Sequential Linked Data: the State of Affairs

Enrico Daga\textsuperscript{a,*}, Albert Meroño-Peñuela\textsuperscript{b}, and Enrico Motta\textsuperscript{a}

\textsuperscript{a} Knowledge Media Institute, The Open University, United Kingdom
E-mails: enrico.daga@open.ac.uk, enrico.motta@open.ac.uk
\textsuperscript{b} Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
E-mail: albert.merono@vu.nl

Abstract. Sequences are among the most important data structures in computer science. In the Semantic Web, however, little attention has been given to Sequential Linked Data. In previous work, we have discussed the data models that Knowledge Graphs commonly use for representing sequences and showed how these models have an impact on query performance and that this impact is invariant to triplestore implementations. However, the specific list operations that the management of Sequential Linked Data requires beyond the simple retrieval of an entire list or a range of its elements – e.g. to add or remove elements from a list, and their impact in the various list data models, remain unclear. Covering this knowledge gap would be a significant step towards the realization of a Semantic Web list Application Programming Interface (API) that standardizes list manipulation and generalizes beyond specific data models. In order to address these challenges towards the realization of such an API, we build on our previous work in understanding the effects of various sequential data models for Knowledge Graphs, extending our benchmark and proposing a set of read-write Semantic Web list operations in SPARQL, with insert, update and delete support. To do so, we identify five classic list-based computer science sequential data structures (linked list, double linked list, stack, queue, and array), from which we derive nine atomic read-write operations for Semantic Web lists. We propose a SPARQL implementation of these operations with five typical RDF data models and compare their performance by executing them against six increasing dataset sizes and four different triplestores. In light of our results, we discuss the feasibility of our devised API and reflect on the state of affairs of Sequential Linked Data.

Keywords: Sequential Linked Data, Benchmark, RDF, SPARQL

1. Introduction

Sequences are representations of real-world sets of entities that require an order and possibly a reference to their position. They support a large variety of domain knowledge, such as scholarly metadata (paper authors — e.g., the last author), historical data (biographies and timelines), media metadata (tracklists — e.g., the fourth track), social media content (recipes, howto) and musical content (e.g., scores as MIDI Linked Data [25]). Applications typically need to perform a variety of operations on lists, including multiple types of access and edits, typically in the form of queries (in, e.g., SPARQL). The practical complexity of these queries can have a potentially tremendous impact on performance and service availability [9].

The Semantic Web community has engineered various list models across the years, for example, the Ordered List pattern [16], which refers to the construct \texttt{rdf:List} in the RDF W3C specification. A pragmatic solution refers to each member of the list using RDF containment membership properties (\texttt{rdf:_1}, \texttt{rdf:_2},... ) within an n-ary relation of type \texttt{rdf:Seq}. Alternative options may involve picking a solution from the Ontology Design Patterns catalogue [12], for example, the Sequence ODP\textsuperscript{1}. However, either of these choices could have a significant impact in terms of query-ability (fitness for use in applications), performance and, ultimately, availability of the data. In our previous work [13, 26], we have shown that most of these practical list models can be re-

\textsuperscript{1}Sequence: http://ontologydesignpatterns.org/wiki/Submissions:Sequence.

*Corresponding author. E-mail: enrico.daga@open.ac.uk.
duced to five methods: rdf:Seq, rdf:List, URI-based, number-based, and the sequence ontology pattern. We also started a benchmark initiative to evaluate their querying performance in various dataset sizes and triplestore configurations. This work is, however, limited to evaluate read operations only. Other SPARQL-based benchmarks evaluate competing storage solutions against generic use cases, deemed to be representative of critical features of the query language [10] or to mirror how users query Linked Data [30].

Therefore, important questions that remain unanswered are:

– Is the usability of Sequential Linked Data modelling solutions affected by the introduction of write operations, like insert, update, and delete?

– Which of these methods should be recommended for Sequential Linked Data, towards the realization of a generic, SPARQL-based Application Programming Interface (API)?

In this article, we propose to extend our empirical evaluation approach of Semantic Web lists with a full set of well-grounded, read-write atomic operations that could constitute the core of a Semantic Web List API. We seek inspiration for these operations in five classic computer science sequential data structures. These data structures give rise to a set of nine atomic read-write operations. This set of operations leads to the core research question: how lists should be modelled in order to support these operations efficiently? How do these operations perform in current RDF list models? To answer this question, we implement these operations as SPARQL queries, and we use these queries to develop our experiments on query performance. For this, we re-use our surveyed methods for modelling sequences in RDF [13], and we extend our proposed pragmatic benchmark [26] for assessing their performance in conjunction with these atomic SPARQL queries in a number of triplestores and dataset sizes. Specifically, we demonstrated in [13] how the efficiency of retrieving sequential linked data depends primarily on how they are modelled and that the impact of a modelling solution on data availability is independent from the database engine (triple-store invariance hypothesis). Here, we complement this analysis by focusing on data management and perform a thorough assessment of List read-write operations for the Semantic Web.

Our contributions are:

– A list of nine atomic, read-write list operations in SPARQL towards the realization of a Semantic Web List API, consisting of: first, rest, append, append_front, prev, popoff, set, get, and remove_at. We base these operations on those that build the basis of the classic computer science sequential data structures of linked lists, double linked lists, stack, queue, and array (Section 3)

– An extension to our RDF list benchmark, supporting those operations (Section 5)

– Experiments to evaluate the performance of these read-write operations in SPARQL with competing list data models, on datasets of increasing sizes, and against four different triplestores (Section 6).

The rest of the paper is structured as follows. We introduce the research methodology in Section 2. We report our findings in atomic read-write list operations based on sequential data structures in Section 3. Section 4 provides background on the modelling solutions we aim at evaluating. In Section 5 we use the atomic read-write list operations to formalise SPARQL update queries and extend our benchmark. Section 6 reports on the experiments. Results are discussed in Section 7. We survey the related work in Section 8 and conclude our paper in Section 9.

In this article we use the following namespace prefixes:

midi: <http://purl.org/midi-ld/midi#>  
midi-note: <http://purl.org/midi-ld/notes/>  
midi-prog: <http://purl.org/midi-ld/programs/>  
prov: <http://www.w3.org/ns/prov#>  
rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
rdfs: <http://www.w3.org/2000/01/rdf-schema#>  
xsd: <http://www.w3.org/2001/XMLSchema#>  
song: <http://purl.org/midi-ld/song/example/>  
ex: <http://www.example.org/>

2. Methodology

In this section, we specialise the methodology previously introduced in [13] to pragmatically evaluate the performance of competing models for the representation of Sequential Linked Data in relation to a standard List API, grouping atomic read-write operations. Phases of the methodology are requirements, survey, formalisation, and evaluation.
3. Requirements from sequential data structures

In this Section, we identify abstract data structures that are typically considered to represent ordered sequences from classic computer science data structure texts [7, 28]. We focus on linear data structures that (a) preserve the order of the items, (b) are continuous sequences, (c) have unrestricted size, and (d) include a single element in each position. In what follows, we describe the data structures by listing the core functions they are meant to support and express their expected behaviour by axiomatic semantics.

2Typically, lists are distinguished from sets because they allow the same item to be repeated multiple times. However, the data structure itself does necessarily have distinct slots, each one of them pointing to some object (that can be referenced multiple times). In RDF, this is typically achieved using intermediate entities, for example, as blank nodes.

3In fact, blank nodes do not require the matching node value to be kept in memory as part of the query solution to be projected.
3.1. Linked List

The abstract list type \( L \) with elements of some type \( E \) is defined by the following functions and axioms:

\[
\begin{align*}
append\_front : (E \times L) &\rightarrow L \\
first : L &\rightarrow E \\
rest : L &\rightarrow L \\
prev : L &\rightarrow L \\
\end{align*}
\]

(1)

\[
\begin{align*}
first(append\_front(e,l)) &= e \\
rest(append\_front(e,l)) &= l \\
e &\in E, l &\in L
\end{align*}
\]

where \( append\_front \) is the operator that constructs memory objects which hold two values or pointers to values. It is implicit that \( append\_front(e,l) \neq l \), \( append\_front(e,l) \neq e \), \( append\_front(e_1,l_1) = append\_front(e_2,l_2) \) iff \( e_1 = e_2 \) and \( l_1 = l_2 \). Note that \( first([]) \) and \( rest([]) \) are not defined, for \([\ ] \) representing the empty list.

Examples. The use of Linked Lists is straightforward for users of most programming languages. E.g.,

\[
\begin{align*}
append\_front([\text{alice}, [\text{bob}, \text{carl}]]) &= [\text{alice}, \text{bob}, \text{carl}] \\
\end{align*}
\]

and then \( first([\text{alice}, \text{bob}, \text{carl}]) = \text{alice} \) and subsequently \( rest([\text{alice}, \text{bob}, \text{carl}]) = [\text{bob}, \text{carl}] \).

3.2. Double Linked List

A variant of a linked list is one in which each item has a link to the previous item as well as the next. This allows easily accessing list items backward as well as forward and deleting any item in constant time. Following the definition of linked lists, a double linked list is defined with the same functions and axioms but defining an additional \( prev \) pointer and \( append \) function:

\[
\begin{align*}
append\_front : (E \times L) &\rightarrow L \\
append : (L \times E) &\rightarrow L \\
first : L &\rightarrow E \\
rest : L &\rightarrow L \\
prev : L &\rightarrow L \\
\end{align*}
\]

(2)

\[
\begin{align*}
first(append\_front(e,l)) &= e \\
rest(append\_front(e,l)) &= l \\
prev(append(l,e)) &= l \\
e &\in E, l &\in L
\end{align*}
\]

Examples. Analogous to Linked Lists, with the addition of the \( append \) and \( prev \) functions that append an element to the end of the list and return the list preceding an element, respectively. For example, \( append([\text{bob}, \text{carl}], \text{alice}) = [\text{bob}, \text{carl}, \text{alice}] \) and then \( prev(append([\text{bob}, \text{carl}], \text{alice})) = [\text{bob}, \text{carl}] \).

3.3. Stack

A stack is a collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are \( append\_front \), already introduced, and \( popoff \). Often \( first \) (also known as \( top \)) is available, too. This data structure is also known as "last-in, first-out" queue (LIFO). These operations have the following axiomatic semantics:

\[
\begin{align*}
popoff : (S) &\rightarrow E \\
first(append\_front(e,s)) &= s \\
first(append\_front(e,s)) &= e \\
append\_front(appendoff(s),s) &= s \\
first(appendoff(s),s) &\neq s \\
s &\in S, e &\in E
\end{align*}
\]

(3)
where $S$ is a stack and $e$ is a value. Here, $S$ is to be considered a sequential data structure analogous to $L$ in the previous section.

**Examples.** In a stack, elements are piled on top of each other, and can be accessed only by popping them off the top first. For example, \( popoff([\text{alice}, \text{bob}, \text{carl}]) = [\text{bob}, \text{carl}] \). As usual, \( first([\text{alice}, \text{bob}, \text{carl}]) = \text{alice} \) but does not imply any change in the sequence.

### 3.4. Queue

A queue is a collection of ordered items that supports addition of items (to the tail) and access (or deletion) to the earliest added item only [2]. Typically, the retrieval of the earliest item (head of the queue), corresponds to its deletion. In what follows we specify three operations: (1) \( append \) - adds an element at the end of the queue; (2) \( first \) - retrieves the element at the top of the queue; and (3) \( popoff \) - deletes the element from the top of the queue. The following axioms summarise the semantics of the operations:

\[
\begin{align*}
  first(append(v, [])) &= v \\
  popoff(append(v, [])) &= [] \\
  first(append(v, append(w, Q))) &= first(append(w, Q)) \\
  &\iff Q = [] \\
  popoff(append(v, append(w, Q))) &= append(v, popoff(append(w, Q))) \\
  &\iff Q = []
\end{align*}
\]

where $Q$ is a queue and $v$ and $w$ are values. Here, $Q$ is to be considered a sequential data structure analogous to $L$ and $S$ in the previous sections.

The queue data structure, also known as FIFO list (first in, first out) is generally conceived as having unlimited size, although there can be implementations that force a fixed number of items (bounded queue [1]). However, here we only consider queues of unlimited length.

**Examples.** Queues complement the behaviour of stacks, by allowing elements to be added on one side and accessed on the other. Following our previous examples, \( append([\text{alice}, [\text{bob}, \text{carl}]) = [\text{bob}, \text{carl}, \text{alice}] \), with \( first([\text{bob}, \text{carl}, \text{alice}]) = \text{bob} \) and its subsequent removal from the queue with \( popoff([\text{bob}, \text{carl}, \text{alice}]) = [\text{carl}, \text{alice}] \).

### 3.5. Array

An Array is a collection of objects that are randomly accessible by an index, often an integer value. Since our focus is on sequential data structures as defined in our methodology, we only consider sorted arrays, where the index is an integer. In addition, we restrict the definition to a data structure whose index also represents the position of the item in the list. For example, the item at index 2 being the second element in the sequence. Also, we do not consider arrays with constrained sizes or including null values (gaps in the sequence). An implication of this is that removing one item implies the shifting of others and a reduction in size of the array. The operations on Arrays are the following:

\[
\begin{align*}
  set &: (E \times I \times A) \to A \\
  get &: (I \times A) \to E \\
  remove@ &: (I \times A) \to A \\
\end{align*}
\]

\[
\begin{align*}
  get(i, set(e, i, a)) &= e \iff |a| \geq i \\
  remove@(i, a) &= b \\
  \to a_j &= b_j \iff j < i; \\
  a_{j-1} &= b_j \iff j > i \\
  e \in E; a, b \in A; i \in I
\end{align*}
\]

where $A$ are arrays, $E$ are elements, and $I$ are the possible indexes in $A$. Here, $A$ is to be considered a sequential data structure analogous to $L$, $Q$ and $S$ in the previous sections.

**Examples.** Arrays are typically used for in-memory scenarios, consequently performing faster but at the cost of memory allocation management and security. \( set \) and \( get \) store and retrieve, respectively, one element of the structure. \( remove@ \) is a convenience function to delete an element by shifting all its subsequent elements of one position. For example, \( set(\text{alice}, 1, [\text{daniel}, \text{bob}, \text{carl}]) = [\text{alice}, \text{bob}, \text{carl}] \).
where random elements can be accessed with \( \text{get}(2, \{\text{alice}, \text{bob}, \text{carl}\}) = \text{bob} \), and removed with \( \text{remove\_at}(2, \{\text{alice}, \text{bob}, \text{carl}\}) = \{\text{alice}, \text{carl}\} \).

Table 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>LL</th>
<th>DLL</th>
<th>ST</th>
<th>QU</th>
<th>ARR</th>
</tr>
</thead>
<tbody>
<tr>
<td>first: ( L \rightarrow E )</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>rest: ( L \rightarrow L )</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>append: ( L \times E \rightarrow L )</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>append_front: ( E \times L \rightarrow L )</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prev: ( L \rightarrow L )</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>popoff: ( L \rightarrow L )</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>set: ( E \times I \times L \rightarrow L )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>get: ( I \times L : E )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>remove_at: ( I \times L \rightarrow L )</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>

In this section we illustrated five computer science data structures and identified several key operations for sequential data management. Table 1 summarises our requirements, showing the list of operations and their relevance to the abstract data structures discussed.

4. RDF Modelling Approaches to Sequential Data

In this section we present a summary of Semantic Web list models and their properties, recalling the research in [13]. These models were surveyed by selecting them from the following sources, including W3C standards\(^4\) ontology design patterns [18], resource track papers in the International Semantic Web Conference (e.g. [5], [25]), and lookups of relevant terms in Linked Open Vocabularies [35]. In what follows, RDF data models are described as a collection of common practices in the Linked Data community, and are not to be understood as recommendations (see Section 7). For further details and a description of the surveying methodology, see [13].

4.1. RDF Sequences

The RDF Schema (RDFS) recommendation [8] defines the container classes \( \text{rdf:Bag} \), \( \text{rdf:Alt} \), \( \text{rdf:Seq} \) to represent collections. Since \( \text{rdf:Bag} \) is intended for unordered elements, and \( \text{rdf:Alt} \) for “alternative” containers, whose typical process-

\[
\text{Operation:}
\begin{align*}
\text{first: } & L \rightarrow E & \text{x} & \text{x} & \text{x} & \text{x} & \text{-} \\
\text{rest: } & L \rightarrow L & \text{x} & \text{x} & \text{-} & \text{-} & \text{-} \\
\text{append: } & L \times E \rightarrow L & \text{x} & \text{x} & \text{-} & \text{-} & \text{-} \\
\text{append\_front: } & E \times L \rightarrow L & \text{x} & \text{x} & \text{x} & \text{-} & \text{-} \\
\text{prev: } & L \rightarrow L & \text{-} & \text{x} & \text{-} & \text{-} & \text{-} \\
\text{popoff: } & L \rightarrow L & \text{-} & \text{-} & \text{x} & \text{x} & \text{-} \\
\text{set: } & E \times I \times L \rightarrow L & \text{-} & \text{-} & \text{-} & \text{-} & \text{x} \\
\text{get: } & I \times L : E & \text{-} & \text{-} & \text{-} & \text{-} & \text{x} \\
\text{remove\_at: } & I \times L \rightarrow L & \text{-} & \text{-} & \text{-} & \text{-} & \text{x} \\
\end{align*}
\]

\(^4\)https://www.w3.org/standards/
model compared to other alternative solutions. In our benchmark data, entities are naturally bound to a single list. Besides, the URI pattern could be used successfully in those cases where entities need to be bound to multiple lists or multiple positions in the same list, by using surrogate entities. However, this does not change the nature of the pattern.

**Properties.** URI-based lists indicate containment through the use of a class membership and a membership property. Order is absolute and given by sequential identifiers embedded in the item URI string.

### 4.3. Number-based Lists

Another practical model, used e.g. in the Sequence Ontology/Molecular Sequence Ontology (MSO) [17], also uses class membership or object properties to specify the elements that belong to a list, but use a literal value in a separate property to indicate order. For instance, the triple

```
<http://purl.org/midi-ld/piece/8cf9897/track00>
  midi:hasEvent
  <http://purl.org/midi-ld/piece/8cf9897/track00/event0006>
```

indicates that the object belongs to a list of events; and the additional triple

```
<http://purl.org/midi-ld/piece/8cf9897/track00/event0006>
  midi:absoluteTick "6"^^xsd:int
```

indicates that the event has index 6 (see Figure 3).

---

5We observe, and are aware of, the challenges introduced by this list data model. Furthermore, embedding list order indexes in URIs can have a critical impact on variance and uncertainty of both data and queries. Specifically for the latter, parsing these indexes may incur in low query performance and generality. Nonetheless, we have decided to include, and to a reasonable extent within its expressivity study, this model due to its popularity in Linked Data datasets.

6https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md

---

7http://motools.sourceforge.net/timeline/timeline.html#

8http://ontologydesignpatterns.org/wiki/Submissions:Sequence

---
5. Query formalisation and benchmark preparation

To evaluate the performance of atomic read-write Semantic Web list operations, we use the List.MID benchmark, “an RDF list data generator and query template set specifically designed for the evaluation of RDF lists” [26]. Following our methodology (Section 2), in this section we extend the benchmark and propose a set of SPARQL query templates for supporting our requirements, according to the patterns described in Section 4. In addition, we recall the procedure for producing the data and the extensions made to the data generator component.

5.1. Queries

We extend the list of supported operations in the benchmark, by considering all the atomic read-write operations that derive from the sequential data structures discussed in Section 3. In summary, the operations are:

- \texttt{FIRST}: returns the first element of the list
- \texttt{REST}: returns all the subsequent elements from the list starting from the second one
- \texttt{APPEND}: adds the specified element at the end of the list
- \texttt{APPEND\_FRONT}: adds the specified element at the beginning of the list
- \texttt{PREV}: returns all the previous elements from the list from the current one
- \texttt{POPOFF}: returns the first element of the list and removes it from the list
- \texttt{SET}: replaces the indicated element of the list with the supplied element
- \texttt{GET}: returns the indicated element of the list
- \texttt{REMOVE\_AT}: removes the indicated element of the list

In order to systematically evaluate these in datasets following one of the RDF list modeling patterns (Section 4), we implement these operations as SPARQL query templates, considering the definitions and axiomatic semantics specified in Section 3.

To satisfy the requirement of supporting Web application development, we assume to always return the sequence in a format that preserves its order. We choose to return items and lists in a tabular representation with the SPARQL result set format. Therefore, all the operations that return something are implemented as \texttt{SELECT} queries. Figure 6 shows the queries develop-
Fig. 6. Queries for the GET operation. The operation is supposed to return the entity at a given position in the list. In the minimal form, the operation returns the URI of the entity. It can be seen how each modelling solution requires a SPARQL query using different language operators. In some cases, it was impossible to avoid subqueries in order to compute the position of each item in the list and compare it with the required index. In addition, when the index parameter is positioned in the OFFSET clause, we reduce its value of 1 to keep it consistent with the semantics of

9 Note that in INDEXn, n refers to the parameter name, that can be more then one, and not the replaced index value.
the operator (OFFSET 1 returning the second element onward).

When the operation requires pointing to a specific item (GET, SET, and REMOVE_AT), we performed two experiments for each operation. The first, by picking a random position in the lower half of the sequence. The second, by picking a random index in the higher half of the sequence. In order to make the experiments comparable, we used the same value in querying all the models on the same list size. This method will give a reasonable approximation of real cases. The values are reported in Table 2, which also includes a decimal number representing the normalised value between 0 (representing the position of the first item in the list) and 1 (representing the last item in the list). The queries can be found online in the GitHub repository of the benchmark\(^{10}\).

### 5.2. Data preparation

Next to updating the benchmark with the aforementioned queries, we recall here the RDF list data generation procedure. The List.MID benchmark implements an algorithm to generate RDF datasets with lists according to the modeling patterns discussed above. The benchmark uses real-world data using MIDI files [34], a symbolic music encoding, as a basis. The reason for this is that MIDI files, and symbolic music notations in general, must encode musical events (the start of a note, the end of a note, the switching of one instrument for another, etc.) in strict sequential order to preserve musical coherence. Consequently, List.MID uses the midi2rdf algorithm proposed in [24] to generate RDF graphs from MIDI files. The generator uses the Semantic Web list models presented in Section 4 to encode lists of MIDI events. Importantly, each dataset is always stored in a different named graph in order to facilitate the systematic management of lists and the benchmarking of quadstores. List elements are given indices in the exact same order they occur in original MIDI files, guaranteeing predictable conversions. Figure 7 shows an excerpt of the MIDI ontology used by the original midi2rdf algorithm [24, 25, 34]. The relevant elements here are midi:Track, each containing a sequence of related musical events (e.g. notes played by one single instrument); and midi:Event, each representing a musical event that happens in a strict order within the track (e.g. the start of a note, the end of a note). In the present work, we extend the data generation component as follows:

- To generate datasets following the number-based list model, the tool can be set to generate unique and sequential IDs for list elements instead of using the original, to avoid collisions in the position of the items and be consistent with the definition of list we consider in this study.
- The list of operations is extended to the atomic read-write Semantic Web list operations derived from Section 3
- New dataset sizes are included to generate lists of 500, 2k, 3k, 5k, and 10k elements

Full instructions for using the benchmark and the source code are available on GitHub\(^1\).

We prepared a dataset for each modelling solution and six MIDI tracks of different sizes: 500, 1k, 2k, 3k, 5k, and 10k list items respectively. Therefore, we generate a dataset with a list of size 500 implementing, for example, the Seq pattern, one of size 1k, and so on for each sizes and model types, for a total of 30 datasets. The number of triples varies depending on the size of the list, the content of the item (the MIDI events),

![Excerpt of the MIDI ontology.](https://github.com/enridaga/list-benchmark/tree/master/benchmark/)

---

\(^{10}\)See https://github.com/enridaga/list-benchmark/tree/master/queries

\(^{11}\)https://github.com/midi-ld/List.MID
and the modelling solutions. In our previous work [13] it was possible to run experiments with very large datasets (up to 120k items), since we only tested a few read operations. In contrast, we observed in preliminary tests that many of the update operations considered here require significant resources and happen to be challenging even on lists of a few thousands items. Therefore, we reduced the overall scale of the benchmark in order to make the differences attributable to the modelling solutions observable in the results. We report statistics about the size of datasets used in the present work in Figure 8.

6. Experiments

We performed experiments with multiple triple stores. Each database was prepared by loading all the data, each one of them in a different named graph. At runtime, the query template was adapted to target a specific named graph, for example, the data for testing the Seq model on a 3k list item. Experiments are performed with the following databases and only considering the SPARQL RDF entailment regime:

- Virtuoso Open Source V7, configured to expect 12G of free RAM, no additional rules enabled except the basic SPARQL 1.1.
- Blazegraph 2.1.5, Java VM configured with 12G of max heap, without reasoning or inferencing support rather than the plain SPARQL 1.1 support.
- Apache Fuseki v3 on TDB, Java VM with 12G of max heap.
- Apache Fuseki v3 In Memory. This is the same system as the TDB-based but using a full in-memory setting, also with 12G of max heap space.

All databases are initialised with the same memory capacity. However, we could have given different resources to the database engines and maybe even different data alongside the graphs used in the benchmark. As long as these variables remain stable for all the experiments, we are still able to compare the results and observe how the different modelling solutions impact on the performance. The objective of the experiments is not to obtain a general measure of the cost of each operation or an evaluation of the efficiency of the database systems but to compare the observations to evaluate the usability of different modelling practices. Therefore, we assume no knowledge of the indexing mechanism of the database and of the nature of the data in the database, outside the presence of a list to be queried. We agree that in theory there may have an impact in the performance of the queries and for this reason we isolated each combination in a different graph, making the reasonable assumption that the FROM clause would be used in query optimisation for the purpose of reducing the impact of surrounding data.

The client application performing the queries and measuring the response time resides on the same machine as the database, in order to avoid the potential impact of network speed - that could change during the experiments - on the overall response time. Experiments are executed on a Linux VM equipped with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz 8-core and 32G RAM. During the experiments, no application was running on the instance apart from system processes, the target database server, and the experiment itself.

To summarise, the dimensions considered in our experiments are, therefore: (a) Model (one of): Seq, List, Number Index, SOP, URI Index (b) Dataset Size (one of): 500, 1k, 2k, 3k, 5k, 10k (c) Operation (one of): FIRST, GET (low index), GET (high index), REST, PREV, APPEND, APPEND_FRONT, POPOFF, SET (low index), SET (high index), REMOVE_AT (low index), and REMOVE_AT (high index). (d) Database (one of): Virtuoso, Blazegraph, Fuseki-TDB, Fuseki-Mem.

Therefore, each experiment combines one of 5 data models, 6 list sizes, 12 operations (each operation implemented in 5 different queries according to the related data model), and 4 database engines. These make a total of 1440 experiments performed using 60 different queries included in the benchmark. The process of running the experiments can be summarised as follows:

1. Select database engine
2. Load all the data
3. Restart the database engine
4. Select one data model and one operation
5. Run the related query on each list size, starting from the smaller and proceeding with lists of increasing sizes, interrupting the query after 5 min-

---

12 One may argue that the use of an index on the graph component may affect performance. However, whatever the impact of using the FROM clause is, it will be equally distributed in the various models.
utes if no response is returned. The client waits 5 seconds between repeating the query 10 times in total.
6. Stop the instance after the queries are all completed and move to point 3 for the next iteration, until all experiments are performed.
7. On results collection, we identify experiments returning an error or timed out as well as verify that the response content is correct.

In what follows, we report on overall response time, meaning the amount of time the client had to wait before obtaining the complete answer. We report on measures referring to average values on 10 repetitions, including the standard deviation (SD). Most of the read operations (FIRST, GET, REST, and PREV) reported an SD value below 10% of the total time. A few cases reported a higher SD, but they all referred to short response times (below the second) and are therefore not problematic (for example, the FIRST operation always completes below 200 milliseconds). Write operations (APPEND, APPEND_FRONT, POPOFF, SET, REMOVE_AT) reported sometimes a SD up to 20% of the total time. We can conclude that the reported averages are significant and represent well the response time of a client application querying lists of that form and size. Figures 11-22 report the average response time, including the standard deviation, in a sequence of bar charts. Supplementary material is available for reproducibility [14].

7. Mapping requirements and RDF modelling solutions

We discuss the results of the experiments by looking into each operation individually first. After that, we derive more general conclusions in relation to the abstract data structures introduced in Section 3. Results for each of the operators are presented in bar charts (see figures 11-22). Each figure presents four bar charts, each one of them dedicated to the results pertaining a database engine. The results are grouped by list size (horizontal axis) and reported as bar charts. The y axis represents the average response time, in milliseconds, and the standard deviation, distributed in logarithmic scale (except for Figure 11, where this was not needed). Unless differently specified in the coming discussion, a bar with diagonal lines indicates an experiment that was interrupted after the time limit of five minutes. Two additional horizontal lines provide a visual reference at 1 and 5 seconds. The following observations refer to the results with all the database engines, unless differently specified. A discussion of the performance of the different database engines would be out of scope. We avoid to do this unless when it is useful to argue on the behaviour of the data models. Reported queries show the SPARQL code sent to the database after any parameter substitution is applied.

**FIRST** The operator requires access and retrieval of the item at the top of the list. For example, the SPARQL query for the LIST data model is the following:

```sparql
SELECT ?event
WHERE {
    song:track00 a midi:Track;
    midi:hasEvents/rdffirst ?event
}
LIMIT 1
```

Results are reported in Figure 11. We consider the fluctuation in the measurements on the various sizes and the related standard deviation as not significant as the
response is always returned in less than 200 milliseconds in all cases.

**GET** *(low index and high index)* Results are reported in Figure 12 and 13. The operator performs a lookup and retrieves the *N*-th element of the sequence. The operation scales well for all models with a materialised index (URL, NUMB, and SEQ). When the index is implicit, it is derivable from the position of the element in the nested structure (SOP and LIST), as in the following query (taking the case of the SOP data model):

```sql
SELECT ?event WHERE {
  song:track00 midi:hasEvent ?event .
  ?event sequence:follows* ?prev .
  GROUP BY ?event
  ORDER BY ?i
  LIMIT 1
  OFFSET 657
}
```

As shown in Figure 14, this harms the scalability of the approach with a response time of more than four seconds with only 1k items in the list!

**PREV** This operation aims to retrieve the sequence except for the last element. The performance of the data models is comparable to the REST operator, except this time, the query needs to know the highest index, as the sequence is of dynamic size. A notable exception is the negative performance of the SEQ data model in combination with Virtuoso. The SPARQL query is akin to the following:

```sql
SELECT ?event WHERE {
  song:track00 midi:hasEvents [ ?seq ?event ] .
  BIND (xsd:integer(SUBSTR(STR(?seq), 45))) AS ?index .
  FILTER (?index < ?max) .
  # Find the Max
  WHERE {
    song:track00 a midi:Track ;
    midi:hasEvents [ ?seq [] ] .
    BIND (xsd:integer(SUBSTR(STR(?seq), 45))) AS ?pos
    ORDER BY ?index
  } ORDER BY ?i
}
```

Extracting the index from the predicate seems more demanding than doing the same from the entity URI. We are not quite sure why this is the case and can only note that the string manipulation is performed on the predicate rather than a subject or object triple. However, this was the only case in which the results have been partly inconsistent across triple stores.

**APPEND** This operation adds an element to the end of the list. To do this, a SPARQL query requires either to compute the new index value (for models materializing indexes) or reaching the last item in the sequence through path traversal. All data models perform reasonably well with small list sizes. The following is the query for the LIST data model:

```sql
DELETE { ?elt rdf:rest rdf:nil }
INSERT { ?elt rdf:rest }
```
model with the worst performance, for similar reasons to the case of APPEND_FRONT. We report the resulting query in Figure 9.

**SET (low index and high index)** This operation gives results that are similar to GET. LIST and SOP suffer from the same shortcomings, as it can be seen in Figure 12 and 20. NUMB, SEQ, and URI are more efficient data models.

**REMOVE_AT (low index and high index)** This operation is the most expensive of all, as it requires to find the item in to be removed and shift all subsequent items, refactoring additional data, when appropriate. Performance data is reported in Figure 19 and 20. SEQ performs better with the low index than with the high one but only going slightly over the five seconds average on some cases. The cost of the operation is on the side of materializing the index, for example, in case of NUMB (the most efficient of the data models):

This difference is reflected in the experiments results that show how SOP is generally more efficient than LIST, reported in Figure 16. However, NUMB, SEQ, and URI perform generally better.

**APPEND_FRONT** This operation is agile for both models based on linking items (SOP and LIST). In contrast, models based on materialised indexes require some refactoring of all the remaining items in the list! Results are displayed in Figure 17. However, only SEQ and URI seem to suffer from this operation, while the index update of NUMB seem very efficient. This difference is reflected in the experiments results illustrated in Figure 17.

**POPOFF** Similarly to the previous operation, moving the head of a list also requires an update of all indexes in models that materialise them. For example, SEQ and URI require to refactor the predicate and the entity names involved, which requires some string manipulation. These operations are reflected in the performance (see Figure 18. For example, the SEQ data model can be updated with the following query:
DELETE {
}
WHERE {
  SELECT ?olduri ?newuri WHERE {
    song:track00 midi:hasEvent ?olduri.
    BIND (xsd:integer(SUBSTR(str(?olduri), 7)) AS ?index) .
    FILTER (?index > 0) .
    BIND (?index-1 AS ?newindex) .
    BIND (IF (STRLEN(str(?newindex)) = 1, CONCAT("000", str(?newindex)),
              IF (STRLEN(str(?newindex)) = 2, CONCAT("00", str(?newindex)),
                  IF (STRLEN(str(?newindex)) = 3, CONCAT("0", str(?newindex)),
                      str(?newindex)))) AS ?strindex) .
    BIND (iri(concat("http://purl.org/midi-ld/piece/2473e18eec6cc55b82c5dddab3bea353/track00/event", ?strindex)) as ?newuri) .
  }
}

INSERT {
  song:track00 midi:hasEvent ?newuri.
}

Fig. 9. The SPARQL Update query for POPOFF + URI

Table 3

<table>
<thead>
<tr>
<th>Meanings of the Likert scale used to evaluate the data models in relation to each operation. In the figures 11-22, two thresholds are visualised at 1 second (green line) and 5 seconds (orange line), respectively.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5</strong> Very good</td>
</tr>
<tr>
<td><strong>4</strong> Good</td>
</tr>
<tr>
<td><strong>3</strong> Medium</td>
</tr>
<tr>
<td><strong>2</strong> Poor</td>
</tr>
<tr>
<td><strong>1</strong> Very Poor</td>
</tr>
</tbody>
</table>

We performed a qualitative analysis of the performance of data models on each operation, illustrated in Table 4. The data models are classified in 5 Likert categories, explained by Table 3. Finally, Table 5 summarises the fitness for use of each surveyed RDF data model with relation to the abstract data structures (the general use cases studied in the requirements section). The most visible performance issues are related to RDF models following the approach of linking subsequent items. Pragmatically, the only use case where they seem usable is when the Web application requires a stack. Indeed, in order to retrieve the pointer to one item, queries need to traverse all the preceding ones. This problem affects negatively the performance of all operations aimed at retrieving portions of the list (PREV, REST) but also the ones depending on finding the n-th elements of the sequence (GET, SET, REMOVE_AT). Operations requiring to switch the position of elements in the list, such as REMOVE_AT and POPOFF, still require to retrieve the target item before performing the index update. Interestingly, materialising the index as an RDF property seems to be the way to go in all cases, as managing the consistency of the index in a data property seems more sustainable than exploiting the links in the graph, also considering eventual book-keeping operations, such as index
DELETE {
  song:track00 midi:hasEvent ?event.
  ?next sequence:follows ?event.
  ?prev sequence:precedes ?event.
  ?event sequence:follows ?prev.
}

INSERT {
  ?next sequence:follows ?prev.
}
WHERE {
  song:track00 midi:hasEvent ?event
  OPTIONAL { ?event sequence:precedes ?next }.
  OPTIONAL { ?event sequence:follows ?prev }.
  {{
    SELECT ?event (count(?prev) as ?i)
    WHERE {
      song:track00 midi:hasEvent ?event.
      ?event sequence:follows ?prev.
    }
  }
  GROUP BY ?event
  ORDER BY ?i
  LIMIT 1
  OFFSET 23789
}]
}

Fig. 10. The SPARQL Update query for the SOP + REMOVE_AT operation.

Table 4

<table>
<thead>
<tr>
<th>Operation</th>
<th>SEQ</th>
<th>URI</th>
<th>NUMB</th>
<th>SOP</th>
<th>LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>GET (low index)</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GET (high index)</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>REST</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PREV</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>APPEND</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>APPEND_FRONT</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>POPOFF</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>SET (low index)</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SET (high index)</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>REMOVE_AT (low index)</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>REMOVE_AT (high index)</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

update. Overall, the efficiency of retrieving sequential linked data depends heavily on how they are modelled and can vary depending on the application use case. This is demonstrated experimentally by our results, where in 59 out of 60 combinations of data models and operations, performance results were coherent across the various databases (the exception being the PREV operation with the SEQ model that gave different performance on Virtuoso, see Figure 15b). Therefore, the triple store invariant hypothesis, introduced in [13] referring to read operations, is confirmed also for update operations. Indeed, modelling practices have an impact on the performance and availability of sequential linked data retrieval and management. Crucially, the behaviour of the various models is consistent among different triple stores and allow us to distinguish de-
Mapping of abstract list data types with RDF data models. In this table we answer the question: Should we adopt this data model to implement the given abstract data type? Possible answers are: Yes, Maybe, or No.

<table>
<thead>
<tr>
<th>Abstract Data Type</th>
<th>SEQ</th>
<th>URI</th>
<th>NUMB</th>
<th>SOP</th>
<th>LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DLL</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ST</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>QU</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Arr</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Finally, it is worth remarking how our evaluation of the data models was done with relation to efficiency of managing Linked Data with the purpose of supporting Web application development, leaving out other dimensions of analysis such as expressivity of the model at the logic level, compliance with high-level ontological requirements, and compliance to entailment regimes. Besides, we only focused on sequences accepting a single item in each position and most of the operations implemented in the benchmark (like the queries for GET and REMOVE_AT) would not be correct outside that assumption.

Our analysis is primarily directed to study sequential linked data in the context of Web application development, when the RDF data needs to move outside the database system possibly in a non-RDF format. This assumption had a significant impact in penalising models based on linked elements (SOP and LIST). Operations such as REST or PREV would be significantly more efficient when returning the list in-memory rather than producing a serialised ordered list as the distributed Web architecture requires.

In this study, we only take care of atomic operations. Realistic applications will be necessarily more complex and involve a number of queries performed in batch or subsequently, having different workload. However, we assume that their complexity and performance will necessarily be a function of the efficiency of the atomic operations studied in this article.

The motivation behind the need for evaluating pragmatically competitive modelling solution comes from the unique socio-technical context of Linked (Open) Data, where datasets are designed and implemented for a multiplicity of potential use cases, some of them not known in advance. This context is radically different from the case of application-specific databases, whose data model can be optimised for specific sets of queries. Our work [13] and this article, report on the first pragmatic study on the efficiency of data management operations from the point of view of alternative
8. Related work

The application of Web APIs backed by SPARQL Endpoints is an active research area, mainly concerned with making it easier for developers to interact with RDF data [15, 23]. This concern is a core motivation for the present work, whose purpose is to characterise the requirements for a Sequential Linked Data API and evaluate possible implementations of such API in SPARQL by comparing a set of prototypical options as data models.

In our previous work [13] we propose a set of list modelling patterns that emerge from global Linked Data publishing. We already reviewed modelling solutions for lists in Section 4. These patterns are used in a subsequent benchmark, List.MID [26], that we also apply and extend here. We refer to [13] for the related work on modelling sequential linked data.

We focus on practical approaches for benchmarking with the SPARQL language. For a theoretical study on the complexity of SPARQL, see [29]. The Semantic Web community has developed a number of benchmarks for evaluating the performance of SPARQL engines, proposing both benchmark queries and benchmark data. The Berlin SPARQL Benchmark (BSBM) [6] generates benchmark data around exploring products and analyzing consumer reviews. The Lehigh University Benchmark (LUBM) [20] facilitates the evaluation of Semantic Web repositories by generating benchmark data about universities, departments, professors and students. SP2Bench [31] is a benchmark for SPARQL processors that enables comparison of optimization strategies, the estimation of their generality, and the prediction of their benefits in real-world scenarios; it includes a benchmark data generator based on the DBLP bibliographic database [22]. Similarly, the DBpedia SPARQL benchmark [27] focuses on human-written queries against non-relational schemas. The Waterloo SPARQL Diversity Test Suite (WatDiv) focuses on “a wide spectrum of SPARQL queries with varying structural characteristics and selectivity classes” [3]. Other datasets, such as Linked SPARQL Queries (LSQ) [30], focus exclusively on offering benchmark queries from (structured) SPARQL query logs but typically miss benchmark data against which to run these queries. More recently, frameworks aiming at the comparability and integration of these benchmarks have emerged, such as IGUANA [10].

Pragmatic approaches to benchmarking are not new, and it is common practice to develop ad-hoc benchmarks to support specific applications (e.g. [33]). Benchmark methodologies have been proposed for covering specific aspects of SPARQL, for example, federation [19].

The Linked Data Benchmark Council (LDBC) is an industry-led initiative aimed at raising state of the art in the area by developing guidelines for benchmark design. For example, LDBC stresses the need for reference scenarios to be realistic and believable, in the sense that should match a general class of use cases. Besides, benchmarks should expose the technology to a workload, and by doing that it is essential to focus on choke points when defining the various tasks [4]. These guidelines inspire our previous work on proposing a benchmark, List.MID, for evaluating the performance of common Semantic Web list representations under various query engines and operations [26].

It is important to stress how our objective is different from existing database benchmarks, that are designed
to evaluate the performance of RDF database systems under certain workloads or to evaluate their overall efficiency to support certain SPARQL operators. In other words, we are not benchmarking RDF database systems and their efficiency in dealing with lists. In contrast, our aim is to compare the alternative modelling approaches to lists in RDF and, therefore, study their usability by Web applications that want to query and update linked data. Where operations need it, we follow existing evaluation approaches on fixed-point sequence segmentation [11] and limiting variance to one single sequence per dataset.

9. Conclusions

In this article, we focused on Sequential Linked Data and evaluated the feasibility of an API specification for managing lists on the Semantic Web. With the aid of a model-centric and task-oriented approach to benchmark development [13], we were able to study pragmatically how to better manage Sequential Linked Data and identified a fundamental performance problem of typical, recommended solutions. A significant result lies in the fact that managing indexes as literals is by far more sustainable than relying on the graph structure to establish order or embedding the index value in an entity or predicate strings.

In the future, we aim at further exploring the applications of Sequential Linked Data. We expect that a thorough analysis of end-user applications will expand the set of operations. Specific cases could require testing membership containment and manipulating portions of the list. Moreover, incorporating our proposed list operations in the SPARQL specification could open new research possibilities in query performance and data storage, for example by encapsulating and pushing down such list operations to underlying store processes and consequently reducing query complexity. Finally, we work towards the development of a fully-fledged linked data Web API for efficient management of ordered sequences in RDF.

References


Fig. 11. **FIRST.** The operation is very efficient in all our experiments. Fluctuation of SD is not significant as the response is always returned in less than 200 milliseconds.
Fig. 12. **GET (low index)**. The operation scales well for all models with a materialised index (URI, NUMB, and SEQ), and it is problematic with LIST and SOP. The errors produced by Fuseki (Figures 12c and 12d) are Java Stack Overflow errors, while the others refer to the client waiting for more than five minutes. (Results are equivalent to the ones of GET with the high index. See Figure 13.)
Fig. 13. **GET (high index)**. The operation scales well for all models with a materialised index (URI, NUMB, and SEQ), and it is problematic with LIST and SOP. The errors produced by Fuseki (Figures 12c and 12d) are Java Stack Overflow errors, while the others refer to the client waiting for more than five minutes. (Results are equivalent to the ones of GET with the low index. See Figure 12.)
Fig. 14. REST. SOP and LIST perform poorly as the databases require to traverse the graph to retrieve all the elements and sort them in memory.
Fig. 15. PREV. The performance of the data models is comparable to the REST operator (see Figure 14), except this time, the query needs to know the highest index, as the sequence is of dynamic size. A notable exception is the negative performance of the SEQ data model in combination with Virtuoso (Figure 15b). This was the only case in which the results have been inconsistent across triple stores.
Fig. 16. **APPEND.** All data models perform reasonably well with small list sizes. In contrast with LIST, the SOP model has the advantage of directly linking all items to the container entity (the list) and, therefore, does not require to traverse the whole list. However, NUMB, SEQ, and URI perform generally better.
Fig. 17. **APPEND FRONT**. This operation is agile for both models based on linking items (SOP and LIST). In contrast, models based on materialised indexes require some refactoring of all the remaining items in the list! However, the index update seem very efficient in the case of NUMB.
Removing the head of a list also requires an update of all indexes for SEQ, URI, and NUMB. However, SEQ and URI require some string manipulation. This is reflected in the overall performance. URI is the model with the worst performance, for similar reasons to the case of APPEND_FRONT (Figure 17).
Fig. 19. SET (low index). LIST and SOP suffer from the same shortcomings. NUMB, SEQ, and URI are more efficient data models. (Results are equivalent to the ones of SET with the high index. See Figure 20.)
Fig. 20. **SET (high index).** LIST and SOP suffer from the same shortcomings. NUMB, SEQ, and URI are more efficient data models. (Results are equivalent to the ones of GET with the low index. See Figure 19.)
Fig. 21. REMOVE AT (low index). This operation is the most expensive of all, as it requires to find the item in to be removed and shift all subsequent items, refactoring additional data, when appropriate. Results are generally similar to the ones of REMOVE_AT with the high index (See Figure 22). NUMB is the most efficient data model.)
Fig. 22. **REMOVE AT (high index)**. This operation is the most expensive of all, as it requires to find the item in to be removed and shift all subsequent items, refactoring additional data, when appropriate. Results are similar to the ones of GET with the low index (See Figure 21), with the exception of SEQ. NUMB is the most efficient data model.