SliceSort: Efficient Sorting of Hierarchical Data

Quoc Trung Tran*
UC Santa Cruz
tqtrung@soe.ucsc.edu

Chee-Yong Chan
National University of Singapore
chancy@comp.nus.edu.sg

ABSTRACT

Sorting is a fundamental operation in data processing. While the problem of sorting flat data records has been extensively studied, there is very little work on sorting hierarchical data such as XML documents. Existing hierarchy-aware sorting approaches for hierarchical data are based on creating sorted subtrees as initial sorted runs and merging sorted subtrees to create the sorted output using either explicit pointers or absolute node key comparisons for merging subtrees. In this paper, we propose `SliceSort`, a novel, level-wise sorting technique for hierarchical data that avoids the drawbacks of subtree-based sorting techniques. Our experimental performance evaluation shows that `SliceSort` outperforms the state-of-art approach, `HErMeS`, by up to a factor of 27%.

Categories and Subject Descriptors

H.2.4 [**System**]: Query processing

Keywords

Hierarchical Data, Slicesort, Sorting

1. INTRODUCTION

Sorting is a fundamental operation in data processing and techniques to optimize sorting "flat" data have been extensively studied for both main-memory and external memory contexts [7, 6]. However, there is very little work on sorting hierarchical data such as XML documents [9, 8]. In a fully sorted hierarchical document, the list of child nodes of every non-leaf node is sorted according to some given criteria (e.g., the key of the child node or some function of the contents in the subtree rooted at the child node). As a simple example, Figures 1(b) and (c) show an unsorted and a sorted hierarchical data, respectively, where the nodes are sorted alphabetically by their key values given by the node labels.

*The work was done when the author was at National University of Singapore.

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

Hierarchical sorting has application in the archival of scientific data, which is predominantly stored in hierarchical data formats. To archive a new data version, an efficient approach is to first sort the new data and then merge it with the existing data version [8, 9].

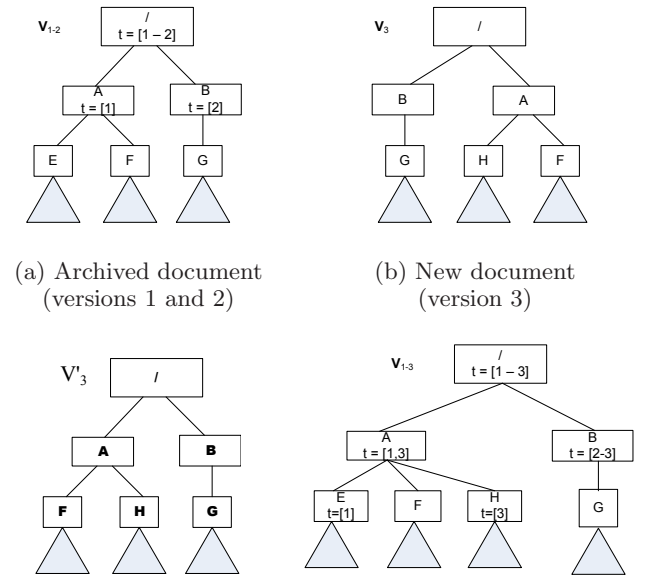
EXAMPLE 1. *Consider the archival approach proposed in [3] to store multiple versions of hierarchical data. Figure 1(a) shows an example archived document, $V_{1\text{-}2}$, that consists of two data versions. Each node in the document has a node label (e.g., /, A, B) and either an explicit version tag (indicated by "t = ...") or an implicit version tag. A node's version tag indicates in which version(s) of the document the node is present; for example, node "A" is present in only version 1, node "B" is present in only version 2, and the root node "/" is present in both versions 1 and 2. A node without an explicit version tag has an implicit tag that is inherited from its parent node. For example, nodes "E" and "F" both inherit the version tag from its parent node "A" and they are all present in only version 1; while node "G" inherits its version tag from node "B" and appears only in version 2. Note that the document $V_{1\text{-}2}$ is hierarchically sorted based on the lexicographical order of its node labels: the child nodes of the root node are sorted with node "A" preceding node "B", and the child nodes of node "A" are sorted with node "E" preceding node "F".*

Consider a new version of the document V_3 (shown in Figure 1(b)) to be merged into the archived document $V_{1\text{-}2}$. An efficient approach to merge the documents is to first sort V_3 into V_3' (shown in Figure 1(c)) and merge them using a synchronized traversal of the pair of sorted documents [3]. The merged archived document is shown in Figure 1(d). □

Another application of hierarchical sorting is in change detection of XML documents, which is useful to control the changes in a warehouse with a large volume of XML documents [8]. Detecting changes in such an environment serves many purposes such as versioning, querying the past, and monitoring the changes [4, 5, 10]. Earlier works on change detection in XML documents (e.g., [4, 5, 10]) operate on unsorted documents that are assumed to be entirely resident in main memory. However, the state-of-the art approaches that can operate on large, disk-based data are based on sorted documents [8].

Hierarchical sorting is also useful for processing batch updates to an existing sorted XML document. The idea is to sort the batch of updates and merge it with the existing document [9, 8].

Hierarchical sorting is also useful in the evaluation of *order by* clause in XPath [1] and XQuery [2] that allows the



(a) Archived document (versions 1 and 2)

(b) New document (version 3)

(c) Sorted new document (version 3)

(d) Updated archived document (versions 1 to 3)

Figure 1: Merging XML documents

results of queries to be output in a specific order. This clause sorts the sequence of result XML fragments, but does not recursively sort the fragments themselves. However, with the help of DTDs, hierarchical sorting can be expressed in XPath and XQuery using explicitly specified nested *order by* clause. The hierarchical sorting can provide an efficient algorithm for processing such queries.

A straightforward method to sort hierarchical XML data is to flatten the XML tree into a file of flat records so that conventional sorting methods can be applied to sort this derived format [9]. In this approach, each XML node is represented by an absolute key, which is formed by the concatenation of the local keys of all nodes along the path from the root node to this node, and the collection of XML nodes are sorted based on their global absolute keys. Although this approach is easy to implement, it has very poor performance as it is sorting the nodes globally using their absolute keys instead of exploiting the hierarchical structure of the input data to sort each collection of sibling nodes locally. To address the limitation of this naive approach, two hierarchy-aware approaches, *NeXSort* [9] and `HErMeS` [8], have recently been proposed. Both of these methods are based on a two-phase approach: the first phase creates sorted runs each of which is a sequence of one or more sorted subtrees, and the second phase merges the sorted runs to produce the final sorted output. A subtree is referred to as sorted iff the list of child nodes of every non-leaf node belonging to this subtree is sorted according to the given criteria.

In *NeXSort*, each sorted run consists of a single sorted subtree, and explicit pointers are used to link the sorted subtrees to maintain the parent-child relationships for parent and child nodes that are stored in different sorted runs. By exploiting these explicit pointers, the final sorted output is produced using a depth-first traversal of the sorted runs by following the appropriate pointers. Thus, the second phase of *NeXSort* essentially merges all its single-subtree sorted runs in one pass by a depth-first traversal of its sorted runs (starting from the sorted run containing the root node) with the help of the pointers to sorted child subtrees. In contrast to *NeXSort*, `HErMeS` is a generalization of the well-known external merge-sort technique. Sorted runs are created using a hierarchy-aware replacement selection algorithm such that each sorted run is a sequence of sorted subtrees, and each data node is associated with an absolute key represented in a compressed form. The sorted runs are iteratively merged in the second phase to form the sorted output.

In summary, the two existing hierarchy-aware approaches are both based on creating sorted subtrees as initial sorted runs and merging sorted subtrees to create the sorted output. The merging of two subtrees requires locating in one subtree the parent node of the root node of the other subtree. Whereas *NeXSort* performs the merging by using explicit pointers, `HErMeS` merges based on comparing absolute node keys.

In terms of performance, `HErMeS` was shown to outperform *NeXSort* [8]. The drawback of *NeXSort* is that as each sorted run consists of only a single sorted subtree, there are many more sorted runs created in the first phase, and the merging of these sorted runs in the second phase using the explicit pointers incurs a lot of random disk I/Os. For `HErMeS`, its drawback is the overhead incurred for manipulating absolute node keys.

In this paper, we propose a novel approach, named `SliceSort`, that does not require using explicit pointers or global absolute keys for sorting hierarchical data. Instead, `SliceSort` sorts in a *top-down, level-by-level* manner by using only local node keys. Given any two nodes at the same level in a hierarchical document, the relative ordering of the two nodes can be determined as follows: if the nodes are sibling nodes (i.e., they have the same parent), then the two nodes are ordered based on their local node keys; otherwise, the nodes are ordered based on the relative ordering of their parent nodes.

`SliceSort` sorts data in three phases. In the first phase, `SliceSort` reorganizes the pre-order-formatted input XML document into "slices", where each slice corresponds to the data at a level in the document. `SliceSort` essentially scans the input document in depth-first order, and transforms and stores the data in a breadth-first ordered format. In the second phase, each slice (starting from the top slice) is sorted using the well-known external merge-sort algorithm; the nodes at each slice are ordered using their local node keys as well as the relative ordering of their parent nodes. In the third phase, `SliceSort` performs a depth-first traversal of the sorted slices to transform them to the sorted, pre-order-formatted document.

Our paper makes the following contributions.

1. We introduce a novel technique, `SliceSort`, for sorting hierarchical data. In contrast to existing hierarchy-aware sorting methods which rely on subtree-based sorted runs, `SliceSort` employs a top-down, level-wise sorting technique to avoid the drawback of subtree-based sorting (Sections 2 &3).

2. We conduct a performance evaluation study to compare `SliceSort` and the state-of-the-art method, `HErMeS`. Our results indicate that `SliceSort` outperforms `HErMeS` by up to a factor of 27% (Section 4).

2. HIERARCHICAL SORTING WITH SLICESORT

The input to the hierarchical sorting problem is a tree-structured XML document T_{in} where the tree nodes are organized in a pre-ordered format. The output is a sorted XML document T_{out} (also in pre-ordered format) that is derived from T_{in} such that for each internal node in T_{out}, all its child nodes are ordered based on some sort key.

In general, the sort key for a node n can be an arbitrary function of the contents of the subtree rooted at n (e.g., size of subtree in terms of number of nodes). For simplicity of presentation, in this paper, we assume that each node n is associated with a "local" key, denoted by $key(n)$, which is a string value that is stored as one of the node's attributes in the input document T_{in}. Thus, given two sibling nodes n and n' in T_{in}, the subtree rooted at n precedes that at n' in T_{out} if and only if $key(n)$ precedes $key(n')$ lexicographically. We will discuss how `SliceSort` can handle more general sort keys that can be derived dynamically at run-time in Section 2.4. We use $payload(n)$ to denote any other attribute values and textual contents associated with node n.

In this paper, we use h to denote the height of T_{in}; thus, T_{in} has $h+1$ levels of nodes, where the root node is at level 0 and the last level of leaf nodes is at level h. We will use the term "tree" and "hierarchical document" interchangeably.

`SliceSort` consists of three phases. In the first phase, `SliceSort` flattens T_{in} into "slices", $L_0, \cdots, L_h$, where each slice L_i corresponds to the nodes at level i in T_{in}. In the second phase, the slices are sorted top-down starting from L_0 to L_h; each slice is sorted using the established external merge-sort algorithm. Finally, in the third phase, `SliceSort` performs a depth-first traversal of the sorted slices to stitch them together to produce T_{out}. In the following, we elaborate on the three phases assuming that all the data and structures can be stored in the main memory. We consider memory management issues in Section 3.

2.1 Phase 1: Flattening Tree into Slices

In the first phase, `SliceSort` transforms the pre-order-formatted T_{in} into a level-wise representation containing $h+1$ slices of data. Each slice, denoted by L_i, is a sequential list of all the nodes at level i in T_{in}. The ordering of the nodes in L_i are consistent with their ordering in T_{in}: a node n precedes another node n' in L_i if and only if n and n' are at level i in T_{in} such that n precedes n' in the pre-order traversal of T_{in}. `SliceSort` creates the $h+1$ slices by performing a single sequential scan of T_{in}: as each node is encountered in the pre-order traversal scan of T_{in}, the node is appended to slice L_i, where i is the level of the node.

To maintain the hierarchical structure of the data, the slices are created such that the parent-child relationships in T_{in} are preserved by recording some additional information in the slices. Specifically, for each slice, each node is assigned a unique integer identifier, denoted by $id(n)$, representing its sequential rank in the slice. That is, if n is the j^{th} node in a slice, then $id(n) = j$. For each node n, we also record the identifier of its parent node, denoted by $parid(n)$[1]. Thus, each node n in a slice is represented by a quadruple $(parid(n), key(n), id(n), payload(n))$; and the information in the slices can be used to reconstruct T_{in}. Note that the $parid(n)$ of a node n is easily derived by simply using a stack to maintain the identifiers of the ancestor path of nodes of the current node being processed in T_{in}.

[1]For the root node n_{root} of T_{in}, $parid(n_{root}) = 0$.

2.2 Phase 2: Sorting Slices

The second phase sorts the slices created by the first phase. Let L'_i denote the sorted slice L_i, $i \in [0, h]$. The idea of `SliceSort` is that once all the slices have been sorted, T_{out} can be generated by stitching the nodes in the sorted slices together based on their parent-child relationships. Therefore, given two nodes n and n' in slice L_i, n precedes n' in T_{out} if and only if n precedes n' in L'_i.

It follows that we can define the hierarchical sorting of T_{in} in terms of the level-wise sorting of the slices as follows. Given two nodes n and n' in slice L_i, n precedes n' in L'_i if one of the following conditions holds:

C1. n and n' are sibling nodes and $key(n)$ precedes $key(n')$; or

C2. n and n' have different parent nodes (given by p and p', respectively) and p precedes p' in L'_{i-1}.

Based on the above recursive sorting definition, `SliceSort` sorts the slices in a top-down manner by sorting L_i before L_{i+1}. For L_0, the sorting is trivial since L_0 consists of only the root node of T_{in}. For L_1, since all the nodes are sibling nodes (their parent is the root node of T_{in}), they are sorted based simply on condition C1. For the general case of L_i, $i > 1$, the sorting of L_i may need to take into account the ordering of the nodes in L'_{i-1} due to condition C2.

To facilitate the checking of the ranks of the nodes in the sorted slices, `SliceSort` creates a mapping table, denoted by MT_i, during the sorting of L_i to L'_i. Given the rank of a node n in L_i, MT_i returns the rank of n in L'_i. Each MT_i is created during the final merging pass to generate L'_i: as each node n is output to L'_i, we set $MT_i[id(n)] = j$, where $id(n)$ and j are the ranks of n in L_i and L'_i, respectively.

2.3 Phase 3: Stitching Sorted Slices

At end of the second phase, all the slices have been sorted. Since the ordering of the nodes in each L'_i is equivalent to the ordering of nodes at level i in T_{out}, all that remains to be done in the third and final phase of `SliceSort` is to stitch together the nodes in the sorted slices based on their parent-child relationships to produce T_{out} organized in pre-order format. This is achieved by essentially performing a depth-first traversal of the sorted slices.

For each sorted slice L'_i, we first initialize a cursor to point to the first node of L'_i. We refer to the node pointed by the cursor in L'_i as the *current node* in L'_i and denote it by n_i. The depth-first traversal of the sorted slices is performed by visiting the current nodes in them in a top-down manner. The traversal first visits the root node n_0 in L'_0 and outputs n_0 to T_{out}. The traversal then recursively visits n_1 and so on. In general, whenever the traversal visits a node n_i, it first checks whether n_i is a child node of n_{i-1} based on the *parid* information. If n_i is a child of n_{i-1}, then the traversal outputs n_i to T_{out} and recursively visits n_{i+1}. Otherwise, if n_i is not a child of n_{i-1}, then it means that the entire subtree rooted at n_{i-1} has been output to T_{out}. In this case, if n_{i-1} is not the last node in L'_{i-1}, then the cursor in L'_{i-1} is updated to point to the next node in L'_{i-1} and the traversal then recurses by visiting the new n_{i-1}; otherwise, if n_{i-1} is the last node in L'_{i-1}, then it means the entire subtree

rooted at n_{i-2} has been output to T_{out} and we recursively check if n_{i-2} is the last node in L'_{i-2} and so on. The depth-first traversal (and hence also SliceSort) terminates once all the nodes in the sorted slices have been visited and output to T_{out}.

2.4 Handling General Sort Keys

Our discussion of SliceSort has so far assumed that the sorting criteria is based on the local keys associated with the nodes in the input tree. It is straightforward to adapt SliceSort to other sorting criteria that is based on the contents of the subtree rooted at each node. Examples include (1) the total size (in bytes) of the subtree, (2) the number of nodes in the subtree, (3) the height of the subtree. To handle such general sorting criteria, SliceSort only needs to modify slightly its first phase. Specifically, for each node n that is encountered in the pre-order traversal, SliceSort maintains n in a data stack and updates its sorting value during the traversal of its subtree. After the entire subtree rooted at n has been visited, SliceSort appends n together with its derived sorting value into the corresponding slice and removes n from the data stack. The maximum number of nodes to be maintained in the data stack is bounded by the height of the input tree.

Note that the two existing hierarchy-aware approaches (*NeXSort* and HErMeS) can also be adapted to handle such general sorting criteria by inserting a node into an initial sorted run only after the subtree rooted at the node has been traversed.

3. MEMORY MANAGEMENT

In our simplified discussion of SliceSort in the previous section, we have assumed that all the data and structures can be stored in main memory. Clearly, this assumption does not hold when the input data is large. In this section, we explain how SliceSort manages memory to sort large data files. Let B denote the total number of memory pages allocated for sorting.

3.1 Phase 1: Creating Slices

In the first phase, memory needs to be allocated among the input data T_{in} and the slices $(L_0, \cdots, L_h)$. SliceSort allocates one memory page for reading T_{in} and the remaining memory pages (denoted by M pages) for storing the slices. Let B_i denote the main memory buffer allocated for slice L_i, $i \in [0, h]$, which has M_i pages; thus $\sum_{i=0}^{h} M_i = M$. As SliceSort scans T_{in}, each data node at level i is appended to B_i. Whenever B_i becomes full, SliceSort will flush B_i to the disk file corresponding to slice L_i.

SliceSort uses a simple heuristic to allocate the memory for the slices such that each M_i is proportional to an estimated size of L_i. Assuming that the height of T_{in} is h, the average size of a node is $navg$, and the average node fan-out of T_{in} is f. The number of nodes in L_i is estimated to be f^i and the size of L_i is estimated to be $f^i \times navg$. The total size of nodes in T_{in} is estimated to be $size = \sum_{i=0}^{h} f^i \times navg$. Therefore, each M_i is $\max\{1, \lfloor \frac{f^i \times navg}{size} \times M \rfloor\}$ pages.

The remaining issue is how are the values of h, $navg$, and f estimated? To estimate h, SliceSort scans the first few pages of T_{in} and estimates h to be the maximum length of the root-to-leaf paths sampled. Similarly, to estimate $navg$, SliceSort scans the first few pages of T_{in} and estimates $navg$ to be the average size of all nodes sampled. Since $size \leq M$, the estimated value of f is derived to be $\lfloor (\frac{B}{navg})^{\frac{1}{h}} \rfloor$. In the event that h has been under-estimated, SliceSort will dynamically reduce the buffer pages for the allocated slices (flushing buffers if necessary) to create new buffers for the additional slices using the same heuristic.

3.2 Phase 2: Sorting Slices

Since SliceSort relies on the well-known external merge-sort algorithm for sorting slices, the memory allocation policy for the sorting is already taken care and optimized. However, there are two additional considerations sorting each L_i:

1. the creation of the mapping table MT_i during the final merging pass (to be used for sorting L_{i+1}), and
2. the use of the mapping table MT_{i-1} to create the initial sorted runs of L'_i.

Since the number of nodes in L_i is known at the end of the first phase, the storage required for MT_i is known before the start of the second phase. In this section, we discuss the construction and usage of MT_i when MT_i cannot fit entirely in main memory.

To construct MT_i during the last merging pass of sorting L_i, we allocate one memory page P to store MT_i and flush P to disk when it becomes full. As each record is output to the final sorted run of L'_i, we append an entry for the record into P. Thus, in contrast to the main-memory resident MT_i discussed in Section 2.2, which is sorted in ascending order of the rank of nodes in L_i, the disk copy of MT_i is ordered in ascending order of the rank of nodes in L'_i.

To create the initial sorted runs of L'_{i+1}, we essentially need to perform a foreign-key join of L_{i+1} and MT_i to map $parid(n)$ of each node n in an initial sorted run to its rank in L'_i. This can be achieved using an appropriate standard join algorithm that is chosen in a cost-based manner. For example, using the nested-loop join method, k memory pages will be allocated for loading L_{i+1} (the "outer relation") and the remaining $(B - k)$ pages will be allocated for loading MT_i (the "inner relation"). Thus, the size of each initial run is at most k pages. To maximize the size of each initial run, k is set to $B - 1$.

3.3 Phase 3: Stitching Sorted Slices

In the third phase, memory needs to be allocated for reading in the sorted slices and writing the sorted output T_{out}. Similar to the allocation principle for the first phase, SliceSort allocates one page for writing T_{out}, and allocates the remaining memory pages among the sorted slices L'_i such that the size of the buffer B_i for each L'_i is proportional to its size (which is known at the end of the second phase).

Thus, for each L'_i, SliceSort sequentially scans L'_i and loads the pages of L'_i into B_i until the buffer B_i is full. Whenever all the nodes in B_i have been output to T_{out}, SliceSort will read into B_i the next sequence of pages from L'_i.

4. EXPERIMENTAL STUDY

This section presents our experimental study to evaluate the efficiency of our proposed SliceSort algorithm. We compare SliceSort against state-of-the-art approach, HErMeS, for sorting of hierarchical data.

Platform. Our experiments were conducted on a dual-core, 2.33GHz PC running Linux 2.6.32-41, 32-bit with 3.75GB

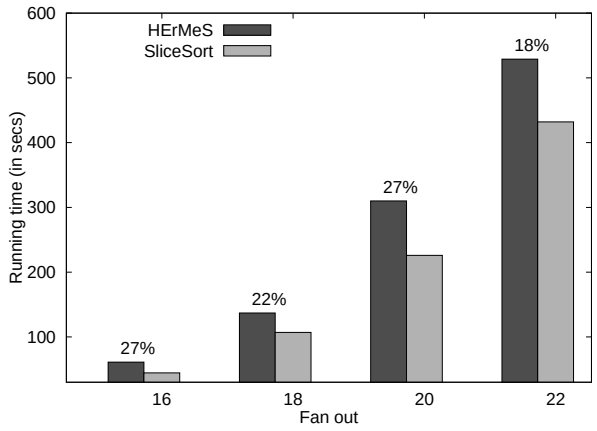


Figure 2: Comparison of running time

of RAM, and a 250GB hard disk. Both `SliceSort` and `HErMeS` were implemented in C++.

Data Sets & Parameters. To create input data sets, we used the same data generator from [8] that generates input XML documents. Each node in the generated tree has a randomly generated character string as its label. Following [8], we used the node label as its key value and set the payload of a node to be empty.

The data generator allows the following three parameters to be varied: the node label (i.e., key) length, the maximum tree height, and node fan-out. The fan-out of each node follows a uniform distribution ranging from 0 to a specified maximum fan-out value.

Due to the space limit, we only present the results on varying fan-out parameter; similar trends are also observed for varying other parameters. We generated 4 data sets, D_1 to D_4, by setting $height = 8$, $keylength = 10$, and varying $fanout$ using the values $16, 18, 20$, and 22. The buffer size used in the experiments is 1000MB.

Metrics. The algorithms are compared in terms of their end-to-end running time which includes the time to read the input tree document, sort, and write the sorted document into an output file. The dominant CPU operations in both algorithms are the number of *key comparisons*, which is counted as follows. For `SliceSort`, the number of key comparisons refers to the number of local node key comparisons. The comparisons of parent node identifiers (during the second phase) are excluded because the identifier comparisons, which are integer value comparison, are much cheaper than the XML node key comparisons, which are string value comparisons. For `HErMeS`, the key comparisons include both the comparisons on local keys of nodes in the tree and absolute keys because `HErMeS` needs to store the absolute keys on sorted runs in some cases. Thus, a key comparison in `HErMeS` is more expensive than that in `SliceSort`.

The dominant I/O operation of `SliceSort` and `HErMeS` is the number of passes to read and write the entire document.

Results. Figure 2 compares the performance of `SliceSort` and `HErMeS` as a function of the fan-out parameter (corresponding to data sets D_1 to D_4). The results show that `SliceSort` outperforms `HErMeS` in the order of 18% to 27%.

The second and third columns in Table 1 show the detailed

	HErMeS	SliceSort
CPU time (secs)	495	357
I/O time (secs)	34	74
#key comparisons	559×10^6	395×10^6

Table 1: Break down comparison on D_4

breakdown of the running times for D_4; we omit showing the results for the other data sets as they exhibit similar trends. Observe that although `SliceSort` incurred more I/O time than `HErMeS`, the CPU time spent by `SliceSort` is much smaller than that of `HErMeS`, resulting in `SliceSort` having an overall better performance than `HErMeS`. Comparing the number of passes required for sorting, `HErMeS` is more efficient as it requires only two passes: one pass to read the input tree to create initial sorted runs, and another pass to merge the initial sorted runs to create the output tree. In contrast, `SliceSort` requires one pass for each of its three phases. However, the number of key comparisons in `SliceSort` is much lower than that of `HErMeS` with `SliceSort` incurring about 30% fewer key comparisons compared to `HErMeS`. Furthermore, as explained, one key comparison in `SliceSort` is less costly than that in `HErMeS`. Consequently, the CPU time incurred by `SliceSort` is much smaller than that of `HErMeS`.

5. CONCLUSION

In this work, we introduced a novel technique, `SliceSort`, for sorting hierarchical data. In contrast to existing hierarchy-aware sorting methods which rely on subtree-based sorted runs, `SliceSort` employs a top-down, level-wise sorting technique to avoid the drawback of subtree-based sorting. Our experimental performance evaluation shows that `SliceSort` outperforms the state-of-art approach, `HErMeS`, by a significant factor.

Acknowledgment We would like to thank the authors of [8] for the code of `HErMeS`, and particularly Ioannis Koltsidas for his help in answering our questions.

6. REFERENCES

[1] http://www.w3.org/tr/xpath20/.
[2] Xquery 1.0: An XML query language. http://www.w3.org/tr/xquery/.
[3] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.
[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD*, pages 493–504, 1996.
[5] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE*, pages 41–52, 2002.
[6] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38, 2006.
[7] D. E. Knuth. *The art of computer programming, volume 3: sorting and searching*. 1998.
[8] I. Koltsidas, H. Müller, and S. D. Viglas. Sorting hierarchical data in external memory for archiving. *Proc. VLDB Endow.*, 1:1205–1216, 2008.
[9] A. Silberstein and J. Yang. NeXSort: Sorting XML in external memory. In *In ICDE*, pages 695–707, 2004.
[10] Y. Wang, D. J. Dewitt, and J.-Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In *ICDE*, pages 519–530, 2003.