Efficient RDF Dictionaries with B+ Trees

Gurkirat Singh*
Computer Science and Engg
Indian Institute of Technology Kanpur
gurkirat@cse.iitk.ac.in

Dhawal Upadhyay*
Computer Science and Engg
Indian Institute of Technology Kanpur
dhawal@cse.iitk.ac.in

Medha Atre
Computer Science and Engg
Indian Institute of Technology Kanpur
atrem@cse.iitk.ac.in

ABSTRACT

Resource Description Framework (RDF) graphs are widely used for representing *semantically linked* data in various domains. Many modern RDF specific storage, indexing, and query optimization systems internally represent the node and edge labels of the RDF graphs as integer IDs. Hence they require *dictionaries* for converting the strings in a SPARQL query into their corresponding IDs, and the SPARQL query results in the ID form into their corresponding strings. Most of the SPARQL query processing systems have focused on the techniques for indexing of RDF graphs and the optimization of the *joins* in the SPARQL *Basic Graph Pattern* (BGP) queries, but the *dictionaries* that map RDF graph string labels to the IDs and back have remained a neglected component. Dictionaries are important for an "*end-to-end user experience*" of SPARQL query processing over large RDF graphs.

Hence, in this paper, we have specifically focused on building efficient RDF dictionaries using B+ trees. Our key contributions are – (a) building an *ensemble* of B+ trees, instead of one giant B+ tree, to maintain a low average height across the ensemble, (b) a hashing technique for storing the string labels as search-keys to reduce the space consumption, maintain a higher B+ tree order, and more uniform search-key distribution across memory pages, (c) using multi-core parallel processing for fast dictionary construction, and (d) novel bulk reverse lookup methods. We have also presented an extensive experimental evaluation of our techniques over a set of 126,444,964 labels of a real-life DBPedia RDF graph.

CCS CONCEPTS

• **Information systems** → *Data scans*;

KEYWORDS

RDF, SPARQL, Dictionaries, B+ tree, Multi-core Processing

ACM Reference Format:
Gurkirat Singh, Dhawal Upadhyay, and Medha Atre. 2017. Efficient RDF Dictionaries with B+ Trees. In *Proceedings of The ACM India Joint International Conference on Data Science & Management of Data (CoDS-COMAD '18).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3152494.3152506

*Equal contribution.

CoDS-COMAD '18, January 11–13, 2018, Goa, India

ACM ISBN 978-1-4503-6341-9/18/01…$15.00
https://doi.org/10.1145/3152494.3152506

1 INTRODUCTION

Resource Description Framework (RDF)[1] is widely being adopted as the standard for representing *semantic relationships*, e.g., DBPedia[2] is an RDFized version of the Wikipedia network, UniProt[3] is an RDFized protein sequence information network derived from genome sequencing project. RDF graphs are directed edge-labeled multi-graphs, where node labels can be *Internationalized Resource Identifiers* (IRIs) or literals, and edge-labels are always IRIs. Every unique edge in the RDF graph is called a *triple*, and is denoted as (S P O), where P is the label on the edge from node S to node O. SPARQL[4] is the standard query language for RDF, and SPARQL's *Basic Graph Pattern* (BGP) queries have a close resemblance to the SQL inner-join queries. In fact, if all the triples in an RDF graph are stored in a relational database as a 3-column table (S P O being the three columns), then every SPARQL BGP query can be methodically translated into an equivalent SQL inner-join query [7].

In Figure 1, we have shown an example of a small RDF graph[5], a SPARQL BGP query over it, and a serialized 3-column tabular representation of this RDF graph, where each triple is a row in the table, and S, P, O are the three columns. We have also shown the SQL query equivalent to the given SPARQL BGP query, which can be run as three *self-joins* on the RDF table. This query asks to find the *friends* of *:Jerry* that have *:actedIn sitcoms* that had their *location* in the *:NewYorkCity*. Note that the SPARQL query denotes joins by *join variables ?friend* and *?sitcom*, whereas *:Jerry*, *:hasFriend*, *:actedIn*, *:location*, *:NewYorkCity* are the fixed values, which become the *selection predicates* in the equivalent SQL query shown in Figure 1. *:Jerry*, *:NewYorkCity* are the node labels, and *:hasFriend*, *:actedIn*, *:location* are the edge labels in the original RDF graph. Due to this close resemblance between SPARQL and SQL, in the recent few years, quite a few systems have been developed specifically to handle RDF graph storage, indexing, and SPARQL query optimization using relational database techniques. A popular optimization technique is to create all six indexes on RDF graphs, since the tabular representation has just three columns (hence 3! = 6). The popular index types used are B+ trees, e.g., RDF-3X [9] uses those, and bitmaps/bitvectors used by BitMat [3–6], TripleBit [12]. Many relational databases have added support for the storage and indexing of RDF, and SPARQL query processing, e.g., Oracle, IBM DB2, Virtuoso etc.

Creating all possible six indexes on an RDF graph seems a good trick for SPARQL query optimization, however, the main challenge is the *variable length* node and edge labels in RDF graphs, which make the *search-keys* in these indexes. Variable length search keys

[1] https://www.w3.org/RDF/
[2] http://wiki.dbpedia.org/
[3] http://www.uniprot.org/
[4] https://www.w3.org/TR/rdf-sparql-query/
[5] Although the shown graph is acyclic, RDF graphs can be cyclic too.

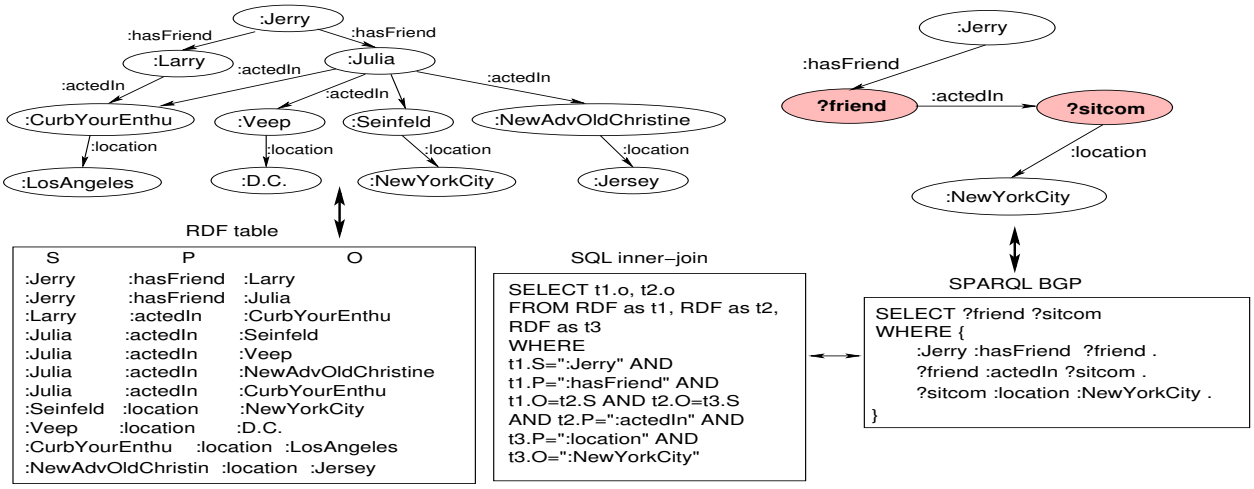

Figure 1: An example of RDF graph, its tabular representation, a SPARQL BGP query, and the equivalent SQL query

can take away the good properties of indexes like B+ trees, and they also make the index performance unpredictable. Further, in case of RDF graphs, the variation in these node/edge label lengths can be really high, e.g., in case of a DBPedia RDF graph, we observed that the longest label is 377,405 bytes, the average label length is 75 bytes, whereas the smallest one is 1 byte! Hence many of the RDF and SPARQL query processing systems first map every unique node/edge label in a given RDF graph to a fixed length unique ID (either integer or hash), and then create the six indexes on these IDs. Thus the search-keys in the indexes become fixed length IDs, and the index lookup performance can be tightly controlled.

However, this also creates a need for maintaining *forward* label → ID and *reverse* ID → label dictionaries. The forward dictionary is required to get the IDs of the *fixed positions* in the SPARQL queries, e.g., *:Jerry*, *:hasFriend*, *:actedIn*, *:location*, and *:NewYorkCity* in our example in Figure 1. That is to lookup their respective IDs in the six-way indexes used in the SPARQL query evaluation.

While the research as well as commercial SPARQL query processing systems have paid a lot of attention to the optimization of *joins* in the SPARQL BGP query, not much attention has been paid to the forward and reverse dictionaries and their performance. For instance, the BitMat system [3, 5], which has been shown to outperform other systems for the *low-selectivity* queries[6], presently does not support the forward and reverse dictionaries. It relies on the end-user to build and maintain these dictionaries. Also, it produces the SPARQL query results as a list of integer IDs instead of producing *user-legible* list of string-labels. While RDF-3X supports forward and reverse dictionaries, it was pointed out by the authors of [5] that RDF-3X's reverse dictionary lookups are much slower than its join query execution times (join query execution uses the integer IDs). A good ID-based join query execution performance certainly highlights the features of the join optimization techniques. However, performance of forward/reverse dictionaries matter in *real-life* scenarios for the "*end-to-end user experience*". Relational database based RDF/SPARQL systems like MonetDB [1, 11] or Virtuoso [2] do not have any special provision for maintaining such forward/reverse dictionaries, and hence suffer in the join query performance.

Thus, in this paper, we have focused on building efficient forward and reverse dictionaries for the RDF graphs. For this purpose we use the popular B+ tree data structure, and additionally make several performance optimizations on it. Our main contributions are as follows.

(1) Instead of string labels, we use hash IDs of these labels as the *keys* in the forward dictionary for saving storage space, and increasing the order of the B+ tree (Section 2.3).
(2) We build an *ensemble* of many B+ trees than just one B+ tree for the forward and reverse lookups, to reduce the average height of each B+ tree in the ensemble, and improve lookup times (Section 2.5).
(3) We exploit the multi-core architecture for speeding up the construction time of these dictionaries (Section 2.6).
(4) We use a novel method targeted towards the typical RDF and SPARQL workloads, for lookups in the reverse dictionaries (Section 3.4).
(5) Lastly, we also present an extensive set of experiments for the performance characterization of our optimization techniques, in isolation from any SPARQL BGP query optimization techniques (Section 4). We trust that this will enable the community to use our techniques with any other RDF/SPARQL processing system that needs label → ID and ID → label dictionaries.

2 BUILDING THE DICTIONARIES

As mentioned in Section 1, many specialized modern RDF and SPARQL processing systems create all possible six-way indexes on the RDF graphs, because these graphs are conceptually 3-dimensional – (S P O) as a triple – hence SPO, SOP, PSO, POS, OPS, OSP indexes. Before creating these six-way indexes, each unique node and edge label in the RDF graph is first mapped to a unique ID, and then

[6]Low selectivity queries are those that need to process a large amount of data, and/or may generate a large number of results.

the system uses this ID based representation of the RDF triples to create the six-way indexes. Since the IDs are fixed length and often smaller than their corresponding string-label lengths, this creates compact six-way indexes. Also the *search-key* size within these indexes remains constant – length of the ID. In the context of our example RDF graph shown in Figure 1, a sample dictionary mapping to 4-byte integers IDs is shown in Figure 2.

:CurbYourEnthu	–> 1	:Jerry	–> 7	:hasFriend	–> 12
:Julia	–> 2	:D.C.	–> 8	:actedIn	–> 13
:Larry	–> 3	:Jersey	–> 9	:location	–> 14
:NewAdvOldChristine	–> 4	:LosAngeles	–> 10		
:Seinfeld	–> 5	:NewYorkCity	–> 11		
:Veep	–> 6				

Figure 2: Labels to intID mapping for RDF graph in Fig 1

This idea removes the problem of creating six-way indexes over variable length labels, but it gives rise to the challenges of maintaining the *forward* and *reverse* dictionaries, for mapping the labels to the respective IDs and back. The forward and reverse dictionaries are required for the systems that use the ID based internal representation of RDF graph node and edge labels, because it is convenient for the user to submit SPARQL queries with string-labels, e.g., *:Jerry*, *:NewYorkCity*, *:hasFriend*, *:actedIn* etc in our example in Figure 1, as opposed to *7, 11, 12, 13* as their respective integer IDs. The systems that use internal ID based representation also produce the results of a SPARQL BGP query in the ID format. However, a user expects the results of a SPARQL query in the string-label (*user-legible*) form, and not in the ID form. Hence, reverse mapping of these IDs to their respective labels becomes imperative, and thus, these dictionaries make an important part of the RDF/SPARQL systems.

2.1 ID assignment for labels

While deciding the fixed-length IDs for mapping the labels, we have two choices – integer or hash IDs. We have chosen to use 4-byte integer IDs, as then we can map over 4 billion (2^{32}) unique labels to the 4-byte integer ID space. This provides us sufficiently large ID space. Also, in the future, if the number of unique labels in the RDF graphs grow higher than 4 billion, we can choose higher than 4-byte ID space. Within this 4-byte integer ID space, we can map labels to the integer IDs in any order. However, we choose to sort the labels in the *alphanumeric* order first – alphabetical and natural number – and then assign IDs incrementally, thus preserving the relative sorted order between two labels in their respective integer IDs too. That is, if $string_1 < string_2$ in the sort order, $intID(string_1) < intID(string_2)$. This specifically helps us in handling the *range lookups* for the SPARQL FILTER clause. For instance, consider the following SPARQL query, that asks for all the *?friends* of *:Jerry* that are *aged* under *40*.

```
SELECT ?friend
WHERE {
    :Jerry :hasFriend ?friend .
    ?friend :age ?age .
    FILTER (?age < 40) .
}
```

The alphanumeric sorting of the labels, and the sort order preserving mapping to integer IDs, allows us to do range searches by fetching the corresponding ID of "40", say $intID(40)$, which will always be greater than $intID(30)$, and smaller than $intID(50)$. For this reason, we specifically avoided IDs generated by a hash function, as the hash functions may not be order preserving. Range lookup procedure is describing in detail in Section 3.2.

2.2 Disk based B+ Tree

The sizes of the RDF graphs nowadays scale from hundreds of millions of nodes, e.g., DBPedia, UniProt graphs have about 200 million nodes, to over a billion nodes as well, e.g., Linked Open Data (LOD) cloud. Thus, for a typical configuration computer, the entire RDF graph as well as the forward and reverse dictionaries cannot be held in memory entirely. Hence these dictionaries need to be disk-based structures that can be dynamically loaded in parts as required.

These dictionaries are often built using the popular indexing structures such as B+ trees or hash-tables. In our implementation, we have used B+ trees. The salient features of a B+ tree are as given below.

- It is balanced – length of the path from root to any leaf is the same.
- It supports range lookups, whereas hash-table does not.
- The internal nodes contain only search keys and no values. Hence the size of the internal nodes remain much smaller than the leaves (as long as the search keys are not very long). If the search-keys are small enough, all the internal nodes can be cached in main-memory, thereby precluding the requirement of disk I/O to go to the next level.
- A good B+ tree design tries to maintain the size of each internal node as close to the system page-size as possible, and adjusts the *order* (number of children per node) of the B+ tree accordingly.

A good rule of thumb is *(B+ tree order ≈ page-size ÷ (size of search-keys + size-of-pointer))*. Note that here RDF labels of highly variable lengths create a problem to maintain the internal node size to one page with a fixed order. E.g., as mentioned in Section 1, DBPedia's longest label (search-key) is 377,405 bytes, whereas a typical page-size for disk I/O is 512–4098 bytes. We handle this problem with our first optimization technique as presented further.

2.3 Forward Dictionary

As noted in the previous section, the *order* of the B+ tree is a function of – (a) page-size, that is standard for computer systems, and (b) the size of the search-keys. Since RDF graphs have widely varying lengths of labels, using these labels as the *search-keys* in a B+ tree forces one to maintain a lower order and thus increases the height of the B+ tree. Varying length of labels also make it difficult to contain one internal B+ tree node in one memory page. A higher order of a B+ tree reduces its height, and in turn reduces the disk I/Os. We handle the varying length labels by computing a hash of each unique label in the RDF graph, and using this hash ID as the *search-key* in the B+ tree to creating the forward *hash(label)* → *intID* mapping.

A hash function maps variable length strings to respective fixed length IDs. The length of these ID depends on the *range of the hash function*, e.g., if the hash function generates a fixed length 64-bit hash ID, the range of the hash function is 2^{64}. In our implementation, we have used a 64-bit hash function. Although a 64-bit hash function gives a large enough range (2^{64}), hash functions can be sensitive to the skew in the data. If most input strings are very similar to each other, there is a possibility of two strings being hashed to the same hash-ID, and hence causing a collision. It has been observed that RDF graph labels indeed can have a heavy skew. Hence in order to avoid the possibility of a collision for a very large set of labels, we use *two different 64-bit hash functions* to generate two hashes, $hash_1$ and $hash_2$, of the same label, and insert a composite key $\langle hash_1(label), hash_2(label)\rangle$ of 128 bits (16 bytes) and associated *intID* in the forward mapping B+ tree. Our hash functions are given below[7].

Algorithm 1: `two-hash-functions`

```
   input  : hash-alg, str[] // string arr
   output : hash-key
   // Init hash key to prime num
1  hash = 5381;
2  if hash-alg == 1 then
       // ≪ is left-bit-shift
3      for each char c in str[] do
4          hash = ((hash ≪ 5) + hash) + c;
5  else if hash-alg == 2 then
6      for each char c in str[] do
7          hash = c + (hash ≪ 6) + (hash ≪ 16) - hash;
8  return hash;
```

Using a 128-bit (16 bytes) search key made of the two different hash functions gives us the following advantages:

(1) Our search-key size in the B+ tree for the forward dictionary remains fixed at 16 bytes, thereby giving us an advantage of choosing a higher order of the B+ tree and reduce its height. Assuming a standard page size of 4 Kilobytes, and child pointer of 8 bytes, up to $4 * 1024/(16 + 8) \sim 170$ keys can be placed in one memory page, which is also the internal node of the B+ tree. In our experiments we have kept the order to be 150, although it can be changed to give flexibility of using another search-key of different length.
(2) Two different 64-bits hash functions in all give us a 128-bit search key space, which is large enough range to thwart any collisions for very large RDF graphs (in our experiments on hashing over 126 *million* unique labels of real life DBPedia RDF graph, we did not encounter any collisions).
(3) As noted in Section 1, an average label size in the DBPedia RDF graph is 75 bytes, and the longest label is 377,405 bytes. Whereas our two hashes make only 16 byte long search key. Thus using a composite hash search key of 16 bytes reduces the size of the search key by more than 75%.
(4) In turn the size of the internal nodes in the B+ tree of the forward dictionary remains small, allowing the system to cache these internal node pages in main memory, and precluding disk I/O for them[8].
(5) The storage space required for the B+ tree, insertion time, and lookup time no longer depend on the label sizes.

2.4 Reverse Dictionary

For the reverse dictionary, we need to do $intID \rightarrow label$ mapping. Since the search-key (4-byte integer ID) in this B+ tree is fixed, we use the integer IDs as-is to build the reverse dictionary. However, instead of storing the labels in the leaf nodes, we use a following simpler way. Recall from Section 2.1, that we first sort all the labels alphanumerically and then sequentially assign integer IDs to them, so that the sort order of *intIDs* is same as their respective labels' sort order. In practice, for the simplicity of the implementation, we maintain a flat text-file of all the sorted labels with one label on each line of the file, and use the line number as the 4-byte *intID* of that label. Thus, in the leaf-nodes of the B+ tree for the reverse dictionary, instead of having labels, we just maintain an offset of the starting position of the label within this flat-file, and retrieve that label by reading the line from the flatfile. For the offset we use 8 bytes (on 64-bit OS). This helps us to save the storage space of the reverse dictionary by removing the need to store the variable length labels in the leaf nodes. In essence, our B+ tree of reverse dictionary has mapping as $intID \rightarrow offset$.

In summary, the forward and reverse dictionaries are built as follows.

(1) Sort all the labels in an alphanumeric order – alphabetical and numerical – and store in a flat text-file with one label on each line.
(2) Allocate the line number of the label in the flat-file as the 4-byte integer ID, thus making intIDs order preserving, $label_1 < label_2$ implies $intID(label_1) < intID(label_2)$.
(3) For each string label, generate $hash_1(label)$, $hash_2(label)$ (see Algorithm 1).
(4) Insert $\langle hash_1(label), hash_2(label)\rangle \rightarrow intID(label)$ key-value pair in the forward B+ tree.
(5) Simultaneously add the corresponding $intID(label) \rightarrow$ *offset(label)* mapping in the reverse B+ tree.

2.5 Ensemble of B+ trees

The height of the forward and reverse B+ trees is $log_B N - 1$ where B is the *order* of the tree, and N is the total number of unique labels in the graph. The lookup for a key consists of traversing the height of the tree from the root to the leaf node, and these are typically the number of I/O operations, assuming that none of the components of the B+ tree are cached in the main memory. Thus, for a large number of labels, the height of the tree is non-negligible. E.g., for 126 million labels used in our experiments (see Section 4) the height of the tree is 3 for order 170. For a SPARQL query processor with high frequency of SPARQL queries (e.g., DBPedia SPARQL endpoint on the web[9]), these lookups can be a major performance bottleneck.

[7] Note that we have not focused on hash function optimization, and that will be part of our future work.

[8] For the scope of this paper, we have not focused on cache optimization techniques.

[9] https://dbpedia.org/sparql

Reducing the height of the B+ tree by even one level can give us significant benefits. This reduction factor particularly becomes significant when we have to do several bulk reverse lookups for the *intID* format results generated of a SPARQL BGP query. This is elaborated in Sections 3.4 and 4.4. Hence to reduce the overall height of one tree, we *split* it into multiple trees, and create an *"ensemble"* of B+ trees. For instance, if we create an ensemble of 1000 B+ trees over 200 million labels, such that each smaller B+ tree consists of 200,000 labels, the overall height of each tree in the ensemble becomes 2 instead of 3.

Recall from Section 2.1 that for intID assignment to each label, we first sort all the labels alphanumerically. For creating an ensemble of B+ trees, we split this sorted list of labels into contiguous chunks of almost equal number of strings, and number of such chunks is equal to the ensemble size. E.g., for an ensemble of size 1000 over 200 million labels, each chunk consists of 200,000 labels (200 million ÷ 1000). We build a separate B+ tree of each of these 1000 chunks. Additionally, we note down the first label in the sorted order of each chunk. These strings are stored as a *metadata* of the ensemble in a flat textfile. The significance of these strings is elaborated when we discuss the forward lookup procedure in Section 3.1.

2.6 Parallel Construction

Notably each of the ensemble B+ trees can be built independently and parallelly by running *multiple threads* and exploiting the multi-core architecture of the modern computer systems, where each thread constructs one smaller B+ tree. This further reduces the construction time for the forward and reverse dictionaries, as shown in our experiments in Section 4.4.

The size of an ensemble depend on the following factors, and may change from one dataset to another.

(1) The total number of labels, and in turn the height of the single B+ tree containing them.
(2) The average length of the labels – as mentioned above, we keep a log of the first string of each tree in the ensemble, and we keep them loaded in main memory for aiding fast lookups in the ensemble. Hence the additional main memory required to maintain this metadata is (*avg-length(labels)* * *size-of-ensemble*).

In our experiments with over 126 million labels of the real life DBPedia RDF graph, we have experimented with ensemble sizes ranging from 1 to 9561. The DBPedia graph's average label length is 75 bytes, hence we need just 700 Kilobytes of additional main memory to keep track of the first strings of each smaller B+ tree in the ensemble. While this additional memory requirement is negligible, the height of each B+ tree in the ensemble reduces by one level compared to one giant B+ tree containing all the labels. Note however that if we make the size of the ensemble too large, the whole B+ tree structure essentially collapses down to become a flat hash-table (since we maintain $\langle hash_1, hash_2 \rangle$ of the labels as the search-keys). Also, for very large size of the ensemble, we require to keep the respective *first strings* of each smaller B+ tree in the ensemble loaded in memory for deciding which smaller tree to lookup. This in turn increases the runtime main-memory requirement. Doing a binary search within a large number of these first strings of ensemble takes away the advantage of the ensemble, and small tree heights in it.

3 LOOKUPS

In this section we discuss the details of the procedures of forward and reverse dictionary lookups.

3.1 Forward Lookups

Recall from Section 2.5, that instead of one big B+ tree containing all the $\langle hash_1(label), hash_2(label) \rangle \rightarrow intID(label)$ mappings, we create an *ensemble* of B+ trees, note the first sorted label from the list of labels allotted to each smaller B+ tree, and keep it as the metadata. We keep these list of first-labels loaded in main-memory. These labels allow us to decide quickly which smaller B+ tree to lookup for a given lookup query. Let us assume that our ensemble is of size 1000. Then in main memory we maintain the first label of the 1000 splits of the original sorted labels. When we need to do a lookup in the ensemble, first we do a binary search over these 1000 first-labels in main-memory to decide the smaller B+ tree for the lookup, and then do a search inside the respective smaller B+ tree. Let us assume that the search label is $\mathcal{L}$. We first find two labels, $label_i$ and $label_j$ from the list of 1000 first-labels, such that $label_i \leqslant \mathcal{L} < label_j$, where i, j are the respective smaller B+ trees. Then we lookup into the i^{th} B+ tree by forming a search key $\langle hash_1(\mathcal{L}), hash_2(\mathcal{L}) \rangle$, and follow the standard B+ tree lookup procedure to retrieve $intID(\mathcal{L})$.

3.2 Range Lookups

Recall that we chose B+ trees over hash-tables to build the dictionaries in order to support range-queries. We also ensured that the *intID* assignment to the labels is sort-order preserving. A curious reader may wonder that when labels are converted to two hash-keys $\langle hash_1(label), hash_2(label) \rangle$, and they are used as the search-keys in the B+ tree of the forward dictionary, how would the range queries be answered? We illustrate that with the following example SPARQL query.

```
SELECT ?friend
WHERE {
    :Jerry :hasFriend ?friend .
    ?friend :age ?age .
    FILTER(?age > 20 && ?age < 40) .
}
```

This query asks for all the friends of *:Jerry* that are aged between 20 and 40. For such a query, we first get the corresponding search-keys of "20" and "40" as $\langle hash_1(20), hash_2(20) \rangle$, let us call it $key(20)$, and $\langle hash_1(40), hash_2(40) \rangle$, let us call it $key(40)$ respectively. We fetch the corresponding $intID(20)$ and $intID(40)$ using $key(20)$ and $key(40)$ from the forward dictionary. Since our *intID* assignment is sort-order preserving, this means that all the intIDs between $intID(20)$ and $intID(40)$ are strictly for labels between "20" and "40", and thus they are used in the SPARQL query processing for satisfying the FILTER condition.

3.3 Reverse Lookups

Recall from Section 2.4 that in the reverse dictionary, we create a mapping of *intID(label)* → *offset(label)*, where *offset* is the offset of the *label* within the flatfile containing all the labels in an alphanumerically sorted order, a unique label on each line. Lookup procedure in the reverse ensemble of B+ trees is the same as forward lookup procedure as elaborated in Section 3.1. In main memory, we maintain the first *intID* of each smaller B+ tree in the ensemble to do quick binary search, and decide the smaller B+ tree to search into. This additional main memory required for reverse lookups is negligible, e.g., 4 Kilobytes for *intIDs* of size 4-bytes with an ensemble size of 1000. Once the *offset* corresponding to an *intID* is retrieved from the B+ tree, we fetch the label from that offset in the flatfile containing alphanumerically sorted labels. Notice here that as described in Section 2.1, our *intID* assignment to the labels is label sort-order preserving. In turn, the offsets of these strings within the flatfile are order preserving too, i.e., $intID(label_1) < intID(label_2) \Rightarrow offset(label_1) < offset(label_2)$. We make use of this observation, and propose an optimization to the reverse lookups as given in the following section.

3.4 Bulk Reverse Lookups

Recall from Section 1 and 2, that a forward $\langle hash_1(label), hash_2(label)\rangle \rightarrow intID(label)$ dictionary is used to convert the fixed labels in a SPARQL BGP query into their respective *intIDs*. Even if we assume a very large complex SPARQL BGP query, the number of labels in such a query still remains quite small, e.g., 5-10 labels. However, the reverse $intID(label) \rightarrow offset(label)$ dictionary is required to map the results of a SPARQL BGP query in the *intID* form, back to their respective labels. These can vary from 10s of results to even a *million* results, depending on the *selectivity* of the query[10]. Thus *unlike* the forward dictionary lookups, the reverse dictionary lookups can be several hundreds of them, and hence can have a high impact on the "*end-to-end*" query execution time. Referring back to our points mentioned in Section 1, owing to these performance intensive reverse dictionary lookups, BitMat [5] system in its experiments has only reported BGP query execution and result generation times over *intIDs* alone, and not in the *end user legible* labels form. Also the authors of the BitMat system have reported that RDF-3X's reverse dictionary lookups take much longer than BGP query execution times.

Noting these challenges, we propose a novel way of doing *bulk* lookups for each SPARQL query results. Before describing our bulk lookup method, we make some observations about the characteristics of a SPARQL BGP query results, which make the foundation of our bulk lookup method.

Each result of a SPARQL BGP query consists of a list of *intIDs*, which are the *bindings* of the join variables in the query. E.g., the query given in Figure 1, along with the forward dictionary mapping given in Figure 2, generates one result *{2, 5}*, where 2 and 5 are the *intIDs* assigned to *:Julia* and *:Seinfeld* labels. Thus, *:Julia* and *:Seinfeld* are the bindings for the join variables *?friend* and *?sitcom* in the BGP query. Hence to answer this query in the *user-legible* form, we need to do a reverse mapping of 2 → *:Julia* and 5 → *:Seinfeld*.

When we have a complex BGP query that generates many results, some (or many) of the results share the same variable binding *intIDs*. E.g., if a SPARQL BGP query generates three results as *{1001, 3456}*, *{1001, 2345}*, *{1001, 1234}*, then 1001 value is shared among all the three results, and the total unique *intIDs* to look up are four instead of six. While mapping the *intIDs* to their respective labels, a naïve way is to lookup the reverse ensemble of B+ tree for each generated *intID*, in the same sequence in which they are generated by the BGP query processor. However, as we can be readily notice, this requires *redundant* lookups of the intIDs that are common among multiple results. E.g., in the above example, intID 1001 is looked up thrice – once for each set of intIDs. When the query is complex, and over a very large RDF graph, many results share common intIDs. Thus in order to avoid redundant lookups of intIDs shared across multiple results, we propose the following optimization method.

Instead of searching the reverse mapping for every intID, we first collect *all* the results of a SPARQL BGP query in the intID form, and store them in a flatfile, and sort all the intIDs numerically. E.g., in case of our example above, we first store the three results *{1001, 3456}*, *{1001, 2345}*, *{1001, 1234}*, and then create a sorted list of intIDs as *[1001, 1234, 2345, 3456]*. Then we lookup the reverse ensemble B+ trees for each of these sorted intIDs. It is important to note that with this we make use of a beneficial feature of the B+ trees – *if the subsequent search-keys of a B+ tree are all sorted and known ahead of time, then we can simply do a sequential scan on the leaves of the B+ trees.* As mentioned in Section 3.3, the retrieved corresponding offsets of the intIDs are all in the sorted order too. Thus essentially retrieving the labels from the offset positions is nothing but doing a single sequential scan on this flatfile. This technique avoids random disk (memory) accesses, and reduces the I/O overheads to a large extent. The benefits of this technique are clearly visible in our experimental results shown in Table 4 in Section 4.4.

We cache these retrieved labels in main-memory. Once all the labels are retrieved, we make a single pass over the original SPARQL BGP query results in the *intID* form, that are stored separately (either in memory or in a flatfile), and generate the respective BGP query results in the user-legible label format. SPARQL BGP query results can range from few hundred to even a few million, which means we may have to lookup about a million *intIDs* and cache their respective labels in memory. In our observation, if labels are not very long, they can be cached in main-memory even on a commodity computer. E.g., for the DBPedia RDF graph, average label length is 75 bytes. Thus a even up to a million labels consume only about 72 MB memory.

Additionally, when the bulk lookup *intIDs* are all in a sorted order, we can exploit the multi-core architecture, by splitting the lookups across multiple threads, and further reducing the lookup times. For the scope of this paper, we have only used single-threaded implementation for the sorted bulk reverse lookups. Multi-core parallel bulk lookups is part of our future work.

[10] Queries that generate small number of results are considered to be more selective than those which generate a large number of results.

4 EXPERIMENTS

In this section we present our experiments to test and characterize the performance of our forward and reverse dictionaries and optimization methods.

4.1 Environment

We ran all of our experiments on an Intel Core i7 octa core 2.8 GHz processor with 32 GB RAM, 32 GB swap space, and a 2 TB Western Digital SATA HDD. Our code was developed using C/C++ as the programming language, and compiled using gcc and g++ version 5.4.0 on a 64 bit 4.4.0-91-generic Linux kernel with Ubuntu 16.04.1 LTS distribution.

4.2 Dataset

For the experiments, we used a *real-life* DBPedia RDF graph[11], and extracted 126,444,964 unique node labels from it, for building our forward and reverse dictionaries. Recall from Section 2.1 that we first sort all the labels in an alphanumeric order and maintain a flatfile of these sorted labels with a unique label on each line. The on-disk size of this flatfile is 9 GB.

4.3 Performance Metrics

For the evaluation of our optimization techniques, we used the following metrics:

(1) Construction time for the forward and reverse dictionaries together, across varying sizes of ensembles, as well as varying the amount of parallelization in the construction, i.e., multi-core construction.
(2) Time taken for the forward and reverse lookup across varying number of queries, as well as varying sizes of the ensembles.
(3) Time taken for *sorted bulk reverse lookups* (refer to Section 3.4), as well as the respective unsorted random lookups across varying number of queries.

Tables 1, 2, 3, and 4 summarize our experiments using the above evaluation metrics, and in the next section, we discuss them in detail.

4.4 Discussion

For the forward dictionary we set the order of the B+ trees in the ensemble to be 150 taking into account the 16 byte key size and 8 byte pointer (to the child node), and similarly set the order to be 220 for the reverse dictionary, with 4 byte *intID* search key and 8 byte pointer. The order of the forward or reverse B+ trees can be changed from one dataset to another depending on the requirements such as the total number of labels and size of the search key. We vary the size of the ensemble from just one B+ tree – a single giant B+ tree containing all the search-keys – to 9561 smaller B+ trees. Table 1 shows our total construction time for the forward and reverse dictionaries, over varying sizes of the ensembles, using a single core (single thread) implementation.

Recall from Section 2.5 that creating an ensemble aids in reducing the height of the each smaller B+ tree in the ensemble. Thus, it is evident from Table 1 that an ensemble of size 9561 reduces the height of the tree to just 1 as opposed to an ensemble of size 1. Also the construction time reduces as the size of the ensemble increases, because while inserting a new key-value pair in each smaller B+ tree, the height of the tree to be traversed remains small. Hence it can be noted that the construction time from ensemble of size 1 to size 9561 reduces by about 75% (222 minutes for ensemble size 1 versus 53 minutes for ensemble size 9561), with just a single-threaded (single core) implementation[12].

Ensemble size	Height	Constr time (minutes)
1	3	222
10	3	117
96	2	73
956	2	65
9561	1	53

Table 1: Construction time, B+ tree height, of forward and reverse dictionaries together (single thread)

#Threads	Constr time (minutes)
1	65
2	38
4	22
8	15

Table 2: Parallel construction of an ensemble of size 956

Creating an ensemble also allows us to parallelize the construction of each smaller B+ tree, using multiple threads and exploiting the multi-core hardware architecture. Hence we conducted separate experiments by keeping the size of the ensemble the same, and gradually increasing the parallelization factor from using just a single thread, to eight threads. Table 2 shows that construction time for an ensemble of size 956 reduces by about 75% when we use eight threads instead of just one thread.

Next, to test the lookup performance of our dictionaries, we ran varying number of *bulk lookups* across varying sizes of the ensembles of forward and reverse dictionaries (refer to Section 3.4 for our bulk reverse lookup procedure). Table 3 summarizes the results of our experiments. We setup these experiments as follows. To run 100,000 bulk reverse lookups, we first randomly generated 100,000 *intIDs* in the range of 1–126,444,964 (total labels in DBPedia). Then we sorted them, and performed bulk reverse lookups. For the scope of this paper, we ran all the bulk reverse lookups using only a single-threaded (single core) implementation. Through these bulk reverse lookups, we retrieved the respective *offsets* of the labels associated with the *intIDs*, and fetched those respective 100,000 labels from the flatfile, and stored them separately to be used in our forward lookup experiments as described further.

To account for the CPU scheduling and system time, we generated *five different* sets of 100,000 *intIDs*, ran each set of lookups five times, and then took an average over all the five sets of bulk lookups and their respective five runs. While reporting these times,

[11] http://wiki.dbpedia.org/news/dbpedia-based-rdf-dumps-wikidata

[12] For these experiments we did not use the well-known B+ tree *bulk loading* method.

Ensemble Size \ # Bulk Lookups	100,000		1,000,000		10,000,000	
	Forward	Reverse	Forward	Reverse	Forward	Reverse
1	0.79	0.58	7.88	5.39	77.41	53.18
10	0.76	0.73	7.55	6.80	74.97	64.31
96	0.58	0.46	5.78	4.20	56.92	40.50
956	0.57	0.44	5.37	4.08	52.32	40.08
9561	0.44	0.45	3.93	3.91	38.14	31.62

Table 3: Forward and reverse lookup time (seconds) using single-thread across varying sizes of ensembles for varying number of queries. Times averaged over multiple query sets and multiple runs, please see the text for the details.

we have only considered the B+ tree ensemble lookup times, and did not count the lookup time into the alphanumerically sorted flatfile of all the labels. We repeated the same procedure for the 1,000,000 and 10,000,000 lookups. For the 100,000 forward lookups, we used the 100,000 labels retrieved through bulk reverse lookups that we stored separately (as described above). Similar to the reverse lookups, we ran the lookups five times, and then took an average of these lookup times. We repeated similar experiments for 1,000,000 and 10,000,000 forward lookups.

#Lookups	Sorted Bulk	Random
100,000	0.44	0.65
1,000,000	4.08	6.50
10,000,000	40.08	64.30

Table 4: Bulk sorted reverse lookup versus corresponding random reverse lookup times (seconds), ensemble size 956, single-threaded

The effect of our optimizations of creating an ensemble is clearly visible by seeing the decrease in the amount of time required to perform bulk lookups, e.g., lookup time for 100,000 reverse lookups is 0.58 sec with an ensemble of size 1, whereas with an ensemble of size 9561, this lookup time has decreased to 0.45 sec. It is important to also note that for a smaller number of lookups the difference in the times across varying sizes of ensemble is small. However, as the number of lookups increase, this difference in the time from a smaller to larger sized ensemble is significant, e.g., for 10,000,000 lookups, with an ensemble of size 1, the lookups take 53.18 sec, whereas with an ensemble of 9561, the lookup time is reduced to 31.62 sec. Thus our optimization method of ensemble creation can fetch significant benefits for *low-selectivity* queries that generate a large number of results (and hence a large number of reverse lookups).

Next, we performed experiments to measure the effectiveness of our optimization technique of *bulk sorted reverse lookups* (see Section 3.4), by varying the number of lookups, and keeping the size of the ensemble the same. Using the same procedure as described before, we generated 100,000 *intIDs* randomly in the range of 1–126,444,964. However, for this experiment, instead of sorting all the *intIDs* before bulk lookups, we performed lookups using the same random order in which the intIDs are generated. Table 4 highlights the benefits of our bulk reverse lookup methods over random lookups. It is important to note here as well, that the difference between the sorted bulk and random lookups for a small number of lookups, say 100,000, is smaller. For a large number of lookups, say 10,000,000, the difference is significant, 64.30 sec for random lookups whereas 40.08 sec for sorted bulk reverse.

Recall from Section 2.3 that we use $\langle hash_1(label), hash_2(label)\rangle$ as the search-key inside the forward dictionary, instead of using the very *label*. We did this optimization, to (1) maintain the same search-key size and, (2) to reduce the search key size, and in turn the forward dictionary size. For the DBPedia RDF graph, the average size of the label is 75 bytes, whereas the hash-based search-key is 16 bytes. In order to evaluate the benefits of this optimization, we do the simple math for the DBPedia's 126,444,964 labels. With 16 byte search-key, 8 byte pointer for the child, the order of this B+ tree can be set to 150. Absolute storage space required for this B+ tree's internal and leaf nodes will be about 2.5 GB. Whereas a forward dictionary built with labels as the search key will have an order of 45 (recall our formula of deciding the order from Section 2.2). With 75 bytes as the average search key size, 8 bytes for pointer, and order 45, this B+ tree will consume approximately 9.6 GB.

We summarize our experimental results, and observations as follows.

(1) The experimental results in Table 1 show that our optimization technique of creating an *ensemble* of B+ trees for forward and reverse dictionaries is beneficial for reducing the *overall height of the ensemble*, as well as reducing the single-threaded construction time.
(2) Results in Table 2 highlight the benefits of our technique of multi-core construction of the ensemble.
(3) With the results in Table 3, we see that creation of an ensemble of B+ trees significantly reduces the bulk lookup times, especially as the number of lookups grow higher. This is specifically beneficial for *low-selectivity* SPARQL BGP queries, that generate a large number of results, and in turn require a large number of reverse lookups.
(4) Finally, with the results in Table 4, we see that our technique of *sorted bulk reverse* lookups, specifically targeted at SPARQL BGP queries, significantly reduces the reverse lookup times, over random lookups of the same *intIDs*. Thus

sorted bulk reverse lookup technique can improve the *end-to-end* user experience, by reducing the overall SPARQL query processing times (please refer to Section 1).

5 RELATED WORK

B+ trees have been one of the most popular and widely used data structures in the research community. B trees are similar to B+ trees where the values are stored along with the search-keys in the internal nodes of the tree. Among the SPARQL query processing systems, BitMat [5] does not support dictionaries. RDF-3X [9] and TripleBit [12] support dictionaries. However, as pointed out in Section 1, the performance of their dictionaries has not been optimized, and that is the focus of our paper, to improve the *end-to-end* user experience of RDF and SPARQL query processors.

In [10], Nguyen et al have proposed a combination of B+ trees and hash-tables for indexing the RDF graphs. The focus of that work is on the indexing of the RDF graphs, whereas in our work, we have predominantly focused on optimizations for maintaining the forward and reverse dictionaries.

Work by Martínez-Prieto et al [8] comes closest to ours. In their work, they have focused on the compression techniques for the RDF dictionaries using their $\mathcal{D}_{comp}$ method. The sizes of the datasets used in their dictionaries range from 5 million to 67 million labels. We have performed our experiments on a set of 126 million DBPedia labels. Also notably, the compressed size of their largest dataset of 67 million labels, is 6.96 GB, and our forward and reverse dictionaries over 126 million labels are sized at 10 GB (including any metafiles). Additionally, in our work, we have proposed specific optimization techniques like ensemble B+ trees, and parallel construction using the multi-core architecture.

6 FUTURE WORK AND CONCLUSION

In this paper, we presented several optimization techniques to build efficient dictionaries for the RDF graphs data. Our techniques include (1) ensemble of B+ trees, (2) converting string labels to hash-based search keys, (3) parallel construction of the ensemble using multi-core hardware architecture, and (4) optimized bulk reverse lookups.

As a part of the future work, we would like to build a robust hash-function to thwart possible collisions while indexing very large RDF graph's labels. In this work, we have not focused on any proactive *page-caching* strategies to avoid the disk I/Os. In our present work, we relied on file system caches, and warm cache benefits. In the future, we plan to specifically look at the active caching strategies.

To conclude, in this paper we have presented several new optimization strategies focusing on building efficient RDF dictionaries to improve the *end-to-end* user experience of SPARQL query processing. Our extensive set of experiments show that our optimization strategies show significant benefits over naïve methods, especially for the RDF/SPARQL specific workloads. Our proposed optimization techniques are independent of any underlying RDF graph indexing methods, and hence can be used by any system that requires label → ID and reverse mapping dictionaries.

REFERENCES

[1] [n. d.]. MonetDB. http://www.monetdb.org/. ([n. d.]).
[2] [n. d.]. Virtuoso Opensource Edition. http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/. ([n. d.]).
[3] Medha Atre. 2015. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *SIGMOD*.
[4] Medha Atre. 2016. For the DISTINCT clause of SPARQL queries. In *WWW (posters and demos)*.
[5] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *WWW*.
[6] Medha Atre, Jagannathan Srinivasan, and James A. Hendler. 2008. BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Querie. In *ISWC (posters and demos)*.
[7] Richard Cyganiak. 2005. A Relational Algebra for SPARQL. *Technical Report, HP Laboratories Bristol* HPL-2005-170 (2005).
[8] Miguel A. Martínez-Prieto, Javier D. Fernández, and Rodrigo Cánovas. 2012. Querying RDF Dictionaries in Compressed Space. *SIGAPP Appl. Comput. Rev.* 12, 2 (June 2012).
[9] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *SIGMOD*.
[10] Minh Khoa Nguyen, Cosmin Basca, and Abraham Bernstein. 2010. B+Hash Tree: Optimizing query execution times for on-Disk Semantic Web data structures. In *SSWS workshop at International Semantic Web Conference*.
[11] Lefteris Sidirourgos et al. 2008. Column-store support for RDF data management: not all swans are white. In *PVLDB*.
[12] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: A Fast and Compact System for Large Scale RDF Data. In *PVLDB*.