Hexastore: Sextuple Indexing for Semantic Web Data Management *

Cathrin Weiss
Department of Informatics
University of Zurich
CH-8050 Zurich, Switzerland
lastname@ifi.uzh.ch

Panagiotis Karras
School of Computing
National University of Singapore
117543, Singapore
lastname@comp.nus.edu.sg

Abraham Bernstein
Department of Informatics
University of Zurich
CH-8050 Zurich, Switzerland
lastname@ifi.uzh.ch

ABSTRACT

Despite the intense interest towards realizing the Semantic Web vision, most existing RDF data management schemes are constrained in terms of efficiency and scalability. Still, the growing popularity of the RDF format arguably calls for an effort to offset these drawbacks. Viewed from a relational-database perspective, these constraints are derived from the very *nature* of the RDF data model, which is based on a *triple* format. Recent research has attempted to address these constraints using a vertical-partitioning approach, in which separate two-column tables are constructed for each property. However, as we show, this approach suffers from similar scalability drawbacks on queries that are not bound by RDF property value. In this paper, we propose an RDF storage scheme that uses the triple nature of RDF as an asset. This scheme enhances the vertical partitioning idea and takes it to its logical conclusion. RDF data is indexed in *six* possible ways, one for each possible ordering of the three RDF elements. Each instance of an RDF element is associated with two vectors; each such vector gathers elements of one of the other types, along with lists of the third-type resources attached to each vector element. Hence, a sextuple-indexing scheme emerges. This format allows for quick and scalable general-purpose query processing; it confers significant advantages (up to five orders of magnitude) compared to previous approaches for RDF data management, at the price of a worst-case five-fold increase in index space. We experimentally document the advantages of our approach on real-world and synthetic data sets with practical queries.

1. INTRODUCTION

The vision of the Semantic Web [13] is to allow everybody to publish interlinked machine-processable information with the ease of publishing a web page. The basis for this vision is a standardized logical data model called Resource Description Framework (RDF) [18, 37]. RDF data is a collection of statements, called *triples*, of the form $\langle \mathsf{s}, \mathsf{p}, \mathsf{o} \rangle$, where s is a subject, p is a predicate, and o is an object; each triple states the relation between the subject and the object. A collection of triples can be represented as a directed typed graph, with nodes representing subjects and objects and edges representing predicates, connecting subject nodes to object nodes.

The increasing amount of RDF data on the Web calls for the development of systems customized for the efficient and effective management of such data. Thus, significant efforts have been dedicated to the development of architectures for storing and querying RDF data, called *triple stores* [8, 9, 11, 17, 25, 26, 46, 19, 45, 39, 44].

Still, most existing triple stores suffer from either a scalability defect or a specialization of their architecture for special-type queries, or both. Traditional approaches have either been limited to a memory based storage, or mapped the triple-based format to a relational data base; therewith, they have forgone the opportunity to optimize triple storage, retrieval, and updates to the graph-based format.

In particular, RDF triples were traditionally stored in a giant triples table, causing serious scalability problems. Other approaches tried to attenuate these problems by constructing relational-like tables that gathered many properties together as attributes. Still, this approach did not escape the scalability defects. The most recent effort towards alleviating these deficiencies [5] focused on taming the scalability drawback, following a vertical partitioning approach. In this approach, a triples table (i.e., a relational table with three columns, one for each RDF resource) is rewritten into n two-column tables where n is the number of unique properties in the data. This approach confers a clear advantage for processing queries in which properties appear as bound variables. Still, this vertical partitioning model is strictly property-oriented.

Arguably, property-bound queries are not necessarily the most representative way of querying RDF data. Hence, the methodology of [5] becomes suboptimal for general queries, in which properties may not be bound. Thus, the property orientation of this model renders it unsuitable for answering such general queries.

In this paper, we argue that an efficient RDF storage scheme should offer both scalability in its data management performance and generality in its data storage, processing and representation. To achieve this double goal, we propose a novel approach to RDF data management. Our framework is based on the idea of indexing the RDF data in a multiple-index framework. Two vectors are associated with each

*In memory of Klaus Dittrich.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

RDF element (e.g., subject), one for each of the other two RDF elements (e.g., property and object). Moreover, lists of the third RDF element are appended to the elements in these vectors. In total, six distinct indices are used for indexing the RDF data. These indices materialize all possible orders of precedence of the three RDF elements. In effect, our approach adopts the rationale of vertical-partitioning [5] and multiple-indexing [27, 47] systems, but takes it further, to its logical conclusion. This scheme allows not only property-based two-column tables, but a representation based on any order of significance of RDF resources and properties.

Outline The remainder of this paper is structured as follows. Section 2 presents related work on RDF storage and discusses the deficiencies of previous approaches. Section 3 outlines the motivation for our research. In Section 4 we introduce the Hexastore, our solution for Semantic Web data management. Section 5 presents the experimental results on the performance of a prototype Hexastore and our representation of the state-of-the-art RDF storage approach, using a real-world and a synthetic data set as well as the benchmark queries used in previous work. Finally, Section 6, discusses our results and potential future research directions and in Section 7 we outline our conclusions.

2. BACKGROUND AND RELATED WORK

In this section we present an overview of existing approaches to RDF data management.

2.1 Conventional Solutions

A variety of architectures for the storage and querying of large quantities of RDF metadata have been proposed. However, systems such as the FORTH RDF Suite [8, 9], Redland [11], Sesame [17], 3store [25, 26], Jena [46, 19, 45], DLDB [39], KAON [44], RStar [36], Oracle [20], and rdfDB [22] utilize a traditional relational databases or Berkeley DB as their underlying persistent data store [38].

In most of these systems, RDF data is decomposed into a large number of single statements (i.e., triples) that are directly stored in relational or hash tables. Thus, simple *statement-based* queries can be satisfactorily processed by such systems. A statement-based query lacks one or two parts of a triple, and the answer is a set of resources that complement the missing parts. Still, statement-based queries do not represent the most important way of querying RDF data. More complex queries, involving multiple filtering steps, are arguably more important. However, conventional approaches are not efficient on such queries [38].

Besides, systems such as Jena [46, 19, 45] attempt to create relational-like *property tables* out of RDF data; these tables gather together information about multiple properties over a list of subjects. Still, these schemes do not perform well for queries that cannot be answered from a single property table, but need to combine data from several tables [5]. Furthermore, such systems impose a relational-like structure on semi-structured RDF data. However, imposing structure where it does not naturally exist results into sparse representation with many NULL values in the formed property tables. Handling such sparse tables, as opposed to denser ones, requires a significant computational overhead [12].

Concurrent work has studied problems such as constructing scalable publish/subscribe systems for RDF metadata [40] and performing continuous queries over them [35].

2.2 Alternative RDF Data Storage Schemes

Several other pieces of work have suggested RDF data storage schemes that depart from the conventional solutions.

2.2.1 Storing RDF Data as a Graph

A stream of research has investigated the possibility of storing RDF data as a graph [16, 28, 10]. Still, these solutions do not sufficiently address the scalability questions either. A related approach has proposed a path-based approach for storing RDF data [34, 38]. However, these path-based schemes are also ultimately based on a relational database: they store subgraphs into distinct relational tables. Thus, these systems do not provide the scalability necessary for query processing over vast amounts of RDF data. Other related work has focused on measuring similarity within the Semantic Web [33] and using selectivity estimation techniques for query optimization with RDF data [41]; still these techniques, based on main-memory graph implementations, also face scaling limitations [41].

2.2.2 Multiple-indexing Approaches

Harth and Decker [27] proposed storing RDF data based on multiple indices, while taking into consideration *context* information about the provenance of the data. It constructs six indexes that cover all $2^4 = 16$ possible access patterns of *quads* in the form $\{\mathsf{s}, \mathsf{p}, \mathsf{o}, \mathsf{c}\}$, where c a is the *context* of triple $\{\mathsf{s}, \mathsf{p}, \mathsf{o}\}$. This scheme allows for the quick retrieval of quads conforming to an *access pattern* where any of s, p, o, c is either specified or a variable [27]. Thus, it is also oriented towards simple *statement-based* queries; it does not allow for efficient processing of more complex queries.

A similar multiple-indexing approach has been suggested by Wood et al. in the Kowari system [47]. Kowari also stores RDF statements as quads, in which the first three items form a standard RDF triple and a fourth, meta item, describes which model the statement appears in. Anticipating [27], Kowari also identifies six different *orders* in which the four node types can be arranged such that any collection of one to four nodes can be used to find any statement or group of statements that match it [47]. Thus, each of these orderings acts as a compound index, and independently contains all the statements of the RDF store. Kowari uses AVL trees to store and order the elements of these indexes [47].

Still, like [27], the Kowari solution also envisions simple *statement-based* queries. The six orders that it takes into consideration all obey the same *cyclic order*; they do not consider the $4! = 24$ possible *permutations* of the four quad items, neither the $3! = 6$ possible permutations of the three items in a triple. Thus, if the meta nodes are ignored, the number of required indices is reduced to 3, defined by the three cyclic orderings $\{\mathsf{s}, \mathsf{p}, \mathsf{o}\}$, $\{\mathsf{p}, \mathsf{o}, \mathsf{s}\}$, and $\{\mathsf{o}, \mathsf{s}, \mathsf{p}\}$ [47]. These indices cannot provide, for example, a sorted list of the subjects defined for a given property. Thus, Kowari does not allow for efficient processing of more complex queries either. However, the multiple-index approach suggested by these two schemes [27, 47] is, in our opinion, an idea worth considering for further exploration. We will come back to this issue when we introduce our solution in Section 4.

2.2.3 The Vertical-Partitioning Solution

Most recently, Abadi et al. [5] suggested a *vertical partitioning* approach. In this scheme, a triples table is rewritten into n two-column tables, one table per property, where n is the

number of unique properties in the data. Each table contains a subject and an object column [5]. *Multi-valued* subjects, i.e. subjects related to multiple objects by the same property, are thus represented by multiple rows in the table with the same subject and different object values. Each table is sorted by subject, so that particular subjects can be located quickly, and fast merge-joins can be used to reconstruct information about multiple properties for subsets of subjects [5]. In [5], this approach is coupled with a column-oriented DBMS [14, 15, 42] (i.e., a DBMS designed especially for the vertically partitioned case, as opposed to a row-oriented DBMS, gaining benefits of compressibility [4] and performance [7]). Then it confers a clear advantage for processing queries in which properties appear as bound variables. The study in [5] has established this scheme as the state of the art for scalable Semantic Web data management.

Furthermore, Abadi et al. [5] suggest that the object columns of tables in their scheme can also be optionally indexed (e.g., using an unclustered B^+ tree), or a second copy of the table can be created clustered on the object column. Hence, [5] has also suggested a form of multiple indexing for Semantic Web data management. However, this multiplicity is limited within a property-oriented architecture. Besides, this suggestion was not implemented as part of the benchmark system used in [5].

In effect, this vertical-partitioning model is oriented towards answering queries in which the property resource is bound, or, otherwise, the search is limited to only a few properties. In fact, while Abadi et al. [5] argue convincingly against the property-table solutions of [46, 19, 45], the property-based two-column-table approach they introduce shares most of the disadvantages of those property-table solutions itself. In fact, the two-column tables used by [5] are themselves a special variation of property tables too. Specifically, these two-column tables are akin to the *multi-valued property tables* introduced in [45]; namely, the latter also store single properties with subject and object columns. In this respect, the most significant novelty of [5] has been to integrate such two-column property tables into a column-oriented DBMS. Besides, Wilkinson [45] did observe the deficiency of the property tables when it comes to unknown-property queries. Thus, both the general property-table approach of [46, 19, 45] and the specific, column-oriented property-table scheme of [5] are bound to perform poorly for queries that have unknown property values, or for which the property is bound at runtime.

Abadi et al. [5] do repeatedly observe the problem of having non-property-bound queries, but do *not* effectively address it. The drawbacks of other schemes in cases where a query does not restrict on property value, or the value of the property is bound during query execution, do in fact apply to the scheme of [5] as well: All two-column tables will have to be queried and the results combined with either complex union clauses, or through joins. Thus, queries that have such unspecified property values (or with such values bound at runtime) are generally problematic for the RDF storage architecture proposed in [5].

Characteristically, the experimental study in [5] was based on the assumption that only a set of 28 out of the 221 unique properties in the studied library catalog data set [2] were interesting for the study; queries that were not property-bound were run on this limited set only. Unfortunately, such an assumption is hard to be realized in a real-world setting. Thus, there is a need for scalable semantic web data management that will not depend on assumptions about the number of properties in the data or the (property-bound) nature of the executed queries.

As a concrete example, the raw data in table of Figure 1(a) show an instance of triples of an LUBM-like [23] data set; the triples in the table store academic information about a group of four people. Noticeably, not all properties are defined for all subjects in the table. A possible, interesting query over these data is to ask what kind of relationship, if any, a certain person has to MIT (upper part of Figure 1b); another interesting query looks for people who have the same relationship to Stanford as a certain person has to Yale (lower part of Figure 1b). Furthermore, one may ask for other universities where people related to a certain university are involved; or for people who hold a degree, of any type, from a certain university; or for people who are anyhow related with both of a pair of universities, and so on. All these queries are not property-bound; they require gathering information about several properties. Besides, some of them require complex joins on the lists of subjects related to a certain object through several properties. Interestingly, in such queries the binding originates neither from a property, nor from a subject, but from an object (e.g., a certain university).

Subj	Property	Obj
ID1	type	FullProfessor
ID1	teacherOf	'AI'
ID1	bachelorFrom	'MIT'
ID1	mastersFrom	'Cambridge'
ID1	phdFrom	'Yale'
ID2	type	AssocProfessor
ID2	worksFor	'MIT'
ID2	teacherOf	'DataBases'
ID2	bachelorsFrom	'Yale'
ID2	phdFrom	'Stanford'
ID3	type	GradStudent
ID3	advisor	ID2
ID3	teachingAssist	'AI'
ID3	bachelorsFrom	'Stanford'
ID3	mastersFrom	'Princeton'
ID4	type	GradStudent
ID4	advisor	ID1
ID4	takesCourse	'DataBases'
ID4	bachelorsFrom	'Columbia'

(a) Example RDF triples

```
SELECT A.property
FROM triples AS A
WHERE A.subj = ID2
  AND A.obj = 'MIT'

SELECT B.subj
FROM triples AS A,
     triples AS B,
WHERE A.subj = ID1
  AND A.obj = 'Yale'
  AND A.property = B.property
  AND B.obj = 'Stanford'
```

(b) SQL queries over (a)

Figure 1: Sample RDF data and queries.

3. MOTIVATION

Arguably, queries bound on property value are not necessarily the most interesting or popular type of queries encountered in real-world Semantic Web applications. In fact, it is usually the case that specific properties may largely be the *unknown* parts of queries; indeed, one may query for relationships between resources without specifying those relationships (consider, for example, the applications arising with the proliferation of social networks).

Still, as we have seen, existing RDF data storage schemes are not designed with such queries in mind. Instead, they typically center around the idea of gathering together triples having the same property, or triples having the same subject (e.g., [46, 19, 45, 5]); alternatively, some may offer an object-subject hash key for identifying properties [27, 47]. However, no existing scheme can directly provide a functionality such as giving a list of subjects or properties related

to a given object. We argue that such functionalities are not only desirable, but also achievable, thanks to the very triple nature of RDF data. Namely, a set of six indices (i.e., 3!) covers all possible accessing schemes an RDF query may require. Thus, while such a multiple indexing would result into a combinatorial explosion for an ordinary relational table, it is quite practical in the case of RDF data. In the next section we proceed to define the Hexastore, an RDF indexing scheme that implements this idea.

4. HEXASTORE: SEXTUPLE SEMANTIC WEB DATA INDEXING

The approach of [5] treats RDF triples as characterized primarily in terms of their *property* attribute. In this section we discuss an approach that does not treat property attributes specially, but pays equal attention to all RDF items.

4.1 Description of a Hexastore

In order to address the aforementioned difficulties, we take the vertical partitioning idea further, to its full logical conclusion. The resulting solution does not discriminate against any RDF element; it treats subjects, properties and objects equally. Thus, each RDF element type deserves to have special index structures built around it. Moreover, every possible *ordering* of the importance or precedence of the three elements in an indexing scheme is materialized. The result amounts to a *sextuple* indexing scheme. We call a store that maintains six such indices a *Hexastore*. Each index structure in a Hexastore centers around one RDF element and defines a prioritization between the other two elements. Thus, the Hexastore equivalent of a two-column property table can be either indexed by subject and allow for a list of multiple object entries per subject, or vice versa. Hence a particular *header* for property p is associated to a vector of subjects $\mathbf{s}(p)$ and to a vector of objects $\mathbf{o}(p)$; in the former case, a list of associated objects $o_{\mathbf{s}(p)}$ is appended to each subject entry in the vector; in the latter, a list of associated subjects $s_{\mathbf{o}(p)}$ is appended to each object entry.

Besides, a Hexastore does not take *any* prioritization of the three triple attributes for granted. RDF triples are not assumed to exist in a property-based universe. Hence, a Hexastore creates not only property-*headed* divisions, but also subject-headed and object-headed ones. In the former case, a given subject header s is associated to a *property vector* $\mathbf{p}(s)$ and to an *object vector* $\mathbf{o}(s)$; a list of associated objects $o_{\mathbf{p}(s)}$ is appended to each entry in the property vector; likewise, lists of associated properties $p_{\mathbf{o}(s)}$ are appended to entries in the object vectors. Interestingly, for a given subject s_x and a property p_y, the object list $o_{\mathbf{p}_y(s_x)}$ in this subject-headed indexing is identical to the object list $o_{\mathbf{s}_x(p_y)}$ in the property-headed indexing. Thus, only a *single* copy of each such list is needed in the indexing architecture.

Likewise, in the latter, object-headed division, each object o in the data set is linked, as a header, to a *subject vector* $\mathbf{s}(o)$ and to an *property vector* $\mathbf{p}(o)$. Now lists of associated properties $p_{\mathbf{s}(o)}$ are appended to subject vector entries; similarly, lists of associated subjects $s_{\mathbf{p}(o)}$ are appended to entries in the object vectors. Again, a list of properties $p_{\mathbf{s}_y(o_x)}$ for object o_x and subject s_y in this object-headed indexing is identical to the property list $p_{\mathbf{o}_x(s_y)}$ of the subject-headed indexing. Similarly, a list of subjects $s_{\mathbf{p}_x(o_y)}$ for property p_x and object o_y in the object-headed indexing is identical to the subject list $s_{\mathbf{o}_y(p_x)}$ featured in the property-headed indexing. Hence, such lists do not have to be replicated; only single copies of them need to be maintained.

Putting it all together, the information for each triple $\langle \mathsf{s}, \mathsf{p}, \mathsf{o} \rangle$ in the data is represented in *six* ways, one for each possible prioritization of the three elements. We name these $3! = 6$ prioritization ways by acronyms made up from the initials of the three RDF elements in the order of each prioritization. For example, the indexing that groups data into *subject*-headed divisions with *property* vectors and lists of *objects* per vector is the spo indexing. Likewise, the osp indexing groups data into object-headed divisions of subject vectors with property lists per subject. In this framework, the column-oriented vertical partitioning scheme of [5], in which two-column property tables are sorted by subject, can be seen as a special, simplified variant of our pso indexing. The six indexing schemes are then called spo, sop, pso, pos, osp, and ops.

We employ a *dictionary encoding* similar to that adopted in [17, 20, 5]. Instead of storing entire strings or URIs, we use shortened versions or *keys*. In particular, we map string URIs to integer identifiers. Thus, apart from the six indices using identifiers (i.e., keys) for each RDF element value, a Hexastore also maintains a mapping table that maps these keys to their corresponding strings. This mapping amounts to a dictionary encoding of the string data.

According to the preceding discussion, three pairs of indices in this six-fold scheme share the same terminal lists. Thus, the spo and pso indices share the same terminal object-lists; the osp and sop indices share the same terminal lists of properties; lastly, the pos and ops indices point to the same terminal subject-lists. In effect, the worst-case space occupied by these six indices in a Hexastore is *not* six times as large as the space occupied when storing the keys in a simple triples table. In the worst case, the space involves a *five*-fold increase in comparison to a triples table. This is due to the fact that the key of each of the three resources in a triple appears in *two* headers and *two* vectors, but only in *one* list. For instance, each subject appears in the headers of the spo and sop indices and in the vectors of the pso and osp indices, but only in a single list used by both the pos and ops indices. In the worst case, assume a triple $\langle \mathsf{s}_i, \mathsf{p}_j, \mathsf{o}_k \rangle$, such that each of s_i, p_j, and o_k appears only once in the given RDF data set; then the key of each resource in this triple requires *five* new entries in the Hexastore indexing. Hence, the worst-case space requirement of a Hexastore is quintuple of the space required for storing the keys in a triples table. In practice, the requirement can be lower, since most RDF resources do *not* appear only once in a given data set.

As an example that illustrates the above discussion, the ops indexing for the data in Figure 1 includes a property vector for the object 'MIT'. This property vector contains two property entries, namely bachelorFrom and worksFor. Each of these property entries is appended with a list of related subjects; in the particular example each list contains one item only, namely ID1 for the bachelorFrom property and ID2 for the worksFor property. A similar configuration holds for the object 'Stanford' in the table. Besides, the osp indexing includes a subject vector for the object 'Stanford' as well. There are two subject entries in this vector, namely ID2 and ID3. Each of those entries holds a list of associated properties. These lists contain the single elements phdFrom and bachelorFrom, respectively. The same pattern

is repeated for all other indexing schemes and for all other resources in the table.

Figure 2 presents a general example of spo indexing in a Hexastore. A subject key s_i is associated to a *sorted* vector of n_i property keys, $\{p_1^i, p_2^i, \ldots, p_{n_i}^i\}$. Each property key p_j^i is, in its turn, linked to an associated *sorted* list of $k_{i,j}$ object keys. These objects lists are accordingly shared with the pso indices. The same spo pattern is repeated for every subject in the Hexastore. Moreover, analogous patterns are materialized in the other five indexing schemes.

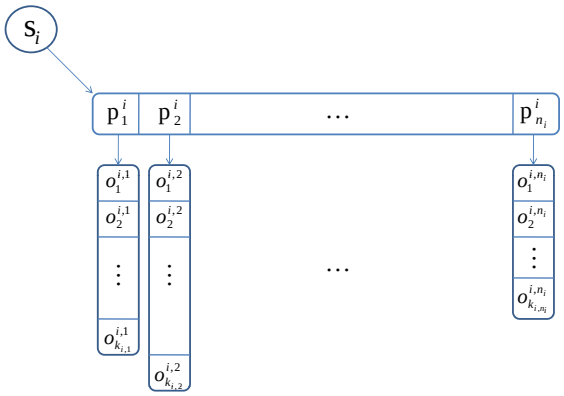

Figure 2: spo indexing in a Hexastore

4.2 Argumentation

The main advantages of a Hexastore in relation to earlier RDF data management schemes [8, 9, 11, 17, 25, 26, 46, 19, 45, 39, 44, 38, 36, 20, 34, 27, 47, 22, 5] can be outlined as follows:

- **Concise and efficient handling of multi-valued resources.** Any resource that would appear as a multi-valued attribute in a relational rendering of RDF is naturally accommodated in the Hexastore. Namely, the lists that appear as the terminal items of any Hexastore indexing can contain multiple items. Moreover, this approach is more concise that the one employed with the two-column, multi-valued property tables proposed by [46, 19, 45]. In this scheme, if a subject is related to multiple objects by the same property, then each distinct value is listed in a successive row in the table for that property. This type of a property table is named *multi-valued property table* and is used to store properties that have maximum cardinality greater than one, or unknown, in [46, 19, 45]. Still, using multiple rows for the same subject is not the most concise solution. The Hexastore stores multi-valued resources in the most concise manner. [5] correctly noted that accommodating multi-valued resources is hard for any relational-based architecture and provided a solution based on run-length encoding. Given the semi-structured nature of Semantic Web data, an approach especially customized for the multi-valued case is needed. The Hexastore provides such a solution.

- **Avoidance of NULLs.** Only those RDF elements that are relevant to a particular other element need to be stored in a particular index. For example, properties not defined for a particular object o_i do not appear in the ops indexing for o_i. Thus, no storage space is wasted for NULL values, as it happens with many relational renderings of RDF data. Again, given the semi-structured nature of Semantic Web data, and the related abundance of NULL values and sparsity in a relational rendering of them, an approach that avoids this sparsity is called for. The Hexastore sufficiently addresses this issue too.

- **No ad-hoc choices needed.** Most other RDF data storage schemes require several ad-hoc decisions about their data representation architecture. For example, a storage scheme may require a decision about which properties to store together in the same table, as in [46, 19, 45], or which path expressions to represent in a relational scheme, as in [38, 34]. The Hexastore eschews the need for such ad-hoc decisions. The architecture of a Hexastore is uniquely and deterministically defined by the RDF data at hand; no ad-hoc decisions that would variably affect its performance need to be made.

- **Reduced I/O cost.** Depending on the bound elements in a query, a mostly efficient computation strategy can be followed; such a strategy accesses only those items that are relevant to the query. In contrast, other RDF storage schemes may need to access multiple tables which are irrelevant to a query; for example, property-based approaches need to access all property tables in a store for queries that are not bound by property; besides, answering a query bound by object is also problematic with most existing storage schemes. Thanks to its sextuple indexing, the Hexastore eliminates redundant data accesses.

- **All pairwise joins are fast merge-joins.** The keys of resources in all vectors and lists used in a Hexastore are sorted. Thus a sorted order of all resources associated to any other single resource, or pair of resources, is materialized in a Hexastore. In consequence, every *pairwise* join that needs to be performed during query processing in a Hexastore is a fast, linear-time merge-join. By contrast, other storage schemes necessitate complex joins over unsorted lists of items extracted with heavy computation cost.

- **Reduction of unions and joins.** A conventional RDF storage scheme would require several union and join operations in order to derive, for example, a list of subjects related to two particular objects through *any* property (e.g., all people involved in both of two particular university courses). For instance, the limited multiple-indexing solutions of [27, 47] do not provide fast access to such a result; instead, they would need to separately access all statements defined for each of the two objects in question and cross-join the resulting subject lists. Similarly, the property-oriented solutions of [46, 19, 45] and [5] would need to access *all* property tables in order to identify instances of ⟨subject, object⟩ pairs that match the two desired objects, union the results for each object, and join them. By contrast, the Hexastore directly supplies the answer to such a query by linearly merge-joining two subject vectors in osp indexing. A similar simplicity of query processing applies in other cases.

The prime drawback of the Hexastore lies in its use of storage space; despite the already mentioned advantages of conciseness, the Hexastore still entails a worst-case five-fold increase in storage space compared to a triples table. This quintuple increase may appear to be a redundancy at first sight. However, we argue that the time-efficiency benefits it confers in query processing are worth the space overhead. These benefits are shown in our experimental section. Besides, the increase in storage is *guaranteed* to be at most quintuple. It does not depend on any parameter, such as the out-degree of a resource, or any ad-hoc choices made in the architecture. Thus, a data engineer can count on a predictable storage requirement for a Hexastore.

Furthermore, in our opinion, this six-fold representation of RDF data is worthwhile for a more fundamental reason; namely, it justifies the choice of RDF as a data model in the first place. In particular, the Hexastore turns the table over on a common argument used against RDF from an RDBMS point of view. That is, it is usually argued that RDF entails an inherent performance disadvantage, being intrinsically limited as a data model in itself. Still, rendering RDF data into this sextuple mapping allows for a boost in performance that a relational database would not be able to afford. Namely, many of the joins that have to be executed during query processing in an RDBMS are not merge-joins. This state of affairs causes a significant performance overhead. However, in a hexastore *all* pairwise joins are rendered as merge-joins, using the appropriate indices in each case. Thus, a hexastore not only allows for the efficient and concise representation of semi-structured data, avoiding the NULLS and sparsity that any relational-oriented solution would incur, but may also allow for more efficient query processing than an RDBMS in certain cases.

Besides, an attempt to render a relational database into a multitude of representations analogous to the sextuple rendering of RDF would result into an explosion of storage requirements; namely, a table of n attributes would require an $n!$-fold rendering. Thus, our exhaustive sextuple-indexing scheme creates an advantage out of the triple nature of the RDF data model; this triple nature is is usually seen as a disadvantage. Thus, a liability of the model is turned into an asset. The price of a worst-case five-fold increase in storage requirements, is, in our opinion, a price worth paying for the performance benefits it confers. Besides, this requirement is well within reach given the current trends of capacity increase in storage technologies.

A particular deficiency of the Hexastore appears when it comes to handling updates and insertions; such operations affect all six indices, hence can be slow. However, existing RDF applications do not involve massive updates.

4.3 Treating the Path Expressions Problem

Abadi et al. [5] have observed that querying path expressions, as common operations on RDF data demand, can be quite expensive and inefficient. This inefficiency is due to the fact that path-expression queries require subject-object joins, as every internal node n in a path serves the subject of an $\langle \mathsf{n}, \mathsf{p}, \mathsf{o} \rangle$ triple but as the object of another, $\langle \mathsf{s}, \mathsf{p}', \mathsf{n} \rangle$ triple. In the vertical-partitioning scheme of [5], the two-column tables for the properties involved in the path expression need to be joined together. All these joins are joins between the subject-column of one two-column property table and the object-column of another property table. However, in the implementation used in [5], property-labeled tables are sorted only by subject (i.e., by their first column). It is suggested that a second copy of each table can be created, sorted on its value-column (i.e., its object-column) [5]. However, this suggestion is not implementated. Hence, the subject-object joins involved in a path expression query are not merge-joins, thus they are expensive.

An approach for tackling the path expression query problem is to store selected path expressions in distinct relational tables [34, 38] (see Section 2.2.1). Still, this approach is based on the assumption that certain path expressions have been pre-selected. Hence, it does not solve the problem in a general fashion. Similarly, the solution that Abadi et al. [5] suggest for this problem is to materialize the results of following selected path expressions as if they were additional regular properties, with their own two-column property tables. Thus, expensive joins for those path expressions that have been materialized are avoided. However, this solution suffers from the same generality drawbacks as those in [34, 38]. Materializing all possible path expressions is not a generally viable approach. In a path of length n, there are $\frac{(n-1)(n-2)}{2} = O\left(n^2\right)$ possible additional properties that need to be calculated.

Seen from a graph-theoretical point of view, the problem of computing all possible path expressions is an instance of the problem of computing a transitive closure, a problem which has attracted several focused studies but has defied efforts for a scalable algorithm applicable on large-scale data management [43, 31, 21]. Still, the Hexastore treats this problem effectively, without requiring precalculating and materializing selected path expressions. Thanks to its inclusion of both pso and pos indices, the first of the $n-1$ joins in a path of length n is a linear merge-join, and the rest $n-2$ ones are sort-merge joins, i.e. require one sorting operation each. Thus, the amount of sorting operations during the processing of a path-expression query is significantly reduced.

5. EXPERIMENTAL EVALUATION

In our experimental study, we compare the performance of our Hexastore approach to our representation of the column-oriented vertical-partitioning (COVP) method of [5].

We represent the COVP method through our pso indexing. This indexing provides an enhancement compared to the purely vertical-partitioning approach of [5]; namely, the pso indexing groups together multiple objects $\{\mathsf{o}_1, \mathsf{o}_2, \ldots, \mathsf{o}_n\}$ related to the same subject s by a unique property p; on the other hand, in the vertical partitioning scheme of [5], a separate $\langle \mathsf{s}, \mathsf{o}_i \rangle$ entry is made for each such object o_i in the two-column property table for property p. Moreover, we heed the suggestion in [5] that a second copy of each two-column property table can be created, sorted on the object column. In fact, this suggestion was not followed in [5]; instead, only *unclusterd* B^+ tree indices were built on the object columns with the vertically-partitioned architecture implemented in Postgres. However, such tree indices were not built when the same vertically-partitioned architecture was implemented in a column-oriented DBMS, which in fact provides the top performance in [5]. Besides, the object column is not sorted in any of the approaches examined in [5]. Still, the suggestion of having a second copy of each two-column property table, sorted on object, is tantamount to having both a pso and a pos index in our scheme. Thus, for the sake of completeness, we also conduct experiments

on such a two-index property-oriented store. In order to distinguish between the two, we call the single-index (i.e., pso) property-oriented store COVP1, and the two-index (i.e., pso and pos) store COVP2. The latter illustrates both the benefits of using a second index in comparison to the single-index COVP1, as well as its limitations in comparison to the six-index Hexastore.

Besides, as we discussed in Section 2.2.3, the experimental study in [5] was based on the assumption that only a pre-selected set of 28 out of the 221 extant properties in the studied library catalog data set [2] were interesting for the study. Four out of seven queries in the study of [5] do not actually bind on property value. The defect of a property-oriented architecture should appear with such queries. Still, in [5], these four queries were executed only on the pre-selected limited set of 28 properties. In our experimental study, we show the results both with and without this 28-property assumption. In order to distinguish between them, we label the methods which are limited to the pre-selected 28 properties with the suffix 28 in their names. Hence, for example, COVP2 28 is the two-index store limited to processing 28 properties only, while COVP2 is the regular two-index store which processes all properties available in the given data set.

Our benchmarking system is a 2.8GHz, 2 x Dual Core AMD Opteron machine with 16GB of main memory running Linux. We implemented a prototype of our indexing scheme in Python 2.5, storing the indices and processing queries in the main memory.

5.1 Description of Data

We have employed two publicly available data sets for our performance evaluation. The former is a real dataset, while the latter data set is synthetic.

5.1.1 Barton Data Set

Our first data set, also used in the experimental study in [5], is taken from the publicly available MIT Barton Libraries data set [2] (Barton), provided by the Simile project at MIT [3], a project that seeks to enhance inter-operability among digital assets, schemata, vocabularies, ontologies, metadata, and services. The data contains records acquired from a dump of the MIT Libraries Barton catalog. These records have been converted from an old library format standard called MARC (Machine Readable Catalog) to the RDF format. Due to the multiplicity of the sources the data was derived from, as well as the diverse nature of the catalogued data itself, the structure of the data is quite irregular. As in the study of [5], we have further converted the Barton data from its native RDF/XML syntax to triples and then eliminated duplicate triples. After data cleaning, a total of 61,233,949 triples were left in our data set, having a total of 285 unique properties. The vast majority of properties appear infrequently. Overall, the Barton data set provides a good example of the semi-structured nature of RDF data.

5.1.2 LUBM Data Set

Our second data set, the Lehigh University benchmark data set (LUBM), is a synthetic benchmark data set used in the comprehensive study of [24]. It models information encountered in an academic setting, as the RDF data in Figure 1. The synthetic data generation process is described in [23]. We created a data set featuring ten universities with 18 different predicates resulting in a total of 6,865,225 triples.

5.2 Description of Queries

Our experiments feature seven queries on the Barton data set and five queries on the LUBM data set. We describe those queries at a high level, and we offer details on their implementation with COVP1, COVP2, and Hexastore.

5.2.1 Barton Queries

In our experimentation with the Barton data set we aim to evaluate the benefits gained by the multiple-indexing schemes in comparison to the single-index column-oriented scheme of [5], represented by COVP1. The evaluation is interesting both with regard to COVP2, which adds a second property-based index to COVP1, and with regard to the full Hexastore. In order to conduct a fair comparison, we have employed the same benchmark query set as in [5]. Accordingly, we have followed the descriptions in [6]. These queries are based on a typical browsing session with the Longwell browser for RDF data exploration [1], hence they are representative of real-world queries [5]. We describe the intent of these queries (see also [5, 6]), as well as their implementation details below.

Query 1 (BQ1). This query has to calculate the counts of each different *type* of data in the RDF store. It involves all triples of the property *Type* and a count of each object value. Both Hexastore and COVP2 only need to report the counts of subjects on the pos index of property *Type* with respect to object. On the other hand, as COVP1 does not feature a pos index, it necessitates a self-join aggregation on object value with its pso index.

Query 2 (BQ2). This query has to display a list of properties defined for resources of *Type: Text*, along with the frequency of each property for such resources. Processing this query with COVP1 requires a selection of all subjects with object value *Text* from the pso index of *Type*; these subjects are stored in a temporary table t, which is then joined with each property's subject vector. During these joins, a count aggregation on objects is performed, in order to compute each property's frequency for subjects in t. The operation is similar with COVP2, except that the *Type: Text* selection is straightforward, thanks to the availability of the pos index. Hexastore maintains this advantage of COVP2 over COVP1, but adds to it an advantage in the second processing step too. Namely, the Hexastore only needs to merge the sorted property vectors of the subjects in t in spo indexing and aggregate their frequencies.

Query 3 (BQ3). As **BQ2**, this query has to display a list of properties defined for resources of *Type: Text*. Still, in this case, the counts of those object values that appear more than once with a certain property should also be reported with the respective property value. Query processing starts out as for **BQ2**. However, now it has to perform a count aggregation with respect to predicate-object pairs, and then select those results that have an object count larger than one. COVP1 operates as for **BQ2**, with the addition that the instances of each object per property are counted separately. COVP2 utilizes its pos index in the final processing step, in order to retrieve the count of each object related to subjects in t for each property. Hexastore maintains the advantage derived from its use of the spo index; however, now it cannot perform an immediate aggregation of object counts per property on the spo index in the final step; instead, it needs to utilize the pos index in the same way as COVP2 does for this query.

Query 4 (BQ4). As **BQ3**, this query reports the proper-

ties, and the frequencies of 'popular' (i.e., appearing more than once) object values for each property, defined for particular subjects. In this case, the desired subjects need not only be of *Type: Text*, but also of *Language: French*. Query processing works as for **BQ3**, but differs in the pre-selection step. Thus, COVP1 jointly selects subjects from the pso indices of *Type* and *Language*, so that they satisfy both constraints. On the other hand, COVP2 and Hexastore need to retrieve and merge-join the subject lists for *Type: Text* and *Language: French* using their pos indices.

Query 5 (BQ5). This query has to perform a type of *inference*. For all subjects that originate in the US Library of Congress, and have the *Records* property defined, it has to report their inferred type, if it is not *Type: Text*. This inferred type is calculated as the *Type* of the object that each of these subjects *Records*. To process this query, COVP1 first selects on *Origin: DLC*. Then it joins the result subject list s with the subject vector of the *Records* property in the pso index, and retrieves an (unsorted) list t of recorded objects. In its turn, t is sort-merge joined with the (sorted) subject vector of the *Type* property, reporting all non-*text* results. In contrast to COVP1, both COVP2 and Hexastore process this query in a more efficient fashion. With them, the subject list s is derived straightforwardly, thanks to the availability of the pos index. Moreover, these schemes do not need to perform an expensive join between the (unsorted) object list t and the (large) subject vector of the *Type* property. Instead, they first *merge-join* the (sorted) object vector of the *Records* property in the pos index with the (sorted) subject vector of *Type*, selecting subjects of non-text *inferred* type, and thus derive a (small) table T of such subjects. Then the (small) list s is sort-merge joined with T, and the desired results of non-text inferred type are reported.

Query 6 (BQ6). This query combines the inference step of **BQ5** and the property frequency calculation of **BQ2**. Its aim is to extract information in aggregate form (as in **BQ2**) about all resources that are either known to be, or can be inferred to be (as in **BQ5**), of *Type: Text*. To process this query, all methods merge the result sets of **BQ2** and **BQ5**, aggregating property frequencies as in **BQ2**. The same advantages of Hexastore and COVP2 as for each individual query hold.

Query 7 (BQ7). This is a simple triple selection query. It aims to retrieve the *Encoding* and *Type* information about all resources whose *Point* value is 'end'. The intent of the user is to learn what this *Point* value means. The result of the query reveals to the user that all such resources are of type *Date*, hence their 'end' value implies that these dates are *end dates* (i.e., as opposed to *start dates*). To process this query, COVP1 performs a selection on *Point: End* first, and merge-joins the derived result with the subject vectors of properties *Encoding* and *Type*. COVP2 and Hexastore retrieve the first result set straightforwardly with the pos index, and proceed in the same fashion as COVP1.

The processing of queries **BQ2**, **BQ3**, **BQ4**, and **BQ6** in [5] was constrained on only 28 pre-selected properties. We have run our evaluation for these four queries both *with* and *without* this constraint. Thus, we investigate the effect of limiting the properties on which non-property-bound queries are processed with the property-oriented scheme of [5]. Furthermore, this detailed evaluation offers us the chance to study the advantage gained with the Hexastore on unconstrained queries that do not bind property values.

5.2.2 LUBM Queries

In our evaluation study with the synthetic LUBM data set, our aim is to compare the multiple-indexing schemes to the single-index column-oriented scheme of [5], represented by COVP1, with general-purpose queries. To that end, we have designed a set of five meaningful benchmark queries. These queries are not oriented towards a particular storage scheme, and do not discriminate in favor of, or against, any particular type of RDF resource.

These queries are based on typical information one may ask for in the context of the domain that the LUBM data represent, i.e. the domain of academic personnel and student enrollment information in higher education. We have designed our queries with this real-world application in mind. Again, we describe the intent of these queries as well as their implementation details below.

Query 1 (LQ1). This query aims to find all people that are somehow related to course[1] `Course10` (i.e. all lecturers, all students). To process **LQ1**, both COVP1 and COVP2 have to perform multiple selections on object `Course10`. COVP2 is faster, thanks to its pos indexing. However, Hexastore retrieves the results straightforwardly using its osp indexing.

Query 2 (LQ2). This query has to find all people that are somehow related to `University0`. The processing of this query works as for **LQ1**.

Query 3 (LQ3). This query aims to find all immediate information about `AssociateProfessor10` (i.e. what degree she received from which university, or whom she advises). Although this query looks like a simple statement retrieval, as **LQ1** and **LQ2**, it needs a bit more elaboration, as `AssociateProfessor10` may appear both as subject and as object. Thus, COVP1 and COVP2 need to perform selection on both subject and object in each property table, and then union them. COVP2 has again the advantage of using the pos index for object-bound selection. On the other hand, Hexastore only has to perform two lookups, one in index spo and one in index ops .

Query 4 (LQ4). This query has to find people who are related to the courses `AssociateProfessor10` is teaching, and group them by course. To process this query, COVP1 first retrieves a list t of relevant courses using the pso index for the *TeacherOf* property. This list is then joined with all object lists in the pso index to retrieve matching subjects. COVP2 uses the pos index in the second step, gaining an advantage. In contrast, Hexastore processes the second step with a lookup in the osp index for each course object in t.

Query 5 (LQ5). This purpose of this query is to find people who received any degree (undergraduate, masters, or doctoral) from at least one of the universities to which `AssociateProfessor10` is related, and group them by university. For this query, COVP1 first selects a list of objects t to which `AssociateProfessor10` is related, scanning all pso property indices. Then it joins t with the subject vector of the *Type* property, selecting for *Type: University*, thus it refines t to a university list t'. The list t' is then joined with the subject vectors of all *degreeFrom* properties and the union of the results in returned. With COVP2 , the refinement of t to t' is straightforward, thanks to a pre-selection of *Type: University* items using the pos index. Furthermore, COVP2 also simplifies the final step, as it performs a lookup

[1]For the sake of conciseness and readability, we do not denote the full URIs of the resources involved in our queries.

in the pos index of each *degreeFrom* property for each university in t'. Hexastore confers an additional advantage in the first step; it straightforwardly retrieves t as the object vector for subject `AssociateProfessor10` in sop indexing.

5.3 Results

In this section, we report the results of our experimental study with the queries described in Section 5.2 for both the Barton and the LUBM data sets. We have experimented with progressively larger prefixes of the working data sets, measuring response times in each case. In all cases, we use logarithmic axes for the response time measurements. We thus show that the Hexastore typically achieves one to three orders of magnitude (a factor of 1000), and can reach five orders of magnitude, of performance improvement compared to COVP1, while it also achieves one to two orders of magnitude better performance than COVP2.

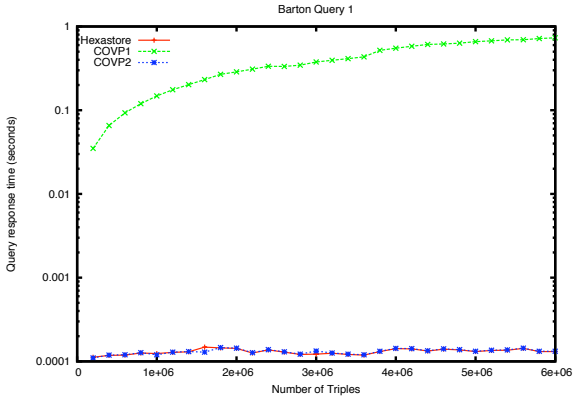

Figure 3: Barton data set, Query 1

5.3.1 *Barton Queries*

Figure 3 shows the results with Barton Query 1. Observably, the performance of both COVP2 and Hexastore achieves a significant gap from that of COVP1, due to the utilization of the pos index by these methods. Moreover, while the response time grows with the number of triples in the store for COVP1, it stays practically constant for COVP2 and Hexastore, as any new relevant triples are all added on the pos index and easily retrieved.

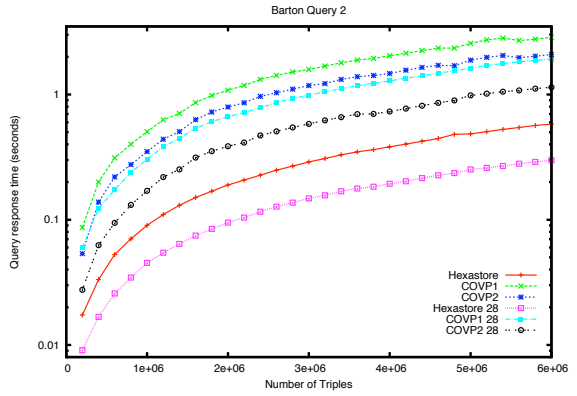

Figure 4: Barton data set, Query 2

The results for Barton Query 2 are shown in Figure 4, for both the 28-property version and the general version. In both variants, the Hexastore gains a distinct performance advantage (one order of magnitude) in comparison to both other schemes. This advantage is due to the ability of the Hexastore to directly merge property vectors, using the spo indexing. Moreover, the relative advantage of COVP2 to COVP1 is visible; the difference is due to the use of the pos index for the original type selection.

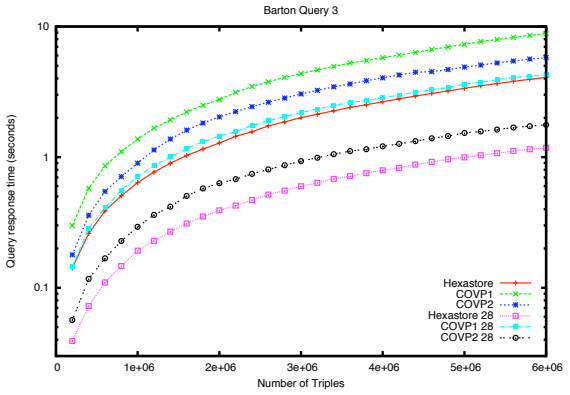

Figure 5: Barton data set, Query 3

Figure 5 depicts the results for Barton Query 3. The advantage of the Hexastore is narrower in this case, due to the more complicated final aggregation step. The increase in the number of examined properties affects all methods proportionally, as all methods need to use a property-based index for all relevant properties in the final step.

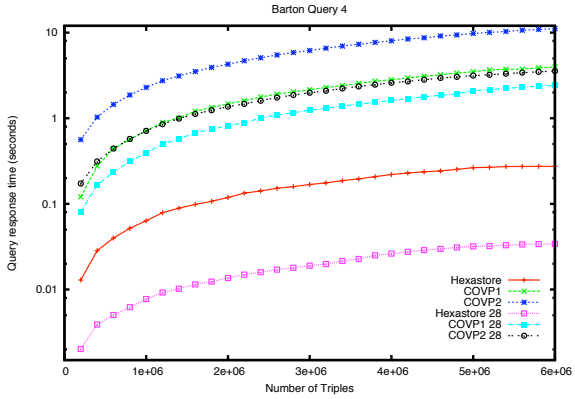

Figure 6: Barton data set, Query 4

Figure 6 shows the results with Barton Query 4. The advantage of Hexastore is now more distinct. Again, all methods are affected proportionally by the increase in the number of examined properties. The additional selection by language reduces the number of results with this query, hence the processing time in the final aggregation step is reduced.

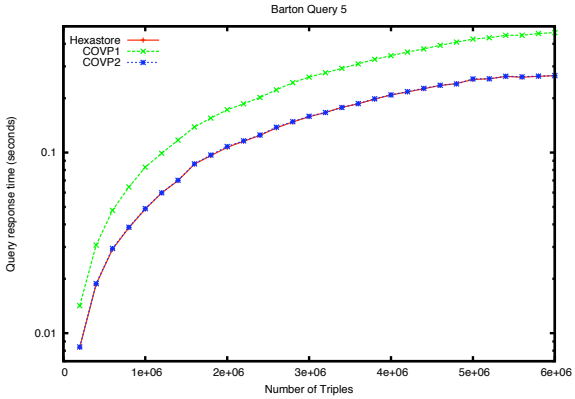

Figure 7: Barton data set, Query 5

The results for Barton Query 5 are presented in Figure 7. The advantage of both COVP2 and Hexastore is chiefly due to their avoidance of an expensive join. This advantage shows the efficient treatment of inference (see Section 4.3).

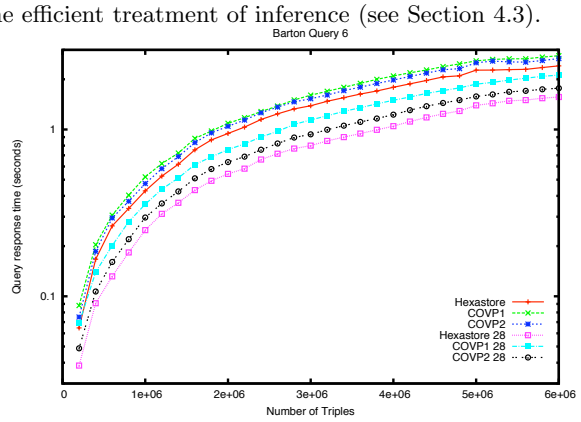

Figure 8: Barton data set, Query 6

Figure 8 shows the results with Barton Query 6, which combines BQ2 and BQ5. Although the Hexastore maintains its advantages, these are obscured by the final aggregation step.

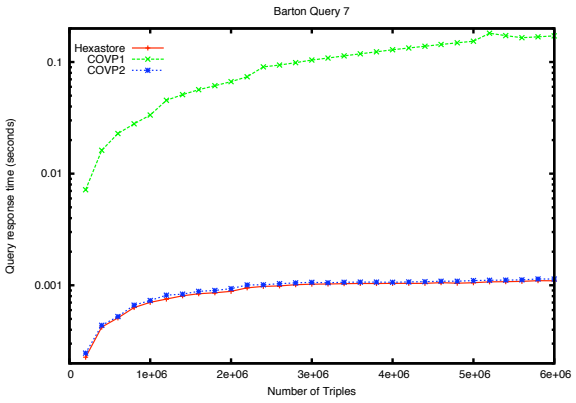

Figure 9: Barton data set, Query 7

Finally, the results for Barton Query 7 are shown in Figure 9. The advantage of both COVP2 and Hexastore, thanks to the fast retrieval with the pos index, is again clear.

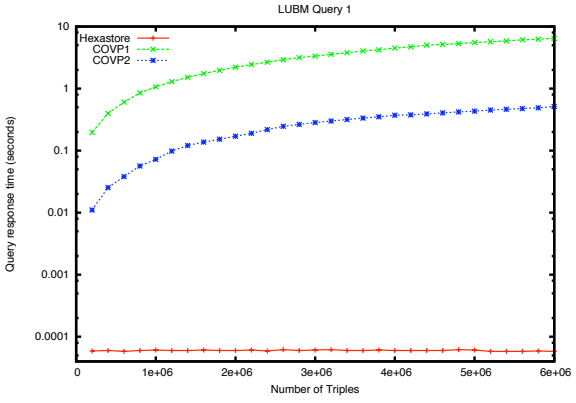

Figure 10: LUBM data set, Query 1

5.3.2 *LUBM Queries*

Figure 10 presents the time measurement results for LUBM Query 1. The advantage of the Hexastore, thanks to the direct exploitation of its osp indexing, is clear. COVP2 also does better than COVP1, but still differs from the Hexastore by two orders of magnitude.

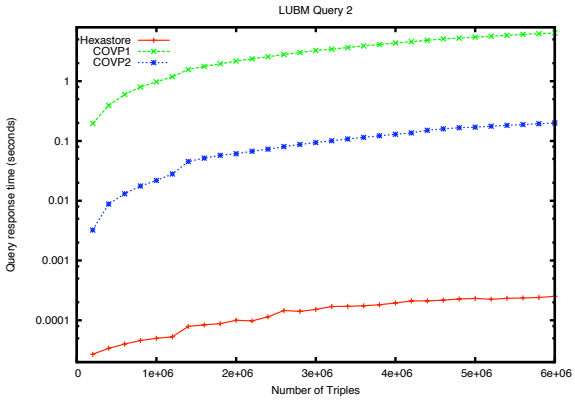

Figure 11: LUBM data set, Query 2

The results with LUBM Query 2 are shown in Figure 11. In this case the Hexastore exhibits a clear performance advantage, while a growth trend is more visible. The growth is due to the fact the more triples associated with the given university object are entered in the data set as it grows.

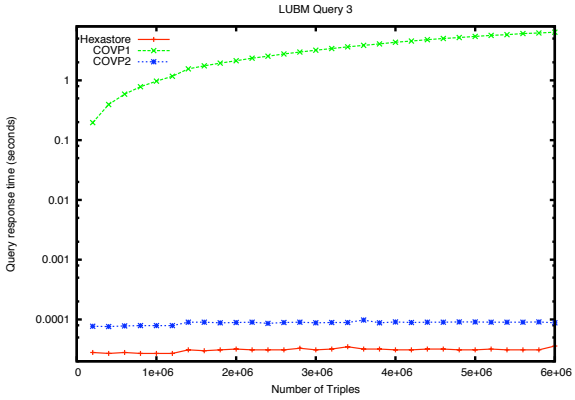

Figure 12: LUBM data set, Query 3

Figure 12 depicts the results with LUBM Query 3. The performance advantage of the Hexastore reaches three orders of magnitude again. COVP2 performs better with this query, thanks to its exploitation of the pos index. However, it still does not match the efficiency of the Hexastore.

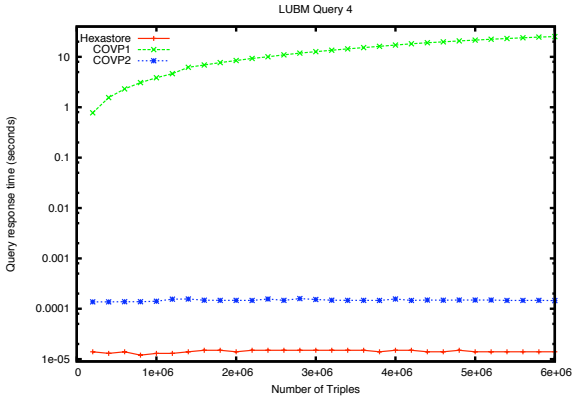

Figure 13: LUBM data set, Query 4

Figure 13 shows the performance picture for LUBM Query 4. In this case, the performance of the Hexastore achieves a difference of four to five orders of magnitude from that of

COVP1. This difference is due to the convenient use of the osp index by the Hexastore, in place of complex joins for all properties by COVP1. COVP2 performs well, but still does not match the efficiency of the Hexastore.

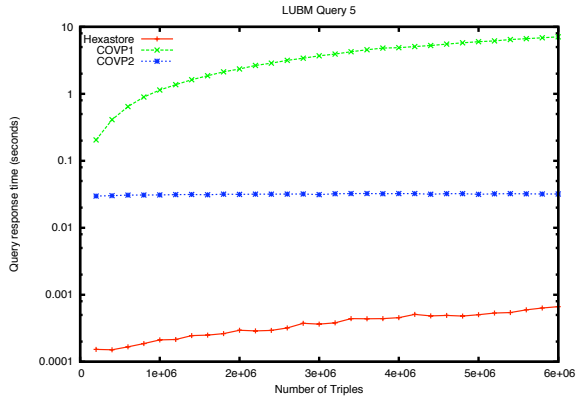

Figure 14: LUBM data set, Query 5

Last, Figure 14 depicts the results for LUBM Query 5. Hexastore gains an advantage of two to three orders of magnitude in comparison to the other schemes. This result highlights the benefits gained from the use of the sop index, as opposed to scanning through all property tables. COVP2 scales better than COVP1 , thanks to the double advantages gained by the use of the pos index, yet it does not match the Hexastore.

5.3.3 *Memory Usage*

Figure 15 presents the memory usage measurements with the two employed data sets. In practice, Hexastore requires a four-fold increase in memory in comparison to COVP1, which is an affordable cost for the derived advantages.

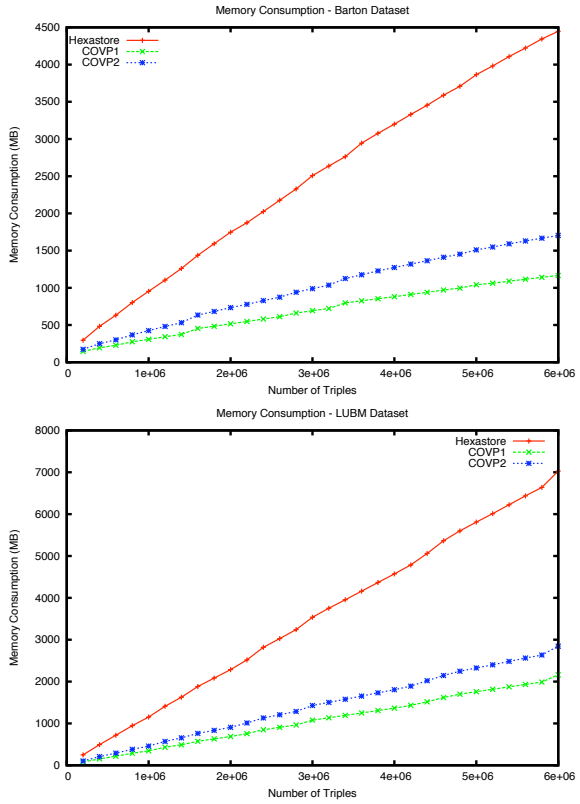

Figure 15: Memory Usage

6. DISCUSSION AND LIMITATIONS

Our results on lower query response time of the Hexastore vis-à-vis our representation of the property-based column-oriented approach as COVP1 were expected. Apart from the question of handling updates, a question of the parsimoniousness of storage emerges. The Hexastore is concise in its individual representations, but the replication of these representations in a five-fold manner may exceed the available storage capacity. As we have argued, it is worthwhile to invest in predictable additional storage in order to gain in efficiency and scalability. Besides, some indices may not contribute to query efficiency based on a given workload. For example, the ops index has been seldom used in our experiments. A subject for future research concerns the selection of the most suitable indices for a given RDF data set based on the query workload at hand. Database *cracking* has been suggested as a method to address index maintenance as part of query processing using continuous physical reorganization [32, 30, 29]. An interesting question is to examine whether such an approach can be adapted to Hexastore maintenance.

7. CONCLUSIONS

In this paper we have proposed the Hexastore, an architecture for Semantic Web data management. This architecture turns the triple nature of RDF from a liability, as it has conventionally been seen, to an asset. We have shown that, thanks to the deterministic, triple nature of RDF data, they can be stored in a sextuple-indexing scheme that features a worst-case five-fold storage increase in comparison to a conventional triples table. Moreover, our scheme enhances on previously proposed architectures, such as limited multiple-indexing schemes, property-table solutions, and the most recently proposed vertical-partitioning scheme implemented on a column-oriented system. Still, we have taken the rationale of those schemes to its full logical conclusion. As our experimental study has shown, the Hexastore allows for quick and scalable general-purpose query processing, and confers advantages of up to five orders of magnitude in comparison to previous approaches. These benefits derive from the convenience that a Hexastore offers. Remarkably, all pairwise joins in a Hexastore can be rendered as merge joins. In the future, we intend to examine how cracking techniques can be applied on a Hexastore. Moreover, we intend to implement a fully operational disk-based Hexastore and compare it to other RDF data storage schemes as well as to relational DBMSs on data management tasks that can be defined on both an RDF and a relational environment.

Acknowledgments

We thank the Swiss NSF for partially supporting this work, the anonymous referees for their insightful suggestions, and Daniel Abadi for our fruitful discussions.

8. REFERENCES

[1] Longwell browser. http://simile.mit.edu/longwell.

[2] MIT Libraries Barton Catalog Data. http://simile.mit.edu/rdf-test-data/barton/.

[3] The SIMILE Project. http://simile.mit.edu/.

[4] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[5] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data

Management using vertical partitioning. In *VLDB*, 2007.

[6] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Using the Barton Libraries dataset as an RDF benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT, 2007.

[7] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.

[8] S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. On storing voluminous RDF descriptions: The case of web portal catalogs. In *WebDB*, 2001.

[9] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *SemWeb*, 2001.

[10] R. Angles and C. Gutiérrez. Querying RDF data from a graph database perspective. In *ESWC*, 2005.

[11] D. Beckett. The design and implementation of the Redland RDF application framework. In *WWW*, 2001.

[12] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, 2006.

[13] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[14] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Jounral*, 8(2):101–119, 1999.

[15] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[16] V. Bönström, A. Hinze, and H. Schweppe. Storing RDF as a graph. In *LA-WEB*, 2003.

[17] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *ISWC*, 2002.

[18] K. S. Candan, H. Liu, and R. Suvarna. Resource Description Framework: Metadata and its applications. *SIGKDD Explorations Newsletter*, 3(1):6–19, 2001.

[19] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the Semantic Web recommendations. In *WWW (Alternate Track Papers & Posters)*, 2004.

[20] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, 2005.

[21] S. Dar and R. Ramakrishnan. A performance study of transitive closure algorithms. In *SIGMOD*, 1994.

[22] R. V. Guha. rdfDB : An RDF database. http://www.guha.com/rdfdb/.

[23] Y. Guo, J. Heflin, and Z. Pan. Benchmarking DAML+OIL repositories. In *ISWC*, 2003.

[24] Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large OWL datasets. In *ISWC*, 2004.

[25] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *PSSS*, 2003.

[26] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *SSWS*, 2005.

[27] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *LA-WEB*, 2005.

[28] J. Hayes and C. Gutiérrez. Bipartite graphs as intermediate model for RDF. In *ISWC*, 2004.

[29] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[30] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[31] Y. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM TODS*, 18(3):512–576, 1993.

[32] M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, 2005.

[33] C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL - a virtual triple approach for similarity-based Semantic Web tasks. In *ISWC*, 2007.

[34] Y. Kim, B. Kim, J. Lee, and H. Lim. The path index for query processing on RDF and RDF Schema. In *ICACT*, 2005.

[35] E. Liarou, S. Idreos, and M. Koubarakis. Continuous RDF query processing over DHTs. In *ISWC*, 2007.

[36] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *CIKM*, 2004.

[37] F. Manola and E. Miller, editors. *RDF Primer*. W3C Recommendation. WWW Consortium, 2004.

[38] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *ADC*, 2005.

[39] Z. Pan and J. Heflin. DLDB: Extending relational databases to support Semantic Web queries. In *PSSS*, 2003.

[40] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS: Fast filtering of graph-based metadata. In *WWW*, 2005.

[41] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[42] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: a column-oriented DBMS. In *VLDB*, 2005.

[43] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. In *SIGMOD*, 1990.

[44] R. Volz, D. Oberle, S. Staab, and B. Motik. KAON SERVER - A Semantic Web Management System. In *WWW (Alternate Paper Tracks)*, 2003.

[45] K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.

[46] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, 2003.

[47] D. Wood, P. Gearon, and T. Adams. Kowari: A platform for Semantic Web storage and analysis. In *XTech*, 2005.