Semantic Web 1 (0) 1–4
IOS Press

A dynamic HDT variant

Eirik Kultorp [a], Wouter Beek [a] and Stefan Schlobach [b]
[a] *Triply, Amsterdam, The Netherlands*
E-mails: eirik.kultorp@triply.cc, wouter@triply.cc
[b] *Department of Computer Science, VU Amsterdam*
E-mail: k.s.schlobach@vu.nl

Editors: First Editor, University or Company name, Country; Second Editor, University or Company name, Country
Solicited reviews: First Solicited Reviewer, University or Company name, Country; Second Solicited Reviewer, University or Company name, Country
Open reviews: First Open Reviewer, University or Company name, Country; Second Open Reviewer, University or Company name, Country

Abstract.
HDT is a compact yet queryable storage format for RDF data. HDT scales to very large datasets and is used in production environments. While it's possible to merge two HDT files, it is not possible to perform removals. This prevents use of HDT for some use-cases which require dynamism. In this paper we propose and evaluate a dynamic HDT variant. Our results show that we achieve performant update operations without a prohibitively high cost for other operations, making our implementation a viable alternative to the existing HDT implementation if dynamism is required.

Keywords: HDT, Dynamic data structures, Linked Data tools

1. Introduction

HDT is a compact queryable format for RDF data [1]. Open-source tools provide functionality for creating HDTs, merging HDTs [2], querying HDTs, and producing plaintext RDF files from HDTs. Combined, these functionalities satisfy the requirements for many use-cases related to storage and distribution of linked data. HDT is a mature project and is used as a storage backend, e.g. by Triply[1].

A limitation of the HDT format is that HDT files are practically static. While it is possible to merge HDTs, it is not possible to remove data. As a consequence, modifying a dataset stored as HDT can usually only be achieved by regenerating the HDT from scratch. This is problematic or even prohibitive for some use-cases, such as fixing small errors in large datasets, incrementally updating a dataset from a dynamic source, or maintaining a merge of an ever-changing set of HDT-files.

A solution to these problems is a dynamic variant of the HDT format. By *dynamic*, we mean a variant which supports applying a diff to an HDT with a time complexity where the size of the diff is the most significant factor, rather than the size of the HDT which we apply the diff to, and where the functionality supported by the static HDT variant is still supported with comparable time complexity.

We make use of prefix trees (tries) as a replacement for HDT's Plain-Front Coded (PFC) dictionaries. While the static variant uses a 'shared' dictionary for terms occurring both in the subject and object positions, this is not

[1] www.triply.cc

feasible for a dynamic variant. We drop this shared dictionary, which allows a fully lexicographic triples-section. In turn, this simplifies and speeds up some procedures. Because the term identifiers of our dictionary correspond to insertion order rather than lexicographic order, we have more entropy than the static variant in the lexicographically sorted triples section. This significantly reduces the benefit of compaction, so we don't apply compaction for the triples section.

We wish to know whether our dynamic variant is a feasible alternative to the static variant. We can decompose this into four research questions, which ask about the cost for existing operations and benefits of the new operations. We ask the following research questions, which we will seek to answer empirically.

1. For features present in both variants, does runtime of the dynamic variant stay within an order of magnitude of the static variant?
2. For features present in both variants, does peak memory usage of the dynamic variant stay within an order of magnitude of the static variant?
3. For new update features, does runtime of the dynamic variant improve over the best workaround for the static variant?
4. For new update features, does peak memory usage of the dynamic variant improve over the best workaround for the static variant?

We evaluate our implementation by running a number of experiments where we compare our dynamic variant against the existing static variant. For operations that are unsupported for the static variant, we compare against a work-around. We find that our implementation is performant compared to the static variant for update operations, both in terms of memory usage and runtime. An exception is the HDT merging operation, where the static variant uses less memory. For existing functionalities, we demonstrate a generally similar performance in terms of runtime, but with a cost to memory usage. We attribute the increased memory costs to the dynamic variant's significantly larger HDT files, caused by introduction of an explicit data structure as well as the dropping of the shared dictionary and compaction in the triples section. The observed cost to memory usage will often be acceptable, because the memory usage of the most memory-intensive operation (initial generation) is lower for the dynamic variant.

We conclude by answering 'yes' to all our research questions, conditional on the size of the modification for the update operations. Our dynamic variant is a significant contribution to the HDT community, as it enables previously unsupported use-cases at cost of increased disk usage and memory usage for some operations. Deployment to Linked Data platforms such as TriplyDB could lead to cheaper ETL pipelines and faster platform-wide searches. Additionally, data quality might increase over time, as the incremental update functionality could be used for supporting small updates.

We assume the reader is already familiar with the HDT format. Otherwise, we recommend reviewing the original paper on HDT [1], the homepage of the HDT project [3], and the paper which introduced HDT-Cat [2]. In this paper we identify a route to a dynamic HDT format, which involves writing new variants for two components of the HDT code, the dictionary and the triples components. We design and implement these components and proceed to run experiments and present the results, which we explain in light of the natures of the static and dynamic variants. We conclude in light of our experimental results that our design allows for performant update operations at an acceptable cost to other existing functionality.

2. Problem analysis

We dissect our problem in more detail, formulating requirements and identifying issues which a design of a dynamic HDT variant must consider.

2.1. Requirements of a dynamic HDT

A dynamic HDT variant should support all the same operations as those of static HDTs. This naturally includes generation of the HDT from a RDF source (RDF-to-HDT) and ability to convert back to RDF (HDT-to-RDF).

Additionally, triple-pattern searches and prefix search over terms should be supported. Runtime and peak memory consumption should not be more than an order of magnitude greater than for the static variant for any of these operations. Further, the lexicographic ordering of terms and triples emitted by search operations and the HDT-to-RDF operation should be at least as well lexicographically ordered as in the static variant, as this is a property which is often useful to other applications using HDTs.

In addition to preserving existing functionality, a dynamic HDT variant should support adding and removing triples. This should be possible through HDT inputs as well as RDF inputs. The time it takes to modify a HDT file should be less than the time it would take to generate a modified RDF file and generate a new HDT file from scratch with the static variant.

2.2. Dynamicity of data structures and files

Consider an inherently dynamic data structure stored in a file. One may load the file and perform update operations on the data structure. Persisting the modified data structure to disk would ideally only involve writing an amount of data proportional to the size of the data structure update. This may be trivial if the modification only involves adding data to the end of the file or replacing old values with new values of the same size. In the more general case, where data of any size may be inserted to or removed from anywhere in the binary representation of a data structure, it becomes a more complex task to minimize the amount of disk writing. While file dynamicity is desirable, it would place restrictions on the encoded data structures.

2.3. Dictionary ID range properties

In the static HDT, the IDs of terms occurring in each triple-position form a continuous range starting at 1. In a dynamic HDT variant, this property would not come for free. If a term is removed, there would either be a hole in the range, or all IDs in the triples section greater than the ID of the removed term would need to be updated. The latter option would be expensive, while the former would be acceptable unless it is explicitly required by the Triples section, as it is for the static version.

In the static HDT, the IDs of terms occurring in each triple-position partially correspond to the lexicographic ordering of the dereferenced terms. Guaranteeing this property would likely be costly for a dynamic variant because insertions across update operations can't be guaranteed to be lexicographic. Maintaining lexicographic correspondence would therefore require updating the IDs of all terms occurring lexicographically after the inserted term. By consequence, all occurrences of the ID in the triples section would also need to be updated. The alternative is to not guarantee this property. This would make sorting operations over lists of ID-triples more expensive, as the comparison of two ID-triples would involve dereferencing the IDs to terms in the dictionary.

2.4. Use of 'shared' dictionary

The static HDT dictionary makes use of a 'shared' sub-dictionary consisting of terms found both in the subject and object positions to reduce data duplication. Making use of this technique would complicate the design of a dynamic dictionary. When a term is moved from the subjects-only or objects-only sub-dictionary to the shared sub-dictionary, the ID may already be occupied by a different term. This would require assigning a new ID to the term and updating all ID-triples in which the term occurs. The chance of conflicts could initially be reduced by reserving half of the ID domain to each position, but this reduces the supported maximum size of a dataset. Further, if a term is initially in e.g. the 'subjects' sub-dictionary and is then moved to the 'shared' sub-dictionary and finally to the 'objects' sub-dictionary, collisions would still be possible although infrequent. Another issue is that use of the 'shared' sub-dictionary may complicate or make infeasible lexicographic iteration over terms and triples, as is the case with the static variant.

2.5. Counters for terms and triples

The static HDT variant's terms and triples are sets. Adding the same triple twice will yield identical data structures. This is fine as long as data is never removed, which is not supported in the static variant. In a dynamic variant

however, this is not acceptable. Terms may occur in multiple triples, and removing only one of the triples must not cause removal of a term which is still used by a different triple. For triples, consider a merge of two HDT files, where the two HDT files partially overlap. If one then subtracts one of the HDT files, the triples which occurred in both source HDT files should not be removed. So, it is necessary to maintain a count of the number of occurrences of each term and each triple.

3. Design

We set out to design a dynamic HDT variant. This involves designing a new dictionary, a new triples section, and new update operations. We will first address the design choices we identified in our problem analysis.

For scope of dynamicity, we are satisfied with dynamicity of our data structures but not dynamicity of the HDT files themselves. File dynamiciy would restrict the freedom in design of both the dictionary and triples section. Regarding the ID ranges of the dictionary, we drop the guarantees of range continuity and lexicographic correspondence between IDs and the dereferenced terms because we estimate the runtime cost of these guarantees to be too high. Without lexicographic correspondence, we will have a more entropic triples section which will not be as compressible. We accept this cost. Further, we will not use the 'shared' sub-dictionary because the cost of the complications it introduces are not worthwhile for the increased compression rate.

3.1. A Dynamic Dictionary

We review relevant literature to find a suitable data structure capable of storing string-integer pairs which supports lookups in both direction, prefix searches, and incremental insertions and removals. There are two main approaches to dynamic string dictionaries: hash maps and trees. While hash maps can be performant and compact, they don't guarantee lexicographic sortedness, nor do they support prefix searches. Prefix trees, also known as tries, are a particularly popular class of trees for dictionaries designed for compactness. Tries appear the most suitable for our purposes, as we have not found any other class of data structure which satisfies all our requirements. Further, tries have been used in other linked data systems for storing terms, such as RDFVault [4]. We also find that trie-based dictionaries have been found to perform well for URI datasets [5]. We therefore decide to make use of tries.

3.1.1. Related work on Tries

A trie is a tree with labeled edges, where the concatenation of the edge labels of a path from the root to a leaf corresponds to an indexed key. The value associated with that indexed string is stored in the leaf node. When two indexed keys share a prefix, they will share a path up until their divergence. This allows for compression while the keys can remain sorted, if desired, by ordering the list of out-edges per internal node. String-to-ID operations are performed by following a path starting from the root while comparing the edge labels with the searched-for key, while ID-to-String operations are done by traversing from a leaf to the root while reading the edge labels. We find that a number of variations of tries exist, each with different optimizations.

For reducing the size of the data structure, a common and trivial technique is compaction. Compaction involves collapsing series of internal nodes with only one outgoing edge into a single edge. A more complex method for data structure minimization involves path-decomposition [6], but this technique handles deletions by simply marking nodes as deleted, which is not acceptable for us. Another proposal involves using 'micro-tries' [7, 8], where the attributes of the host architecture are exploited to fit small pieces of data which are likely to be accessed together into single RAM-words. [8] finds however that their implementation is dominated by other implementations for updates and lookups, which are important operations for us.

The Height-Optimized Trie (HOT) [9] handles keys on the binary level and considers a variable number of bits at each node in order to minimize the Trie's height. Naturally this also leads to a higher number of children per node. This means we have fewer node traversals, while a number of additional low-level techniques minimize the impact of the high fan-out. The paper demonstrates high space efficiency and good performance on lookups and insertions for URL datasets. While it is not stated in the paper, their code repository states a maximum key length of 255 bytes [10], which is an unacceptable limitation.

While String-to-ID operations always start at the root, ID-to-String operations require locating the correct node before starting the traversal towards the root. How to find these nodes is another design issue. Most papers either don't address the issue or suggest that if support for ID-to-String operations are desired, some secondary data structure should be used to map IDs to memory addresses for the leaves. An exception is RDFVault, where the IDs are memory addresses for leaf nodes, eliminating the need for the secondary structure.

Some papers present methods of reusing paths beyond the prefixes. A highly compact representation was proposed for the CDAWG structure [11], where tries are transformed into directed acyclic graphs with compaction. This yielded a very high efficiency, as substrings can be used in paths of keywords that do not share a common prefix. However, it came at a cost of high time complexity for updates.

The Double Trie [12] uses two Tries, one for suffixes and one for prefixes. Indexed strings are split in half, and the first half is inserted in the prefix Trie. The other half is reversed and inserted in the suffix Trie. The leaves of the Tries connect. This adds the capability of suffix search and doesn't reduce the node count by as much as the DAG of [11], but it achieves far less complex update operations.

Merged Tries [13] are similar to Double Tries, but merge the two tries into a single one. The root of the trie serves both as an indicator of the beginning and the end of a string. Leaves always link to another leaf, which is guaranteed to have a path back to the root. This allows reuse of string segments if the suffix of one string matches the prefix of another. The authors demonstrate a significant gain in memory efficiency over sets of natural-language words of the English languge. The authors do not evaluate their structure on other datasets, but we expect it would be less useful for datasets where prefixes are less likely to occur as reversed suffixes. For example, most URLs have *http* as a prefix, but very few URLs end with *ptth*. Thus it would not be likely to hold significant advantage over the Double Trie for RDF data.

3.1.2. Customizing our Trie

For our design we use a standard trie but make use of compaction as it is a simple technique which greatly reduces the size of the trie. We also take inspiration from RDFVault and use addressable IDs in order to support ID-to-String operations without the need for an additional data structure. While RDFVault uses this technique for in-memory addresses without consideration for persistence, we can use integers as offsets to compute addresses. This should allow keeping the IDs dereferencable upon persisting to disk and sharing files across computers. We will also keep a count of occurrences in each leaf node in order to not prematurely remove still needed data. Further, we will maintain lexicographic sorting of the outgoing edges of each node in order to keep each trie as a whole cheaply lexicographically iterable. This also allows us to terminate searches over out-edges early, which will be beneficial when checking whether new keys are already present in the trie.

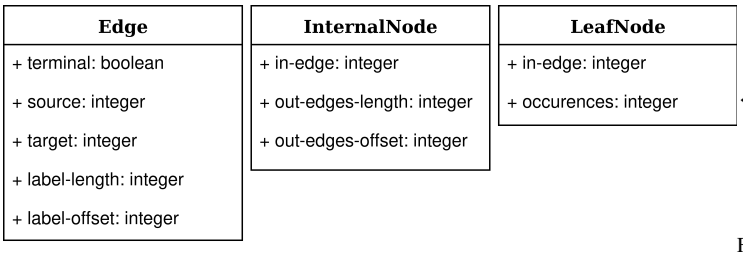

Fig. 1. Class diagrams for trie components

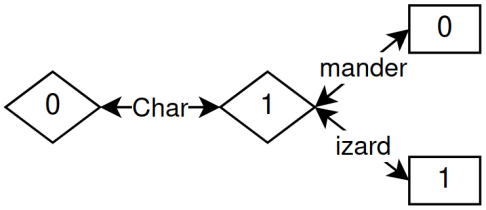

Fig. 2. An example trie, with the root at the left. Diamond boxes represent internal nodes and rectangles represent leaf nodes. Integers within shapes indicate node IDs.

For our internal representation, we have classes for internal nodes, leaf nodes, and edges, shown in Figure 1. These are stored in lists and reference each other by IDs. The ID of an item is implicitly defined by the item's position in its list. Internal nodes reference out-edges-offsets and out-edges-lengths, and edges reference label-offsets and label-lengths. Labels and out-edge IDs are stored in separate lists of characters and integers, respectively. These lists are sectioned by the entity referencing them, i.e. the *out-edges-length* values starting at index *out-edges-offset* of the list of out-edges are all IDs of edges emanating from the same internal node. The labels and out-edges are kept outside the only entities which reference them, because the number of out-edges per internal node and number of

characters per edge label are inconsistent. Thus, keeping e.g. the label within the edge class would make the size of the edge class inconsistent, and dereferencing an edge in the list of edges by ID would not be practical.

As an example, figure 2 shows a trie consisting of the two keys 'Charmander' and 'Charizard', inserted in that order. Note that IDs are independent per class of object, so we see the same IDs reoccurring for internal and leaf nodes. We represent the internal nodes with the sequence *0,0,1,0,1,2*. The first three integers represent the ID of the incoming edge, the out-edges offset, and the number of out-edges for the first internal node (ID 0), as seen in the class diagram. The following three integers represent the same fields for the second internal node (ID 1). The first node in the list is always the root node, as it is always the first internal node to be inserted and will never be removed. For the root node, *0* is used as a dummy value for the in-edge ID. This value is never read, because the root node has no incoming edge. Similarly, edges are internally represented as the sequence *0,0,1,4,0,1,1,0,6,0,1,1,1,5,6* and leaf nodes as *1,1,1,1*. There are five values for each of the three edges, and two values for each of the two leaf nodes. The meaning of each value can be read from the class diagram. Not shown is the sequence *c,h,a,r,m,a,n,d,e,r,i,z,a,r,d* representing the labels of the trie, and the sequence *0,1,2* representing the out-edges of the internal nodes.

Our description so far is accurate for existing tries that have been memory-mapped and which we only perform read-operations on. When constructing a new trie, we deviate slightly. We store labels in mappings from edge IDs to character vectors, and out-edge IDs in mappings from internal node IDs to numeric vectors. We do this because it is impractical to keep a single shared array for all out-edge IDs and label characters, as the data is at this point dynamic and we want to avoid repeatedly shifting the data of the array as new out-edges or labels are inserted.

Another case is modification of an existing trie, in which case we have data spread both over a memory-mapped area, containing previously persisted data, in addition to newly created data which is only present in-memory. Managing data spread in such a way introduces a few complications. First, we resolve conflicting IDs by offsetting the IDs of the in-memory data by the length of the corresponding lists in the persisted memory-mapped data. Determining in which location an item can be found can be achieved by comparing the ID to the length of the memory-mapped list of the item's class. Second, we are free to make edits in the memory-mapped area as long as the edit doesn't change the size of the elements. We make use of this ability to replace in-edge IDs of leaf and internal nodes if an insertion causes a split of a memory-mapped edge, to update the number occurrences for leaf nodes, to decrease the number of out-edges for internal nodes, and to decrease the lengths of edge labels upon splits. Such changes are not propagated to the existing file, but remain in our memory map. Thus, the input HDT file remains intact. Third, we also need to be able to keep lexicographic ordering of out-edges when a node has out-edges spread over both memory-mapped and new in-memory data. We resolve this by keeping using a two-layered iterator where the main iterator is a driver over two iterators, one for each data location. The main iterator compares the labels of each of the two edges.

During construction and modification, it's possible that data items are removed. We initially handle this by marking the item as deleted. When it's time to persist our new or modified trie to disk, we skip writing deleted items. Since this changes the position of the following items in the persisted array, it's also necessary to update ID references in other data elements. We do this by updating all ID references to internal nodes, leaf nodes, and edges at the beginning of our save procedure. An extra complication applies for leaf nodes, since their IDs are exposed to the triples section and should not change. To handle this, we maintain a list of 'holes' which we persist to disk. We define a hole by a starting ID and a size. Additionally, we store the cumulative size of this and all preceding holes . The list of holes is sorted by the starting IDs. As we remove leaf nodes, we either create a new hole or we expand an existing hole by decrementing its starting ID or incrementing its size. As an example, if we have four leaf nodes with IDs *0* until *3* and delete the two middle leaf nodes, we will persist node *0* immediately preceding node *3* to disk, along with a list of one hole which indicates that two leaf nodes starting at ID *1* were removed, and that the total number of deletions up until the end of this hole is two. This technique allows us to avoid storing large sparse arrays after removing large amounts of data, but it comes at the cost of needing to search the list of holes when converting an exposed ID to an internal leaf node ID. Since we maintain ordering of the list of holes, we can do this with a binary search. An improvement which we leave as future work would be to re-use IDs of deleted leaf nodes. This would positively impact performance, as the list of holes could shrink and searches would become faster.

3.1.3. A dynamic Triples section

Our design for the triples section is quite straight-forward. As in the existing BitMapTriples component, we use term IDs from the dictionary. We apply no compression beyond this, and simply represent our statements as a list of numeric 4-tuples. The first three numbers represent IDs for the subject, predicate, and object of a statement. The last number represents the number of occurrences of the statement. Our triples section supports insertion of a statement, iteration over statements, and application of a lexicographic sort over the set of statements that have been inserted in the current run. Lexicographic sorting is done by comparing the dereferenced terms from dictionary. Removals are done by decrementing a statement's counter. When persisting to disk, we skip writing statements with zero occurrences.

3.2. RDF-to-HDT

When creating a new HDT from RDF data we parse each statement and insert each term in our dictionary to obtain ID-triples. We then insert each ID-triple directly to our dynamic triples list. When the whole input has been processed, we lexicographically sort our triples section. We then persist our HDT to disk.

3.3. Updating from HDT sources

The static HDT variant supports merging two HDT files together with the *hdtCat* tool. This is almost what we want to do, except that we wish to keep track of the number of occurrences of statements in addition to obtaining a union. There are two key design differences between the static and our dynamic variant in this context. One is that we have a strictly lexicographically sorted triples section while the static variant does not. Another is that we do not have a 'shared' sub-dictionary. These differences greatly simplify our design compared to *hdtCat*. In our dynamic variant, it suffices to iterate over each of the HDT files' triples sections while inserting into a new triples section the lexicographically lesser of the two iterators' next triple. When a triple occurs in both inputs, the new triple's number of occurrences is set to be the sum of the occurrences of the two identical triples. Additionally, for each statement occurring in triples added from the second HDT file, we insert (or increment the number of occurrences of) each of the terms into our primary HDT's Dictionary.

Our design for subtracting one HDT from another is similar to our procedure for merging HDTs, except that instead of adding triples from both iterators we will only insert from the first iterator. When the first and second iterators next triples dereference to identical triples, we decrement the counter in the triple from the first iterator by the counter of the triple from the second iterator and remove or decrement the number of occurrences for each of the terms in our dictionary. We insert the triple to our new list only if the counter did not reach zero. We expect for all statements set to be removed to already be present in the original HDT.

3.4. Updating from RDF sources

To update from RDF sources, we first parse two input RDF files. One contains statements which should be removed, and the other contains statements which should be added. As with HDT subtraction, we expect that all statements in the RDF of removals are already present in the HDT file. We first read through the RDF of additions while inserting terms into our dictionary and adding ID-triples to a new triples component, which we sort lexicographically after the whole input has been processed. We similarly read and parse each statement of the RDF of removals and obtain ID-triples which we add to another new triples component. However, we do not immediately update our dictionary with the removals because we need to still be able to dereference IDs of removed triples later. Finally, we sort the second triples section as well. Sorting of the new triples components, and the String-to-ID operations done while parsing through the RDF file of removals, are read-only events. We are therefore able to sort the additions' triples in parallel with parsing the removals. Finally, we proceed to iterate over our three triples sections, i.e. that of our original HDT and the ones containing ID-triples which should be added or removed. This is done similarly to the updates from HDT sources.

4. Evaluation

We configure five different experiments where we compare the runtime and peak memory usage of the dynamic and static variants in order to answer our research questions. Specifically we report the peak resident set size, which is the peak amount of memory reserved by the process. We repeat each configuration three times and report the average of the outcomes. Before each run, we clear the memory cache.

We use 5% subsets of three datasets with different sizes and data characteristics. Largest is the crowd-sourced DBpedia dataset [14] which has an unusually large number of distinct predicates [15]. The object-positioned terms are diverse, and includes numbers, URLs, IRIs for other entities in the dataset, as well as names, natural-language titles, and paragraphs in several languages. We also use the Dutch land registry Kadaster's dataset of land features [16], which contains many WKT-shapes in the object position. WKT-shapes are series of coordinates which form shapes on 2D projections. These literals have a low degree of prefix overlap. Finally, we include the Dutch RVO's energy label dataset [17], which primarily has short literals in the object position.

4.1. RDF to HDT

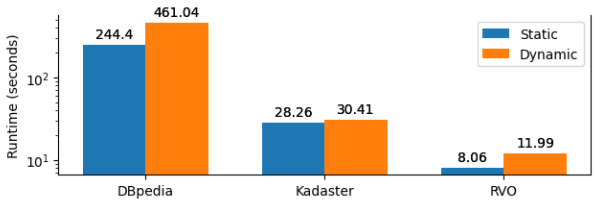

Fig. 3. Runtime, RDF-to-HDT

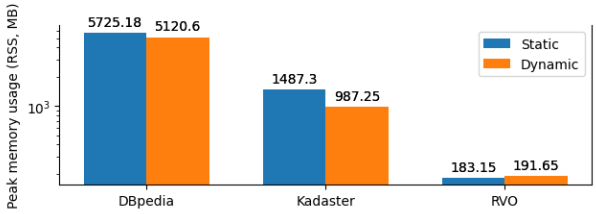

Fig. 4. Peak RSS, RDF-to-HDT

We find that converting from RDF to HDT is generally slower for the dynamic version than the static version, with some data-dependent variation. Still, they are within the same order of magnitude and the result is acceptable. For peak memory usage, we see that the dynamic variant slightly outperforms the static variant. This is due to that the static variant loads the entire uncompressed set of terms into memory before compression, while the dynamic variant incrementally adds new terms to the compressed data structure immediately. The difference is still not very large, since the final data structure produced by the dynamic variant is significantly larger than that of the static variant, as we shall see. We summarize the sizes of the original RDF files and produced HDT files from each variant, in addition to their compression rates in parentheses.

Table 1
Dataset file sizes (MiB) and compression rates (parenthesized)

Dataset	Original RDF	Static HDT	Dynamic HDT
DBpedia	3538	473 (7.49)	1869 (1.89)
Kadaster	623	258 (2.42)	468 (1.33)
RVO	158	6 (29.8)	54 (2.97)

We find that the static variant is consistently more compact than the dynamic variant. This is expected, as the dynamic variant uses an explicit data structure while the static variant stores only the compressed strings. Additionally, we don't attempt to compress the list of triples and we don't use a shared sub-dictionary.

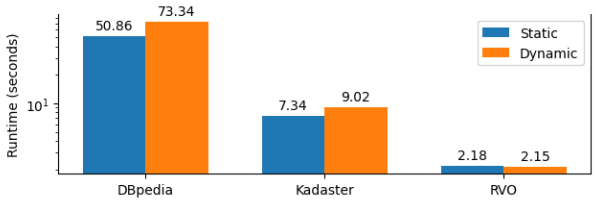

Fig. 5. Runtime, HDT-to-RDF

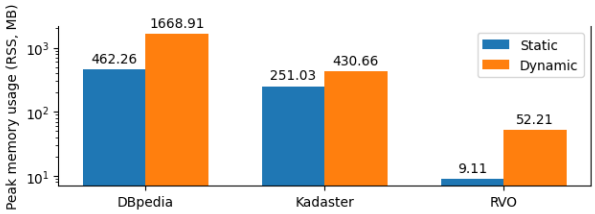

Fig. 6. Peak RSS, HDT-to-RDF

4.2. HDT to RDF

Converting from HDT to RDF is generally slower for the dynamic version, although not by a large margin. As for memory usage, we can observe that peaks are higher for the dynamic variant compared to the static variant. Since the HDT-to-RDF operation needs to access the entirety of the HDT file, the size of the HDT file becomes the primary factor for the peak RSS. Since the dynamic variant produces less compact files, it uses the most memory when converting the entire file. Since memory lookups have a runtime cost, this could also explain the difference in runtime.

4.3. Merging HDTs

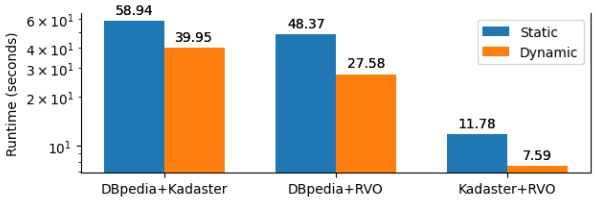

Fig. 7. Runtime, HDT merge

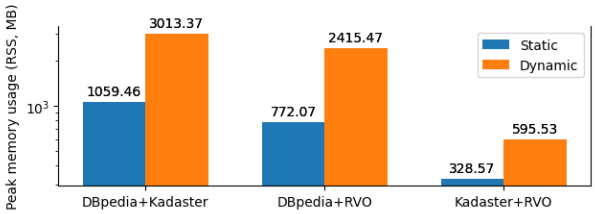

Fig. 8. Peak RSS, HDT merge

We find that our dynamic variant is faster for all configurations. This is likely due to that our HDT merging procedure is simpler than that proposed by HDT-Cat [2]. While HDTcat needs to account for the shared set of triples in its dictionary, we have dropped this sub-dictionary and benefit from already strictly sorted triples section. Still, HDTcat was primarily motivated by achieving a reduction in memory usage, and their implementation is still superior to ours in this regard. Again, we explain this by the larger HDT sizes for the dynamic variants.

4.4. Subtracting a HDT from a merge

We compare the performance of removing one of our HDT files from a merge consisting of all three HDT files. For the dynamic variant, this is done by using the new utility *hdtSub*. For the static variant, we generously assume we already have sorted RDF versions of both files. We use the *comm* tool to produce a new RDF file which contains only the statements of the larger merge file that aren't also present in the smaller file containing subtractions. We choose *comm* since it is streaming-based and therefore scales to arbitrarily large datasets in terms of memory use.

We find that our dynamic variant is faster when removing the Kadaster and RVO datasets, but not for the DBpedia dataset. This is no surprise, considering that the DBpedia dataset takes up about 83% of the merged HDT file's statements. We see that there is very little difference between the runtime of removing the smaller two datasets, indicating that most of the runtime is overhead involved in page loads for the and writing of the output file.

For memory usage, we see that the static variant's memory usage scaled based on the size of the remaining data. This makes sense because the process involves *rdf2hdt* loading the terms of the remaining data, which is smaller when more data is removed, into memory. For the dynamic variant memory use scales with the size of the removed data, because it is the removed data which we access and manipulate. The relationship is not linear, because each page load increases the chance that the data needed for the next removal is already present in a loaded page.

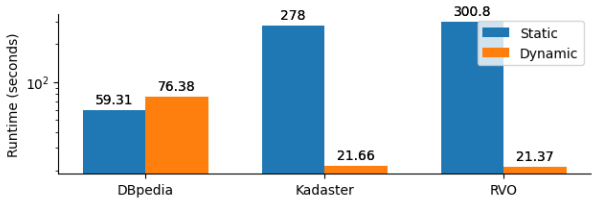

Fig. 9. Runtime, HDT subtraction

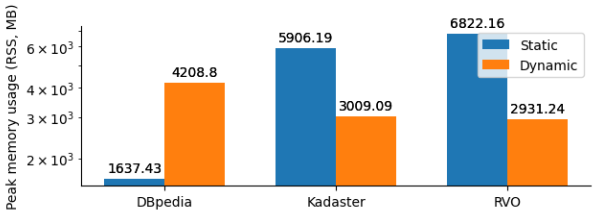

Fig. 10. Peak RSS, HDT subtraction

4.5. Modifying HDT from RDF sources

In this experiment we simultaneously add and remove statements from RDF files. For the dynamic variant, we use the new *modifyHdt* utility, to which we pass two RDF files. One RDF file contains a set of triples which is present in the HDT file and which should be removed. The other contains a set of triples which is not present in the HDT file, and which should be added. The RDF files are sourced from the HDT file itself, extracted systematically from every N lines of the HDT file. The triples which should be added to the HDT file were first removed from the HDT file, so as not to re-add them. We vary the value of N to observe the effect of differently sized update batches. Note that this systematic extraction is a worst-case scenario for our dynamic variant, as it means the data we will need to access and update is evenly distributed throughout the data structure.

For the static variant, we generously assume that a modified RDF file has already been produced, on which we perform the *rdf2hdt* operation. Since the size of the update batch shouldn't have any significant influence on the static variant, we simply use the unmodified RDF version of each dataset.

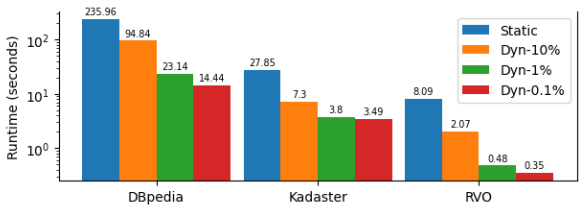

Fig. 11. Runtime, Update from RDF

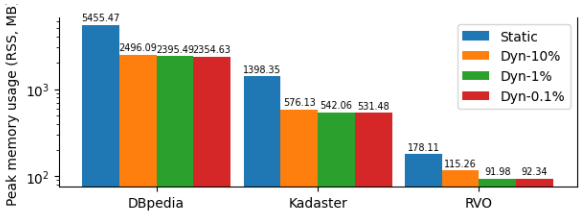

Fig. 12. Peak RSS, Update from RDF

We find that updating the dynamic dictionary is generally faster than regenerating the static dictionary from scratch, up to a certain update batch size. This is acceptable in most cases, as this feature is meant for small incremental updates to potentially large datasets. While it may appear that the cost of the update operation rises exponentially as the size of the update operation, it should be kept in mind that there is a constant cost to writing the roughly equally sized output files to disk.

In terms of memory usage, we see that updating the dynamic dictionary uses significantly less memory for updates than the static dictionary uses for regeneration. This makes sense, as we're only accessing a proportion of the dynamic dictionary's data in memory, while the static dictionary needs to load the complete sets of terms into memory.

4.6. Results summary

We present numeric summaries of the relative speedups (or slowdowns) for the various operations. The values represent the runtime or peak RSS for the dynamic variant divided by the corresponding value for the static variant. For the 'RDF updates' row, the slash-separated values within each cell refer to the relative performance for each update batch size.

For RDF-to-HDT and HDT-to-RDF, we found that the static variant consistently outperforms the dynamic variant to various degrees depending on the dataset. This is an expected and acceptable performance loss, where we are

Table 2
Runtime, dynamic/static

Experiment	DBpedia	Kadaster	RVO
RDF-to-HDT	1.89	1.08	1.49
HDT-to-RDF	1.44	1.23	0.98
HDT subtraction	1.29	0.08	0.07
RDF updates	0.42/0.1/0.06	0.29/0.13/0.12	0.27/0.06/0.04
	DB.+Ka.	DB.+RVO	Ka.+RVO
HDT merging	0.68	0.57	0.64

Table 3
Peak memory usage (RSS), dynamic/static

Experiment	DBpedia	Kadaster	RVO
RDF-to-HDT	0.89	0.66	1.05
HDT-to-RDF	3.61	1.72	5.73
HDT subtraction	2.57	0.51	0.43
RDF updates	0.5/0.42/0.41	0.46/0.37/0.36	0.68/0.51/0.5
	DB.+Ka.	DB.+RVO	Ka.+RVO
HDT merging	2.84	3.13	1.81

satisfied to stay within an order of magnitude from the static variant. The one exception is memory usage for converting RDF to HDT, where the dynamic variant outperforms the static variant for the larger two datasets. For merging HDT files, we find that our dynamic variant is faster than the static variant, but consumes more memory. For HDT subtraction, we observe a performance gain when the HDT we remove does not make up the majority of the file we're subtracting from. For updates from RDF files we observe speedups for update sizes at least up to 10% of the HDT file, while also using less memory than the static variant requires for regenerating the HDT from scratch.

Overall, our dynamic variant stays within an order of magnitude of the static variant when comparing existing functionality, while we see speedups for the new functionalities when using typical problem sizes. So let us answer our research questions. For questions one and two, we can confidently answer with a yes. For question three and four, we find that the answer depends on the update batch size. When specifically considering the realistic use-cases we mentioned in the introduction, namely making small incremental changes and maintaining a large union HDT file, our dynamic variant appears superior.

5. Conclusion

The HDT format has been around for more than a decade. The addition of HDT-Cat introduced the notion of dynamicity in HDTs, but is limited to additions. In this project we've presented a first fully dynamic version. The new dynamic HDT variant is generally a downgrade from the static variant for most existing functionality, with some exceptions, but we demonstrate that it is still a feasible alternative to the familiar static HDT. Our dynamic HDT variant allows for performant update operations, unblocking some important use-cases such as maintaining HDT files over dynamic data sources, keeping a merge of a changing set of HDTs, and enabling small manual changes to large datasets.

5.1. Future work

Our contribution is a proof-of-concept rather than a production-ready HDT variant. While we've outlined a design which allows for update operations at an acceptable cost for existing functionalities, there may be variations or alternatives to aspects of our design, not considered in our work, which would either yield strictly better results or trade performance in one area for performance in another which may be more important for some use-cases.

We also specifically mention some potential improvements of which we are aware. Our procedure for adding and subtracting HDTs could be enhanced by first processing all changes to the dictionary, and then writing from the triple iterators directly to disk instead of first constructing a new Triples section in memory. This would incur a lower memory cost than the current approach. Additionally, the independence of the sub-dictionaries could be exploited for parallelization. While the overhead of creating three new threads for each insertion is too high to be beneficial, it might be feasible to keep workers that process queues of insertions or removals in each sub-dictionary. Another suggestion is to make use of Double Tries [12] or a Merged Tries [13] to enhance compression rate by also merging shared suffixes. This is something we considered, but decided against in order to avoid a too complex design in this first proposal for a dynamic HDT. Further, re-using IDs of deleted terms would reduce the size of our list of holes and speed up lookups in modified HDT files. Finally, as the author of this paper has limited experience

with the language in which the implementation is written (C++), it is possible that there are ways to enhance the implementation on the lower level, without necessarily modifying the design itself.

References

[1] Javier D. Fernández, Miguel A. Martínez-Prieto, Mario Arias, Claudio Gutierrez, Sandra Álvarez-García, and Nieves R. Brisaboa. Lightweighting the web of data through compact rdf/hdt. In Jose A. Lozano, José A. Gámez, and José A. Moreno, editors, *Advances in Artificial Intelligence*, pages 483–493, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Dennis Diefenbach and José M Giménez-García. Hdtcat: let's make hdt generation scale. In *International Semantic Web Conference*, pages 18–33. Springer, 2020.

[3] Mario Arias Gallego, Claudio Gutierrez, Axel Polleres, Javier Fernández, and Miguel Martínez-Prieto. Hdt - your binary format for rdf.

[4] Hamid R. Bazoubandi, Steven de Rooij, Jacopo Urbani, Annette ten Teije, Frank van Harmelen, and Henri Bal. A compact in-memory dictionary for rdf data. In *The Semantic Web: Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Proceedings*, volume 9088, pages 205–220. Springer/Verlag, 2015.

[5] Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux. A comparison of data structures to manage uris on the web of data. volume 9088, pages 137–151, 05 2015.

[6] Shunsuke Kanda, Dominik Köppl, Yasuo Tabei, Kazuhiro Morita, and Masao Fuketa. Dynamic path-decomposed tries. *ACM Journal of Experimental Algorithmics*, 25:1–28, 11 2020.

[7] Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E100.A, 02 2016.

[8] Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. c-trie++: A dynamic trie tailored for fast prefix searches. pages 243–252, 03 2020.

[9] Robert Binna, Eva Zangerle, Martin Pichl, Guenther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. pages 521–534, 05 2018.

[10] Robert Binna, Eva Zangerle, Martin Pichl, Guenther Specht, and Viktor Leis. Hot git repository.

[11] Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34:578–595, 1987.

[12] Katsushi Morimoto, Hirokazu Iriguchi, and Jun-Ichi Aoe. A dictionary retrieval algorithm using two trie structures. *Systems and Computers in Japan*, 26(2):85–97, 1995.

[13] Antonio Ferrández and Jesús Peral. Mergedtrie: Efficient textual indexing. *PLOS ONE*, 14:e0215288, 04 2019.

[14] DBpedia 2017. Dbpedia association. https://triplydb.com/DBpedia-association/dbpedia. Accessed: 2022-06-01.

[15] Sparql query yielding the number of distinct predicates in dbpedia. https://api.triplydb.com/s/JIfJbDiFH. Accessed: 2022-06-01.

[16] Kadaster. Kadaster knowledge graph. https://data.labs.kadaster.nl/kadaster/kg. Accessed: 2022-06-01.

[17] Energielabels. Rijksdienst voor ondernemend nederland (rvo). https://triplydb.com/rvo/energielabels. Accessed: 2022-06-01.