Resource centered RDF data management

Ralf Heese[1] and Martin Znamirowski[2]

[1] Freie Univeristät Berlin, Corporate Semantic Web
[2] Humboldt-Univeristät zu Berlin, Databases and Information Systems

Abstract. Since the first publication of RDF researchers developed systems to store and to query RDF data efficiently on secondary storage. First, they mapped the RDF data model to the relational one. As relational databases performed poorly for generic RDF graphs researches focused on native repositories using B-trees to index triples in most cases. However, native repositories still calculate the result of a query by joining single triples.
Because queries can be decomposed into star-like graph patterns we argue that managing RDF data as a set of subgraphs has advantages over existing approaches. Thus, we propose a storage model that manages an RDF graph as a collection of subgraphs and evaluates queries by joining the results of star-like subqueries. In cause of the results of some preliminary tests we are confident that a system based on our approach can efficiently process queries.

1 Introduction

The Resource Description Framework (RDF) [15] is a recommendation of the W3C that basically defines a data model for describing information about entities (resources) across system boundaries. A piece of information is represented as a triple consisting of the considered resource, a property resource, and the value for that property (a resource or a literal). Every resource is uniquely identified by a URI. The components of the triple are often referred to as subject, predicate, and object, respectively. Having unique identifiers for resources a set of triples forms a graph considering subjects and objects as nodes and predicates as edges.

Since the first publication of the RDF specification researchers developed various systems to store and to query RDF data efficiently. At the beginning they focused on mapping the RDF data model to relational one because relational database management systems (RDBMS) have been researched for many many years and perform very well in almost all application scenarios. However, researchers recognized soon that storing RDF data into a relational database raises performance issues when querying the data (e.g., results in many (self) joins). Due to the openness of the RDF data model it is also difficult to define a fixed relational schema (e.g., many multivalued and optional properties).

As a consequence, researches developed hybrid and native RDF repositories. Hybrid RDF repositories use object-oriented features of object-relational

databases to transfer parts of the RDF schema into the relational schema. In contrast, native repositories do not rely on RDBMS at all but store the RDF triples in a set of indexes, e.g., (specialized) B-trees. Native stores still calculate the result of a query by iterating over the variable bindings of its triple patterns. Thus, the processing is similar to the one of joins in RDBMS.

In this paper we present a native RDF repository that manages an RDF graph as a collection of subgraphs and evaluates queries by joining the results of star-like subqueries. We consider a subquery as star-like if it contains a basic graph pattern that triple patterns have all the same subject (either resource URI or variable). In contrast to existing approaches all triples matching a star-like subqueries can be accessed by loading a single database page.

In Section 2 we give an overview of existing approaches to store RDF graphs. Afterwards, we describe our approach: We outline first the design goals of our storage model in Section 3. We describe then the storage model in detail (Section 4) and explain basic operations on RDF graphs (Section 5). In Section 6 we present an estimation of the required disc space for storing RDF data and analyze the complexity of the basic operations. In Section 7 we present and discuss preliminary results of evaluating our approach. Finally, we conclude and give an outlook to future work in the last section.

2 Related work

In this section we give a short overview of existing approaches to manage RDF data. Hereby, we concentrate on repositories that store the data on secondary storage and are designed to run on single machines (cf. Table 1). We exclude explicitely clustered approaches from our considerations because they are designed for a different use case than our storage model and, thus, other challenges are in the focus of research.

We can distinguish two main categories of RDF storage models: Relational based and native approaches (cf. Figure 1). Regarding relational storage models we can distinguish between schema oblivious, schema aware, and hybrid. *Schema oblivious* approaches rely on a single database relation storing all triples. The key advantage of this model is that any RDF model can easily be imported without considering its schema. However, the query engine has to consider the complete database for evaluating a query and large queries require expensive self-join operations. *Schema aware* approaches convert the schema of an RDF graph to a relational schema, e.g., the database contains a relation for each property of the graph. Since the data is distributed over several relations the query engine can selectively access the underlying RDF data for processing of a query; although the resulting SQL queries get more complex. Thus, it can answer queries more efficiently than in the schema oblivious case. In return for efficient query processing changes of the RDF data require sophisticated update operations because they may also affect the database schema (e.g., adding or deleting relations). *Hybrid* approaches try to combine the advantages of the two other models by using dedicated relations for certain (frequent) properties and

System	Reference	Storage model	Status
3store	[11]	schema oblivious	open source, inactive
AllegroGraph	[10]	native	commercial
Hexa-Store	[24]	native	open source
HPRD	[17]	native	scientific
Jena2 SDB	[26]	hybrid	open source
Jena2 TDB	[1]	native	open source
Kowari	[27]	native	open source, inactive
Oracle	[8]	schema oblivious	commercial
RDF-3X	[20]	triple index	scientific
RDFSuite	[2]	hybrid	inactive
RStar	[18]	hybrid	scientific
Sesame	[7]	schema aware, native	open source
System Π	[28]	native	scientific
Virtuoso	[21]	native	commercial
YARS2	[13]	native	scientific
N.N.	[19]	schema aware	scientific

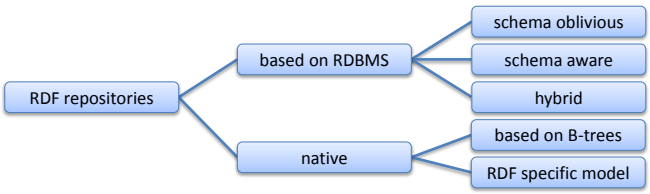

Fig. 1. Categories of RDF storage models

storing the remaining triples in a single relation. The characteristics of query processing are similar to the one in schema aware approaches. In each category there exist several variations and optimizations which we do not discuss here due to limited space.

Most *native* storage models rely on indexing triples (quadruples) in several B-trees. In [12] the author came to the result that eight indexes are sufficient to cover all possible access patterns to the data of named graphs. Only three are needed if named graphs are not considered. Besides B-trees some other systems such as RDF-3X use also other types of indexes (e.g., bitset indexes) to improve query performance and to reduce the amount of disc space occupied by indexes. So far only a few approaches considered indexing property paths separately [17] but this may change with the next version of SPARQL. Besides storage models based on B-trees there exist also approaches that propose different DBMS (e.g., deductive DBMS [25]).

In contrast to all mentioned approaches we propose a native RDF storage model that groups related triples on a database page. In addition, all triples matching a star-like subqueries can be accessed by loading a single database

page. Another difference is that our indexes point to database pages containing the triples rather than storing them.

3 Design goals

Although the storage models presented in the previous section are based on very different access structures, we derived the following fundamental ideas and best practices for realizing an RDF repository.

Reduce join operations. As a consequence of the triple based data model joining triples is an essential operation. Causing significant costs during query processing it is important minimizing the number of join operations.

Normalize URIs. Managing strings unmodified, e.g., URIs and literals, causes performance problems because they are often long. On the one hand comparisons are expensive and on the other one they consume more disc space if they occur often. Thus, strings are normalized in all performant repositories.

Adequate disc consumption. Although disc space is inexpensive nowadays the system should deal with the disc space sparingly. Although replicating data for indexing purposes improves query performance the time for importing and updating data may increase.

Equal weighting of queries. A variable can occur at all positions in a triple pattern. Although some positions are more common an RDF repository should support all queries with similar performance.

Multivalued, optional properties. The semantics of RDF allow multiple values for a property of a resource [14]. In addition, it does not guarantees the presence of a property. As a result RDF graphs have often an unpredictable schema which the repository has nevertheless to handle efficiently.

Although (expensive) solid state discs allow random access to data nowadays we also aim at reducing the number of IO operations. This goal is not induced by the RDF data model but can be considered as a generally accepted rule for improving query performance in database context.

4 Storage concept

To achieve the above design goals we exploit characteristics of RDF data and its usage: (i) resources have unique identifiers, (ii) many properties are assigned to a single resource, (iii) resources of the same `rdf:type` have similar properties, and (iv) locality of queries.

The first characteristic is directly defined in the RDF specification and helps us to locate a resource easily in our repository. The second one we derived from the fact that RDF was designed to describe resources by defining their properties. Taking the well-known DBpedia dataset as an example, we derived from Table 2 in [4] that the ration between the number of resources and the number of triples is about $2.8 \cdot 10^6 : 70.2 \cdot 10^6 = 1 : 25$. Our assumption is also supported by

widely accepted benchmarks SP^2Bench [23] and BSBM [5] which are based on real-world scenarios. The third point is of course only meaningful if the type is not `rdfs:Resource`. It is based on the assumption that similar entities share similar properties and, therefore, all entities are relevant to a query if one of them is. The thought behind the last one being that queries are likely to be a combination of star-like subqueries. As a consequence, a query engine has to process several triples with the same subject.

The starting point of our storage model is the lowest level of database design, the physical data organization on database pages (page for short). Developing the storage model we focused mainly on reducing the number of join operations during query processing and the number of pages loaded from secondary storage. Instead of having tuples with a predefined set of attributes on a database page as it is the case in relational databases we pay tribute to the openness of the RDF data model and store a resource (subject) together with all properties and their values on a page. Due to this organization of triples the query engine can easily determine the one page containing all triples of a given resource and does not need to perform join operations to retrieve them. In the unlikely event that the triples belonging to a resource do not fit onto a page – a 32k page can store about 8k triples – then the system creates an overflow page. As a consequence, the storage model is independent of the underlying RDF schema, e.g., the system can easily manage optional and multivalued properties. To optimize the number of IO operations further we also group resources according to their `rdf:type` because we think that queries refer often to same kind of entities. Both design decisions increase the probability that a query engine finds relevant triples on the currently processed page or on a recently loaded one.

In the following we formalize the mapping of RDF graphs to database pages. First, we introduce the terms database page and database.

Definition 1 (Database page). *A* database page p *is a persistent storage area of a predefined size* $\|\mathsf{p}\|$*. We refer to the set of all pages as the symbol* $\mathcal{P}$*.*

Each database page is assigned a unique identifier. To express that a page p contains a triple of an RDF graph, $t \in G$, we write $t \in \mathsf{p}$. We refer to the size of a page as $\|\mathsf{p}\|$ and to the number of triples per page as $|\mathsf{p}|$. We use the notations $\|t^s\|$, $\|t^p\|$ and $\|t^o\|$ to note the required space for the subject, predicate, or object of a triple t, respectively.

Definition 2 (Database). *A database* $\mathcal{D}$ *is defined as a set of database pages:* $\mathcal{D} = \{\mathsf{p} : \mathsf{p} \in \mathcal{P}\}$*.*

We write $|\mathcal{D}|$ to refer to the number of pages belonging to a database. Finally, the following definition describes how triples are distributed onto pages.

Definition 3 (Storage mapping). *Let* $G_1, \ldots, G_n \in \mathcal{G}$ *be RDF graphs and* $\mathcal{D}$ *a database. The triples of the graphs are stored w.r.t. the following criteria:*

(i) All triples t *of a page* p *belong to the same graph* G_i*:*

$$\forall \mathsf{p} \in \mathcal{D} \forall t_i, t_j \in \mathsf{p} \exists G_i \in \{G_1, \ldots, G_n\} : t_i \in G_i \wedge t_k \in G_i$$

(ii) All resources t^s of a page p *have the same* `rdf:type` τ*:*

$$\forall \mathsf{p} \in \mathcal{D} \forall t_i, t_j \in \mathsf{p} : \tau(t_i^s) = \tau(t_k^s)$$

(iii) All triples t having the same subject t^s are stored on the same page p*:*

$$\forall G_i \in \{G_1, \ldots, G_n\} \forall t_i, t_k \in G_i : t_i^s = t_k^s \wedge t_i \in \mathsf{p}_1 \wedge t_k \in \mathsf{p}_2 \Rightarrow \mathsf{p}_1 = \mathsf{p}_2$$

(iv) URIs and literals are normalized using bijective mapping $\iota : \mathbb{I} \cup \mathbb{L} \rightarrow \mathbb{N}$.

Figure 2 illustrates the mapping of an RDF graph to database pages. After normalizing URIs and literals triples are stored on different pages according to the ty
pages

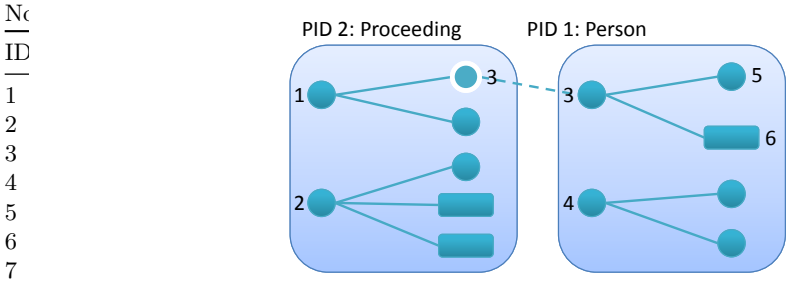

Fig. 2. Mapping of an RDF graph to database pages (properties are omitted)

The internal structure of a page is schematically shown on the left side of Figure 3. It is divided into head and body. The head contains metadata such as the page ID as well as the types and the number of stored subject resources[3]. The body contains the actual triples, however, split into their components. A triple index is used to manage the relationship between subjects and their corresponding predicates and objects (right side of Figure 3). For example, the first entry of the triple index contains the address of the first predicate/object of the first subject, the second entry for the first predicate/object of the second subject, and so on. The advantages of this column-store-like layout are that it eases the implementation (e.g., mapping a page to the domain model in a programming language), subjects are not repeated, and the system can easily access all components of the triples and iterate over them.

Besides the mapping of triples to pages we need also access structures allowing an efficient localization of triples in the database. For this reason we maintain indexes on subjects, predicates, and objects providing the addresses of triples. In the case of indexing subjects it is sufficient to know a mapping from a subject ID to the corresponding page because all triples of a given subject can only occur on a single page. In the case of predicates and objects we use bitset indexes that encode the address of matching triples (cf. Figure 4) – one bitset for each distinct property and object.

[3] A subject may have multiple types but all subjects on a page must have the same.

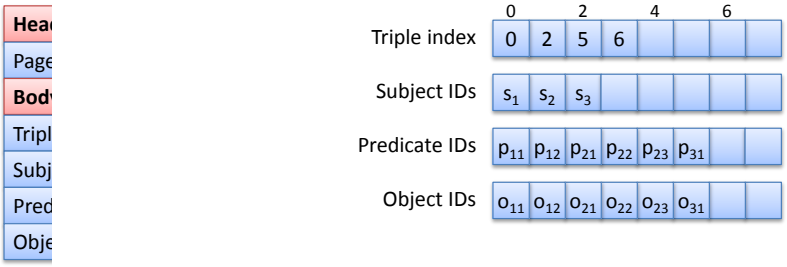

Fig. 3. Internal structure of a page: Schematic representation (left) and detail view of the body (right)

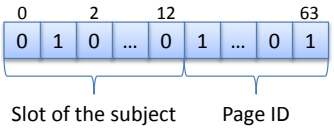

Fig. 4. Interpreting a position within a bitset

Bitset indexes have the advantage that the system can answer multidimensional point queries efficiently by combining them (logical AND or OR). Thus, it is possible locate efficiently triples having a certain combination of predicate and object or subjects having certain objects as property values. We use the algorithms of [29] to process bitsets in a compressed form.

For providing a fulltext search on literals the system has to maintain a dedicated index (e.g., a Lucene index) because bitsets are not very useful in this case. Since text retrieval is well researched we do not consider it in this paper.

5 Operations

In this section we describe the most important operations on the previously defined storage model: querying the repository as well as adding and removing triples. Executing update operations the system has to ensure the consistency of the database according to Definition 3.

Query processing. Regarding query processing we only consider basic graph patterns in this paper because they form the fundamental building block of SPARQL queries [22]. First of all we want to note that any query can be decomposed into joins of star-like patterns – a special form is a single triple pattern. We do not go into details on join algorithms at the moment because they are well-known from relational databases.

To compute the variable bindings for a star-like basic graph pattern (BGP) we distinguish two cases: (1) patterns with a given subject and (2) patterns with a variable as subject. For queries of the first type (lines 4–6, Algorithm 1) the query engine uses the subject index to determine the page ID storing the triples with the given subject. Then, it loads this page from secondary storage and generates the variable bindings by matching the BGP and the stored triples. In the second case (lines 8–22) the query engine loads all bitsets corresponding

to the constant predicates and objects of the query and computes their logical AND. Based on the resulting bitset the query engine can determine the pages which may contain relevant triples (line 19). It loads these pages one after the other and scans them for matching subgraphs.

Algorithm 1 matchStarBGP(BGP bgp) : VariableBindings

```
 1: bindings = ∅
 2: tp = bgp.first {first triple pattern}
 3: if (!isVariable(tp.subject)) then
 4:    address = subjectIdx.get(tp.subject)
 5:    page = load(address.pageId)
 6:    bindings = match(page, address.slot, tp)
 7: else
 8:    bitset = ∼ 0 {bitset with all bits set}
 9:    for (TriplePattern tp : bgp) do
10:       if (!isVariable(tp.predicate)) then
11:          bitset &= predicateIdx.get(tp.predicate)
12:       end if
13:       if (!isVariable(tp.object)) then
14:          bitset &= objectIdx.get(tp.object)
15:       end if
16:    end for
17:
18:    pageIds = subjectIdx.get(bitset) {determine the page IDs of subjects}
19:    for (pid : pageIds) do
20:       page = load(pid)
21:       bindings += match(page, tp)
22:    end for
23: end if
24: return bindings
```

Algorithm 1 realizes only a naive approach to determine relevant pages. A real query engine would also consider statistical data about resources. The query engine can furthermore use bitset indexes to support evaluating filter expressions at a very early stage, e.g., before loading any data from secondary storage.

Adding triples. Algorithm 2 gives the pseudocode for adding triples. Its input is a normalized triple constructed by applying the mapping function to the triple. Next, the system checks if the predicate of the new triple is `rdf:type` because this information is required to locate an appropriate page for storing it. If the triple does not contain type information then the type `rdfs:Resource` is assumed as defined in [6] (line 1–4). Depending on the existences of triples having the same subject as the new triple the system has either to load the corresponding page (line 7) or to locate a page with an appropriate type using the predicate index (line 9) – if such a page does not exist then a new one is allocated. The new triple is then added to the page. Lines 13 and 15 handle two special cases: Balancing the pages and restoring the storage constraints. In the first case the system has to ensure that there is enough free space on the page to store the triple because

a page can only store a limited number of triples. In the event of an overflow it allocates a new page and distributes the triples as equally as possible over the two pages. Hereby, it always guarantees the constraints of Definition 3. In the second case the type constraint is violated, e.g., the new triple states a type of the subject different from the types of the page. The new type of the subject resource is determined as follows: $\tau(t^s) = t^o \cup \tau(\mathtt{p})$. This type is used to find a new page for storing all triples with the given subject. Finally, all indexes have to be updated. Adding a completely new triple this step requires only updating three index entries. In the worst case – pages need to be balanced – the system has to update all index entries belonging to moved triples.

Algorithm 2 insertTriple(NormalizedTriple t) : void

```
 1: type = rdf:resource
 2: if (t.predicate = rdf:type) then
 3:     type = t.object
 4: end if
 5: pageId = subjectIdx.get(t.subject)
 6: if (pageId) then
 7:     page = load(pageId)
 8: else
 9:     page = findOrAllocatePage(type)
10: end if
11: addTriple(page, t)
12: if (type ∈ τ(page)) then
13:     balancePage(page)
14: else
15:     restoreTypeContraint(page, t) {pages are also balanced}
16: end if
17: updateIndexes(t)
```

Deleting triples. As illustrated in Algorithm 3 the system uses first the subject index to locate the page containing the subject of the triple to be removed. The corresponding page is then loaded from secondary storage and the triple is deleted from the page (lines 3 and 4). Deleting triples can lead to underfull pages. To avoid loading many almost empty pages during query processing the system balances pages and ensures a minimal allocation. Furthermore, the predicate of the triple may be `rdf:type`. In this case the system has to restore the consistency of the repository (line 6). Finally, it has also to update the indexes – the best and worst cases are similar to adding a triple.

6 Analysis

In this section we first estimate the disc space needed to store RDF data. We look then at the complexity of the operations described in the previous section.

Algorithm 3 deleteTriple(NormalizedTriple t) : void

```
 1: pageId = subjectIdx.get(t.subject)
 2: if (pageId) then
 3:    page = load(pageId)
 4:    deleteTriple(page, t)
 5:    if (t.predicate = rdf:type) then
 6:       restoreTypeContraint(page, t) {pages are also balanced}
 7:    else
 8:       balancePage(page) {removes empty pages}
 9:    end if
10:    updateIndexes(t)
11: end if
```

6.1 Data complexity

We measure data complexity of our storage model in terms of allocated database pages. To estimate the number of pages needed for storing an RDF graph we first have to know how many triples can be stored onto a page. In the following we consider only the size of the body of a page because the size of the head is almost constant and negligible small in relation to the size of the body. Thus, the number of triples that fit onto a page depends on the page size $\|\mathsf{p}\|$ and the required space for triples $m\|t^s\| + n(\|t^p\| + \|t^o\|)$ with m being the number of subjects and n being the number of triples. Since subject IDs are not repeated we have a minimal (formula 1) and a maximal (formula 2) number of triples per page, e.g., all triples have a distinct or the same subject, respectively. If the space of a page is fully allocated then we can calculate these values as follows:

$$\|\mathsf{p}\| = m\|t^s\| + n(\|t^p\| + \|t^o\|)$$

$$\overset{m=n}{\Longrightarrow} \quad n_{min} = \frac{\|\mathsf{p}\|}{\|t^s\| + \|t^p\| + \|t^o\|} \tag{1}$$

$$\overset{m=1}{\Longrightarrow} \quad n_{max} = \frac{\|\mathsf{p}\| - \|t^s\|}{\|t^p\| + \|t^o\|}$$

$$\overset{\|\mathsf{p}\| \gg \|t^s\|}{\Longrightarrow} \quad n_{max} \approx \frac{\|\mathsf{p}\|}{\|t^p\| + \|t^o\|} \tag{2}$$

In Table 2 we compiled the approximate minimal and maximal numbers of triples per page for typical page sizes. In this calculation the sizes of an ID and an entry in the triple index are 8 and 2 bytes, respectively.

Given the above formula we can derive formula for the minimal and maximal number of pages required to store an RDF graph G (formula 3 and 4, respectively). However, there is a lower bound (formula 5) for even small graphs that is induced by Definition 3; the system has to allocate at least one page for each type of resource (τ_G).

$\|\mathsf{p}\|$	8 kB	16 kB	32 kB	64 kB
$\sim n_{min}$	300	600	1,200	2,400
$\sim n_{max}$	500	1,000	2,000	4,000

Table 2. Number of triples per page

$$|\mathcal{D}|_{min} = \sum_{k \in \tau_G} \frac{|\{t \in G : \tau(t^s) = k\}|}{n_{max}} \tag{3}$$

$$|\mathcal{D}|_{max} = \sum_{k \in \tau_G} \frac{|\{t \in G : \tau(t^s) = k\}|}{n_{min}} \tag{4}$$

$$|\mathcal{D}|_{lb} = |\tau_G| \tag{5}$$

Besides the space required to store the triples the indexes for subjects, properties, and objects occupy disc space as well. As already mentioned the subject index contains only a mapping of subject IDs to page IDs; thus, a B-tree can be used. Based on [3] the data complexity is $O(N)$, N being the number of indexed values. According to [4] DBpedia contains for example $2,8 \cdot 10^6$ resources. Assuming a page size of 16 kB and a pointer size of 8 bytes the subject index would require about 2,740 pages and occupy 42 MB of disc space. Since the nodes of a B-tree have typically a large number of entries, the system can hold at least the first two levels of a B-tree in main memory.

The predicate and object indexes in contrast are based on bitset indexes. Our estimations for the required disc space are based on the WAH compressed bitset [29]. WAH bitsets are organized in words of 32 or 64 bits and are compressed word by word. The author estimates the size of a bitset as $2N$ in the worst case, e.g., all bits are set. To put the size of a bitset into relationship with B-trees, Wu et al. [29] mentions that a B-tree occupies $3N \sim 4N$ words. In the context of our storage model N equals the number of managed triples. Due to the characteristics of RDF data we can assume that the worst case will never occur and the sizes of bitsets are much smaller. We can use the following formula to calculate the number of pages occupied by an index:

$$|\mathsf{p}_{Bitset}| = \frac{\|w\| \cdot 2N}{\|\mathsf{p}\|},\ \|w\| = \text{word size}$$

Further compression strategies are applicable, e.g., K-of-N encoding, but they would require decompression during query evaluation and, thus, increase the overall processing time most likely.

In the proposed storage model we need a bitset for each distinct property and object. Therefore, the system has to manage $O(|P|N + |O|N)$ bitsets, where $|P|$ and $|O|$ are the number of properties and objects, respectively. We use a B-tree to locate the bitset corresponding to a resource. Please note, Lemire et al. assess in [16] that a system can efficiently manage even 100 million bitsets.

As an example, we determined the number of distinct properties and objects of DBpedia (Version 3.4). It contains about $44 \cdot 10^6$ triples, about 52.500 distinct properties, and $9.8 \cdot 10^6$ distinct objects ($5.9 \cdot 10^6$ literals and $3.8 \cdot 10^6$ Resources). Since the required space depends on the number of set bits, we also determined the number of triples having a given property or object.

Only a few properties occurred in more that a thousand triples (cf. Table 3) which is a very acceptable fraction w.r.t. bitsets. Assuming the worst case for

WAH bitsets – each set bit has to represented with 2 bytes – then a bitset with 1000 k set bits would require 250 pages and occupy 8 MB (page size 32 k). In general, however, the system can access almost all bitsets (about 98.2 %) with a single load operation.

cardinality	$> 1000k$	$500k$	$> 100k$	$> 50k$	$> 10k$	> 5000
absolute	3	3	59	65	449	379
percentage	0.006 %	0.006 %	0.112 %	0.129 %	0.854 %	0.721 %
cardinality	> 1000	> 500	> 100	> 50	> 10	≤ 10
absolute	1688	1212	6560	4361	9098	32546
percentage	3.209 %	2.304 %	8.292 %	5.191 %	17.298 %	61.879 %

Table 3. Number of triples per property in DBpedia

Analyzing DBpedia w.r.t. objects we only considered resources because literals are separately stored in a fulltext index (cf. Table 4). In contrast to the above numbers there are even fewer triples per object. The largest index has a size of 0.8 MB and requires only 25 pages (page size 32 k). Thus, the system can load actually all bitsets (about 99.9 %) accessing the disc once.

cardinality	$500k$	$> 100k$	$> 50k$	$> 10k$	> 5000
absolute	0	10	12	105	125
percentage	0.000 %	$3 \cdot 10^{-4}$ %	$3 \cdot 10^{-4}$ %	0.003 %	0.003 %
cardinality	> 1000	> 500	> 100	> 50	> 10	≤ 10
absolute	991	1228	10057	12010	93049	$3.8 \cdot 10^6$
percentage	0.026 %	0.032 %	0.261 %	0.311 %	2.412 %	96.953 %

Table 4. Number of triples per object in DBpedia

6.2 Complexity of operations

In the following we analyze the complexity of the operations defined in Section 5 which we measure in number of pages loaded from and written to secondary storage, including data pages as well as index pages. Before going into details of operations, we want to note that the complexity of accessing B-trees is $O(\log_b N)$ where b is the fanout and N is the number of indexed values [3] because the system has often to access B-trees to locate data pages.

Query processing. In the following we analyze only the complexity of Algorithm 1 which constructs variable bindings for star-like basic graph patterns. Since arbitrary basic graph patterns are decomposed into star-like pattern we have to add the complexity of joining the variable bindings for arbitrary basic graph pattern.

The complexity of Algorithm 1 depends on if the subject of the pattern is given. If it is known then the query engine has only to access the subject index (B-tree) to find the corresponding data page. Independently of the pattern size, this page contains all triples to answer the query. Thus, the complexity is $O(\log_b N)$.

In the case that the subject is not given the complexity is made up of (1) identifying the data pages to be loaded and (2) loading these pages to match the pattern. To identify the relevant pages the query engine combines the bitsets of the constant predicates and objects, e.g., a pages can only be relevant if it contains all resources contained in the BGP. This step requires to locate the relevant bitsets and to load them. In the worst case the complexity is as follows:

$$O(\underbrace{(|t_Q^p| + |t_Q^o|)\log_b N}_{\#\text{accesses B-tree}} + \underbrace{(|t_Q^p| + |t_Q^o|)2N)}_{\#\text{accesses bitsets}})$$

where $|t_Q^p|$ and $|t_Q^o|$ are the numbers of constant predicates and objects, respectively. The $2N$ in the second part is the complexity of accessing the bitsets. As illustrated in the previous section these values are almost always very small, e.g., a page per bitset in the worst case. Thus, the complexity of step (1) depends only on the number of constants of the BGP but not on the size of the database. The complexity of step (2) is determined by the size of the solution $O(|S|)$, e.g., in the worst case the query engine has to access a page per triple. However, since we cluster the triples according to the type of the subject the worst case will occur rarely. As a result the complexity of both steps is

$$O((|t_Q^p| + |t_Q^o|)\log_b N + (|t_Q^p| + |t_Q^o|)2N + |S|)$$

Adding and deleting triples Considering both operations only the following operations are relevant for determining their complexity: (1) Locating and loading of the affected page and writing it back, (2) updating the subject index, and (3) updating the predicate and object indexes. Regarding the first two steps the complexity is dominated by the operations on B-tree of the subject index ($O(\log_b N)$ because the system needs to read and to write only a few pages, e.g., one page in the best case and two pages if balancing is required. In the worst case of a type conflict the system has to find an appropriate page which requires accessing the bitset of `rdf:type` and the one of the type resources. The complexity is comparable to the one of a query. Considering the second step the system has update two bitsets in the best case resulting in $\frac{2 \cdot 2N}{\|\mathsf{p}\|} + \frac{2 \cdot 2N}{\|\mathsf{p}\|}$ read and write operations – all bits being set. In the worst case when balancing is required then the system has to access more pages. Their number depends on the number of distinct predicates objects of the moved triples.

7 Evaluation

We implemented the proposed storage model in Java and run some preliminary test to evaluate our approach. As illustrated in Figure 5 the key components of our system are based on existing software. At the lowest level we use a Berkeley DB for storing database pages, the subject index, as well as the bitsets of the predicate and object indexes. The main reasons for a Berkeley DB are that it can efficiently manage key value pairs – values can be arbitrary objects – and we wanted to focus on implementing our storage model.

On top of the Berkeley DB we implemented a page manager and an index manager. The page manager is responsible for mapping page IDs (key) to pages (value) and to convert them to Java objects. The index manager maintains the subject, predicate, and object indexes. In case of the subject index the key is a resource ID and the value a page ID; in case of the others the key is a resource ID and the value is a bitset.

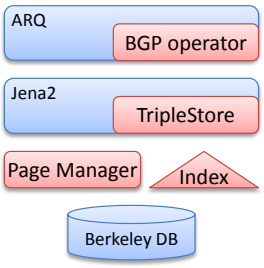

Fig. 5. Architecture of the RDF repository

To reduce the effort for implementing components of an RDF repository such as the RDF data model and SPARQL query engine we adapted Jena2 and ARQ [26], a SPARQL processor for Jena. Inside Jena we created the infrastructure for reading and writing RDF graphs using the presented storage model, e.g., a `TripleStore` based on normalized triples. In the query processor ARQ we had to replace the implementation of filtered BGPs because the original classes could not take advantage of the new storage model. Thus, we implemented an operator that recognizes star-like BGPs and evaluates them separately. If the query contains several star-like pattern the join is currenty performed by ARQ.

We run some preliminary tests to validate the feasibility of our storage model. Table 6 presents the processing time of evaluating star-like BGPs of different sizes and varying result sizes; the subject was always a variable. The underlying dataset was generated using BSBM [5] and containing 300 k triples. As a reference we also present the processing times of the same queries using Jena SDB [26] and RDF-3X [20]. The first one we chose because our system is based on Jena + ARQ and RDF-3X because it is one of the most performant RDF repositories. Looking at the result we feel confident that our storage model is competitive. While the processing time of the other two systems increases with larger BGPs the one of our system remains almost constant. The last query shows that retrieving larger result sets increases the processing time significantly in our system. Our explanation for this behavior is that the data distribution is not favorable, e.g., a clustering of resources based on `rdf:type` is not sufficient.

Query	$\|Q\|$	$\|S\|$	SDB	RDF-3X	ours
Q_1	2	1,000	415	30	110
Q_2	4	1,000	849	45	91
Q_3	8	1,000	1577	63	151
Q_4	11	402	4,440	138	104
Q_5	5	20,000	11,000	846	2,042

Fig. 6. Processing time (ms) of BGPs of different sizes and selectivities

Query	$\|Q\|$	|Filter|	σ	TDB	ours
Q_1	3	0		11,850	11,800
Q_2	7	0		19,575	25,590
Q_3	3	1	2.5 %	4,057	195
Q_4	7	1	2.5 %	4,440	452
Q_5	3	9	12 %	16,174	1,151
Q_5	7	9	12 %	17,253	2,751

Fig. 7. Processing time (ms) of filtered BGPs

Furthermore, we evaluated the processing of filtered BGPs to validate the feasibility of using bitset indexes for predicates and objects. Since the indexes store only URIs and bitsets are best suited for testing on equality the queries consisted of a star-like pattern with and a filter expression containing zero, one,

or nine predicates. For example, such filters are used to restrict resources to certain types. We constructed the filter expressions manually so that they selected about 2.5 % or 12 % of the dataset (3.5 million triples). Table 7 gives the results for these queries – in this test set we compared our system with Jena TDB only. Comparing the processing times of the queries without a filter (rows 1 and 3) and the one with a filter we are confident that the query engine was able to reduce the number of candidate pages significantly.

8 Conclusion and future work

In this paper we presented a new approach for managing RDF data. Our storage model is based on the assumption that a query will often involve several properties of a resource. Thus, we store all triples having the same subject on the same database page. To ensure an efficient retrieval of triples we index them using bitsets. We implemented our storage model using Jena and ARQ as framework. Although we run only preliminary test – not being exhaustive – they indicate that the presented model has advantages in computing the results of star-like BGPs. The system can also process filter expression testing on equality efficiently.

However, our experiments also showed us that we are still open issues which we address in our future work. For example, joins between two BGPs are currently computed by ARQ; its join algorithm performs very poorly. Another important topic is reducing the load time significantly. Besides these two topics we also work on indexing graph pattern which could span over several star-like pattern. In combination with our storage model the key advantage is that only little information is needed even to index complex pattern, e.g., an index entry would only consist of addresses of subject resources.

References

1. Jena TDB, August 2011. http://openjena.org/TDB/.
2. S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The RDFSuite: Managing Voluminous RDF Description Bases. In S. Decker et al., editors, *2nd International Workshop on the Semantic Web*, pages 1–13, May 2001.
3. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET, Workshop on Data Description, Access and Control*, pages 107–141, New York, NY, USA, 1970. ACM.
4. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the web of data. *Web Semantics*, 7(3):154–165, 2009.
5. C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
6. D. Brickley and R. Guha. RDF Vocabulary Description Language: RDF Schema. http://www.w3.org/TR/rdf-schema/, February 2004. W3C Recommendation.
7. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In I. Horrocks and J. A. Hendler, editors, *ISWC*, volume 2342 of *LNCS*. Springer, June 2002.

8. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In K. Böhm et al., editors, *Proceedings of VLDB*, pages 1216–1227. ACM, September 2005.
9. I. F. Cruz, V. Kashyap, S. Decker, and R. Eckstein, editors. *Proceedings of the First International Workshop on Semantic Web and Databases*, September 2003.
10. Franz Inc. Allegrograph rdfstore^{TM}, 2010. Zuletzt besucht am 16.3.2010.
11. S. Harris. SPARQL query processing with conventional relational database systems. In M. Dean et al., editors, *Scalable Semantic Web Knowledge Base Systems*, volume 3807 of *LNCS*, pages 235–244, November 2005.
12. A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Proceedings of the 3rd Latin American Web Congress*. IEEE Press, October 2005.
13. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In K. Aberer et al., editors, *ISWC/ASWC*, volume 4825 of *LNCS*, pages 211–224. Springer, November 2007.
14. P. Hayes. RDF Semantics, February 2004. W3C Recommendation.
15. G. Klyne and J. J. Carroll. RDF: Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/, 2004. W3C Recommendation.
16. D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.
17. B. Liu and B. Hu. HPRD: A High Performance RDF Database. In K. Li et al., editors, *Int. Conference Network and Parallel Computing*, volume 4672 of *LNCS*, pages 364–374. Springer, September 2007.
18. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Int. Conference on Information and Knowledge Management*, pages 484–491, New York, NY, USA, 2004. ACM Press.
19. A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *Australasian conference on database technologies*, pages 95–103, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
20. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
21. OpenLink Software. Virtuoso, 2010. Zuletzt besucht am 16.3.2010.
22. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, Januar 2008. W3C Recommendation.
23. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. sp^2bench: A sparql performance benchmark. In *ICDE*, pages 222–233. IEEE, 2009.
24. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. of the VLDB Endowment*, 1(1):1008–1019, 2008.
25. T. Weithöner, T. Liebig, and G. Specht. Storing and Querying Ontologies in Logic Databases. In Cruz et al. [9].
26. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In Cruz et al. [9].
27. D. Wood, P. Gearon, and T. Adams. Kowari: a platform for semantic web storage and analysis. In *Proceedings of XTech 2005*, 2005.
28. G. Wu, J. Li, J. Hu, and K. Wang. System *Pi*: A Native RDF Repository Based on the Hypergraph Representation for RDF Data Model. *Journal of Computer Science and Technology*, 24(4):652–664, July 2009.
29. K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.