Sequential Linked Data: the state of affairs

Enrico Daga [a,*], and Albert Meroño-Peñuela [b]
[a] *Knowledge Media Institute, The Open University, United Kingdom*
E-mail: enrico.daga@open.ac.uk
[b] *Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands*
E-mail: albert.merono@vu.nl

Abstract. A sequence is a useful representation of many real-world phenomena, such as co-authors, recipes, timelines, and media. Consequently, 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 shown the sequence models that Knowledge Graphs commonly use; that these models have an impact in query performance; and that this impact is invariant to specific triplestore implementations. However, the specific list operations that management of Sequential Linked Data requires, beyond the simple retrieval of an entire list or a range of its elements, remain unclear. Besides, the impact of the different models in data management operations remains unexplored. Consequently, there is a knowledge gap on how to implement a real Semantic Web list Application Programming Interface (API) that enables standard list manipulation and generalizes beyond specific data models. In order to address these challenges, here we build on our previous work and propose a set of read-write Semantic Web list operations in SPARQL, towards the realization of such an API. 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 typical 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 engineered various list models across the years. For example, the Ordered List pattern [16], which refers to the `rdf:List` of W3C specifications. A pragmatic solution is referring to each member of the list with RDF containment membership properties (`rdf:_1`, `rdf:_2`,...) within an n-ary relation of type `rdf:Seq`. Another alternative option 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 reduced to five common representations: sequence, list, URI-

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

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

based, number-based, and sequence ontology pattern; and we have started a benchmark initiative to evaluate their querying performance in various dataset sizes and triplestore configurations. Other SPARQL-based benchmarks evaluate competing storage solutions against generic use cases, deemed to be representative of critical features of the query language [11] or to mirror how real users query linked data [29].

However, an important question remains: what methods a generic, SPARQL-based Application Programming Interface (API) should support? In other words, what are the standard, atomic read-write operations for the management of Sequential Linked Data? 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 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. 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 Semantic Web List API proposal consisting of nine atomic, read-write list operations in SPARQL: *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 4)
- An updated survey and catalogue on RDF list model (Section 5)
- We accordingly update our benchmark to evaluate them (Section 6)
- Experiments to evaluate the performance of these read-write API operations in SPARQL with competing List data models, on datasets of increasing sizes, and against four different triplestores (Section 7).

The rest of the paper is structured as follows. We survey the related work in Section 2. We introduce the research methodology in Section 3. We report our findings in atomic read-write list operations based on sequential data structures in Section 4. Sections 5 and 6 provide background on our survey and benchmark. Section 7 reports on the experiments. Results are discussed in Section 8, and we conclude our paper in Section 9. In what follows, we use these 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. 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. We consider research in two overlapping areas with our work: modelling of sequential RDF data; and performance of querying over such data using benchmark queries and datasets.

The Resource Description Framework (RDF) specification [32] and the RDF Schema (RDFS) recommendation [8] define container classes for representing collections. These containers are: `rdf:Bag` for containers of unordered elements; `rdf:Alt` for "alternative" containers whose typical processing will be to select one of its members; and `rdf:Seq` for containers

of elements whose order is indicated by the numerical order of the container membership properties. Additionally, [8] also defines a collection vocabulary to describe a closed collection, i.e. one that can have no more members, through the class `rdf:List` and the properties `rdf:first`, `rdf:rest`, and `rdf:nil`. In JSON-LD [33] ordered lists like `"@list": [ "joe", "bob", "jaybee" ]` have equivalent representations as `rdf:List` in RDF. Similarly, the RDF 1.1 Turtle [5] syntax allows for the specification of `rdf:List` instances, e.g. `:a :b ( "bob" "alice" "carol").` Apart from W3C standards, a number of ontology design patterns [18] have been proposed to represent sequences, e.g. the Sequence Ontology Pattern[2] (SOP) and the Collections Ontology [10] that focus on handling lists in OWL 2 DL, specifically. In our previous work [13] we propose a set of list modelling patterns that emerge from global Linked Data publishing. These patterns are used in a subsequent benchmark, `List.MID` [26], that we also apply and extend here.

We focus on practical approaches that assess querying sequential RDF data; for a theoretical study on the complexity of SPARQL, see [28]. 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) [7] 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^{2}Bench [30] 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) [29], 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 [11][3]. Pragmatic approaches to benchmarking are not new, and it is common practice to develop ad-hoc benchmarks to support specific applications (e.g. [34]). 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].

3. Methodology

In this section, we specialise the methodology previously introduced in [13] to pragmatically evaluating the performance of competing models for the representation of Sequential Linked Data with 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 systematically analyse classic computer science data structures, and study the operations they support in their definitions. To focus on lists, we restrict ourselves to *sequential* data structures, i.e. an element can only reference linearly following or preceding elements (this excludes data structures such as trees or graphs). The main objective is to identify the fundamental operations for sequential data access and

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

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

manipulation. By surveying classic computer science data models for sequences, we aim to collect all these operations, primarily the overlapping ones, to design such an API.

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 concerning 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 purest form, for example, adopting good practices for SPARQL query optimization [31]. 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[4]. We build upon our previous work [13, 26] to achieve this.

Evaluation The objective of this phase is to evaluate the different solutions empirically. Being the Linked Data standing on a Web application architecture (the client/server approach), the performance measure we focus on is *overall response time*. In order for results to be relevant to real applications, we measure 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, database engines may perform differently, or experiments results differ depending on the nature of the modelling solutions, showing a *trend* independently from the actual implementation. Here, we focus on the more complex 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.

[4] 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.

4. Sequential data structures

In this Section, we identify abstract data structures that are typically considered to represent ordered sequences. 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.

4.1. Linked List

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

$$\begin{aligned} append\_front &: E \times L \to L \\ first &: L \to E \\ rest &: L \to L \end{aligned} \tag{1}$$

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

where $append\_front$ is the operator that constructs memory objects which hold two values or pointers to values. It is implicit that $append\_front(e,l) \neq l$, $append\_front(e,l) \neq e$, $append\_front(e_1,l_1) = append\_front(e_2,l_2)$ if $e_1 = e_2$ and $l_1 = l_2$. Note that $first(nil())$ and $rest(nil())$ are not defined.

4.2. Double Linked List

A variant of a linked list 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:

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

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

4.3. 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 $append\_front$ and $popoff$. Often $first$ (or *top*) is available, too. Also known as "last-in, first-out" or LIFO. These operations, already introduced in the previous sections, in the case of Stack have a different axiomatic semantics, as follows.

$$\begin{aligned} popoff(append\_front(v,S)) &= S \\ first(append\_front(v,S)) &= v \end{aligned} \tag{3}$$

where S is a stack and v is a value.

4.4. 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:

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

where Q is a queue and v and w are values.

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

4.5. 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, 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. One implication of this is that removing one item may imply the shifting of others. The operations on Arrays are the following:

$$\begin{aligned} set &: (e \times i \times L) \rightarrow L \\ get &: (i \times L) \rightarrow e_i \\ remove\_at &: (i \times L) \rightarrow L \\ set(e,i,L) = L &\iff L_{i-1} \neq \varnothing \\ remove\_at(i,L) &= L \rightarrow \\ \forall E_n \in L : n > i &\rightarrow n = n-1 \end{aligned} \tag{5}$$

5. Survey of the RDF data models

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 se-

Table 1
Summary of operations and relevant abstract data structures.

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

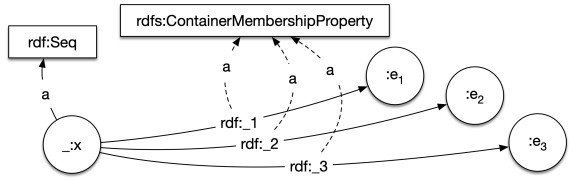


Fig. 1. The RDF Sequence model.

lecting them from the following sources, including W3C standards[5] ontology design patterns [18], resource track papers in the International Semantic Web Conference (e.g. [6], [25]), and lookups of relevant terms in Linked Open Vocabularies [36]. For a further detail and a description of the surveying methodology, see [13].

5.1. 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 processing 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 ("_").

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

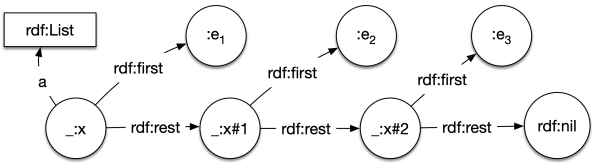


Fig. 2. The RDF List model.

5.2. RDF Lists

The RDFS recommendation [8] also 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 2).

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.

5.3. URI-based Lists

A more practical approach followed by many RDF datasets [6, 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> a <http://purl.org/ontology/bibo/Periodical> indicates that the subject belongs to a list of periodicals with list order 14809234; 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 3).

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.

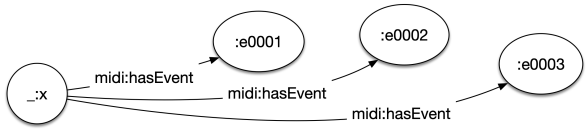


Fig. 3. The URI-based list model.

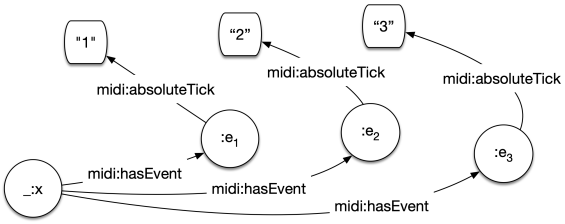


Fig. 4. The Number-based list model.

5.4. 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/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` indicates that the event has index 6 (see Figure 4).

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.

5.5. 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 (see Section **??**), 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 5).

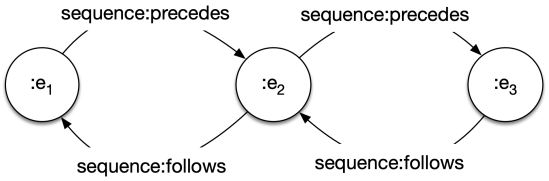


Fig. 5. The Sequence Ontology Pattern model.

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

6. Benchmark

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]. We introduce the following extensions:

- The `prop_number` order model can be set now to generate unique and sequential IDs for list elements, to avoid collisions
- The list of operations is extended to the atomic read-write Semantic Web list operations derived from Section 4
- New dataset sizes are included to generate lists of 500, 2k, 3k, 5k, and 10k elements

6.1. The `List.MID` benchmark

The first component of the List.MIDbenchmark is an algorithm to generate RDF datasets with lists according to the modeling patterns discussed above. The source code and all documentation are available on GitHub at https://github.com/midi-ld/List.MID.

[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

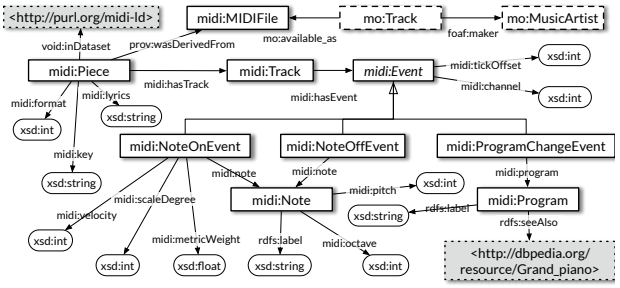


Fig. 6. Excerpt of the MIDI ontology. Tracks contain lists of sequential MIDI events.

The benchmark uses real-world data using MIDI files [35], 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 5 to encode lists of MIDI events.

Figure 6 shows an excerpt of the MIDI ontology used by the original `midi2rdf` algorithm. 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). For more details on MIDI event encoding see [24, 25, 35].

The original `midi2rdf` algorithm generates *implicit* lists of events by encoding their order in the URI of the event (e.g. `ex:track00/event02` happens immediately before `ex:track00/event01` and immediately after `ex:track00/event01`), and hence adhering to the *URI-based Lists* pattern discussed in Section 5. We extend this generation to the remaining patterns.

6.1.1. Usage

The first step is to find a MIDI file with the desired *list size*. The MIDI Linked Data cloud API[9] incorporates a query[10] to retrieve all track sizes in number of events in descending order from the dataset [25]. Since this query is expensive, we include a resulting dump in the benchmark. An inspection of this result allows users to select a MIDI identifier of the chosen size; this identifier can be used in a second query[11] to download an RDF dump for the MIDI file. This dump can be transformed into an input MIDI file with the included `rdf2midi` command [24].

Once the chosen input MIDI file has been generated, the `midi2rdf` CLI tool of the List.MIDbenchmark can be used to generate its RDF graph according to the requested list pattern. The syntax is:

```
midi2rdf [-h]
         [--format [{xml,n3,turtle,
                nt,pretty-xml,trix,
                trig,nquads,
                json-ld}]]
         [--gz] [--order [{uri,
                prop_number,
                prop_time,
                seq,list,sop}]]
         [--version]
         filename [outfile]
```

The relevant introduced argument is `--order`, which lets the user select the RDF list modeling to use for data generation. The mapping for the values of this argument with the patterns of Section 5 is: *RDF Sequences* → `seq`, *RDF Lists* → `list`, *URI-based Lists* → `uri`, *Number-based Lists* → `prop_number`, *Sequence Ontology Pattern* → `sop`. For example, to generate benchmark data of a preselected http://purl.org/midi-ld/pattern/bc7d9c25f81a4d90c000c30b6efc887d MIDI with 16,638 list elements using the RDF List pattern, we do:

```
midi2rdf
  --format turtle
  --order list
  bc7d9c25f81a4d90c000c30b6efc887d.mid
  benchmark.ttl
```

The output `benchmark.ttl` file is ready to be used in a standard compliant RDF store. As shown in the syntax above, the benchmark is agnostic with respect to serialization formats, and the most frequent (including JSON-LD) are supported.

[9] See http://grlc.io/api/midi-ld/queries/

[10] http://grlc.io/api/midi-ld/queries/#/default/get_events_count_per_track_piece

[11] http://grlc.io/api/midi-ld/queries/#/default/get_pattern_graph

6.2. Queries

In this section we propose a set of SPARQL query templates for retrieval of elements of lists, according to the patterns described in Section 5.

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 on Section 4. Specifically, these operations are:

- *FIRST*: returns the first element of the list
- *REST*: returns all the subsequent elements from the list from the current 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 5), we implement these operations as SPARQL query templates, considering the definitions and axiomatic semantics specified in Section 4. When the operation requires pointing to a specific item (GET, SET, and REMOVE_AT), we picked a random position in the second-half of the sequence and kept the same value in all experiments of the same list size. The values are reported in Table 2. The queries can be found online in the GitHub repository of the benchmark[12].

Size	Position
500	332
1k	657
2k	1222
3k	2472
5k	3789
10k	7322

Table 2

Position of the *n-th* item for benchmarking the operations GET, SET, and REMOVE_AT

7. Experiments

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, there will be a dataset with a list of size 500 implementing, for example, the *Seq* pattern, one of size 1k, and so on for each model type, 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), and the modelling solutions. We report statistics about the size of datasets in Table 3. 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[13].

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.

The client application performing the queries and measuring the response time resides on the same machine as the database, in order to avoid the impact of network bandwidth on the overall response time. It is worth reminding that the objective of the experiments is not the compare the various data management solutions but to compare the performance of the different modelling practices and their scalability with lists of

[12] See https://github.com/enridaga/list-benchmark/tree/master/queries

[13] 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.

Table 3
Datasets

Dataset	Triples
<data:500k-uri>	2025
<data:500k-seq>	2029
<data:500k-list>	2530
<data:500k-prop_number>	3025
<data:500k-sop>	3238
<data:1k-uri>	5944
<data:1k-seq>	5948
<data:1k-list>	6949
<data:1k-sop>	7942
<data:1k-prop_number>	7944
<data:2k-uri>	12011
<data:2k-seq>	12015
<data:2k-list>	14016
<data:2k-sop>	16009
<data:2k-prop_number>	16011
<data:3k-uri>	17989
<data:3k-seq>	17993
<data:3k-list>	20994
<data:3k-sop>	23987
<data:3k-prop_number>	23989
<data:5k-uri>	30015
<data:5k-seq>	30019
<data:5k-list>	35020
<data:5k-sop>	40013
<data:5k-prop_number>	40015
<data:10k-uri>	58642
<data:10k-seq>	58646
<data:10k-list>	68647
<data:10k-sop>	78640
<data:10k-prop_number>	78642

growing sizes. Experiments are executed on a Linux VM equipped with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz 8-core and 32G RAM. The details of the virtual machine used for hosting the experiments are the following:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             8
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:
        Intel(R) Xeon(R) CPU E5-2640
            v4 @ 2.40GHz
Stepping:              0
CPU MHz:               2399.998
BogoMIPS:              4799.99
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0-7
```

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) Query (one of): FIRST, GET, REST, PREV, APPEND, APPEND_FRONT, POPOFF, SET, REMOVE_AT (d) Database (one of): Virtuoso, Blazegraph, Fuseki-TDB, Fuseki-Mem.

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 repeated each experiment 10 times and report measures referring to average values. A timeout of 300 seconds has been set. We also analysed 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. Write operations (APPEND, APPEND_FRONT, POPOFF, SET, REMOVE_AT) reported a higher SD; in all cases below the 20%. 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. However, here we focus on the performance concerning client applications and not on studying resource consumption on the server-side.

Tables 6-14 report the average values response time. Figures 9-17 report on performance and scalability. Results are coherent for all the queries and datasets and demonstrate clear trends across different database engines. Supplementary material is available for reproducibility [14].

8. Mapping requirements and RDF modelling solutions

We discuss the results of the experiments by looking into each operation individually first. After that, we derive general conclusions with relation to the core data models introduced in the requirements section.

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
```

All modelling solutions seem to perform very well at the scales considered. Table 6 and Figure 9 display the values in milliseconds and the scalability. Fluctuation in the performance is not significant as the response is always returned in less than 200 milliseconds in all cases.

GET 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, PROP_NUMBER, 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 a midi:Track ;
        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 Table 7 and Figure 10. 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 10k sized lists). This result is significant as it impacts several operations that depend on retrieving items at specific positions.

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 a midi:Track ;
    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
```

Again, 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 a midi:Track ;
    midi:hasEvents ?top .
        ?top rdf:rest* ?mid  .
        ?mid rdf:rest* ?elt .
        ?elt rdf:first ?event .
        FILTER (?elt != ?top)
   }
   GROUP BY ?event
   ORDER BY ?i}}
}
```

Similarly, this harms the scalability of the approach (see Table 8 and Figure 11.

PREV This operation mirrors the previous one, aiming at retrieving 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 case 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 a midi:Track ;
  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.

APPEND This operation adds an element to the list. The operation requires either computing the index value (for models materializing indexes) or reaching the last item in the sequence. 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 a midi:Track ;
 midi:hasEvents ?events .
 ?events rdf:rest* ?elt .
 ?elt rdf:rest rdf:nil
}
```

With 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 a midi:Track ;
   midi:hasEvent ?event .
 FILTER NOT EXISTS {
   ?event sequence:precedes []
 }
}
```

This difference is reflected in the experiments results, in Table 10 and Figure 10.

APPEND_FRONT This operation is easy, and all models perform well, as in the case of FIRST.

POPOFF Removing the head of a list is an interesting operation as it 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. 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 a midi:Track ;
 midi:hasEvents ?events .
 ?events ?seq ?event .
 BIND (xsd:integer(SUBSTR(str(?seq), 45))
    AS ?index) .
 FILTER (?index > 1)
}
```

The impact of refactoring URIs is difficult to benchmark as it partly depends on how the index is inserted in the URI string. The solution implemented in the MIDI benchmark assumes a zero-padded string at the end of the URI. Embedding the index in URIs seems the least efficient option as it requires the rewriting of all triples associated with that entity! We report the resulting query in Figure 7.

SET This operation has a behaviour similar to GET. List and SOP suffer from the same shortcomings, as illustrated by Table 13 and Figure 16.

REMOVE_AT 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 Table 14 and Figure 17. The cost of the operation is on the side of materializing the index, for example, in case of PROP_NUMBER:

```
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> a midi:Track ;
        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 8).

Table 4 summarises the *fitness for use* of each surveyed RDF data model with relation to the sequential

```
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 a midi:Track ;
     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. 7. The SPARQL Update query for POPOFF + URI

Table 4

Performance of data models with relation to the operators. 5: very good, 1: very poor.

	SEQ	URI	P_N	SOP	LIST
FIRST	5	5	5	5	5
GET	5	5	5	1	1
REST	5	5	5	1	1
PREV	3	5	5	1	1
APPEND	5	5	5	5	3
APPEND_FRONT	4	4	5	5	5
POPOFF	5	3	5	5	5
SET	5	5	5	1	1
REMOVE_AT	3	3	4	1	1

data structures. Also, we summarized the results concerning the scalability of the data models for each operator, classifying each one of them in 5 Likert categories. Results are reported in Table 5. The major problem seems to be related to RDF models following the nested-tree approach. Pragmatically, they only perform well if the application requires a *stack*. Indeed, in order to retrieve pointers to the last item, we need to traverse the whole list. 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, materializing 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 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. Indeed, modelling practices have an impact on the performance and availability of sequential retrieval. Crucially, the behaviour of the various models is consis-

```
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 a midi:Track ;
                                 midi:hasEvent ?event .
                                 ?event sequence:follows* ?prev .
                }
                GROUP BY ?event
                ORDER BY ?i
                LIMIT 1
                OFFSET 23789
                }}
        }}
}
```

Fig. 8. The SPARQL Update query for the SOP + REMOVE_AT

Table 5

Mapping of abstract list data types with RDF data models.

Abstract Data Type	Operations	SEQ	URI	PROP_NUMBER	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

tent among different triple stores and allow us to distinguish design patterns that perform well in practice from others that perform worse —from the point of view of the identified requirements. The most efficient way of representing order is by using indexes in values like in *prop_number*.

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 circum-**

stances, their use should be discouraged for managing lists in RDF.

Finally, it is worth remarking how our evaluation of the data models was done with relation to *efficiency* of managing Linked Data, 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.

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 problem of typical, recommended solutions. A significant result lies in the fact that managing index as literal 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. Finally, we work towards the development of a full-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 07/02/2019

[2] "Queue", in Dictionary of Algorithms and Data Structures [online] (2019), https://www.nist.gov/dads/HTML/queue.html, accessed 07/02/2019

[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] Beckett, D., et alBeek: RDF 1.1 Turtle – Terse RDF Triple Language. Tech. rep., World Wide Web Consrotium (2014), https://www.w3.org/TR/turtle/

[6] 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)

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

[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] Ciccarese, P., Peroni, S.: The collections ontology: creating and handling collections in owl 2 dl frameworks. Semantic Web **5**(6), 515–529 (2014)

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

[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 (2020). , https://doi.org/10.5281/zenodo.3752887

[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] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: The Semantic Web - ISWC (2006)
[29] Saleem, M., et al: LSQ: Linked SPARQL Queries Dataset. In: The Semantic Web - ISWC 2015. LNCS, vol. 9367. Springer (2015)
[30] Schmidt, M., et al: SPˆ 2Bench: a SPARQL performance benchmark. In: Data Engineering, 2009. ICDE'09. IEEE (2009)
[31] Schmidt, M., et al: Foundations of sparql query optimization. In: 13th International Conference on Database Theory. ACM (2010)
[32] Schreiber, G., Raimond, Y.: RDF 1.1 Primer. Tech. rep., World Wide Web Consrotium (2014), https://www.w3.org/TR/rdf11-primer/
[33] Sporny, M., Kellogg, G., Lanthaler, M.: JSON-LD 1.0. Tech. rep., World Wide Web Consrotium (2014), https://www.w3.org/TR/2014/REC-json-ld-20140116/
[34] Thakker, D., et al: A pragmatic approach to semantic repositories benchmarking. In: Extended Semantic Web Conference. Springer (2010)
[35] The MIDI Manufacturers Association: MIDI 1.0 Detailed Specification. Tech. rep., Los Angeles, CA (1996-2014), https://www.midi.org/specifications
[36] 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)

Table 6
FIRST: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	24.90	36.90	26.40	26.40	36.20	82.80
sop	37.40	53.70	52.30	58.70	79.10	175.50
uri	65.90	56.60	62.30	46.20	84.30	139.10
prop_number	34.50	47.60	45.40	47.80	64.80	89.40
list	26.10	40.50	43.20	21.90	42.90	36.00

(b) Virtuoso

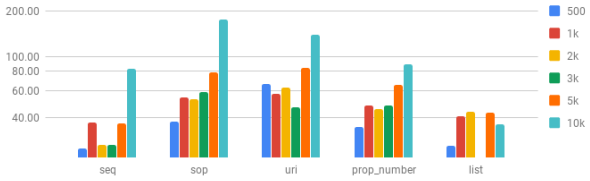
Model	500	1k	2k	3k	5k	10k
seq	14.60	16.90	14.20	13.00	14.60	14.10
sop	15.20	17.30	20.00	14.60	18.10	13.50
uri	15.30	28.40	21.60	19.20	29.10	20.80
prop_number	13.90	22.50	15.50	16.80	25.10	16.30
list	15.90	13.80	16.00	13.90	31.90	13.80

(c) Fuseki (TDB)

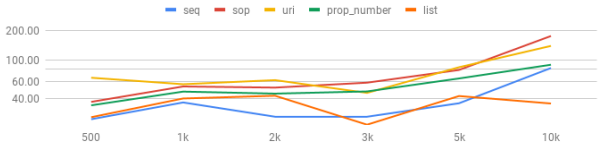
	500	1k	2k	3k	5k	10k
seq	15.40	20.60	15.20	16.00	17.40	14.40
sop	22.20	26.50	37.20	47.10	65.70	116.80
uri	33.70	27.80	22.80	31.40	39.40	60.70
prop_number	23.60	27.90	38.10	46.90	70.90	123.20
list	16.00	26.20	16.30	14.20	15.30	14.60

(d) Fuseki (Mem)

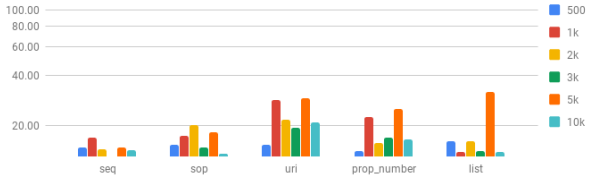
	500	1k	2k	3k	5k	10k
seq	19.90	21.50	16.20	13.20	15.70	13.90
sop	20.80	29.60	32.60	39.00	62.10	103.60
uri	22.90	24.60	20.90	24.20	32.00	43.00
prop_number	22.50	25.00	33.40	38.90	59.90	105.60
list	26.50	23.60	16.50	14.50	15.20	16.60



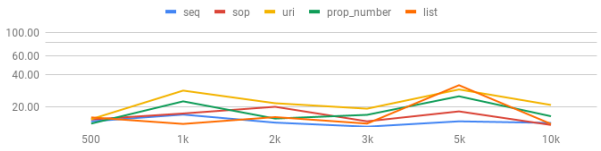
(a) Blazegraph (models)



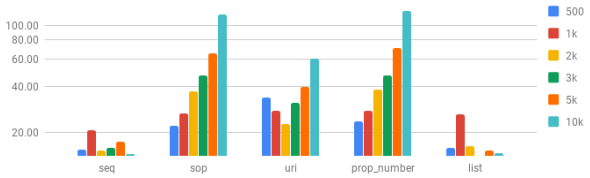
(b) Blazegraph (datasets)



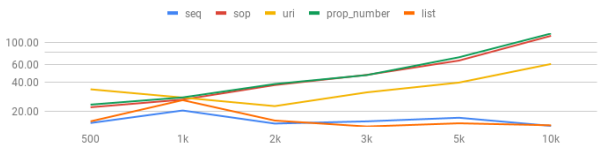
(c) Virtuoso (models)



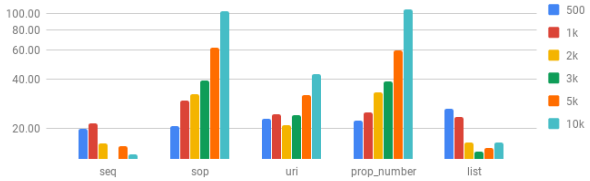
(d) Virtuoso (datasets)



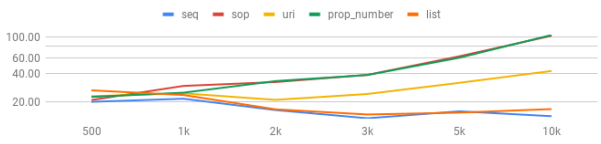
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 9. **FIRST**: Performance and scalability

Table 7

GET: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	43.70	51.40	45.40	52.70	76.80	156.7
sop	2213.80	12492.40	32941.00	62560.20	133663.50	E
uri	51.50	50.00	51.10	50.30	72.10	164.8
prop_number	33.10	48.70	41.60	50.30	57.40	121.6
list	3263.10	11976.20	44989.10	84454.00	167979.60	E

(b) Virtuoso

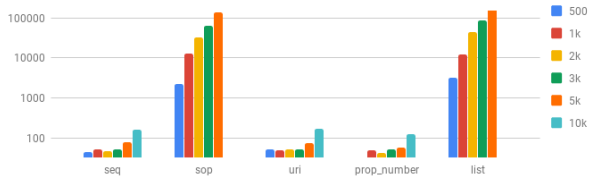
	500	1k	2k	3k	5k	10k
seq	897.00	19.20	123.30	177.00	285.70	531.20
sop	803.60	16788.60	146970.80	E	E	E
uri	14.30	21.40	26.10	25.70	32.80	44.90
prop_number	13.30	14.20	17.10	16.80	16.90	30.10
list	834.00	17183.60	159166.70	E	E	E

(c) Fuseki (TDB)

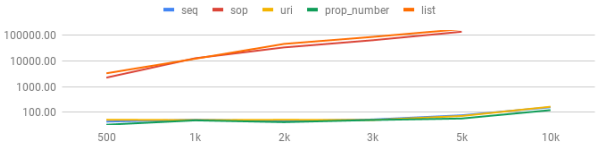
	500	1k	2k	3k	5k	10k
seq	19.10	26.20	27.70	35.50	39.70	65.60
sop	925.10	3822.30	15129.20	35024.90	97671.10	E
uri	20.10	20.50	23.20	31.30	36.60	56.00
prop_number	25.50	30.00	38.10	48.00	68.50	120.80
list	1819.10	7036.20	30232.40	70424.40	201876.20	E

(d) Fuseki (Mem)

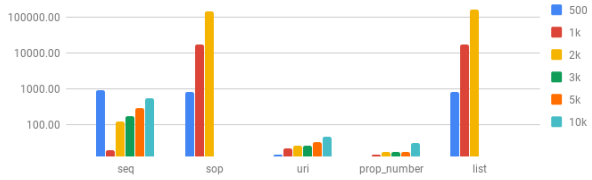
	500	1k	2k	3k	5k	10k
seq	20.20	24.20	26.10	23.80	33.30	49.20
sop	908.30	25283.30	14139.80	33207.20	96399.40	E
uri	20.40	22.30	24.40	20.90	28.40	43.10
prop_number	26.40	33.20	46.10	61.70	103.60	199.70
list	1608.20	46404.50	23774.30	54725.20	129584.80	E



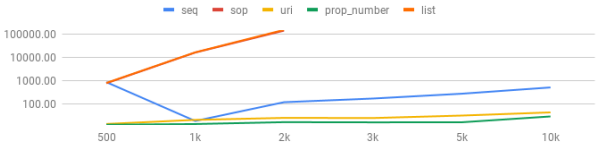
(a) Blazegraph (models)



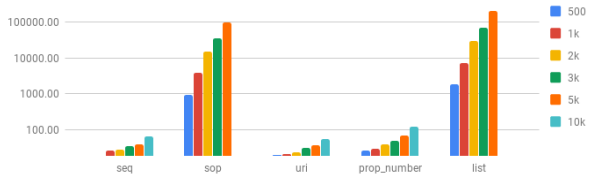
(b) Blazegraph (datasets)



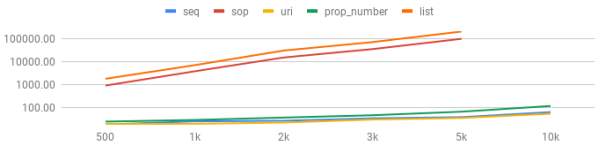
(c) Virtuoso (models)



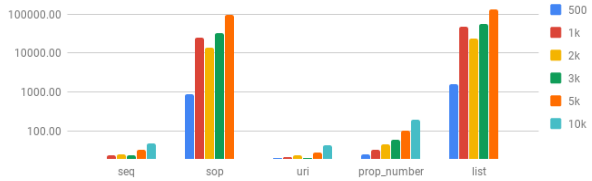
(d) Virtuoso (datasets)



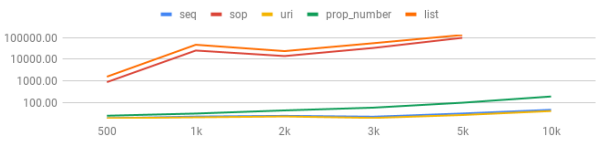
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 10. **GET**: Performance and scalability

Table 8

REST: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	56.30	50.30	59.60	77.90	110.30	210.12
sop	2103.90	12042.00	32384.50	61278.70	130310.50	E
uri	55.90	48.70	60.40	66.70	96.20	235.77
prop_number	38.80	47.50	52.50	71.20	83.30	176.66
list	3393.70	11573.00	44714.00	87013.70	166621.20	E

(b) Virtuoso

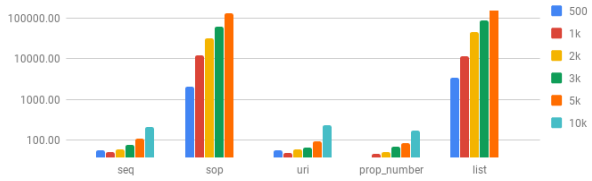
	500	1k	2k	3k	5k	10k
seq	89.54	32.30	152.60	225.80	361.40	703.00
sop	852.70	16586.00	146744.10	E	E	E
uri	23.40	32.40	49.80	62.90	97.00	179.20
prop_number	20.20	26.50	34.70	43.90	65.80	118.40
list	891.80	17140.60	146361.90	E	E	E

(c) Fuseki (TDB)

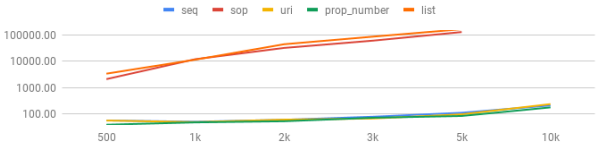
	500	1k	2k	3k	5k	10k
seq	23.30	31.70	40.20	53.30	76.60	139.70
sop	927.70	3821.40	15173.90	34814.20	98412.60	E
uri	27.20	33.30	33.70	47.90	64.10	110.60
prop_number	24.30	32.70	48.00	61.40	91.70	167.00
list	1841.60	7099.20	30637.30	72646.00	203595.80	E

(d) Fuseki (Mem)

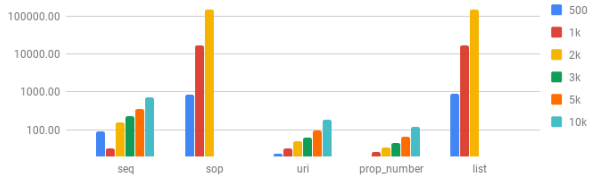
	500	1k	2k	3k	5k	10k
seq	23.80	35.10	46.10	65.50	107.40	213.80
sop	892.90	21599.80	14212.60	33000.00	97940.80	E
uri	26.10	34.70	49.00	61.10	101.70	200.30
prop_number	24.40	36.80	56.10	76.50	128.30	251.90
list	1179.60	38955.70	24048.30	57729.90	138462.70	E



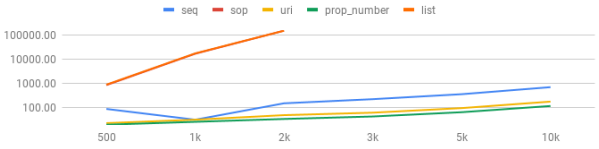
(a) Blazegraph (models)



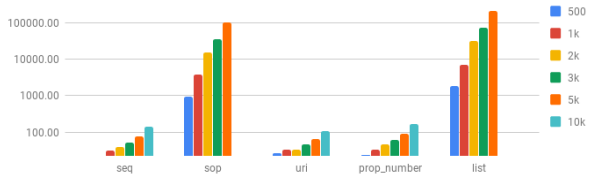
(b) Blazegraph (datasets)



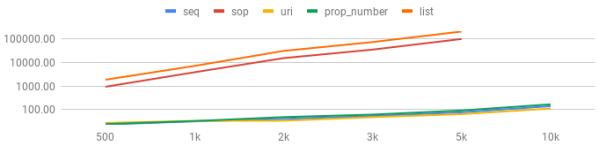
(c) Virtuoso (models)



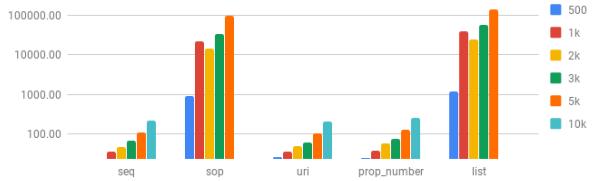
(d) Virtuoso (datasets)



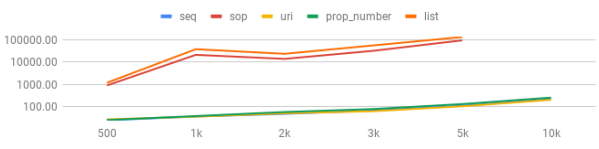
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 11. **REST**: Performance and scalability

Table 9

PREV: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	79.00	70.90	92.30	121.10	151.60	2605.00
sop	2086.50	9010.00	34912.50	65806.60	135775.30	E
uri	78.80	72.80	85.60	109.10	138.60	245.20
prop_number	36.70	43.40	47.30	63.60	72.80	186.80
list	3296.70	10981.60	43160.00	82901.60	153101.70	E

(b) Virtuoso

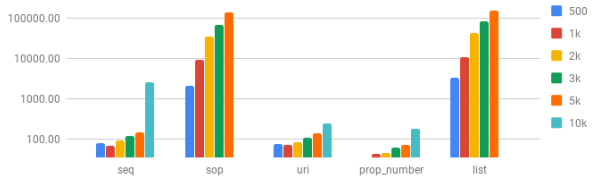
	500	1k	2k	3k	5k	10k
seq	2243.30	5114.60	34866.40	79253.80	215425.70	E
sop	849.20	17612.40	147105.20	E	E	E
uri	68.10	76.70	60.80	85.90	120.70	234.80
prop_number	16.40	21.30	25.40	39.10	47.40	85.50
list	881.60	17097.40	143865.00	E	E	E

(c) Fuseki (TDB)

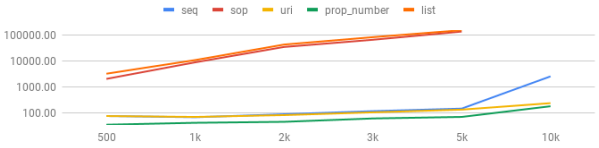
	500	1k	2k	3k	5k	10k
seq	26.40	37.00	54.50	73.50	104.10	199.40
sop	923.50	3784.60	14966.80	34206.80	98235.20	E
uri	31.40	36.90	46.10	62.20	90.80	161.00
prop_number	23.00	29.60	43.10	60.10	87.10	151.10
list	1814.40	6926.90	30165.80	70452.40	200824.90	E

(d) Fuseki (Mem)

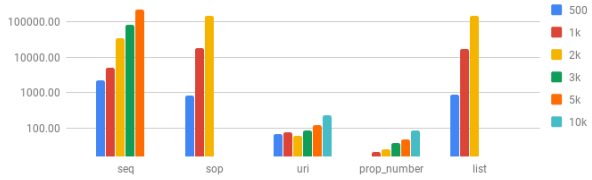
	500	1k	2k	3k	5k	10k
seq	28.30	37.10	59.50	80.80	135.90	266.20
sop	900.30	3457.50	14032.00	32710.40	94139.30	E
uri	29.10	46.70	55.50	77.40	123.90	248.20
prop_number	24.80	33.60	44.20	69.50	109.10	205.90
list	1153.10	6221.90	23491.50	57660.70	136128.30	E



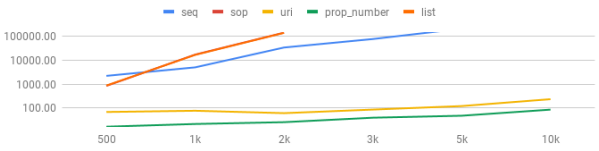
(a) Blazegraph (models)



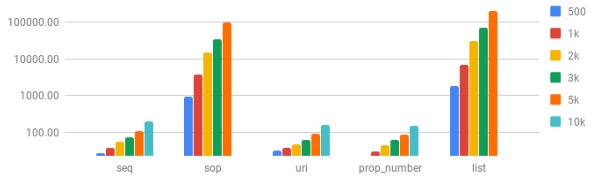
(b) Blazegraph (datasets)



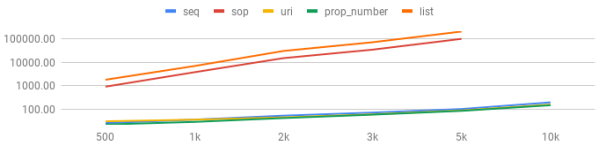
(c) Virtuoso (models)



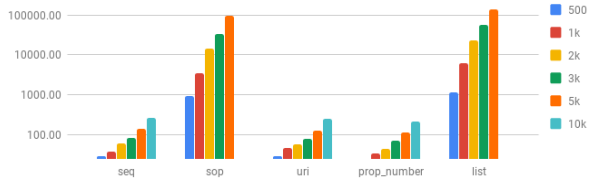
(d) Virtuoso (datasets)



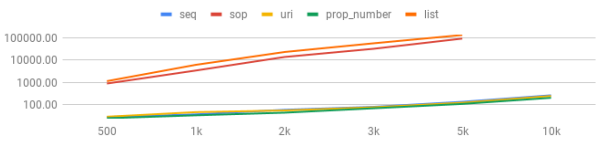
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 12. **PREV**: Performance and scalability

Table 10
APPEND: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	64.40	56.90	57.90	68.40	75.20	754.90
sop	34.80	50.50	54.40	59.70	96.10	135.60
uri	92.00	63.80	62.70	60.70	75.00	105.10
prop_number	43.70	47.50	41.50	46.20	43.70	47.20
list	739.60	1134.10	3091.50	3846.10	6143.20	12105.00

(b) Virtuoso

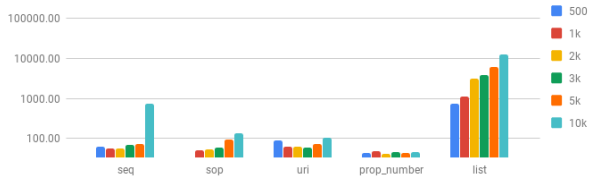
	500	1k	2k	3k	5k	10k
seq	26.30	24.40	194.60	344.60	712.40	1782.30
sop	21.20	19.70	27.90	30.80	61.50	227.70
uri	30.10	21.10	31.40	30.60	40.90	59.80
prop_number	25.80	18.70	184.50	356.30	710.60	1930.40
list	54.90	58.60	126.30	224.70	520.10	1782.90

(c) Fuseki (TDB)

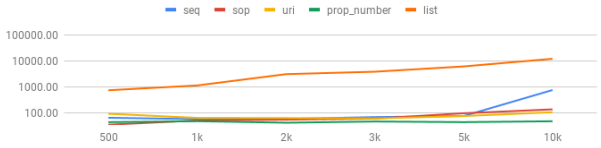
	500	1k	2k	3k	5k	10k
seq	25.20	31.80	36.80	43.30	59.30	106.30
sop	25.80	26.80	30.60	36.30	43.30	69.30
uri	29.80	30.10	34.80	43.50	56.30	93.40
prop_number	19.40	19.80	20.90	18.50	21.20	19.10
list	33.30	60.10	58.90	77.80	97.50	E

(d) Fuseki (Mem)

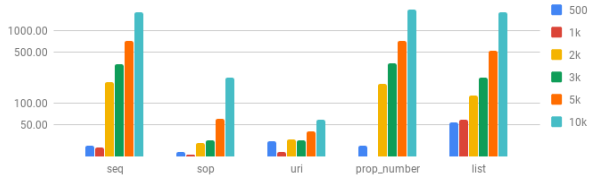
	500	1k	2k	3k	5k	10k
seq	20.90	21.50	24.40	24.60	37.40	56.20
sop	17.20	22.10	19.90	27.60	37.00	51.80
uri	18.80	25.40	22.00	22.20	35.10	43.80
prop_number	17.60	17.30	15.00	14.20	15.40	14.50
list	24.40	28.00	23.70	31.20	36.80	E



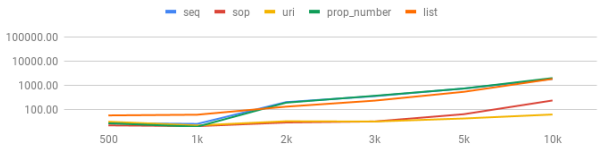
(a) Blazegraph (models)



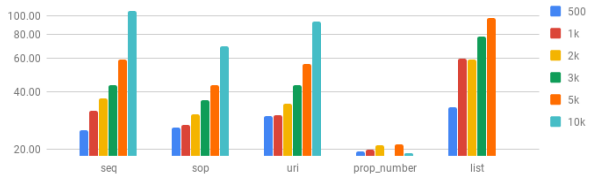
(b) Blazegraph (datasets)



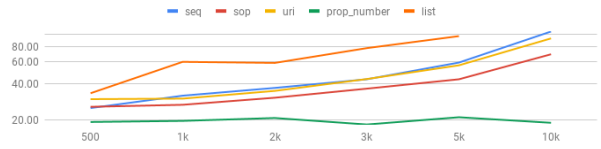
(c) Virtuoso (models)



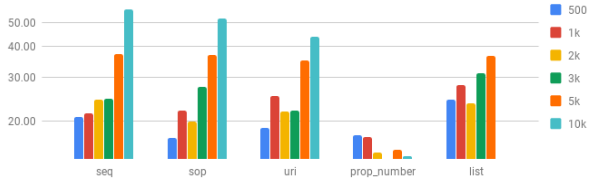
(d) Virtuoso (datasets)



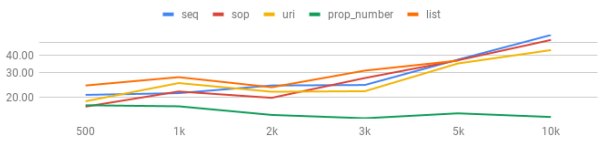
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 13. **APPEND**: Performance and scalability

Table 11
APPEND_FRONT: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	395.30	376.40	463.80	695.60	886.90	14564.40
sop	32.90	53.40	69.40	73.90	71.00	98.20
uri	602.10	1008.30	1862.90	2649.20	4823.90	9314.10
prop_number	32.40	33.20	30.50	30.80	29.70	36.30
list	38.30	38.90	37.40	33.90	39.70	37.00

(b) Virtuoso

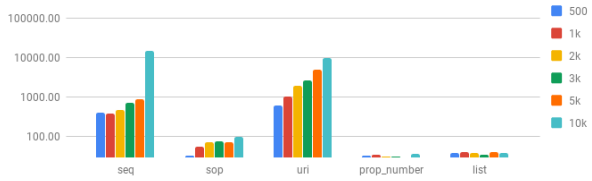
	500	1k	2k	3k	5k	10k
seq	117.10	41.90	75.40	102.30	157.60	297.40
sop	18.00	16.10	25.80	35.10	40.30	201.60
uri	108.50	52.90	64.80	84.70	129.00	204.90
prop_number	15.80	14.00	18.20	16.60	15.00	13.40
list	14.50	14.90	16.40	15.60	16.50	16.00

(c) Fuseki (TDB)

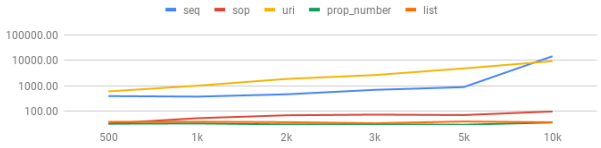
	500	1k	2k	3k	5k	10k
seq	32.30	42.30	43.90	47.60	57.00	74.50
sop	23.00	27.20	31.90	49.10	45.80	69.90
uri	40.40	60.20	76.30	104.70	151.30	267.00
prop_number	19.00	20.30	18.90	17.70	22.70	19.50
list	30.00	41.30	30.10	33.70	31.60	28.70

(d) Fuseki (Mem)

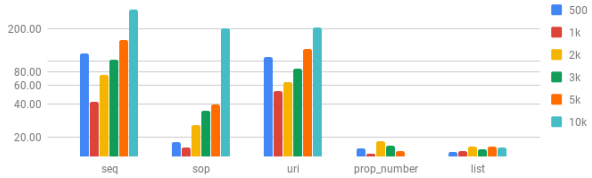
	500	1k	2k	3k	5k	10k
seq	19.80	22.50	23.30	20.70	31.20	42.60
sop	16.70	20.40	19.50	26.90	33.80	50.30
uri	25.00	37.90	39.00	49.60	83.70	130.40
prop_number	16.70	17.70	14.80	17.60	13.20	14.10
list	21.50	19.00	17.90	17.20	13.40	15.10



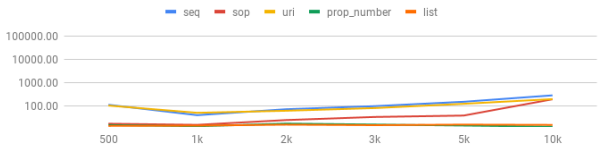
(a) Blazegraph (models)



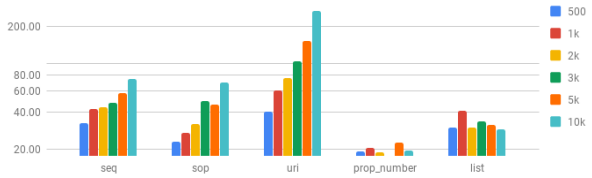
(b) Blazegraph (datasets)



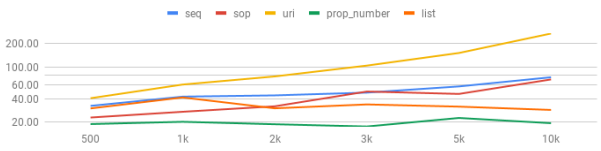
(c) Virtuoso (models)



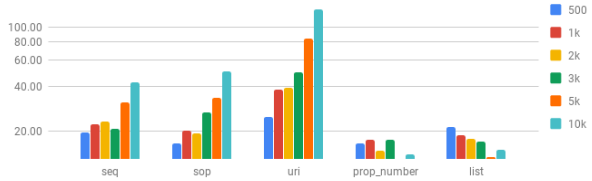
(d) Virtuoso (datasets)



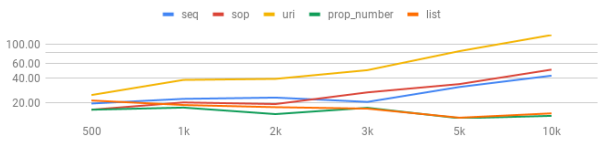
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 14. **APPEND_FRONT**: Performance and scalability

Table 12
POPOFF: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	356.00	272.30	406.30	595.40	914.20	2453.40
sop	34.50	45.90	52.90	55.60	74.00	95.80
uri	70664.10	9355.10	39383.70	83051.90	185348.00	E
prop_number	197.00	300.70	575.80	783.20	1233.30	2540.80
list	38.30	39.00	43.30	33.30	40.20	31.60

(b) Virtuoso

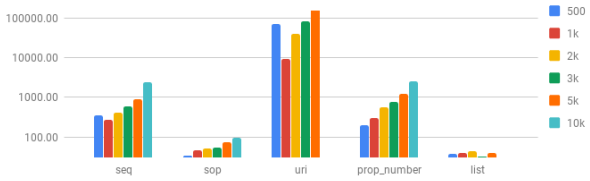
	500	1k	2k	3k	5k	10k
seq	29.60	23.40	128.30	185.60	296.10	561.80
sop	18.20	22.30	48.80	44.40	88.50	406.20
uri	7604.30	20684.80	48958.20	71571.10	158446.70	238475.90
prop_number	15.30	11.70	15.60	16.20	12.30	11.60
list	17.40	17.50	15.70	20.20	15.50	17.30

(c) Fuseki (TDB)

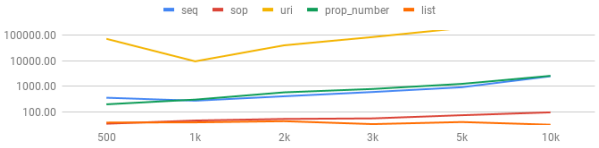
	500	1k	2k	3k	5k	10k
seq	19.20	20.80	19.60	20.20	21.50	16.90
sop	25.30	31.00	36.30	46.10	61.90	120.80
uri	4059.20	2005.70	4137.60	7623.80	11055.40	16026.00
prop_number	17.80	18.60	20.40	17.90	19.70	18.90
list	28.50	38.30	27.20	30.70	32.80	24.20

(d) Fuseki (Mem)

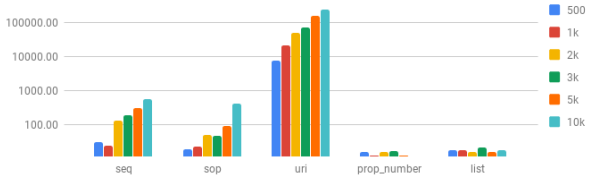
	500	1k	2k	3k	5k	10k
seq	19.00	19.30	15.80	13.00	16.10	15.40
sop	18.10	23.40	24.30	31.90	44.50	73.00
uri	2630.00	2843.23	2427.90	3420.70	6760.00	9748.00
prop_number	16.30	15.70	15.00	18.10	15.00	15.10
list	20.00	18.20	16.80	14.70	13.20	14.50



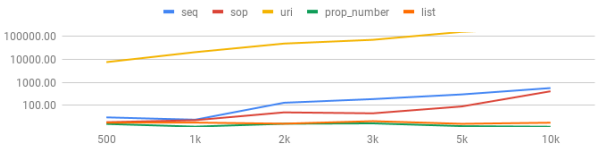
(a) Blazegraph (models)



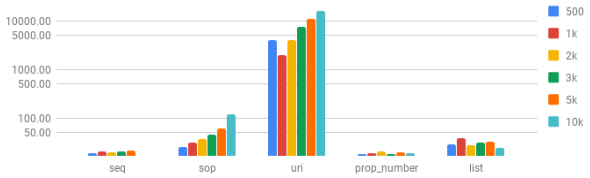
(b) Blazegraph (datasets)



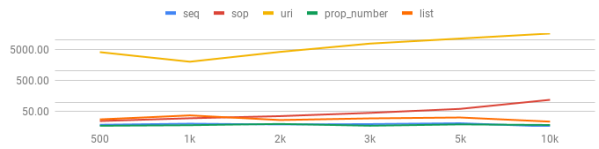
(c) Virtuoso (models)



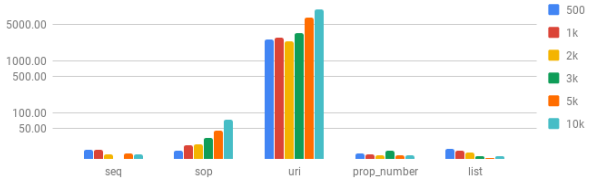
(d) Virtuoso (datasets)



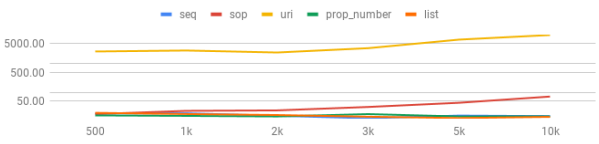
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 15. **APPEND_FRONT**: Performance and scalability

Table 13
SET: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	38.20	35.40	32.40	33.10	33.70	36.60
sop	28700.20	17533.20	43421.40	92375.12	281717.12	E
uri	57.70	59.80	70.20	63.60	79.20	136.20
prop_number	31.70	31.10	30.20	31.20	32.40	54.10
list	5402.00	11556.90	43382.30	80875.50	153111.10	E

(b) Virtuoso

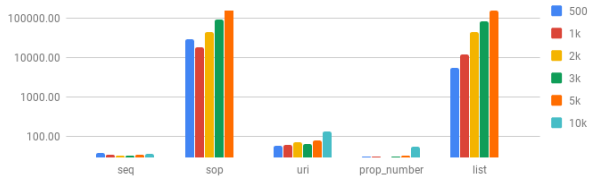
	500	1k	2k	3k	5k	10k
seq	23.30	19.40	17.90	24.50	20.50	23.90
sop	812.20	16397.50	147289.90	E	E	E
uri	14.70	14.40	14.70	14.70	19.20	17.40
prop_number	16.30	14.00	15.60	18.30	14.20	14.50
list	879.00	17585.90	148189.90	E	E	E

(c) Fuseki (TDB)

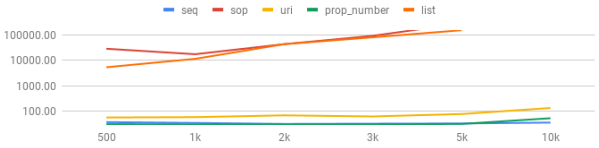
	500	1k	2k	3k	5k	10k
seq	19.70	19.50	21.00	19.50	18.40	17.10
sop	739.90	2778.30	11091.80	25874.30	72732.90	E
uri	19.90	21.90	19.30	19.00	21.00	18.90
prop_number	25.90	37.80	33.40	33.60	39.80	57.30
list	821.50	2706.60	11287.20	26321.40	76054.90	E

(d) Fuseki (Mem)

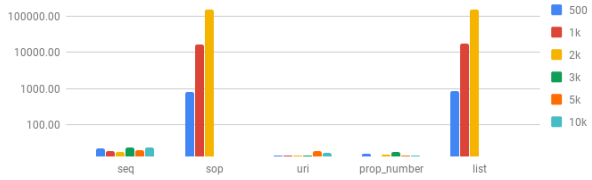
	500	1k	2k	3k	5k	10k
seq	16.60	15.70	16.00	12.40	14.50	13.90
sop	316.70	7272.60	4684.90	10506.40	31507.70	E
uri	20.10	21.00	15.50	14.10	17.60	15.70
prop_number	18.10	20.40	19.80	22.80	28.00	33.40
list	345.90	11482.70	4487.10	11191.80	33846.20	E



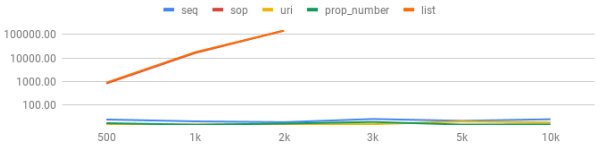
(a) Blazegraph (models)



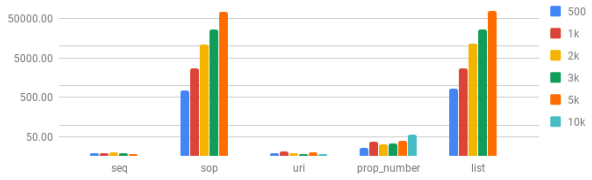
(b) Blazegraph (datasets)



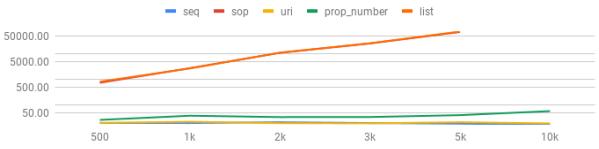
(c) Virtuoso (models)



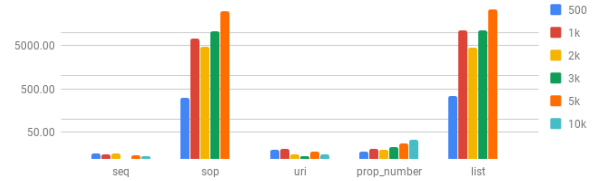
(d) Virtuoso (datasets)



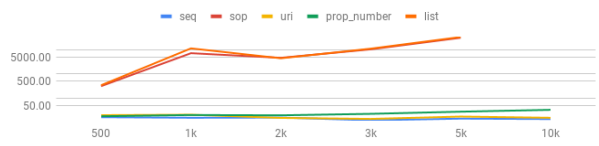
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 16. **SET**: Performance and scalability

Table 14
REMOVE_AT: Response Time (milliseconds)

(a) Blazegraph

	500	1k	2k	3k	5k	10k
seq	7609.50	1818.30	3292.60	3670.50	6695.80	1922.20
sop	3231.30	17452.20	43268.30	89654.65	212340.25	E
uri	143.20	41.60	19298.90	19969.20	77279.10	E
prop_number	86.60	124.50	216.20	158.60	294.10	613.50
list	5308.90	12583.60	48341.40	91485.80	180755.10	E

(b) Virtuoso

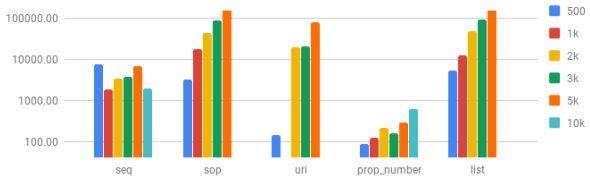
	500	1k	2k	3k	5k	10k
seq	968.40	25.40	130.20	189.20	294.50	566.50
sop	796.90	15993.20	146233.10	E	E	E
uri	2721.70	432.50	944.90	663.90	7312.30	10146.70
prop_number	14.20	14.50	14.70	14.00	12.80	11.30
list	879.30	16765.70	145936.40	E	E	E

(c) Fuseki (TDB)

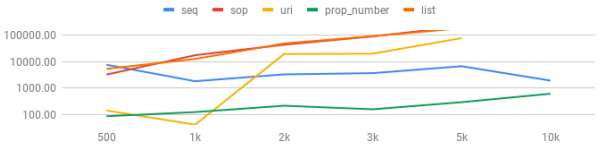
	500	1k	2k	3k	5k	10k
seq	22.20	21.10	20.50	20.90	21.60	16.50
sop	716.00	2757.60	11410.30	25974.00	72920.80	E
uri	3195.00	1039.80	2302.70	2453.30	4322.40	8883.90
prop_number	17.90	18.70	18.50	18.60	20.20	18.70
list	1629.60	5346.50	21630.30	50342.70	146528.80	E

(d) Fuseki (Mem)

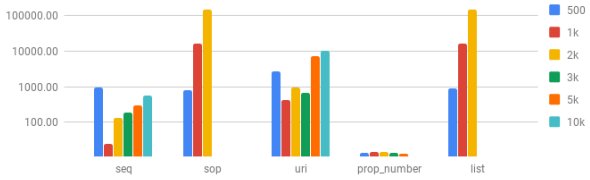
	500	1k	2k	3k	5k	10k
seq	18.50	16.30	16.00	13.60	16.90	14.80
sop	310.10	10209.30	4563.60	10471.20	31377.60	E
uri	2006.40	1245.40	1408.40	1352.00	2646.20	4325.21
prop_number	15.30	20.10	14.10	12.70	16.90	13.50
list	713.80	18898.60	8728.00	21462.90	64339.40	E



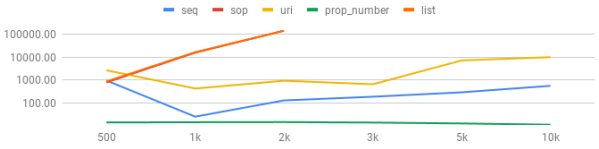
(a) Blazegraph (models)



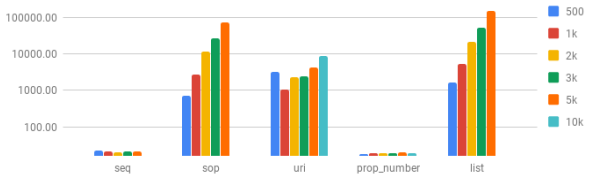
(b) Blazegraph (datasets)



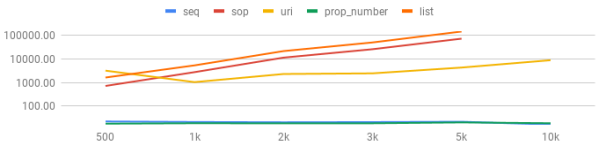
(c) Virtuoso (models)



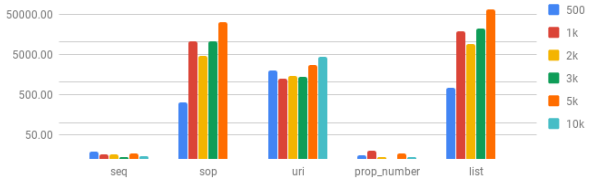
(d) Virtuoso (datasets)



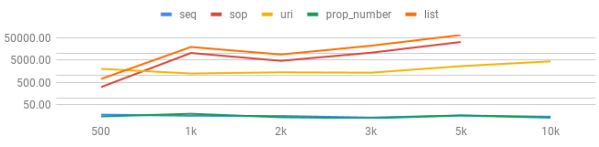
(e) Fuseki TDB (models)



(f) Fuseki TDB (datasets)



(g) Fuseki In Memory (models)



(h) Fuseki In Memory (datasets)

Fig. 17. **REMOVE_AT**: Performance and scalability