A design space for RDF data representations

Sagi, Tomer; Lissandrini, Matteo; Pedersen, Torben Bach; Hose, Katja

Published in:
The VLDB Journal

DOI (link to publication from Publisher):
10.1007/s00778-021-00725-x

Creative Commons License
CC BY 4.0

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Sagi, T., Lissandrini, M., Pedersen, T. B., & Hose, K. (2022). A design space for RDF data representations. *The VLDB Journal*, *31*(2), 347-373. https://doi.org/10.1007/s00778-021-00725-x

The VLDB Journal (2022) 31:347–373
https://doi.org/10.1007/s00778-021-00725-x

SPECIAL ISSUE PAPER

A design space for RDF data representations

Tomer Sagi[1] · Matteo Lissandrini[1] · Torben Bach Pedersen[1] · Katja Hose[1]

Received: 31 August 2020 / Revised: 8 September 2021 / Accepted: 9 December 2021 / Published online: 21 January 2022

Abstract
RDF triplestores' ability to store and query knowledge bases augmented with semantic annotations has attracted the attention of both research and industry. A multitude of systems offer varying data representation and indexing schemes. However, as recently shown for designing data structures, many design choices are biased by outdated considerations and may not result in the most efficient data representation for a given query workload. To overcome this limitation, we identify a novel three-dimensional design space. Within this design space, we map the trade-offs between different RDF data representations employed as part of an RDF triplestore and identify unexplored solutions. We complement the review with an empirical evaluation of ten standard SPARQL benchmarks to examine the prevalence of these access patterns in synthetic and real query workloads. We find some access patterns, to be both prevalent in the workloads and under-supported by existing triplestores. This shows the capabilities of our model to be used by RDF store designers to reason about different design choices and allow a (possibly artificially intelligent) designer to evaluate the fit between a given system design and a query workload.

Keywords RDF · Data representation · SPARQL · Database · Query · Knowledge graphs

1 Introduction

The resource description framework (RDF) [44] is a popular standard for storing and sharing factual information, predominantly created from sources on the World Wide Web. RDF is represented as subject–predicate–object triples, usually modeled as a graph, whose nodes serve as subjects and objects and edges as predicates. RDF stores, often called triplestores, are designed to support the storage of RDF data and its efficient querying by exposing a declarative query API standardized in the SPARQL standard [63]. Interest in triplestores has grown steadily over the past decade. In particular, both research and industry are employing these systems to store and query knowledge bases augmented with semantic annotations [59,85]. Thus, a multitude of triplestore implementations are available, ranging from academic prototypes (e.g., RDF-3X [55], Hexastore [81]) and community projects (e.g., JENA TDB [58], and Rya [64]) to commercial products (e.g., Virtuoso [24], GraphDB [57], and Neptune [13]).

RDF data differs from relational data in the complexity of its structure. In practice, real-life RDF datasets are highly heterogeneous in structure, especially compared to relational datasets [22]. This structural complexity causes query performance to vary substantially [3]. Converting RDF data to the relational model, utilizing existing, mature, RDBMS technologies (e.g., by Oracle [20] and IBM [17]), introduces unique challenges due to the substantial information heterogeneity and the inherent absence of a strict schema [67]. Together with the need to accommodate increasingly large RDF graphs, these unique properties of RDF data have led researchers and companies to suggest RDF-specific designs rather than just mappings from RDF to relational formats. Reviewing the features of the numerous available systems reveals each to employ its own list of design choices each of which seem equally compelling to the casual observer. Recently, a different set of systems has been proposed to handle graph data represented as node and edge labeled multigraphs annotated with properties, i.e., *property graphs*

✉ Tomer Sagi
tsagi@cs.aau.dk

Matteo Lissandrini
matteo@cs.aau.dk

Torben Bach Pedersen
tbp@cs.aau.dk

Katja Hose
khose@cs.aau.dk

1 Department of Computer Science, Aalborg University, Aalborg, Denmark

(PG) [14]. Despite the fact that both the RDF model and the PG model handle graph-shaped data, the two models differ substantially in terms of the functionalities they offer. In particular, while RDF represents data as a set of triples, PG DBMS are designed to query labeled objects annotated with properties in the form of key-value pairs. Therefore, the analysis of core operations supported by triplestores substantially differs from those supported by PG DBMS. (This can be seen, for instance, by comparing our analysis with the operations studied in a recent PG DBMS microbenchmark [47].) For instance, in a PG, we can select nodes having a specific label and a specific attribute set to a specific type accessing only node objects, while in an RDF triplestore an equivalent query will need to query a set of triples instead.

Although there are several surveys of the many existing triplestores [2,52,56,59], they are either limited to providing a taxonomy of the features offered by the systems (e.g., API and data load facilities) or classify them according to their underlying technology (e.g., relational versus native graph). Thus, these surveys offer a vast compendium of alternative systems but only little information regarding the design space in which their internal architecture reside.

Instead, in this work, we provide a *unifying three-dimensional design space* across three axes: *Subdivision, Compression*, and *Redundancy* (SCR, see Fig. 4). The design space defines the dimensions along which any storage system for RDF data must be designed. We then provide a review of the current choices made over these dimensions as well as a discussion into unexplored options. To guide the analysis of this space and of current solutions within it, we define a corresponding *feature space for query access patterns*[1] and a cost model to tie the different choices in the design space to their impact on the performance of specific query workloads.

Our approach is, in part, inspired by the *Data Calculator* [41]. However, while the Data Calculator focuses on optimizing a single low-level data structure for generic (i.e., non-SPARQL-specific) data access operations, our approach works at a higher level and focuses on evaluating the set of data representations employed by a triplestore; choosing the right design choices to match the RDF access patterns this system must support. Thus, a triplestore system developer could use our approach to identify access patterns that are currently not optimized for by the data representations in their triplestore. For example, the Jena (TDB) system [58] does not feature any optimized data representation for filters selecting triples involving a specific data type or with a specific language tag, even though they are quite common in existing workloads (see Sect. 7). This is evident from analyzing Jena's existing data representations using our proposed design space (see Tables 5, 6, and 7). Once the developer has identified this unmet pattern, they can examine which combination of decisions is available in the three design space dimensions they wish to employ to satisfy this new pattern. In this example, one possible solution would be to employ an additional subdivision of the existing ID space to differentiate between data types and languages. This can lead to a different data organization within an existing index or to a new access-pattern-specific index. (In this case we also move within the redundancy dimension.) Since the Data Calculator is not aware of the presence of annotations such as language tags and the possibility to represent them separately by subdividing the ID space, it cannot identify this unmet need and it cannot provide an optimized solution for this access pattern. Similarly, the Data Calculator is not aware of other RDF-specific access patterns, e.g., reachability and path queries. Moreover, Jena employs three different main triple data representations based on standard B+trees and identical in all the low-level features considered by the Data Calculator. In the proposed access patterns feature space, we provide a way to differentiate between the RDF-specific access patterns each of these representations is required to support. Thus, we are able to evaluate and assess the compatibility of a set of data representations used in a triplestore with RDF-specific access patterns.

Our design space is based on intuitions widely shared across different data models and their DBMS [10]. However, these intuitions have not been formalized for RDF systems and existing RDF systems have not been analyzed based on them. We thereby make the following contributions.

1. A design space for RDF data representations employed in a triplestore that is simple enough to be intuitive, yet, as we show, powerful enough to analyze the benefits and trade-offs of each design choice.
2. A review of existing triplestores positioning them in this design space and identifying unexplored choices.
3. A feature space for SPARQL query access patterns allowing to characterize query execution over RDF.
4. A software tool to parse query workloads and analyze the patterns used.
5. A comprehensive analysis of how design choices impact system performance when answering access patterns with specific features.
6. An empirical evaluation of the prevalence of access patterns in commonly used query workloads.

The rest of the paper is structured as follows: Section 2 provides preliminary definitions followed by the query access pattern feature space in Sect. 3. We then introduce our design space in Sect. 4, followed by the review of existing systems

[1] Here, we refer to access patterns as logical operations in a query that, given some input values, establish what kind of output values must be returned. The difference from the use of the term *access patterns* in the relational model [27] and from the term *access paths* [72] is described in Sect. 3.

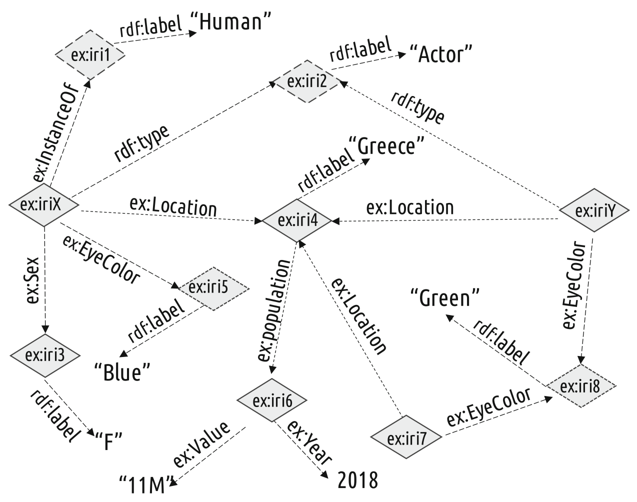


Fig. 1 Example of RDF data in graph format

(Sect. 5). Section 6 presents our impact analysis, distilling important findings obtained by the analysis of the systems design in our design space. This analysis is followed by the empirical evaluation in Sect. 7. Finally, we highlight the differences between this work and previous surveys in Sect. 8 and conclude by discussing the implications of our analysis and findings in Sect. 9.

2 Preliminaries

In this section, we formally define the basic concepts of RDF graphs and RDF graph patterns. Both revolve around the concept of RDF triples [44]. An RDF triple is a factual statement comprised of a subject (s), a predicate (p), and an object (o). Subjects and objects can be resources identified by International Resource Identifiers (IRI), anonymous nodes identified by internal IDs (called blank nodes), or literals. Predicates are always IRIs (resources) and never literals [62]. For example, the triple *(ex:iri1, rdfs:label, "Human")* states that the resource identified by the IRI *ex:iri1* has an *rdfs:label* which is the literal string *"Human"*. Collectively, nodes (i.e., resources, blank nodes, and literals) can be referred to as *atoms*. RDF allows to explicitly record knowledge codified as a graph where subjects and predicates serve as nodes and triples as edges.

Definition 1 *(RDF Triple/Statement & Graph)* Given a set of IRIs $\mathcal{I}$, blank nodes $\mathcal{B}$, and literals $\mathcal{L}$, a triple $(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times (\mathcal{I}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$ is called an *RDF triple*. In the (s, p, o) triple, also called an *RDF statement*, s is the subject, p is the predicate, and o is the object. An *RDF graph* $\mathcal{G}$ is a set of RDF triples.

Example 1 Consider the graph in Fig. 1, which is a graphical representation of a set of triples. In this case, `ex:iri1`, `ex:iri3`, `rdfs:label`, and `ex:Sex` are examples of IRIs in $\mathcal{I}$, the first two with the role of nodes and the second two with the role of predicates. On the other hand, `"11M"` and `2018` are literals in $\mathcal{L}$. Finally, (`ex:iri1`, `ex:Sex`, `ex:iri3`) is an (s, p, o) triple.

An RDF graph is queried by issuing a SPARQL [63] query to an evaluation engine, called a triplestore, whose core functionality is to compute answers based on the graph structures matching it. SPARQL queries contain one or more basic graph patterns, which are sets of triples with zero or more of their components replaced by variables, formally defined as follows.

Definition 2 (*Basic Graph Pattern* [62]) Assume an infinite countable set of variables $\mathcal{X}$. A *Basic Graph Pattern* (BGP) P is defined as a conjunction of a finite set of triple patterns $P = \{t_1, \ldots, t_n\}$, with $t_i \in P$ being a triple pattern defined as $t_i \in (\mathcal{I} \cup \mathcal{X}) \times (\mathcal{I} \cup \mathcal{X}) \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{X})$.

Thus, triple patterns are (s, p, o) triples where any position may be replaced by a variable. Solutions to the variables are found by matching the triple patterns in the BGP with triples in the RDF graph.

Moreover, there are special types of patterns that match sequences of edges satisfying a specific set of predicates, e.g., all nodes reachable by an arbitrary sequence of `ex:childOf` predicates. These patterns are called *property paths*. They are defined via a specialized form of expressions called *path expressions* (similar to regular expressions) and offer a succinct way to write parts of basic graph patterns and also extend matching of triple patterns to arbitrary length paths [35].

Definition 3 *(Property paths* [35]*)* SPARQL 1.1 defines a property path $\mathbf{p}$ recursively as follows. A property path is (1) any resource $a \in \mathcal{I}$; (2) given property paths $\mathbf{p}_1$ and $\mathbf{p}_2$, a property path is either a sequence of paths denoted by $\mathbf{p}_1/\mathbf{p}_2$, a disjunction paths denoted by $\mathbf{p}_1|\mathbf{p}_2$, a negation of a path denoted by $\hat{}\mathbf{p}_1$, a sequence of zero or more of the same path denoted by $\mathbf{p}_1*$, a sequence of one or more repetitions of the same path denoted by $\mathbf{p}_1+$, or either zero or one occurrences of the path denoted by $\mathbf{p}_1?$; alternatively (3) given resources $a_1, \ldots, a_n \in \mathcal{I}$, then any of the following expressions $!a_1$, $!\hat{}a_1$ $!(a_1| \ldots |a_n)$, $!\left(\hat{}a_1| \ldots |\hat{}a_n\right)$ and $!\left(a_1| \ldots |a_j|\hat{}\, a_{j+1}| \ldots |\hat{}a_n\right)$ where ! denotes negation, ^ denotes inversion and | denotes disjunction.

Hence, property paths are expressions over vocabulary $\mathcal{I}$ of all IRIs [86]. The language does not allow to express negated property paths, but it is possible to express negation on IRIs, inverted IRIs and disjunctions of combinations of IRIs and inverted IRIs. A property path triple is a tuple t of the form $(s, \mathbf{p}, o)$, where $s, o \in (\mathcal{I} \cup \mathcal{X})$ and $\mathbf{p}$ is a property path. Such a triple is a graph pattern that matches all pairs of nodes $\langle s, o\rangle$ in an RDF graph that are connected by paths that conform to **p**.

In its simplest form, a SPARQL query has the form "SELECT $\mathbf{V}$ WHERE P", with $P = \{t_1, \ldots, t_n\}$ being a set of triple patterns (BGP). Optionally, one or more FILTER clauses further constrain the variables in P. Let $\mathcal{X}_P$ denote the finite set of variables occurring in P, i.e., $\mathcal{X}_P \subset \mathcal{X}$ [63], then $\mathbf{V}$ is the vector of variables returned by the query such that $\mathbf{V} \subseteq \mathcal{X}_P$. Additional operators such as UNION or OPTIONAL allow more than one BGP in a query by defining the non-conjunctive semantics of their combination. Finally, SPARQL queries can also make use of GROUP BY and aggregate operators.

SPARQL queries are declarative and are therefore designed to be decoupled from the physical data access methods[2] used to retrieve the data. This decoupling allows specific triplestore implementations to use different data representations and query processing designs to dynamically match an appropriate execution plan with a given query. Furthermore, when answering a query, a single BGP can be decomposed into several component BGPs and subsequently recombined before or after each BGP is solved.

SPARQL queries, as traditional queries, can also specify how to change the content of the graph. In this case, the query can either list a set of new RDF triples to be inserted into the graph or a set of triples to be deleted from the graph.

3 Access patterns

To design a performant RDF triplestore, it is of crucial importance not only to understand the type of information that it will store but also how this information will be queried, i.e., the expected query workload. Specifically, SPARQL queries, decomposed into BGPs and their associated triple patterns, access the data representation in different ways. In this section, we describe how SPARQL queries and their constituent BGPs can be analyzed to identify standard access patterns.

Definition 4 *(Access Pattern)* An access pattern is the set of logical operations that, given a set of variable bindings over a graph, determine what data to access (and optionally to change) and what output to produce.

Note that the term is used differently by relational model [27] analyses, where, given a relation, it only refers to what data is required as input and what tuples of the relation to produce as output. Moreover, this concept also differs from the concept of *access paths*, which refers instead to alternative data structures that can be navigated to reach the desired data [72]. The need for satisfying the requirements of these access patterns guides the selection and design of appropriate data representations along the design space dimensions defined in the following section.

Given a query and a specific BGP from the query, the access pattern is determined by the triple patterns in the BGP and any additional operators assigned to it from the SPARQL query, e.g., filters, grouping, or aggregations. Here, we identify the *feature space of access patterns* (summarized in Table 1). This feature space is comprised of six dimensions (each dimension containing a set of alternative features), namely:

- **Constants** the presence of constant values (as opposed to variables which have bindings).
- **Filter** The presence (or absence) of a filter on a range of values.
- **Traversal** The complexity and type of the traversal described by different triple patterns of the BGP.
- **Pivot** How different triple patterns of the BGP are linked together by some common atom in the BGP (here called a *pivot*).
- **Return** The information expected to be returned.
- **Write** Whether and how the BGP causes a change in the contents of the database.

Constants A common feature of any BGP is the presence of variables in one or more of the subject (s), predicate (p), or object (o) positions (see Definition 2). The presence of a variable requires finding all the triples that match the remaining (non-variable, hence constant) positions. Therefore, since the triplestore must find all triples that contain the same constants in those positions, their presence in the BGP provides higher selectivity that can be exploited by filtering and indexing schemes. In particular, a triple pattern that is *fully instantiated* translates into an existence clause for a single specific triple. *Partially instantiated* BGPs are those where some of the (s, p, o) positions are expressed as variables. Finally, when all three (s, p, o) positions are variables, we have an *uninstantiated* access pattern which matches all the triples in the database.

Filter Filters with conditions limiting the values that can be bound to one or more variables within a specific subset. Since filter operators have proper semantics defined for literals, and since literals can only appear as objects in a triple, filter operators retain triples based on the value assumed by a variable in the object position. Filter operations usually define open intervals (e.g., $?o > 10$), but when combined the query engine can translate them into closed intervals (e.g., $10 > ?o > 100$). Moreover, for SPARQL, a special type of filters depends on the type of the object. In practice, a query could return only values that are all literals or all IRIs, or all blank nodes, i.e., a filter based on their membership to $\mathcal{L}$, $\mathcal{I}$,

2 Access methods are low-level algorithms and data structures for organizing and accessing data [37].

Table 1 Feature space of access patterns for a SPARQL query

Dimension	**Features**		
Constants	*S, P, O* *1 Constant*	*SP, SO, PO* *2 Constants*	*SPO* *3 Constants*
Filter	<&> *Closed range*	<|> *Open range*	$\mathbb{T}$ *Special type range*
Traversal	$s{\to}o$ *1-hop over p*	$s \xrightarrow{k} o$ *k-hop over* $p_1, \ldots, p_k$	$p^{*/+}$ *?-hop over p*
Pivot	$s{\equiv}s$ / $o{\equiv}o$ / $p{\equiv}p$ binary on same S/P/O position	$o{\equiv}s$ / $o{\equiv}p$ / $s{\equiv}p$ binary on different S/P/O positions	$v_0{\equiv}v_1 \wedge v_0{\equiv}v_2 \wedge ...v_0{\equiv}v_N$ N-way on arbitrary S/P/O pivot positions
Return	Values (all) Values (distinct)	Values (sorted) Existence	Aggregate
Write	Update	Insert	Delete

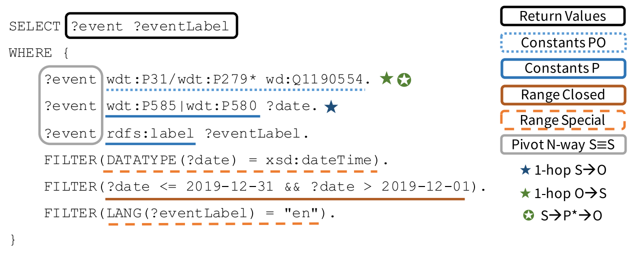
or $\mathcal{B}$. For this purpose, SPARQL has special native operators `isLiteral`, `isBlank`, and `isURI`. Similarly, RDF literal strings can be annotated with language tags allowing a query to filter based on those tags. For instance, to distinguish that the string "Rome" is in French and not in English, RDF represents it as `"Rome"@fr` instead of `"Rome"@en`. Hence, a query could select only strings marked as `@fr`.

Traversal Traversal types determine the number of triples that need to be traversed or considered, substantially impacting the query's complexity. We identify three cases, namely (i) 1-hop traversal $s{\to}o$ for a given specific predicate p; (ii) k-hops for a given sequence of predicates $p_1, \ldots, p_k$; and (iii) a path for an unconstrained number of hops over some predicate p. Excluding the simple 1-hop, the other traversals are usually expressed by (and referred to as) *property paths* [62]. In all cases, a traversal can go in either direction (given a source, find targets, or given a target, find all sources). Moreover, in traversals with more than one hop, the intermediate nodes are not required to be returned.

Pivot Related to traversals, an essential feature of a BGP is the ability to connect multiple triple patterns in other structural forms than a sequence. When an atom appears in two or more different triple patterns of the BGP, we refer to it as a *pivot*. We note that the pivot can appear in the same position, e.g., always in the subject position, or in different positions, e.g., subject of one triple pattern and object in another. This feature effectively determines the topology of the graph structure described by the query and can serve as the building block for more complex structures such as stars and snow-flakes [16]. We use the term *pivot* and not *join* to separate between the access pattern required (pivot) and its physical instantiation (e.g., a join operation), which is dependent on the data representation. Some specific data representations can speed up pivot operations, avoiding relational table join-type operations altogether. In particular, we distinguish between a pivot involving only two triple patterns and a pivot where the same variable is shared between more than two triple patterns, e.g., in a star pattern [77]. In the latter case, recent studies have shown the advantage in employing worst-case optimal join algorithms [38] where multiple triple patterns sharing a pivot can be evaluated altogether. Moreover, while a pivot over the object of one triple and the subject of another can be seen as a 2-hop traversal, in general, one cannot categorically say that using two pivots is better than a 2-hop reachability index. Thus, in our analysis, these features are considered separately.

Return The solution to a SPARQL query is the set (or a subset) of the *bindings* of atoms (from the matching triples) to some variables. Given a BGP, most queries return all variable bindings that match the triple pattern. Yet, in other cases not all matching triples and variable bindings are directly returned in the query result. When only a subset of the variables are returned, this may cause duplicate values to be returned, representing all of the triples found to match the BGP. We name this case *Values (all)* since the query returns values and all duplicates are returned (Table 1). When the `DISTINCT` keyword is used, only the set of distinct bindings for each variable is required. This case is named *Values (distinct)*. Moreover, some queries could require the bindings for some variable to be returned in some predefined order. We refer to this case as *Values (sorted)*. For some queries, it is sufficient to identify whether a variable binding *exists* satisfying the pattern, but we are not required to return the binding. In some other cases, a query just needs to verify whether a specific path exists or vice versa; when a specific connection does not exist, this is also addressed by a negation on an existence check. SPARQL `ASK` queries are an extreme example where the entire query returns no variable bindings but only the value true (exists) or false (does not exists) for a specific BGP. Finally, a query could be required to return aggregated values, i.e., the count of matchings for a BGP or max and min values for literals bindings.

Write SPARQL allows not only to retrieve information from the database but also to modify its contents, i.e., *write* operations. Note that update operations are often unsupported, requiring a deletion and subsequent insertion of new



```
SELECT ?event ?eventLabel
WHERE {
    ?event wdt:P31/wdt:P279* wd:Q1190554. ★ ✪
    ?event wdt:P585|wdt:P580 ?date. ★
    ?event rdfs:label ?eventLabel.
    FILTER(DATATYPE(?date) = xsd:dateTime).
    FILTER(?date <= 2019-12-31 && ?date > 2019-12-01).
    FILTER(LANG(?eventLabel) = "en").
}
```

Fig. 2 SPARQL query with annotated access patterns

triples instead. We consider these access patterns to distinguish the case of read-only workloads from read-write workloads.

Example query analysis Consider the example in Fig. 2 (from a WikiData query log [16]) retrieving location coordinates of archaeological sites. Property path `wdt:P31/wdt:P279* wd:Q1190554` is of the form `p1 / p2* o`, meaning that it would match two alternative paths: (1) 1-hop traversals over `p1 = wdt:P31` reaching the target node `o=wd:Q1190554` directly, and (2) *-hop traversals starting with one edge for `wdt:P31` and reaching the object through a sequence of arbitrarily long paths matching the `p2 = wdt:P279` triple pattern. Hence, the query contains two 1-hop traversals (marked with stars, one in each direction, since `wdt:P279*` is optional) and a composite property path (`wdt:P31/wdt:P279*`, see Definition 3). It contains a triple pattern with constants in both (p) and (o) (highlighted in dashed blue) and two triple patterns with a constant only in (p) (in solid blue). Moreover, the query contains both a closed range filter and two different special filters: one for the `DATATYPE` and one for the `LANG` property of literals. Finally, the `?event` variable is a 3-way pivot, which can also be executed as a set of binary $s \equiv s$ pivot access patterns.

Therefore, this feature space allows us to characterize each query with the requirements of the corresponding *access patterns* needed to answer it. In the following, we define a design space for data representations. Decisions for the different dimensions within the design space impact how efficiently the query's access patterns are supported by the resulting data representations.

4 A design space for RDF data representations

Triplestores implement a set of data representations that support the data access patterns needed to solve the BGPs as described in the previous section. A *data representation* stores a subset of the graph $G \subseteq \mathcal{G}$. Given an access pattern comprised of a BGP P, a vector of variables to return $\mathbf{V}$, and optionally filter and aggregation operations over these variables O, a data representation to answer the given access pattern needs to provide a way to retrieve the values of $\mathbf{V}$ from the information stored in G or to modify G accordingly to $\mathbf{V}$. Therefore, given an access pattern $\mathcal{A}$ and a data representation $\mathcal{D}$, if $\mathcal{D}$ holds the information necessary to answer the access pattern $\mathcal{A}$, we want to evaluate the performance of $\mathcal{D}$ to provide the correct instantiations of $\mathbf{V}$ given P and O. Thus, the question is what is the cost, in terms of time, needed to compute the instantiations of $\mathbf{V}$ and execute $\mathcal{A}$ over $\mathcal{D}$. Answering such a question allows, given two distinct data representations, to select the most appropriate one.

In this section, we begin by defining a basic cost model (Sect. 4.1) and the notion of compatibility between access patterns and data representations (Sect. 4.2). Then, we propose a design space for data representations over which design decisions can be made (Sect. 4.3). Together, these allow the evaluation of fit between an access pattern and a design choice which is later showcased in Sect. 6.

4.1 Cost model

To answer a BGP using a particular data representation incurs a cost usually expressed in terms of the time it takes to retrieve all such answers. Estimating this cost for the different access patterns is crucial, both at design time and during query processing, and relies on a cost model assigning a cost to each type of basic operation. Several such models have been proposed for RDF [21,29,66], but were biased by a simple two-tiered memory hierarchy and underlying a relational data representation. In an era of shared memory clusters, large-RAM machines, SSD, NVRAM, SIMD, and vectorized processing (e.g., [51]), assuming a single cost relation paradigm is no longer sensible.

What still holds true over all novel (existing or future) machine architectures and memory types is the fact that random seek operations incur a different cost than sequential read operations. As echoed by the authors of the RUM conjecture [9], ".. in the 1970s one of the critical aspects of every database algorithm was to minimize the number of random accesses on disk; fast-forward 40 years and a similar strategy is still used, only now we minimize the number of random accesses to main memory ". The use of compression to minimize memory requirements and speedup retrieval of large result sets incurs additional costs in compression and decompression times. Although novel compression methods utilizing hardware to speedup these times are increasingly available (e.g., [6]), there remains a difference between compressed and uncompressed data. We, therefore, follow the recent convention of employing a set of cost constants [12,41] differentiating between random and sequential and between compressed and uncompressed costs. The instantiation of these constants for the different representation options on the current hardware can be done at design time by measuring operation times on the current configuration. We use these constants to induce a ranking over different representations in the same design space dimension.

Definition 5 *(Basic Operation Cost Constants) Read random* (R_r) represents the average cost of accessing a random single data item (e.g., a key or a value) in a data representation given a pointer to the item. *Read sequential* (R_s), instead, measures the cost of reading a data item stored in a position succeeding the current one, i.e., stored sequentially within a contiguous region of memory (e.g., the next block on disk, or the next position in an array). For compressed data, we distinguish between *read compressed sequential* (R_{cs}) and *read uncompressed sequential* (R_{us}). Similarly defined constants for write operations are W_r, W_s , W_{cs}, and W_{us}, respectively.

4.2 Data representation compatibility

To assess whether a data representation can efficiently support a specific access pattern, we define the notion of *compatibility*.[3] Therefore, given an access pattern $\mathcal{A}$ and a data representation $\mathcal{D}$ able to answer $\mathcal{A}$, we assume that there is a sequence of operations specified by an algorithm Γ defined over $\mathcal{D}$ that can compute such an answer $\mathcal{S}$, i.e., $\mathcal{S} = \Gamma(\mathcal{A}, \mathcal{D})$. The number and cost of these operations specified by Γ directly determine whether $\mathcal{D}$ is a representation suitable for computing efficiently the answers to $\mathcal{A}$. When measuring the number of operations required to be executed over $\mathcal{D}$, we distinguish between random seek and sequential operations.[4] In particular, an RDF data representation can be *seek compatible* or *sequence compatible*, defined as follows.

A data representation is *seek compatible* with an access pattern if one or more of the results required by the access pattern can be retrieved in a single random access step. For example, if the access pattern requires retrieving the objects that are related to the same subject $s1$ via the same predicate $p1$, i.e., $\langle s1,\ p1,\ ?o\rangle$, a seek compatible representation is one that, given the pair of values $\langle s1,\ p1\rangle$, returns a pointer to the first element of this set (e.g., a hash table).

A representation is *sequence compatible* if all results required by the access pattern can be retrieved through sequential accesses without requiring (after the initial seek) any additional random seek to complete the result set. In the case of a hash table for $\langle s1,\ p1,\ ?o\rangle$, if the pointer returned from the first seek is to a contiguous area of memory/disk containing all objects satisfying the access pattern, the representation is sequence compatible. However, if the objects are stored in a linked list, requiring additional random seeks to read, the representation is not sequence compatible. Moreover, we say that a representation is *selection compatible* with an access pattern if no unneeded results are retrieved. For example, if a variable ?o in our example is restricted to literals (with `isLiteral`), then any triple with a IRI as object is unneeded.

Definition 6 *(Compatibility)* Let $\mathcal{A}$ be an access pattern and let $\mathcal{S}$ be the set of results that are the answer to $\mathcal{A}$ over a graph $\mathcal{G}$. Let $\mathcal{D}$ be a data representation of $\mathcal{G}$. $\mathcal{S}$ is a, possibly empty, set of tuples containing any combination of literals, blank nodes, and IRIs. Then, $C^r_1(\mathcal{A}|\mathcal{D})$ is the number of random seek operations required to reach the first result of $\mathcal{S}$ in $\mathcal{D}$ or to ascertain that $\mathcal{S} \equiv \emptyset$. We define $C^r_\Omega(\mathcal{A}|\mathcal{D})$ to be the cost in terms of number of random seek operations required to retrieve all results in $\mathcal{S}$ from $\mathcal{D}$ after having reached the first result in $\mathcal{S}$. Correspondingly, $C^s_\Omega(\mathcal{A}|\mathcal{D})$ is the number of sequential read operations required to retrieve all subsequent results. Hence, the total cost to retrieve the results $\mathcal{S}$ of $\mathcal{A}$ over $\mathcal{D}$ is:

$$C^r_1(\mathcal{A}|\mathcal{D})\times R_r + C^r_\Omega(\mathcal{A}|\mathcal{D})\times R_r + C^s_\Omega(\mathcal{A}|\mathcal{D})\times R_s. \tag{1}$$

Hence, we say that a data representation $\mathcal{D}$ is:

- *seek compatible* with $\mathcal{A}$ if $C^r_1(\mathcal{A}|\mathcal{D}) = \varepsilon$, where $\varepsilon\in\mathbb{R}^+$ is some small system-dependent constant independent of $|\mathcal{G}|$ (e.g., if $\mathcal{D}$ is a hash-table the fixed cost to search elements in $\mathcal{D}$ is usually approximated to the constant 1.2).
- *sequence compatible* with $\mathcal{A}$ if $\mathcal{S}$ can be sequentially retrieved after the initial seek, that is $C^r_\Omega(\mathcal{A}|\mathcal{D}) = 0$ and $C^s_\Omega(\mathcal{A}|\mathcal{D}) \leq \max(|G|, |\mathcal{S}|)$.
- *selection compatible* with $\mathcal{A}$ if no excess results are retrieved, that is $C^r_\Omega(\mathcal{A}|\mathcal{D}) + C^s_\Omega(\mathcal{A}|\mathcal{D}) \leq |\mathcal{S}|$.

Example 2 (Compatibility of different representations) For instance, one way to store $\mathcal{G}$ is to represent each triple as a 3-tuple, and the entire dataset as a list of 3-tuples sorted by subject and then by predicate and object (Fig. 3a and d below) with a clustered B+ tree index over them. In this representation, the cost of query processing would resemble that of a relational table with three attributes (s, p, o), all part of a primary index. This representation is sequence compatible with any 1-hop access pattern that binds s, both s and p, or all three positions. That is, the algorithm Γ(*⟨s1, p1, ?o⟩, sorted(G);B+ tree*) would first find the first tuple performing $log(G)$ steps traversing the B+ tree looking for *s1, p1* and then perform a linear scan over the file to retrieve the remaining tuples. However, the B+ tree is not seek compat-

[3] We define this notion for read operations only under the assumption that a read operation is required before deletion (to locate the data to be deleted) and before insertion (to locate where to insert the data and to avoid duplicates) and is therefore required for every type of access pattern. Following such a read operation, the set of pointers returned can be used to perform an update, deletion, or insertion.

[4] In the following, for ease of presentation, we refer to a simplified cost model where we consider the amortized average cost of the data structures, e.g., hash maps access methods have average constant cost.

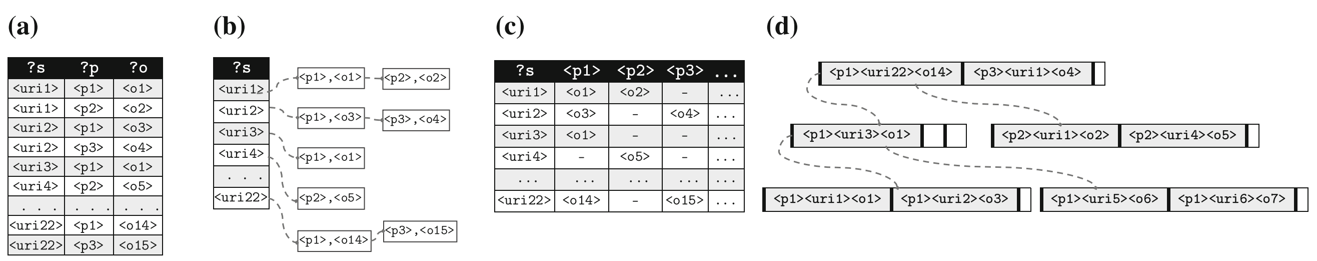


Fig. 3 Examples of data representations: **a** sorted file, **b** hash map, **c** property table, and **d** B+ tree

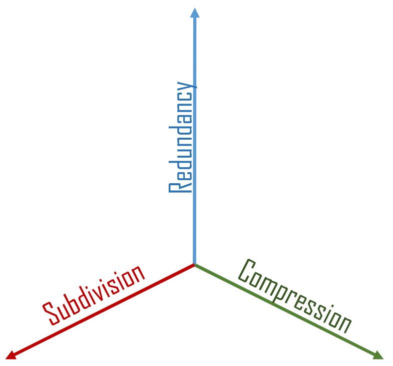


Fig. 4 The SCR system design space for RDF stores

ible, because the time to find the first result depends on the size of the graph, i.e., it involves $log(|G|)$ seek operations to reach the first result.

A different data representation is to employ a key-value data structure (similar to Fig. 3b) and use the pair subject-predicate as the key, and the object as the value. In this data structure, triples sharing the same s and p will store the list of objects contiguously. This data structure is both seek and sequence compatible for a traversal that, given s and p, retrieves all corresponding objects. Nevertheless, this representation is neither seek compatible nor sequence compatible if the query requires all edges for predicate p regardless of s.

In the following, we present a design space within which each data representation can be embedded. The definitions of cost and compatibility presented above allow to analyze the advantages and drawbacks of the different choices in each dimension of the design space.

4.3 The design space dimensions

When designing data representations for an RDF store, one can model the design decisions over three axes: *Subdivision*, *Compression*, and *Redundancy* (SCR, Fig. 4). Each of these orthogonal axes, whose properties are summarized in Table 2, represents a continuum along which a system can be positioned.

Subdivision The *subdivision* axis determines how fragmented the data is. At one extreme, all data is stored contiguously in a single structure; at the other extreme, each edge and node is stored as a separate object with pointers to its neighbors in the induced graph. In between there are various data structures such as B+ trees or hash maps adapted to these settings. This axis contains design decisions such as sorting, grouping, and hashing. Each of these decisions creates an additional subdivision in the data. Increasing the extent of subdivision allows us to minimize the number of *unneeded* data items accessed to answer an access pattern (yielding fewer R_s and W_s operations that return items not in $\mathcal{S}$). For example, when using the single file approach, we would potentially need to read the entire file before finding a single required triple, while with a hash table, we move directly to the first matching tuple. Filter access patterns on large ranges of values, however, can be costly when data representations utilize extensive subdivision. The use of multi-core parallelized processing can ameliorate this cost, to some extent, by dividing the sequential retrieval tasks among cores that access the subdivided data in parallel. Subdivided data representations provide an additional benefit for multi-core systems, as they allow the creation of locking mechanisms with finer granularity, reducing wait times. Consider Fig. 3. Note that, in a sorted file (Fig. 3a), the sorting keys act as the simplest subdivision by collecting all triples of the same value together. On the other hand, with a hash table (Fig. 3b), given the target IRI, the hash function separated all relevant triples sharing the same key. Note that every time we move across subdivisions, we move into a different non-contiguous region in either disk or memory, increasing the R_r cost. Therefore, decisions across the subdivision axis easily determine whether a data structure is selection compatible, i.e., if it bounds all and only the answers within a specific subdivision, but it also impact random and sequence compatibility.

Compression The second axis is the *compression* axis. The goal of compression is to minimize the number of bits read to reach the first tuple in the result set (R_r) and the number of bits required to read and potentially store the result set for further processing R_{cs}. The potentially negative impact of compression is, of course, the decompression required to evaluate a predicate in the access pattern if the

Table 2 Summary of data representation design space axes

Axis	Minimal Extreme	Maximal Extreme	Positive Effect	Negative Effect
Subdivision	Unsorted file	Pointers between every related data item	$\downarrow$ # unneeded data items	$\uparrow$ # random seeks
Compression	No compression	Compress all subdivisions in all representations	$\downarrow$ # read bytes	$\uparrow$ Decompression cost
Redundancy	Single representation	One representation for each possible BGP	$\downarrow$ # random seeks	$\uparrow$ maintenance cost, storage cost, query optimization time

access pattern is incompatible or partially incompatible. For example, consider a compressed data representation tuned for queries of the form $(s, ?p, ?o)$ and $s{\equiv}s$ access patterns (e.g., BitMat [11]). Using this representation to answer an s $p^{*/+}$ o pattern (i.e., is o reachable from s through a edges labeled with p) would require a potentially large number of row decompression operations to perform the $o{\equiv}s$ pivot operations required resolve the traversal. Since these may be spread around the data structure, this would incur a large number of R_r operations. Therefore, decisions across the compression axis impact heavily the selection compatibility when data that does not contribute to the answer are compressed together with data relevant for an access pattern.

Redundancy The third axis is the *redundancy axis* which causes (redundant) copies of the data to be stored in the system. By adding redundant data representations and indexes, it is possible to define ideal (seek, sequence, and selection compatible) data representations for each access pattern. However, this comes at the cost of having to store the same information multiple times. For example, by holding both an SPO clustered index and a PSO clustered index, each triple is stored twice. Thus, this hinders the compatibility with write access patterns. Moreover, design decisions including full/partial replication need to find a trade-off between storage space and efficient support of query access patterns. Hence, the cost of maintaining multiple representations is threefold: (i) Increased latency for delete and insert operations (higher W_r and W_s) with possibly reduced performance of read operations (higher R_{us}) as well, since additional (uncompressed) auxiliary data structures are required to store deltas until the cost of updating can be amortized over a large enough set of updates (e.g., as in RDF-3X [55]). The use of multi-core parallelized processing can avoid this additional cost, to some extent, by dividing the redundant update tasks among cores in parallel, allowing for more rapid update of the compressed structures. (ii) An increase in space requirements, subsequently straining the limited space in main memory and causing an increase in all R costs. (iii) An increase in query optimization time because alternative structures to access the data result in a higher number of query execution plans that have to be considered.

Example 3 (SCR Space analysis.) To illustrate the connection between the SCR space and the proposed cost model, consider again the different data representations in Fig. 3. A sorted file (Fig. 3a) divides the triples (cardinality $|\mathcal{G}|$) by the values of the first sort field (e.g., s) and eventually by all remaining sort key fields. Finding the first tuple with some specific value for s requires $\log_2(|\mathcal{G}|)$ random seeks. Subsequently, reading all the relevant records (assuming a cardinality of $|\mathcal{S}|$) is now a sequential read. The final total cost for the *Constant s* access pattern is then $R_r \times \log_2(|\mathcal{G}|) + R_s \times |\mathcal{S}|$. Hash tables (Fig. 3b) improve upon this, i.e., moves further along the Subdivision axis, by subdividing the space into buckets with the same key value. Then, all keys with the same value are divided into a linked list of blocks of predefined maximum size (k, with $k = 1$ in the figure). Therefore, this incurs a random seek cost to reach the first record and then sequentially read the whole bucket, and ($m = \lceil |\mathcal{S}|/k \rceil$) random seeks to move from bucket to bucket, with a total cost of $R_r \times (1+m) + R_s \times |\mathcal{S}|$. While random seek costs often dominate sequential ones by orders of magnitude, in the average case of small answer set $\mathcal{S}$, this additional subdivision improves read costs for this access pattern. Deploying a B+ tree, instead, would keep the same Subdivision, but move across the Redundancy axis.

We now identify these options in existing RDF triplestores and place them within the SCR space.

5 Data representations in RDF triplestores

We now review a wide range of existing triplestores and the data representations they employ within the SCR design space. Here, we focus on centralized systems[5] that have either been published as research prototypes or commercially available systems. Our inclusion principle is a system, which allows the ingestion of RDF data and supports the

[5] We specifically exclude distributed and cloud-based solutions as the need for distributing and replicating data between nodes skews the system's cost model and is a mostly orthogonal decision to that of choosing which core data representations to use. Some distributed systems (e.g., 4-store [34] and Partout [30]), directly extend centralized systems.

SPARQL query language (including insertions and deletions). We exclude systems for which we could not find sufficiently detailed information regarding their data representation. Notable commercial exclusions are, therefore, Amazon Neptune [13], AllegroGraph [28], and StarDog [74], which do not disclose their internals. Notable non-commercial exclusions are HexaStore [81] and RDFBroker [73], which do not support SPARQL, and KAON2 [78], which is designed for OWL reasoning rather than SPARQL answering over RDF triples. Moreover, we also exclude HDT [25], which only supports data serialization.

Tables 3 and 4 summarize the features of the reviewed systems over the design space dimensions. Different triplestores can now be compared based on the choices they have made on how to subdivide the data (Subdivision), whether and how to compress IDs, literals and triples (Compression), and which redundant representations to maintain (Redundancy). Table 3 lists systems based on an underlying relational database system (RDBMS), and Table 4 lists systems using a *native graph storage*. The year stated next to the system name represents the publication year of the latest paper or technical report describing its features. It seems that recent solutions favor a native storage mechanism over using an RDBMS. Furthermore, the use of B+trees is becoming less prevalent, with recent solutions favoring hash-based solutions. We now present a classification of the systems into the different choices in each dimension.

5.1 Subdivision

Table 5 summarizes the design choices over the subdivision dimension. Recall that subdivision aims to minimize the number of unneeded items read when seeking and reading. This minimization may come at the cost of increasing the number of random seeks required to reach the data items needed by the access pattern. Within the subdivision dimension, we identified four choices which system designers can make: (1) how to subdivide the main triple data, (2) whether and how to divide the ID space assigned to IRI/literals, (3) how to subdivide the IRI/Literal→ID data representation, and (4) how to subdivide the reverse ID→IRI representation. Systems in the table are divided into four groups according to their approach toward subdividing the main triple data. The largest group of systems utilizes an underlying relational representation, either a column-based (SW-store [1]) or a row-based one (the rest). The 2nd-largest group utilizes tree-based representations. While Mulgara [54] uses an AVL tree, the rest use a B+tree. There are four systems utilizing a hash-based representation and four that opted for more specialized representations. We now review the systems in Table 5 by each of the main design choice categories.

Main triple data When reusing the underlying infrastructure and technologies of relational databases, designers must define how the RDF structure is mapped into a relational structure. 3store [33], RDFLib [46], and Virtuoso [24] use a large triple table with a field for each s,p,o atom together with some auxiliary indexes. Other relational-based systems use dynamic subdivision to create a set of relational tables.

Dynamic subdivision restructures the data representation according to the specific contents of the graph, its schema/ontology, or the query load.[6] SW-Store [1] subdivides the triples by creating a collection of property tables (one for each predicate IRI), thus adding some compression as the predicate value is encoded in the name of the subdivision rather than in each triple. However, the number of distinct predicates may be extremely large, causing a large schematic overhead. DB2 Graph [17], RDFBroker [73], and FlexTable [80] create tables for groups of predicates. DB2 Graph uses a fixed number of predicate columns populated according to their prevalence. RDFBroker creates a different table for each combination of predicates. FlexTable [80] allows a table to have different predicate columns on the data page level, thus dynamically opening new table pages with a slightly different schema when such instances appear in the data. 3XL [48] use the backing ontology of the triplestore to create complex tables containing related information from several triples.

Tree-based representations index triples using B+trees as entry point. Jena(TDB) [58] and RDF3X [55] are the most notable in this category. Trident [76] follows the relational subdivision but then concatenates all such subdivisions and exploits a B+tree on IDs during search. Hash-based solutions differ on the method in which the hash key is computed and on the organization of the buckets to which the key points. TripleT [82] and Oracle Spatial and Graph [84] use multiple hash-tables for different permutations of s, p, o, such that one of the positions serves as the hash key and the other two positions are stored in the buckets. gStore [88] hashes the IRI and holds its adjacency list in the value as well as a secondary tree-based construct (see Sect. 5.3). Chameleon [5] utilizes a hash table from IRI to a subdivision of the graph. Subdivisions are restructured dynamically according to the query workload.

We also report on four systems that utilize instead special representations. TripleBit [87] and BitMat [11] subdivide the data by predicate thus creating a bit-matrix for each predicate with rows representing subjects and columns representing objects. Parliament [45] uses a sorted file with offset pointers from the triple to the next one with the same subject to allow traversal. Effectively, this divides the triples into variable length blocks of same-value parts. To navigate between these blocks, Parliament uses either the offset pointers or

[6] We only mention mechanisms beyond auto-balancing algorithms prevalent in B+tree structures that are based on the number of keys in each subdivision.

Table 3 RDBMS-based Stores. <Prefix>#: representations that provide the number of triples with a given prefix

System (Year)	Subdivision	Compression	Redundancy
3store (2003) [33]	Different string tables for resources and literals.	int64 ids, strings in separate tables	SPO table. Literal flag on O. Inferred Flag on O. B+Tree index on string tables.
RDFLib with SQLAlchemy backend (2006) [46]	B+ tree over SPO table, separate triples by literal/Resource O.	None	B+ tree over SPO, and S, P, O
RDFBroker (2006) [73]	Set of S→PO tables, clustering S with same set of P	None	None
SW-Store (2009) [1]	IRI↔id encoding table, SO table for each value of P	Per-table column compression, bitmap O index in low-cardinality SO tables	IRI↔id B+ trees over encoding table, B+ tree on S for each SO table
FlexTable (2010) [80]	Dynamic split of SPO tables by groups of S sharing the same Ps	None	None, only SPO
Virtuoso (2010) [24]	IRI↔id using 2 hash tables, B+ tree over triple table	int64 ID for S & P, IRI/Literal→ID for O's $>$ 12 byte, sparse bitmap indexes, triple page compression	SPO, OPS, POS
3XL (2011) [48]	Separate non-ontological triples from those described by the ontology into a B+ tree indexed SPO table for the former and an ontologically induced schema containing multiple tables for the latter. ID→IRI in a B+ tree indexed table.	int64 ids	IRI→(ID,table) hash table.
DB2 Graph (2013) [17]	B+ tree over triple tables	hashing PO/PS to fixed number of columns	SPO, OPS fixed number of P tables with B+ tree index over S/O, SP→O, OP→S B+Tree indexed tables
Oracle Spatial & Graph (2014) [20, 84]	Separate schema from data triples, key-value hash tables	Hash-encoded keys	Triple tables; P→SO, P→OS, S→OP, O→PS in Key-Value Store

$X \rightarrow Y$ represents a hash/dictionary from key X to value(s) Y

binary search. YARS [36] augments this approach by subdividing the sorted file into blocks accessible via a sorted sparse index.

Recently, several data representations have been suggested specifically for the support of worst-case-optimal join (WCOJ) algorithms since those have been shown to be particularly efficient for processing n-way pivot access patterns. Although different implementations of these join algorithm exist [8,18], most have been implemented as extensions to existing triplestore systems and often they are both read-only and provide limited support of the SPARQL standard. The most extensive implementation is provided in the Jena framework to support the Leapfrog Triejoin [38]. Here, this n-way pivot access pattern is supported through a fully-redundant representation that adds three additional SPO permutation indexes to the default indexes in Jena (TDB). Moreover, it utilizes the prefix-sorted subdivision of a B+tree to provide support for an implementation of the WCOJ algorithm that exploits the standard single-position pivot access patterns. On the other hand, an extreme case of non-redundant subdivision instead exploits a ring data structure comprised of a bidirectional cyclic suffix-strings index that allows the retrieval of any permutation of SPO triples [8]. This representation is highly subdivided and incurs a substantial number of seek operations. To minimize the impact of these seeks, the authors propose an extensive compression mechanism allowing to maintain the structure in memory and thus minimize the seek cost. However, as mentioned above, this comes at the expense of allowing this structure to support additional access patterns, most notably write and delete patterns.

ID Space Most systems do not to subdivide the ID space. Of those that do, most subdivide the space between IRI and literals by assigning different ID ranges rather than storing them separately as only 3store [33] and Blazegraph [49] do.

Oracle [84] and BitMat [11] present unique choices for separating values with different roles. Oracle separates between IDs assigned to IRI of classes and properties from those assigned to other resources. BitMat separates between resources used as properties, as subjects only, as objects only, or either as subjects or objects.

IRI/Literal→ID & reverse ID→IRI For systems that choose to replace IRI/literals with IDs, the mapping mecha-

Table 4 Non-RDBMS Stores. <Prefix># representations that provide the number of triples with a given prefix

System (Year)	Subdivision	Compression	Redundancy
Mulgara (2006) [54]	IRI→ID AVL-Tree, B+ tree over triples	None	SPO, POS, OSP, IRI/Literal file
YARS2 (2007) [36]	Sparse index over triple files	Blocks compressed with Huffman coding	SPO, SOP, OSP, OPS, PSO, POS index-file pairs
Apache Jena with TDB (2009) [58]	ID→IRI hash table, IRI→ID is B+ tree, short literals not separated	int64 ids	IRI→id, SPO, POS, OPS. Hogan et al. [38] add *OSP, SOP, PSO*.
BitMat (2009) [11]	SO matrix for each P, separate id-space into SO, S-SO, P, & O-SO	Sparse bit cube with D-gap compression	PSO bit-cube
Parliament (2009) [45]	Resource table (ID, 3 traversal pointers, IRI/Literal pointer to file), IRI→iD B+ tree, Triple table (SPO, 3 traversal pointers, bit flags)	None	Triple counts per resource, IRI/Literal file (second representation to support rapid lookup from resource table)
RDF3X (2010) [55]	IRI→ID B+ tree , B+ tree for triples	ID compression in B+ tree leaves	SPO, SOP, OSP, OPS, PSO, POS, SP#, PS#, SO#, OS#, PO#, OP#, P#, S#, O# B+ trees, ID→IRI direct mapping idx
GraphDB (2011, previously OWLIM [15])	Literal Index, B+ tree over triples	Unknown	POS and PSO at least, details omitted
Sparqling kleene (2013) [31]	Same as RDF-3X	Same as RDF-3X	Same as RDF-3X + Reachability indexes (S-*-O) for each value of P
Blazegraph (2013) [49] (previously systap bigdata)	BLOB index for large IRI and large literals, id↔IRI B+Trees, triple B+ trees	Front-coding of keys	SPO, OSP , and POS B+ trees
TripleBit (2013) [87]	IRI→ID 2 level hash table, ID→IRI sorted file, P-segmented triple tables + block index.	Variable sized ID: avg. 3 Byte, triples columns delta-compressed	P→SO, P→OS, O/S→P
TripleT (2015) [82]	IRI↔id using 2 hash tables, Partially sorted triple files with IRI-based B+ tree index over them.	int64 IDs	SO, SP , OP buckets for each ID, B+ tree Index contains pointer + triple counts per resource
gStore (2018) [88]	Triples in hash tables by edge/vertex	bit-string encoded adjacent edges and vertices	P→SO, S→PO, O→PS, OP→S, SP→O hash tables, VS*-tree for graph pattern matching (B+ tree of bit string values).
Chameleon (2019) [4, 5]	Dynamic triple partitioning with a Hash-Table from IRI to partition. IRI are in a dictionary while literals are inlined.	int64 ids for IRI, order-preserving compression of literals	IRI→Part index, range index P+Part→MinMaxVal
Trident (2020) [76]	IRI→ID using 2 B+tree, B+Tree pointing nodes to binary tables for 6 SPO permutations	Variable size IDs, run-length encoding of IDs	6 permutations indices of edge list with adaptive redundancy, SP#, PS#, SO#, OS#, PO#, OP#, updates are stored in differential indexes

nism is either B+ trees, which support range queries well, or hash tables, which are more efficient for single lookup.

5.2 Compression

Table 6 presents the triplestores' compression choices grouped by the choice for the main triple store. Most early systems (circa 2003–2009) do not compress the main data at all. A number of systems (RDF-3X [55], Sparqling kleene [31], Virtuoso [24], YARS2 [36]) employ block-level compression. This approach entails organizing the triples in memory blocks in a manner that allows compression using techniques such as Huffman encoding [39]. YARS2 [36] also employs sparse representations (also used by gStore [88] and Triple-T [82]) where the indexes contain only the used values of a position, rather than the full domain. SW-store [1] and TripleBit [87] both use some form of column-compression, where the values are stored by column, rather than by row,

Table 5 Subdivision design choices

	Triples		ID Space				IRI→ID			ID→IRI			
System	BDR	DS	None	S-D	U-L	SPO	None	BT	HT	None	SF	BT	HT
Relational representations													
3store [33]	RR				✓				✓			✓	
3XL [48]	RR	Ontology		✓	✓				✓			✓	
DB2 Graph [17]	RR	Data	✓				✓			✓			
FlexTable [80]	RR	Data	✓				✓			✓			
RDFBroker [73]	RR	Schema					✓			✓			
RDFLib [46]	RR				✓		✓			✓			
SW-store [1]	RC	Schema						✓				✓	
Virtuoso [24]	RR		✓						✓				✓
Tree-based representations													
BlazeGraph [49]	BT				✓			✓				✓	
GraphDB [15]	BT				✓		?	?	?	?	?	?	?
Jena(TDB) [58]	BT				✓			✓					✓
Mulgara [54]	AVL		✓					✓			✓		
RDF-3X [55]	BT		✓					✓					✓
SPARQL-kleene [31]	BT		✓					✓					✓
Trident [76]	RR/RC	Data				✓		✓				✓	
Hash-based representations													
Chameleon [5]	HT	Queries			✓				✓				✓
gStore [88]	HT				✓		✓			✓			
Oracle [84]	HT			✓					✓				✓
TripleT [82]	HT		✓						✓				✓
Matrix/Other representations													
BitMat [11]	MAT					✓			✓				✓
Parliament [45]	SF		✓					✓			✓		
TripleBit [87]	MAT		✓						✓		✓		
YARS2 [36]	SI		✓				✓			✓			

BDR Base data representation, *BT* B+tree, *AVL* AVL-tree, *RR* Relational row store, *RC* Relational column store, *HT* Hash table, *SF* Sorted file, *SI* Sparse index over sorted file, *MAT* Matrix, *DS* Dynamic subdivision, *S-D* Schema data, *U-L* IRI-Literal/Large literal, *SPO* IRI divide by their position (e.g., only as *p*)

allowing to skip empty rows, store only the difference from the previous value (delta-compression) or other column-compression methods. Trident [76] uses a combination of different compression mechanisms. In particular, triples are sorted in so-called binary tables, since they encode only two of the 3 SPO positions. The triples are also represented either row-wise or column-wise within different blocks in order to exploit run-length encoding and other types of compression when some values are repeated. The final three solutions are derived from the subdivision choices made by the system. Using a classic relational row-based representation, DB2 Graph [17] chose to limit the number of predicate columns in its table to a fixed K, this reduces the chances of empty columns in record rows and can be considered a form of compression. BitMat [11] employs row-based compression of its bit-matrix using similar methods as those employed for column-based compression, and Chameleon [5] utilizes order-preserving compression of its in-lined literals. We also note that the ring data structure employed to represent the graph via a set of suffix strings [8] is a way to compress the triples and to index all their SPO permutations, yet the current implementation for RDF is a read-only solution that requires further study.

A second design choice in the compression dimensions is whether to avoid storing IRIs and literal values as repeated strings and instead use numerical IDs. Of the 20 systems reviewed, only eight chose not to do so. 3XL [48] and Chameleon [5] replace only IRI with IDs. 3XL leave all literals in their original form and Chameleon compress the literals in place using order-preserving string compression. Virtuoso and Jena(TDB) [58] inline small literals and encode the larger ones. The rest of the systems replace all strings with IDs, regardless of their type.

ID compression is presented by increasing strength of compression. *Int* denotes the usage of sequential integers instead of the original string-based IRI/literal. More compression can be achieved by compressing the IDs into fixed-length integers (e.g., Virtuoso [24]). In an example of extreme compression, TripleBit [87] compress IDs into an average of 2–3 bytes by using variable-length compres-

Table 6 Compression design choices

	Triples						ID Replacement				ID Compression			
System	B-C	Sparse	C-C	N-Col	Cube	ISC	None	IRI	Long	All	None	Int	Fix	Var
Block-level compression of the main triple store														
RDF-3X [55]	✓									✓		✓		
Sparqling kleene [31]	✓									✓		✓		
Virtuoso [24]	✓								✓				✓	
YARS2 [36]	✓	✓					✓				✓			
Other compression methods of the main triple store														
gStore [88]		✓					✓				✓			
Triple-T [82]		✓								✓			✓	
SW-store [1]			✓							✓		✓		
TripleBit [87]			✓							✓				✓
DB2 Graph [17]				✓			✓				✓			
BitMat [11]					✓					✓		✓		
Chameleon [5]	✓		✓			✓		✓			✓		✓	
Trident [76]	✓		✓							✓				✓
No compression of the main triple store														
3store [33]										✓			✓	
BlazeGraph [49]										✓			✓	
Jena(TDB) [58]									✓				✓	
3XL [48]								✓				✓		
FlexTable [80]							✓				✓			
Mulgara [54]							✓				✓			
Parliament [45]							✓				✓			
RDFBroker [73]							✓				✓			
RDFLib [46]							✓				✓			

B-C: Block-level compression; Sparse: compressed sparse index; C-C: Column-level compression; N-Col: Fixed number of columns; Cube: Compressed bit-cube; ISC: Inlined string compression; IRI: Only replace IRI with ID; LS: Replace long strings only; All: replace all strings with integers; Int: a sequential integer (uncompressed, usually 8 bytes); Fix: ID compressed into a fixed-size integer (6 bytes); Var: variable-sized compressed integers (2–3 Bytes)

sion instead. Notable missing systems from this analysis are Oracle [84] and GraphDB [15] which do not provide details regarding this design choice in their publicly available documentation.

5.3 Redundancy

Table 7 presents a high-level summary of the implementations employing multiple redundant representations for the main triple data. On average, there are 3 redundant representations per system. Only one system (Sparqling kleene [31]) offers reachability indexes to answer triple patterns such as `s p* o` (i.e., property paths with kleene-start patterns, see Definition 3). Only one system (gStore [88]) offers a data representation tuned to matching an N-way same-position pivot (a star-shaped subgraph structure). gStore is also notable for being the only system employing a *Bloomier filter* construct [19] (the VS*-tree) as a secondary data representation. Bloom filters allow determining the existence of a data item using a compact representation with low latency at the expense of false positives but never false negatives. Bloomier filters [19] extend this capability to allow several functions, in this case returning the vertices matching a set of PO/SP patterns. The use of a PO/PS hash map as part of the VS*-tree lookup process is an almost singular example of using two-position combination hash maps (e.g., sp→o). Notably, a recent extension of the Jena (TDB) system to support a worst-case-optimal join algorithm [38] has added three additional B+-tree indexes to support this pattern. This results in a high level of redundancy of the data, which new approaches are trying to overcome with more compressed (and way less redundant) indexed representations [8]. Also, Trident [76] employs the replication of the six permutations of the SPO positions, but employs adaptive storage representation to reduce the memory footprint.

5.4 Summary

Table 8 summarizes the keys insights obtained while reviewing the surveyed systems as detailed above. In particular, we list recent trends in implemented design choices and unexplored data representations.

Dimensional interdependence As a final note, while the three SCR dimensions are orthogonal and independent for the most part, the surveyed space shows a few cases, where a design decision over one dimension impacts the availability of design decisions in others. This impact can either limit the availability of options or enable options that could not be chosen otherwise. For instance, choosing to replace URI with numerical IDs as a design choice over the compression

Table 7 Redundancy

	BT/AVL/Sparse trees						Hash Tables							Aggregates			Other			
System	spo	sop	pos	pso	ops	osp	V-spo	V-p	p-so	p-os	s-op	s-po	o-ps	TC	2VC	MM	SF	BC	VST	RI
Relational representations																				
3store [33]	✓																			
3XL [48]	✓							✓												
DB2 Graph [17]	✓				✓															
FlexTable [80]							✓													
RDFBroker [73]	✓																			
RDFLib [46]	✓																			
SW-store [1]				✓																
Trident [76]	✓	✓	✓	✓	✓	✓									✓					
Virtuoso [24]	✓		✓		✓															
Tree-based representations																				
BlazeGraph [49]	✓		✓			✓														
GraphDB [15]			✓	✓																
Jena(TDB) [58],[38]*	✓	*	✓	*	✓	*														
Mulgara [54]	✓		✓			✓														
RDF-3X [55]	✓	✓	✓	✓	✓	✓								✓	✓					
Sparqling kleene [31]	✓	✓	✓	✓	✓	✓								✓	✓					✓
YARS2 [36]	✓	✓	✓	✓	✓	✓														
Hash-based representations																				
Chameleon [5]							✓									✓				
gStore [88]									✓			✓	✓						✓	
Oracle [84]									✓	✓	✓		✓							
Triple-T [82]		✓		✓		✓								✓						
Matrix/Other representations																				
BitMat [11]																		✓		
Parliament [45]														✓			✓			
TripleBit [87]								✓	✓	✓						✓				

V: Vertices (s/o). X-YZ: A hash table $X = x_1$ will lead to a datastructure sorted by XYZ where $X = x_1$ or with a small number of other x_i values. TC: Triple counts, a hash table of #-triples per resource (s/p/o). 2VC: 2 Variable counts, for each 2 variable combination (sp, ps, …) maps to the count of the number of triples for values of that combination. MM : Min/Max Values for each p/block. SF: spo Sorted File with pointers. BC: Compressed bit-cube of |P| concatenated SO matrices. VST: VS* tree, Hash(PO/PS)$\rightarrow$B+ tree$\rightarrow$V (VS*-tree) RI: |p| s-*-o reachability indexes

Table 8 Summary of the survey of the features of existing systems

	Subdivision	Compression	Redundancy
Prevalent approaches	The increasing use of hash-maps rather than B+-trees.	Encoding of literals as integers. Triple store compression using bit-based representations.	Multiple ($\geq$ 3) representations per system.
Gaps	Limited use of complex and hybrid indexes. Limited support for representations for optimal N-way joins.	No compression of secondary indexes.	No support for special filters (e.g., language). No use of bloom-filters for existence checks. Limited support for complex traversals such as k-hops and reachability. No support for two-position (e.g., sp$\rightarrow$o) hash maps.

dimension, both enables and requires a choice of subdivision for the ID→IRI and IRI→ID mappings. Similarly, the compression design choice of using non-sequential integers to encode literals limits the ability of B+-tree and other sorted structures to support sequence-compatible access patterns such as closed ranges making these subdivision choices less attractive.

6 Design space analysis

In this section, we study the data representations within the design space introduced in Sect. 4, which can be employed to satisfy the requirements of each access pattern analyzed in Sect. 3. Each design decision is motivated by a specific use case but comes with inherent trade-offs. Our SCR model (Sect. 4) provides the necessary framework to explicitly analyze the implications of different design decisions. Therefore, we also provide an analysis of the existing design decisions listed in Sect. 5 identifying unexplored design choices.

6.1 Matching access patterns to the design space

In the following, we provide an analysis cross-referencing the access patterns in Table 1 to the SCR dimensions (Fig. 4) and their embodiments in Tables 5, 6, and 7. This links the requirements of a specific access pattern with an appropriate design choice.

Constants (Table 9) Our model shows that, for access patterns matching partially instantiated triples, the close link between the effects of *subdivision*, *redundancy*, and *compression* is particularly evident. As mentioned earlier, the presence of constants in an access pattern increases its selectivity. A high degree of *subdivision* for fully and partially instantiated search patterns is highly beneficial to reduce the search space. Nevertheless, for partially instantiated triple patterns we obtain a benefit only if the *subdivision* is at least *sequence compatible* with the positions of the instantiated variables. Otherwise, when the representation is not sequence compatible, the system has to traverse more subsets with a larger cost in random reads (e.g., if we search for all triples with a specific predicate and we are using a hash table representation with buckets on subjects like in Fig. 3). We conclude that, for partially instantiated triple patterns, high *subdivision* can be highly beneficial if they are *perfectly compatible* with the structure of the query, or strongly limit the query performance if not. For very large range scans (or when no range condition applies), instead, the common approach to store the entire dataset within a specific index can be highly detrimental if the data structure does not have the option to perform an efficient sequential scan (e.g., a linked hash table).

Compression could be beneficial for fully instantiated access if it allows searching efficiently for the compressed values. Yet, employing *compression* is detrimental if several decompression steps are required to check the existence of specific values. On the other hand, for partially instantiated triple patterns, given that we expect a larger intermediate result, *compression* could reduce data transfer times. Moreover, we could implement search through bitwise comparisons [87]. For uninstantiated access patterns, we will instead usually gain a substantial benefit from *compression* due to the reduced data transfer cost.

Finally, on the *redundancy* dimension, in the *fully instantiated* case a data representation designed to answer existence queries (e.g., Bloom filter) is highly beneficial. Instead, for partially instantiated queries we should exploit different tree or hash indexes as we search for bindings to the variable positions given the instantiated positions. Hence, multiple indexes are necessary to cover different combinations of instantiated and uninstantiated positions (e.g., $s, p \mapsto o$ or $s, o \mapsto p$). In both cases, high *redundancy* provides advantages for partially and fully instantiated access patterns. Conversely, *redundancy* is ineffective for uninstantiated queries since an entire scan of the data will be required.

Filter (Table 10) For filters, when the *subdivision* is sequence compatible with the search range, it has a positive effect. When the representation is not sequence compatible, i.e., the values span multiple partitions, the *subdivision* will cause a larger cost. Instead, when the filter condition requires membership to a particularly small set of values and it is possible to *enumerate those values* (e.g., years between 2015 and 2020), we can actually exploit a hash table index translating this access pattern into fully or partially instantiated triple patterns. Still, this approach requires the system to be aware of the domain or to have a precomputed set of values. For the case where *no filter condition* is specified, in the worst case we require a full scan of the data or to execute a partially instantiated BGP (see above). *Compression* is beneficial in terms of data transfer but could be detrimental when attribute values need to be compared in a filter operation. In the *redundancy* dimension, on the other hand, for filters involving a *large range* of values or a range of values that cannot be enumerated, a tree index is a common effective solution, as long as it supports range scans, i.e., the data representation is sequence compatible after paying an initial seek cost, e.g., a B+ tree.

Traversals (Table 11) For traversal access patterns, *1-hop* traversals are a special case of a partially instantiated triple pattern (i.e., SP and PO). Therefore, for 1-hop traversals the advantages of *subdivision* and *redundancy* are the same as those of the corresponding partially instantiated access pattern in Table 9. When the traversal is bound by a specific predicate, *subdivision* across edges of the same predicate is usually highly effective in reducing the search space and achieves particularly good performance when the data representation is seek-compatible to the access pattern retrieving

Table 9 Effect of different storage solutions for each SCR dimension to address access patterns for **Constants** in the query

	Fully instantiated	Partially instantiated	Uninstantiated
Subdivision	↑ smaller search space	↑ smaller search space if compatible with query; ↓ more search steps if incompatible	↓ more search steps
Compression	↓ decompression required to search exact values; ↓ intermediate resultset is small	↓ decompression required to search exact values; ↑ compressing intermediate results	↑ compressing intermediate results
Redundancy	↑ A Bloom filter to check existence	↑ different SPO tree or hash indexes	∼ requires full scan of data

↑ points to an advantage, ↓ points to a drawback, ∼ points to a mostly unaffected performance (also in the following tables

Table 10 Effect of different storage solutions for each SCR dimension to address **Filter** access patterns in the query

	Enumerable range	Large range	No-range
Subdivision	↑ faster search of single values if compatible; ↓ more search steps if incompatible	↑ faster search of single values if compatible; ↓ more search steps if incompatible	↓ more search steps
Compression	↓ decompression to access small set; ↑ compressing intermediate results	↑ faster read; ↑ compressing intermediate results	↑ compressing intermediate results; ↓ decompression required for search
Redundancy	↑ hash-table search	↑ B+Tree search & sorted scan	∼ requires full scan

the next edge in the path. On the other hand, for traversals that are not bound by a specific predicate, *subdivision* across the same subjects can provide some benefits, but *subdivision* across predicates or other kinds of subdivisions usually increase the number of required search steps since they are not seek-compatible. In general, *compressing* intermediate results can be highly effective for large intermediate result sets. However, using a compressed representation could be expensive if the *compression* does not allow (for instance) fast intersections (e.g., intersect objects and subjects lists). Finally, exploiting *redundancy*, when the access pattern starts from a node and traverses a fixed set of *k-hops* with the same property or an unbounded set of hops (**/+ -hops*), specialized indexes can be highly effective [31].

Pivot (Table 12) Access patterns that are joined together on *the same subject*, with the pivot joining either binary or N different triple patterns (forming so-called *star shapes* [77]), are among the most optimized by existing triplestores (see Sect. 5). On the *subdivision* dimension, different schemes to partition over subjects are commonly applied to provide sequence compatibility and are then very effective. In particular, representing attributes as table columns in a relational model (e.g., Fig. 3c) has also proven highly beneficial [17]. Other similar clustered representations could in theory be adapted to answer specific types of pivots joining N different triple patterns. In the general case, WCOJ [38] requires different representations to access triples in sorted order. For pivot patterns on *object to subject*, the trade-offs are similar to a special case of a 2-hop traversal, although in this case, the predicates involved are also distinct and the intermediate variable binding is usually returned. Therefore, one could employ ad hoc indexes for different 2-hop paths, deploy *subdivisions* based on predicates, or *compressions* that enable bit-wise intersections. *Compression* can of course help data transfer, but can help search, e.g., with id intersection across compressed representations. Finally, employing *redundancy*, most indexes provide fast execution of these patterns, beneficial for modern join algorithms, e.g., WCOJ [38].

Return (Table 13) The requirements of access patterns are also dependent on the type of information they extract from the evaluation of the BGP, i.e., the *returned values*. This means that, to return *all the variable bindings*, *Subdivision*, depending how implemented, can either increase the number of steps required to obtain the answers or instead provide a benefit by allowing to read less data, depending on its compatibility to the access pattern. One important advantage is provided by sorted representations when those are compatible with the sorting order imposed by the query (e.g., by an

Table 11 Effect of different storage solutions for each SCR dimension to address **Traversal** access patterns in the query

	1-hop	k-hops over p	*/+-hops over p
Subdivision	↑ property-table or clustered on S or SP, faster search/retrieval	↑ clustered on P or PS, faster search/retrieval; ↓ more search steps if divided over O	↑ clustered on P or PS, faster search/retrieval; ↓ more search steps if cluster is incompatible
Compression	↓ requires decompression; ↑ compressing intermediate results	↓ requires decompression; ↑ compressing intermediate results; ↑ bitwise set intersection	↓ requires decompression; ↑ compressing intermediate results; ↑↑ bitwise set intersection
Redundancy	↑ hash-index on S or SP, faster search/retrieval	↑ execute as set of 1-hops; ↑ specialized index [31]	↑ execute as set of 1-hops; ↑ specialized index [31]

Table 12 Effect of different storage solutions for each SCR dimension to address **Pivot** access patterns in the query

	Two-way same S/O/P	Two-way different S/O/P	N-way pivot
Subdivision	↑ clustered-properties [17] less retrieval steps	↑ restrict search space	↑ clustered-properties [17] less retrieval steps for the same position; ↑ sorted representations provide WCOJ [38] guarantees
Compression	↓ requires decompression; ↑ compressing intermediate results; ↑ bitwise set intersection		
Redundancy	↑ hash-index fast search	↑ Hash-index fast search	↑ B+-tree WCOJ [38] guarantees

ORDER BY clause). On the other hand, when only *distinct* value bindings are needed, a key-value data structure could be exploited to just scan the keys, thus allowing even a scan-compatible access. Note that this may not be possible for arbitrary hash indexes. Also, for group by aggregates, *Subdivision* could be beneficial when all the values over which to compute the distinct operator reside within a single partition. A *subdivision* that is at least sequence compatible with the access via the group-by key would also provide improved performance.

Compression can be advantageous for transmission of large intermediate results when returning all answers, but when returning results to the user, we will need to decompress all the variable bindings, incurring extra work. Note that some compression methods are particularly effective for, or even require, sorted data. Moreover, for returning distinct values, compression of repeated values (e.g., ⟨key;count⟩ pairs) could improve the scan performance and avoid generating duplicates in the first place. When verifying existence, *compression* could be detrimental if decompression is required before checking the predicate. Finally, in case of *aggregate* information, representations in the form of pre-aggregated values provide a large advantage, while *compression* increases the cost, except for counts when the actual values to be counted are not needed.

Redundancy does not introduce any further advantage or disadvantage when all values are required to be returned, unless the values need to be returned in a particular oder. In this case, having a redundant representation compatible with the required order will save an expensive sorting step. Furthermore, when a search pattern verifies only *existence*, e.g., to verify whether node n_1 is reachable from n_0 without actually listing the paths that connect the two, Bloom filters and similar types of set-based indexes (i.e., higher *redundancy*) could be exploited effectively and provide perfect compatibility. Finally, storing aggregate data in distinct redundant data structures is highly beneficial to save computations.

Write (Table 14) Write operations usually incur two steps: (i) determining which tuples to insert or delete and (ii) materializing their insertion or deletion. The first step requires the same form of access pattern as read queries. However, in the second step different data representations need to be updated to have one or more triples added or removed. In particular, with higher *subdivision* usually we only need to update small localized data structures containing the tuple to insert or delete. This can also allow smaller localized locks,

Table 13 Effect of different storage solutions for each SCR dimension to address **Return** access patterns in the query

	All (unsorted)	All (sorted)	Distinct	Existence	Aggregate
Subdivision	↓ more search steps if divided results; ↑ better performance for sorted retrieval		↓ more search steps if divided results; ↑ small search space if compatible.		↑ favorable to group-by operations
Compression	↑ compressing intermediate results; ↓ decompression final results		↑ fast skip over repeated values	↓ decompression required to check existence	↓ decompression required to compute aggregate
Redundancy	~ all results are returned unprocessed	↑ avoid sorting over sorted representation	↑ in a key-value data structure access only keys	↑ Bloom filter search	↑ precomputed values

Table 14 Effect of different storage solutions for each SCR dimension to address **Write** access patterns in the query

	Insert	Delete
Subdivision	↑ localized updates; ↑ smaller locks; ↓ resizing of single subdivision; ↓ re-distribute subdivisions	↑ localized updates; ↑ smaller locks; ↓ re-distribute subdivisions
Compression	↓ requires decompression; ↑ insertion does not require resizing;	↓ requires decompression;
Redundancy	↓ multiple updates are required	

improving concurrency. Yet, when an insertion takes place, we may need to resize the data structure. In some cases, this results in a chain effect for the redistribution of elements among partitions such as in a B+ tree node that reached the maximum capacity, or in a sorted list. In *compressed* representations, both insert and delete may require decompression and re-compression. On the other hand, structures like fixed-size bitmaps can be updated more easily with simple bit flips. Other structures, like Bloom-filters, instead do usually not allow deletes but only insertions. Finally, higher *redundancy* requires each write operation to be mirrored in every redundant copy.

6.2 A compatibility-based analysis

After exploring how the the design space dimensions intersect with the access patterns, we now show how the notion of compatibility and the cost model presented in Sect. 4 can be used to asses the fit between existing solutions (as surveyed in Sect. 5) and these access patterns. Furthermore, our classification enables identifying unexplored design options and characterization of the optimal design choice for an access pattern.

Analysis methodology Starting with a specific cell in one of Tables 9, 10, 11, 12, or 13, one can investigate whether a new data representation could be employed in order to improve specific trade-offs. For example, consider range queries (i.e., a filter access pattern) and the *redundancy* dimension. In this case, we did not identify any existing solution that adopts skip-list implementation (e.g., S3 [91]) or succint indexes (e.g., SuRF [90]) optimized for both key-value search and range queries in order to support partially instantiated queries and ranges. Similarly, few systems explicitly employ stored representations (e.g., increasing *redundancy*) for the special type ranges (Filter $\mathbb{T}$ in Table 1) which could be mapped to an enumerable range of predefined values (e.g., language tags).

A more advanced analysis can be performed for a specific query. Consider as an example the case of returning the count of values over a partially instantiated BGP with fixed predicate and variable subject and object, e.g., `SELECT ?s COUNT(?o) WHERE { ?s ex:friendOf ?o } GROUP BY ?s`. For this case, we did not find systems that have a compressed representation of ⟨key;count⟩ and a seek compatible subdivision over predicates.

Analysis results Table 15 presents a summary of the analysis insights. In particular, we show for read-only access patterns the best choice found across existing implementations as well as design choices that have not been explored despite (theoretically) providing better performance. We also point out those cases where some widespread design choices have important incompatibility with some access patterns, identifying cases where better options could be studied.

Considering the *Subdivision* dimension, many systems implement separate tables for triples sharing a specific predicate. For instance, queries that match triples with the same subject can usually exploit the clustered representation where several properties for the same subject are stored together in a single row [17]. Similarly, BGPs describing triples sharing the same object can exploit an analogous data organization. On the other hand, a constant in s or o can be exploited by a hash index of the form $s \mapsto po$. Yet, no representation exploits subdivision for a pair of constants, e.g., $sp \mapsto o$. For fixed N-way pivot on the same variable, subdivisions like a property-table, i.e., a table where each column is a property for the same subject, can be seen as efficient solutions, but are compatible only with a limited set of access patterns. Conversely, in all their forms, filter operators are, in general, efficiently executed over clustered and sorted representations for values (e.g., B+trees), or other indexes that preserve order (e.g., skip lists). Yet, overall, we see that there is a strong incompatibility between filters on literals and the fact that all representations store literals and special types in mixed subdivisions. Also, we note that subdivision alone is not sufficient because it can only be compatible with few access patterns, so (as also discussed in the paragraphs below) a system will always require redundant representations.

Our survey also reveals the absence of index *compression* solutions (e.g., [89]). For instance, we do not find any application of effective compression solutions for secondary representations, such as counting indexes (e.g., sp→#) and structures like bloom-filters [26] or compact LSM-tries like SuRF [90] that can also support existence checks, which are prevalent in the existing workloads (as discussed in the next section).

For *redundancy*, while 1-hop $s \to o$ queries are the most common and involve a single triple, specialized indexes have been proposed to speed up reachability and multi-hop path expressions (e.g., Sparqling kleene [31]). Apart from Parliament [45], one can rarely find data representations supporting traversal patterns as required by k-hop and *-hop access patterns. Furthermore, solutions tend to rely on standard B+-tree and hash map implementations, avoiding the use of hybrid structures (e.g., hybrid indexes [65]). An exception is the VS*-tree adopted by gStore [88], which, among others, can also return all subjects or objects that match a specific set of predicates and values, thus addressing the needs of a subset of the N-way pivot access pattern where the variable has the same position in all triple patterns. Data representations that can efficiently return pre-computed match counts or other pre-computed aggregate values provide direct benefits (e.g., group-by-query clusters [4]). Nonetheless, redundancy is used to a large extent.

Recent advancement in query optimization have led to the introduction of worst-case optimal join algorithms (WCOJ) [8,38] that make use of highly redundant representations.

Table 15 Compatibility between Access patterns and design choices

Access pattern	Best found	Incompatible	Unexplored
S, P, O 1 Constant	Hash table(+sL, +sK, +sQ), Separate Table(+sL, +sQ)	Incompatible Hash table(-sL,-sK,-sQ)	Surf(+sL, +sK, +sQ), S3(+sL, +sQ)
SP, SO, PO 2 Constants	Hash Table(+sK, +sQ), B+Tree(+sL, +sQ)		SuRF(+sL, +sK, +sQ), $so \mapsto p$, $sp \mapsto o$(+sL, +sK, +sQ), S3(+sL, +sQ)
SPO 3 Constants	Hash Table(+sK, +sQ), B+Tree(+sL, +sQ)		Bloom filter(+sL, +sK)
$<$ & $>$ Closed range	B+Tree(+sL, +sQ)	Mixed Literals/URI(-sL, -sQ)	SuRF(+sL, +sK, +sQ), Skip list(+sL, +sQ)
$<$ | $>$ Open range	B+Tree(+sL, +sQ)	Mixed Literals/URI(-sL, -sQ)	SuRF(+sL, +sK, +sQ), Skip list(+sL, +sQ)
$\mathbb{T}$ Special type range	Partial subdivision of ID ranges(+sL, +sQ)	representations mix these(-sL,-sK,-sQ)	Separate Tables/Files(+sL, +sQ)
$s \rightarrow o$ 1-hop over p	Hash Table(+sK, +sQ), Matrix(+sK, +sQ), Separate Table(+sL, +sQ)		$sp \mapsto o$(+sL, +sK, +sQ)
$s \xrightarrow{k} o$ k-hop $p_1, \ldots, p_k$	Separate p matrix(+sL), Reachability Index(+sL, +sQ), VS*-tree(+sL)		$sp \mapsto o$(+sL, +sK)
$p^{*/+}$?-hop over p	Separate p matrix(+sL), Reachability Index(+sL)		
2-way same position	Single Hash table(+sK, +sQ)	B+Tree(-sK, +sL)	$sp \mapsto o$(+sL, +sK, +sQ)
2-way different position	Pre-joined materialization(+sL, +sQ); Reachability Index(+sL, +sQ)	B+Tree(-sK, -sL, -sQ)	4-position hash-map(+sK, +sL, +sQ)
N-way same position	VS*-tree (+sK); Indexed Property table(+sL,+sQ); Multiple B+Tree(-sK, -sL, -sQ)	Incompatible Hash table(-sL,-sK,-sQ)	Dyadic tree [43] (+sL)
N-way arbitrary position	Multiple B+Tree(-sK, -sL, -sQ)	Incompatible Hash table(-sL,-sK,-sQ)	Dyadic tree [43] (+sL)
All values	Single file(+sQ)		
Sorted values	B+tree(+sL, +sQ); Sorted table(+sL, +sQ)	Hash Table (-sL, -sQ)	
Distinct values	Hash table(+sK, +sL)		Value counts(+sL, +sK, +sQ)
Existence	Hash table(+sK)		Bloom filter (+sK, +sL)
Aggregate	Store counts(+sL, +sK, +sQ), group-by clusters(+sL, +sK, +sQ)		Compressed Value counts(+sL, +sK, +sQ)

In brackets we highlight the compatibilities: selection compatible (+sL), seek compatible (+sK), sequence compatible (+sQ), and the corresponding incompatibilities (-sL, -sK, -sQ)

These algorithms limit query complexity to a factor of the final result size by performing an N-way join with a pivot over the same variable in all positions. Current implementations require sorted access to a single triple pattern for triples matching the BGP sharing the same variable. However, conceptually, a solution providing all mappings of a variable at once could support these algorithms better. The dyadic tree proposed to index *gap boxes* in the geometric resolution approach to multi-way joins [43] provides a possible avenue for providing such support. A *gap box* is a multi-dimensional representation of the uncovered portions of the domains of variables participating in a multi-way join. In the absence of such a representation for RDF data, one solution [38] (extending a Jena TDB system) uses a total of 6 B+-tree indexes to cover all position permutations and thus settles for a non-optimal (non seek-compatible) fully redundant solution. Another recent solution [8] opts for a highly compressed and extremely subdivided wavelet-tree-based structure that is built as a bidirectional *spo* ring which represents the entire graph as a string and indexes all its suffixes. However, the current implementation does not support many other access patterns, most notably, from the *write* dimension.

Thus, in practical applications one is required to consider a full-system design and to analyze the access patterns of an entire workload. We provide an example of such analysis in the following section.

7 Workload case analysis

In this section, we provide an empirical analysis of the access patterns present in different workloads from existing benchmarks and datasets. Moreover, we show how the analysis can be used to guide the choice of which data representations to adopt with respect to a query workload.

7.1 Analyzed workloads

Previous analyses (e.g., [70]) focused on specific features of the language and query complexity (i.e., the topology of the graph patterns). Therefore, such analyses are centered around those characteristics that would impact mainly the query optimizer. Instead, *here we present an analysis of the access patterns specific to the storage layer* (described in Sect. 3) whose performance have direct correlation with the available physical data representations. In this sense, this analysis also provides a complementary view to existing studies.

In particular, we employ 10 workloads: 5 from popular synthetic SPARQL benchmarks and 5 based on real-world query-logs. The synthetic benchmarks are the Lehigh University Benchmark (LUBM) [32], SP2Bench [71], the LDBC Social Network Benchmark (LDBC) [23], the FEASIBLE Benchmark [69], and the Waterloo SPARQL Diversity Test Suite (WatDiv) [3], comprising 14, 17, 19, 50, and 50 queries, respectively.

The real workloads are based on queries against public biological knowledge graphs (BioBench [83]), query logs from the DBpedia endpoint [16], user submitted queries to the public WikiData endpoint [79], the Semantic Web Dog Food query log [53], and a popular benchmark for complex natural question answering over Freebase (Complex) [75]. The workloads contain 22, 46, 3.4 k, 64 k, and 169.7 k queries.

To identify the access patterns, we implemented a static analysis parser for SPARQL, available online.[7] The analyzer extracts the parse tree from each query with the Jena query parser [7] and automatically maps query constructs matching the aforementioned access patterns (see Sect. 3 and Fig. 2). Hence, the input of the parses is a given query workload, and the output is the analysis of the prevalence of each specific access pattern in the given workload. We envision that the provided tool can be used by practitioners to evaluate their own query logs for the prevalence of specific access patterns. These patterns can then be compared with the support provided by their systems of choice. Thus, for example, if a workload is characterized by an abundance of k-hops, it would be prudent to evaluate the support by the current system compared to an alternative system that provides a suitable representation, such as the VS*-tree provided by gStore [88] or the reachability index in Sparqling kleene [31].

7.2 Limitations

In general, the existence of an access pattern in a query does not mean that a specific system must utilize it to answer the query. For example, *k-hop* patterns such as {s1 p1/p2 ?o1} can be answered by converting the BGP into the $O \equiv S$ pivot pattern {s1 p1 ?v1. ?v1 p2 ?o}. Another example is the sorted value return pattern, which can be used by different algorithms to answer other query patterns as well, e.g., by WCOJ [38] to perform the efficient intersection in an N-way pivot. Yet, in our analysis, we counted only queries explicitly requiring sorted output in the form of an `ORDER BY` clause. Also, we excluded from this analysis those `ORDER BY` clauses that followed an aggregation resulting from a `GROUP BY` clause, since the sorted output could not be directly derived from a sorted retrieval operation.

7.3 Results and discussion

The aggregated results over all benchmarks are presented in Table 16. Constants are mainly used in predicate (P) and subject-predicate (SP) combinations. The abundance of P-constant patterns justifies the prevalent use of *subdivision* by predicates in the systems reviewed, which allows rapid reduction of the search space. SP combinations are better served by point-lookup mechanisms such as hash maps, yet this alone does not justify the *redundancy* of having indexes for all permutations of SPO since, for instance, patterns selecting objects (O) are quite rare.

The large number of binary same-position (e.g., $S{\equiv}S$) pivots requires systems to supplement the *subdivision* of data representations by predicate with other redundant data representations supporting access patterns compatible with selection by subject/object. Within the binary different-position pivot (BiD in Table 16), the $O{\equiv}S$ pivot dominates since very few queries involve the pivot in the predicate positions. The N-way pivots are highly prevalent in all benchmarks. Perhaps surprisingly, arbitrary position variable centric pivots (NwA in Table 16), are almost as prevalent as star patterns. The recent emergence of WCOJ algorithms [38]

[7] https://github.com/tomersagi/rdf_access_patterns.

Table 16 Prevalence of access patterns in popular RDF benchmarks

		Constants							Filter			Traversal						Return					Pivot			
Set	#	S	P	O	SP	PO	SO	SPO	[ ]	[ )	Sp*	S·O	O·S	SkO	OkS	SP*O	OP*S	All	DV	ST	EX	Σ	BiS	BiD	Star	NwA
BioBench	22	0.00	0.77	0.00	0.14	0.86	0.00	0.00	0.00	0.00	0.14	0.77	0.86	0.64	0.36	0.00	0.00	0.86	0.14	0.14	0.86	0.00	0.77	0.68	0.45	0.41
WikiData	46	0.00	0.93	0.02	0.00	0.83	0.00	0.00	0.11	0.15	0.11	0.89	0.80	0.22	0.26	0.04	0.50	0.43	0.41	0.30	0.85	0.22	0.93	0.43	0.52	0.20
ComplexQ	3443	0.00	0.88	0.00	0.34	0.96	0.00	0.00	0.00	0.02	0.98	1.00	0.96	0.38	0.40	0.00	0.00	0.0*	1.00	0.05	1.00	0.0*	0.73	0.74	0.0*	0.19
SWDF	64030	0.24	0.23	0.01	0.31	0.03	0.0*	0.10	0.00	0.00	0.0*	0.56	0.05	0.21	0.0*	0.00	0.00	0.23	0.34	0.11	0.33	0.01	0.32	0.21	0.08	0.20
DBpedia	169721	0.05	0.69	0.03	0.19	0.71	0.0*	0.0*	0.0*	0.01	0.64	0.87	0.74	0.56	0.07	0.0*	0.0*	0.78	0.20	0.03	0.67	0.00	0.66	0.60	0.54	0.57
LUBM	14	0.00	0.43	0.00	0.14	1.00	0.00	0.00	0.00	0.00	0.00	0.50	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.86	0.00	0.79	0.43	0.29	0.36
SP2Bench	17	0.00	0.65	0.06	0.00	0.82	0.00	0.06	0.00	0.06	0.00	0.82	0.88	0.41	0.29	0.00	0.00	0.47	0.35	0.12	0.88	0.00	0.82	0.53	0.24	0.29
LDBC	19	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.11	0.32	0.00	1.00	1.00	0.95	0.79	0.16	0.16	0.74	0.42	0.53	1.00	0.37	1.00	0.95	0.89	0.79
Feasible	50	0.14	0.72	0.10	0.12	0.58	0.00	0.00	0.02	0.04	0.42	0.86	0.72	0.08	0.08	0.00	0.00	0.44	0.56	0.32	0.58	0.00	0.70	0.30	0.42	0.18
WATDIV	50	0.00	1.00	0.00	0.58	0.28	0.00	0.00	0.00	0.00	0.00	1.00	0.40	0.02	0.02	0.00	0.00	1.00	0.00	0.00	0.98	0.00	0.40	0.82	0.18	0.20

Color marks prevalence: Yellow/Medium **0.33−0.66**, High/Red *>0.66*; 0.0∗ marks values in 0.0 − 0.01. [ ] represents a closed range, [ ) an open range, and Sp* a special range. SkO and OkS represent K-step S→O and O→S traversals, respectively. All: all values, DV: distinct values, ST: sorted, EX: existence check, Σ aggregate. Pivot acronyms: BiS: Binary same position, BiD: Binary different position, Star: N-way same position is also known as a star pattern, NwA: N-way arbitrary position

optimizing this access pattern has exposed the limitations of existing data representations in supporting this pattern and required the introduction of novel representations such as the recently proposed ring [8], that allows all-position pivots on a selected variable in a single representation.

Notably, a large number of benchmark queries employ *special range* filters, e.g., filters for specific languages (e.g., `@en`), or specific types of literals, i.e., numeric values vs. strings vs. dates. This is especially prevalent when querying multi-language knowledge graphs and when filtering values only for literals of a specific data type. This calls for indexes or *subdivisions* of triple objects by languages and data types. *Yet, none of the data representations in the reviewed storage systems explicitly support such access patterns although the benchmark queries suggest they could be especially useful.* Conversely, the relative absence of *closed/open range* filters from most benchmarks corresponds well with the decreased reliance of systems on B+-trees observed in Section 5.3 as the relative advantage of B+trees over hash-based lookup mechanisms is greatly diminished in the absence of closed/open range queries or other access patterns that require sorted access.

When examining the traversal patterns, it is evident that 1-hop traversals dominate although k-hop traversals have a substantial presence as well. It is interesting to note how LDBC queries represent an outlier since almost all queries employ k-hop traversals. For those special workloads, a system supporting efficient k-hops (e.g., gStore [88]) could outperform systems that are forced to break these k-hops up into multiple 1-hop patterns. Moreover, traversals with unbounded path queries (marked as $P*$) are quite rare. This rarity renders reachability indexes not so useful in practice when compared to the extra space and update cost they require. It is an open question whether the currently limited presence of these queries is due to the complexity of the query language (i.e., users are not familiar in expressing such queries) or to most systems not being optimized for these access patterns and hence known to result in slow queries that are avoided by users.

Another important finding is that over 80% of all the queries can make use of *existence* access patterns. Recall that in these patterns, no triple values are required, but only a simple test of existence. This prevalence is especially striking when considering that almost no existing data representations specifically target existence queries. In many ways, the success of bit representations such as TripleBit [87] and BitMat [11] can be attributed to this prevalence. Therefore, data representations that are compatible with existence access patterns, such as Bloom filters, are an unexplored but highly promising addition to existing systems and could very likely replace less-frequently used indexes (e.g., B+-trees on OSP permutations). There is a relatively low number of queries requiring an explicit sorted result. This allows for the use of unsorted representations and algorithms for much of the retrieval, leaving the sorting for post-processing. Finally, we see that distinct values are often required while aggregation is present in only a few queries. However, once more, since aggregation queries are currently computationally expensive, we could not exclude that better performance for such queries would result in more frequent use.

One could also use Table 16 to perform retrospective analysis of previously published benchmarks. For example, upon its introduction, *DB2 Graph* [17] was compared with RDF-3X [55] over both LUBM and DBpedia. Two major differences between these two benchmarks are the prevalence of P-position constant queries in DBpedia as well as more queries returning distinct values and existence checks rather than full results. Thus, one would expect RDF-3X with its P-centric data representations (which DB2 Graph lacks) and usage of IRI replacement to perform comparatively better on DBpedia than on LUBM. Indeed, the more modern DB2 Graph was not able to outperform RDF-3X on DBpedia although doing twice as well on LUBM.

8 Related surveys

Sakr and Al-Naymat [68] defined a taxonomy of data representations for relational-based RDF triplestores. In this work, we review all centralized RDF triplestores.

Modoni et al. [52] review triplestores in the context of their usefulness as meta data management systems used by other software in an enterprise scenario. Thus, they focus on technical and non-functional features such as user management, security, and programming language support, unlike our focus on data representation.

Ma et. al. [50] review RDF triplestores by their logical data representation, dividing the landscape into relational (traditional) and non-relational (NoSQL). They further divide relational representations into *vertical* (single triple table with 3 columns s p o), *horizontal* (s as row p as column o as value / s as row, p as table and o as value) and *type* (multiple standard horizontal tables partitioned by the s type). In this work, we abstract beyond relational/non-relational, looking at three orthogonal design dimensions.

Özsu [59] focuses on several representative solutions contrasting the centralized versus the distributed approach, and within the centralized approach lists a few architectural choices such as whether to rely on a relational mapping from RDF to an RDBMS. Pan et al. [60] present a similar taxonomy, complementing it with a review of the prominent benchmarks used in the field and a short comparative empirical evaluation. In this work, we instead abstract across different architectural choices to identify the underlying design space.

Abdelaziz et al. [2] focus in their review on distributed triplestores and on large-scale benchmarks. Kaoudi and Manolescu [42] similarly limit their analysis to cloud-native solutions. In this work, we focus on centralized stores since the distribution of a data structure can be handled as an orthogonal aspect.

Wylot et al. [85] provide a taxonomy whose two main branches are centralized and distributed, albeit with a more detailed categorization of the centralized branch into six architectural approaches. The six types offered are: triple tables, property tables, index permutations, vertical partitioning, graph-based, and binary storage. While these correspond to some of the options in our subdivision and redundancy dimensions, the authors do not discuss additional solutions we identify, e.g., specific indexes for specific access patterns, such as hash indexes and reachability indexes. They do not explicitly address compression options (as we do in Table 6) and they do not discuss the compatibility between access patterns and design choices. In this paper, we identify the fine-grained design space options available to system designers and align them across three orthogonal dimensions to allow identifying new, previously unexplored, combinations. Therefore, our analysis is significantly more generalizable, as demonstrated by our ability to analyze 22 systems over the proposed SCR space, overcoming the limitations of Wylot et al. [85].

Pérez et al. [62] performed worst-case complexity analysis of SPARQL operators regardless of any indexes or physical data representations by counting the number of edges traversed in a conceptual graph representation for each pattern. In our work, we align between design choices for data representation and their effect on the cost of different access patterns.

The *Data Calculator* is based on the analysis provided by the RUM Conjecture [10], which explores the trade-offs between read times (R), update cost (U), and memory (or storage) overhead (M). The intuition behind the RUM conjectures is also shared by our SCR space, although we perform a complementary analysis.

Finally, in the design of Peloton [61], Pavlo et al. present an overview of *self-driving actions*, i.e., the types of actions that a *self-tuning* (also called self-driving) relational system must support. These actions are divided in three classes: runtime actions, physical actions, and data actions. Moreover, these actions are limited to the relational data model. In practice, for the storage level, they only allow either to move from columnar layout to row layout, and vice versa, or to add and drop indices. Nonetheless, we envision the possibility to expand the Peloton framework to the case of triplestores and believe that the analysis provide here is a fundamental contribution in this direction.

9 Conclusions and future work

In this work, we introduce the new Subdivision-Compression-Redundancy (SCR) design space of data representations for RDF databases. We also introduce a new feature space for analyzing query workload access patterns. Together, they allow the analysis of RDF store design decisions, specifically, which data representations can effectively support a given workload. We performed an analysis of popular RDF store benchmarks under these assumptions and showed that multi-hop traversals and existence checks are broadly underserved by existing RDF store designs, which are oriented toward providing support for 1-hop traversals, pivots, and access patterns featuring a constant predicate value. Many of these design choices can be made at run-time (e.g., by using an automated tuning mechanism) following an inspection of a query load or in advance for a planned workload. To map between query loads and design choices, we offer a simple cost model, a feature space over which query loads can be evaluated, and an analysis of the impact of these choices. Thus, we lay the ground for future RDF stores to design novel solutions over this space in an informed manner. Such designs can be achieved either manually or even semiautomatically as proposed by the recent effort in self-designing data structures [40] and self-organizing relational database systems [61]. By cross-referencing the access patterns in a query workload with SCR design options, future RDF stores will be able to add components of the system to match expected workload. In future work, we intend to explore the architecture and design principles of such systems.

Acknowledgements We wish to thank the anonymous reviewers for the detailed and insightful comments that allowed us to greatly improve the final version of this manuscript.

Funding This work is supported by the Independent Research Fund Denmark under Grant No. DFF-8048-00051B (RelWeb) and the SEMIOTIC project, Aalborg University's Talent Programme, the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant agreement No. 838216, and the Poul Due Jensen Foundation.

Data availability The data in this work were collected from public datasets (see Sect. 7 for references and a link to our code).

Declarations

Conflict of interests The authors declare that they have no conflict of interest.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for semantic web data management. VLDB J. **18**(2), 385–406 (2009)
2. Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proc. VLDB Endow. **10**(13), 2049–2060 (2017)
3. Aluç, G., Hartig, O., Tamer Özsu, M., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC. pp. 197–212 (2014)
4. Aluç, G., Tamer Özsu, M., Daudjee, K., Hartig, O.: Executing queries over schemaless RDF databases. In: ICDE. 807–818 (2015)
5. Aluç, G., Tamer Özsu, M., Daudjee, K.: Building self-clustering RDF databases using tunable-LSH. VLDB J. **28**(2), 173–195 (2019)
6. Andrzejewski, W., Wrembel, R.: GPU-WAH: applying GPUs to compressing bitmap indexes with word aligned hybrid. In: Database and Expert Systems Applications, pp. 315–329. Springer, Berlin (2010)
7. Apache: Apache Jena. Accessed jan. 4, 2020. http://jena.apache.org (2020)
8. Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J.L., Rojas-Ledesma, J., Soto, A.: Worst-case optimal graph joins in almost no space. In: Proceedings of the 2021 International Conference on Management of Data. pp. 102–114 (2021)
9. Athanassoulis, M., Idreos, S.: Design tradeoffs of data access methods. In: SIGMOD. pp. 2195–2200 (2016)
10. Athanassoulis, M., Kester, M.S., Maas, L.M., Stoica, R., Idreos, S., Ailamaki, A., Callaghan, M.: Designing access methods: the RUM conjecture. In: EDBT. pp. 461–466 (2016)
11. Atre, M., Srinivasan, J., Hendler, J.A.: BitMat: a main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In: ISWC (Posters & Demonstrations). pp. 1–2 (2008)
12. Bausch, D., Petrov, I., Buchmann, A.: Making cost-based query optimization asymmetry-aware. In: Proceedings of the Workshop on Data Management on New Hardware. pp. 24–32 (2012)
13. Bebee, B.R., Choi, D., Gupta, A., Gutmans, A., Khandelwal, A., Kiran, Y., Mallidi, S., McGaughy, B., Personick, M., Rajan, K., Rondelli, S., Ryazanov, A., Schmidt, M., Sengupta, K., Thompson, B.B., Vaidya, D., Wang, S.: Amazon neptune: graph data management in the cloud. In: ISWC (Posters & Demonstrations). (2018)
14. Besta, M., Peter, E., Gerstenberger, R., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., Hoefler, T.: Demystifying graph databases: analysis and taxonomy of data organization, system designs, and graph queries. Technical Report. (2019) arXiv:1910.09017
15. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: a family of scalable semantic repositories. Semantic Web **2**(1), 33–42 (2011)
16. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. VLDB J. **29**(2), 655–679 (2020)
17. Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient RDF store over a relational database. In: SIGMOD. pp. 121–132 (2013)
18. Brisaboa, N.R., Cerdeira-Pena, A., Fariña, A., Navarro, G.: A compact RDF store using suffix arrays. In: Costas, I., Simon, P., Emine, Y. (eds.) String Processing and Information Retrieval, pp. 103–115. Springer, Cham (2015)
19. Chazelle, B., Kilian, J., Rubinfeld, R., Tal, A.: The bloomier filter: an efficient data structure for static support lookup tables. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 30–39 (2004)
20. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: Integrating RDF data into a relational database system. US Patent US8719250B2 (2014)
21. Cyganiak, R.: A relational algebra for SPARQL query developers with a powerful tool to extract information from large A relational algebra for SPARQL. Technical Report, HP Laboratories Bristol, Bristol, UK (2005)
22. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In: SIGMOD. 145–156 (2011)

23. Erling, O., Averbuch, A., Larriba-Pey, J.-L., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC Social Network Benchmark: Interactive Workload. In: SIGMOD. pp. 619–630 (2015)
24. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: Semantic Web Information Management - A Model-Based Perspective. Springer, pp. 501–519 (2009)
25. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semant. Sci. Serv. Agents World Wide Web **19**, 22–41 (2013)
26. Ficara, D., Giordano, S., Procissi, G., Vitucci, F.: Multilayer compressed counting bloom filters. In: Proceedings of the 27th Conference on Computer Communications. IEEE, pp. 311–315 (2008)
27. Florescu, D., Levy, A., Manolescu, I., Suciu, D.: Query optimization in the presence of limited access patterns. In: SIGMOD. pp. 311–322 (1999)
28. Franz Inc. 2020. AllegroGraph. Accessed jan. 14, 2020. https://franz.com/agraph/allegrograph
29. Frasincar, F., Houben, G.-J., Vdovjak, R., Barna, P.: RAL: an algebra for querying RDF. WWW **7**(1), 83–109 (2004)
30. Galárraga, L., Hose, K., Schenkel, R.: Partout: a distributed engine for efficient RDF processing. WWW **2014**, 267–268 (2014)
31. Gubichev, A., Bedathur, S.J., Seufert, S.: Sparqling kleene: fast property paths in RDF-3X. In Workshop on Graph Data Management Experiences and Systems, GRADES (2013)
32. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. **3**(2–3), 158–182 (2005)
33. Harris, S., Gibbins, N.: 3store: efficient bulk RDF storage. In: Proceedings of the International Workshop on Practical and Scalable Semantic Systems (PSSS). 1 (2003)
34. Harris, S., Lamb, N., Shadbolt, N.: 4store : The design and implementation of a clustered RDF store. In: Scalable Semantic Web Knowledge Base Systems (SSWS). pp. 81–96 (2009)
35. Harris, S.: Andy. Seaborne. 2012. SPARQL 1.1 Query Language. W3C Recommendation 21 March (2013)
36. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In: ISWC. pp. 211–224 (2007)
37. Hellerstein, J.M., Stonebraker, M., Hamilton, J.: Architecture of a database system. Found. Trends Databases **2007**, 141–259 (2007)
38. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: ISWC. Springer, pp. 258–275 (2019)
39. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
40. Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., Zhu, Z.: Design continuums and the path toward self-designing key-value stores that know and learn. In: CIDR (2019)
41. Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M.S., Guo, D.: The data calculator: data structure design and cost synthesis from first principles and learned cost models. In: SIGMOD. pp. 535–550 (2018)
42. Kaoudi, Z., Manolescu, I.: RDF in the clouds: a survey. VLDB J. **24**(1), 67–91 (2015)
43. Khamis, M.A., Ngo, H.Q., Ré, C., Rudra, A.: Joins via geometric resolutions: worst case and beyond. ACM Trans. Database Syst. (TODS) **41**(4), 1–45 (2016)
44. Klyne, G., Carrol, J.J., McBride, B.: RDF 1.1 Concepts and Abstract Syntax. World-Wide Web Consortium (2014)
45. Kolas, D., Emmons, I., Dean, M.: Efficient linked-list RDF indexing in Parliament. In: Proceedings of the Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS). Washington DC, USA, pp. 17–32 (2009)
46. Krech, D.: RDFlib: A Python Library for Working with RDF. Accessed jan. 14, (2020). https://rdflib.readthedocs.io
47. Lissandrini, M., Brugnara, M., Velegrakis, Y.: Beyond macrobenchmarks: microbenchmark-based graph database evaluation. Proc. VLDB Endow. **12**(4), 390–403 (2018)
48. Liu, X., Thomsen, C., Pedersen, T.B.: 3XL: supporting efficient operations on very large OWL Lite triple-stores. Inform. Syst. **36**(4), 765–781 (2011)
49. SYSTAP LLC. 2013. *The bigdata RDF Database*. Technical Report. SYSTAP LLC. https://blazegraph.com/docs/bigdata_architecture_whitepaper.pdf
50. Ma, Z., Capretz, M.A.M., Yan, L.: Storing massive resource description framework (RDF) data: a survey. Knowl. Eng. Rev. **31**(4), 391–413 (2016)
51. Menon, P., Mowry, T.C., Pavlo, A.: Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. Proc. VLDB Endow. **11**(1), 1–13 (2017)
52. Modoni, G.E., Sacco, M., Terkaj, W.: A survey of RDF store solutions. In: Proceedings of the Conference on Engineering, Technology and Innovation (ICE). pp. 1–7 (2014)
53. Möller, K., Heath, T., Handschuh, S., Domingue, J.: Recipes for semantic web dog food — the ESWC and ISWC metadata projects. In: ISWC. pp. 802–815 (2007)
54. Muys, A.: Building an enterprise-scale database for RDF data. Technical Report. The Mulgara Project. (2006) https://code.mulgara.org/projects/mulgara/wiki/ImperfectIndexes
55. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
56. Nitta, K., Savnik, I.: Survey of RDF storage managers. In: Proceedings of the International Conference on Advances in Databases, Knowledge, and Data Applications. pp. 148–153 (2014)
57. OntoText: GraphDB, The Best RDF Database for Knowledge Graphs. Accessed jan. 14, (2020). https://www.ontotext.com/products/graphdb/
58. Owens, Alisdair, Seaborne, Andy, Gibbins, Nick: Clustered TDB': A Clustered Triple Store for Jena. Univ. of Southampton, Technical Report (2009)
59. Özsu, M.T.: A survey of RDF data management systems. Front. Comput. Sci. **10**(3), 418–432 (2016)
60. Pan, Z., Zhu, T., Liu, H., Ning, H.: A survey of RDF management technologies and benchmark datasets. J. Ambient Intell. and Humanized Comput. **9**(5), 1693–1704 (2018)
61. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T.C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Van Aken, D., Wang, Z., Wu, Y., Xian, R., Zhang, T.: Self-driving database management systems. In: CIDR (2017)
62. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 1–45 (2009)
63. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation 15 January 2008 (2008)
64. Punnoose, R., Crainiceanu, A., Rapp, D.: Rya: A scalable RDF triple store for the clouds. In: Proceedings of the Workshop on Cloud Intelligence (Cloud-I). Article 4 (2012)
65. Qu, W., Wang, X., Li, J., Li, X.: Hybrid indexes by exploring traditional B-tree and linear regression. In: International Conference on Web Information Systems and Applications. Springer, pp. 601–613 (2019)
66. Ravindra, P., Kim, H., Anyanwu, K.: An intermediate algebra for optimizing RDF graph pattern matching on MapReduce. In: ESWC. pp. 46–61 (2011)
67. Sahoo, S.S., Halb, W., Hellmann, K., Idehen, S., Jr Thibodeau, T., Auer, S., Sequeda, J., Ezzat A.: A survey of current approaches for mapping of relational databases to RDF. Technical Report. W3C RDB2RDF Incubator Group (2009)

68. Sakr, S., Al-Naymat, G.: Relational processing of RDF queries: a survey. SIGMOD Record **38**(4), 23–28 (2009)
69. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.-C.: FEASIBLE: a feature-based SPARQL benchmark generation framework. In: ISWC. pp. 52–69 (2015)
70. Saleem, M., Szárnyas, G., Conrads, F., Ahmad Chan Bukhari, S., Mehmood, Q., Ngomo, A.-C.N.: How representative Is a SPARQL benchmark? An analysis of RDF Triplestore benchmarks. In: WWW. pp. 1623–1633 (2019)
71. Schmidt, M., Hornung, T., Meier, M., Pinkel, C., Lausen, G.: SP^{2}Bench: a SPARQL performance benchmark. In: Semantic Web Information Management - A Model-Based Perspective. Springer, pp. 371–393 (2009)
72. Griffiths Selinger, P., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD, pp. 23–34. Association for Computing Machinery, New York, NY, USA (1979)
73. Sintek, M., Kiesel, M.: RDFBroker: a signature-based high-performance RDF store. In: ESWC. pp. 363–377 (2006)
74. Stardog Union. (2020). Stardog. Accessed jan. 14, 2020. https://www.stardog.com/
75. Talmor, A., Berant, J.: The web as a knowledge-base for answering complex questions. In: NAACL-HLT. pp. 641–651 (2018)
76. Urbani, J., Jacobs, C.: Adaptive low-level storage of very large knowledge graphs. In: Proceedings of The Web Conference 2020 (Taipei, Taiwan) (WWW '20). Association for Computing Machinery, New York, NY, USA, pp. 1761–1772. (2020) https://doi.org/10.1145/3366423.3380246
77. Vidal, M.-E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently joining group patterns in SPARQL queries. In: Extended Semantic Web Conference. Springer, pp. 228–242 (2010)
78. Volz, R., Oberle, D., Staab, S., Motik, B.: KAON SERVER - a semantic web management system. In: WWW. online (2003)
79. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014)
80. Wang, Y., Xiaoyong, D., Jiaheng, L., Wang, X.: FlexTable: using a dynamic relation model to store RDF data. In: Database Systems for Advanced Applications (DASFAA), pp. 580–594. Tsukuba, Japan (2010)
81. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1), 1008–1019 (2008)
82. Wolff, B.G.J., Fletcher, G.H.L., Lu, J.J.: An extensible framework for query optimization on TripleT-based RDF stores. In: Workshops of EDBT/ICDT. pp. 190–196 (2015)
83. Wu, H., Fujiwara, T., Yamamoto, Y., Bolleman, J., Yamaguchi, A.: BioBenchmark Toyama 2012: an evaluation of the performance of triple stores on biological data. J. Biomed. Semant. **5**(1), 32 (2014)
84. Zhe W., Moreno, G.M., Banerjee, J.: Storing and querying graph data in a key-value store. US Patent US20140310302A1 (2014)
85. Wylot, M., Hauswirth, M., Cudré-Mauroux, P., Sakr, S.: RDF data storage and query processing schemes: a survey. ACM Comput. Surv. **51**(4), 36 (2018)
86. Yakovets, N., Godfrey, P., Gryz, J.: Evaluation of SPARQL property paths via recursive SQL. AMW 1087 (2013)
87. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: TripleBit: a fast and compact system for large scale RDF data. Proc. VLDB Endow. **6**(7), 517–528 (2013)
88. Zeng, L., Zou, L.: Redesign of the gStore system. Front. Comput. Sci. **12**(4), 623–641 (2018)
89. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In: SIGMOD. pp. 1567–1581 (2016)
90. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: Surf: Practical range query filtering with fast succinct tries. In: SIGMOD. pp. 323–336 (2018)
91. Zhang, J., Wu, S., Tan, Z., Chen, G., Cheng, Z., Cao, W., Gao, Y., Feng, X.: S3: a scalable in-memory skip-list index for key-value store. Proc. VLDB Endow. **12**(12), 2183–2194 (2019)