Sorting Hierarchical Data in External Memory for Archiving

Ioannis Koltsidas
School of Informatics
University of Edinburgh
i.koltsidas@sms.ed.ac.uk

Heiko Müller
School of Informatics
University of Edinburgh
hmueller@inf.ed.ac.uk

Stratis D. Viglas
School of Informatics
University of Edinburgh
sviglas@inf.ed.ac.uk

ABSTRACT

Sorting hierarchical data in external memory is necessary for a wide variety of applications including archiving scientific data and dealing with large XML datasets. The topic of sorting hierarchical data, however, has received little attention from the research community so far. In this paper we focus on sorting arbitrary hierarchical data that far exceed the size of physical memory. We propose HERMES, an algorithm that generalizes the most widely-used techniques for sorting flat data in external memory. HERMES efficiently exploits the hierarchical structure to minimize the number of disk accesses and optimize the use of available memory. We extract the theoretical bounds of the algorithm with respect to the structure of the hierarchical dataset. We then show how the algorithm can be used to support efficient archiving. We have conducted an experimental study using several workloads and comparing HERMES to the state-of-the-art approaches. Our results show that our algorithm (*a*) meets its theoretical expectations, (*b*) allows for scalable database archiving, and (*c*) outperforms the competition by a significant factor. These results, we believe, prove our technique to be a viable and scalable solution to the problem of sorting hierarchical data in external memory.

1. INTRODUCTION

Sorting has always been important in data management. Its usefulness is even greater for database systems as sorting plays a significant role in a number of key query processing algorithms, including join evaluation, duplicate elimination, and aggregation, to name a few. The vast majority of algorithms, however, focuses on flat datasets; the problem of sorting hierarchical data has, surprisingly enough, received little attention from the research community. This is due to the relational data model being inherently flat. The need for sorting hierarchical data has re-emerged in the context of managing scientific data archives, which tend to be largely hierarchical, complicated in structure, and quite voluminous. In this paper we present the *Hierarchical ExteRnal MErgeSort* (HERMES) algorithm for sorting hierarchical data in external memory. The algorithm takes into account the hierarchical structure and by exploiting it, it is able to efficiently sort large datasets while minimizing disk I/O and, at the same time, using a minimal amount of main memory. As an example, one is able to sort two gigabytes of hierarchical and highly complex data using only five megabytes of main memory in a little over five minutes. Moreover, the algorithm can scale to efficiently sort petabytes of data performing minimal I/O while only using one gigabyte of physical memory.

Archiving scientific data. The main impetus for this work is managing scientific data for archiving purposes. Scientific data sources on the Web play a major role for ongoing research efforts. Annotated protein databases like UniProt [8], or sequence databases like EMBL [11], are the primary sources of information in, *e.g.*, selecting targets for conducting biological experiments, or in pharmaceutical research. As is the case in any kind of research, reproducibility of results is of paramount importance. Problems arise due to the dynamic nature of scientific databases: they continuously change as new results become available. Pitfalls include the identification of erroneous entries in a database, and therefore their modification, which results in invalidating scientific results that have used the erroneous entries as input. In addition, as research progresses, more accurate results are generated through improved experimental methods. It is common practice for scientific database providers to overwrite existing database states when changes occur and publish new releases of the data on the Web on a regular basis. Failure to archive earlier states of the data may lead to loss of scientific evidence, as the basis of findings may no longer be verifiable.

Scientific data is predominantly kept in well-organized hierarchical data formats. To support versioning, in [5] the authors propose an archiving approach that efficiently stores multiple versions of hierarchical data in a compact archive. Version numbers denote time and become a first-class citizen of the process: time is added as an extra attribute to the data being archived. To generate a new version of the archive the authors propose *nested merge*: multiple versions are *merged* on the time attribute, with the archiver storing each element only once in the merged hierarchy to reduce storage overhead. An archived element is annotated with timestamps representing the sequence of version numbers in which the element appears. By merging elements into a single data structure the archiver is able to retrieve any version from the archive in a single pass over the data.

Example 1.1: In Figure 1 we see an archive A_{1-2} containing two versions of data and an incoming version V_3. For ease of presentation, we assume that nodes are compared on their values. Nodes in the archive are annotated with their version number (denoted by t in the figure); version numbers act as timestamps representing the points in time that a node is present in the archive. Nodes without a timestamp are assumed to inherit the timestamp of their parent. Corresponding elements are connected by dotted edges.

Starting from the root, corresponding nodes in V_3 and in A_{1-2} are merged recursively. When a node y from V_3 is merged with a node x from A_{1-2}, the timestamp of x is augmented with the new version number (*e.g.*, the root of the archive and node A). The sub-

PVLDB '08, August 23-28, 2008, Auckland, New Zealand

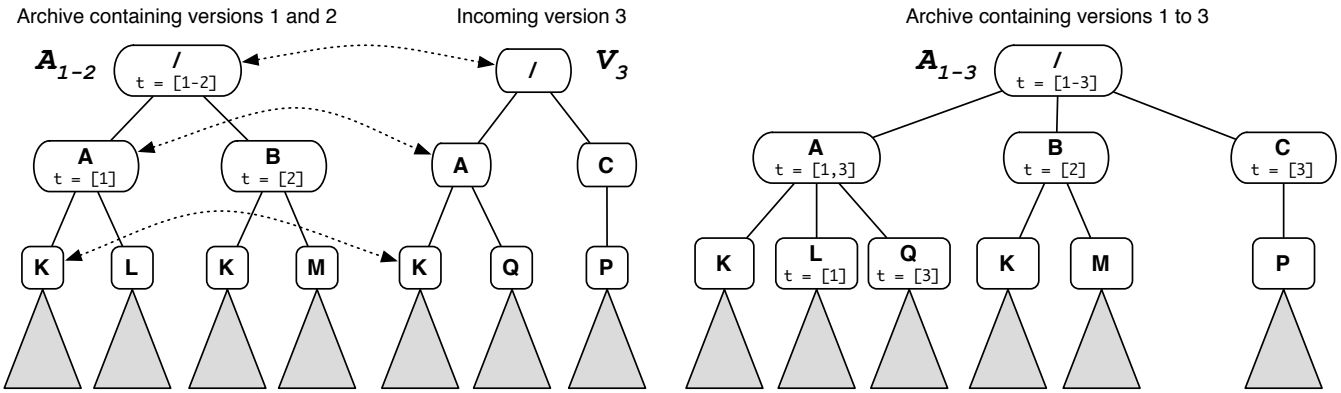


Figure 1: Merging an incoming version into an existing archive.

trees of nodes x and y are then recursively merged by identifying correspondences between their children. Nodes in V_3 that do not have a corresponding node in A_{1-2} are added to A_{1-3} with the new version number as their timestamp (*e.g.*, node Q). Nodes in A_{1-2} that no longer exist in the current version V_3 have their timestamp terminated, *i.e.*, these nodes do not contain the new version number (*e.g.*, node B). The process is repeated for all levels. □

With serialized hierarchical data formats, like XML, one usually traverses the data depth-first. The problem with nested merge as described in [5] is that it does not manifest this natural access pattern. To identify correspondences between children of merged nodes, one must process complete subtrees. Thus, numerous passes over the data may be required. If, however, the nodes of the datasets are ordered on their keys the situation greatly improves. Assuming an ascending order, as shown in Figure 1, whenever two nodes x and y are to be merged, one can sequentially scan the children and compare their key values. The child with the smaller value is output to the new archive, after having been annotated with the proper timestamp. This ensures a total ordering among all children of any node. This process ensures that the archive is always sorted on node keys. More importantly, only the incoming version has to be sorted before nested merge. As the new version may be comparable in size to the archive, sorting is a salient operation.

Change Detection. Apart from archiving, sorting different versions of hierarchical datasets enables efficient change detection. This is useful to systems that support incremental query evaluation and trigger condition evaluation. Existing approaches to change detection in XML documents (*e.g.*, [6, 7, 18]) operate on unsorted documents and only work in main memory. However, an algorithm similar to *nested merge* can be employed to efficiently spot differences. With the input documents sorted, change detection can be supported for datasets larger than the size of main memory.

Sorting hierarchical data. The complexity of sorting large hierarchical datasets is shown to be below that of sorting flat data. This is due to the smaller number of possible sorting outcomes, as in any sorted result the initial parent-child relationships from the original data have to be retained. For example, there is no need to compare nodes in different subtrees of the dataset, or located at different levels of the hierarchy. As with any external memory algorithm, the major challenge is to reduce the overall number of I/O operations.

The common approach to sorting large datasets is external mergesort and its variations [14]. External mergesort splits the dataset into multiple *runs* that are sorted in main memory during a single pass over the data. Runs are then merged to generate the sorted output. Sorting hierarchical data is, however, not straightforward; the hierarchical structure has to be retained and each sorted run has to represent a proper hierarchy itself. An obvious approach would be to "*flatten*" the data by writing the complete set of root to leaf paths to a file and then sorting the entries in the file using standard external mergesort. As shown in [16], this approach does not exploit the hierarchical structure and is very inefficient in terms of memory, storage space and processing power. Bottom-up approaches [16] for sorting hierarchical data, on the other hand, operate by splitting the input in complete subtrees that are sortable in main memory. These subtrees are stored as sorted runs in separate files. Once the children of each node are sorted, the data is output by reading the sorted subtrees from the run files. This employs a random access pattern: though each run will be sequentially scanned, entire runs will be read in a different order than the one they were generated. Such approaches do not perform well on the high-branching, widespread structure of scientific datasets. As an example, the current release of the EMBL Nucleotide Sequence Database [11] (Release 93, December 2007) has over 100 million entries below a single root node. The average size of each entry is four kilobytes. Therefore, during output, a large number of small files will have to be accessed in random order, which penalizes I/O performance.

Contributions. In what follows we present our approach to sorting arbitrary hierarchical datasets. Our main contributions are:

- We propose an algorithm that generalizes the most widely-used techniques for sorting flat data in external memory. The algorithm efficiently exploits the hierarchical structure in order to minimize the number of disk accesses and optimize the utilization of available memory.
- We extract and verify the theoretical bounds of the algorithm with respect to the structure of the hierarchical dataset.
- We present how the algorithm can be applied for archiving databases and the performance gains it results in.
- We have implemented the algorithm and conducted a detailed experimental study of its performance for both archiving and stand-alone sorting; we include a comparison to the state-of-the-art approaches. Our results show that our algorithm outperforms the competition by a large margin and its performance is the one expected from its theoretical analysis.
- Though motivated by sorting scientific datasets for archiving purposes, the algorithm is general and efficient enough to be applicable in a variety of problems where the need for sorting arbitrary hierarchical datasets arises.

The rest of this paper is organized as follows. Related work in the area is presented in Section 2. Our algorithm for sorting arbitrary hierarchical datasets is given in Section 3. The algorithm's theoretical analysis is presented in Section 4, while its use in archiving is presented in Section 5. In Section 6 we present the results of a detailed experimental study. Finally, we conclude and present our future work directions in Section 7.

2. RELATED WORK

Sorting is a fundamental computing problem and as such it has received considerable attention. Departing from internal memory

implementations, the basis of most external memory sorting algorithms over flat, record-based datasets is external mergesort. Various extensions have been proposed over time, with [14] presenting an extensive study of most external sorting techniques, while [13] presents the details of implementing external mergesort as part of a relational database engine. There have been numerous proposals for improving the algorithm's performance, ranging from increasing its internal sorting efficiency, to enhancing its CPU utilization, or to its parallelization.

In [20] the authors propose placing blocks from different runs in consecutive disk addresses to reduce the seek overhead during the merging phase (at the expense of additional seek cost during run creation). They also study reading strategies, like forecasting and double buffering, and propose a read planning technique. The latter uses heuristics to precompute the order in which records will be read from disk during merging. It then utilizes this order to reduce seek overhead, based on knowledge of the physical location of the blocks on the medium. These improvements can be almost verbatim applied to our algorithm, provided they are adapted to hierarchical data (see Section 4.3 for a discussion on how this can be achieved).

Moving on to strictly hierarchical data models, the most widespread one is XML. It was used as the serialization protocol for archiving scientific data and the definition of the nested merge operation [5], which provided the motivation for the development of HERMES. Similar concepts were provided in [17] and [19] where the semantics of generalized XML tree merging were defined (albeit in different ways). Regardless of the exact semantics, efficient merging implementations depend on having their inputs sorted, and therefore our proposal is immediately applicable. Furthermore, XML query languages like XPath [2] and XQuery [3] provide an *order by* clause that may be used in conjunction with a DTD to completely sort XML documents. However, the specification does not mention any particular implementation. We believe that our algorithm is one such possible implementation to be used by XML query engines.

The XML Toolkit (XMLTK) provides a tool named XSort for sorting XML documents [1]. XSort allows the specification of the context nodes the subtrees of which should be sorted. For each context node multiple XPath expressions identify the actual elements to be sorted. Only user-specified elements are sorted and the subtrees of these elements are not sorted recursively. Sorting proceeds by generating a global key for each element to be sorted. It then uses a standard external mergesort algorithm to sort elements based on the value of this global key. XSort does not exploit the hierarchical structure of the data. Indeed, it might not be possible to sort the entire document without making multiple calls to XSort. By collapsing hierarchical data to their flat counterparts, the hierarchy reconstruction step is left to the user. Our algorithm does not impose such restrictions.

The most relevant piece of work we are aware of, and the state-of-the-art in sorting XML datasets, is NEXSORT [16]. The NEXSORT algorithm takes into account the properties of hierarchical datasets and consists of two phases: sorting and output generation. During the sorting phase, NEXSORT scans the input document depth-first, detects complete subtrees, and decides, based on a user-given threshold, whether to sort these subtrees in main memory or not. Only subtrees of size no less than the specified threshold are sorted and stored on disk as a sorted run. Sorted subtrees are replaced in the tree by just their root and a pointer to the sorted run stored on disk. Conceptually, NEXSORT processes the input document bottom-up, collapsing subtrees into their roots until only the root of the entire tree remains. In the output phase, NEXSORT performs a depth-first traversal of the collapsed tree to generate the final sorted document. Generated sorted runs need to be accessed in a random fashion during the merging phase, therefore penalizing I/O. Furthermore, the choice of threshold is a critical part for the performance of NEXSORT making performance dependent on the structure of the document. For documents like EMBL [11], where only a few subtrees are large, this approach is very inefficient. Our algorithm, by making efficient use of compression and carefully laying out runs on external memory, is able to achieve much better I/O performance, as we shall see in Section 6.

3. SORTING HIERARCHICAL DATA

We now present our algorithm: *Hierarchical ExteRnal MErge-Sort* – HERMES, an adaptation of external mergesort for hierarchical data. HERMES runs in two phases: (*a*) first, the hierarchical document is "vertically" split into sorted runs on disk; (*b*) then, the runs are iteratively merged into greater ones until the final sorted output is generated. HERMES extensively exploits the fact that one needs to perform key comparisons only for nodes having the same parent node (*i.e.*, siblings). Nodes belonging to different subtrees do not need to have their keys compared. This enables us to apply *local* replacement selection for every in-memory node.

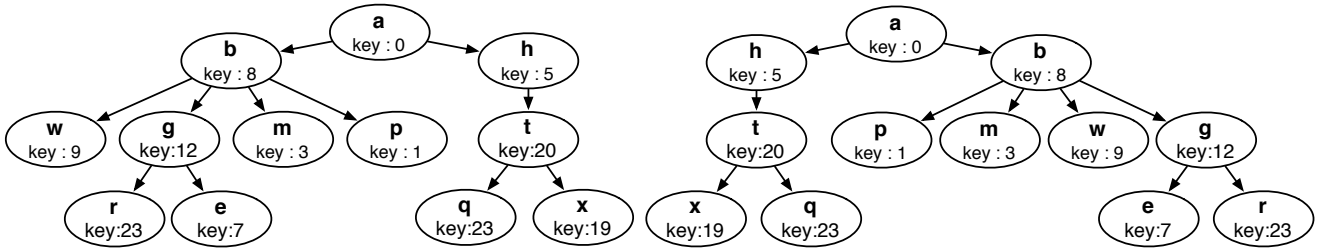
3.1 Sort Keys

A hierarchical dataset is a tree whose nodes have an *identifier* (or label), a *type*, and an optional *value* (or payload). To sort hierarchical datasets we have to specify a sorting criterion for nodes. This criterion may include the node label, its value, a combination of the two, or a well-defined subset of the subtree rooted under that node.[1] We assume a hierarchical sort key specification (*key specification* for short) similar to [4, 5]. The key specification K is a set of *key definitions* $k = (Q,S)$, where Q is an absolute path of node labels and S is a *sort value expression*. We assume that path Q of key definition k is unique among all elements in K. We distinguish between *keyed* and *unkeyed* nodes. Keyed nodes have a path that matches path Q of a $k \in K$. The sort expression S determines the values on which nodes having path Q are sorted. We refer to these values as *sort keys*.

For sorting, every node has an additional sort key attribute. Given a key specification K, we assign each keyed node its key value in a preprocessing step as described in [5]. For unkeyed nodes, the sort key is the maximum value of the sort domain, followed by a placeholder denoting its position in the input. For each node n, let its *local key* (or simply *key*) $k(n)$ be the value of its sort key attribute. The key is the *local ordering criterion* by which we decide the rank of a node with respect to its siblings (of the same type, if different types are present). To sort the entire tree, one has to recursively sort the children of every non-leaf node, starting from the root. We assume the local key of a node to be unique among all of its siblings of the same type (we can always ensure uniqueness by appending the position or the identifier of the node to the local key). Let the *absolute* key $a(n)$ of a node be the concatenation of the local keys of all its ancestors: the concatenation of the local keys for all nodes from the root of the tree up to n. Therefore, the absolute key for a node is given by its unique path from the root if we replace each node label in the path with the corresponding local key value.

Using absolute node keys we can define the *total ordering criterion* for all nodes in the tree. Two nodes x and y are *equal iff* they have the same absolute key. In all other cases, and for any two nodes x and y, their absolute keys, $a(x)$ and $a(y)$ respectively, share a proper prefix (at the very least the value of the local key of the

[1]For sorting trees that exceed main memory, we assume that, though arbitrarily long, node keys do not exceed the size of available memory.



(a) Unsorted (b) Sorted

Figure 2: A hierarchical dateset annotated with the local key values for its nodes

root node). Suppose that the common prefix of $a(x)$ and $a(y)$ is of length c. Then x is *less than* y *iff* $a_{c+1}(x) < a_{c+1}(y)$,[2] or $a(x)$ is of length c, where a_i denotes the i^{th} local key component of $a(x)$. Correspondingly, node x is *greater than* y *iff* they are not equal and x is not less than y. Naturally, any node is considered less than any of its children to preserve hierarchical relationships. The definition of local keys for unkeyed nodes means that all unkeyed nodes will follow their keyed siblings in input order in the total ordering. The children of unkeyed nodes may be keyed, in which case we need to recursively sort them. A tree is sorted if the children of all nodes are sorted on their local keys (if they are keyed).

Example 3.1: A hierarchical dataset is shown in Figure 2(a). Each node is annotated with its local key. The absolute key for node `r` is `/0/8/12/23`; for node `t`, it is `/0/5/20`. In Figure 2(b), the same dataset is shown sorted. □

Note that the local key value of a node may be a subtree. In such cases we serialize the subtree into a string and perform string comparison, when comparing keys. Also, sibling nodes may be of different type and therefore keyed on different sort value expressions. To address this, we prepend every local key with the type of the node. We then define a total order on node types to distinguish between multiple types of key and for grouping siblings of the same type in the output tree. If keys are not unique, two siblings may have the same key value even though they are different nodes. In such cases, we append the position of the node to its key. The same applies if we want to preserve the order of unkeyed nodes: we set their local key to be their position, prepended by a symbol that the node is unkeyed.

In the case that complex keys are present (*i.e.*, if a node has two or more key paths), the local key of a node consists of the concatenation of all its key values (possibly prepended by their path or type) separated by a special character. For instance, if a node is keyed by the values of attributes `firstname` and `lastname` (possibly found in its payload), its local key can be recorded as `firstname:john,lastname:smith`. When comparing two such nodes, corresponding components of the complex key can be identified and be compared. If the key of a node n has q key paths, then the value of the key value for the i-th key path is denoted as $k(n)[i]$. For two such nodes n_1 and n_2, of the same type, we define n_1 to be less than n_2, *i.e.*, $k(n_1) < k(n_2)$, when for some j with $1 \leq j < q$, $n_1[i] = n_2[i]$ for all $1 \leq i < j$ and $n_1[i+1] < n_2[i+1]$.

3.2 The HERMES Algorithm

During the first phase of the algorithm, we create sorted runs using a hierarchy-aware adaptation of replacement selection. Our goal is to exploit the hierarchical structure. This can reduce the number of possible sorting outcomes from $N!$ (for a flat file of N records) to $(F!)^{\lfloor (N-1)/F \rfloor} \cdot ((N-1) \mod F)!$ for a tree of N nodes and a maximum fan-out of F [16]. Sorted runs contain the keys in a compressed form (to eliminate redundancy). During the second phase, sorted runs are merged to create the sorted output.

3.2.1 Standard external mergesort

External mergesort uses replacement selection to create the initial runs. For flat data, replacement selection reads the input record by record and starts filling a *min* priority heap. When the heap is full, the first (and thus smallest) item is removed and written to the first run. Then, it is replaced in the heap by the next record from the input. The (new) smallest item in the heap is examined. If it has a key greater than the one just written, it is written to the current run and replaced in the heap by the next record from the input. Otherwise, the item cannot be included in the current run and is therefore marked for the next run. Marking a record implies placing it at the end of the heap and considering it greater than unmarked ones during heap comparisons. At some point, all records in the heap will have been marked for the next run. Then, the current run will be closed and the algorithm will start creating the next run. Repeating this process until the input has been exhausted yields runs that contain sorted subsets of the input. In the next phase the runs are merged using a priority heap. The priority heap is initially filled with the first item of each run and the smallest item is selected and written to the output run. Subsequently, it is replaced by the next record of the same run and the process continues. If we use one memory page for each run being read and one for the output run, it is possible that the amount of available memory is not sufficient for all runs to be simultaneously processed. In this case multiple merge levels are necessary. At each such level l, as many runs of level l as the memory can accommodate are merged into a single run of level $l+1$, until only one run is obtained at some level l_n. This run contains all the records in sorted order.

3.2.2 Hierarchy-aware replacement sort

For hierarchical data, one needs to sort the children of the root of the tree on their key values, and then recursively repeat this process for the root's children until the whole tree is sorted. The children of a node can be sorted, however, independently of other node keys in the tree. Thus, sorting in the traditional sense, *i.e.*, ordering a group of items on their values, only needs to be performed "locally" at a node. Our algorithm is based on employing replacement selection using a priority heap locally at a node to produce a sorted run of its children.

We use a tree serialization protocol much like XML, *i.e.*, the tree is stored in a depth-first manner: the start and end of a node are specified with starting and ending tags and all its children lie within. All node-specific information (*i.e.*, its type, name, local key annotation (if any), and payload) follow its starting tag. The input tree is thus retrieved in depth-first fashion and for each node we take appropriate action. The output is a file that contains the tree in the same serialized format, except that children of all nodes are ordered by their key values (or by their position in the input tree, if they are not keyed).

The algorithm operates on an in-memory representation of tree

[2]Here, "$<$" denotes an arbitrary ordering of local key values. If the key values are character strings, this is their lexicographical order; if the key values are numbers, "$<$" corresponds to arithmetical comparison.

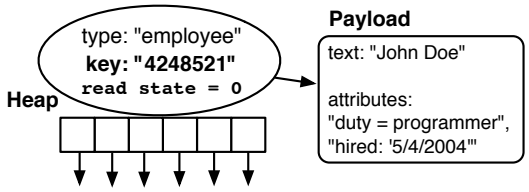


Figure 3: An example of a SortNode

nodes, termed SortNodes. A SortNode holds: (*a*) the type of the node (and possibly its position in the document), (*b*) the key of the node which can be constructed when the node is first read into memory,[3] (*c*) a pointer to the payload of the node (the payload might be text associated with the node, an attribute value, *etc.*), (*d*) an array of pointers to the SortNodes corresponding to the children of the node, which is used as a priority heap and is referred to simply as *heap* hereafter, and (*e*) one bit called *read state bit*. The read state bit is set to 0 when the start of the node is read and is set to 1 when the end of the node has been encountered (*i.e.*, when the whole subtree rooted at the node has been fully read). The payload of a node can be reached using the pointer mentioned above, but plays no role during sorting. For this reason it is copied elsewhere in memory; further discussion of payloads is deferred to Section 4.2. A typical SortNode is shown in Figure 3.

Sorting. The sorting phase of our algorithm for the creation of the initial runs is shown in Figure 4. The input is read tag by tag. Stack *inputPath* holds the SortNodes of all ancestors of the last read node (initially it contains the SortNode for the root). Stack *outputPath* holds the SortNodes for the ancestors of the last output node. Traversal of the input starts at the root node. When the start tag of a node is encountered, a SortNode is created in main memory. If the available memory is full, one or more nodes need to be written to the current run to make room for the newly read node (lines 7-11). The nodes to be output are the nodes that form the subtree rooted at a node that (*a*) has the least absolute key among all subtrees in the tree, and (*b*) has been completely read from the input. The root of this subtree is located by findLeastKey and output by outputSubtree; both procedures will be explained later on. Nodes are output to the current run until enough memory has been freed for the new node.

When enough memory becomes available (lines 12-16), we identify the local key for the node and create a SortNode for it (with its read state bit set to 0). The new SortNode is inserted as a child to its parent's SortNode: it is pointed to by an element of the heap array of its parent's SortNode. This is performed by procedure insertNode of Figure 4. If the first place in the heap of *inputPath*.top() is free, then a pointer to the new SortNode is inserted there; otherwise the pointer is inserted at the end of the heap. If the first place of the array is free, it means that the array had the heap property at some point in time, after which its first element was removed (*i.e.*, it was output to a run). Thus, placing the new SortNode in the first place of the array enables us to maintain the heap property with a single call to heapify. We use the terminology of [9] with respect to heap operations: buildHeap constructs a heap from an array, while heapify (0) maintains the heap property of the array when its first element is removed and substituted by a new one. Once the SortNode is connected to its parent, it is pushed to the top of *inputPath*, so that the top of *inputPath* always holds the SortNode for the currently processed node. When the end of a node is encountered, the top of *inputPath* corresponds to its SortNode. The SortNode's read state bit is set to 1, marking that the subtree rooted at the node has been fully read; the *inputPath* stack is then popped (lines 17-20).

[3]If the key of a node is the value of one of its descendants, we assume the node has been annotated with this value in a previous annotation step, as in [5], so that its key can be found locally.

Algorithm 1: HERMES - Sorting Phase (Tree *T*)

```
1.  Stack inputPath
2.  Stack outputPath
3.  while (the input has not been exhausted)
4.     Read the next tag from T
5.     if (a start tag was encountered)
6.        Read the new node n
7.        while (not enough memory left for n)
8.           SortNode min = findLeastKey ()
9.           outputSubtree (min)
10.          free(min)
11.       end while
12.       Extract the local key of n
13.       Create a SortNode sn for n
14.       sn.readstate = 0
15.       insertNode (inputPath.top(), sn)
16.       inputPath.push(sn)
17.    else /* an end tag was encountered */
18.       inputPath.top().readstate = 1
19.       inputPath.pop()
20.    end if
21. end while

Procedure insertNode (SortNode p, SortNode sn)
22. if (the first place in p.heap is free)
23.    insert sn at the first place of p.heap
24. else
25.    insert sn at the end of p.heap
26. return

Procedure outputSubtree (SortNode root)
27. Write root to current run
28. sort (root.heap)
29. for each (SortNode cn child of root)
30.    outputSubtree (cn)
31. free(root)
32. return
```

Figure 4: The sorting phase of HERMES

The path to the parent node of the most recently output subtree is maintained in stack *outputPath* (initially this stack only contains the SortNode for the root of the tree). For each node in *outputPath*, the algorithm stores the key value of its last output child. This enables heapify to identify children nodes that have key values less than that key value and mark them for the next run by moving them to the end of the heap (*i.e.*, by considering them "greater" than their siblings with keys greater than the last output key). Note that the memory space required for storing this information is equal to the size of the *outputPath* stack. To keep the presentation as simple as possible, we omit details of how this information is implemented. It suffices to note that when the array of a SortNode has the heap property, the children of the node that should be written to the next run are all placed at the end of the array.

Subtree output during sorting. We now turn to procedure findLeastKey (shown in Figure 5), which selects the next node (or subtree) to be output when memory is full. The algorithm locates the root of the subtree that (*a*) has the least absolute key that is greater than the last key output to the current run, and (*b*) has been fully read. When findLeastKey is called, the top of *outputPath* holds the parent of the previously output node. Using getLeastChild, we obtain the minimum child of the top of *outputPath*. Procedure getLeastChild, shown at the bottom of Figure 5, operates on some SortNode *p*. It initially checks if the first element in *p*'s heap array is free. This happens when no new child of *p* was read since the last time a child of *p* was output; then, the last element of the array is brought to the first position. Otherwise, a new node has been inserted as the first element of *p*'s heap array. In both cases, heapify

```
Procedure: findLeastKey ()
1.  SortNode sn = getLeastChild (outputPath.top())
2.  while (all children of sn have been written)
3.      free(sn)
4.      sn = getLeastChild (outputPath.top())
5.      outputPath.pop()
6.  end while
7.  while (sn == null)
8.      outputPath.pop()
9.      sn = getLeastChild (outputPath.top())
10. end while
11. if (outputPath is empty)
12.     Start a new run
13.     outputPath.push(root of the tree)
14.     Write the root of the tree to the new run
15. end if
16. while (the end of sn has not been read)
17.     sn.buildHeap()
18.     outputPath.push(sn)
19.     Write sn to current run
20.     sn = the first item of sn.heap
21. end while
22. return sn

Procedure getLeastChild (SortNode p)
23. if (the first place in p.heap is free)
24.     move the last element of the heap to the front
25. p.heapify (0)
26. if (all children of p are marked for the next run)
27.     mark p for the next run
28.     return null
29. else
30.     return the first item of the heap
```

Figure 5: Procedures findLeastKey and getLeastChild

is called to adjust the heap. A more complex case arises when (*a*) p is the top of both the *inputPath* and the *outputPath*, (*b*) at least one child of p has been output, and (*c*) two or more children of p are read consecutively into p's heap (which has already been constructed using buildHeap) before any child is output. In this case, buildHeap would have to be called again. To avoid this we force a child of p to be output before the second consecutive child of p is inserted as the first element of p's heap array. This case is not shown in insertNode, however, to keep the presentation simple. Procedure getLeastChild returns the first item of the heap, thus the node with the least key value. The only exception is the case when all children of p have been marked for the next run; this marks p for the next run as well.

Returning to findLeastKey, we first dispose of nodes of the *outputPath* that have been fully output (lines 1-5). A node has been fully output if it has been fully read and its heap array is empty. Next, while the top of the *outputPath* stack has all its children marked for the next run, we ascend the tree by popping the stack to find the greatest ancestor that does not have all its children marked for the next run (lines 7-10). If no such node has been found after the root of the tree has been popped, the current run is closed and a new run has to be created (lines 11-15). The root of the tree is output to the new run and pushed onto the *outputPath* stack. At this point, we have reached a node some children of which are eligible to be written to the current run. However, if the subtree rooted at this node has not been fully read yet we need to descend into the tree to find such a complete subtree (lines 16-21). While descending, nodes are visited for the first time since the creation of the current run; therefore, we call buildHeap for each and push onto the *outputPath* stack their child with the least key value (without calling getLeastChild– it will always be the first element of the heap array). As we traverse a path of the tree we output visited nodes (line 19). The procedure returns the node closest to the root after the iteration (line 22).

The fully read node returned by findLeastKey is passed to outputSubtree (shown in Figure 4). The absolute key for the root of the subtree is smaller than all the subtree's nodes (as it is their prefix) so it precedes them in the output run. After the root of the subtree is output, outputSubtree sorts the root's children (all of which are present – line 28). It then recurses until the whole subtree has been sorted and written to the current run (lines 29-30). At the same time, the memory occupied by the subtree is freed (line 31).

Example 3.2: For the dataset of Figure 2 we show a part of the sorting phase in Figure 6, assuming that seven nodes fit in main memory. In Figure 6(a), node 3 has just been read and *inputPath*.top() points to node 8, the parent of the last read node. Since memory is full, the tree is traversed from the root towards the leaves, at each step heapifying and following the pointer to the child with the least key value. In Figure 6(b), the state of the system when the heap for node 8 has been constructed is shown. Stack *outputPath* points to that node, as it is the parent of the node to be output. In Figure 6(c), that node has been output and at the next step (Figure 6(d)), node 1 has been read and placed in the first place of its parent's heap. Node 8 has then been fully read and *inputPath* is popped. A node needs to be output again and heapify is called for the heap of node 8. Since 1 is less than 3, the key of the last written node, node 1 is placed at the end (shown in Figure 6(e)). Node 9 is then output and the next node is read from the input. Note that at the next output step, the subtree rooted at node 12 will be output as a whole, while node 1 will be written to the next run. □

Node serialization in runs. Keys are written to disk runs in a compressed form, as all absolute keys in the tree share common prefixes. Had absolute keys been written uncompressed, the run files would be polluted with redundant information. This would not only be wasteful in terms of secondary storage, but also would heavily increase the I/O cost of writing each run to disk and subsequently reading it back in during merging. We use a typical tree compression scheme. Each time a node is output to the current run, its type, local key value and payload are serialized to the disk, preceded by the following special characters:

- A "|" if the node is a sibling of the last written one.
- A "/" if the node is a child of the last written one.
- A "^" for each level in the tree that the node is higher than the last written one.

Merging. During the merging phase, the sorted runs will be merged to produce the final output. One memory page is used as the input buffer for each input run and one page as the output buffer of the resulting merged run. As with external mergesort for flat data, it is possible that the available physical memory does dot suffice for all sorted runs to be merged in one merging phase. In this case more than one merging levels are required [13]. Merging at each level is identical from an algorithmic perspective, thus we only describe the algorithm for a single merging phase.

The merging algorithm is shown in Figure 7. Nodes are read from the sorted runs into memory. When a node needs to be output a hierarchical priority queue, similar to the one used during the sorting phase, is used to locate the node with the least absolute key. This node is output and the next one is read from the run to which the output node belonged (lines 6-8). Identifying the node to output (line 5) is performed by findLeastKey, but with some modifications with respect to the sorting phase. The main difference is that the first element of a SortNode's heap array is always greater than the one previously written from that heap to the current run. This means that if a node n is visited by findLeastKey, then the subtree rooted at n will be written entirely to the output run before findLeastKey leaves the node. Thus, findLeastKey only pops a

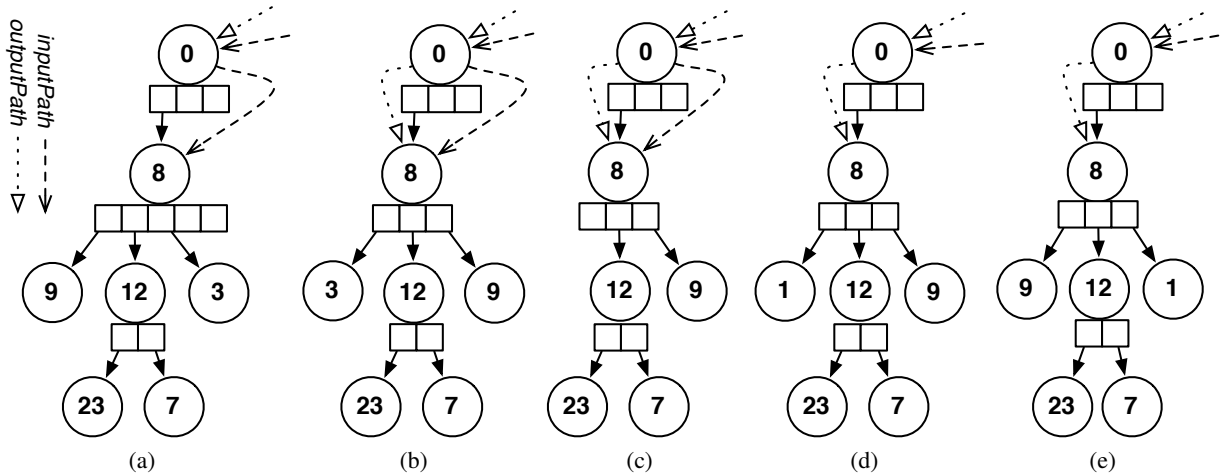


Figure 6: An example of the sorting phase for the creation of the initial runs

```
Algorithm 2: HERMES - Merging Phase (Run [] runs)
1.  for each run file r in runs
2.      readNextLeaf (r)
3.  end for each
4.  do
5.      SortNode min = findLeastKey ()
6.      let r' = run from which min was read
7.      Write min to the output run
8.      readNextLeaf (r')
9.  while (not all runs have been exhausted)
10. return

Procedure readNextLeaf (Run r)
11. while (a leaf node has not been reached)
12.     Read the next node n from r
13.     sn = lookUp (r.path.top(), n)
14.     if ( sn == null)
15.         Create a SortNode sn for n
16.         insertNode (r.path.top(), sn)
17.     end if
18.     inputPath.push(sn)
19. end while
20. return sn
```

Figure 7: The merging phase of HERMES

node from *outputPath* (line 8 in Figure 5) when the subtree rooted at this node has been written completely. Note that there is no way one can tell if the subtree rooted at a node has been entirely read or not without reading the next symbol of all runs that contain children of that node. Therefore, findLeastKey only returns leaf nodes during the merging phase, and outputs internal nodes as it descends to find the leaf nodes. The node returned by findLeastKey is written to the output run and the next node from the run from which the output node came is read. The process ends when all runs have been exhausted.

Note that each time we read from a run, we read as many nodes as required to reach a leaf, accomplished through procedure readNextLeaf of Figure 7. Always reaching a leaf ensures that when an internal node has been read, at least one of its children will have been read as well. For each input run we maintain a stack, termed *path*, that holds the ancestors of the last read node from that run. Initially, *path* contains the SortNode for the root of the tree (which is the first node written to all runs given the run serialization protocol). Note that an internal node may appear in more than one run (*i.e.*, its descendants may appear in multiple runs). When an internal node is read, we check if it is already present in the heap of its parent's SortNode (line 16). If the node is already there, no new SortNode is created; otherwise, a SortNode is created and inserted into its parent's heap (lines 14-17).

4. ALGORITHM ANALYSIS

In this section we study the theoretical properties of HERMES with respect to the size of the initial sorted runs and the I/O cost of the algorithm. We show that our algorithm maintains the most important advantages of external mergesort with replacement selection for the creation of initial runs. At the same time it does not repeat redundant information both in main memory and on disk, and does no redundant comparisons between nodes. We also present how the core algorithm takes advantages of the hierarchical structure to boost its performance.

4.1 Run Size

For flat data, the average size of a run produced by replacement selection is twice the size of the memory used [14]. Hereafter, we refer to the "size" or "length" of a run not in terms of bytes, but in terms of the number of nodes that it contains. The same applies to the size of main memory. For standard external mergesort each run is expected to contain twice as many records as can fit in a full main memory priority heap. After the priority heap becomes full, its size remains constant: before a new record is inserted into the heap, one is first output to the current run (and the heap shrinks when the input has been exhausted). We now prove that the initial runs created by HERMES have the same property:

Theorem 4.1: *The average size of a run is twice the size of available main memory for sorting.* □

PROOF. Consider the priority heap h_n of a node n in main memory, which holds pointers to the children of n. The size of h_n, which we denote as $|h_n|$, grows as children of n are read from the input. When the first child of n is to be output to the current run, h_n stops growing and from that point on its size remains constant. This is because from that point on, as explained in Section 3.2, before a new node is to be inserted into h_n, one node from h_n is output to the current run. When all children of n have been read, h_n begins to shrink. In other words, HERMES outputs the children of n in the same order that standard replacement selection would output them if hey were records of a flat file *and* in the same number of runs. Consequently, for each node n the expected number of n's children written to the current run is twice the size of h_n, *i.e.*, twice the size of h_n when the first child of n is output.

Let m be the maximum number of nodes that fit into memory and consider the point in time t_0 at which a new run is created. At that point in time the memory is full, *i.e.*, a node needs to be written to the run (the first node of the run). At t_0 the size of the heap h_n of a node n ($1 < n < m$) is $|h_n|_0$. All $|h_n|_0$ children of n will be written to the new run, since each one of them is neither marked for the next

run at t_0, nor will ever be marked for the next run (as all of them are present when output to the run starts). When the first child of node n is to be written to that run at a later time t_i ($t_i > t_0$), the size of h_n will be either equal to $|h_n|_0$ (if no child of n was read in the interval $[t_0, t_i]$, *i.e.*, the node had been fully read at t_0) or greater than $|h_n|_0$ if more children of n were read during that interval. In both cases $|h_n|_i \geq |h_n|_0$ holds. However, the expected number of children of n that will be written to that run is $2|h_n|_i$, *i.e.*, the total number of nodes written to that run is:

$$\sum_{n=1}^{n=m} 2|h_n|_i = 2\sum_{n=1}^{n=m} |h_n|_i \geq 2\sum_{n=1}^{n=m} |h_n|_0$$

At time t_0 the memory is full and holds m nodes. Of these nodes, all but the root of the tree are pointed to by some heap h_ℓ, that is $\sum_{n=1}^{n=m} |h_n|_0 = m-1 \approxeq m$ for large values of m. Hence,

$$\sum_{n=1}^{n=m} 2|h_n|_i \geq 2m$$

holds. □

From the description of the algorithm it follows that each node appears only once in main memory, not only during the creation of the initial runs, but also during the merging phase. Also, nodes appear only once in the sorted runs; only the key value of some internal nodes may be written to more than one runs, at most once in each run, and only when it is necessary for the reconstruction of the subtree that the run represents. More importantly, the algorithm only compares local key values of sibling nodes. Never are absolute keys used to compare two nodes that belong to different subtrees.

4.2 I/O Behavior

Regarding the I/O cost of HERMES, suppose that available memory is M memory pages. During the merging phase, $M-1$ memory pages are used for reading the input and 1 page is used to write to the output run. If the total size of the input tree is T memory pages then each initial run will have an average size of $2(M-1)$. Also, since $M-1$ buffers can be used for merging, there are going to be $\lceil \log_{M-1} \lceil \frac{T}{2(M-1)} \rceil \rceil$ levels of merging. Adding the first pass over the input to create the initial runs, we have that there will be $1 + \lceil \log_{M-1} \lceil \frac{T}{2(M-1)} \rceil \rceil$ passes over the input. In each of these passes, the whole tree is read and written to disk. This makes a total I/O cost of $2T \cdot \left(1 + \lceil \log_{M-1} \lceil \frac{T}{2(M-1)} \rceil \rceil\right)$.

As pointed out in [13], one of the main concerns with replacement selection is how one can handle the payloads of the nodes that reside in main memory at any given time, *i.e.*, the payloads of the nodes whose keys are in the heap at that time. If these nodes are kept in the original buffer pages there is a great waste of space: only half of the nodes of any given page are expected to be in the priority heap of their parent node at any given point in time. This would mean that half of the available memory is not effectively used for sorting keys. Therefore, the benefits from replacement selection are cancelled (and quicksort could be used instead, probably yielding better results). As also pointed out in [13], the solution to this problem is to copy the payloads of those nodes to a temporary space in memory until they are written to the run, so that no space is wasted. Assuming that nodes of the same type have similar size, this can be a viable solution. However, large variations in the size of the nodes of the tree require complex and potentially overhead-inducing in-memory management primitives.

Double buffering certain pages during the merging process is also a technique that can improve the performance of our algorithm. For instance, using more than one memory pages for the output run at each merge level can eliminate the need for the CPU to wait for a write I/O call to complete after the output buffer is flushed (as is the case if a single output buffer is used). Regarding the input buffers, the situation is somewhat different. Reserving two memory pages (or more) per input run would reduce the number of runs we can merge by half. What we can do is reserve a number of k memory pages in order to prefetch the next page from the k input buffers that contain one of the k smallest maximum keys among all buffers (since we then know that the next page to be read will be the next from one of those k runs).

4.3 Improvements

We now present how the hierarchical structure has been further exploited to improve the core algorithm.

Processing entire subtrees. A useful optimization arises when all descendants of a node n have been read into main memory and the first needs to be output. In that case, the whole subtree rooted at n is output, with the children of nodes of that subtree being sorted (see Section 3.2). As n will be the first node of the subtree to be written, one can place a mark on the run file indicating that the whole subtree follows, *i.e.*, all descendants of n are written to that same run, following n. That way, when n is read during the merging phase, we know that it is followed by the entire subtree rooted at n. Therefore, only n needs to be brought to memory and be placed in the heap of its parent. The rest of the subtree needs not be constructed in main memory. When n is to be output, the subtree of n is copied from the input run to the output run without any in-memory processing, as it is already sorted. Furthermore, if this subtree spans many pages, these pages can all be prefetched.

Properties of runs. We now turn to the properties of initial runs.

Lemma 4.1: *A group of nodes that co-exist in memory at some point in time will be written either to the same run, or to two consecutive ones.* □

PROOF. During the creation of the initial runs, a node that has been read from the input will be written either (*a*) to the current run, if it has a greater key than all its siblings that had been written to the current run when the node was read, or (*b*) to the next run otherwise. Thus, if two nodes n_1 and n_2 co-exist in memory at some point in time, they will eventually be written either to the same run or to consecutive ones. That is, if r_i is the current run, n_1 and n_2 will either both be included in r_i, or both in r_{i+1}, or one of them in r_i and the other in r_{i+1}. Following an inductive argument, it is easy to see that the same applies for any number of nodes that at some point in time co-exist in memory. □

We shall now show that this is the case for all groups of sibling nodes. Recall that the input tree is read depth-first. For this reason, all nodes with the same parent will be read as a *batch*. Let n_f be the first node of the batch written to the output and n_l the last one.

Lemma 4.2: *Every node of the batch other than n_f and n_l will at some point in time co-exist in memory with some other node of the batch, i.e., with one of its siblings.* □

PROOF. The statement holds due to depth-first traversal. Assuming it does not hold, there must be at least one run of length 1. Any other case implies that at least one node that does not belong to the batch has been read between two nodes of the batch. The latter is eliminated by the depth-first traversal of the tree. The former is negligible since it implies that the memory allocated for the heap can only hold one node (*i.e.*, it has a size of 1); fortunately, we have been able to use larger memory sizes for building heaps! □

Thus, all co-existing nodes will eventually be written to consecutive runs, say, $r_w \ldots r_z$. The only nodes of the group not accounted

for so far are n_f and n_l. To show that *all* groups of sibling nodes will be placed in consecutive runs, it suffices to prove the following.

Lemma 4.3: *Node n_f will be written either to r_{w-1} or to r_w and n_l will be written either to r_z or to r_{z+1}.* □

PROOF. The trivial cases are (*a*) when n_f and n_l co-exist with some other node of the batch in their parent's heap at some point in time, or (*b*) they are written to r_w and r_z respectively; then the statement holds. If neither case holds, we need to show that n_f will be written to r_{w-1} and n_l will be written to r_{z+1}. Therefore, n_f is the only node of the batch written to $r_{w'}$ ($w' < w$) and n_l is the only node of the batch written to $r_{z'}$ ($z' > z$). Then, for n_f we have that when it was to be written to a run, it was the only entry in its parent's heap (otherwise we fall into the first trivial case mentioned above). This implies that memory became full just when n_f was read. After it was output to the run, the next node read, say, n_g, was a sibling of n_f, as input is processed in a depth-first manner (if n_f has no siblings the lemma holds). If the key of n_g is greater than the key of n_f, then n_g is written to that same run, *i.e.*, $w \equiv w'$. Otherwise, it is marked for the next run and outputting continues from a sibling of n_g's parent. The parent of n_g is visited again by the algorithm when the next run is being produced and n_g is output to that run. In that case, $w \equiv w-1$. Similar arguments show that n_l will either be written to r_z or r_{z+1}. □

A different merging scheme. The previous lemmata prove that all children of a node are written in consecutive runs. During the merging phase, one can utilize this fact when not all siblings of a node have been read into its parent's heap and the first sibling is written to a run. A bit of extra book-keeping is needed to identify the first and last of these consecutive runs (*i.e.*, runs pertaining to the set of siblings). Assuming these runs are no more than k, we can prefetch at least one page from each run and in this way avoid having the CPU wait for I/O to complete while merging the siblings.

We shall now present a different merging scheme that takes advantage of all nodes in a batch being written to consecutive runs. In the following we assume that all children of a node n have been written to k consecutive runs $r_w, \ldots, r_{w+k}$. Such information can be recorded for n during the replacement selection phase and stored in the SortNode for n, which remains in memory until the last child of n is flushed to a run. During the merging phase, runs $r_w, \ldots, r_{w+k}$ can be merged into a single run $r_{w,w+k}$. Before we start merging these runs, we can be certain that $r_{w,w+k}$ contains all children of n in the proper order. We therefore record this when the first node of the batch is written during merging. During the next merging level, when n is read, we can identify that this run contains the whole subtree rooted at n, so that we can output all nodes of the batch consecutively, similarly to what we mentioned for the case of merging initial runs. Again, in this way we avoid the cost of re-constructing the subtree in memory and inserting its nodes into the priority heaps of their parents. Considering the prefetching techniques mentioned earlier this can turn into a significant advantage. The only caveat is that this technique is applicable only if multiple merging levels are needed. In case of a single merging level, this variation will probably have worse performance since it will introduce another pass over the records of $r_{w,w+k}$, if these runs $r_w \ldots r_{w+k}$ are merged individually. If multiple merging levels would be used, however, we can use this technique for disjoint groups of k runs. The larger the fan-out of the tree, the greater the length of a batch will be and, thus, the more efficiently this technique is expected to perform.

5. ARCHIVING REVISED

The main motivation for HERMES was archiving of scientific datasets. We now focus on merging a new *sorted* version of a hierarchically represented database into an existing *sorted* archive. Having the incoming version and the archive sorted allows us to overcome restrictions on archive sizes due to main memory merging. The process is outlined in Figure 8. The three main steps are: (*a*) assigning key values to nodes in the input version given a key specification, (*b*) sorting nodes in the new version according to their key values, and (*c*) merging the sorted archive and the incoming version into a new version of the archive, which is both annotated and sorted. During merging, corresponding nodes are identified based on their key values. The archiver stores each node only once in the resulting archive and annotates it with a timestamp representing the sequence of version numbers the node appears in.

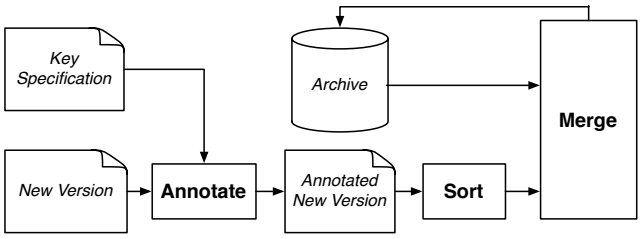


Figure 8: The archiving process

Keys and Annotation. The local key of an element is used to distinguish it from its siblings, as described in Section 3.1. We use hierarchical key specifications as in [4, 5], where all keys of a node are defined relative to its parent. For archiving we make some extra assumptions on the key structure of Section 3.1: key values are unique among siblings, *i.e.*, the document adheres to the key specification. Also, a node below an unkeyed node in the tree cannot be keyed itself. Thus, each key specification defines the maximum depth at which a keyed node can be found. We say that a node is a *frontier* one, if it is the deepest possible keyed node on a path from the root; no unkeyed nodes exist above frontier nodes. We assume that nodes that exist beneath some key path cannot be keyed themselves. This ensures that key values do not change as a result of reordering keyed nodes when sorting or merging them (see [4, 5] for a more comprehensive discussion of these assumptions). One pass over the new version of the database is sufficient to annotate the input with key values [5]. The archive does not need to be annotated as its elements are already annotated with their keys.

Merging sorted documents. Nested merge (*a*) identifies corresponding nodes in the archive and the incoming document and (*b*) merges them into single node in the new archive by recursively merging their corresponding children. The efficiency of identifying corresponding children is greatly improved when the document and the archive are sorted. We then perform the merging in a single depth-first scan of the document and the archive.

The algorithm is shown in Figure 9. It accepts as input: (*a*) a node from the archive (*archNode*), (*b*) a node from the new version (*docNode*), (*c*) a set T of timestamps at which *archNode* existed, and (*d*) the timestamp of the new version t. Nodes *archNode* and *docNode* have been identified as the same, *i.e.*, they have the same key values. For each archive node we can call its method *hasTimestamp*() that returns *true* if the node has a non-inherited timestamp. In the beginning, if *archNode* has such a timestamp, we adjust it to include the timestamp of the new version and write the node and its timestamp to the new archive (lines 1-3). We traverse depth-first the subtrees rooted at *archNode* and *docNode* to merge them. In each step, *childA* and *childD* are the current children of *archNode* and *docNode* being examined, respectively. If the key value of *childA* is less than that of *childD* (line 7), then *childA* is output and the next child of *archNode* is read (line 13). When outputting a node we write the node with its time annotation (if any) and the subtree rooted at the node to the new archive. If *childA* has an in-

herited timestamp, it is output with $T-t$ as timestamp, as it is not present in the new version. Otherwise, it is output with its own timestamp (lines 8-12). If the key of *childD* is less than the key of *childA*, *childD* is output, annotated with t (the timestamp of the new version) and the next child of *docNode* is read (lines 14-16). Otherwise, *childA* and *childD* have the same key value, *i.e.*, they are the same node. They are then merged recursively (line 19) unless *childA* is a frontier node. If so, no merging is required and the whole subtree rooted at *childA* is output, with the timestamp of the new version added to the timestamp of *childA* if the latter is different than the timestamp of *archNode* (lines 21-25). The next children of both *archNode* and *docNode* are then read (lines 26-27), with `null` values indicating that all children have been read.

Recall from Section 3.1 that when comparing a keyed node with an unkeyed one, the keyed node is always found to be less and therefore all keyed nodes will be processed before their unkeyed siblings (if any). This means that if *childA* is unkeyed at some time, all remaining keyed children of *docNode* (*i.e.*, *childD*'s) will be processed in lines 14-17. Respectively, if *childD* is unkeyed, all remaining keyed children of *archNode* will be processed in lines 7-13. Also, an unkeyed node from the archive is *always* considered less than an unkeyed node from the new version. Hence, if both *childA* and *childD* are unkeyed at some point in time, *childA* will be output in lines 7-13. Alternatively, one could compare unkeyed nodes on their value to avoid repeating unkeyed nodes with the same value in the output. In line 30, all children of *archNode* or *docNode* will have been output. Any remaining children below the other node are output with their respective timestamps adjusted (lines 31-42). Procedure nestedMerge is called on the roots of the archive and the new version; on exit the new archive is sorted.

Archiving-aware sorting. Using HERMES as the sorting algorithm, we further improve the efficiency of merging. First, we combine the merging of sorted runs with the nested merging of the archive. This saves us one full scan of the new version. Furthermore, the subtrees under key path values and those under nodes that are keyed by subtree expressions are stored redundantly as key values and as normal nodes. We store these nodes only once in the corresponding key values and extract the respective subtrees when writing the document. This approach reduces I/O cost significantly, as we remove *ca.* 50% of the nodes that need to be kept in main memory and read/written when sorting.

Algorithm 3: nestedMerge (*archNode*, *docNode*, T, t)

```
1.  if (archNode.hasTimestamp())
2.      T = T + t
3.  Write archNode with T as timestamp
4.  childA = archNode.nextChild()
5.  childD = docNode.nextChild()
6.  while (childA ≠ null or childD ≠ null)
7.      if (childA < childD)
8.          if (childA.hasTimestamp())
9.              output(childA, childA.timestamp())
10.         else
11.             output(childA, T − t))
12.         end if
13.         childA = archNode.nextChild()
14.     else if (childD < childA)
15.         output(childD, t)
16.         childD = docNode.nextChild()
17.     else
18.         if (childA is not a frontier node)
19.             nestedMerge (childA, childD, T, t)
20.         else
21.             if (childA.hasTimestamp())
22.                 output(childA, childA.timestamp()+t)
23.             else
24.                 output(childA)
25.             end if
26.             childA = archNode.nextChild()
27.             childD = docNode.nextChild()
28.         end if
29.     end if
30. end while
31. while (childA ≠ null)
32.     if (childA.hasTimestamp())
33.         output(childA, childA.timestamp())
34.     else
35.         output(childA, T − t)
36.     end if
37.     childA = archNode.nextChild()
38. end while
39. while (childD ≠ null)
40.     output(childD, t)
41.     childD = docNode.nextChild()
42. end while
```

Figure 9: Nested Merge of sorted trees

6. EXPERIMENTAL RESULTS

HERMES has already been deployed as part of the archive management system XARCH [15]. XARCH is a stand-alone Java application that allows one to maintain, populate, and query archives of hierarchical data with a key specification. XARCH uses HERMES for sorting incoming versions before applying nested merge (as in Section 5). We also wanted to evaluate the performance of HERMES under various workloads as a stand-alone hierarchical data sorting solution. To that end we re-implemented it in C++. HERMES was compiled using the GNU C++ compiler, version 4.1.2. All our experiments were run on an Intel Core 2 Duo processor clocking at 2.33GHz with 2GB of physical memory. The box was running Ubuntu Linux 7.10 with the 2.6.22 kernel. We report performance for both Java and C++ implementations. The first presents the archiving performance of HERMES in the context of a prototype implementation; the second presents the "raw" sorting performance for arbitrary hierarchical data. For each experiment we report the average wall clock time of five runs over cold data.

Archival testing. We used XARCH to archive ten major releases of the *SwissProt* database [12], *i.e.*, releases 40 (October, 2001) to 49 (February, 2006). The first release has 17.1 million XML elements and a size of 403MB while the last has 51.6 million elements and a size of 1.2GB. We used 300MB of main memory for sorting. In the first archiving step, version 40 is added to the empty archive; in each following step a new version is added to the archive. For each archiving step me measured the time for annotating and sorting the new version, and the time for merging it with the existing archive. We report these times in Figure 10, showing the size of each version next to the version number. The sorting time increases linearly with the size of each version. The first archiving operation is the shortest, as there is no existing archive. For the rest of the operations, merging needs one pass over the existing archive and one pass over the new version. As the size of both the archive and the new version grows, merging takes slightly longer. The size of the archive after all ten operations was 2.28GB. Overall, merging time is comparable to sorting time for all operations after the first one.

Sorting testing. We implemented the algorithm of Section 3 and the improvements of Section 4. To create the input data we used a custom data generator. Each input file was an XML document, each node of which had a randomly generated character string as its label. We used the label of a node as its key. Node label lengths, and thus key value lengths, were variable. The generator allowed us to specify the maximum depth of the tree and a maximum fan-out for all nodes. The fan-out of each node was uniformly distributed between 0 and the specified maximum. As a result, the average fan-out was half the maximum. We compare the performance of HERMES to that of NEXSORT using the original implementation of

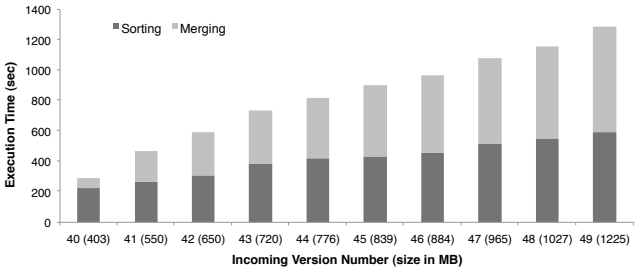


Figure 10: Archiving operations time

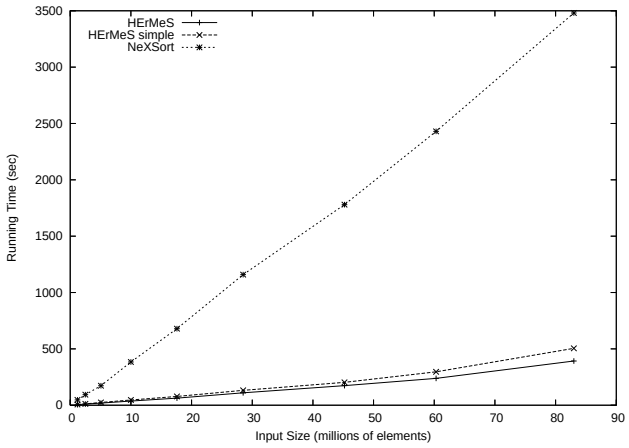


Figure 11: Impact of input size

NEXSORT in combination with the *Transparent Parallel I/O Environment* [10], as in [16]. As suggested in [16], we set the sorting threshold for NEXSORT to be roughly twice the block size, which, for our system was 64KB, making the threshold equal to 128KB.

6.1 Impact of Input Size

We first measured the impact of input size on the running times of HERMES and NEXSORT. We used input trees of a depth of six, which is a typical depth for most real-world datasets. For each depth, we generated trees of different sizes by varying the average node fan-out. For both algorithms, the size of available main memory for sorting was set to 10MB. The sizes of the input trees varied between 30MB (or, 1.2 million nodes) and 2.2GB (or, 83 million nodes). The average length of a node key was set to ten bytes. The results are presented in Figure 11. We also report under the "HERMES simple" plot the response time for our algorithm if the improvements of Section 4 are not used.

In all cases, HERMES performs 8.5 to 10.8 times faster than NEXSORT. This is due to the way NEXSORT writes sorted runs. NEXSORT employs a stack over secondary storage to store the next subtree to be sorted. When a subtree residing in the stack has been sorted, the sorted subtree is written to disk and a pointer is written back to the stack. At any point in time there are multiple such secondary storage stacks used for book-keeping purposes. Pushing nodes onto these stacks and popping them involves disk accesses. Therefore, during its sorting phase, NEXSORT reads and writes both to the on-disk stack pages and to the current sorted run at the same time. This introduces a severe performance penalty. In addition, NEXSORT accesses the disk in a random pattern when reconstructing the output tree: it follows pointers to sorted runs that have not been sequentially written to disk (*i.e.*, one run after an other). Hence, a lot of time is spent with the CPU stalling for I/O.

On the other hand, during replacement selection for the creation of initial runs, HERMES accesses secondary storage only for the purpose of flushing data to the current run. As additional evidence,

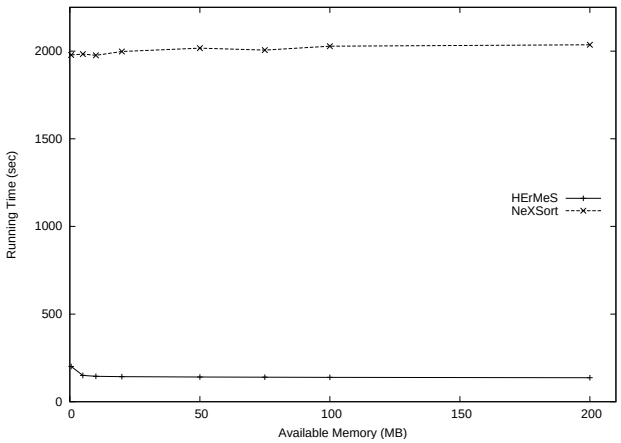


Figure 12: Impact of available main memory

during our experiments we observed that NEXSORT's reconstruction time amounted to almost 40% of its total running time. On the contrary, the merging phase for HERMES took no more than 20% of the total running time (when a single merging level is required). As one can observe from the results, the improvements discussed in Section 4, give another 15% to 25% performance boost to HERMES. Thus, we believe such improvements are a worthwhile addition to the main algorithm and have used them in all experiments.

6.2 Impact of Available Main Memory

In our next experiment, we examined the performance of both algorithms for different sizes of main memory available for sorting. We generated an 1GB input tree with 41 million nodes. This tree was six levels deep and the average node fan-out was set to 35. We varied the available memory size to seven different values ranging between 0.5MB to 200MB and measured the running time for both HERMES and NEXSORT. The results are presented in Figure 12.

When only 0.5MB of physical memory are used, HERMES requires two merging levels. We observed that processing each merging level takes about 45 seconds, or, about 25% of the total running time. For larger sizes of available main memory only one merging level is required and thus the total running time is almost constant. When the amount of available memory grows much larger than the amount needed to achieve a single merge level, running time drops slightly as available memory increases. This is because the average size of a sorted run grows much bigger and the number of sorted runs drops, *i.e.*, merging has a smaller fan-in. When only one merging level is needed HERMES performs almost ten times faster than NEXSORT, irrespective of the amount of available memory. When two merging levels are required, HERMES performs about eight times faster than NEXSORT.

6.3 Impact of Tree Depth

We next examined how the depth of the input tree affects the performance of our algorithm. We experimented with trees three, five, seven, and nine levels deep. For each of these depths we generated trees of different sizes (by varying the average fan-out). The results are presented in Figure 13. As shown, the running time of our algorithm is not heavily affected by the depth of the tree, especially for trees deeper than five levels. One can observe that the deeper a tree is, the more efficiently it is sorted by HERMES.

To understand why this is the case, consider trees of the same size (*i.e.*, equal numbers of nodes) but of different depths. To keep the number of nodes fixed, shallow trees will have a much greater fan-out than deep trees. For instance, the largest tree we generated for each depth had a size of about 1.6GB. However, the average

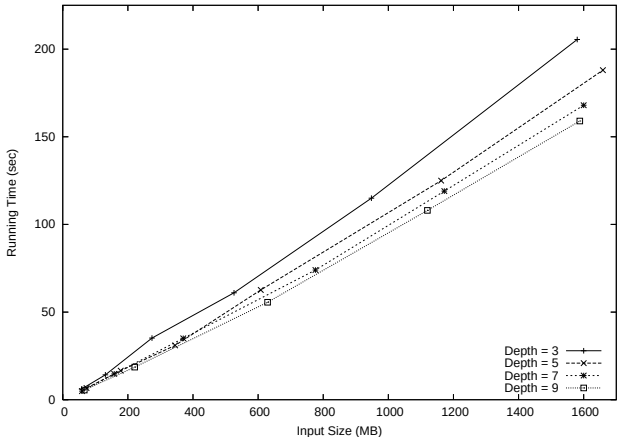


Figure 13: Impact of tree depth

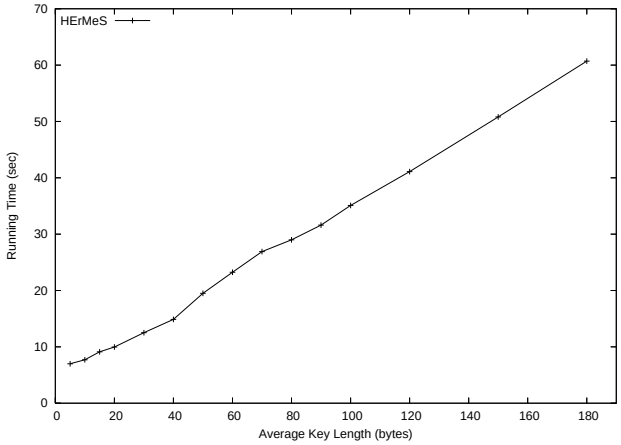


Figure 14: Impact of key length

fan-out for a tree of depth three is 7,500 nodes, while the average fan-out for trees of depth five, seven, and nine are 85, 21 and 9 respectively. When sorting using HERMES, the larger the fan-out of a tree is, the greater the average size of a heap is in main memory. As a result, each heapify operation takes longer. Moreover, for a fixed memory size, the probability of a subtree having been fully read during the sorting phase grows are the fan-out shrinks. Thus, the algorithm processes entire subtrees more frequently, albeit for smaller subtrees. Thus, the improvements described in Section 4.3 are more frequently applicable. This behavior verifies our claim that HERMES takes advantage of the hierarchical structure of a dataset: it performs better for deeper hierarchies. Also, these results verify the theoretical expectations of [16]: the possible sorting outcomes for a tree with a fixed number of nodes increases as the maximum fan-out of the tree grows.

6.4 Impact of Key Length

We then moved on to evaluate the performance of HERMES with respect to different key lengths. We used our generator to create regular trees with the same number of nodes. All trees had a depth of five and a constant fan-out of eighty. They only differed in the average length of node keys. We created trees with the average key length varying from 5 bytes to 180 bytes. For each such tree, we ran HERMES having set the available main memory to 10MB. The resulting running times are shown in Figure 14.

As expected, the running time increases linearly with the length of the key. When longer keys are used all in-memory operations on keys, such as in-memory copying and comparisons, take longer to execute. The same applies to secondary storage operations, since the amount of data to be read/written for each key increases. The increase in execution time is linear to the length of the keys, which is also expected since all the aforementioned operations take linear time with respect to key length.

7. CONCLUSIONS AND FUTURE WORK

We have presented HERMES, an algorithm for efficiently sorting hierarchical data in external memory. HERMES generalizes widely adopted sorting techniques, such as replacement selection and external mergesort. We have studied the performance of our algorithm theoretically and proposed improvements. Experimental results show that HERMES clearly outperforms competition by a significant factor and can scale up well to meet real-world needs.

As mentioned, a version of HERMES has already been deployed as part of the XARCH database archiver [15]. In the near future we plan to deploy the stand-alone version of HERMES for sorting arbitrary XML data. We also aim to explore further uses of the algorithm in different application domains. We believe that HERMES is an optimal solution to the problem of sorting hierarchical data.

8. REFERENCES

[1] I. Avila-Campillo *et al.* XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *PLANX 2002*, 2002.

[2] A. Berglund *et al.* XML Path Language (XPath) 2.0, January 2007.

[3] S. Boag *et al.* XQuery 1.0: An XML Query Language, January 2007.

[4] P. Buneman *et al.* Keys for XML. In *WWW*, 2001.

[5] P. Buneman *et al.* Archiving scientific data. *ACM Trans. Database Syst.*, 29(1), 2004.

[6] S. S. Chawathe *et al.* Change detection in hierarchically structured information. In *SIGMOD*, 1996.

[7] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. *ICDE*, 2002.

[8] Consortium. The Universal Protein Resource (UniProt). *Nucleic Acids Res.*, 35(Database Issue):D193–197, 2007.

[9] T. H. Cormen *et al. Introduction to Algorithms (Second Edition)*. MIT Press, 2001.

[10] D. E. Vengroff. A Transparent Parallel I/O Environment. In *DAGS Symposium on Parallel Computation*, 1994.

[11] European Bioinformatics Institute. EMBL Nucleotide Sequence Database, 2007.

[12] European Bioinformatics Institute. Swiss-Prot Protein Knowledgebase, 2007.

[13] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.

[14] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd edition*. Addison-Wesley, 1998.

[15] H. Müller, P. Buneman, and I. Koltsidas. XArch: Archiving Scientific and Reference Data. In *SIGMOD*, 2008.

[16] A. Silberstein and J. Yang. NEXSORT: Sorting XML in External Memory. In *ICDE*, 2004.

[17] K. Tufte and D. Maier. Aggregation and Accumulation of XML Data. *IEEE Data Eng. Bull.*, 24(2), 2001.

[18] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: an effective change detection algorithm for XML documents. In *ICDE*, 2003.

[19] Wanxia Wei and Mengchi Liu and Shijun Li. Merging of XML Documents. In *ER*, 2004.

[20] L. Zheng and P.-Å. Larson. Speeding up external mergesort. *IEEE Trans. Knowl. Data Eng.*, 8(2), 1996.