Open Access

Open Journal of Databases (OJDB)
Volume 2, Issue 1, 2015
http://www.ronpub.com/ojdb
ISSN 2199-3459

PatTrieSort - External String Sorting based on Patricia Tries

Sven Groppe [A], Dennis Heinrich [A], Stefan Werner [A],
Christopher Blochwitz [B], Thilo Pionteck [B]

[A] Institute of Information Systems (IFIS), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany, {groppe, werner, heinrich}@ifis.uni-luebeck.de

[B] Institute of Computer Engineering (ITI), University of Lübeck, Ratzeburger Allee 160, D-23562 Lübeck, Germany, {blochwitz, pionteck}@iti.uni-luebeck.de

Abstract

External merge sort belongs to the most efficient and widely used algorithms to sort big data: As much data as fits inside is sorted in main memory and afterwards swapped to external storage as so called initial run*. After sorting all the data in this way block-wise, the initial runs are merged in a merging phase in order to retrieve the final sorted run containing the completely sorted original data. Patricia tries are one of the most space-efficient ways to store strings especially those with common prefixes. Hence, we propose to use patricia tries for initial run generation in an external merge sort variant, such that initial runs can become large compared to traditional external merge sort using the same main memory size. Furthermore, we store the initial runs as patricia tries instead of lists of sorted strings. As we will show in this paper, patricia tries can be efficiently merged having a superior performance in comparison to merging runs of sorted strings. We complete our discussion with a complexity analysis as well as a comprehensive performance evaluation, where our new approach outperforms traditional external merge sort by a factor of 4 for sorting over 4 billion strings of real world data.*

Type of Paper and Keywords

Regular research paper: *External sorting, string sorting, patricia tries, merge sort, big datasets*

1 Introduction

Sorting is one of the fundamental problems of computer science [14]. Sorting data not fitting into main memory is called *external sorting*. Although the sizes of the main memories of computers increase continuously, the data sets also become larger and larger (categorized as big data trend).

In the area of the Semantic Web, there are masses of data with over 30 billions triples in nearly 300 datasets with over 500 million links between these datasets freely available to the public - thanks to the efforts of the linked data initiative [15]. Most of these datasets are too large to fit into main memory. Efficient processing of these data sets requires indexing approaches (e.g., [19, 27]), and sorting the data is one of the basic steps, which are typically done for index construction [12]. The most widely used index type in databases is the B^+-tree. B^+-trees can be built very efficiently from sorted data by avoiding costly node splitting (see [16] and extend its results to B^+-trees). Thus, the performance of index construction from scratch relies heavily on the techniques of data sorting.

The *external merge sort* [14] first generates initial runs of sorted data. An initial run is typically computed by

reading as much data as possible from input into main memory, and sorting these data using main memory sorting algorithms. The alternative approach replacement selection [9] uses a heap to generate longer initial runs. Runs are written into external storage and merged afterwards to larger sorted runs until all the data is sorted.

Patricia tries (e.g., [1]) are space-efficient data structures for storing strings. Common prefixes of strings are stored only once in this data structure. We can retrieve the contained strings in sorted order by just one left-order traversal through its internal data structure in form of a tree. Especially the terms of Semantic Web data consist of many long common prefixes, such that using patricia tries for Semantic Web data offers obviously a good compression and low memory consumption.

Hence we propose a new external sorting algorithm *PatTrieSort* based on patricia tries: We generate the initial runs by inserting the input strings into patricia tries and we are swapping these patricia tries as initial runs to disk if they are not fitting into main memory any more. For the merging phase, we developed a merging algorithm, the input of which are the swapped patricia tries. There are advantages of PatTrieSort in the phase of initial run generation as well as in the merging phase: Initial runs can be quite large because of the good compression of patricia tries, such that more strings can be held in main memory. In the merging phase, merging patricia tries instead of strings avoids the comparison of common prefixes, which is significantly more efficient.

The main contributions of this paper include:

- a new sorting approach *PatTrieSort* as variant of external merge sort for sorting strings, where the initial runs are generated by using patricia tries. The initial runs are swapped in form of patricia tries to disk and are merged in a later processing step.
- a new algorithm for merging patricia tries used within the merge phase of the new sorting approach.
- a complexity analysis for the new merging algorithm in terms of runtime, I/O costs and memory consumption.
- a comprehensive performance evaluation and analysis of PatTrieSort compared to the other external merge sort variants using large-scale datasets with over 1 billion strings.

2 Basic Data structures and Sorting Algorithms

In this section we shed light on the foundations of the most important family of external sorting algorithms, the *external merge sort* algorithms, as well as introduce its variants. Furthermore, we shortly introduce patricia tries which are extensively used by our new sorting approach for strings.

2.1 Heap

A (min-) heap is an efficient data structure to retrieve the smallest item from the items stored in the heap (see [17]). Adding an item into the heap and removing the smallest item from the heap is done in $O(\log n)$, where n is the size of the heap, i.e., the maximum number of items of the heap. Internally, the heap is organized as tree, most often as complete binary tree. The root of each subtree contains the smallest item of the subtree. Complete binary trees can be memory-efficiently stored in arrays, where the index of the children and the parent can be computed by simple formulas. Adding an item is done by inserting the item as leaf to the heap tree and swapping the item with its parent as long as it is smaller than its parent (*bubble-up*). Therefore, the smallest item in the heap is always stored in the root of the whole tree. When the smallest item is taken away, the item in the most-right leaf of the bottom level is moved to the root. Afterwards, the root item is recursively swapped with its minimum child if the minimum child is smaller than it (*bubble-down*). Therefore, the smallest item in the heap is always stored in the root of the tree.

After the smallest element in the heap is taken away, a succeeding insertion of a new element can be optimized by first placing the new element in the root and then performing a bubble-down operation. This approach to optimize a pair of removing and insertion operations avoids the bubble-up operation. We use this improvement during initial run generation of the sorting approach replacement selection (see Section 2.3).

2.2 (External) Merge Sort

(External) merge sort (see [14]) is known to be one of the best sorting algorithms for external sorting, i.e., where the data is too large to fit into the main memory. The merge sort algorithm first generates several initial runs, which contain already sorted data. The initial runs are afterwards merged to generate a new round of runs. A new run contains the sorted data of its merged runs, so that the number of new runs becomes less while the size of each new run increases. This process is repeated until all the data is sorted.

Instead of merging only two runs, it is more efficient to merge several runs. In order to merge a new run from several runs, we always need to find the smallest items from these runs. Thus, a heap is the ideal data structure to perform this task.

The runs can be generated by reading as much data into main memory as possible, sorting this data and write

this run to external storage. For sorting the data in main memory, any main memory sort algorithm can be chosen [10], e.g. quicksort, (main memory) merge sort (and its parallel version) or heapsort, which are well-known to be very fast.

2.3 Replacement Selection

Another variant of external merge sort, called replacement selection [9], uses a heap to increase the length of the initial runs on average by a factor of 2. Whenever the heap is full, its root item is retrieved and written to the current run. In the heap, items, which still can be written into the current run (i.e., which are greater than or equal to the last item written into the current run), are ordered before those items, which must be written into the next run (i.e., which are smaller than the last item written into the current run). The values of the items are the second order criterion in the heap. If the value of the root item is smaller than the last item of the current run, the current run is closed, and a new run is created and becomes the current run.

2.4 Patricia Tries

Tries (e.g., [1]) serve as efficient data structure for storing strings of characters with common prefixes (e.g., see Figure 1 a)). Common prefixes of all strings, which are contained in the trie, are stored only once in the trie. For this purpose, the trie is a tree structure, in which each edge is labeled with one character, and the concatenation of the characters along the path from the root to a leaf is one of the stored strings in the trie. The edges of a parent node must contain different characters as labels, and are ordered according to the lexicographical order of their labels.

Patricia tries are compressed tries (see Figure 1), where the edges can contain not only one character as label, but a string of several characters. In comparison to tries all nodes (except of the root node) with only one child are therefore melt together with their single child (and the edge between them is removed). The label of the incoming edge of the parent node is set to the concatenation of its previous label and the label of the old edge to the child. The order of the edges corresponds to the lexicographical order of their labels, such that searching within the patricia trie and therefore also update operations are more efficient. Note that the label can be an empty string (denoted by $\emptyset$) occurring if the patricia trie contains a string, which is a substring of another one in the patricia trie.

Semantic Web data typically consist of many strings with common prefix, as often IRIs [6] are used. Thus, patricia tries are the ideal data structure to store Semantic Web data in main memory. We show in the experiments that patricia tries are also ideal for sorting "normal" strings, not only Semantic Web data.

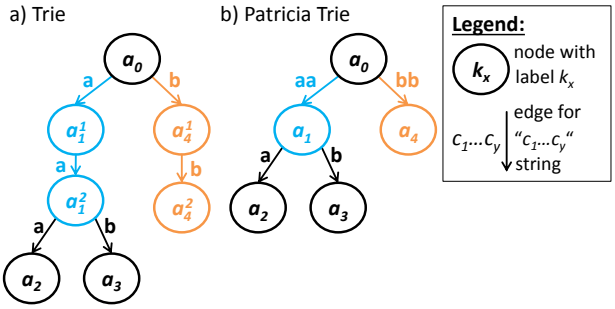

Figure 1: a) Trie and an equivalent b) patricia trie containing the strings "aaa", "aab" and "bb"

2.5 Further Related Work

While [22, 5] introduce basic sorting algorithms in more detail, [26, 3] are appropriate as surveys on external string sorting.

Some contributions utilize tries already for sorting (e.g., burstsort and its variants [25, 24]). In burstsort, a trie is dynamically constructed as strings are sorted, and is used to allocate a string to a bucket. For full buckets new nodes of the trie are constructed the leafs of which are again buckets. However, these algorithms work only in main memory for the purpose of lowering the rate of cache miss and are not developed for external sorting.

The main idea (and conclusion) of [28] is that it is faster to compress the data, sort it, and then decompress it than to sort the uncompressed data. This approach reduces disk and transfer costs, and, in the case of external sorts, cuts merge costs by reducing the number of runs. The authors of [28] propose a trie-based structure for constructing a coding table for the strings to be sorted. In comparison, we do not use codes, but we also store *compressed runs* by storing the patricia trie containing all the entries of the run, which reduces the space on disk and in memory, too.

The contributions in [4] lay the foundations for a complexity analysis for I/O costs for the string sorting problem in external memory. Its contribution covers the discussion of optimal bounds for this problem under different variants of the I/O comparison model, which allow or not allow strings to be divided in single characters in main memory and/or on disk.

3 PatTrieSort

We propose a new sorting algorithm PatTrieSort as variant of external merge sort, where patricia tries are exten-

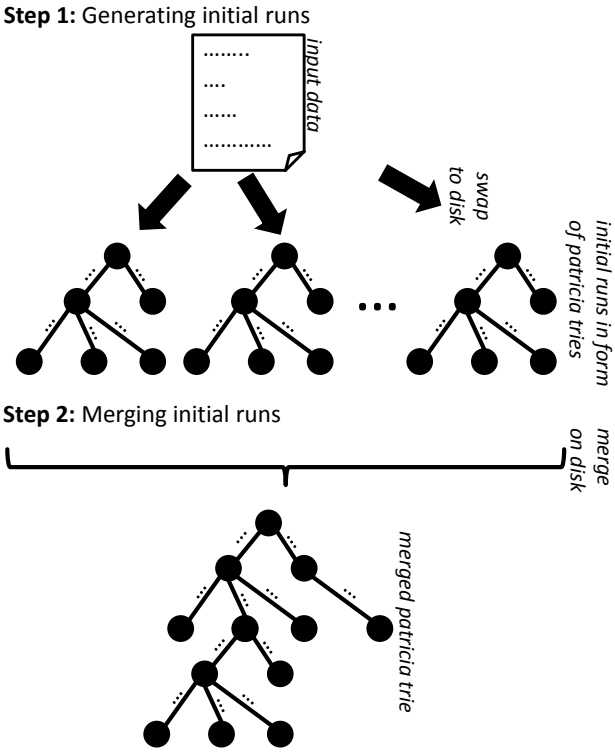

Figure 2: Overview of the main phases of PatTrieSort

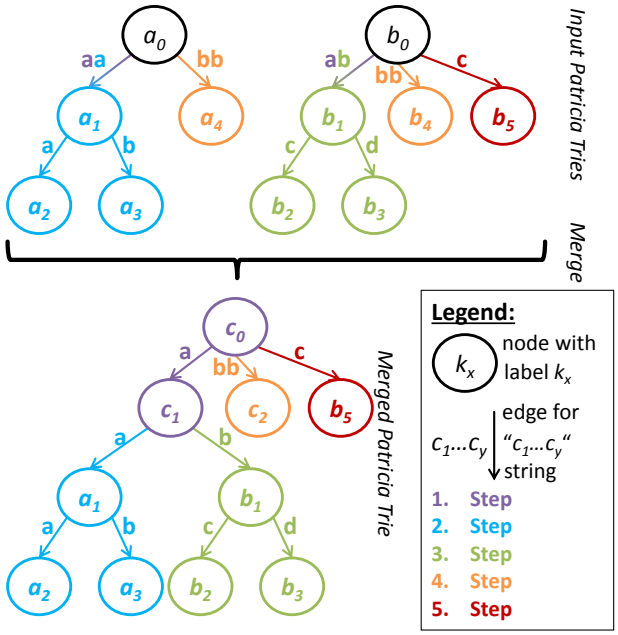

Figure 3: Example of merging 2 patricia tries

sively used.

As external merge sort, PatTrieSort has two phases (see Fig. 2): In the first phase initial runs (in form of patricia tries) are generated and swapped to disk. In the second phase the initial runs are merged until only one run (in form of a patricia trie) remains, which contains the sorted result. An optional step may be used to retrieve the sorted list of strings from the final run.

Within the first phase, the initial runs are generated by inserting the strings to be sorted into a patricia trie. If the main memory is full, the patricia trie is swapped to disk. It is important that the patricia tries are swapped to disk in a format, where the structure of the patricia tries remains, and the nodes of the patricia trie are stored by a left-order traversal through the patricia trie.

Within the second phase, the initial runs are merged and stored in a merged patricia trie. We present the merge algorithm in the next subsection. The merge algorithm can have an arbitrary number of patricia tries as input, reads and processes all these input patricia tries by a left-order traversal, and stores the resultant merged patricia trie again in a left-order traversal. Hence the merge algorithm can merge as many patricia tries at once, as many nodes of patricia tries can be intermediately held in main memory. Typically there is only one merging step necessary even for huge data sets to be sorted, which further improves the speed of the overall sorting algorithm.

3.1 Merging Patricia Tries

While the algorithm for inserting in a patricia trie is well-known [8], merging patricia tries, which we extensively use in PatTrieSort for merging the initial runs in form of patricia tries, has not been investigated to the best of our knowledge.

3.1.1 Examples of Merging Patricia Tries

We will start with two examples in Fig. 3 and Fig. 4 for merging patricia tries. Based on the examples, we will afterwards formulate the algorithm for merging patricia tries.

In Fig. 3, two patricia tries are merged. The different nodes and edges to be considered in the input patricia tries as well as the nodes and edges created (or copied from an input patricia trie respectively) in the different steps are marked by different colors. In the first step the common prefix 'a' of the labels of the first edges (be-

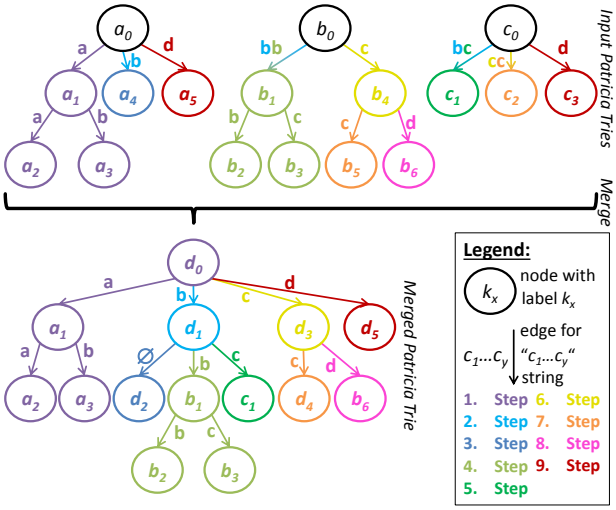

Figure 4: Example of merging 3 patricia tries

tween a_0 and a_1 as well as between b_0 and b_1) in both input patricia tries are considered and lead to the nodes c_0 and c_1 and the edge labeled with 'a' between them. Because of the distinct postfixes 'a' and 'b' of the labels of the edges between a_0 and a_1, and between b_0 and b_1 respectively, the subgraph with nodes a_1, a_2 and a_3 is copied from the first input patricia trie in the second step, and the subgraph with nodes b_1, b_2 and b_3 is copied from the second input patricia trie in the third step. The string 'bb' is contained in both input patricia tries. Hence, the nodes and edges for 'bb' are created only once in the merged patricia trie. If we want to support duplicates in our sorting algorithm, we need to hold a counter at each leaf node for representing the number of occurrences. In the latter case, we would need to compute the sum of the occurrences of 'bb' in both input patricia tries, and just store the sum in the counter of the merged patricia trie. Our merge algorithm will have only few additional steps when duplicates should be considered. In the last step, the remaining edge and node b_5 for the string 'c' is copied into the merged patricia trie.

If the patricia tries are held in main memory with a pointer structure and the input as well as the merged patricia tries are not modified any more after merging, we can speed up performance: we can avoid the costly operation of copying whole subtries and just use references to the subtrie in the corresponding input patricia trie. If we would allow to modify an input or the merged patricia trie, then the modification would lead to side-effects in the other patricia trie.

In Fig. 4, three patricia tries are merged. There are analogous steps as for merging two patricia tries, we just have to consider the nodes and edges of an additional patricia trie. Merging even more patricia tries is also possible.

3.1.2 Merge Algorithm

One can imagine that the merge algorithm can be easily generalized to merge an arbitrary number of patricia tries. We only present the generalized merge algorithm in the following.

Algorithm 1: MainMergePatTries

Input : patricia tries $T_1 \ldots T_n$, where $\forall i \in \{1, ..., n\} : T_i = (r_i, V_i, E_i)$ with r_i root node of the patricia trie i, V_i its set of nodes and E_i its set of edges

Output: Merged patricia trie

1 Create node r
2 return $MergePatTries(T_1 \ldots T_n, (r, \{r\}, \emptyset), r)$

A patricia trie is represented in the merge algorithms by a triple (r, V, E), where r is its root node, V its set of nodes and E its set of edges. An edge $e \in E$ is represented by a triple (v_s, v_e, l), where $v_s, v_e \in V$. This edge is a directed edge from node v_s to node v_e and is labeled with a string $l = c_1 \ldots c_m$, where c_i are characters. We use the notation $l[k]$ for the k-th character in l. We define $l[k]$ to return the empty character for $k \in \mathbb{N}$ if l is the empty string. We define the empty character to be the smallest character and use a function min to retrieve the smallest character from a given set of characters. The notation $\mid l \mid$ represents the number of characters in l. Hence, $l[k+1] \ldots l[\mid l \mid]$ represents the substring of l after the first k characters (and is the empty string in the case that l contains only k characters). A node v is a leaf node if $\nexists v_e, l : (v, v_e, l) \in E$. For a leaf node v, $count(v)$ represents the number of occurrences of the string represented by the leaf node v. We extend the standard definition of patricia tries at this point in order to deal also with duplicates during sorting.

Algorithm 1 contains the main algorithm, which just creates a dummy patricia trie with one node for holding the resultant merged patricia trie later and calls Algorithm 2 with it (additionally with its root node and the input patricia tries).

Algorithm 2 first considers the left-most (unmarked) edges of all input patricia tries in line 3. Already considered edges will be marked later (in line 6).

The minimum first character (or the empty character respectively) among the labels of the left-most (unmarked) edges in E is computed in line 4. In line 5 all those edges are filtered from E, the labels of which start with the minimum character (or which labels are the empty string respectively), and stored in M. In line

Algorithm 2: MergePatTries

Input : • patricia tries $T_1 \dots T_n$, where $\forall i \in \{1, ..., n\} : T_i = (r_i, V_i, E_i)$ with r_i root node of the patricia trie i, V_i its set of nodes and E_i its set of edges
• patricia trie $R = (r_R, V_R, E_R)$ for the result
• root node r of the current subtrie in R

Output: Merged patricia trie

1 $jobs \leftarrow \emptyset$ // for collecting remaining pat. tries to store the merged patricia trie according to a left-order traversal
2 **while** *any T_i contains unmarked edge(s)*
do
3 $E \leftarrow \{e \mid \exists v \in V_i, l : e = (r_i, v, l) \in E_i \wedge e$ is left-most unmarked edge in $T_i\}$
4 $c_1 \leftarrow min(l[1] \mid \exists v \in V_i, l : (r_i, v, l) \in E)$
5 $M \leftarrow \{e \mid \exists v \in V_i, l : (r_i, v, l) = e \in E \wedge l[1] = c_1\}$
6 Mark all edges of M
7 $P \leftarrow c_1 \dots c_k$, where $\forall (r_i, v, l) \in M : c_1 \dots c_k = l[1] \dots l[k] \wedge k$ is maximal
8 Create new node w
9 $V_R \leftarrow V_R \cup \{w\}$
10 $E_R \leftarrow E_R \cup \{(r, w, P)\}$
11 $Y \leftarrow \emptyset$ // for holding the remaining pat. subtries to be recursively merged
12 **foreach** $e = (r_i, v, l) \in M$ **do**
13 **if** *v is a leaf node* $\wedge \exists t^*, v^* : (t^*, \{t^*, v^*\}, \{(t^*, v^*, l[k+1] \dots l[|\ l\ |])\}) \in Y$ **then**
14 $count(v^*) \leftarrow count(v^*) + count(v)$
else
15 Create new node t
16 $V_X \leftarrow \{t\}$
17 $E_X \leftarrow \{(t, v, l[k+1] \dots l[|\ l\ |])\}$
18 Copy subtrie with root node v to V_X and E_X
19 $Y \leftarrow Y \cup \{(t, V_X, E_X)\}$
20 $jobs \leftarrow jobs \cup \{(Y, R, w)\}$
21 **foreach** $(Y, R, w) \in jobs$ **do**
22 $R \leftarrow mergePatTries(Y, R, w)$
23 **return** R

7 the longest possible common prefix $c_1 \dots c_k$ of the labels in M are determined. Lines 8 to 10 add a new node w and an edge to this new node with the longest possible common prefix $c_1 \dots c_k$ as label.

Line 11 initializes a data structure for holding the remaining patricia subtries to be recursively merged (later in line 22) and added as subtrie to w in the final merged patricia trie.

For each edge e in M, in line 13 our algorithm checks whether or not its target node is a leaf node and there exists already a patricia trie in Y, which contains only one edge with the same postfix as label: original label minus the common prefix $c_1 \dots c_k$. If it is the case, then the number of occurrences are added together to be the new number of occurrences for the string represented by this leaf node (see line 14). If duplicates should not be considered, this line 14 just may do nothing. Otherwise in lines 15 to 19, a patricia trie is added to Y containing an edge from a new node to the target node v of the considered edge e with the postfix (original label minus the common prefix) as label, as well as the whole patricia subtrie of v. Actually in a more efficient implementation, copying subtries in line 18 is not necessary and the same can be achieved by storing references to these subtries and accessing the original subtries when needed. Line 20 stores a job for merging the sub-patricia tries of Y. Afterwards, the algorithm continues to handle the remaining unmarked edges at line 3.

If all edges have been considered (and all edges are marked, line 2), the sub-patricia tries stored in $jobs$ are merged (lines 21 and 22). Note that instead of holding the sub-patricia tries to be merged in $jobs$, they could already be merged in line 20. However, first collecting the sub-patricia tries in $jobs$ has the advantage, that the nodes of the merged patricia trie can be stored in the order of a left-order traversal (by just storing the nodes and

edges of the merged patricia trie in lines 9 and 10, and storing a mark for the start of a next node after leaving the loop of line 2 to 20). This has considerable benefits for stream processing, which requires the patricia trie in a stream to be read in the order of a left-order traversal (see next subsection).

Finally, the merged patricia trie R is returned in line 23.

3.1.3 Dealing with Streams

Investigating Algorithm 2, we notice that the nodes and edges of the input patricia tries are accessed in the order of a left-order traversal. Hence, if the input patricia tries can be accessed in streams in left-order traversal, the merge algorithm works correctly also with streams. Furthermore, the merged patricia trie can be stored in a stream in left-order traversal.

These properties have several benefits:

- The merge algorithm can efficiently handle large patricia tries, which are serialized on disk in left-order traversal. Furthermore, the merged patricia trie can be easily serialized on disk in the order of a left-order traversal (and be the input for further merge steps).
- The merge algorithm can be deployed (as merge service) in distributed scenarios, where the patricia tries are sent and consumed in streams. The output stream of one merge service can be the input of another, leading to possibly complex merge trees in a distributed fashion.

3.2 Complexity Analysis

We will discuss the complexity of the merging algorithm in Algorithm 1 in terms of memory consumption, I/O costs and runtime in the following subsections. Let n be the number of input patricia tries, x the number of strings contained in all input patricia tries, l the maximum length of contained strings and c the size of alphabet used within the strings.

3.2.1 Memory Consumption

We have already observed that copying subtries in line 18 is not necessary and the same can be achieved by a delayed access to these subtries in succeeding recursion steps. With this observation, we can conclude the following memory consumption: Because of the recursion step in line 22 of Algorithm 2 and for each input patricia trie, only the nodes of a complete path from the root to a leaf node must be held in main memory. The maximum number of nodes in such a path from the root to a leaf node is l (but is typically much smaller for real-world data). However, for a more precise analysis we consider the number r of nodes of a complete path from the root to a leaf node. The maximum size of a single node is the maximum number of edges c of a single node multiplied with the maximum size p of the labels of the edges: $O(c \times p)$. For the merged final patricia trie, only one node must be temporarily held in main memory before it is written out. Hence, altogether the upper bound of the memory consumption is $O(n \times c \times p \times r)$, but is much less if we consider the properties of real-world data (see Section 3.2.4).

3.2.2 I/O Costs

Each node of the input patricia tries must be loaded only once into main memory under the condition that the nodes are held in main memory until the recursion (in line 23 of Algorithm 2) is left again. This means that not whole subtries must be held in main memory, but only the ancestor nodes of the currently processed node, which has low memory footprint even for large datasets. Each node of the merged patricia trie is stored only once (lines 8 to 10). Under the assumption that optimal I/O costs are *loading* the input tries only *once* and *storing* the resultant patricia trie only *once*, we have optimal I/O costs for merging patricia tries.

3.2.3 Runtime

We will first consider each non-trivial step in Algorithm 2 before we discuss the overall complexity.

As all edges in a patricia trie node are ordered according to their labels, not all edge labels of the current nodes in the input patricia tries must be compared with each other, but only a part of them. Hence, the check of the loop condition in line 2 as well as the determination of the set E in line 3 can be done in time linear to the number of input patricia tries: $O(n)$. As at most one edge of each input patricia trie is added to E, the size of E is at most n. The determination of the minimum first character c_1 of the edge labels in E in line 4 is therefore also $O(n)$. For the same reason, the number of edges in M as well as the determination of M in line 5 are $O(n)$ (but for real-world data often much smaller). Marking all edges of M in line 6 is obviously in $O(n)$. The determination of the longest common prefix in line 7 is restricted by the size of M and the maximal size p of the labels of the edges and thus is $O(n \times p)$. Note that by intelligent coding, lines 3 to 7 could be done by iterating only once through the current edges of the input patricia tries (but this does not affect the complexity in O notation).

The loop from line 12 to 19 is iterated at most n times. Checking if an edge with the currently considered postfix to a leaf node in line 13 already exists in the patricia tries

Y still to be merged, can be done in $O(p)$ (by choosing a good data structure for searching for the postfix, e.g. a hash table with the postfix as key). As already mentioned, copying subtries in line 18 is not necessary and the same can be achieved by a delayed access to these subtries in succeeding recursion steps. Hence, line 18 can be done in constant time (by storing a reference). Thus, the loop from line 12 to 19 is in $O(n \times p)$.

The loop from line 2 to 20 is iterated at most $O(c)$, as the single characters of the strings are from the alphabet with c different characters, and there are therefore at most c different common prefixes. Altogether the loop from line 2 to 20 is in $O(n \times p \times c)$.

We assume to have r recursion steps. The final merged patricia trie has at most x leaf nodes.

Hence, the overall runtime complexity is $O(n \times x \times c \times p \times r)$ and much less for real-world data (see Section 3.2.4).

3.2.4 Complexities for Real-World Data

For typical real-world data, c is not too large and can be seen as constant. Note that p and r depend on each other: as larger p is, as smaller is r and the other way around. Actually we can assume that l is in $O(p \times r)$ for typical real-world data.

Memory Consumption: For typical real-world data, the upper bound of the memory consumption is hence $O(n \times l)$.

Runtime: Assuming the properties of real-world data, the upper bound of the runtime is $O(n \times x \times l)$. Assuming that the sum L of the sizes of all strings is in $O(x \times l)$ for typical real-world data, we achieve a runtime complexity of $O(n \times L)$.

4 Experimental Analysis

We compare different variants of external merge sort: Our proposed approach *PatTrieSort*, *string merging*, *external merge sort* and *replacement selection*. The implementations of these sorting approaches are open source and publicly available as part of the LUPOSDATE project [10, 11].

4.1 Implementation Details

We varied the number of elements after which initial runs are swapped in *PatTrieSort*. For example, PatTrieSort with paramater 100 000 means that after 100 000 entries have been inserted into the patricia trie in main memory, this patricia trie is swapped to disk as initial run.

String merging is very similar to PatTrieSort. However, instead of writing the patricia trie as initial run, string merging writes the sorted list of strings as run. String merging writes the sorted list in a compressed way by leaving out common prefixes and writing instead an integer number for the number of characters in the common prefix with the previous entry. In experiments, this achieved a much better performance in comparison to writing the whole strings. In the merging phase, strings need to be merged instead of patricia tries. A heap is used for merging the strings of typically a huge number of initial runs. Again we have chosen as parameter the limit of entries after which an initial run is swapped to disk.

External merge sort just uses an in-memory sorting algorithm to generate the initial runs and stores the initial runs in the same way as string merging. This approach has also benefits for parallel sorting of data streams [7], not only for sorting large data sets on a local machine. We have done experiments with several in-memory sorting algorithms like quicksort, LSD radixsort (specialized to string merging) and several variants of merge sort, and present here only the results with the best one of our experiments, a parallel merge sort algorithm with 8 threads for merging. Also for external merge sort we have chosen as parameter the number of entries, which are sorted in main memory and afterwards swapped as initial run to disk.

Replacement selection uses a heap for increasing the size of the initial runs (in typical cases by a factor of 2 [9] on average). Replacement selection with the parameter x means here that the sorting algorithm reserves a heap of height x (containing therefore $2^{x+1} - 1$ entries) for generating the initial runs. We use an optimized heap, which avoids a bubble-up operation in the two succeeding operations retrieving the smallest item of the heap and adding a new item to the heap: After the root item is taken away, instead of the standard operations (i.e., moving the last leaf node to the root, performing a bubble-down operation, adding the new item as the last leaf node and performing a bubble-up operation), we directly insert the new item at the root, and perform just one single bubble-down operation. This optimization significantly speeds up the replacement selection by avoiding a bubble-up operation.

Although we have used input data with over 1 billion entries, we only use 1 merging step for all approaches.

4.2 Configuration of the Test System

The test system for the performance analysis uses an Intel Xeon X5550 2 Quad CPU computer, each with 2.66 Gigahertz, 72 Gigabytes main memory, Windows 7 (64 bit) and Java 1.6. We have used a 500 GBytes SSD for reading in the input data and writing out the runs. The input data is read asynchronous using a bounded buffer. For saving space on the SSD, we compressed the in-

Table 1: String Length Statistics for Sort Benchmark

Number of Strings:	1,000,000,000
Average String Length:	98
Standard Deviation of the Sample:	0
Minimum String Length:	98
Maximum String Length:	98

put data by using BZIP2 [23]. Decompression is done on-the-fly during reading in the input data. For all approaches, we write the final run to disk and iterate once through the final run. We have run the experiments ten times and present the average execution times.

4.3 Sort Benchmark

We have used the Sort Benchmark [21] for testing the performance of PatTrieSort in comparison to the other external merge sort algorithms. More concretely, we have used the input of PennySort [20], which is part of the Sort Benchmark, with 1 billion entries and measured the time for sorting these entries. The statistics of the string length (see Table 1) show that the dataset is homogeneous, as all strings have the same length of 98 characters.

It is obvious that larger initial runs lead to a faster merging phase. However, the generation of larger initial runs itself slows down performance: For PatTrieSort and string merging, inserting an entry into a larger patricia trie is slower. For external merge sort the in-memory sorting of more entries takes more time as well as for replacement selection inserting an entry into a larger heap.

Hence, there is an optimum for the amount of main memory reserved for initial run generation, after which sorting becomes slower again. For PatTrieSort and for string merging, this optimum is swapping after 8 million entries (see Figure 5), for external merge sort sorting of 2 million entries in-memory and for replacement selection using a heap of height 16 (with space for 131 071 entries).

Overall, PatTrieSort is the fastest among the external merge sort variants, followed by the traditional external merge sort, then string merging and finally replacement selection. Due to today's larger main memories, replacement selection does not save merging steps in comparison to the other sorting approaches, as all initial runs can be merged within one merging step. Merging a large number of patricia tries avoids comparing common prefixes, such that PatTrieSort is much faster than string merging and even gets ahead of external merge sort. PatTrieSort is at least 30 % faster than the other approaches (see Table 2).

The number of initial runs is the same and relatively

Table 2: Speed Comparison of PatTrieSort with the other approaches for Sort Benchmark (only best chosen parameters)

x	$\frac{\text{Time of } x}{\text{Time of PatTrieSort}}$
String Merging	1.38
External Merge Sort	1.3
Replacement Selection	1.52

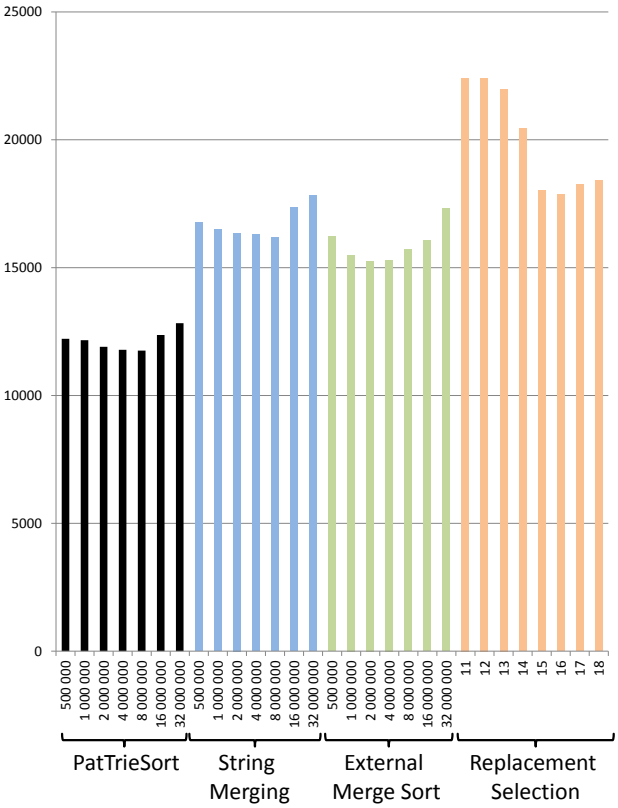

Figure 5: Results of Sort Benchmark in seconds

small for PatTrieSort, string merging and external merge sort (see Figure 6). For their optimal parameters, we have 500 initial runs for external merge sort and 125 initial runs for PatTrieSort and string merging. Figure 7 shows the number of initial runs for replacement selection in relation to their parameters. Replacement selection with the best chosen height 16 of the heap generates 3817 initial runs. Although the merge phase is much slower for replacement selection for the optimal parameters, using larger heap heights becomes slower, as the slower insertion of entries in these larger heaps outweighs faster merge phases.

Storing patricia tries as initial runs results in some overhead in comparison to storing a sequence of sorted strings. This is reflected in the I/O-costs: See Figure 8

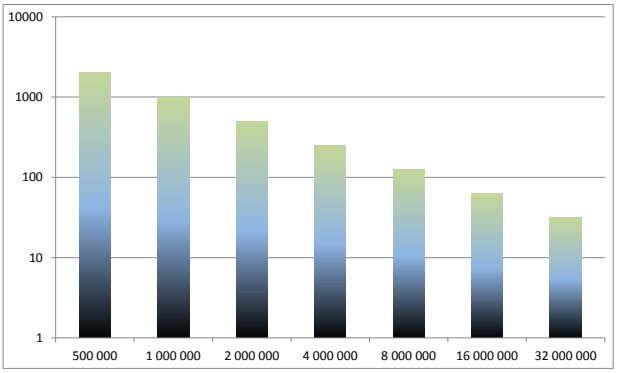

Figure 6: Sort Benchmark: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for PatTrieSort, string merging and external merge sort, which all have the same number of initial runs

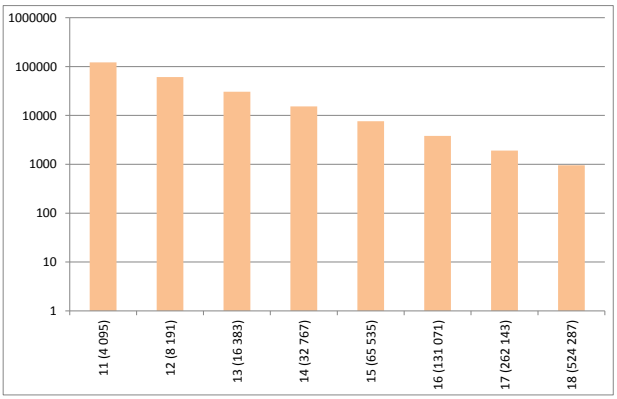

Figure 7: Sort Benchmark: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for replacement selection. h (s) at the x axis means reserving a heap of height h containing s ($= 2^{h+1} - 1$) entries.

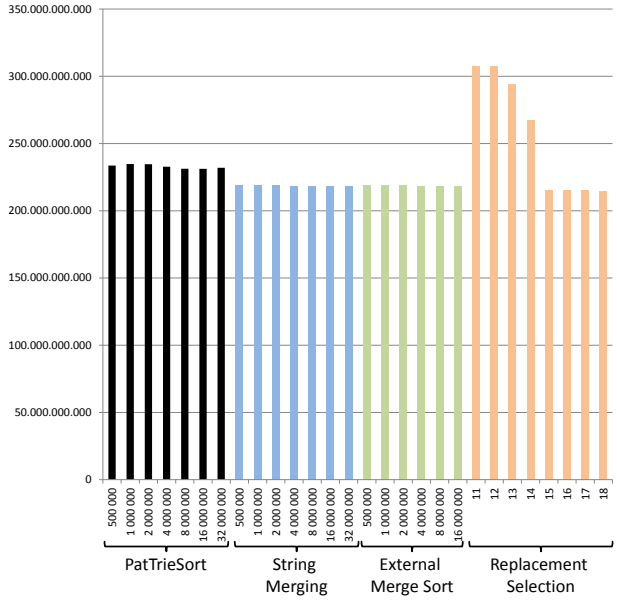

Figure 8: Sort Benchmark: Number of read bytes

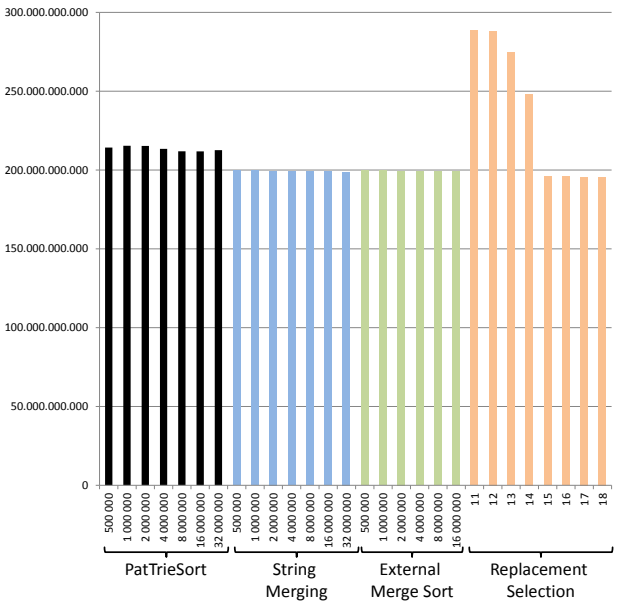

Figure 9: Sort Benchmark: Number of written bytes

for the number of read bytes during sorting the data of the Sort benchmark, Figure 9 for the number of written bytes and Figure 10 for the total I/O-costs as sum of the read and written bytes. Hence, the I/O-costs are not completely the dominant factor in our considered sorting approaches. Not surpisingly is the number of read bytes a little bit higher than the number of written bytes, and the I/O-costs are (only slightly) lower when consuming more memory. Replacement Selection has competitive I/O costs only with high memory consumption.

4.4 Billion Triples Challenge

The overall objective of the Semantic Web challenge is to apply Semantic Web techniques in building online end-user applications that integrate, combine and deduce information needed to assist users in performing tasks [13]. For this purpose, in last years large-scale datasets were crawled from online sources which are used by researchers to showcase their work and compete with each other. The *Billion Triples Challenge* (BTC) dataset of 2012 [2] consists of 1 436 545 545 triples crawled from different sources like Datahub, DBpedia, Freebase, Rest and Timbl. BTC is perfectly suited as example for large-scale datasets consisting of real world data with varying quality and containing noisy data.

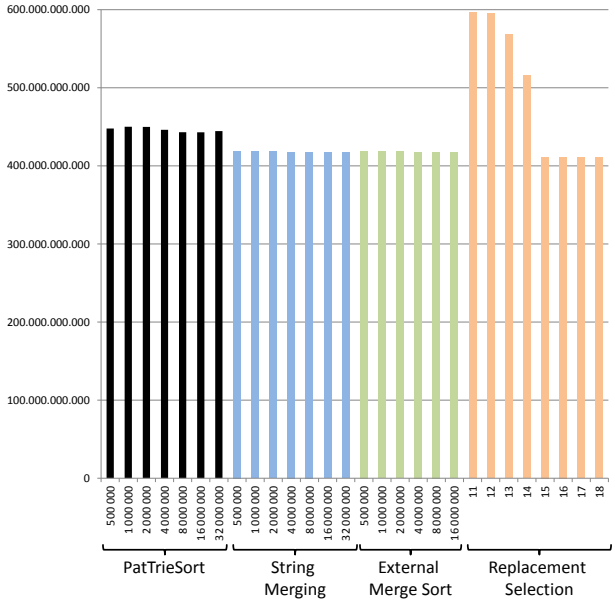

Figure 10: Total I/O costs of Sort Benchmark: Sum of read and written bytes

Table 3: String Length Statistics for Billion Triples Challenge of 2012

Number of Strings:	4,309,636,635
Average String Length:	55
Standard Deviation of the Sample:	73
Minimum String Length:	2
Maximum String Length:	335,085

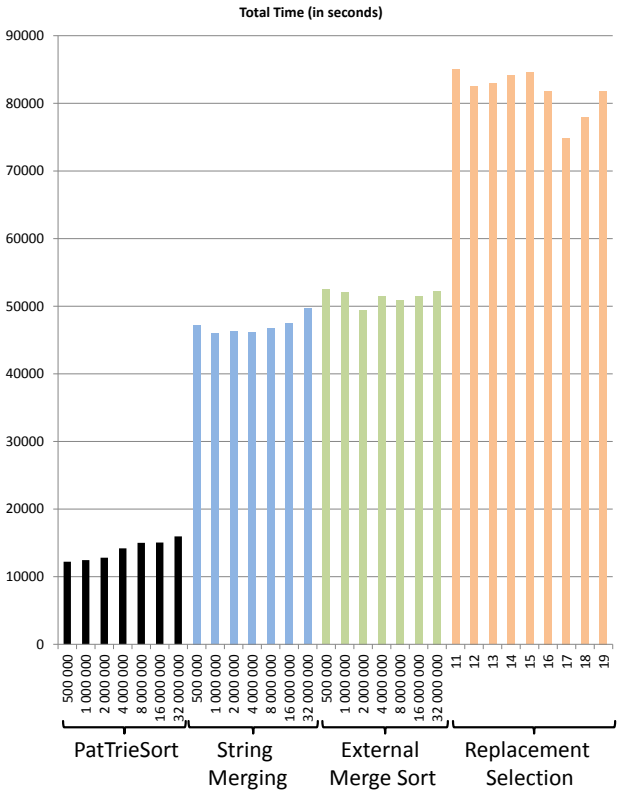

Figure 11: Results of sorting BTC data in seconds

Table 4: Speed Comparison of PatTrieSort with the other approaches for sorting BTC data (only best chosen parameters)

x	$\frac{\text{Time of } x}{\text{Time of PatTrieSort}}$
String Merging	3.76
External Merge Sort	4.05
Replacement Selection	6.13

We have sorted the string representations of the three components (called subject, predicate and object) of the triples of the BTC data resulting in 4 309 636 635 strings to be sorted. The string length statistics (see Table 3) reflects the noisy nature of the input: While having an average length of 55 characters, the lengths vary between 2 and 335 085 characters. Sorting is one basic step when constructing a dictionary for the BTC data, which maps each component of a triple to a unique number. Using unique numbers instead of the space-consuming string representations greatly reduces space used for indices on disk, improves performance and lowers the memory footprint [18]. There are many duplicates among the strings to be sorted: Only 14 669 339 unique strings are among the strings of the BTC data.

Semantic Web data consists mainly of Internationalized Resource Identifiers (IRIs) [6]. The syntax of IRIs corresponds to the one of Uniform Resource Locators (URLs), i.e., they consist of many characters and many of them have a long common prefix. Hence, patricia tries are the ideal data structure to space-efficiently store IRIs in main memory. For this reason our proposed approach PatTrieSort performs extremely well (see Figure 11): The string merging approach is already 3.76 times slower than PatTrieSort, external merge sort 4.05 times and replacement selection 6.13 times slower (see Table 4).

Real-world data does not have a regular structure like synthetic data as in the case of the Sort Benchmark has. Hence the development of the execution times dependent on the main memory consumption is not so regular as well. However, the tendencies remain similar to those with synthetic data.

Because of the huge number of duplicates and their space-efficient storage in Patricia Tries, the number of initial runs is much lower for the PatTrieSort and string merging approaches compared to external merge sort

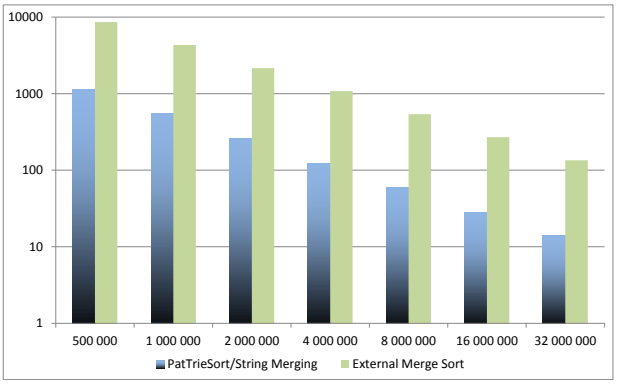

Figure 12: BTC: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for PatTrieSort, string merging and external merge sort

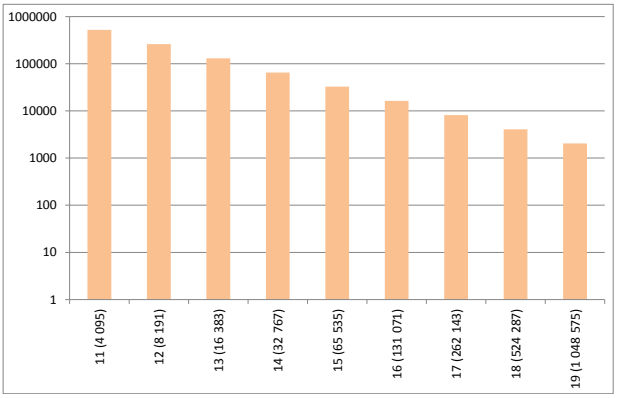

Figure 13: BTC: Number of initial runs (y axis in logarithmic scale) in relation to size used in main memory (x axis) for replacement selection. h (s) at the x axis means reserving a heap of height h containing s ($= 2^{h+1} - 1$) entries.

(see Figure 12) and replacement selection (see Figure 13). Not surprisingly the factor

$$f := \frac{\text{\#Initial runs of PatTrieSort and String Merging}}{\text{\#Initial Runs External Merge Sort}}$$

even increases from about 7.5 to 9.6 when more memory is reserved for generating the initial runs (see Table 5). This is another reason for PatTrieSort beating the other approaches, although it is not the dominant factor (as string merging has the same number of initial runs as PatTrieSort). However, in the results of the Sort benchmark (see Section 4.3) external merge sort was faster than string merging, for BTC data it is the other way around.

Many duplicates lower the I/O-costs of the PatTrieSort approach, as fewer bytes need to be transferred between

Table 5: BTC data: Comparing approaches by factor $f := \frac{\text{\#Initial runs of PatTrieSort and String Merging}}{\text{\#Initial Runs External Merge Sort}}$

Entries inserted before swapping	Factor f
500 000	7.5
1 000 000	7.8
2 000 000	8.3
4 000 000	8.7
8 000 000	9.1
16 000 000	9.6
32 000 000	9.6

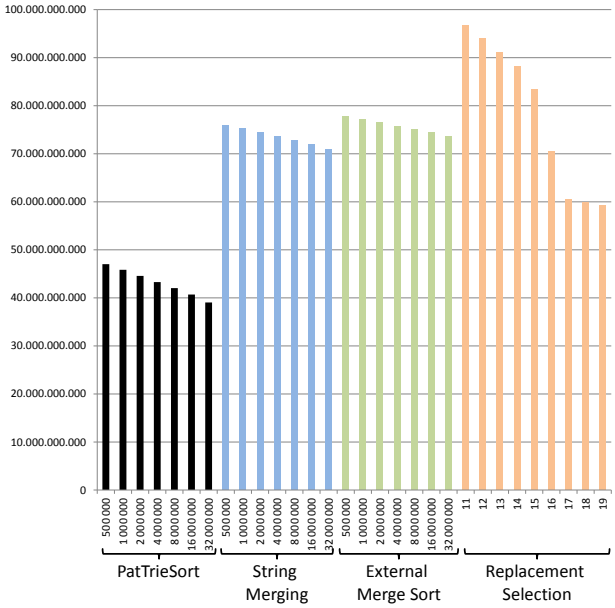

Figure 14: BTC: Number of read bytes

main memory and external storage: See Figure 14 for the number of read bytes during sorting the data of the BTC benchmark, Figure 15 for the number of written bytes and Figure 16 for the total I/O-costs as sum of the read and written bytes. For real data with more irregular properties higher memory consumption leads to much less I/O-costs. Replacement Selection has again competitive I/O costs only with high memory consumption, but achieves the second best I/O-costs (after the PatTrieSort approach).

5 Summary and Conclusions

Patricia tries are one of the most space-efficient data structures for strings. Considering the size of main memory as limit we can store much more strings in main memory than just adding strings to lists or arrays. Adding a string to a patricia trie is efficient as well as

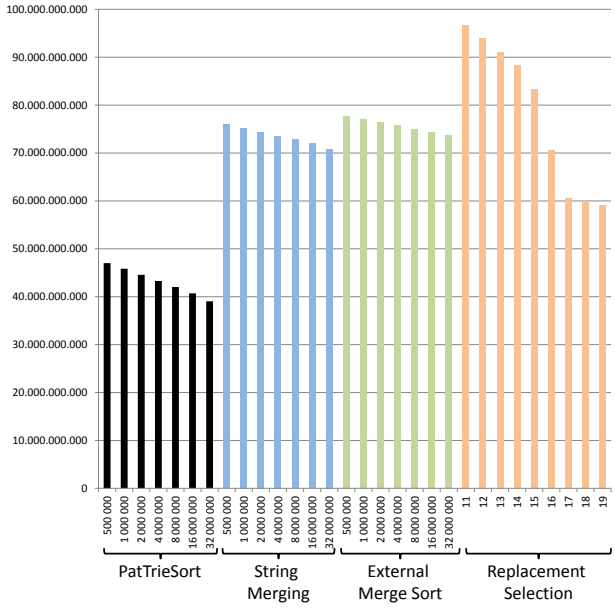

Figure 15: BTC: Number of written bytes

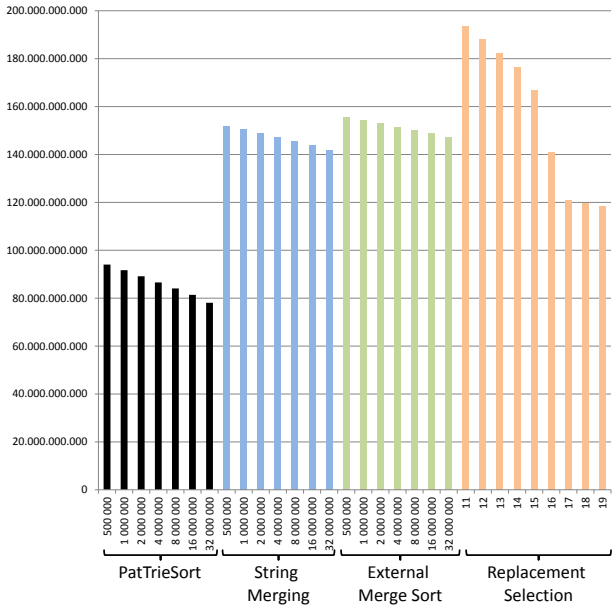

Figure 16: Total I/O costs of BTC Benchmark: Sum of read and written bytes

we can iterate over the contained entries of a patricia trie in sorted order by traversing the tree of the patricia trie. Hence, the first idea is to utilize patricia tries for generating large initial runs of an external merge sort variant.

In a second phase, external merge sort merges already sorted initial runs until only one sorted run remains (which is the result). If strings with many common prefixes are merged, these common prefixes are compared unnecessarily often. Patricia tries store common prefixes only once. Hence the second idea for sorting strings is to store the (initial) runs as patricia tries and to integrate a merging algorithm based on patricia tries into external merge sort.

The complexity analysis shows best results for the new merging algorithm for patricia tries in terms of memory consumption, I/O costs and runtime. While we have optimal I/O costs, the used memory is linear to the number of patricia tries to be merged multiplied with the maximum length of contained strings, and the runtime depends on the factor of number of patricia tries and the total size of all strings.

The performance analysis highlights the new sorting algorithm for strings as the best one in its family of external merge sort algorithms. Especially sorting Semantic Web data like the large-scale BTC data consisting of many string with common prefixes benefits extremely from merging patricia tries in external merge sort with speed-ups higher than 3.7 compared to other external merge sort variants.

ACKNOWLEDGEMENTS

This work is funded by the German Research Foundation (DFG) project GR 3435/9-1.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms*. Reading, Mass.: Addison-Wesley, 1983. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=0201000237

[2] Andreas Harth, "Billion Triples Challenge 2012 Dataset," http://km.aifb.kit.edu/projects/btc-2012/, 2012.

[3] R. Angrish and D. Garg, "Efficient string sorting algorithms: Cache-aware and cache-oblivious," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 1, 2011. [Online]. Available: http://www.ijsce.org/attachments/File/Vol-1_Issue-2/A022031211.pdf

[4] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, "On sorting strings in external memory (extended abstract)," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 540–548. [Online]. Available: http://doi.acm.org/10.1145/258533.258647

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3.*

ed.). MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[6] M. Duerst and M. Suignard, "Rfc 3987: Internationalized resource identifiers (iris)," *Internet Engineering Task Force (IETF)*, http://www.ietf.org/rfc/rfc3987.txt, 2005.

[7] Z. Falt, J. Bulánek, and J. Yaghob, "On parallel sorting of data streams," in *Advances in Databases and Information Systems*. Springer, 2013, pp. 69–77.

[8] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960. [Online]. Available: http://doi.acm.org/10.1145/367390.367400

[9] E. H. Friend, "Sorting on electronic computer systems," *J. ACM*, vol. 3, no. 3, pp. 134–168, 1956.

[10] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.

[11] S. Groppe, "LUPOSDATE Semantic Web Database Management System," https://github.com/luposdate/luposdate, 2015, [Online; accessed 26.1.2015].

[12] S. Groppe and J. Groppe, "External Sorting for Index Construction of Large Semantic Web Databases," in *Proceedings of the 25th ACM Symposium on Applied Computing, Vol. II (ACM SAC 2010)*. Sierre, Switzerland: ACM, Mar. 2010, pp. 1373–1380.

[13] A. Harth and S. Bechhofer, "A new application award - Semantic Web Challenge," http://challenge.semanticweb.org/, 2014.

[14] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[15] Linked Data, "Linked Data - Connect Distributed Data across the Web," 2015, [Online; accessed 27.1.2015]. [Online]. Available: http://www.linkeddata.org

[16] R. Miller, N. Pippenger, A. Rosenberg, and L. Snyder, "Optimal 2-3 trees," in *IBM Research Lab. Yorktown Heights*, NY, 1977.

[17] R. E. Miller, N. Pippenger, A. L. Rosenberg, and L. Snyder, "Optimal 2, 3-trees." *SIAM J. Comput.*, vol. 8, no. 1, pp. 42–59, 1979.

[18] T. Neumann and G. Weikum, "RDF3X: a RISC-style Engine for RDF," in *VLDB*, Auckland, New Zealand, 2008.

[19] T. Neumann and G. Weikum, "Scalable join processing on very large RDF graphs," in *SIGMOD*, 2009.

[20] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet, "Alphasort: A cache-sensitive parallel external sort," *VLDB J.*, vol. 4, no. 4, pp. 603–627, 1995.

[21] C. Nyberg and M. Shah, "Sort Benchmark Home Page," http://sortbenchmark.org/, 2015, [Online; accessed 27.1.2015].

[22] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[23] J. Seward, "bzip2 - bzip2 and libbzip2," http://www.bzip.org/, 2014.

[24] R. Sinha and J. Zobel, "Using random sampling to build approximate tries for efficient string sorting," *J. Exp. Algorithmics*, vol. 10, p. 2.10, 2005.

[25] R. Sinha and J. Zobel, "Efficient trie-based sorting of large sets of strings," in *Proceedings of the 26th Australasian Computer Science Conference - Volume 16*, ser. ACSC '03. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 11–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=783106.783108

[26] R. Sinha and J. Zobel, "Cache-conscious sorting of large sets of strings with dynamic tries," *J. Exp. Algorithmics*, vol. 9, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1005813.1041517

[27] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," in *VLDB*, 2008.

[28] J. Yiannis and J. Zobel, "Compression techniques for fast external sorting," *The VLDB Journal*, vol. 16, no. 2, pp. 269–291, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00778-006-0005-2

AUTHOR BIOGRAPHIES

Sven Groppe earned his diploma degree in Informatik (Computer Science) in 2002 and his Doctor degree in 2005 from the University of Paderborn. He earned his habilitation degree in 2011 from the University of Lübeck. He worked in the European projects B2B-ECOM, MEMPHIS, ASG and TripCom. He was a member of the DAWG W3C Working Group, which developed SPARQL. He was the project leader of the DFG project LUPOSDATE, an open-source Semantic Web database, and one of the project leaders of two research projects, which research on FPGA acceleration of relational and Semantic Web databases. His research interests include databases, Semantic Web, query and rule processing and optimization, Cloud Computing, peer-to-peer (P2P) networks, Internet of Things, data visualization and visual query languages.

Stefan Werner received his Diploma in Computer Science (comparable to Master of Computer Science) in March 2011 at the University of Lübeck, Germany. Now he is a research assistant/PhD student at the Institute of Information Systems at the University of Lübeck. His research focuses on multi-query optimization and the integration of a hardware accelerator for relational databases by using run-time reconfigurable FPGAs.

Dennis Heinrich received his M.Sc. in Computer Science in 2013 from the University of Lübeck, Germany. At the moment he is employed as a research assistant at the Institute of Information Systems at the University of Lübeck. His research interests include FPGAs and corresponding hardware acceleration possibilities for Semantic Web databases.

Christopher Blochwitz received his M.Sc. Diploma in Computer Science in September 2014 at the University of Lübeck, Germany. Now he is a research assistant/ PhD student at the Institute of Computer Engineering at the University of Lübeck. His research focuses on hardware acceleration, hardware optimized data structures, and partial reconfiguration of FPGAs.

Thilo Pionteck is an associate professor for Organic Computing at the Universität zu Lübeck, Germany. He received his Diploma degree in 1999 and hIs Ph.D. (Dr.-Ing.) degree in Electrical Engineering both from the Technische Universität Darmstadt, Germany. In 2008 he was appointed as an assistant professor for Integrated Circuits and Systems at the Universität zu Lübeck. From 2012 to 2014 we was substitute of the Chair of Embedded Systems at the Technische Universität Dresden and of the Chair of ”Computer Engineering” at the Technische Universität Hamburg-Harburg. His research work focus on adaptive system design, runtime reconfiguration, hardware/software codesign and network-on-chips.