B+Hash Tree: Optimizing query execution times for on-Disk Semantic Web data structures

Minh Khoa Nguyen, Cosmin Basca, and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
{lastname}@ifi.uzh.ch

Abstract. The increasing growth of the Semantic Web has substantially enlarged the amount of data available in RDF format. One proposed solution is to map RDF data to relational databases (RDBs). The lack of a common schema, however, makes this mapping inefficient. Some RDF-native solutions use B+Trees, which are potentially becoming a bottleneck, as the single key-space approach of the Semantic Web may even make their $O(log(n))$ worst case performance too costly. Alternatives, such as hash-based approaches, suffer from insufficient update and scan performance. In this paper we propose a novel type of index structure called a B+Hash Tree, which combines the strengths of traditional B-Trees with the speedy constant-time lookup of a hash-based structure. Our main research idea is to enhance the B+Tree with a Hash Map to enable constant retrieval time instead of the common logarithmic one of the B+Tree. The result is a scalable, updatable, and lookup-optimized, on-disk index-structure that is especially suitable for the large key-spaces of RDF datasets. We evaluate the approach against existing RDF indexing schemes using two commonly used datasets and show that a B+Hash Tree is at least twice as fast as its competitors – an advantage that we show should grow as dataset sizes increase.

1 Introduction

The increasing growth of the Semantic Web has substantially increased the amount of data available in RDF[1] format. This growth necessitates the availability of scalable and fast data structures to index and store RDF. Traditional approaches store RDF in relational databases. Mapping RDF to a relational database typically follows one of the following approaches: (1) all triples are mapped to a single three column table – an approach which will result in numerous inefficient self-joins of that table, (2) every property gets mapped to its own three column table [1] – resulting in a high number of Unions for property-unbound queries and a table creation for every newly encountered property type, or (3) draws upon domain-knowledge to map properties to a relational database schema – forgoing some flexibility when adding new properties. Moreover, according to Abadi and Weiss, storing dynamically semi-structured data such as RDF in relational databases may cause a high number of NULL values in the tables, which imposes a significant computational overhead [15, 1]. As a consequence, many native RDF databases have been proposed [15, 10].

[1] `http://www.w3.org/RDF/`

Most native RDF databases propose mapping the RDF-graph to some existing indexing scheme. The most straightforward approach, RDF-3X [10] essentially proposes to store all possible subsets of the triple keys (i.e., s, p, and o from every $< subject, predicate, object >$ triple) as composite keys in a traditional B+Tree structure. This approach results in 15 B+Trees, each of which having large-keyspace (e.g., sizes of $|s| \cdot |p| \cdot |o|$, $|s| \cdot |o|$, etc.) and many entries. Given the $O(log(n))$ access time for single key lookup, this can result in a considerable time overhead for some queries. Consequently, given the ever increasing amount of data to be stored in RDF stores, traditional approaches relying on B+Trees in the sprit of RDF-3X have the potential of becoming a main bottle-neck to scalability [14]. Taking the adaptation to RDF structures to the extreme, Weiss and colleagues [15] propose a specialized index consisting of 3-level cascades of ordered lists and hashes. This approach provides a constant (i.e., $O(c)$) lookup time but has the drawback that updating hashes can become quite costly. Whilst the authors argue that updates in the Semantic Web are oftentimes rare, they are, however, common and should not be dismissed.

In this paper we propose a novel type of index structure called a B+Hash Tree, which combines the strengths of traditional B+Trees (i.e., ease of use and updatability) with the speedy lookup of a hash-based structure. Our main research idea is to enhance the B+Tree with a Hash Map to enable constant retrieval time instead of the common logarithmic one of the B+Tree. The result is a **scalable**, **updatable** and **lookup-optimized**, on-disk index-structure that is especially suitable for the large key-spaces of RDF datasets. Consequently, the main contribution of this paper is the presentation, formalization, and evaluation of the B+Hash Tree.

This paper is structured as follows. In Section 2 we set the stage with a discussion of related work, its benefits and drawbacks. Section 3 then introduces the B+Hash Tree, provides a formalization as well as a cost model thereof, and discusses some its limitations. In Section 4 we empirically compare the B+Hash Tree to the RDF-3X like approach storing the key in a B+Tree. In the final section we summarize our conclusions and discuss future work.

2 Related work

Several architectures for storing Semantic Web data have been proposed. Many of them use relational databases to index RDF data. Row store systems such as *Jena* [16, 17] map RDF triples into a relational structure, which results in creating a giant three column $< subject, predicate, object >$ table. Having a single large table, however, oftentimes results in expensive self-joins; in particular if the basic graph patterns of a query are not very selective. To counter this problem, Jena creates *property tables*, which combine a collection of properties of a resource in one table. Whilst this approach reduces the number of self-joins it (1) assumes that the RDF actually has some common exploitable structure that does not change often over time and (2) has the potential to result in a large number of NULL values where properties are missing from some resources in a table [1]. Hence, this inflexibility and the NULL values may lead to a significant computational overhead [15].

An alternative approach to the property table solution are column stores such as *SW-Store* [1]. For each unique property of the RDF dataset, SW-Store creates a two column table containing the *subject* and the *object*. Assuming a run-length encoding of the column, this provides a compact storage mechanism for RDF data that allows efficient joins, as only the join columns are retrieved from disk (as opposed to the table in row-stores). Nevertheless, if a graph pattern has an unbound property (e.g., $< s, ?p, ?o >$) then an increased number of joins and unions are inevitable [15].

A recent approach is *Hexastore* [15], which stores RDF data in a native disk vector-based index. Hexastore manages all six possible orderings of the RDF triple keys in their own index. In each of the six indices, a triple is split into its three levels: All levels are stored in native on-disk sorted vectors. A lookup of the triple $< s, p, o >$ would, hence, result in a hash-lookup in the first level of s, the result of which would point to a second-level hash that would be used to lookup p, which would point to an ordered list containing o. Given that a hash-lookup can be achieved in constant time, Hexastore provides a constant-time lookup for any given triple. Its six-fold indexing allows a fast lookup of any triple pattern at the cost of a worst-case five fold space cost. The biggest drawback, however, is that Hexastore in its "pure" implementation does not support incremental updates, as inserts would require resorting the vectors of each of the six indices – a time consuming process.

RDF-3X [10] avoids this drawback by storing the data in B+Trees instead of vector lists. Specifically, it stores every possible subset combination of the three triple keys in a separate B+Tree. This approach allows updates and has an $O(log(n))$ complexity for retrieval, updates, and deletion. Note, however, that this approach leads to a huge key-space for some B+Trees (at worst $|s| \cdot |p| \cdot |o|$), as the composite key-space grows in the square of the number of nodes of the RDF graph. Hence, even $O(log(n))$ can become a bottleneck. A similar approach to RDF-3X or Hexastore is *YARS2* [5], which indexes a certain subset of triples in 6 separate B-Trees or Hash Tables. By not treating all possible $< s, p, o >$ subset combination equally, the missed indexes must be created by joining other indexes, which can be a time-consuming process.

A more recent approach is *BitMat* [2], which is a compact in-memory storage for RDF graphs. BitMat stores RDF data as a 3D bit-cube, where each dimension represents the subjects, the objects, and the predicates. When retrieving data the 3D bit-cube can be sliced for example along the "predicates-dimension" to get a 2D matrix. In each cell of the matrix, the value 1 or 0 denotes the presence respectively the absence of a subject and object bounded by the predicate of that matrix. Since bitwise operations are cheap, the major advantage of the BitMat index is its performance when executing low-selectivity queries. Nevertheless, BitMat is constrained by the available memory and, as the authors have shown in their evaluation, traditional approaches such as RDF-3X or MonetDB [8] outperform BitMat on high-selectivity queries.

In real-time systems, *Hybrid Tree-Hashes* [11] have been proposed to provide a fast in-memory access structure. To index data, the Hybrid Tree-Hash combines the use of a T-Tree and a Chained Bucket Hash. Lehman describes the T-Tree [7] as a combination of the AVL-Tree and the B-Tree: similarly to B+Trees, nodes contain multiple elements whilst the binary search strategy of the AVL-Tree is employed for retrieval. To enhance the retrieval time the keys in the nodes of the T-Tree are hashed and the offset to that node is stored as the value. Data retrieval is accomplished by a lookup in the Chained Bucket Hash, which retrieves the offset value for the given search key. Then, the node that holds the data is accessed directly without traversing the T-Tree. Whilst T-Trees perform well as an in-memory data structure, their usage as an on-disk structure is problematic: First, the use of a binary tree results in deep trees, which in turn results in many disk pages being accessed. Second, the binary tree nature of the T-Tree makes it cache oblivious: range queries are highly expensive, as one has to continuously "jump" up and down the tree for traversal, leading to costly disk-seek operations. This is in contrast to the B+Tree, where the data is stored in the leaves as a linked list, resulting in fewer disk page seeks and cache awareness. To the best of our knowledge the Hybrid Tree-Hash is the most similar structure to our proposed B+Hash Tree index. The main difference is that we optimized our index structure for disk-based operations, whereas Hybrid Tree-Hashes were optimized for in-memory retrieval.

3 B+Hash Tree

In this section we introduce our B+Hash Tree, a scalable, updatable and lookup-optimized, on-disk index-structure combining the strengths of B+Trees and Hashes. First, we describe the structure of the B+Hash Tree and elucidate its operations. Second, we provide a time and space complexity analysis of the relevant B+Hash Tree operations. Finally, we discuss some limitations of the B+Hash Tree and suggest appropriate solutions.

Note that throughout this section we propose to index a RDF graph akin to the RDF-3X approach. In other words, we propose a separate index for each possible subset combination of the $< s, p, o >$ triples. Hence, a $s_1p_3o_2$ triple is stored in level 1 with the key s_1, in level 2 with the composite key s_1p_3, and finally in level 3 using the composite key $s_1p_3o_2$. This structure allows retrieval of all triple patterns with a single lookup [15, 10]. In contrast to RDF-3X, we propose to use B+Hash Trees as opposed to B+Trees. Like all other approaches we also propose to dictionary encode all literals.

3.1 B+Hash Tree Description

The architecture of the B+Hash Tree comprises two core elements: A B+Tree and the Hash Map. Here, we first explain how these elements are combined to form a joint index and then elaborate on the main operations.

We use the standard B+Tree as the basis for our B+Hash Tree. Recall that B+Trees are optimized for disk access. In particular, nodes of the trees are adapted to the size of a disk page to facilitate caching and limiting disk access. Additionally, all values are stored in the leaf nodes of the tree, which are interlinked, allowing *fast index-range queries*. More information about B+Trees can be found in [4].

As Figure 1 illustrates, the B+Hash Tree combines the B+Tree with a *Hash Map*. Specifically, to improve retrieval performance the leaf nodes of the B+Tree are being hashed. Each bucket entry in the Hash Map contains a key (or ID), an offset containing the address of the element designated by the key on disk and the count of distinct elements sharing the same prefix. The prefix being the anterior part of the key. As an example consider the $s_2p_1o_4$ triple: in a level 2 index the prefix should be s_2p_1; in a level 1 index s_2.

A retrieval operation in a B+Hash Tree starts with a lookup of the offset in the Hash Map using the key and then accesses the node holding the search key directly without traversing the tree. The count aids both the query optimizer (e.g., to gauge selectivity) and the execution of range scans (indicating the number of elements to be read). As exemplified in Figure 1, "Bucket 1" indicates that there are two predicates for subject S_1 ("count = 2").

Note that not every leaf node needs to be hashed. Usually, only the leaf nodes containing the smallest suffix for a given prefix have to be hashed – an approach which we refer to as *overall hashing*. For example, on level 1 of the *spo* index only the leaf nodes where the S key changes need to be hashed, as illustrated in Figure 1.

Alternatively, the hash can be tuned to contain the most popular keys – an approach which we refer to as *cached hashing*, as it employs a Hash Map akin to a cache of the B+Hash Tree's contents. Cached hashing can be tuned to reduce space consumption compared to overall hashing. This space saving comes at a cost of slowing down non-frequent accesses. Therefore, the empirical evaluation in this paper focuses on *overall hashing*.

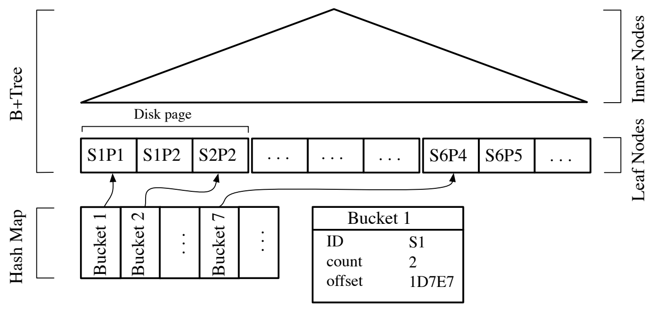

Fig. 1. Architecture of the B+Hash Tree: spo index level 1

3.2 Basic Operations

The B+Hash Tree has three basic operations: get, insert, and delete. To enable the same model interface as Hexastore [14], we split the get operation into two distinct ones:

getIdx(a) Given a triple pattern a, look up the offset and the count of elements in the Hash Map: $getIdx(a) : \text{offset}_a,\ \text{count}_a$

getSet(offset, count) Given an offset and a count, retrieve $count_a$ elements from the leaf node:

$$getSet(\text{offset}_a,\ \text{count}_a) : \text{set}_a$$

Hence, a traditional lookup would be composed of $getSet(getIdx(a))$, a very common index range-query, not to be mistaken with a SPARQL range-query. Furthermore, in contrast to Hexastore, we have the following data changing operations:

insert(a) Insert triple a into the B+Hash Tree:

$$insert(a) : \text{void}$$

delete(a) Delete triple a from the B+Hash Tree:

$$delete(a) : \text{void}$$

Note, that an insert, respectively a delete operation may cause a rebalancing of the B+Tree. If this occurs then the keys in the newly created leaf node have to be verified if an update of their page offset's in the Hash Map is required – a process whose cost depends on the on-disk implementation of the B+Tree.

Index range-scans are quite common operations in Semantic Web applications. Just consider retrieving a list of all predicates that connect subject s with an object o. Such an operation results in the triple pattern $< s, ?p, o >$. In a B+Tree, neighboring leaf nodes are connected to each other, enabling sequential iteration through the pertinent leaf nodes. Hence, the logical way to retrieve the answers for this triple pattern in a RDF-3X like index is to iterate through the level 2 *sop* index starting from the "smallest" (or first indexed) p_s. Hence, the B+Tree is first being traversed from the root node down to the leaf node to find the node for sop_s and then sequentially iterating through the leaf nodes until a different object is encountered. When using a B+Hash Tree, in contrast, we first lookup the key *so* in the Hash Map of the appropriate index followed by the B+Tree traversal like in the traditional tree. Hence, we reduce the tree traversal down from the root node to the first leaf node – a $O(log(n))$ operation – to a single hash lookup – a constant time ($O(c)$) operation. From a certain data size the B+Hash Tree will, hence, outperform the B+Tree. We elaborate this fact by doing a simple complexity analysis of the most important operations in Section 3.3.

3.3 Time and Space Complexity Considerations

In this section we provide a formalization for the time and space complexity of the most important B+Hash Tree operations. Note that since we are talking about a disk-based index structure, the hard drive access times, as the slowest component, are likely to dominate in-memory operations. Hence, the time complexities of B+Tree operations are measured by the number of page reads. Given that the B+Tree only stores the actual data in the leaves (the inner nodes of the trees are "only" used to organize the index) and that the data for a single key typically fits into one page, the number of page reads for any simple lookup is solely dependent on the tree height. Assuming that we denote the order (i.e., the # of index elements per inner node) of the B+Tree as d and the number of entries as n, Comer [4] elucidates the height of a B+Tree as:

$$height = log_d\,(n) \tag{1}$$

Time complexity: Given that most queries do not solely rely on simple key lookups, but actually retrieve multiple elements (e.g., find all objects for a subject) we also need to account for the number of leaf pages that need to be read. Assuming that the values fit on s pages then the number of page reads for a query can be defined as:

$$Reads_{B+Tree} = log_d\,(n) + s \tag{2}$$

Note that this formula assumes that the values are (i) stored on consecutive pages and (ii) that the leaf-pages are interlinked, which the B+Tree guarantees.

A logarithmic complexity is, obviously, excellent and has served the IT community well in many applications such as relational databases. If the key-space, however, grows enormously and the number of separate accesses for any given query is large – both of which are especially true for SPARQL queries – then even a logarithmic complexity may slow the execution down. The main rationale behind the B+Hash Tree is to cut down the time complexity of these reads using a Hash Map with its constant-time accesses. As a result, the complexity of a simple lookup is 2 – one access to retrieve the bucket hash entry and another one to access the leaf node. In addition, as before, when performing index range-scans we need to read all the pages which contain the data, resulting in:

$$Reads_{HashMap} = 2 + s \tag{3}$$

Consequently, using the Hash Map results in fewer disk page reads than the B+Tree, if the height of the B+Tree exceeds two levels. With Equations 2 and 3 we can calculate the number of disk page accesses for a set retrieval (range query). To estimate the total retrieval time we multiply the number of disk page reads with the average disk page access time of a common hard disk. However, in reality there are three different types of disk page reads: `random read`, `sequential read`, and `cache read`. Random disk page reads are the slowest kind, as the seek operation requires the HDA (Head Disk Assembly) to jump to another track. Sequential reads (i.e., reading some data from the same track) consists of waiting until the required disk page arrives under the HDA, which depends mainly on the rotational speed of the hard disk. The fastest form of access is the cache read, i.e. when a previously read page is found in the on-board disk cache and no mechanical action is required to retrieve the data.

In contrast to B+Trees – on-disk optimized data-structures enabling efficient (sequential) scans – Hash Map data lookup and retrieval is usually random, due to the lack of locality. Again, this is dependent on the actual hash implementation.

Inserting and deleting in the B+Hash Tree adds an additional level of complexity. Assuming that the Hash Map has a sufficient number of free buckets, then insert/delete operations in a B+Hash Tree add – in theory – just one more write operation over the B+Tree: the update of the Hash Map. However, if a leaf node in the B+Tree has to be split or merged during an insert, respectively delete operation, then the page offsets of the keys in the affected leaf nodes may have to be updated in the Hash Map. Depending on the B+Tree implementation, usually, only the keys where the prefix changes in the newly created or merged node, may need an update. Worst case, this is an $O(n)$ operation where n denotes the number of keys with different prefixes in the affected node.

If the Hash Map is full, however, or there are too many collisions, then the HashMap needs to be reorganized (rehashed) resulting in a higher cost operation [6, 9].

Space complexity: The space consumption of a B+Tree depends on the number of nodes. In practice the size of the node is chosen as such to match the size of a disk page (usually 4, 8 or 16 KB), therefore the space occupancy of a B+Tree is the number of inner nodes plus the number of data containing leaf nodes times the size of a disk page:

$$Size_{B+Tree} = Size_{Page}(\sum_{level=0}^{h} Page_{level} + Page_{leafs}) \tag{4}$$

where $Page_{level}$ denotes the number of nodes respectively disk pages on level l of the tree and $Page_{leafs}$ denotes the number of data containing leaf-nodes.

The space consumption of a B+Hash Tree *additionally* adds the size of the Hash Map, which can be expressed by the number of chunks holding hash buckets. Chunks in turn, are typically sized to match a disk page. Consequently, the size of the Hash Map is:

$$Size_{HashMap} = Size_{Page} \cdot \#Chunks \tag{5}$$

where $\#Chunks$ denotes the number of chunks needed for the Hash Map.

Summarizing, we find that the B+Hash Tree provides a better complexity for reads compared to B+Trees. This advantage comes at the cost of additional space shown in Equation 5 and some time to maintain the Hash Map. We would argue that the cost in most cases is relatively small for Semantic Web applications. Addressing the former, we believe that given the price of disk space the additional space complexity for the Hash Map is negligible. Addressing the latter, it can be argued that, assuming a sufficiently large hash and a higher ratio of reads than writes/updates, the frequency of hash map reorganization operations can be limited to a few instances.

Database space complexity: Consequently, given the multi-ordering multi-level index structure chosen, the total space consumption of a full database index (all possible index orderings for triples) is:

$$Size_{index} = \sum_{ord}^{ORDS} \sum_{lvl=0}^{2} (Size_{ord,lvl}(B+Tree) + Size_{ord,lvl}(HashMap)) \tag{6}$$

where ord represents the current index type (i.e. `SPO`, `OPS`, etc. $\in ORDS$), while lvl denotes the current index level.

3.4 Limitations and Solutions

The *overall hashing* technique enhances the retrieval time of the B+Tree at the cost of space consumption as described previously. Considering empirical evidence such as Kryder's Law [13], space consumption entails a rapidly decreasing economical cost versus the high cost of query answering in today's DBMS

(Database Management Systems), where real time or near real time response is often required.

For high update rate scenarios, the extra overhead induced by the B+Hash Tree structure can be nullified by considering a parallel architecture where the traditional B+Tree part of the index would reside on one disk unit while the Hash Map would be served/updated on another disk. Furthermore, if the update cost of the Hash Tree is higher than that of the B+Tree at one time, one can still serve queries by reverting to the $O(log(N))$ worst case performance of the B+Tree with no time penalty versus traditional indexing approaches. Hence, in such a setup the worst-cost complexity of a B+Hash Tree is equal to a B+Tree (whenever the Hash Map is reorganized). In most cases (when the Hash Map is available), however, the retrieval complexity would be linear (as shown in equation 3).

If space is a hard constraint (e.g. in an embedded system) one can change the policy of updating all keys (*overall hashing*) to *cached hashing*, where similar to cache policies, only "popular" keys are stored. In general, if incremental updates are rare, then using the *overall hashing* approach is recommended.

System cache impact: As argued above, the Hash Map significantly improves access to leaf nodes in comparison to a "pure" B+Tree. Nonetheless, in reality any modern computing system employs a hierarchy of cache systems starting with the *disk cache* at the lowest level.

When considering the disk cache, there are four possible situations: (i) none of the structures are in cache – in this case the the B+Hash Tree will provide the highest performance as described previously, (ii) the Hash Map of the B+Hash Tree is in cache while the B+Tree is not – the B+Hash Tree will outperform the former, (iii) the B+Tree is cached and will gain the highest performance, and (iv) both structures are in cache, which is the most likely scenario.

Nowadays, the typical size of the disk cache varies between 8 and 64 MB. Performing a simple space consumption estimation for a B+Tree holding 30 million triples, the total number of inner nodes can be approximated to 0.5 million assuming a standard page size of 4KB. This results in an approximately 2 GB inner node index of the B+Tree. Given usual cache eviction approaches (such as *Least Recently Used*) it is likely that only the higher inner node levels of such a B+Tree will be cached while the lower levels will mostly reside on disk. In this case, the B+Tree structure will have to read $h - k$ pages from disk to reach the queried leaf page, where h is the height of the tree and k represents the number of levels in the cache. When the dataset is large enough then $h - k > 2$ (where 2 is the lookup cost in the B+Hash Tree). In these cases, neglecting the OS filing system cache, the B+Hash Tree will outperform the traditional approach. Due to the growth of the Semantic Web we expect that this gap will grow.

4 Evaluation

To evaluate the performance of the B+Hash Tree compared to a B+Tree we created a prototype for both indices, which we used in conjunction with an in-memory simulation of the on-disk structures. The advantage of the disk simulation is that it can monitor actual disk page accesses regardless of wall-clock-time confounding factors such as disk-cache and other operating system processes. The main disadvantage of this method is that all evaluations were constrained by the available memory (72GB). Hence, the largest dataset we could run contained 31 million triples.

Given that the B+Hash Tree is solely an index and not a full-fledged triple store, we used the query optimizer of a triple store called TokyoTyGR[2] to obtain the index access traces for a query and then ran the traces on the B+Hash Tree. Since we solely evaluate the index and not the overall triple store in this paper, we limited our measurements to the retrieval time of the B+Hash Tree (or B+Tree) index structure and not complete query answering such as parsing of the SPARQL query or selectivity estimations – these measures would have been the same for both index structures. Moreover, we do not monitor the additional sequential page reads for range scans, because as discussed in Equations 2 and 3, the number of additional page-accesses due to the scan is identical (i.e., s) for both considered approaches. Therefore we only track the differentiating parts of the equations, which excludes the scans. To maximize disk access performance we set the B+Tree page size to the size of the disk page size (4 KB).

Consequently, each result presented in Figures 2a–f shows the number of disk page reads of a whole index access trace during a single query execution and not just a single lookup of an element in the index. All test simulations use the *overall hashing* technique.

For the traditional B+Tree approach, we discriminate between sequential and random disk page reads in the diagrams. In the case of the B+Hash Tree, we present only "all reads", as most reads from a Hash Map are random.

The space consumption was calculated by applying the formulas provided in our complexity analysis of the B+Hash Tree.

In the remainder of the section we first present the two datasets—the Berlin SPARQL Benchmark dataset and Yago—with their associated queries as well as performance measures, and then discuss the results and their limitations.

4.1 The Berlin SPARQL Benchmark Dataset

The first dataset, the Berlin SPARQL Benchmark[3] (BSBM), is a synthetic dataset. It simulates an enterprise setting, where a set of products are offered by different vendors and clients can review them [3]. The dataset can be generated using the available data generator and the BSBM provides twelve distinct SPARQL queries.

Some of these SPARQL queries contained "REGEX", "OFFSET", "UNION", "DESCRIBE", and other expressions, which TokyoTyGR does not support. Therefore, we selected a subset of 5 queries without these elements from which

[2] The TokyoTyGR is an extension of the Hexastore [15] triple store. It can easily accommodate inserts/deletes and has a state of the art query optimizer based on selectivity estimation techniques.

[3] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark`

we further removed "FILTER"-expressions and "OPTIONALS", as they would be handled in the exact same way by both the B+Hash Tree and the B+Tree. The result are the 5 queries (denoted as Query 1 to 5), which we list in Appendix A in ascending query complexity (in terms of variables and triple-patterns).

To compare the performance of the index structures with increasingly larger dataset sizes, we created five different datasets ranging from 1 million to 31 million triples. The details of these datasets are shown in Table 1. Note that the number of unique predicates in all datasets is the same while the number of unique subjects and objects increases. The results of running the five queries on

Dataset	# Triples	S	P	O
BSBM	1,075,597	99,527	40	224,032
	2,160,331	200,096	40	443,753
	4,969,590	452,507	40	966,120
	10,632,426	966,013	40	1,979,668
	31,741,096	2,863,525	40	5,297,862
Yago	16,348,563	4,339,591	91	8,628,329

Table 1. Number of triples, unique subjects, predicates, and objects in the datasets

the different dataset sizes are shown in Figure 2a–e. For every query and dataset, the B+Hash Tree is in each case faster (i.e., uses fewer disk reads) than the traditional B+Tree approach. Furthermore, the difference between the number of disk page reads for both structures increases with the size of the dataset. Therefore, for the 31 million triple dataset, the B+Hash Tree performs twice as fast as the B+Tree.

Figure 2g) shows the disk space consumption of both data structures. As expected, the B+Hash Tree consumes more space. We believe, however, that for most applications, trading-in 20% of the space against a halved access-time is a worthy trade-off.

4.2 Yago Dataset

To complement the synthetic BSBM dataset with a real-world representative, we employed Yago[4] as a second dataset, which consists of facts extracted from Wikipedia. Again, Table 1 shows the characteristics of the Yago dataset, which contains about 16 million distinct triples and almost 13 million resources.

The Yago dataset does not come with a defined set of queries. Therefore, we constructed three queries (numbered Queries 6–8; shown in Appendix A), which simulate a realistic information request such as "What actors play in American movies?" or "Which scientist is born in Switzerland?". In addition, we ensured that two queries (Q6 and Q7) have a low selectivity and, therefore, "touch" a lot of the data, and one query (Q8) is highly selective and is, therefore, expected to "touch" fewer disk pages.

To simplify the comparison, and as Yago has only one dataset size, we graphed all results in a single plot (Figure 2f). Also, given the large number distribution, the plot employs a *logarithmic scale* on the x-axis.

[4] `http://www.mpi-inf.mpg.de/yago-naga/yago/n3.zip`

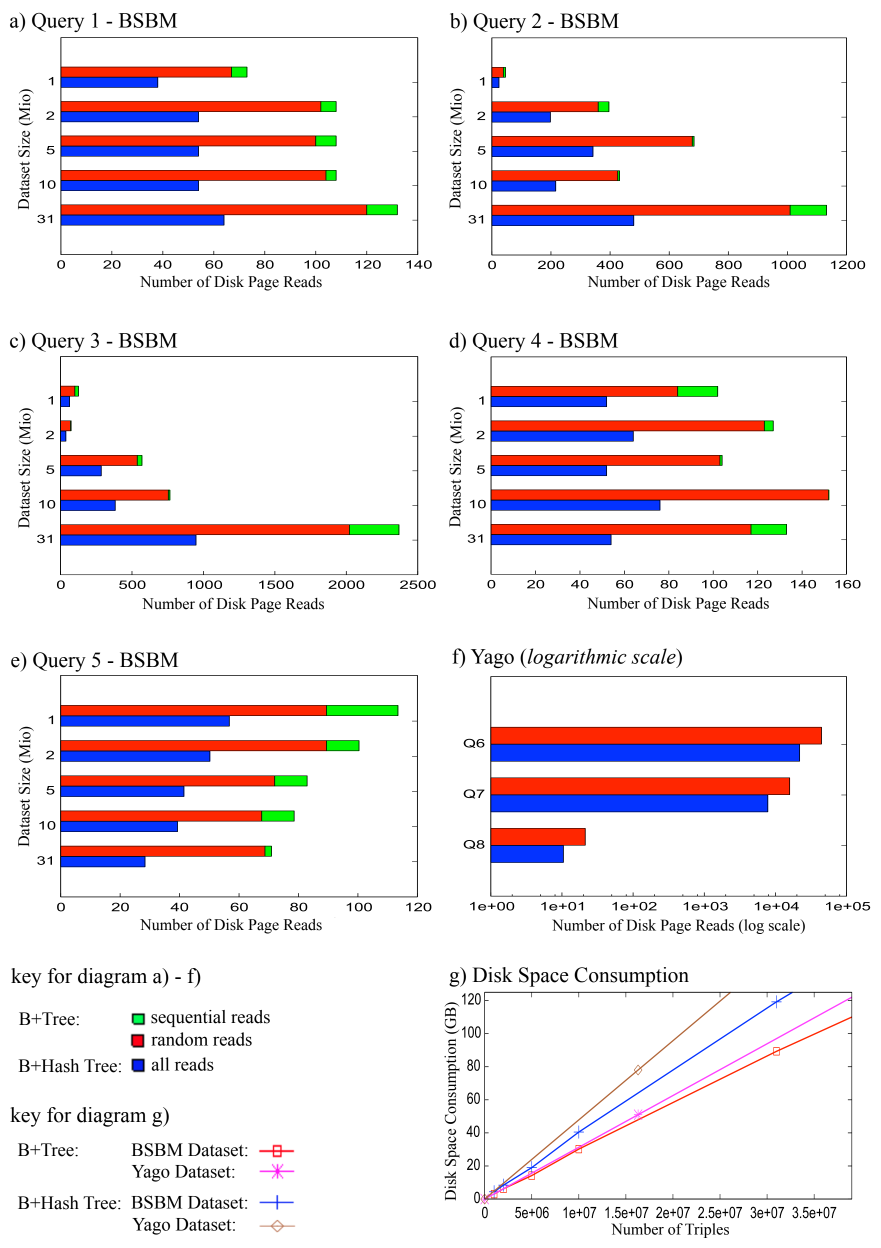

Fig. 2. Number of disk reads for the queries and disk space consumption

Again, the B+Hash Tree outperforms the traditional B+Tree by accessing about half the pages. As expected, Query 8 reads fewer disk pages. It is note-worthy to observe that the performance improvement seems independent of the query's selectivity.

Figure 2g again graphs the space consumption. Note that the higher space consumption of Yago compared to BSBM can be attributed to the number of distinct values for URIs and literals: while the 31 million BSBM dataset has 8 million distinct values, Yago's 16 million triples have 13 million distinct values.

4.3 Discussion and Limitations

Our results confirm the theoretical analysis that the performance improvement of the B+Hash Tree compared with the normal B+Tree increases with the size of the dataset. To further illustrate the result, Figure 3 graphs the speed-up factor against the dataset size. Observably, the speed-up factor increases with the size of the number of triples inserted and therefore confirms Equations 2 and 3 in practice. Given that Equation 2 grows logarithmically and Equation 3 is constant (ignoring the scan element) we would expect the difference to grow logarithmically with dataset size.

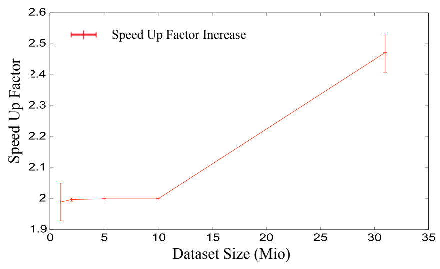

Fig. 3. Speed Up Factor vs. Dataset Size (error bars show results for different queries)

We also investigated, if the query complexity in terms of number of query variables (which varies from 2 to 11) and triple patterns (which varies from 2 to 12) affects the speed performance. Visually, Figure 2 indicates no such difference. Indeed, a pairwise, two-tailed t-test confirmed that the speed-up between queries remains constant with a signifiance of far below 0.1%.

This B+Hash Tree prototype consumes a considerable amount of space which can be traced to four main reasons. First, by storing all possible subset combinations of a triple we gain speed in query answering, as highlighted in the Hexastore project. Second, we set the size of a disk page to 4KB which entails B+Trees containing more inner nodes, thus consuming more space. Third, we use 20 byte instead of 8 byte keys typically used in DBMS, as we wanted a global rather than a "table-local" key space. And last, we have not yet considered index compression, further reducing the consumed space while still maintaining access speed, as shown by Neumann [10].

Building the RDF index in the B+Hash Tree from scratch can increase the build time significantly compared to the B+Tree depending on the Hash Map implementation. A more thorough investigation of this issue is still open.

Limitations: We see three major limitations in our evaluation; not of our proposed approach. First, all our empirical calculations are based on an in-memory simulation of an on-disk B+Hash Tree structure. To mitigate this problem we ensured that our hard-disk model was as accurate as possible and we parameterized it with present-day hard disk parameters. In addition, we measured disk page accesses rather than wall-clock time, essentially focusing on the most time-intensive element of the queries. Consequently, we are confident that our findings generalize to the on-disk setting.

Second, our simulation disregards disk caches. Disk-caches in modern day operating systems are intricate structures that any on-disk index would share with other disk-accessing processes. This makes an adequate simulation a highly-complex issue and may mislead evaluations in a real, on-disk setting. As discussed in Section 3.4, however, we would expect a B+Hash Tree to outperform a traditional B+Tree even in the presence of disk-caches.

Last, we used the TokyoTyGR RDF store to obtain index-access traces for each of the experimental queries. It could be argued that the results of our experiments are biased towards its query optimizer. Given that TokyoTyGR uses a selectivity-based optimizer [12] like most other modern triple stores (such as Hexastore or RDF-3X) the danger of a systematic bias seems limited—especially since it seems unlikely that other query optimizations would lead to a vastly different access pattern between a B+Hash Tree and a B+Tree. Nonetheless, a completely different query optimization approach might require the re-evaluation of our results.

Note that even in light of these limitations, the use of a disk-simulation had several advantages: First, it allowed us to isolate the evaluation from confounding effects (e.g., by the operating system). Second, it allowed us to meticulously distinguish between different types of disk accesses—an undertaking that is non-trivial in a real on-disk structure. Nonetheless, a future on-disk evaluation will have to complement our current findings.

5 Conclusion and Future Work

In this paper we proposed the B+Hash Tree—a scalable, updatable, and lookup-optimized on-disk index-structure especially suitable to the Semantic Web with its large key-space. We showed that using a Hash Map to store the offsets of the leaf nodes successfully trades a slight increase in database size against significantly reduced retrieval time. When used in the context of existing index approaches such as Hexastore and RDF-3X, this will allow for effective retrieval of all possible triple patterns.

To evaluate the B+Hash Tree empirically, we benchmarked the number of page reads (and hence indirectly retrieval time) using two well-established Semantic Web test datasets. As the results show, the B+Hash Tree consistently requires approximately half the page reads of a B+Tree. Note that this difference is expected to grow with the logarithm of the dataset size.

The current implementation of the B+Hash Tree was only used in the simulated measurements. We, therefore, intend to implement a fully operational disk-based version of the index and evaluate it with several "truly" large datasets. In this context we also want to investigate the interaction between the B+Hash Tree and the disk-cache. Last but not least, we intend to consider the use of index-compression to develop even more efficient index structures.

Research in index structures has come a long way, from the early days of simple re-use of traditional, relational, row-based data base indices to the construction of specialized structures such as Hexastore and RDF-3X. We believe that the B+Hash Tree provides a new quality to this exploration. It does not smartly reuse existing structures like its predecessors but investigates a Semantic Web data specific algorithmic extension. As such it calls for the exploration of index structures that exploit the structural and statistical idiosyncrasies of Semantic Web data. The result of this exploration should be truly web-scalable triple stores, which lie at the very foundation of the Semantic Web vision, and the B+Hash Tree can provide a major building block towards that foundation.

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

References

1. D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.
2. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th international Conference on World Wide Web*, pages 41–50, Raleigh, North Carolina, USA, 2010. ACM.
3. C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Sematnic Web and Information Systems*, 5:1–24, 2009.
4. D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121—137, 1979.
5. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: a federated repository for querying graph structured data from the web. In *ISWC'07/ASWC'07: Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, pages 211–224, Berlin, Heidelberg, 2007. Springer-Verlag.
6. J. Keller. Hashing and rehashing in emulated shared memory. In *Proceedings of the 3rd Workshop on Parallel Algorithms (WOPA)*, 1993.
7. T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303. Morgan Kaufmann Publishers Inc., 1986.
8. S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2(2):1648–1653, 2009.
9. W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, 7(1):5–19, 1975.
10. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008.

11. C. Ryu, E. Song, B. Jun, Y. Kim, and S. Jin. Hybrid-TH: a hybrid access mechanism for Real-Time Main-Memory resident database systems. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, page 303. IEEE Computer Society, 1998.
12. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th International World Wide Web Conference*, pages 595–604, Beijing, China, 2008. ACM.
13. C. Walter. Kryder's law. *Scientific American*, Aug. 2005.
14. C. Weiss and A. Bernstein. On-disk storage techniques for semantic web data - are B-Trees always the optimal solution? In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, Oct. 2009.
15. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1:1008–1019, 2008.
16. K. Wilkinson. Jena property table implementation. In *Proceeding of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2006.
17. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *Proceedings of the First International Workshop on Semantic Web and Databases (SWDB)*, volume 3, pages 7–8, 2003.

A Appendix

The Berlin SPARQL Benchmark (BSBM) Dataset queries:

Query1:
```
SELECT ?product ?label WHERE { ?product <label> ?label;
    <type> <Product>; <productFeature> <ProductFeature50>;
    <productFeature> <ProductFeature580>; <productPropertyNumeric1> ?value1 . } LIMIT 10
```

Query2:
```
SELECT ?offer ?price WHERE { ?offer <product> <Product62>; <vendor> ?vendor;
    <publisher> ?vendor . ?vendor <country> <US> . ?offer <deliveryDays> ?deliveryDays;
    <price> ?price; <validTo> ?date . } LIMIT 10
```

Query3:
```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName WHERE {
    ?review <reviewFor> <Product197>; <title> ?title; <text> ?text; <reviewDate> ?reviewDate;
    <reviewer> ?reviewer . ?reviewer <name> ?reviewerName . } LIMIT 20
```

Query4:
```
SELECT ?product ?productLabel WHERE { ?product <label> ?productLabel .
    <Product613> <productFeature> ?prodFeature . ?product <productFeature> ?prodFeature .
    <Product613> <productPropertyNumeric1> ?origProperty1 .
    ?product <productPropertyNumeric1> ?simProperty1 .
    <Product613> <productPropertyNumeric2> ?origProperty2 .
    ?product <productPropertyNumeric2> ?simProperty2 . } LIMIT 5
```

Query5:
```
SELECT ?label ?comment ?producer ?productFeature ?propTextual1
    ?propTextual2 ?propTextual3 ?propNumeric1 ?propNumeric2 WHERE {
    <Product2227> <label> ?label; <comment> ?comment; <producer> ?p .
    ?p <label> ?producer; <publisher> ?p; <productFeature> ?f . ?f <label> ?productFeature .
    <Product2227> <productPropertyTextual1> ?propTextual1;
    <productPropertyTextual2> ?propTextual2; <productPropertyTextual3> ?propTextual3;
    <productPropertyNumeric1> ?propNumeric1; <productPropertyNumeric2> ?propNumeric2 . }
```

The YAGO Dataset queries:

Query6:
```
SELECT ?actor ?p WHERE { ?actor <actedIn> ?p .
    ?p <type> <wikicategory_American_films>  . }
```

Query7:
```
SELECT ?scientist WHERE { ?scientist <type> <wordnet_scientist>; <bornIn> ?city .
    ?city <locatedIn> <Switzerland> . }
```

Query8:
```
SELECT ?person WHERE { ?person <graduatedFrom> <University_of_Zurich>
    ?person <hasWonPrize> ?price . }
```