Journal Title 0 (2020) 1
IOS Press

Sequential Linked Data: the State of Affairs

Enrico Daga [a,*], Albert Meroño-Peñuela [b], and Enrico Motta [a]
[a] *Knowledge Media Institute, The Open University, United Kingdom*
E-mails: enrico.daga@open.ac.uk, enrico.motta@open.ac.uk
[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 `rdf:List` in the RDF W3C specification. A pragmatic solution refers to each member of the list using RDF containment membership properties (`rdf:_1`, `rdf:_2`,...) within an n-ary relation of type `rdf:Seq`. Alternative options may involve picking a solution from the Ontology Design Patterns catalogue [12], for example, the Sequence ODP[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-

*Corresponding author. E-mail: enrico.daga@open.ac.uk.

[1] Sequence: http://ontologydesignpatterns.org/wiki/Submissions:Sequence.

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 questions, 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 (triplestore 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#>
xml: <http://www.w3.org/XML/1998/namespace>
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.

Requirements The reference scenario is the access and manipulations of Lists stored as RDF data and exposed on a Semantic Web API, backed by a SPARQL endpoint, following an approach akin to [15]. In the initial phase, we identify the core set of standard, atomic read-write operations that a generic, SPARQL-based API for the management of lists should support. To do this, we select computer science data structures that could be used to encode such data requirements, to infer possible operations. Then, we collected Linked Data models for encoding the same data requirement, and encoded the operations using SPARQL to return an ordered sequence for developers to use. By evaluating the efficiency of the SPARQL queries, we want to answer the question: how should we encode sequences to support fast and agile development of applications with Linked Data? To focus on lists, we restrict ourselves to *sequential* data structures, i.e. an element can only reference linearly a single following or preceding element (this excludes data structures such as trees or graphs). We define a list as an *unbound ordered distinct set*, where each element appears only once in the sequence, which does not have any restriction of length[2]. The main objective is to identify the fundamental operations for sequential data access and manipulation. By relying on computer science data models for sequences, we aim to collect all these operations.

Survey Modelling solutions should be relevant to practitioners by referring to a real dataset adopting the modelling practice. After listing the modelling solutions, we abstract them in *structural patterns* and ensure these patterns are minimal with respect to the data model. Surveyed schemas can incorporate other requirements (for example, a list of authors may include components to express things other than the order such as affiliation or email). Here, we reuse the survey of RDF Lists included in [13].

Formalisation Each list modelling solution and operation should be encoded in RDF and SPARQL. Notably, each list modelling pattern must be challenged to fit the API operations designed in the requirements phase and the respective solutions encoded in SPARQL queries. By doing this, it is fundamental to ensure that the output is *semantically equivalent*, ideally the same, for all query variants. Besides, it is fundamental that queries are *minimal* by keeping them in the simplest form, for example, adopting good practices for SPARQL query optimization [32]. Particularly: (a) avoiding subqueries, when possible, (b) reducing the use of SPARQL operators to the minimum necessary, (c) projecting variables only when strictly necessary, and (d) preferring blank nodes to named variables[3]. We build upon our previous work [13, 26] to achieve this.

Evaluation The objective of this phase is to evaluate the different solutions empirically. As Linked Data are based on a Web application architecture (the client/server approach), the performance measure we focus on is *overall response time*. This means that the cost of the operation includes both the query evaluation and also the output serialisation and the transfer of the HTTP response payload to the client. In order for results to be relevant to real applications, we measure the overall response time with different data sizes and generate a set of realistic datasets at different scales. We perform experiments for each modelling prototype, each atomic read-write operation, with different dataset sizes, and with different database engines.

Therefore, although database engines may perform differently, we expect that the experimental results would evidence a difference that depends on the nature of the modelling solutions. Here, we focus on the relationship between models, operations, dataset sizes, and triplestore implementations, to foster a broader discussion on where the strengths, and possible weaknesses, of a SPARQL-based list manipulation API lie.

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.

[2]Typically, 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.

[3]In 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:

$$\begin{aligned} append\_front &: (E \times L) \rightarrow L \\ first &: L \rightarrow E \\ rest &: L \rightarrow L \\ \\ first(append\_front(e,l)) &= e \\ rest(append\_front(e,l)) &= l \\ e \in E, l &\in L \end{aligned} \tag{1}$$

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 th empty list.

Examples. The use of Linked Lists is straightforward for users of most programming languages. E.g., $append\_front(\text{alice}, [\text{bob}, \text{carl}]) = [\text{alice}, \text{bob}, \text{carl}]$; 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:

$$\begin{aligned} append\_front &: (E \times L) \rightarrow L \\ append &: (L \times E) \rightarrow L \\ first &: L \rightarrow E \\ rest &: L \rightarrow L \\ prev &: L \rightarrow L \\ \\ first(append\_front(e,l)) &= e \\ rest(append\_front(e,l)) &= l \\ prev(append(l,e)) &= l \\ e \in E, l &\in L \end{aligned} \tag{2}$$

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:

$$\begin{aligned} popoff &: (S) \rightarrow E \\ \\ popoff(append\_front(e,s)) &= s \\ first(append\_front(e,s)) &= e \\ append\_front(popoff(s),s) &= s \\ first(popoff(s),s) &\neq s \\ \\ s \in S, e &\in E \end{aligned} \tag{3}$$

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:

$$
\begin{aligned}
first(append(v, [])) &= v \\
popoff(append(v, [])) &= [] \\
\\
first(append(v, append(w, Q))) &= \\
first(append(w, Q)) & \\
\iff Q &= [] \\
\\
popoff(append(v, append(w, Q))) &= \\
append(v, popoff(append(w, Q))) & \\
\iff Q &= []
\end{aligned}
\tag{4}
$$

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:

$$
\begin{aligned}
set &: (E \times I \times A) \to A \\
get &: (I \times A) \to E \\
remove\_at &: (I \times A) \to A \\
\\
get(i, set(e, i, a)) = e &\iff |a| \geqslant i \\
\\
remove\_at(i, a) &= b \\
\to a_j = b_j &\iff j < i; \\
a_{j-1} = b_j &\iff j > i \\
\\
e \in E; a, b \in A; i &\in I
\end{aligned}
\tag{5}
$$

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_at* 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 $get(2, [\text{alice}, \text{bob}, \text{carl}]) = \text{bob}$, and removed with $remove\_at(2, [\text{alice}, \text{bob}, \text{carl}]) = [\text{alice}, \text{carl}]$.

Table 1

Summary of operations and relevant abstract data structures.

Operation	*LL*	*DLL*	*ST*	*QU*	*ARR*
$first : L \to E$	x	x	x	x	-
$rest : L \to L$	x	x	-	-	-
$append : L \times E \to L$	x	x	-	x	-
$append\_front : E \times L \to L$	x	x	x	-	
$prev : L \to L$	-	x	-	-	-
$popoff : L \to L$	-	-	x	x	-
$set : E \times I \times L \to L$	-	-	-	-	x
$get : I \times L : E$	-	-	-	-	x
$remove\_at : I \times L \to L$	-	-	-	-	x

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 `rdf:Bag`, `rdf:Alt`, `rdf:Seq` to represent collections. Since `rdf:Bag` is intended for unordered elements, and `rdf:Alt` for "alternative" containers, whose typical process-

[4] https://www.w3.org/standards/

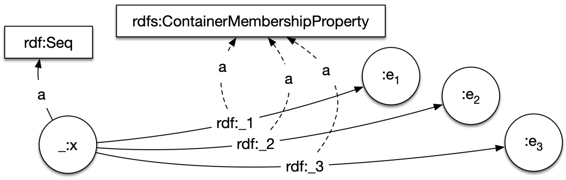


Fig. 1. The RDF Sequence model. `_:x` represents the list entity, here an instance of `rdf:Seq` according to the standard.

ing will be to select one of its members, these two models do not fit our sequence definition, and thus we do not include them among our candidates. Conversely, we do consider *RDF Sequences*: collections represented by `rdf:Seq` and ordered by the properties `rdf:_1`, `rdf:_2`, `rdf:_3`, ... instances of the class `rdfs:ContainerMembershipProperty` (see Figure 1).

Properties. RDF Sequences indicate membership through various *properties*, which are used in triples in *predicate position*. Ordering of elements is *absolute* in such predicates through an integer index after an underscore ("_").

4.2. URI-based Lists

An approach followed by some resources in the LOD community [5, 25] consists of establishing list membership through an explicit property or class membership, and assigning order by a unique identifier embedded in the element's URI. For instance, the triple

```
<http://ld.zdb-services.de/resource/1480923-0>
  rdf:type
    <http://purl.org/ontology/bibo/Periodical>
```

indicates that the subject belongs to a list of periodicals with list order 14809234, while the triple

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

identifies the 7th event in a MIDI track [25] (see Figure 2). Clearly, a URI-based data model could have many ways to incorporate the index value and this may affect the performance of the string manipulation task in the overall query execution. However, here we evaluate how the necessity of a string manipulation function (whatever it is) impacts the usability of the data

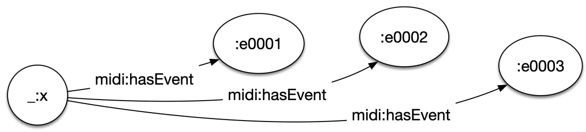


Fig. 2. The URI-based list model. `_:x` represents the list entity, not necessarily a blank node, linked to all list elements. In this example, the order of the elements is implicit in the tail of the URI, although it could appear before.

model compared to other alternative solutions.[5] 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],[6] 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/8
                    cf9897/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).

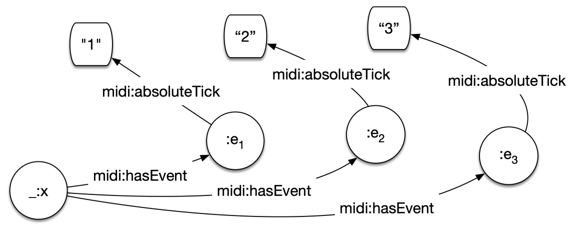


Fig. 3. The Number-based list model. `_:x` represents the list entity, here connecting to all list elements through a specific `midi:hasEvent` property that links tracks to events. Order is explicit in each list element through other arbitrary properties (e.g. `midi:absoluteTick`).

Properties. Number-based lists indicate containment through the use of *class membership* and *a membership property*. Order is *absolute* and given by an integer index in a literal as an object of an additional property.

4.4. Sequence Ontology Pattern

A number of models use RDF, RDFS and OWL to represent sequences in domain specific ways. For example, the Time Ontology [21] and the Timeline Ontology[7] offer a number of classes and properties to model temporality and order, including timestamps, but also before/after relations. The *Sequence Ontology Pattern*[8] (SOP) is an ontology design pattern [18] that "represents the 'path' cognitive schema, which underlies many different conceptualizations: spatial paths, time lines, event sequences, organizational hierarchies, graph paths, etc.". We select SOP as an abstract model representing this group of list models (see Figure 4).

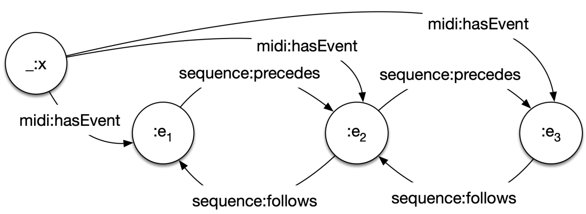


Fig. 4. The Sequence Ontology Pattern model. `_:x` represents the list entity that connects to list elements through the `midi:hasEvent` property. The first list element follows no other element, and the last precedes no other element.

[5] We 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 generability. 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.

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

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

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

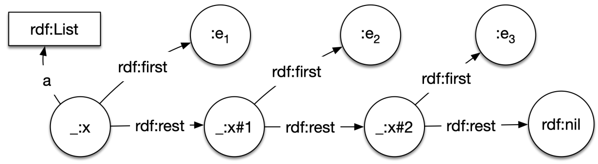


Fig. 5. The RDF List model. `_:x` represents the list entity in a blank node of type `rdf:List`, as defined in the standard. Subsequent sublists are defined analogously through blank nodes connected via `rdf:rest`; the list ends with `rdf:nil`.

Properties. SOP lists indicate list membership through *properties*. Order is *relative* and given by the sequential forward or backward traversal of the sequence.

4.5. RDF Lists

The RDFS recommendation [8] defines a vocabulary to describe closed collections or *RDF Lists*. Such lists are members of the class `rdf:List`. Resembling LISP lists, every element of an RDF List is represented by two triples: <L_k `rdf:first` E_k>, where E_k is the k-th element of the list; and <L_k `rdf:rest` L_{k+1}>, representing the rest of the list (in particular, `rdf:nil` to end the list) (see Figure 5).

Properties. RDF Lists indicate membership through the use of a *unique property* `rdf:first` in *predicate position*. Ordering of elements is *relative* to the use of the `rdf:rest` property, and given by the sequential forward traversal of the list.

In the following sections we will refer to the data models using the following names:

SEQ Sequences are represented using the `rdf:Seq` model, where the position is hard-coded into a predicate URI.
URI The position of the items in the list is hard-coded into the list item URI identifier.
NUMB The position of the item is recorded as a literal value of a list item property.
SOP The sequence is modelled following the Sequence ontology design pattern.
LIST The sequence is modelled according to the RDF List specification.

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:

- *FIRST*: returns the first element of the list
- *REST*: returns all the subsequent elements from the list starting from the second one
- *APPEND*: adds the specified element at the end of the list
- *APPEND_FRONT*: adds the specified element at the beginning of the list
- *PREV*: returns all the previous elements from the list from the current one
- *POPOFF*: returns the first element of the list and removes it from the list
- *SET*: replaces the indicated element of the list with the supplied element
- *GET*: returns the indicated element of the list
- *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 SELECT queries. Figure 6 shows the queries devel-

```
SELECT ?event
FROM <%%DATASET%%>
WHERE {
   <%%TRACK%%> midi:hasEvents [ ?seq ?event ] .
   BIND (xsd:integer(SUBSTR(str(?seq), 45)) AS ?
        index)
   FILTER (?index = %%INDEX1%%)
} LIMIT 1
```

(a) SEQ

```
SELECT ?event
FROM <%%DATASET%%>
WHERE {
   <%%TRACK%%> midi:hasEvent ?event .
   BIND (xsd:integer(SUBSTR(str(?event), 77)) AS ?
        index)
   FILTER (?index = %%INDEX1%%)
}
LIMIT 1
```

(b) URI

```
SELECT ?event
FROM <%%DATASET%%>
WHERE {
   <%%TRACK%%> midi:hasEvent ?event .
   ?event midi:id ?tick
} ORDER BY ASC(?tick)
LIMIT 1 OFFSET %%INDEX1%%
```

(c) NUMB

```
SELECT ?event
FROM <%%DATASET%%>
WHERE {{
    SELECT ?event (count(?prev) as ?i)
    WHERE {
        <%%TRACK%%> midi:hasEvent ?event .
                ?event sequence:follows* ?prev .
    } GROUP BY ?event
}}
ORDER BY ?i LIMIT 1 OFFSET %%INDEX1%%
```

(d) SOP

```
SELECT ?event
FROM <%%DATASET%%>
WHERE {{
    SELECT ?event (count(?mid) as ?i)
    WHERE {
       <%%TRACK%%>  midi:hasEvents ?events .
       ?events rdf:rest* ?mid .
       ?mid rdf:rest* ?elt .
       ?elt rdf:first ?event
    } GROUP BY ?event
}}
ORDER BY ?i LIMIT 1 OFFSET %%INDEX1%%
```

(e) LIST

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.

oped for the GET operation. All queries return the list item at a given position. The queries include three template parameters: (a) DATASET, pointing to a named graph with the list of a specific size; (b) TRACK, to be substituted with the URI of the song MIDI Linked Data database used by the benchmark to represent the list in that dataset; and (c) INDEX*n*, used when the operation requires to reference a specific position in the list (as in the case of GET or REMOVE_AT)[9]. As it can be seen from the examples in Figure 6, the queries are developed in a compact way, following the minimality principles explained in Section 2. 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 using 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 INDEX*n*, *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].

Size	Low value (position)	High value (position)
500	168 (0.33)	332 (0.66)
1k	343 (0.34)	657 (0.65)
2k	578 (0.29)	1222 (0.61)
3k	728 (0.24)	2472 (0.82)
5k	1211 (0.24)	3789 (0.75)
10k	2678 (0.26)	7322 (0.73)

Table 2

Positions of the *n-th* items for benchmarking the operations GET, SET, and REMOVE_AT with the various list sizes.

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

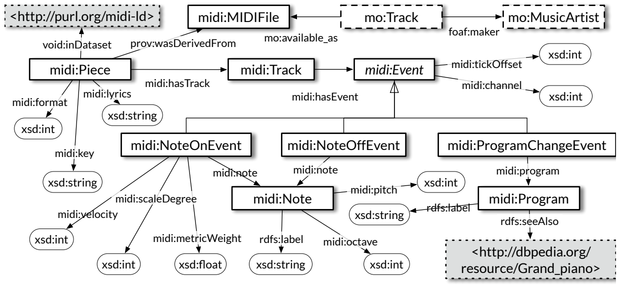


Fig. 7. Excerpt of the MIDI ontology. Tracks contain lists of sequential MIDI events.

is always stored in a different named graph in order to falicitate 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 predicable 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[11].

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),

[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[12].

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 then 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 klong 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 these 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 makes 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.

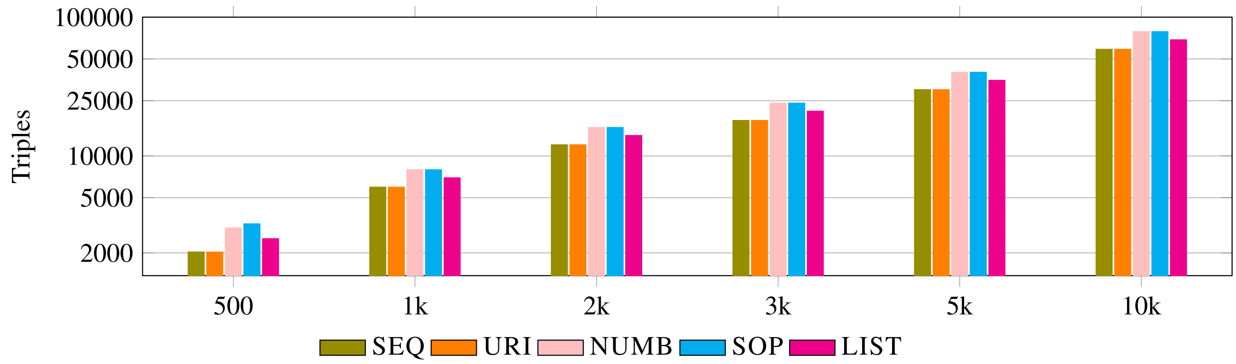


Fig. 8. Size of the datasets in number of triples

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 values 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:

```
SELECT
  ?event
WHERE {
  song:track00 a midi:Track ;
    midi:hasEvents/rdf:first ?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 (URI, 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):

```
SELECT
 ?event
WHERE {{
 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 657
```

However, this happens at a high cost, as it can be seen from the results of both experiments. In particular, with large lists, the operation either times out (Blazegraph and Virtuoso), or returns an error (a StackOverflow Java exception, in the cases of Fuseki with a 10k-sized list). This result is significant and it will be shown how the reasons behind it impact several operations that depend on retrieving items at specific positions in the list.

REST The operation returns the content of the sequence, except for the first element, following the sequence order. The following is an example query for the SEQ data model:

```
SELECT
  ?event
WHERE {
  song:track00 midi:hasEvents [ ?seq ?event ] .
  # extracted from, e.g. rdf:_32
  BIND (xsd:integer(SUBSTR(STR(?seq), 45))
        AS ?index) .
%   FILTER (?index > 1)
}
ORDER BY ?index
OFFSET 1
```

Models based on a nested structure perform poorly as the databases require to traverse the graph for retrieving all the elements, performing an aggregation to compute the index, and sort the returned elements, as in the case of LIST:

```
SELECT ?event
WHERE {
 {{SELECT ?event (count(?mid) as ?i)
  WHERE {
  song:track00 midi:hasEvents ?top .
       ?top rdf:rest* ?mid  .
       ?mid rdf:rest* ?elt .
       ?elt rdf:first ?event .
       FILTER (?elt != ?top)
  }
  GROUP BY ?event
  ORDER BY ?i}}
}
```

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:

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

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:

```
DELETE { ?elt rdf:rest rdf:nil }
INSERT {
 ?elt rdf:rest [
```

```
  a rdf:List ;
  rdf:first <http://example.org/appended-event> ;
  rdf:rest rdf:nil
 ]
} WHERE {
 song:track00 midi:hasEvents ?events .
 ?events rdf:rest* ?elt .
 ?elt rdf:rest rdf:nil
}
```

In relation to the latter problem, it is interesting to note how 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:

```
INSERT {
 song:track00
  midi:hasEvent
   ex:appended-item .
 ex:appended-item
   sequence:follows ?event .
 ?event sequence:precedes
   ex:appended-item .
} WHERE {
 song:track00 midi:hasEvent ?event .
 FILTER NOT EXISTS {
   ?event sequence:precedes []
 }
}
```

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, removing 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 {
 ?events rdf:_1 ?event
}
INSERT {
 ?events ?shifted ?event
}
WHERE {
 BIND (iri(concat("http://www.w3.org/1999/02/22-rdf
      -syntax-ns#_", str(?index - 1))) as ?shifted)
      .
 song:track00 midi:hasEvents ?events .
 ?events ?seq ?event .
 BIND (xsd:integer(SUBSTR(str(?seq), 45))
    AS ?index) .
 FILTER (?index > 1)
}
```

URI is the 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):

```
DELETE {
 song:track00 midi:hasEvent ?e .
 ?e midi:id 23789 .
 ?event midi:id ?oldId
}
INSERT {
 ?event midi:id ?newId
}
WHERE {{
 SELECT ?event ?oldId ?newId WHERE {
  song:track00> midi:hasEvent ?event .
  ?event midi:id ?oldId .
  FILTER ( ?oldId > 23789 ) .
  BIND ((?oldId-1) AS ?newId)
 }
}}
```

For SOP and LIST, the query needs to traverse the links and perform multiple joins to refactor the graph structure (see Figure 10).

Looking at our results, we can now answer the key questions of our research:

– (1) *Do RDF lists modelling practices have an impact on the performance and availability of sequential Linked Data?*
– (2) *Can we distinguish between patterns and anti-patterns able to model lists in representative generic use cases?*

```
DELETE {
 song:track00 midi:hasEvent ?popthis ;
                   midi:hasEvent ?olduri .
 ?popthis ?p1 ?o1 .
 ?olduri ?p ?o .
}
INSERT {
 song:track00 midi:hasEvent ?newuri .
 ?newuri ?p ?o .
}
WHERE {
 # Point to the first element
 BIND (<http://purl.org/midi-ld/piece/2473e18eec6cc55b82c5dddab3bea353/track00/event0000> as ?popthis) .
 OPTIONAL { ?popthis ?p1 ?o1 }
 OPTIONAL { ?olduri ?p ?o }
 {{
   SELECT ?olduri ?newuri WHERE {
    song:track00 midi:hasEvent ?olduri .
    BIND (xsd:integer(SUBSTR(str(?olduri), 77))
        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) .
   }
 }}
}
```

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

Table 3

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.

5	Very good	Average response time below 1 second on all list sizes and database engines
4	Good	Average response time below 1 second, with exceptions on some list sizes or database engines, never above the 5 seconds
3	Medium	Average response vary often above 5 seconds
2	Poor	Average response time always above 5 seconds
1	Very Poor	Average response time always above 5 seconds with some errors or interruptions after timing out

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 .
                ?event sequence:precedes ?next .
                ?next sequence:follows ?event .
                ?prev sequence:precedes ?event .
                ?event sequence:follows ?prev .
}
INSERT {
        ?prev sequence:precedes ?next .
        ?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

Table 4

Performance of data models with relation to the operators. 5: very good, 1: very poor. (*) The combination PREV/SEQ is good on three out of four database engines.

	SEQ	URI	NUMB	SOP	LIST
FIRST	5	5	5	5	5
GET (low index)	5	5	5	1	1
GET (high index)	5	5	5	1	1
REST	5	5	5	1	1
PREV	3*	5	5	1	1
APPEND	4	5	4	5	1
APPEND_FRONT	4	4	5	5	5
POPOFF	4	5	4	5	5
SET (low index)	5	5	5	1	1
SET (high index)	5	5	5	1	1
REMOVE_AT (low index)	4	1	5	1	1
REMOVE_AT (high index)	3	1	5	1	1

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-

Table 5

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: **Y**es, **M**aybe, or **N**o.

Abstract Data Type	Operations	SEQ	URI	NUMB	SOP	LIST
LL	FIRST, REST, APPEND, APPEND_FRONT	Y	Y	Y	N	N
DLL	FIRST, REST, PREV, APPEND, APPEND_FRONT	M	Y	Y	N	N
ST	FIRST, APPEND_FRONT, POPOFF	Y	M	Y	Y	Y
QU	FIRST, APPEND, POPOFF	Y	M	Y	Y	M
Arr	SET, GET, REMOVE_AT	M	M	Y	N	N

sign patterns that perform well in practice from others that perform worse —from the point of view of the identified requirements. Finally, the most efficient way of representing order is by using indexes in property values like in *NUMB*.

Embedding the ordering semantics in string URIs does not seem an elegant solution. Indexes hidden in URIs perform less well in the case of management operations, both on the entity (subject/object) and the `rdf:Seq` method (predicate). The reasons are probably related to database indexes on the basic triple patterns. However, here we focus on trends observed among the various database engines and do not discuss specific differences between them. Using the `rdf:Seq` pattern may be a reasonable solution *iff* SPARQL engines would account of the special meaning of container membership properties and sort those predicate URIs accordingly. A small update to the SPARQL specification seems a reasonable way to go.

In our previous work [13], we hypothesised that modelling solutions that do not store an index (SOP and LIST) would, in principle, better fit management operations. In this article, we considered a thorough set of core operations and evaluated the various modelling solutions with relation to the problem of *managing* sequences as Linked Data. **With the given results, the methods relying on `rdf:List` (the recommended standard) and `SOP` (*a high-quality ontology engineering solution*) underperform in commonly used triple stores and, under these circumstances, their use should be discouraged** for managing lists in Linked Data Web applications. We can only recommend their usage when requirements such as the expressivity of the representation or the compliance to OWL2 reasoning are a priority over SPARQL query performance. The inclusion of the URI-based pattern in our study indeed corroborated the unpredictability, variance, and low query generality of this data model; and despite its popularity in Linked Data we cannot recommend its usage under those requirements.

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 then 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

RDF modelling patterns. We establish our approach on the triple store invariant hypothesis, introduced in this article. As a result, in the proposed benchmark, we don't give much importance to the way system-specific optimisations may interfere with the measurements, focusing exclusively at keeping those elements equivalent across the various experiments. All models are evaluated with increasing list sizes on the same running instance. In our experiments, all data are in the same database. We consider this aspect a good feature of our benchmark, representing the existence of a "data context" in which our data models (the objects of our observations) reside. In real applications, the data that is being queried and used typically resides alongside other data. Although indexes in the database may be shared across named graphs, and this may contribute to general performance, we assume that this is of secondary importance and that it is sufficient to keep the data context the same for all the experiments. However, this is an assumption of our approach, and we do not evaluate that. Future work could focus on studying the role of the data that is not needed by the query but is still in the database. Finally, our results show a broad consistency in behaviour across different triple stores, which is what we wanted to prove: that the impact of modelling solution is *triple store invariant*. In future work, we may want to also evaluate how the impact of data models is *data context invariant*.

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. SP^2Bench [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][13]. 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

[13]See also https://github.com/dice-group/triplestore-benchmarks

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 triplestore data structures 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

[1] "Bounded queue", in Dictionary of Algorithms and Data Structures [online] (2019), https://xlinux.nist.gov/dads/HTML/boundedqueue.html, accessed 22/09/2019

[2] "Queue", in Dictionary of Algorithms and Data Structures [online] (2019), https://www.nist.gov/dads/HTML/queue.html, accessed 22/09/2019

[3] Aluç, G., et al: Diversified Stress Testing of RDF Data Management Systems. In: The Semantic Web – ISWC. pp. 197–212. Springer, Cham (2014)

[4] Angles, R., et al: The linked data benchmark council: a graph and rdf industry benchmarking effort. ACM SIGMOD Record **43**(1) (2014)

[5] Beek, W., et al: LOD Laundromat: a uniform way of publishing other people's dirty data. In: Semantic Web – ISWC. pp. 213–228. Springer (2014)

[6] Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal on Semantic Web & Information Systems **5**(2), 1–24 (2009)

[7] Black, P.E.: Dictionary of algorithms and data structures (1998), https://www.nist.gov/dads/HTML/, accessed 22/09/2019

[8] Brickley, D., Guha, R.: RDF Schema 1.1. Tech. rep., World Wide Web Consrotium (2014), https://www.w3.org/TR/rdf-schema/

[9] Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: Sparql web-querying infrastructure: Ready for action? In: International Semantic Web Conference. pp. 277–293. Springer (2013)

[10] Conrads, F., et al: Iguana: A generic framework for benchmarking the read-write performance of triple stores. In: The Semantic Web - ISWC 2017 (2017)

[11] Czajka, P., Radoszewski, J.: Experimental evaluation of algorithms for computing quasiperiods (2019)

[12] Daga, E., Blomqvist, E., Gangemi, A., Montiel, E., Nikitina, N., Presutti, V., Villazon-Terrazas, B.: D2. 5.2: pattern based ontology design: methodology and software support. Tech. rep., NeOn Project. IST-2005-027595. (2007)

[13] Daga, E., Meroño-Peñuela, A., Motta, E.: Modelling and querying lists in rdf. a pragmatic study. In: ISWC Workshops: QuWeDa. pp. In–Press (2019)

[14] Daga, E., Meroño-Peñuela, A.: Software and data for the experiments (Sep 2020). , https://doi.org/10.5281/zenodo.4030439

[15] Daga, E., Panziera, L., Pedrinaci, C.: A basilar approach for building web apis on top of sparql endpoints. In: CEUR Workshop Proceedings. vol. 1359, pp. 22–32 (2015)

[16] Dodds, L., Davis, I.: Linked data patterns. Online: http://patterns. dataincubator. org/book (2011)

[17] Eilbeck, K., et al: The sequence ontology: a tool for the unification of genome annotations. Genome biology **6**(5) (2005)

[18] Gangemi, A.: Ontology Design Patterns for Semantic Web Content. In: The Semantic Web – ISWC. Springer (2005)

[19] Görlitz, O., Thimm, M., Staab, S.: Splodge: Systematic generation of sparql benchmark queries for linked open data. In: International Semantic Web Conference. pp. 116–132. Springer (2012)

[20] Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics – Science, Services and Agents on the World Wide Web **3**(2), 158–182 (2005)

[21] Hobbs, J.R., Pan, F.: Time Ontology in OWL. W3C working draft **27**, 133 (2006)

[22] Ley, M.: The dblp computer science bibliography: Evolution, research issues, perspectives. In: International symposium on string processing and information retrieval. pp. 1–10. Springer (2002)

[23] Meroño-Peñuela, A., Hoekstra, R.: grlc Makes GitHub Taste Like Linked Data APIs. In: The Semantic Web – ESWC 2016 Satellite Events. pp. 342–353. Heraklion, Greece (2016)

[24] Meroño-Peñuela, A., Hoekstra, R.: The Song Remains The Same: Lossless Conversion and Streaming of MIDI to RDF and Back. In: The Semantic Web: ESWC Satellite Events (ESWC 2016). LNCS, vol. 9989, pp. 194–199. Springer (2016)

[25] Meroño-Peñuela, A., et al.: The MIDI Linked Data Cloud. In: The Semantic Web - ISWC 2017. vol. 10587, pp. 156–164 (2017)

[26] Meroño-Peñuela, A., Daga, E.: List.MID: A MIDI-Based Benchmark for Evaluating RDF Lists. In: The Semantic Web – ISWC 2019 (2019)

[27] Morsey, M., et al: Dbpedia sparql benchmark–performance assessment with real queries on real data. In: The Semantic Web – ISWC. Springer (2011)

[28] Pierce, B.C., Benjamin, C.: Types and programming languages. MIT press (2002)

[29] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: The Semantic Web - ISWC (2006)

[30] Saleem, M., et al: LSQ: Linked SPARQL Queries Dataset. In: The Semantic Web - ISWC 2015. LNCS, vol. 9367. Springer (2015)

[31] Schmidt, M., et al: SPˆ 2Bench: a SPARQL performance benchmark. In: Data Engineering, 2009. ICDE'09. IEEE (2009)

[32] Schmidt, M., et al: Foundations of sparql query optimization. In: 13th International Conference on Database Theory. ACM (2010)

[33] Thakker, D., et al: A pragmatic approach to semantic repositories benchmarking. In: Extended Semantic Web Conference. Springer (2010)

[34] The MIDI Manufacturers Association: MIDI 1.0 Detailed Specification. Tech. rep., Los Angeles, CA (1996-2014), https://www.midi.org/specifications

[35] Vandenbussche, P.Y., Atemezing, G.A., Poveda-Villalón, M., Vatant, B.: Linked Open Vocabularies (LOV): a gateway to reusable semantic vocabularies on the Web. Semantic Web **8**(3), 437–452 (2017)

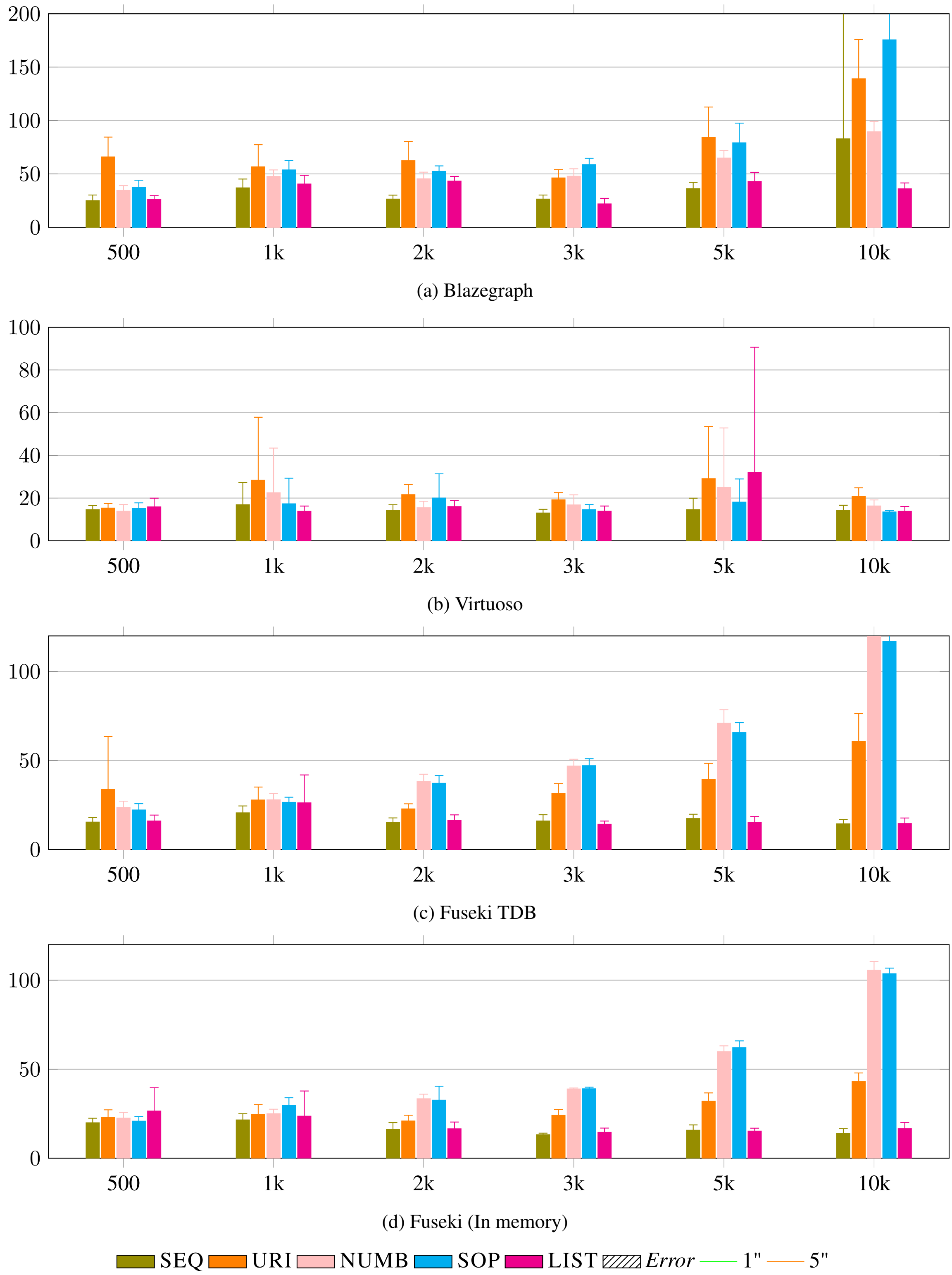
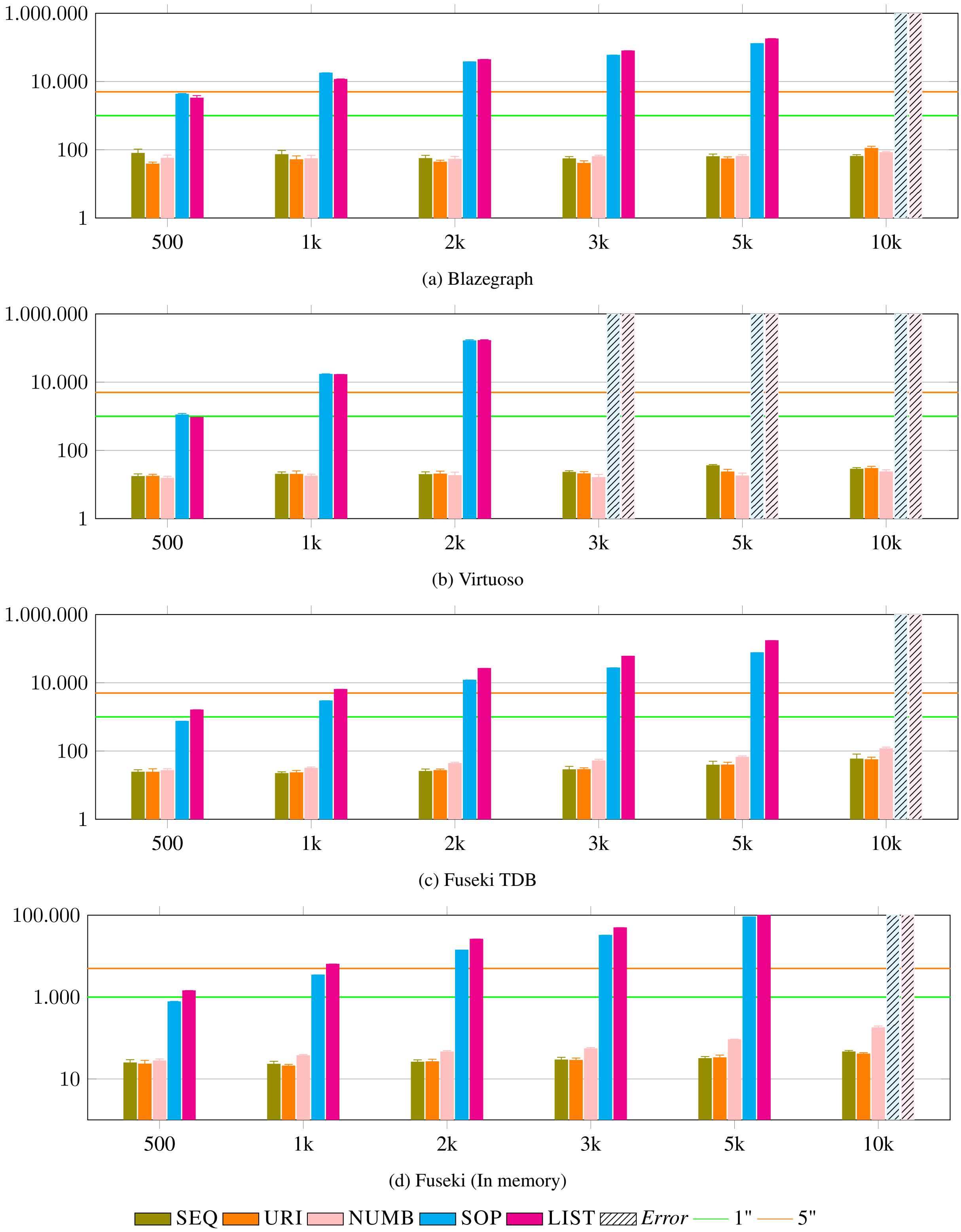


Fig. 11. **FIRST**. The operation is very efficient in all our experiments. Fluctuation of SD is not significant as the response is always returned in less than 200 milliseconds.



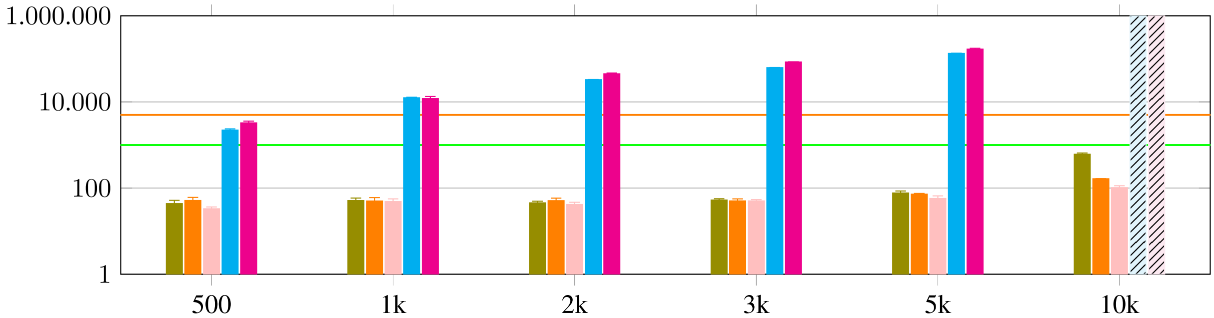
(a) Blazegraph

(b) Virtuoso

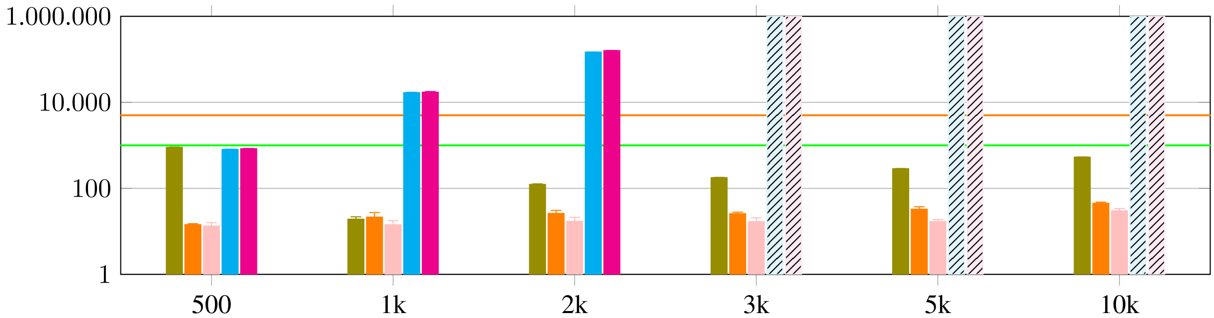
(c) Fuseki TDB

(d) Fuseki (In memory)

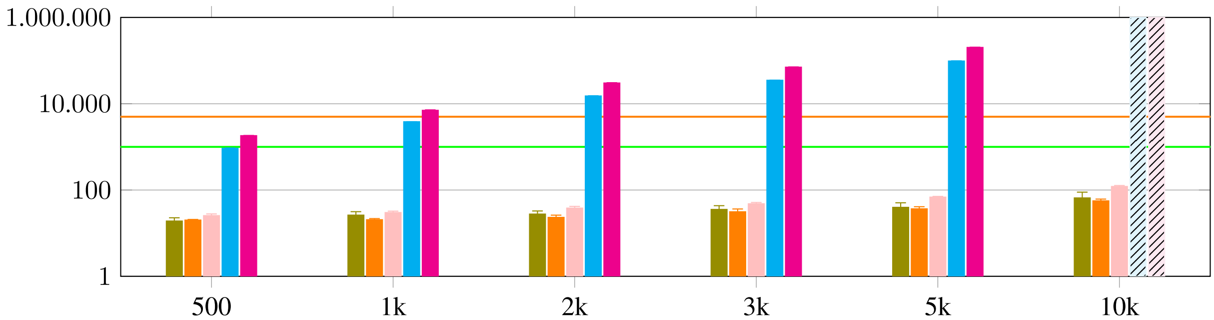
Fig. 12. **GET (low index)**. The operation scales well for all models with a materialised index (URI, NUMB, and SEQ), and it is problematic with LIST and SOP. The errors produced by Fuseki (Figures 12c and 12d) are Java Stack Overflow errors, while the others refer to the client waiting for more then five minutes. (Results are equivalent to the ones of GET with the high index. See Figure 13.



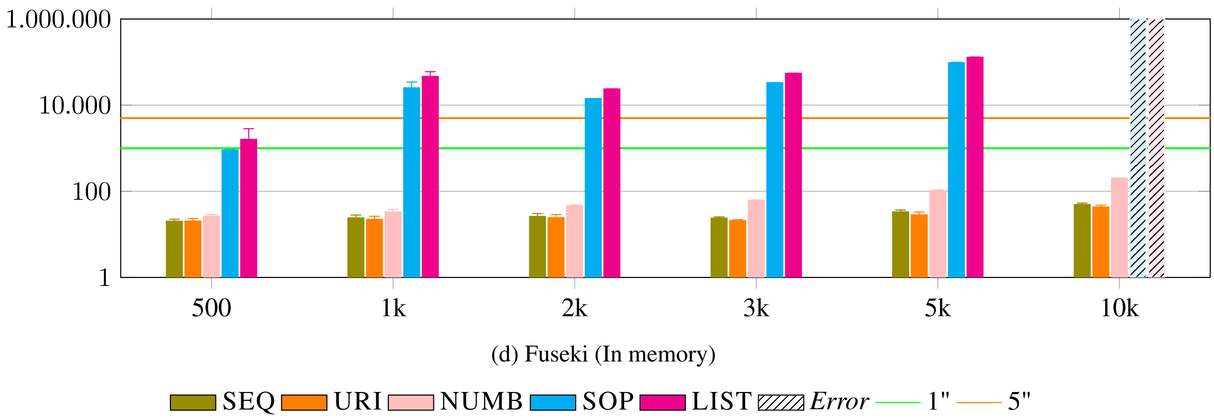
(a) Blazegraph



(b) Virtuoso



(c) Fuseki TDB



(d) Fuseki (In memory)

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 then five minutes. (Results are equivalent to the ones of GET with the low index. See Figure 12.)

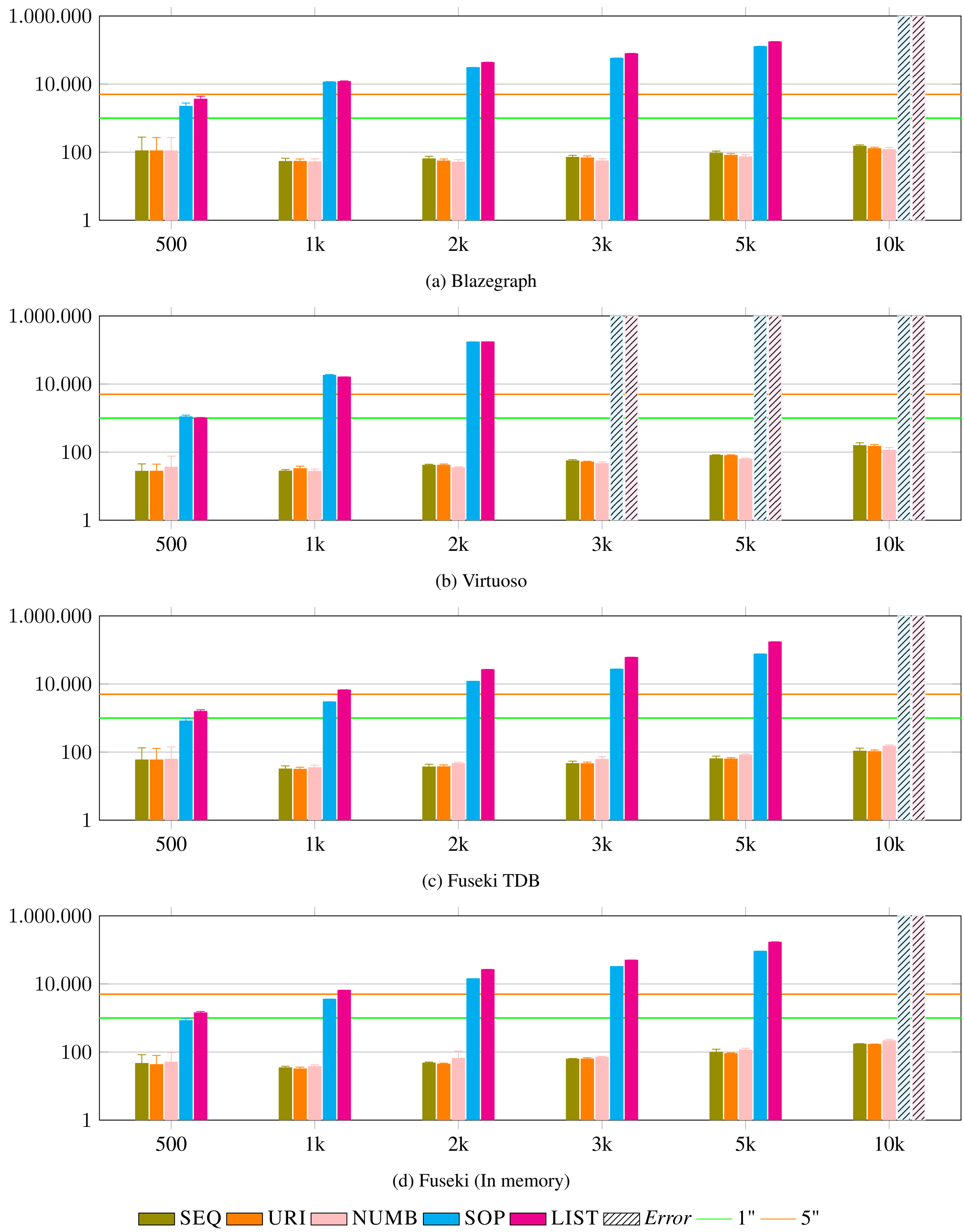
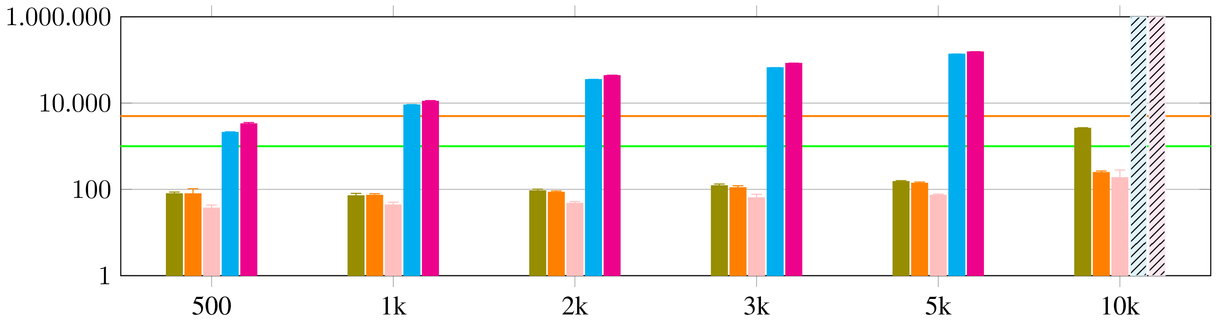
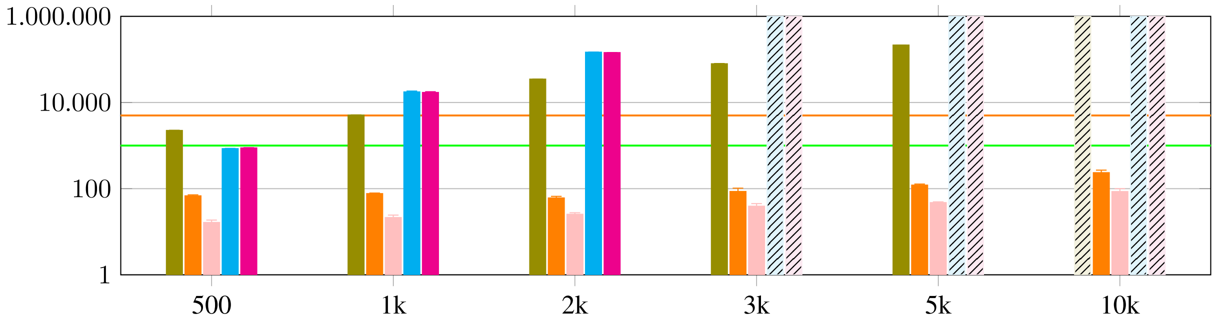


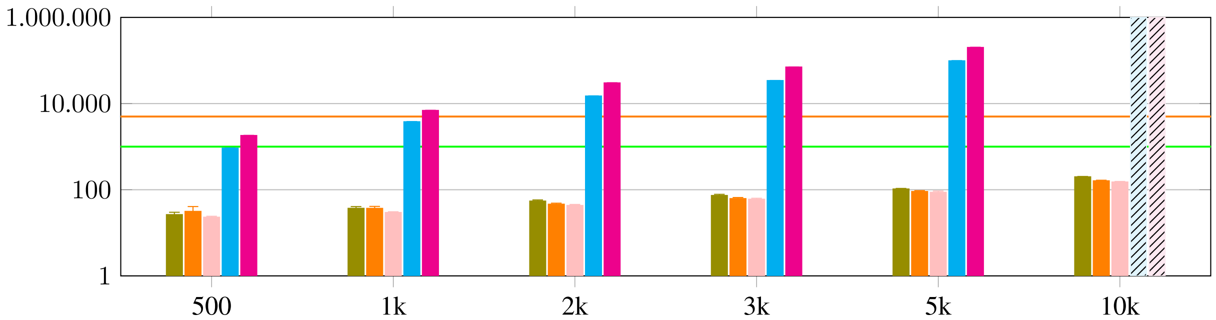
Fig. 14. **REST**. SOP and LIST perform poorly as the databases require to traverse the graph to retrieve all the elements and sort them in memory.



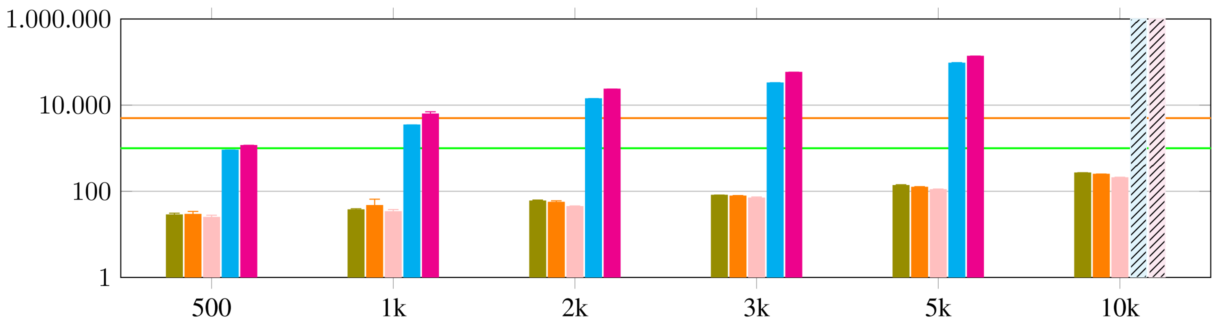
(a) Blazegraph



(b) Virtuoso



(c) Fuseki TDB



(d) Fuseki (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.

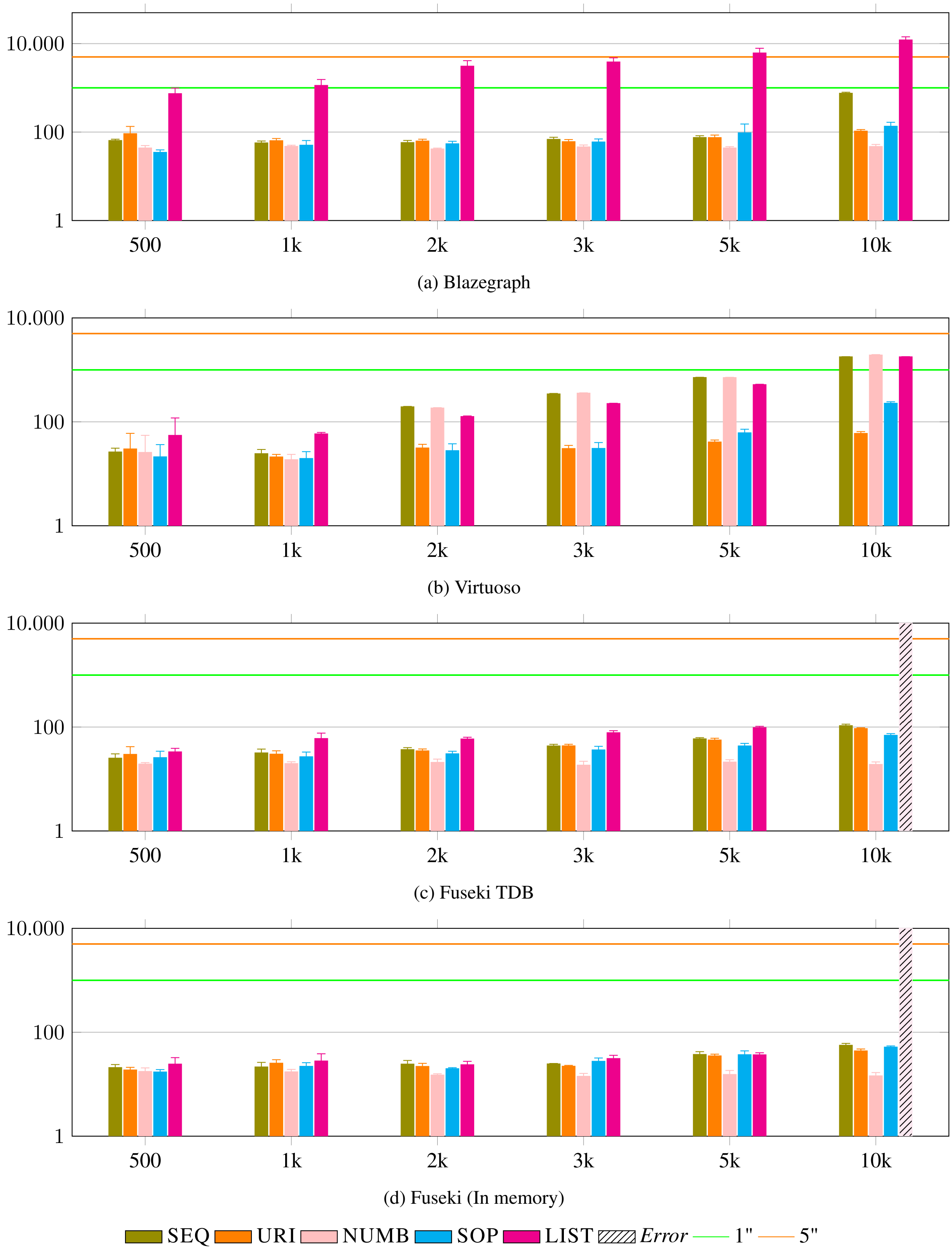


Fig. 16. **APPEND**. All data models perform reasonably well with small list sizes. In contrast with LIST, the SOP model has the advantage of directly linking all items to the container entity (the list) and, therefore, does not require to traverse the whole list. However, NUMB, SEQ, and URI perform generally better.

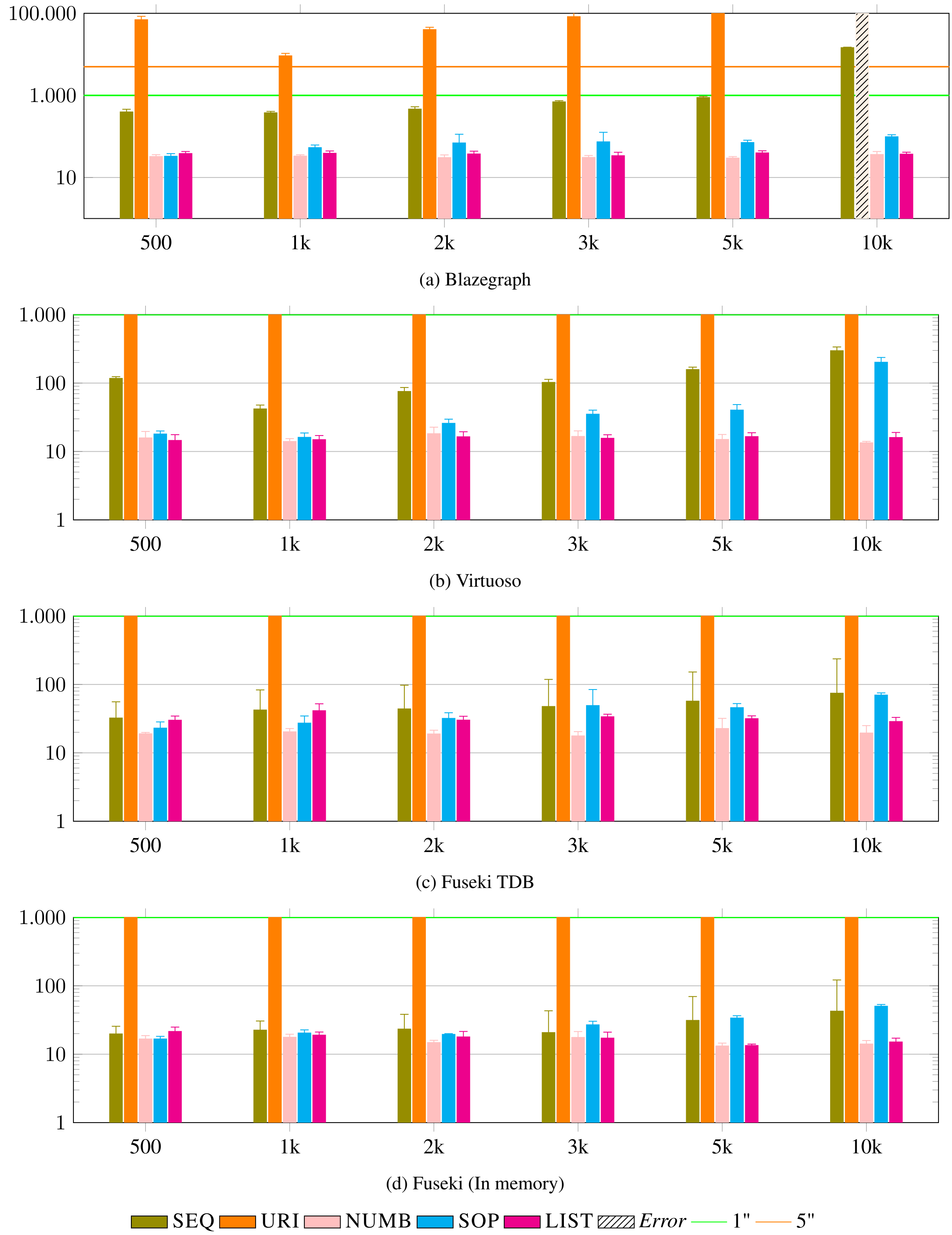
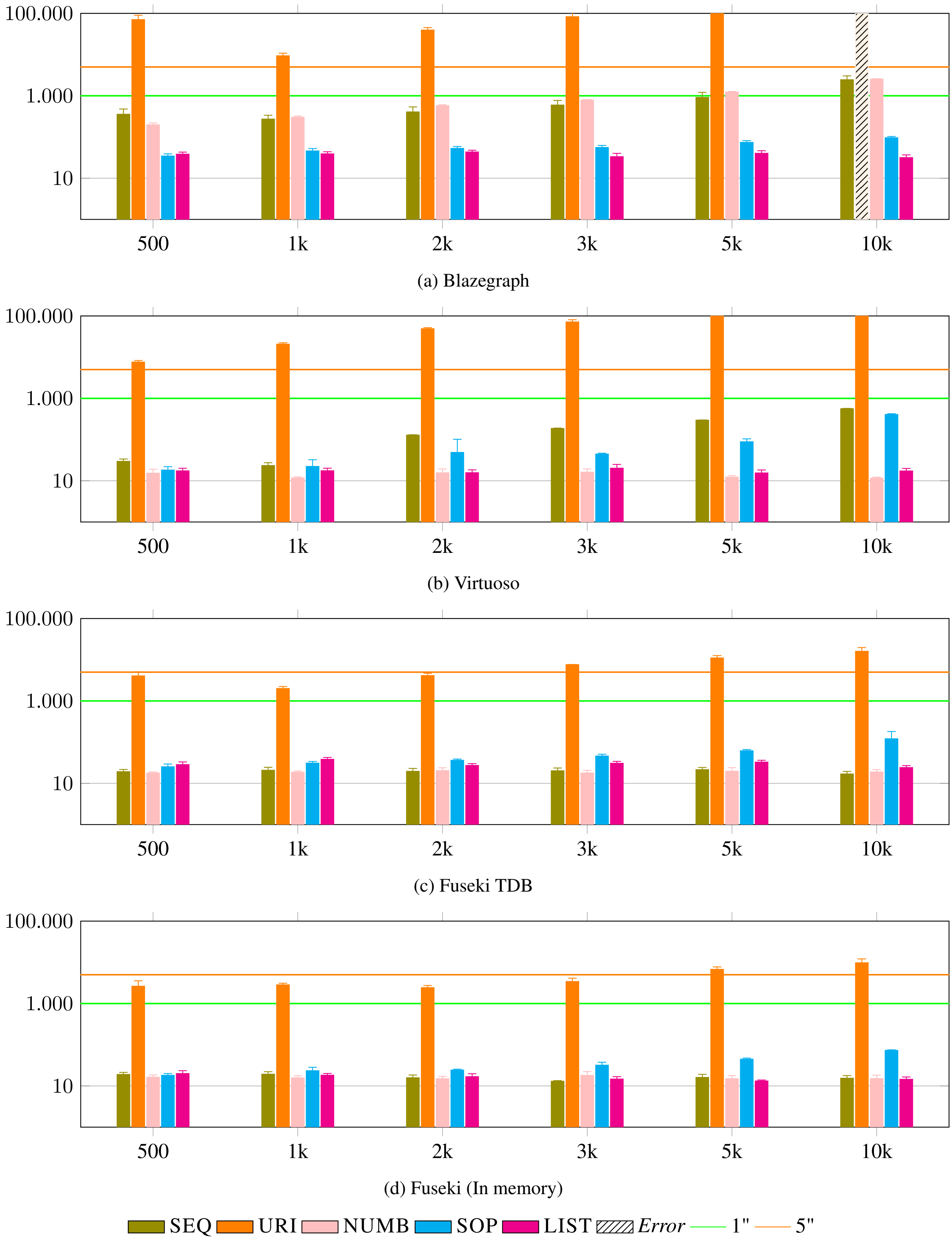


Fig. 17. **APPEND FRONT**. This operation is agile for both models based on linking items (SOP and LIST). In contrast, models based on materialised indexes require some refactoring of all the remaining items in the list! However, the index update seem very efficient in the case of NUMB.



(a) Blazegraph

(b) Virtuoso

(c) Fuseki TDB

(d) Fuseki (In memory)

Fig. 18. **POPOFF**. 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).

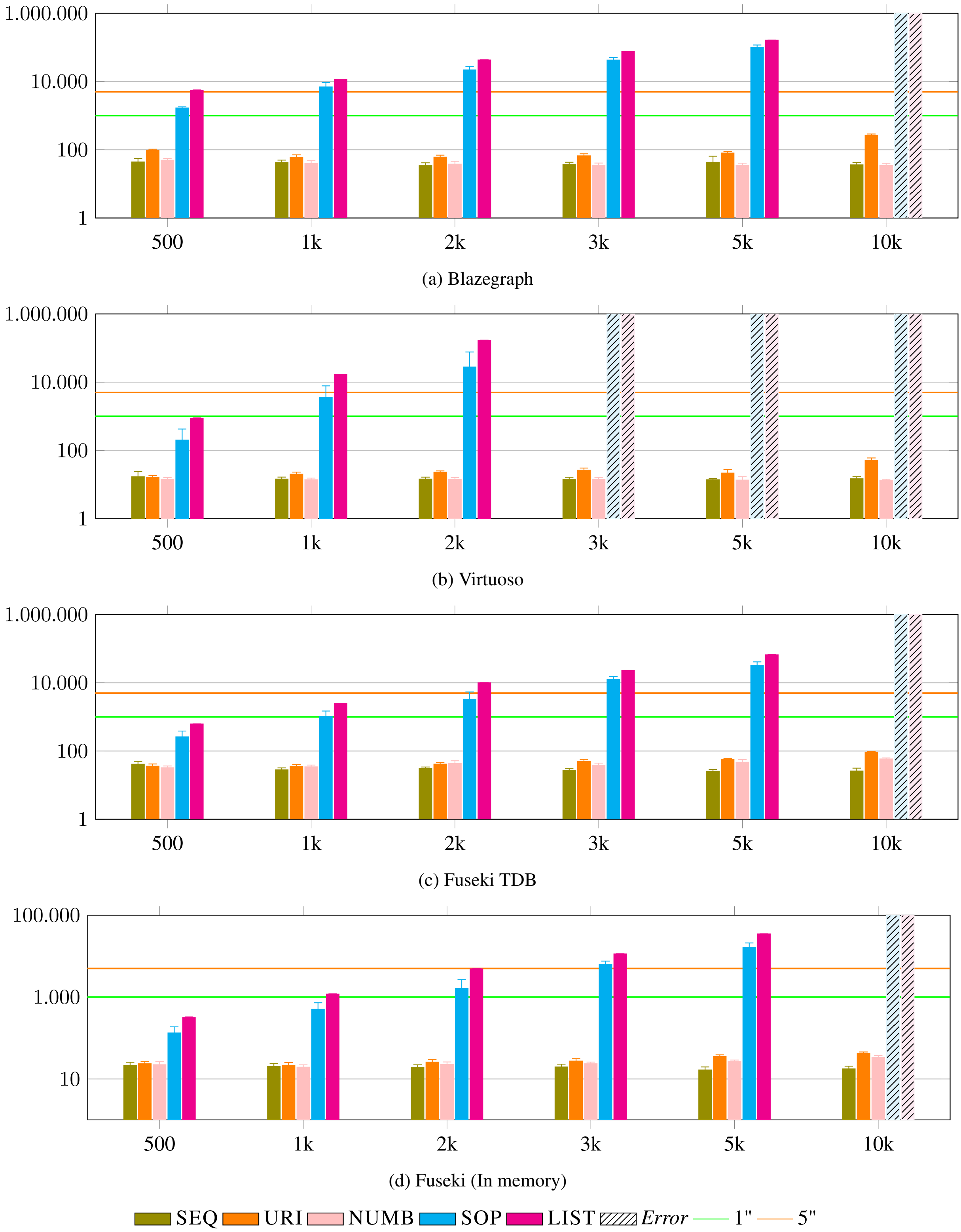


Fig. 19. **SET (low index)**. LIST and SOP suffer from the same shortcomings. NUMB, SEQ, and URI are more efficient data models. (Results are equivalent to the ones of SET with the high index. See Figure 20.)

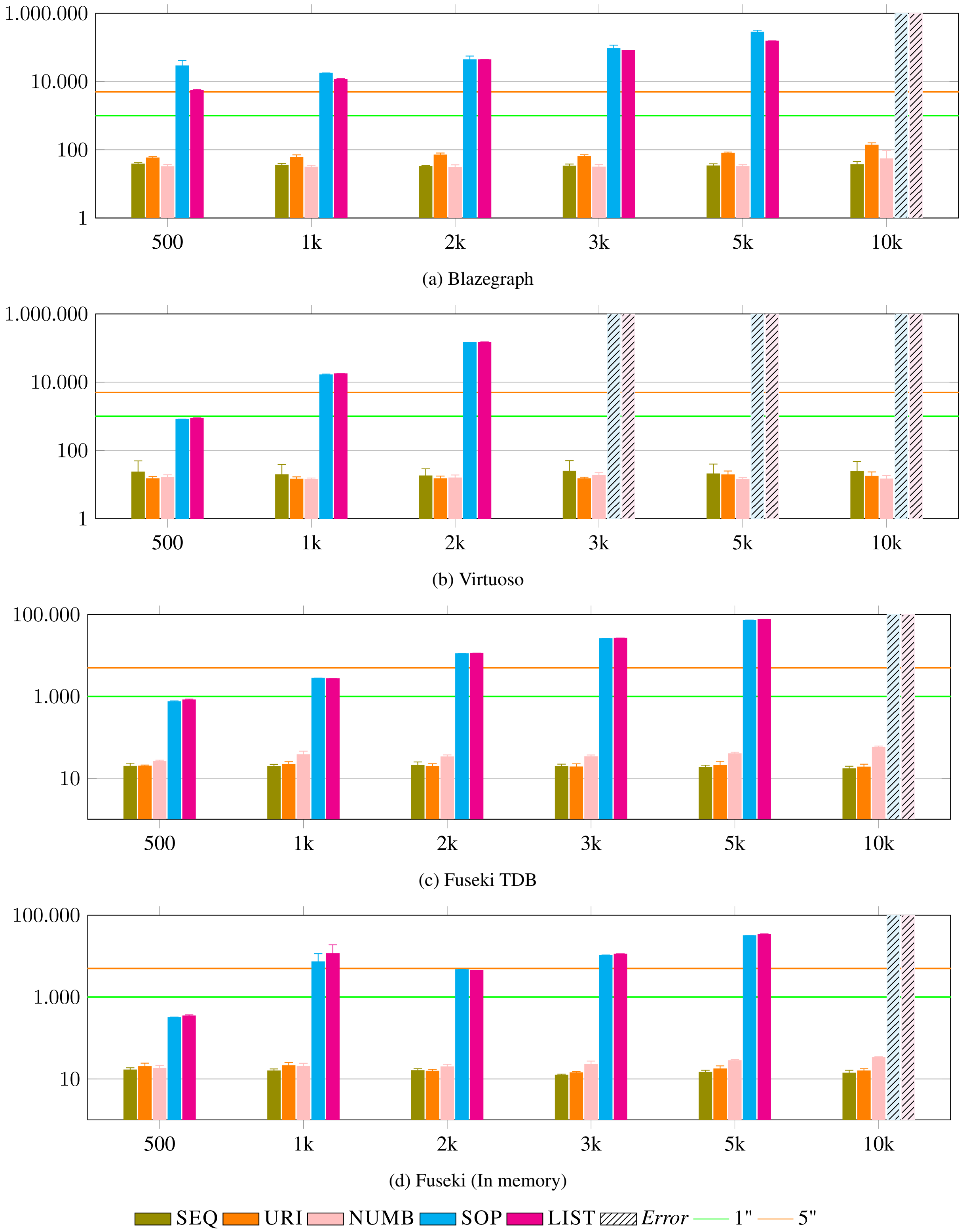


Fig. 20. **SET (high index)**. LIST and SOP suffer from the same shortcomings. NUMB, SEQ, and URI are more efficient data models. (Results are equivalent to the ones of GET with the low index. See Figure 19.)

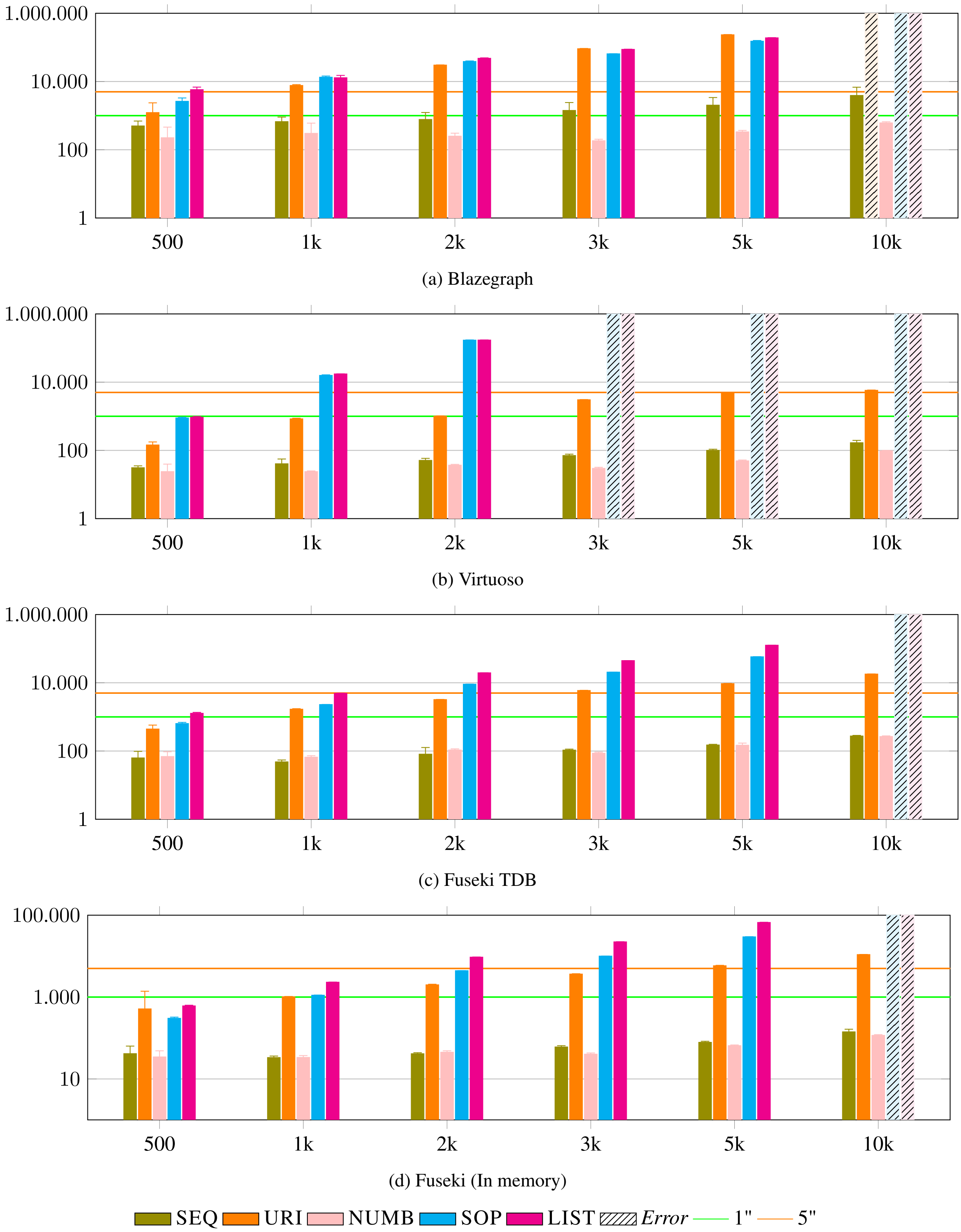


Fig. 21. **REMOVE AT (low index)**. This operation is the most expensive of all, as it requires to find the item in to be removed and shift all subsequent items, refactoring additional data, when appropriate. Results are generally similar to the ones of REMOVE_AT with the high index (See Figure 22). NUMB is the most efficient data model.)

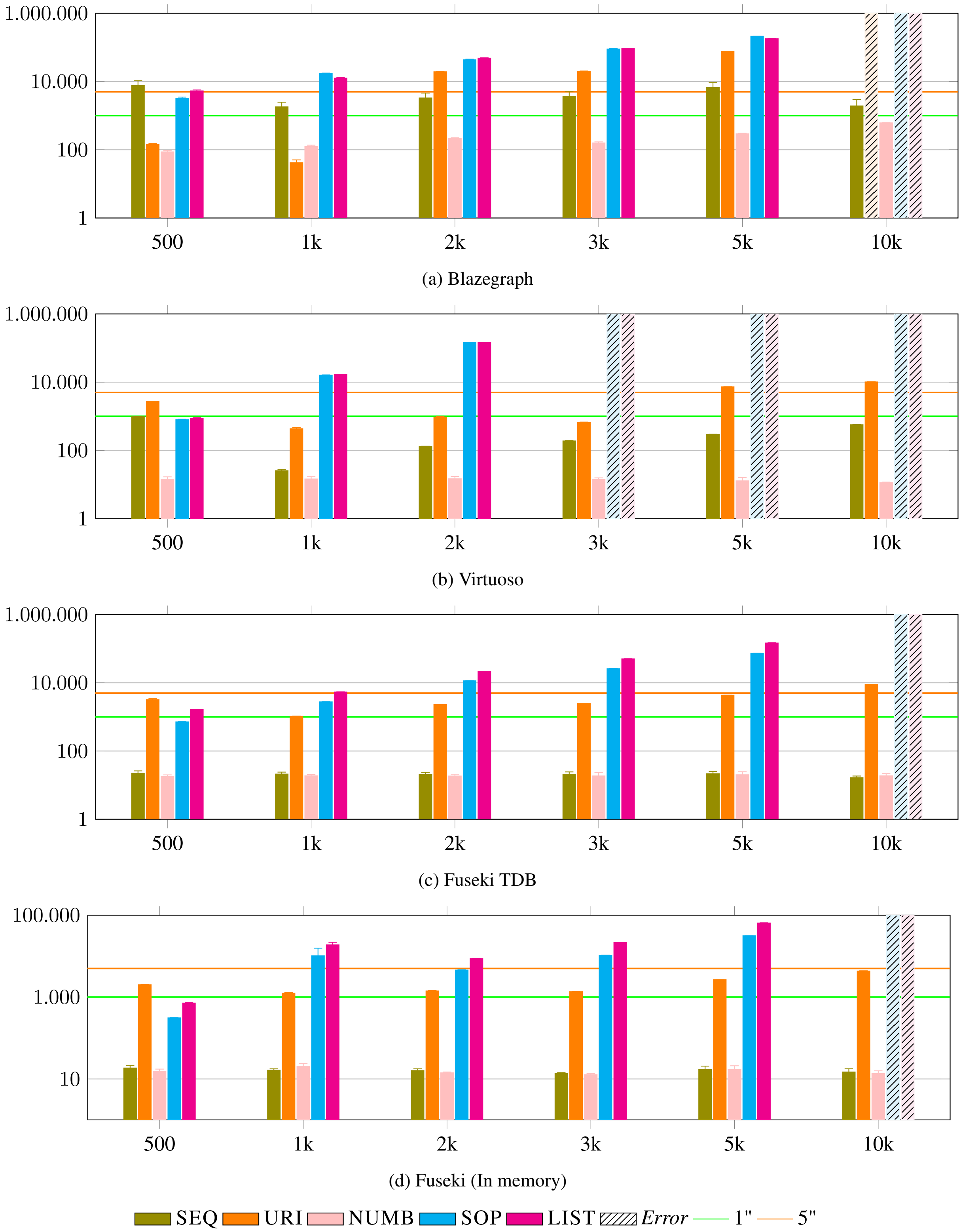


Fig. 22. **REMOVE AT (high index)**. This operation is the most expensive of all, as it requires to find the item in to be removed and shift all subsequent items, refactoring additional data, when appropriate. Results are similar to the ones of GET with the low index (See Figure 21), with the exception of SEQ. NUMB is the most efficient data model.