On-disk storage techniques for Semantic Web data - Are B-Trees always the optimal solution?

Cathrin Weiss, Abraham Bernstein

University of Zurich
Department of Informatics
CH-8050 Zurich, Switzerland
{weiss,bernstein}@ifi.uzh.ch

Abstract. Since its introduction in 1971, the B-tree has become the dominant index structure in database systems. Conventional wisdom dictated that the use of a B-tree index or one of its descendants would typically lead to good results. The advent of XML-data, column stores, and the recent resurgence of typed-graph (or triple) stores motivated by the Semantic Web has changed the nature of the data typically stored.
In this paper we show that in the case of triple-stores the usage of B-trees is actually highly detrimental to query performance. Specifically, we compare on-disk query performance of our triple-based Hexastore when using two different B-tree implementations, and our simple and novel vector storage that leverages offsets.
Our experimental evaluation with a large benchmark data set confirms that the vector storage outperforms the other approaches by at least a factor of four in load-time, by approximately a factor of three (and up to a factor of eight for some queries) in query-time, as well as by a factor of two in required storage. The only drawback of the vector-based approach is its time-consuming need for reorganization of parts of the data during inserts of new triples: a seldom occurrence in many Semantic Web environments.
As such this paper tries to reopen the discussion about the trade-offs when using different types of indices in the light of non-relational data and contribute to the endeavor of building scalable and fast typed-graph databases.

1 Introduction

The increasing interest in the Semantic Web has motivated a lot of recent research in various areas. That is because the dynamic graph-structured character of Semantic Web data is challenging many traditional approaches, for example those of data indexing and querying. So it does not come as a surprise that there has been done a lot of work to improve state of the art Semantic Web engines. Recent publications show how to index Semantic Web data efficiently [1, 8, 17], how to improve query optimization processes [9, 14], and how to represent the data. Most of them aim to avoid mapping data to the relational scheme.

However, beyond optimizing in-memory indices, state-of-the-art systems still make use of traditional data structures, such as B-trees, when it comes to on-disk storage. It is surprising that so far no effort has been made on analyzing whether those storage structures are a good match for the new indices. Will traditional approaches still work well with the newly developed indexing techniques?

In this paper we discuss this issue and propose a novel, but simple lookup-based, on-disk vector storage for the Hexastore indices. The vector storage employs key offsets rather than a tree structure to navigate large amounts of disk-based data. The use of fixed-length indices makes lookups highly efficient ($O(1)$, with at most three page reads) and loads efficient. While updates are a bit more costly, the fast load time makes it, oftentimes, simpler to "just" reload the whole data. We benchmark our vector storage for Hexastore with two Hexastore-customized B-tree based approaches. One has a B-tree index for each of the Hexastore indices. Another has one B-tree index which combines all indices. We will see that for typical queries the vector storage outperforms a B-tree based structure by a factor of eight.

In summary our contributions in this paper are the following:

– We propose a simple but novel approach for on-disk storage of triples that relies on an offset-based vector storage. The approach allows for a highly compact representation of the data within the index while preserving a fast retrieval forgoing some insert/update efficiency – a tradeoff that doesn't seem too disadvantageous given the nature of the data.
– We experimentally compare the load-time and space requirement performance of the vector index with two different implementations of the a B-tree style index and two different versions of a traditional table-based triple store. We show that the vector storage based Hexastore has a smaller storage footprint than a B-tree based approach and that it is much faster in loading the data. Furthermore, we show that the vector storage based Hexastore has about three to eight times the speed of the B-tree based approaches in answering queries confirming the theoretical considerations.

The remainder of the paper is structured as follows. First we discuss the work related to investigating non-tree data structures. Section 3 summarizes the structure of Hexastore, introduces the novel vector storage structure, and explains how the two B-tree-based back-ends for Hexastore are constructed. Section 4 discusses the advantages and disadvantages of each of the indices and is complemented by the experimental evaluation. We close with a discussion of limitations and our conclusions.

2 Related Work

We found three areas of related work: other work on typed-graph (or RDF) stores, projects focusing on the native storage of XML data, and other papers investigating the limitations of usefulness of tree-based storage.

Efficient indexing of Semantic Web data has become an active research topic [1, 8, 13, 17]. Most approaches propose methods how to rearrange data in-memory or in given database systems respectively such that query processing can be performed more efficiently compared to straightforward approaches like triple tables. Abadi et al [1] store their vertical partitioning approach in a column-oriented database [2, 15]. Specifically they store each predicate in a two-column $< subject, object >$ table in a column store, indexed with an unclustered B-tree. Neumann et al. [8] as well as Guha [12] use native B-tree storage approaches. The goal with our proposed vector storage is to avoid storage in existing DBS and also the usage of B-tree structures and to show that applying these approaches is detrimental to query performance.

One of the oftentimes used serializations of an RDF graph is in XML. Projects in the XML domain have investigated a plethora of approaches to efficient storage and querying of this type of data. We can distinguish between non-native storage strategies, which map the data onto the relational model, and native strategies, that try to store the data more according to its nature. Native XML databases [3, 5, 7] typically store their data either as the XML document itself, or they store the tree structure, i.e., the nodes and child node references. For indexing, some index-related information may be stored as well, such as partial documents, sub-tree information, and others. All of these approaches have in common that they store their data (and usually their indices) in tree like structures, as the underlying data is also in that format. The one exception is the native on-disk XML-storage format proposed by Zhang et al. [18]. It provides an optimized, non-tree disk-layout for XML-trees optimized to answer XPath queries. Akin to this last project, we also propose to shed the limitations of the underlying data format. Indeed, our on-disk structure consists only of the indices themselves, which helps to answer queries about the underlying data and, hence, enables their reconstruction.

It was most difficult to find work investigating the boundaries of tree-based indices. Idreos et al. [4] address the slow build-time of an index in general by proposing not to build an index at load time but to initially load the data in its raw format and reorganize it to answer each query. They show that under certain conditions the data organization converges to the ideal one. In the most radical attack on the general applicability of trees, Weber et al. [16] discuss similarity searches in high-dimensional spaces. They find that the performance of tree-based indices radically degrades below the performance of a simple sequential scan. They propose a novel vector-approximation scheme called VA-file that overcomes this “dimensionality curse”. Our work can be seen in the spirit of these studies in that we also try to explore the limitations of the predominant tree structures and propose alternatives that excel under certain conditions.

3 The Storage Structure

In this section we describe our vector storage for Hexastore indices and data. Hexastore, proposed in [17] as an in-memory prototype, is an efficient six-way indexing structure for Semantic Web data. In order to explain the functionality of the vector storage we first briefly review the functionality of Hexastore itself and its requirements towards an index. We then introduce the vector storage and discuss its technical and computational characteristics. This is followed by a brief explanation on how to build the B-tree based Hexastore back-ends.

3.1 Hexastore: A Sextuple index based graph store

A Semantic Web triple $< s, p, o >$ consists of a *subject* s, which is a node in the graph, a *predicate* p designating the type of edge, and an *object* o, which is either a node in the graph or a literal. In RDF, the nodes are identified by a URI. To limit the amount of storage needed for the URIs, Hexastore uses the typical dictionary encoding of the URIs and the literals, i.e. every URI and literal is assigned a unique numerical *id*. Furthermore, Hexastore recognizes that queries are constructed by a collection of graph patterns [14] which may (i) bind any of the three elements of the triples to a value, (ii) may use variables for any of the triple elements effectively resulting in a join with other graph pattern, and (iii) define any element with a wild card to be returned. Consequently, any join order between triple patterns in the query is possible. Hexastore, therefore, indexes the data to allow for retrieving values for each order of a triple pattern respectively joining over every element (s, p, or o) of a triple pattern resulting in six indices designated in the order in which the triple elements are indexed (e.g., SPO, OSP, etc.). This structure allows to retrieve all connected triple residuals with one index lookup.

Figure 1 illustrates the structure of the six indices. It shows that each index essentially consists of three different elements: a first-level `Type1` index, a second-level `Type2` index, and a third-level `Type3` ordered set, where the `TypeN`'s are one of the three triple elements $\{subject, predicate, object\}$. Given a key a_i the first-level `Type1` index returns a second-level `Type2` index. Given a key b_j the `Type2` index returns a `Type3` ordered set, which lists all the matches to the query $< a_i, b_j, ? >$. As an example consider trying to match a query that tries to find all papers authored by "Bayer". This query could result in the triple structure $< Bayer, authored, ? >$ and could be executed by consulting the *spo* index (i.e., `Type1` $= s$, `Type2` $= p$,and `Type3` $= o$). Hence, first the `s` index would be asked to return the `p` index matching the key "Bayer", then the returned index would be asked to return the ordered set for the key "authored", which would be the result of the query. Note that this structure has the advantage that every lookup is of amortized cost of order $O(1)$.

Our implementation of Hexastore presented in [17] used an in-memory prototype for all experiments. Storing the first and second level indices on disk so that all Hexastore performance advantages can be preserved is not straightforward. Clearly the proposed indexing technique does not adhere to the traditional relational model. Still, taking inspiration from RDF-3X, we could use B-trees as the well established indexing technique – an approach that we use to compare our results to (see Section 3.3). But as we argued in the introduction we believe that since our data adheres to different underlying assumptions, we would loose

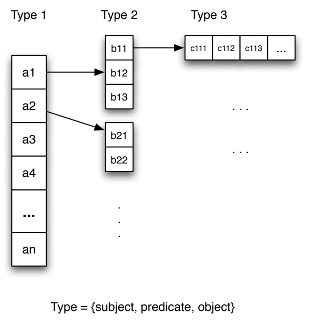


Fig. 1. Sketch of a Hexastore index

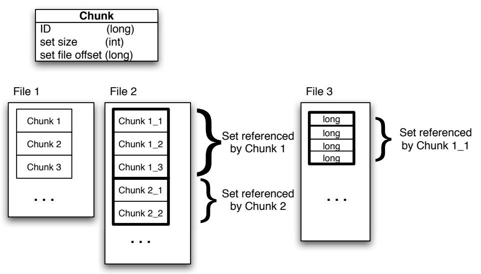


Fig. 2. Vector storage file layout

some of the advantages of the Hexastore index. Consequently, we need an on-disk index structure/storage layout that adheres to the following requirements:

1. Given an item a of `Type1` and a desired target `Type2`, there should be an operation on the first-level index that efficiently retrieves the associated second-level index of `Type2`, denoted as $\mathcal{I}(\texttt{Type2})$. Hence we need an efficient implementation of the operation:

$$getIdx(a) : \texttt{Type1} \mapsto \xrightarrow{PointerTo} \mathcal{I}(\texttt{Type2})$$

2. Given an item b of `Type2` and a desired target `Type3`, there should be an operation operating on the second-level index that efficiently retrieves the associated third-level ordered set `Type3`, denoted as $\mathcal{S}$. Hence we need an efficient implementation of the operation:

$$getSet(b) : \texttt{Type2} \mapsto \xrightarrow{PointerTo} \mathcal{S}(\texttt{Type3})$$

3. The operations $getIdx(a)$ and $getSet(a)$ should require as few read operations as possible.
4. The third-level sorted set should be accessible in as few read operations as possible.

Obviously, this linked structure can be implemented in various ways. Given that the structure is reminiscent of linked vectors one approach would be to store each vector-like structure in a column store. This would fulfill requirements 3 and 4 above. In practice, however, this approach does not scale, as the number of vectors is huge: while the number of first-level indices is only six, the number of second level indices has an upper bound of $2|S| + 2|P| + 2|O|$ and the number of third-level ordered sets would be, due to the fact that the third-level sets can be shared by two indices, $|S|(|P| + |O|) + |O||P|$ – a prohibitively large number. MonetDB [2], for example, allocates one file for each column. Since most vectors are small (requiring far less space than 4 KB, which is the default minimal size for each new file), this approach would lead to a tremendous waste of space. In the following, we discuss three implementations. First, we introduce our own vector storage. Then, we show two different approaches of how the two necessary structure operations ($getSecondLevelIndex(a)$ and $getThirdLevelSet(a)$) can be implemented using a traditional B-tree.

3.2 Vector Storage

The Hexastore structure favors vector-like indices connected with pointers. To mimic this behavior on-disk we needed to establish the analogues of vectors and pointers on-disk whilst limiting the number of files needed (to avoid having a large numbers of almost empty second and third level files). As an analogue for a pointer we chose a storage structure we call a *chunk* (see also Figure 2 at the top). A chunk contains the particular *id* (the *id* that results of the dictionary encoding) of a URI or literal represented as a `long integer`, the number of elements in the associated second-level index (or third-level sorted set) it points to represented as `integer`, which can be chosen as `long integer` as well, if necessary, and the offset information where to find the associated set data in the level-two, respectively level-three file, again represented as `long integer`.

Chunks allow for the efficient storage of the first and second-level indices in a single file each. Figure 2 shows the layout for each of the six Hexatore indices. If an element with *id* i is stored, the appropriate chunk in the ith position of File 1 has the value `ID` set to i and `size` set to the number of chunks it points to in the second-level index. The value `offset` contains the position of the start of the files referenced by the chunk within File 2. Some nodes that appear as *subjects* might also be *objects* in some triples, while other nodes are only referred to as either *subjects* or *objects*. In the latter case the first-level index of the SPO, SOP, OSP, and OPS has to return a NULL value, as those ids are not used as *subject* respectively *object*. If that is the case the particular entry is filled with a “zero” chunk, i.e. the `ID`, `size` and the `offset` are set to 0. While this approach “wastes” some space to “zero” chunks, it maintains a placeholder for every possible *subject* and/or *object* id allowing to compute the location of a chunk associated with a given id by simply multiplying the *id* times the disk footprint of a chunk (i.e., $(\texttt{id} - 1) \cdot \texttt{sizeof}(\texttt{chunk})$). In the *predicate*-headed first-level indices (belonging to the overall Hexastore indices PSO and POS) we avoid the necessity of “zero” chunks all together by using a different key-space for the dictionary encoding. The second-level index is also stored as a collection of chunks grouped in a single file denoted as File 2 in Figure 2. Since the lookup in the first-level index provided us with the offset as well as the size (or number) of chunks associated with its key in the second-level index we can again start to directly read the relevant chunks and know how far we need to read. Note that the second-level index does not use “zero” chunks, since the entries associated with a first-level key are typically much fewer than the number of nodes (or edges). Consequently, the offset-jump method of finding a second-level chunk associated with a key requires a search for the chunk. The size of the second-level group of chunks corresponds to the degree of a given node (or the number of differing types of predicates it is associated with) in the case of a *subject* or *object* first-level index, or the number of nodes connected with a certain type of edge in the case of a first-level *predicate*. Hence, in most cases, the typical size of a second-level group of chunks is going to be small enough to fit into main memory allowing an efficient binary search. The third-level sorted sets are again all stored in a single file denoted as File 3 in Figure 2. Reading the sorted set associated with a second-level chunk results in a simple reading of the `size` ids starting from the `offset` in the file.

The presented on-disk structure allows to store each index in 3 files resulting in a total of 15 files (in addition to the dictionary store). Note that the number of File 3s can be halved, as two Hexastore indices can share them (e.g., the SPO and PSO index can share their third-level list File 3). Returning to our requirements of Section 3.1 we can summarize:

1. $getIdx(a)$ can be implemented as a simple lookup based on an off-set calculation (i.e., $(\texttt{id} - \texttt{1}) \cdot \texttt{sizeof}(\texttt{chunk})$). It requires at most one page read.
2. $getSet(b)$ can be implemented as a search over an ordered set of chunks in File 2: In the best case it will involve one page read followed by a binary search (or a simple binary search if the page is already in memory). In the worst case it might involve multiple page reads (if the chunk group is larger than the page size) and partial binary searches.
3. Few page reads (first and second level): In the first-level index lookup only the relevant page is read. In the second-level index only the pages associated with the relevant chunk group are read.
4. Third-level page reads: only pages containing the relevant third-level ordered set are read.

3.3 B-tree Based Implementations

As a base-line comparison and following the conventional wisdom we implemented two different Hexastore implementations based on B-trees. We decided to use BerkeleyDB B-trees rather than from-the-scratch implemented ones, as they a) are efficiently implemented, and b) allow for flexible key-value definitions per default.

The first approach stores each Hexastore index in a separate B-tree. We refer to it as the *One-For-Each (OFE)* approach. The second approach stores all Hexstore indices in a single B-tree. Therefore, we refer to it as the *All-In-One (AIO)* approach.

OFE: Storing each Hexastore index in one B-tree In this approach we store each Hexastore index in a separate B-tree. Thus, instead of using the structure shown in Figure 2 we implement it as a B-tree with a compound key consisting of the lookup value for the first-level and the second-level index in the form of:

$$BtreeLookup(< a, b >): \begin{cases} <\texttt{Type1}, \texttt{Type2}> \mapsto \mathbb{S}(\texttt{Type3}) & \text{if } \; b > 0 \\ <\texttt{Type1}> \mapsto \mathbb{S}(\texttt{Type2}) & \text{otherwise.} \end{cases}$$

This provides an easy lookup for each of the operations necessary to implement a Hexastore operation. The typical lookup of the sorted set of third-level keys for a given set of first-level and second-level keys is a straightforward call of the $BtreeLookup(a, b)$ function. Note that the vector storage requires calling $b = getIdx(a)$ followed by calling $getList(b)$ to achieve the same operation. If only the second-level keys are needed then the second-level key is passed as 0. Hence, $getIdx(a) = BtreeLookup(< a, 0 >)$. The result of this implementation is that all lookups are optimized using the chosen B-tree implementation

AIO: Storing Hexastore in one B-tree In this approach we even further reduce the number of needed B-trees by adding the type of index (SPO, SOP, ...) to the compound key. Thus:

$$\begin{aligned} &BtreeLookup(< a, b, t >): \\ &< \texttt{Type1}, \texttt{Type2}, \texttt{idxType} > \mapsto \mathcal{S}(\texttt{Type3}) \ \text{ if } \ b > 0 \\ &\qquad < \texttt{Type1}, \texttt{idxType} > \mapsto \mathcal{S}(\texttt{Type2}) \ \text{ otherwise,} \end{aligned}$$

where $\texttt{idxType} \in \{\mathsf{SPO}, \mathsf{SOP}, \mathsf{PSO}, \mathsf{POS}, \mathsf{OSP}, \mathsf{OPS}\}$. Again partial lookups are achieved with setting b to 0 and efficiency is handled by the B-tree implementation.

Note, that assuming an efficient implementation of a B-tree the main difference between *OFE* and *AIO* lies in the implementation ability to reuse/share elements of the tree and the approach to dealing with compound keys. The simpler key structure of *OFE* suggests a faster index build time. But an efficient handling of compound keys in *AIO* could lead to a better reuse of already read pages and could lead to better retrieval times.

4 Experimental Evaluation

In our evaluation we wanted to provide empirical evidence for our claim that B-trees can be suboptimal in some situations and that they are outperformed by our vector storage. To compare the vector storage, which we implemented in C++, with B-tree based approaches, we implemented both B-tree variants, *AIO* and *OFE*, described in Section 3.3 using the Oracle Berkeley DB [10,11] library in release 4.7. Inspired by [13], in which the authors show that proper B-tree indices over triple tables can already be highly efficient and scalable, we also inserted the data into a standard MySQL 5 database table with three columns, one for subject, one for predicate, and one for object. We then created indices over all three columns. This indexed MySQL table is referred to as *iMySQL*. Finally, we also used the unindexed MySQL database as a baseline for some of the comparisons.

According to our goal we wanted to benchmark the systems with respect to index creation times, required disk space, and data access (or retrieval) time. All experiments were performed on a 2.8GHz, 2 x Dual Core AMD Opteron machine with 16GB of main memory and 1TB hard disk running the 64Bit version of Fedora Linux.

Data Sets As we wanted to measure the behavior of the different storages under various sizes we looked for a suitable typed-graph data set. We chose the Lehigh University Benchmark (LUBM) [6] data set, which models typical academic setting of universities, classes, students, instructors, and their relationships. The advantage of this synthetical data set is that it has an associated data generator which can generate an arbitrary number of triples within the given schema and that it has a number of associated queries with their correct answers. It is, therefore, widely used for benchmarking Semantic Web infrastructure applications. Table 1 summarizes the number of subjects, predicates, objects, and triples for the used data sets.

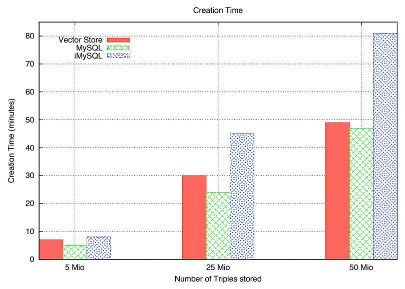
# Triples $\|T\|$	$\|P\|$	$\|S\|$	$\|O\|$	$\|N\|$
5 Mio	18	787,288	585,784	1,191,500
25 Mio	18	3,928,780	2,921,363	5,948,606
50 Mio	18	7,855,814	5,842,384	11,894,568

Table 1. Number of triples, predicates, subjects, objects, and nodes of the used LUBM data sets used

Experimental Procedure To ensure that all systems would have the same starting conditions regardless of any string-handling optimizations we replaced all URIs to unique numerical ids. In addition we also replaced all literals with unique numerical ids mimicking a dictionary index. Note that according to the approach chosen in Hexastore the numerical ids came from two different sets of numbers: one for *predicates* as well as a second one for *subjects* and *objects* (since nodes can be both *subjects* and *objects*).
To further ensure that the data would be loaded uniformly in the same way and equalize possible differences between data-loaders we first built an in-memory index from the data source files and then built the different on-disk indices in exactly the same order. The exception was the MySQL loads, which had to rely on SQL `INSERT` statements instead of calling a direct index API function.

4.1 Creation Time

To compare the index creation time under practical conditions using the LUBM data set we measured the time from the first API-call until the data was entirely written to disk. The results are presented in Figures 3 and 4. Surprisingly, as depicted in Figure 3 the vector storage takes only marginally longer than the simple (unindexed) MySQL insert batch process and already outperforms the indexed MySQL table. As Figure 4 clearly indicates, both Berkeley DB B-tree implementations take a very long time to create the on-disk index and are clearly outperformed by the vector storage: to write a 50 million triples index, vector storage requires about 50 minutes, whereas *OFE* requires 11.5 hours, and *AIO*



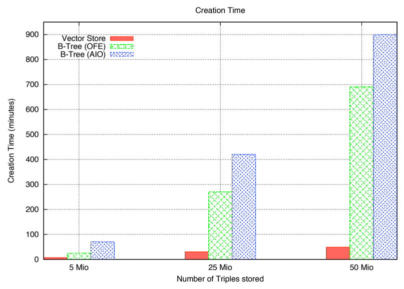


Fig. 3. Comparison of index creation times of the vector storage and indexed (iMySQL) and unindexed MySQL

Fig. 4. Comparison of index creation time of vector storage and OFE/AIO

15 hours. These numbers show clearly that the vector storage index is built much faster than any of the B-tree based indices.

4.2 Required Disk Space

The required disk space was determined by looking at the sum of the file sizes of the respective stores containing a particular amount of triples. The results for that are shown in Figures 5 and 6. We can see in Figure 5 that the ordinary unindexed MySQL triples table requires the least amount of space, which is no surprise. In fact it requires constantly approximately 4.3 times less space than our vector storage approach. The B-tree-indexed MySQL version requires about 33% less space than the vector storage. These findings weaken the original criticism of Hexastore that its six indices use too much space. Indeed we argued in in [17] that Hexastore would use at most 5 times the space than a single index variant under worst-case conditions. We see here that a relational triple store with index on each column (which is necessary given the types of queries typically used in typed graph stores) uses only 33% less space. As shown in Figure 6 both Hexastore-customized Berkeley DB approaches clearly require more disk space than the vector storage. The *OFE* approach requires most storage, i.e. 2.3 times as much as vector storage. The *AIO* approach requires less but still twice as much space as the vector storage. A final observation is that for the LUBM data the vector storage requires about 100 MB per million triples. This linear growth is probably observed due to the uniformity of the interconnections of the data generated by the LUBM generator, which adds new universities when additional numbers of triples are required.

4.3 Retrieval Time

The most important question was how the different systems would compare in terms of retrieval time. To ensure that we measure the time spent by querying

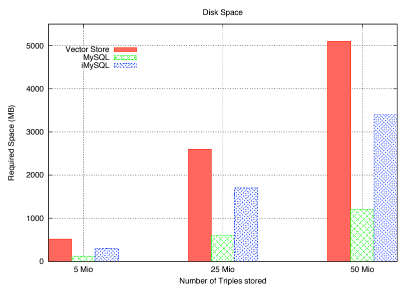


Fig. 5. Comparison of required disk space for indices stored with vector storage and (i-)MySQL

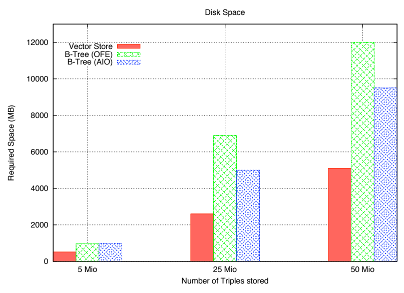


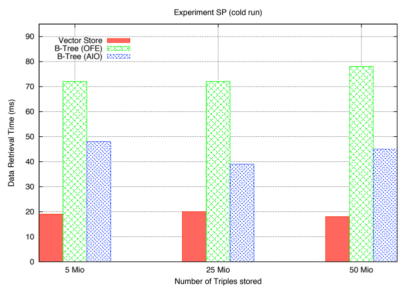
Fig. 6. Comparison of required disk space for indices stored with vector storage and AIO/OFE

the index and not any performed pre- or post-processing operations, we measured retrieval time from the moment all strings in the queries were converted to ids until the final numerical results were retrieved. We also did not take into account the final materialization step because we wanted to compare the immediate behaviors of the different Hexastore implementations without being biased by an additional lookup index. Thus the total time includes finding the key position(s) and fetching the desired data from disk. To avoid measuring the quality of query optimization we restricted ourselves to single triple patterns. Note that we refrained from including retrieval times for MySQL in this experiment, since they were significantly worse than those of the other approaches. Specifically, we evaluated the following requests, each as cold and warm run (i.e., uncached and cached):

1. Retrieve all predicates for a given subject: $< s, ?p, \cdot >$
2. Retrieve all objects for a given subject and predicate: $< s, p, ?o >$
3. Retrieve all subjects for a given predicate: $<?s, p, \cdot >$

The ids chosen for the given subject or predicate were determined by a random generator.

The results of the first request are shown in Figures 7 (cold run) and 8 (warm run). This request is highly selective and does not fetch much data. For the vector storage approach it chooses the SPO index and fetches the subject information from the first file and the set of associated predicates. There is no need to fetch any information from the third file. Both B-tree approaches have to perform key searches and fetch a small corresponding data chunk (in our particular experiment run the given subject had three associated predicates). We can see that the vector storage approach allows data retrieval in this case for a cold run 2–3 times faster than the B-tree approaches and for a warm run about 8 times faster. Given its structure the vector storage can probably answer this query with only two page reads, which are then cached in the warm-run. The B-trees have to perform more page-reads as they had to ”navigate” down the tree. Interestingly, this navigation takes longer for the *OFE*, which has six smaller indices, in the cold-run case but exactly the same time in the warm-run case.



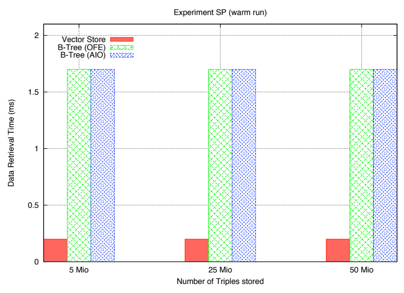


Fig. 7. Comparison of retrieval time for vector storage and AIO/OFE, all predicates for a given subject $< s, ?p, \cdot >$ (cold run)

Fig. 8. Comparison of retrieval time for vector storage and AIO/OFE, all predicates for a given subject $< s, ?p, \cdot >$ (warm run)

The second request fetches all objects for a given subject and predicate. The vector storage needs to touch all three files to collect the required data, including the predicate position determination in the second file within the known range (compare Section 3.2). Both B-tree approaches work similarly to the first request except for searching for a different key. This request is again highly selective, i.e. in our experiment we had five objects connected to the given subject over the given predicate.
The results are presented in Figures 9 and 10. Again we can see that the B-trees are outperformed by vector storage by a factor of $1.5 - 3$ for a cold run and 8 for warm runs respectively. All three approaches have in common that retrieval times remain constant with increasing number of stored triples for highly selective data retrieval. Comparing the results of this query to the last one it is interesting to observe that the cold-run case of this higher specified query is actually evaluated faster than the one that "only" resolves one level. In the warm-run case any advantage is lost due to caching.

The last request retrieving all subjects for a given predicate is analogous to the first one, but has a low selectivity and requires a lot of data to be fetched from disk. In this concrete case, the number of subjects associated with the particular predicate were $376,924$ in the 5 million triples case, $1,888,258$ in the 25 million triples case, and $3,776,769$ in the 50 million triples case. It becomes therefore obvious that in this case page size for a single retrieval is clearly exceeded. Figures 11 and 12 show the corresponding results, where each of the approaches has to retrieve a significant amount of more result triples as the size of the data set increases. Correspondingly, the time needed for retrieval also increases. The large number of retrieved result keys necessitates an increasing number of serial page reads in the second-level index file for the vector storage. Due to the big

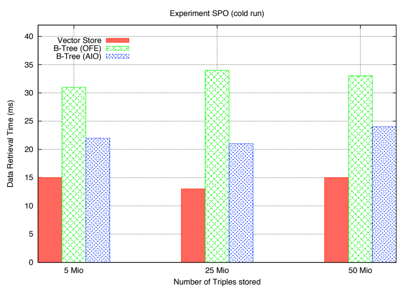


Fig. 9. Comparison of retrieval time for vector storage and AIO/OFE, all objects for a given subject and predicate $< s, p, ?o >$ (cold run)

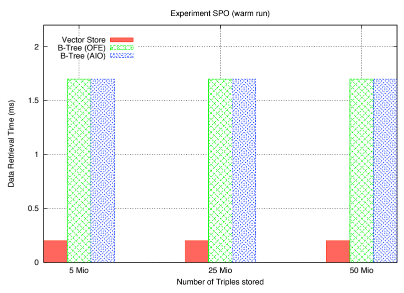


Fig. 10. Comparison of retrieval time for vector storage and AIO/OFE, all objects for a given subject and predicate $< s, p, ?o >$ (warm run)

data amount to be read, the B-tree needs to read more pages and, in case of overflows, determine their positions respectively beforehand. The vector storage outperforms both B-trees by a factor of 1.5 as it can leverage the sequential structure of the second-level index, which is cheaper than traversing overflow pages.

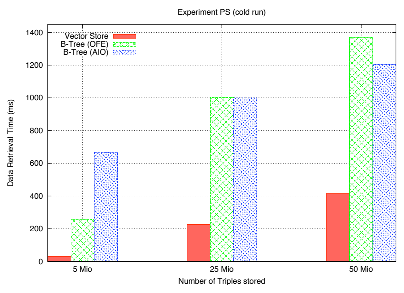


Fig. 11. Comparison of retrieval time for vector storage and AIO/OFE, all subjects for a given predicate $<?s, p, \cdot >$ (cold run)

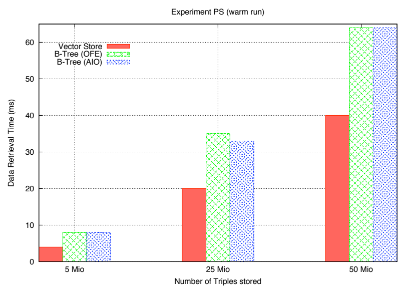


Fig. 12. Comparison of retrieval time for vector storage and AIO/OFE, all subjects for a given predicate $<?s, p, \cdot >$ (warm run)

4.4 Results Summary

The presented results clearly show the advantages of our vector storage scheme. We have shown empirically that it provides significantly lower data retrieval times compared to B-trees. The differences are "only" in the tens or hundreds of milliseconds. However, real world applications are likely to store even more triples and to combine a series of such single graph pattern to a query. This would require a series of lookups such as the ones we discussed here and further widen the gap.

Most computer science algorithms employ some type of time/space tradeoff. In this case, we do not: besides being faster, the novel vector storage approach also only requires half of the space of B-trees. Creation of the on-disk vector storage is fast. Indeed, it is several orders of magnitude faster than creating the corresponding B-trees and almost on par with creating a simple triple table in MySQL without indices.

5 Limitations and Future Work

The goal of our work was twofold. First, we tried to re-open the discussion on the general applicability of trees as the "one-size-fits-all" index and second to present our vector storage format as an efficient on-disk format for semantic web data storage. Even though the experimental evaluation shows that the vector storage exhibits better performance characteristics our findings have some limitations.

First and foremost, our findings are limited by the introductory permanence assumption. If the data stored in a Hexastore would entail many updates, then the overall performance balance might not be so clearly in favor of the vector storage. Again, our basic assumption, supported by many usages of RDF stores in the semantic web, is that such updates are rare and that reloading the whole store in those rare occasions would be faster than using a slower index.

Second, our approach is limited to storing numerical ids requiring a dictionary index for URI-based ids and literals. This approach requires that literal-based query processing elements would be handled by the dictionary index, for example for SPARQL[1] `FILTER` expressions. To address this limitation we are currently investigating an extension of the dictionary index to efficiently handle such elements. Its discussion was, however, beyond the scope of this paper.

Third, as illustrated in the evaluation of the $<?s, p, \cdot >$ query in the last section, all approaches suffer when they need to retrieve very large amounts of data from the second-level index. This is oftentimes the case if the first-level index is not very selective such as when the number of predicates $|P|$ is small compared to the number of triples $|T|$ (18 versus 50 millions in the example query). If such a triple pattern would be queried in the context of a query containing many patterns then the query optimizer would call it at a later stage due to its low selectivity. To speed things up in other cases (or for very loosely bounded queries with lots of results) one could consider to forgoing some of the vector storage compactness for highly unselective first-level indices and introduce the zero-chunks at the second-level. We would foresee that such an approach would only seldomly be chosen: mostly in the PSO and POS second-level indices when $|P| << |T|$. We hope to investigate this further in future work.

Fourth, the LUBM data set is obviously only one possible choice and has its limitations. It is a synthetic data set and has the limitations associated with

[1] `http://www.w3.org/TR/rdf-sparql-query/`

such data. Nonetheless, it is realistic in that it has a small number of relationship types but many entries/triples – a typical observation in real-world data. Also, it is a heavily used data set, which makes our results comparable to a series of other studies. We hope to extend our evaluation to other large data sets in the future. Given our careful theoretical evaluation we are, however, confident that the new experiments will reflect the results presented here.

Last but not least, our approach was conceived in the context of RDF data and the evaluation only employs such data. As a consequence our findings should only be generalized beyond graph-basd data with caution. In particular, the vector storage index we proposed was geared towards serving as a back-end to Hexastore and might not be quite as useful in other setting. Nonetheless, we believe that scrutinizing the basic assumption of the ubiquitous applicability of B-trees is a fruitful takeaway in itself and should be considered in all areas of database research.

6 Conclusions

In this paper we set out to question the universal superiority of B-tree-based index structures and presented a simple vector-based index structure that outperforms the former in typical graph-based RDF-stye data. Specifically, we departed from three assumptions about RDF data: its structurelessness, its permanence, and its mostly path-style queries. Based on these assumptions we proposed to exploit the structurelessness in favor of a novel storage format based on storing the data in indices rather than in their raw format. Exploiting the difference in permanence (compared to traditional transaction-focused RDBMS) we optimized the indices for loads and reads. We showed empirically that the proposed vector storage index for Hexastore outperforms state-of-the-art B-tree implementations both in terms of load time (by over one order of magnitude) and retrieval time (up to eight times faster). We also showed, that the proposed structure had a load time comparable to an unindexed MySQL $< s, p, o >$ table and even slightly outperformed the load into a similar table providing an index over all three columns (which would be needed to answer any realistic queries). Consequently, as the vector-storage-backed Hexastore so clearly outperformed the other solutions, we can confirm that under certain conditions following conventional wisdom can be considered harmful.

As such the presented paper and its vector storage index can be seen as a first step in developing new on-disk storage structures that are better suited for Semantic Web data. The goal of this endeavor is to gather the building blocks for truly scalable fast stores for typed graph (and thus Semantic Web) data.

7 Acknowledgments

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07*, pages 411–422. VLDB Endowment, 2007.
2. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, Asilomar, CA, USA, January 2005.
3. G. Feinberg. Native XML database storage and retrieval. *Linux J.*, 2005(137):7, 2005.
4. S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, Asilomar, CA, USA, January 2007.
5. C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE '00*, page 198. Society Press, 2000.
6. Lehigh University Benchmark. LUBM Website. `http://swat.cse.lehigh.edu/projects/lubm/`, September 2008.
7. X. Meng, D. Luo, M. L. Lee, and J. An. OrientStore: a schema based native XML storage system. In *VLDB '2003*, pages 1057–1060. VLDB Endowment, 2003.
8. T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. In *Vol. 1 of JDMR (formely Proc. VLDB) 2008*, Auckland, New Zealand, 2008.
9. T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD 2009*, Providence, USA, 2009. ACM.
10. M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.
11. Oracle. Berkeley DB Reference Guide, Version 4.7.25. `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/toc.html`.
12. R. V. Guha. rdfDB : An RDF database. `http://www.guha.com/rdfdb/`.
13. L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. In *Vol. 1 of JDMR (formely Proc. VLDB) 2008*, Auckland, New Zealand, 2008.
14. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW '08*, pages 595–604, New York, NY, USA, 2008. ACM.
15. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB '05*, pages 553–564. VLDB Endowment, 2005.
16. R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB '98*, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
17. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Vol. 1 of JDMR (formely Proc. VLDB) 2008*, Auckland, New Zealand, 2008.
18. N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *ICDE '04*, page 54, Washington, DC, USA, 2004. IEEE Computer Society.