. T-SPARQL: A TSQL2-like Temporal Query Language for RDF. In *ADIBIS*. 21–30. http://ceur-ws.org/Vol-639/021-grandi.pdf

[35] Fabio Grandi, Federica Mandreoli, and Paolo Tiberio. [n. d.]. Temporal Modelling and Management of Normative Documents in XML Format. 54, 3 ([n. d.]), 327–354. https://doi.org/10.1016/j.datak.2004.11.002

[36] Thomas Hütter, Nikolaus Augsten, Christoph M. Kirsch, Michael J. Carey, and Chen Li. 2022. JEDI: These Aren't the JSON Documents You're Looking For.... In *SIGMOD*. Association for Computing Machinery, New York, NY, USA, 1584–1597. https://doi.org/10.1145/3514221.3517850

[37] C. S. Jensen and C. E. Dyreson (editors). 1998. A Consensus Glossary of Temporal Database Concepts - February 1998 Version. In *Temporal Databases: Research and Practice, LNCS 1399*. Springer-Verlag, 367–405.

[38] Nattiya Kanhabua and Avishek Anand. 2016. Temporal Information Retrieval. In *SIGIR*. ACM, 1235–1238. https://doi.org/10.1145/2911451.2914805

[39] Lingbo Kong, Rémi Gilleron, and Aurélien Lemay. 2009. Retrieving Meaningful Relaxed Tightest Fragments for XML Keyword Search. In *EDBT*.

[40] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Record* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[41] Changqing Li and Tok Wang Ling. 2005. An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In *DASFAA*. 125–137.

[42] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. 2008. EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-Structured and Structured Data. In *SIGMOD*. Association for Computing Machinery, New York, NY, USA, 903–914. https://doi.org/10.1145/1376616.1376706

[43] Jianxin Li, Chengfei Liu, Rui Zhou, and Wei Wang. 2010. Suggestion of Promising Result Types for XML Keyword Search. In *EDBT*. 561–572.

[44] Yunyao Li, Cong Yu, and H. V. Jagadish. 2004. Schema-Free XQuery. In *VLDB*. 72–83.

[45] Chunbin Lin, Jianguo Wang, and Chuitian Rong. 2017. Towards Heterogeneous Keyword Search. In *Proceedings of the ACM Turing 50th Celebration Conference - China (ACM TURC '17)*. Association for Computing Machinery, New York, NY, USA, Article 46, 6 pages. https://doi.org/10.1145/3063955.3064802

[46] Ziyang Liu and Yi Chen. 2007. Identifying Meaningful Return Information for XML Keyword Search. In *SIGMOD*. 329–340.

[47] Ziyang Liu and Yi Chen. 2008. Reasoning and Identifying Relevant Matches for XML Keyword Search. *PVLDB* 1, 1 (2008), 921–932.

[48] Ziyang Liu, Chong Wang, and Yi Chen. 2017. Keyword Search on Temporal Graphs. *IEEE Trans. Knowl. Data Eng.* 29, 8 (2017), 1667–1680. https://doi.org/10.1109/TKDE.2017.2690637

[49] Vijini Mallawaarachchi, Lakmal Meegahapola, Roshan Madhushanka, Eranga Heshan, Dulani Meedeniya, and Sampath Jayarathna. 2020. Change Detection and Notification of Web Pages: A Survey. *ACM Comput. Surv.* 53, 1, Article 15 (feb 2020), 35 pages. https://doi.org/10.1145/3369876

[50] Federica Mandreoli, Riccardo Martoglia, and Enrico Ronchetti. 2006. Supporting Temporal Slicing in XML Databases. In *EDBT 2006 (Lecture Notes in Computer Science)*, Vol. 3896. Springer, 295–312. https://doi.org/10.1007/11687238_20

[51] Edimar Manica, Carina F. Dorneles, and Renata Galante. [n. d.]. Supporting Temporal Queries on XML Keyword Search Engines. 1, 3 ([n. d.]), 471. https://seer.lcc.ufmg.br/index.php/jidm/article/view/53@misc

[52] Alberto O. Mendelzon, Flavio Rizzolo, and Alejandro Vaisman. [n. d.]. Indexing temporal XML documents. In *VLDB* (2004). VLDB Endowment, 216–227. http://dl.acm.org/citation.cfm?id=1316710

[53] Mehdi Naseriparsa, Md. Saiful Islam, Chengfei Liu, and Irene Moser. 2018. No-but-semantic-match: computing semantically matched xml keyword search results. *World Wide Web* 21, 5 (2018), 1223–1257. https://doi.org/10.1007/s11280-017-0503-8

[54] Flavio Rizzolo and Alejandro A. Vaisman. 2008. Temporal XML: Modeling, Indexing, and Query Processing. *VLDB J.* 17, 5 (2008), 1179–1212. https://doi.org/10.1007/s00778-007-0058-x

[55] Simonas Saltenis and Christian S. Jensen. 2002. Indexing of Now-Relative Spatio-Bitemporal Data. *VLDB J.* 11, 1 (2002), 1–16. https://doi.org/10.1007/s007780100058

[56] Virginie Sans and Dominique Laurent. 2008. Prefix based numbering schemes for XML: techniques, applications and performances. *PVLDB* 1, 2 (2008), 1564–1573.

[57] Richard T. Snodgrass. 1987. The Temporal Query Language TQuel. *ACM Trans. Database Syst.* 12, 2 (1987), 247–298. https://doi.org/10.1145/22952.22956

[58] Richard T. Snodgrass (Ed.). 1995. *The TSQL2 Temporal Query Language*. Kluwer.

[59] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. 1997. Transitioning Temporal Support in TSQL2 to SQL3. In *Temporal Databases, Dagstuhl*. 150–194. https://doi.org/10.1007/BFb0053702

[60] Ba Quan Truong, Sourav S. Bhowmick, Curtis E. Dyreson, and Aixin Sun. 2013. MESSIAH: missing element-conscious SLCA nodes search in XML data. In *SIGMOD*. ACM, 37–48. https://doi.org/10.1145/2463676.2463699

[61] Alejandro A. Vaisman and Alberto O. Mendelzon. 2001. A Temporal Query Language for OLAP: Implementation and a Case Study. In *DBPL*. 78–96. https://doi.org/10.1007/3-540-46093-4_5

[62] Santiago Vargas, Utkarsh Goel, Moritz Steiner, and Aruna Balasubramanian. 2019. Characterizing JSON Traffic Patterns on a CDN. In *Proceedings of the Internet Measurement Conference (IMC '19)*. Association for Computing Machinery, New York, NY, USA, 195–201. https://doi.org/10.1145/3355369.3355594

[63] Fusheng Wang and Carlo Zaniolo. 2004. XBiT: an XML-based Bitemporal Data Model. In *Conceptual Modeling at ER 2004*. Springer, 810–824. http://link.springer.com/chapter/10.1007/978-3-540-30464-7_60

[64] Fusheng Wang and Carlo Zaniolo. 2005. An XML-Based Approach to Publishing and Querying the History of Databases. *World Wide Web* 8, 3 (2005), 233–259. https://doi.org/10.1007/s11280-005-1317-7

[65] Jingjing Wang and Shengli Wu. 2017. Information Retrieval with Implicitly Temporal Queries. In *IDEAL (Lecture Notes in Computer Science)*, Vol. 10585. Springer, 103–111. https://doi.org/10.1007/978-3-319-68935-7_12

[66] Yu Xu and Yannis Papakonstantinou. 2005. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*. 537–538.

[67] Li Yan, Ruizhe Ma, and Zhangbing Hu. 2022. A Temporal JSON Data Model and Its Query Languages. *J. Database Manage.* 33, 1 (may 2022), 1–29. https://doi.org/10.4018/JDM.299556

[68] Jeffrey Xu Yu, Daofeng Luo, Xiaofeng Meng, and Hongjun Lu. 2005. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web* 8, 1 (2005), 5–26.

[69] Gongsheng Yuan, Jiaheng Lu, and Peifeng Su. 2021. Quantum-Inspired Keyword Search on Multi-model Databases. In *Database Systems for Advanced Applications*. Springer International Publishing, Cham, 585–602.

[70] Peisen Yuan, Chaofeng Sha, Xiaoling Wang, Bin Yang, Aoying Zhou, and Su Yang. 2010. XML Structural Similarity Search Using MapReduce. In *WAIM*. 169–181. https://doi.org/10.1007/978-3-642-14246-8_19

[71] Feng Zhang, Xinjun Wang, and Shaolong Ma. 2009. Temporal XML Indexing Based on Suffix Tree. IEEE, 140–144. https://doi.org/10.1109/SERA.2009.20

[72] Junfeng Zhou, Wei Wang, Ziyang Chen, Jeffrey Xu Yu, Xian Tang, Yifei Lu, and Yukun Li. 2016. Top-Down XML Keyword Query Processing. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1340–1353. https://doi.org/10.1109/TKDE.2016.2516536

Received October 2023; revised January 2024; accepted February 2024