# Modelling and Querying Lists in RDF. A Pragmatic Study

Enrico Daga[1], Albert Meroño Peñuela[2], and Enrico Motta[1]

[1] Knowledge Media Institute, The Open University, UK
enrico.daga@open.ac.uk
[2] Dept. of Computer Science, Vrije Universiteit Amsterdam, NL
albert.merono@vu.nl

**Abstract.** Many Linked Data datasets model elements in their domains in the form of lists: a countable number of ordered resources. When publishing these lists in RDF, an important concern is making them easy to consume. Therefore, a well-known recommendation is to find an existing list modelling solution, and reuse it. However, a specific domain model can be implemented in different ways and vocabularies may provide alternative solutions. In this paper, we argue that a wrong decision could have a significant impact in terms of performance and, ultimately, the availability of the data. We take the case of RDF Lists and make the hypothesis that the efficiency of retrieving sequential linked data depends primarily on how they are modelled (triple-store invariance hypothesis). To demonstrate this, we survey different solutions for modelling sequences in RDF, and propose a pragmatic approach for assessing their impact on data availability. Finally, we derive good (and bad) practices on how to publish lists as linked open data. By doing this, we sketch the foundations of an empirical, task-oriented methodology for benchmarking linked data modelling solutions.

**Keywords:** Linked Open Data · RDF Lists · Benchmarking methodology · SPARQL Benchmark

## 1 Introduction

When publishing Linked Open Data a major concern is to make the data understandable and easy to consume [27]. Despite the extensive documentation around good practices for Linked Data publishing [20, 12], the decision about how to reuse a certain modelling practice is left to subjective choices of the data engineer. However, in Linked Open Data applications, ease of querying with SPARQL is crucial, particularly in relation to performance and its impact on service *availability* [7]. An interesting case are **RDF Lists**, a fundamental data structure crucial to support domain knowledge such as scholarly metadata (the order of authors), historical data (biographies and timelines), media metadata (track lists), social media content (recipes, *howto*) and musical content (e.g., scores as MIDI Linked Data [22]). Applications typically need to query for the

$n$-th element of a list —e.g. the first author of a paper, the 2nd track of an album— or to get the audio events between minutes 2.00 and 2.10. The Semantic Web community offers several options to the practitioner; for example, the Ordered List pattern [12], 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 a n-ary relation of type `rdf:Seq`. Another, alternative option, may involve picking a solution from the Ontology Design Patterns catalogue [10], for example, the Sequence ODP[3]. 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 this paper, we propose an empirical and task-oriented approach for evaluating competing modelling solutions for *list sequential data* in Linked Open Data. So far, SPARQL-based benchmarks have been developed to evaluate competing storage solutions against generic use cases, deemed to be representative of the key features of the query language [9] or, alternatively, to how real users query linked data [26]. However, a given conceptual model can be encoded in RDF in different ways, providing alternative (and competing) solutions for the data engineer. This viewpoint calls for a *task-driven* approach to benchmarking.

In this work we survey methods for modelling sequences in RDF, and propose a pragmatic approach for assessing their performance in typical SPARQL queries and triplestores. The objective is to discuss the various modelling practices and provide recommendations to developers in understanding the trade-offs in encoding lists in a large-scale Linked Open Dataset. To do this, we develop and use a benchmark of datasets and queries [23] to compare the competing models against a set of core requirements reflecting the *query-ability* of the resulting data. This paper is complementary to [23], that we use, and it is focused on introducing the methodology behind its development and the experimental results. The research questions we aim to answer are:

- (1) *Do RDF lists modelling practices have an impact in the performance and availability of sequential retrieval of Linked Data?*
- (2) *Can we identify patterns enabling the publishing of RDF lists in a sustainable way?*

Crucially, we intend to evaluate the following hypothesis: *The efficiency of retrieving lists of linked data depends primarily on how they are structured. Without ad-hoc optimisations, the impact of modelling solutions on data availability is independent of the database engine (triple-store invariance hypothesis)*[4]

Specifically, we contribute: (1) **A survey** of modelling practices for representing RDF lists in the Linked Data world. (2) **A set of paradigmatic *structural***

[3] Sequence: `http://ontologydesignpatterns.org/wiki/Submissions:Sequence`.

[4] We are aware that one way to solve these problems is to optimize a database engine to support a specific RDF model. Here, we propose to evaluate existing engines pragmatically, and find evidence to eventually justify the development of such optimizations.

***design patterns*** for representing sequences, derived from the survey. (3) **Extensive experiments** aimed at evaluating the competing modelling solutions in terms of scalability for data retrieval, focusing on a set of basic requirements.

The rest of the paper is structured as follows. After scoping our contribution in the context of related work, we introduce the research methodology in Section 3. We discuss reference scenarios and express generic, task-based requirements for RDF lists in Section 4. Section 5 reports on our survey and presents a set of reference structural design patterns. These are then formalised in SPARQL to match the requirements, in Section 6. Section 7 reports on the experiments. Results are discussed in Section 8, that concludes our paper.

## 2 Related work

We consider research in two overlapping areas with our work: modeling of sequential RDF data; and performance of querying over such data using benchmark queries and datasets.

The Resource Description Framework (RDF) specification [30], and more specifically the RDF Schema (RDFS) recommendation [6] define container classes for the purpose of 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. Besides containers, [6] 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 [31] ordered lists like `"@list": [ "joe", "bob", "jaybee" ]` have equivalent representations as `rdf:List` in RDF. Similarly, the RDF 1.1 Turtle [3] 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 [14] have been proposed to represent sequences, e.g. the Sequence Ontology Pattern[5] (SOP) and the Collections Ontology [8] that focus on handling lists in OWL 2 DL, specifically.

We focus on practical approaches that assess querying sequential RDF data; for a theoretical study on the complexity of SPARQL, see [25]. 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) [5] generates benchmark data around exploring products and analyzing consumer reviews. The Lehigh University Benchmark (LUBM) [17] facilitates the evaluation of Semantic Web repositories by generating benchmark data about universities, departments, professors and students. SP$^2$Bench [28] 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 [21]. Similarly,

[5] `http://ontologydesignpatterns.org/wiki/Submissions:Sequence`

the DBpedia SPARQL benchmark [24] 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" [1]. Other datasets, such as Linked SPARQL queries (LSQ) [26], 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 [9][6]. Pragmatic approaches to benchmarking are not new and it is common practice to develop ad-hoc benchmarks to support specific applications (e.g. [32]). Benchmark methodologies have been proposed for covering specific aspects of SPARQL, for example federation [15]. The Linked Data Benchmark Council (LDBC), an industry-led initiative aimed at raising the 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. In addition, benchmarks should expose technology to a workload and to do that it is important focusing on *choke points* when defining benchmark tasks [2]. Our methodology is inspired by these guidelines.

We can identify three open issues: (1) To the best of our knowledge, none of these benchmarks assess querying RDF data with sequential information, e.g. `rdf:List` or `rdf:Seq`. In our work, we propose data and queries to evaluate RDF sequences specifically, thus addressing a new benchmark task. (2) Among the variety of solutions for modelling sequences, the `rdf:List` method appears to be the "official" one, being part of the RDFS specification and also recommended as a good design pattern in a Linked Data Pattern book [12]. However, it is well known how such method is poorly supported in SPARQL [12]. (3) Despite the proliferation of SPARQL benchmarks (and methodologies), there is no clear guideline or methodology on how benchmarks to compare competing modelling solutions for Linked Open Data should be designed.

## 3 Methodology

In this section we describe a methodology specifically designed to pragmatically evaluating the performance of competing modeling solutions for Linked Data publishing. Phases of the methodology are: requirements, survey, formalisation, and evaluation, that we illustrate.

**Requirements** In the initial phase, we identify the model that is the object of study. (In our case, it is the well-known data structure *list*, a *finite collection of ordered elements.*) However, in order to evaluate the performance of a concrete implementation of the model we need to identify a set of core functional requirements. Requirements should be formulated as competency questions [16]. After identifying the competency questions, possible modelling solutions should be looked for.

**Survey** Modelling solutions should be relevant to practitioners, by referring to

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

at least 1 concrete dataset adopting the modelling practice. Ideally, we want to ensure they are a complete set of solutions and that there is no relevant approach left out. After listing the modelling solutions, they should be abstracted in *structural patterns* and ensure these patterns are minimal with relation to the data model.

**Formalisation** Each modelling solution should be encoded in RDF and in SPARQL. Particularly, each pattern must be challenged to fit the competency questions designed in the initial phase, and respective solutions encoded in SPARQL queries. By doing this it is fundamental to ensure that the output is *semantically equivalent*, ideally the exact same, for all query variants. In addition, it is fundamental that queries are *minimal* by keeping them in the simplest form, for example adopting good practices for SPARQL query optimization [29]. Particularly: (a) subqueries should be avoided, when possible, (b) SPARQL operators should be reduced to the minimum necessary, (c) variables should be projected in the result set only when necessary, and (d) blank nodes should be preferred to named variables[7].

**Evaluation** The objective of this phase is to empirically evaluate the different solutions. 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, response time must be measured at different scale with respect to the data size. To do that, it is suggested to generate a set of realistic datasets at different scales. Experiments are performed for each modelling prototype with different dataset sizes and, crucially, with different database engines. From the results, derive recommendations on how to better represent the given model in Linked Data.

## 4 Requirements

We derive reference scenarios from Linked Open Data implementing sequences in various relevant domains. These are:

*Author lists.* The order of authors of scientific publications is relevant in several research areas, and may reflect the contribution of each author in a quantitative way. It is important to know which is the first author, for example, or to display them in the right order.

*Album tracks.* Tracks are ordered sequentially in a music album. Users may choose to play the 3rd or 7th song in the list.

*Recipes.* Recipes are lists of actions that should be executed in order.

*MIDI LD.* A MIDI object encoded in Linked Data is a sequence of encoded audio *events* [22]. These should be returned in order for a MIDI song to be played, and client applications should be able to jump to a specific event in the sequence, or to select a range of events to play.

From the above scenarios, we identify three *necessary* and *sufficient* requirements: (a) the capability of obtaining the list as an ordered sequence of items.

[7] In fact, blank nodes don't require the matching node value to be kept in memory as part of the query solution to be projected.

This list does not need to include additional data, the returning object can just be a list of strings; (b) the capability of obtaining an item at a certain position in the list; and (c) the capability of obtaining a ordered set of items with the scope of a specific range. From these considerations, we formulate the following competency questions:

- $CQ1$ (All Ordered): What is the ordered content of the sequence?
- $CQ2$ (Nth Lookup): Which is the *n-th* item in the sequence?
- $CQ3$ (Ordered Range): What are the items from the N to M in the sequence?

Clearly, other operations are possible and useful, for example checking the presence of an item in the list or comparing lists. Here we focus on atomic operations related to *access* list items by *order*, and leave a more complete requirement analysis as future work. In addition, we only consider lists with one item only in a certain position, although some models may support otherwise.

## 5 Modeling Sequences in RDF

There are various models for representing a sequence, a *finite collection of ordered elements*, in RDF. In this section we describe such models, argue abstractions that integrate some of them, and discuss their properties. We conduct a systematic survey of these models, by selecting them from the following sources: W3C standards[8] and recommendations; the ontology design patterns [14] portal[9]; practical choices in RDF datasets published as resource track papers in ISWC (e.g. [4], [22]); and lookups of relevant terms (e.g. `list`, `sequence`) in the Linked Open Vocabularies [33] portal[10].

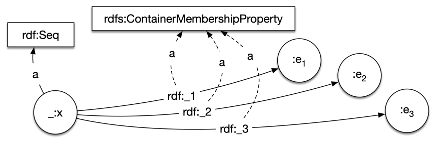


Fig. 1: The RDF Sequence model.

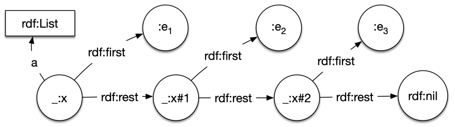


Fig. 2: RDF List model.

***RDF Sequences*** The RDF Schema (RDFS) recommendation [6] 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).

[8] `https://www.w3.org/standards/`
[9] `http://ontologydesignpatterns.org`
[10] `https://lov.linkeddata.es/dataset/lov/`

**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 ("_").

***RDF Lists*** The RDFS recommendation [6] 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 using 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.

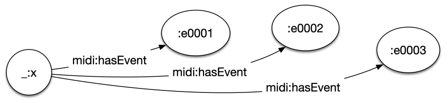


Fig. 3: URI-based list model.

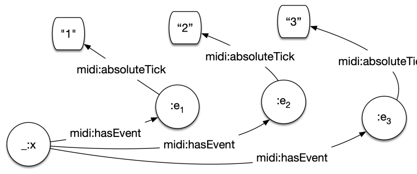


Fig. 4: Number-based list model.

***URI-based Lists*** A more practical approach followed by many RDF datasets [4, 22] 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 [22] (see Figure 3).

**Properties.** URI-based lists indicate membership through the use of *class membership* or through *properties*. Order is *absolute* and given by URI-embedded sequential identifiers.

***Number-based Lists*** Another practical model, used e.g. in the Sequence Ontology/Molecular Sequence Ontology (MSO) [13],[11] 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

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

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 membership through the use of *class membership* or through *properties*. Order is *absolute* and given by an integer index in a literal as an object of an additional property.

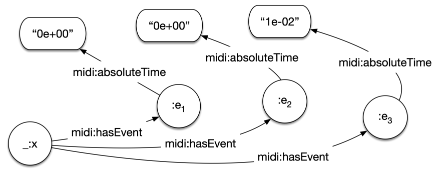


Fig. 5: Timestamp-based list model.

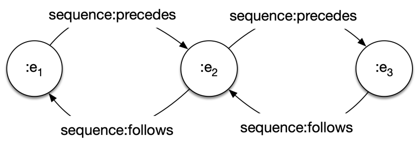


Fig. 6: The Sequence Ontology Pattern model.

***Timestamp-based Lists*** Similarly to Number-based lists, other lists modeled by e.g. the Simple Event Model (SEM) [18], use timestamp markers instead of integer indexes to indicate the time in which the element of the list occurs. This is particularly useful in event-based applications, in which order clashes in the list are of lesser importance, as long as the timestamp order is preserved. For instance, the triple `<http://purl.org/midi-ld/piece/8cf9897535d79e68c33a3076aa06d073/track00/event0006> midi:absoluteTime 0e+00` indicates that the 7th event occurs at the start of the list, possibly simultaneously with other events (see Figure 5).

**Properties.** Timestamp-based lists indicate membership through the use of *class membership* or through *properties*. Order is *absolute* and given by a timestamp in a literal as an object of an additional property.

***Sequence Ontology Pattern*** A number of models use general RDF/RDF-S/OWL semantics to model sequences in domain specific ways. For example, the Time Ontology [19] and the Timeline Ontology[12] offer a number of classes and properties to model temporality and order, including timestamps (see Section 5), but importantly also before/after relations. The *Sequence Ontology Pattern*[13] (SOP) is an ontology design pattern [14] 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 6).

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

[12] `http://motools.sourceforge.net/timeline/timeline.html\#`
[13] `http://ontologydesignpatterns.org/wiki/Submissions:Sequence`

## 6 Formalisation

In this phase we aim at formally represent each solution in RDF/SPARQL in order to answer the requirements. We consider that in competency questions *CQ*1 and *CQ*3, we aim at returning a list of URIs representing ordered set of items, while we aim at returning a single item for *CQ*2. Particularly, we report on how each way of representing a MIDI Linked Data event sequence can be queried in SPARQL. Therefore, we use the vocabularies of the `List.MID` benchmark datasets, developed following our methodology [23]. In what follows, the target *graph* contains a single list linked to an entity of type `midi:Track` including the ordered collection of MIDI events. The *result set* is expected to be the projection of the single variable `?entity`. For simplicity, we name the target list `:list` and omit redundant aspects of the query (such as SELECT and FROM clauses). In order to ensure *minimality*, according to our methodology, we omit to include additional application-specific data, e.g. attributes and values of the MIDI event.

*RDF Sequences (Seq).* This modelling solution relies on a predicate for indexing the position of the item in the list. Although each predicate has the capability of representing a cardinal number, the predicate itself cannot be used for ordering operations as its encoding as URI has the effect of being ordered as string. In order to answer the first `CQ1` (All Ordered) it is necessary to rely on the `ORDER BY` operator and extract the cardinal number from the predicate URI string as follows (where `?seq` is the container membership property):

```
WHERE {:list a midi:Track ; midi:hasEvents [ ?seq ?event ] .
  BIND (xsd:integer(SUBSTR(str(?seq), 45)) AS ?index)
} ORDER BY ?index
```

In principle, the `Nth Lookup` (CQ2) could be resolved by replacing the `?seq` variable with the property `_N` (for example, `rdf:_995` for the 995th element of the list). However, containers are meant to be open-ended. The specification does not declare that the predicate number represents anything other than the order of the elements in the `rdf:Seq`. Therefore, it would be unsafe to assume that the range of `rdf:_5` to be the item in fifth position. The minimal and safe approach would be to operate on the sequence at query time:

```
WHERE {:list a midi:Track ; midi:hasEvents [ ?seq ?event ] .
  BIND (xsd:integer(SUBSTR(str(?seq), 45)) AS ?index)
} ORDER BY ?index OFFSET 995 LIMIT 1
```

A similar approach can be adopted to select an ordered range (CQ3).

*RDF Lists (List).* This modelling solution requires to be queried with a property path in order to traverse the list from the root to the tree to the last item. However, there is no guarantee that the projections of the `?event` variable would keep the sequence order. In order to derive the index from the data, we need to perform aggregation and ordering, as follows:

```
SELECT ?event (count(?step) as ?index)
WHERE {
 :list a midi:Track ; midi:hasEvents ?events .
 ?events rdf:rest* ?step . ?step rdf:rest* ?elt .
 ?elt rdf:first ?event
} GROUP BY ?event ORDER BY ?index
```

In our benchmark, like in many of the use cases observed in our survey, entities are directly *attached* to the list. However, an entity can be shared among lists and therefore the list element itself may refer to it indirectly, for example with a pattern such as `_:bnode123 rdf:first/rdf:value ?event`, resulting in an expansion of the given query. CQ2 could be implemented by specifying a path of `rdf:rest/rdf:first` as long as the position of the item we want to retrieve. However, this would require to write a different query for each item and result in a large query string that could be possibly rejected by the Web server. Therefore, we keep the same query layout for `CQ2` and `CQ3`[14].

*URI-based lists (Uri).* This practical approach is very economic and can be expressed as follows:

```
WHERE { [] a midi:Track ; midi:hasEvent ?event .
BIND (xsd:integer(SUBSTR(str(?event), 77)) AS ?id) } ORDER BY ?id
```

This query can be expanded to include `OFFSET` and `LIMIT` clauses to satisfy `CQ2` and `CQ3`.

*Number-based Lists (Prop_number) and Timestamp-based Lists (Prop_time)* This solution relies on a data value incorporating a numeric index. The main difference here is that an additional triple pattern needs to be employed, although the value is meant to be a number and can be passed *as-is* to the `ORDER BY` clause:

```
WHERE { [] a midi:Track ; midi:hasEvent ?event .
    ?event midi:absoluteTick ?tick . } ORDER BY ?tick
```

where `?tick` is of datatype `xsd:integer`. Timestamp-based Lists are similar to *Prop_number* but with a time datatype. Additionally, they use a different approach as the numeric value is not necessary an incremental index and can also include multiple entities in the same position.

*Sequence Ontology Pattern (SOP)* The sequence ontology pattern uses the predicates *precedes* and *follows* to model the sequence. The query can be formulated combining two triple patterns. However, we need to combine this solution with an index-based on, for example, the *Uri* pattern, in order to ensure that the order is preserved in the output.

```
WHERE { [] a midi:Track ; midi:hasEvent ?event .
  ?event sequence:precedes? ?next_event .
  ?next_event sequence:follows? ?event .
  BIND (xsd:integer(SUBSTR(str(?event), 77)) AS ?id)
} ORDER BY ?id
```

We can conclude that all the modelling solution will need to rely on the SPARQL `ORDER BY` clause and that their main difference is in the way the index is represented in the data and in the necessary operations to serve it for sorting[15].

[14] Some triple stores supports the operator `rdf:rest{n}` targeting the $n$-th item in the path. However, this syntax, introduced during the development of SPARQL 1.1, was discarded in the final specification.

[15] Although in some cases a system may return triples reflecting the order they had at insertion time, we cannot assume that triples are returned respecting any particular order. Therefore, the `ORDER BY` clause is necessary in all cases.

In addition, although some of the models explored accept *ties*, i.e. lists with multiple items in the same position, these are not considered in this work.

## 7 Evaluation

Each model was mapped to a number of queries covering the three competency questions listed in Section 4. We analysed the MIDI linked data endpoint for deciding the size of the benchmark dataset: the `List.MID` benchmark [23]. The dataset has approximately 300K midi songs including an average of 5 tracks each for a total of 1.5M tracks. A single track contains an ordered list of *events*, and these are the ones we are going to query in our benchmark. Very long tracks are rare in the database and only 10 of them have more the 120k events. We prepared a dataset for each modelling solution and 5 MIDI tracks of different sizes: 1k, 30k, 60k, 90k, and 120k triples respectively. Therefore, there will be a dataset with a list of size 1k implementing, for example, the *Seq* pattern, one of size 30k, and so on for each model type, for a total of 25 datasets.

We performed experiments with multiple triple stores. Each database was prepared by loading all the datasets in different named graphs. At runtime, the query was rewritten to target a specific named graph[16].

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. This is 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 growing sizes. Experiments are executed on a Linux VM with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz 8-core and 32G RAM. During the experiments, no other application was running on the instance rather then the experiment's client process and the database server.

To summarise, the dimensions considered in our experiments are therefore: (a) Model (one of): Seq, List, Number Index, Time Index, ODP, URI Index (b) Dataset Size (one of): 1k, 30k, 60k, 90k, 120k (c) Query (one of): All, Lookup, Range (d) Database (one of): Virtuoso, Blazegraph, Fuseki-Tdb, Fuseki-Mem.

[16] 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.

In what follows we report on overall *response time*, meaning the amount of time the client had to wait before obtaining the complete answer. Each experiment was repeated 10 times, reported measures refer to average values. A timeout of 300 seconds has been set, although it was never reached. We also analysed standard deviation and in most cases the value was below 10% of the total time. The cases where it was higher relate to very fast response times (below the second) and are therefore not problematic. 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[17].

Tables 1d-1l report the average values response time. Figures 7a-7l report on scalability. The most inefficient method is the one relying on `rdf:List`, for which most of the experiments timed out and a number of experiments with Fuseki failed with a server error (probably due to excessively deep property path in the query). The SOP method is also not very efficient. The performances of the URI, prop_number, and prop_time methods are very similar, although they behave slightly different with the various triple stores. All models scale linearly with the amount of data (except for the failed experiments). Results are coherent for all three CQs and demonstrate a clear trend among different engines. Supplementary material is available for reproducibility [11].

## 8 Discussion and conclusions

The experiments demonstrate our hypothesis that the efficiency of retrieving sequential linked data depends heavily on how they are structured. Indeed, modelling practices have an impact in the performance and availability of sequential retrieval. Crucially, the behavior of the various models is consistent 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 `CQ`s. The most efficient ways of representing order is by using indexes in values like in *prop_ number* and *prop_ timestamp.*Also indexes hidden in URIs perform well, both on the entity (subject/object) and on the `rdf:Seq` method (predicate). The reasons are probably related to database indexes on the basic triple patterns[18] However, embedding the order semantics in string URIs does not seem an elegant solution. 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 solution. **With the given**

[17] We also collected information about RAM usage and CPU. We could not observe particular differences in those, except in the case of Apache Fuseki on queries with multiple joins and property paths, mainly related to the SOP pattern. However, here we focus on the performance with respect to client applications and not on studying resource consumption on the server side.

[18] Fuseki behaves a bit different and seems to badly tolerate many joins, generally. Also, on Fuseki URI processing is faster than picking indexes in values (number/date). However, here we focus on trends observed among the various database engines and do not discuss specific differences between them.

Table 1: Response Time (milliseconds)

(a) Q1: Blazegraph

| Model | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 44 | 402 | 746 | 1,121 | 1,413 |
| sop | 111 | 1,439 | 2,640 | 3,809 | 5,096 |
| uri | 42 | 355 | 685 | 993 | 1,337 |
| prop_number | 41 | 315 | 603 | 839 | 1,091 |
| prop_time | 63 | 324 | 587 | 882 | 1,092 |
| list | 11,336 | - | - | - | |

(b) Q1: Virtuoso

| | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 31 | 301 | 479 | 650 | 858 |
| sop | 93 | 569 | 1,043 | 1,509 | 2,035 |
| uri | 30 | 275 | 433 | 582 | 764 |
| prop_number | 37 | 161 | 215 | 297 | 343 |
| prop_time | 49 | 171 | 235 | 314 | 413 |
| list | 22,653 | - | - | - | - |

(c) Q1: Fuseki (TDB)

| Model | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 42 | 374 | 669 | 972 | 1,341 |
| sop | 64 | 1,333 | 2,793 | 4,416 | 6,083 |
| uri | 26 | 251 | 597 | 675 | 981 |
| prop_number | 38 | 435 | 915 | 1,349 | 2,346 |
| prop_time | 180 | 461 | 1,005 | 1,307 | 2,143 |
| list | - | - | - | - | - |

(d) Q1: Fuseki (MEM)

| | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 32 | 628 | 1,235 | 1,800 | 2,572 |
| sop | 76 | 1,905 | 3,770 | 5,626 | 7,692 |
| uri | 34 | 598 | 1,229 | 1,778 | 3,132 |
| prop_number | 38 | 764 | 1,483 | 2,257 | 3,077 |
| prop_time | 34 | 770 | 1,540 | 2,309 | 3,225 |
| list | 5,104 | - | - | - | - |

(e) Q2: Blazegraph

| Model | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 46 | 243 | 480 | 693 | 947 |
| sop | 107 | 1,267 | 2,290 | 3,381 | 4,506 |
| uri | 44 | 220 | 424 | 650 | 830 |
| prop_number | 46 | 190 | 350 | 470 | 592 |
| prop_time | 52 | 176 | 317 | 445 | 566 |
| list | 14.70 | 20,767 | - | - | - |

(f) Q2: Virtuoso

| | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 19 | 173 | 309 | 484 | 632 |
| sop | 31 | 427 | 898 | 1,320 | 1,706 |
| uri | 19 | 144 | 282 | 392 | 543 |
| prop_number | 14 | 19 | 27 | 29 | 36 |
| prop_time | 14 | 20 | 27 | 36 | 38 |
| list | 22,760 | - | - | - | - |

(g) Q2: Fuseki (TDB)

| Model | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 34 | 172 | 290 | 389 | 573 |
| sop | 59 | 1,103 | 2,289 | 3,657 | 5,204 |
| uri | 35 | 145 | 231 | 329 | 460 |
| prop_number | 40 | 275 | 511 | 761 | 1,664 |
| prop_time | 46 | 288 | 504 | 755 | 1,523 |
| list | - | - | - | - | - |

(h) Q2: Fuseki (MEM)

| | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 29 | 174 | 293 | 386 | 474 |
| sop | 68 | 1,671 | 3,306 | 4,911 | 6,969 |
| uri | 30 | 170 | 259 | 332 | 426 |
| prop_number | 38 | 271 | 498 | 672 | 818 |
| prop_time | 40 | 291 | 470 | 612 | 743 |
| list | - | - | - | - | - |

(i) Q3: Blazegraph

| Model | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 41 | 248 | 489 | 711 | 881 |
| sop | 111 | 1,224 | 2,260 | 3,362 | 4,599 |
| uri | 35 | 227 | 422 | 645 | 840 |
| prop_number | 37 | 200 | 354 | 477 | 602 |
| prop_time | 39 | 181 | 325 | 457 | 575 |
| list | 14.70 | - | - | - | - |

(j) Q3: Virtuoso

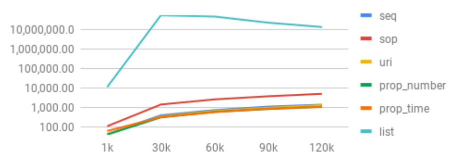
| | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 21 | 161 | 309 | 448 | 616 |
| sop | 34 | 424 | 894 | 1,265 | 1,734 |
| uri | 19 | 143 | 275 | 394 | 533 |
| prop_number | 14 | 19 | 27 | 33 | 35 |
| prop_time | 15 | 22 | 29 | 32 | 38 |
| list | 22,655 | - | - | - | - |

(k) Q3: Fuseki (TDB)

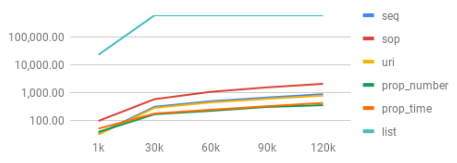
| Model | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 34 | 170 | 284 | 405 | 600 |
| sop | 56 | 1,103 | 2,300 | 3,488 | 5,355 |
| uri | 35 | 130 | 228 | 340 | 471 |
| prop_number | 42 | 285 | 510 | 755 | 1,635 |
| prop_time | 43 | 284 | 520 | 749 | 1,508 |
| list | - | - | - | - | - |

(l) Q3: Fuseki (MEM)

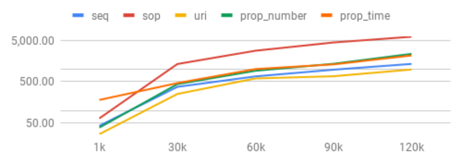
| | 1k | 30k | 60k | 90k | 120k |
|---|---|---|---|---|---|
| seq | 28 | 180 | 284 | 384 | 446 |
| sop | 68 | 1,733 | 3,282 | 5,087 | 6,920 |
| uri | 32 | 171 | 253 | 353 | 422 |
| prop_number | 36 | 282 | 493 | 689 | 823 |
| prop_time | 35 | 269 | 467 | 601 | 786 |
| list | - | - | - | - | - |



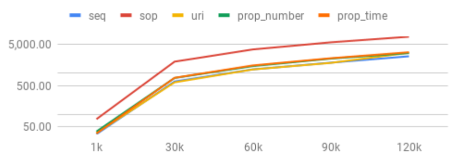
(a) Q1: Blazegraph



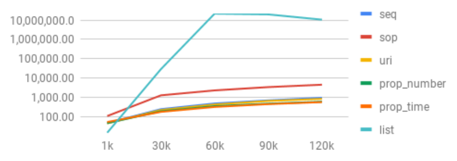
(b) Q1: Virtuoso



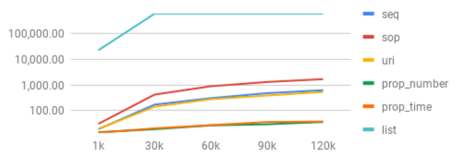
(c) Q1: Apache Fuseki (TDB)



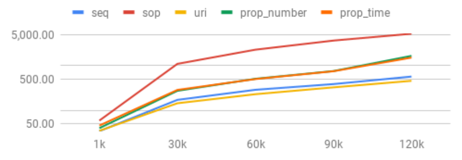
(d) Q1: Apache Fuseki (In-Memory)



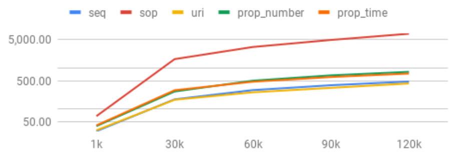
(e) Q2: Blazegraph



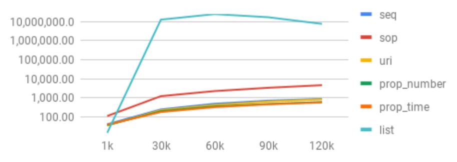
(f) Q2: Virtuoso



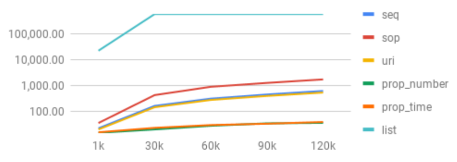
(g) Q2: Apache Fuseki



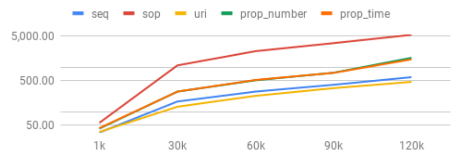
(h) Q2: Apache Fuseki In-Memory



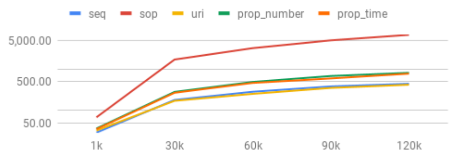
(i) Q3: Blazegraph



(j) Q3: Virtuoso



(k) Q3: Apache Fuseki



(l) Q3: Apache Fuseki In-Memory

Fig. 7: Performance scalability

**results, the methods relying on `rdf:List` (the recommended standard) and `SOP` (*a high-quality ontology engineering solution*) clearly underperform in commonly used triplestores and, under this circumstances, their use should be discouraged for publishing lists as Linked Data.**

However, in this paper we only measure query-ability of 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. Particularly, we do not consider data management operations such as adding or removing elements from a list. For those operations, solutions that do not store an index such as `rdf:List` or `SOP` would require less operations, possibly overturning the final judgement. However, these aspects are left to future work. With the aid of a task-based approach for benchmark development we were able to study pragmatically how to better publish sequential linked data and identified a fundamental problem of typical, recommended solutions. Intuitively, we can argue that there is a trade-off between incorporating an index in the data and allowing for easier and faster data management. We aim at expanding our experiments to also consider scenarios of Linked Data management and benchmark operations for list manipulations. Finally, we presented a preliminary work on *model-centric and task-oriented benchmarks* for Linked Data. The sketched methodology allowed us to identify design patterns that could negatively affect SPARQL endpoints availability using an approach that is independent from the concrete modelling problem. We aim at evaluating such methodology with more modelling scenarios, such as tabular structures, sets, or n-ary relations.

## References

1. Aluç, G., et al: Diversified Stress Testing of RDF Data Management Systems. In: The Semantic Web – ISWC. pp. 197–212. Springer, Cham (2014)
2. Angles, R., et al: The linked data benchmark council: a graph and rdf industry benchmarking effort. ACM SIGMOD Record **43**(1) (2014)
3. 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/
4. 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)
5. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal on Semantic Web & Information Systems **5**(2), 1–24 (2009)
6. Brickley, D., Guha, R.: RDF Schema 1.1. Tech. rep., World Wide Web Consrotium (2014), https://www.w3.org/TR/rdf-schema/
7. 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)
8. Ciccarese, P., Peroni, S.: The collections ontology: creating and handling collections in owl 2 dl frameworks. Semantic Web **5**(6), 515–529 (2014)
9. Conrads, F., et al: Iguana: A generic framework for benchmarking the read-write performance of triple stores. In: The Semantic Web - ISWC 2017 (2017)
10. 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)

11. Daga, E., Meroño-Peñuela, A.: Software and data for the experiments (Jul 2019). https://doi.org/10.5281/zenodo.3355543
12. Dodds, L., Davis, I.: Linked data patterns. Online: http://patterns. dataincubator. org/book (2011)
13. Eilbeck, K., et al: The sequence ontology: a tool for the unification of genome annotations. Genome biology **6**(5) (2005)
14. Gangemi, A.: Ontology Design Patterns for Semantic Web Content. In: The Semantic Web – ISWC. Springer (2005)
15. 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)
16. Grüninger, M., Fox, M.S.: The role of competency questions in enterprise engineering. In: Benchmarking—Theory and practice, pp. 22–31. Springer (1995)
17. 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)
18. Hage, V., et al: Design and use of the simple event model (sem). Web Semantics: Science, Services and Agents on the World Wide Web **9**(2) (2011)
19. Hobbs, J.R., Pan, F.: Time ontology in OWL. W3C working draft **27**, 133 (2006)
20. Hyland, B., et al: Best practices for publishing linked data. Tech. rep., W3C Working Group Note (2014), https://www.w3.org/TR/2014/NOTE-ld-bp-20140109/
21. 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)
22. Meroño-Peñuela, A., et al.: The MIDI Linked Data Cloud. In: The Semantic Web - ISWC 2017. vol. 10587, pp. 156–164 (2017)
23. Meroño-Peñuela, A., Daga, E.: List.MID: A MIDI-Based Benchmark for Evaluating RDF Lists. In: The Semantic Web – ISWC 2019 (2019)
24. Morsey, M., et al: Dbpedia sparql benchmark–performance assessment with real queries on real data. In: The Semantic Web – ISWC. Springer (2011)
25. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: The Semantic Web - ISWC (2006)
26. Saleem, M., et al: LSQ: Linked SPARQL Queries Dataset. In: The Semantic Web - ISWC 2015. LNCS, vol. 9367. Springer (2015)
27. Schaible, J., et al: Survey on common strategies of vocabulary reuse in linked open data modeling. In: European Semantic Web Conference. Springer (2014)
28. Schmidt, M., et al: SPˆ 2Bench: a SPARQL performance benchmark. In: Data Engineering, 2009. ICDE'09. IEEE (2009)
29. Schmidt, M., et al: Foundations of sparql query optimization. In: 13th International Conference on Database Theory. ACM (2010)
30. Schreiber, G., Raimond, Y.: RDF 1.1 Primer. Tech. rep., World Wide Web Consrotium (2014), https://www.w3.org/TR/rdf11-primer/
31. 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/
32. Thakker, D., et al: A pragmatic approach to semantic repositories benchmarking. In: Extended Semantic Web Conference. Springer (2010)
33. 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)