NeXSort: Sorting XML in External Memory[*]

Adam Silberstein Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{adam,junyang}@cs.duke.edu

Abstract

XML plays an important role in delivering data over the Internet, and the need to store and manipulate XML in its native format has become increasingly relevant. This growing need necessitates work on developing native XML operators, especially for one as fundamental as sort. In this paper we present NeXSort*, an algorithm that leverages the hierarchical nature of XML to efficiently sort an XML document in external memory. In a fully sorted XML document, children of every non-leaf element are ordered according to a given sorting criterion. Among* NeXSort*'s uses is in combination with structural merge as the XML version of sort-merge join, which allows us to merge large XML documents using only a single pass once they are sorted.*

The hierarchical structure of an XML document limits the number of possible legal orderings among its elements, which means that sorting XML is fundamentally "easier" than sorting a flat file. We prove that the I/O lower bound for sorting XML in external memory is $\Theta(\max\{n, n\log_m(k/B)\})$, where n is the number of blocks in the input XML document, m is the number of main memory blocks available for sorting, B is the number of elements that can fit in one block, and k is the maximum fan-out of the input document tree. We show that NeXSort *performs within a constant factor of this theoretical lower bound. In practice we demonstrate, even with a naive implementation,* NeXSort *significantly outperforms a regular external merge sort of all elements by their key paths, unless the XML document is nearly flat, in which case* NeXSort *degenerates essentially to external merge sort.*

1. Introduction

Because of its portability and flexibility, XML is rapidly becoming a standard format for exchanging data over the Internet. However, as a relatively new data model, XML does not yet have as strong a theoretical foundation as the relational model. The current set of operations for XML data is still limited and often implemented on top of relational databases, which adds the overhead of converting XML to and from relational model. There is a need to develop native XML operations, and especially for one as fundamental as sort. To define and motivate the sorting problem, let us first consider the following example of merging two large XML documents.

Example 1.1 (Merging XML documents) Shown on the top of Figure 1 are two XML documents that we wish to combine. D_1 comes from the personnel department of a fictitious company and contains personal information of employees. D_2 comes from the payroll department and contains salary information. The company has many branches in different regions, and employees are organized accordingly. Intuitively, we want to merge matching employee elements in D_1 and D_2 into ones that contain both personal and salary information. The matching employee elements should have the same ID and belong to matching branch elements, which in turn belong to matching region elements.

This merge operation is analogous to performing a "join" (strictly speaking, an outerjoin) of two tables. A naive approach corresponds to the nested-loop join method. For each employee element, we find the matching element in the other document by traversing through the matching region and branch elements. This approach may be acceptable when D_1 and D_2 fit in internal memory. However, when dealing with large XML documents, this approach performs poorly because it generates element access patterns that do not at all correspond to the natural depth-first element ordering of disk-resident XML documents. For example, looking for a particular branch in a region requires scanning half of the region subtree on average, unless there is an additional index.

A much more efficient solution that works well in external memory corresponds to the sort-merge join method. We first sort both input documents such that for any company, region, or branch element, the list of child elements is ordered according to the same criterion for both documents. This ordering criterion should be based on the at-

[*] This work was supported by a National Science Foundation CAREER Award under grant IIS-0238386.

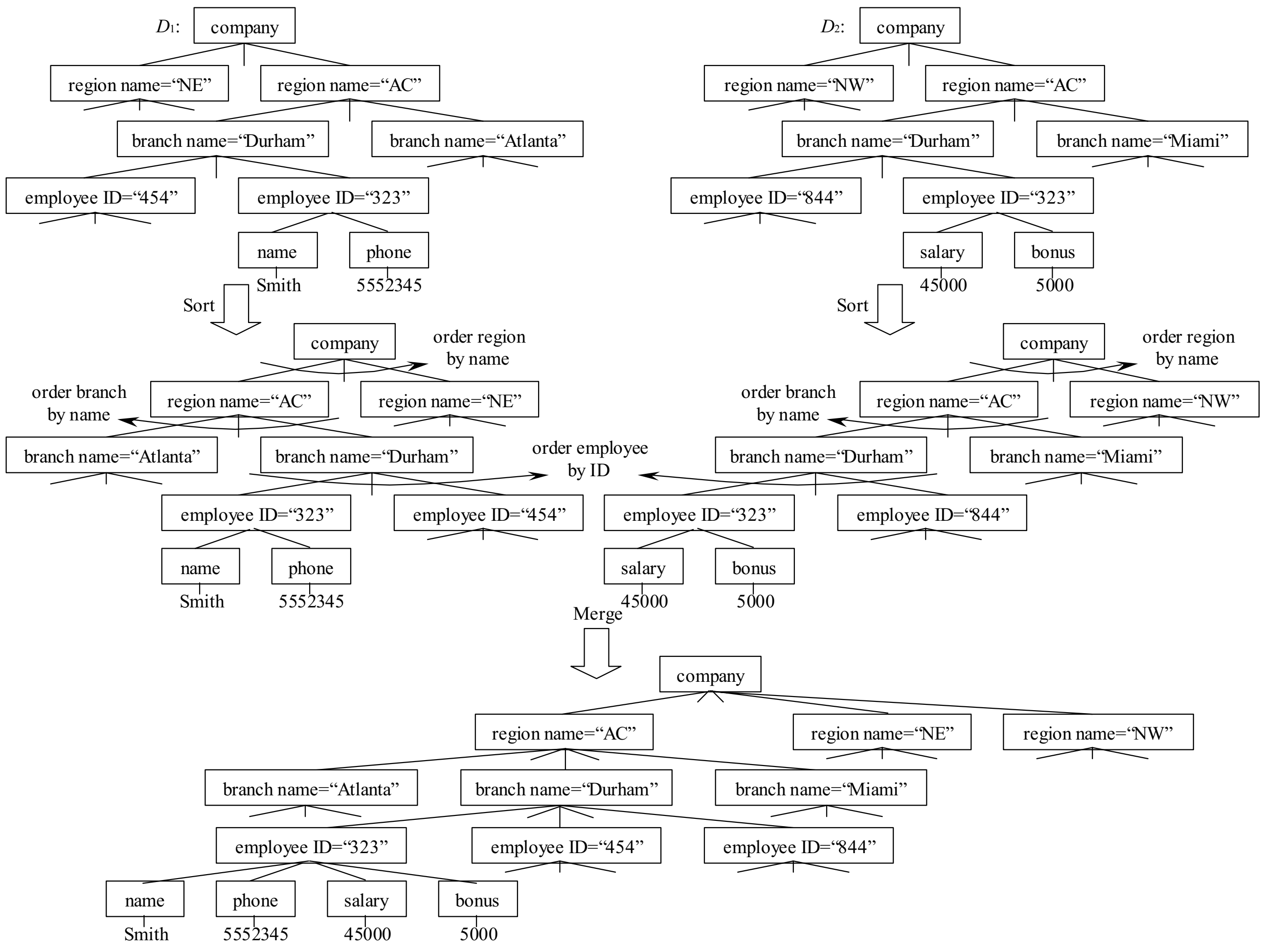

Figure 1. Merging XML documents.

tributes used in matching, as illustrated in Figure 1. Then, we can perform merge in a single pass over both sorted documents. This approach also can be adapted to preserve the original document ordering (by recording an additional sequence number attribute for each child element and performing a final sort according to this sequence number). □

Another application of sorting is processing batch updates to an existing XML document. Assume that the existing document is already sorted. We first sort the batch of updates according the same ordering criterion as the existing document. Then, we can process the batched updates in a way similar to merging them with the existing document. The result document remains sorted.

Note that we need to sort recursively on every level down to the one required by merge. Hence, what we mean by "sorting XML" is interestingly different from many other notions of sorting XML (or hierarchical data in general) discussed in the literature. We defer the detailed comparison to Section 2. For simplicity of presentation, we shall assume that we sort "from head to toe" (i.e., all the way down to the leaf level). That is, *in a fully sorted XML document, for every non-leaf element, the list of children is ordered according to some given criterion*. This assumption does not impact our discussion in any major way, and we show how to lift it in Section 3.2.

The major challenge in sorting XML in external memory is to minimize the number of I/O's while capitalizing on the hierarchical nature of XML. The document hierarchy limits the number of possible legal orderings among elements, because any legal ordering should preserve all parent-child relationships in the original document. For example, it does not make sense to permute an employee element out of its parent branch element to become an employee of another branch. Thus, sorting XML is fundamentally "easier" than sorting a flat file. In this paper, we show that the I/O lower bound for sorting XML in external memory is $\Theta(\max\{n, n\log_m(k/B)\})$, where n is the number of blocks in the input XML document, m is the number of main memory blocks available for sorting, B is the number of elements that can fit in a block, and k is the maximum fan-out of the document tree (i.e., the maximum number of children that any element can have). This bound is obviously lower (and potentially much lower) than the well-established bound of $\Theta(n \log_m n)$ for sorting a flat file, since $n = N/B > k/B$, where N is the number of elements in the input.

Existing approaches to sorting XML either do not work well in external memory or fail to take full advantage of the document structure. Consider the following two popular algorithms:

- **Internal-memory recursive sort.** We read in the entire input document and convert it to some internal-memory representation (such as DOM [1]). To sort a subtree rooted at an element, we first recursively sort the subtree rooted at every child element. Then, we sort the list of children, which simply involves reordering the pointers to them. This algorithm takes full advantage of the document structure but assumes that the entire document fits in internal memory; if not, the performance may suffer because of constant paging caused by excessive random accesses.
- **External merge sort.** We read in the entire input document and generate its alternative key-path representation as shown in Table 1. The *key path* of an element is the concatenation of the sort key values of all elements along the path from the root. We assume that the sort key value of an element is unique among its siblings (if not, we can make it unique by appending it with the element's location in the input). We sort the key-path representation using the well-known external merge-sort algorithm. Encoding of key paths ensures that the sort preserves all parent-child relationships. One problem with this approach is that it explicitly generates all key paths. If the input XML tree is tall, the key-path representation may potentially consume many times more space than the original input. Furthermore, the ubiquitous merge-sort algorithm does not take full advantage of the special key-path patterns that arise from the hierarchical nature of the input XML. Thus, the algorithm can only achieve the lower bound for sorting flat files, which is still higher than the lower bound for sorting XML.

In this paper, we propose an algorithm called NEXSORT (for ***Ne**sted Data and **X**ML **Sort**ing*) that solves the problems of the existing algorithms mentioned above. It is both I/O-efficient and structure-aware. It is simple to implement and has an I/O complexity that matches the theoretical lower bound (up to a constant factor) when either $k \geq B^\alpha$ or $M \geq B^\alpha$, where M is the number of elements that can fit in internal memory and $\alpha > 1$ is a constant. In practice, we demonstrate that even with a naive implementation, NEXSORT significantly outperforms the regular external merge sort in most cases. The obvious exception is when the input XML is almost flat. In this case, NEXSORT degenerates essentially to external merge sort.

Key path	Element content
`/`	`<company>`
`/NE`	`<region name="NE">`
`/AC`	`<region name="AC">`
`/AC/Durham`	`<branch name="Durham">`
`/AC/Durham/454`	`<employee ID="454">`
`/AC/Durham/323`	`<employee ID="323">`
`/AC/Durham/323/name`	`<name>Smith`
`/AC/Durham/323/phone`	`<phone>5552345`
`/AC/Atlanta`	`<branch name="Atlanta">`

Table 1. Key-path representation of D_1.

The rest of this paper is outlined as follows. Section 2 discusses related work. Section 3 describes the NEXSORT algorithm, proves its correctness, and discusses its extensions and possible improvements. Section 4 presents both the theoretical lower bound of XML sorting and the analysis of NEXSORT. Section 5 validates our theoretical analysis with experiments. Finally, we conclude the paper in Section 6, with some ideas on future work.

2. Related Work

XML was initially intended as a data exchange format, but as its popularity grows, so does the need to store and manipulate XML in a database. A number of research projects and commercial products have tackled this problem using different approaches. One approach, exemplified by [11, 16, 10], is to convert XML into relational data to be stored and queried using a relational database system in the backend. An alternative approach, exemplified by [13, 14], is to develop a native XML database with special support and optimization for XML data. Our work can be broadly classified as taking the second approach.

The problem of merging XML documents, one of the main motivations for sorting, is considered by Tufte and Maier [19, 18]. They focus on formally defining the semantics of the merge operator, which in general can be much more powerful than the simple version described in our example. The *deep union* operator proposed by Buneman et al. [8] is also a simpler version of the general merge operator. In [9], Buneman et al. present a method for archiving XML scientific data by merging new versions of a document into an archive document using the *Nested Merge* operation, which needs to sort the input documents at every level. Our work complements theirs by providing an I/O-efficient sort that supports more scalable merge operations. Our key-path representation of XML data is similar to the *path-set representation* in [8, 19].

A number of projects consider the problem of sorting XML or hierarchical data. However, as mentioned earlier,

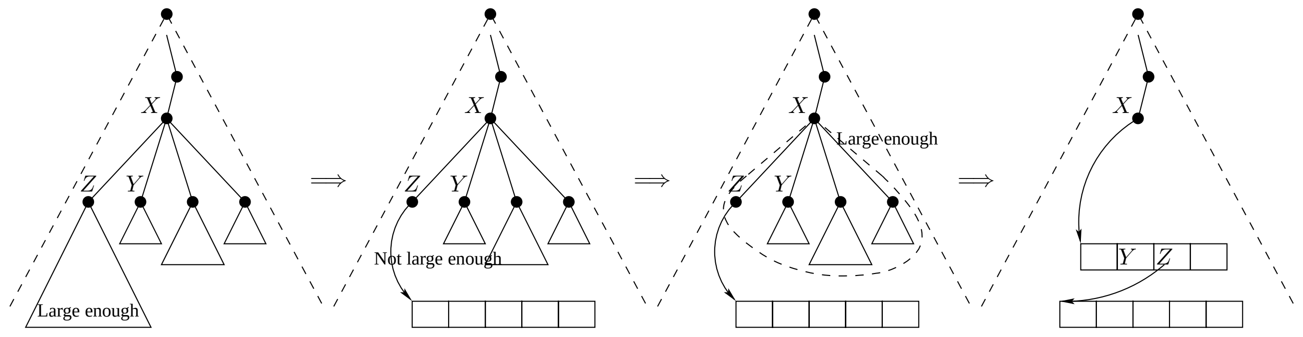

Figure 2. Sorting complete subtrees.

sorting means different things for these projects. Earlier work on implementing the NF^2 data model [12, 15] predates XML, but does address the problem of sorting hierarchical data. The NF^2 model allows attributes of tuples to contain complex objects including lists of other tuples. The sorting problem studied in this context focuses on handling complex ordering criteria that may be recursively defined (e.g., the ordering of employees is defined according to the ordering of their names, which is in turn defined according to the last and first name components). Sorting nested data recursively on every level is not considered.

XSort, described in [7], is an XML sorting algorithm that solves a different and simpler problem than NEXSORT. XSort traverses the document tree to some user-specified elements and then sorts their children; the child subtrees are not sorted recursively. XSort is implemented as standard external merge sort. The hierarchical nature of XML is irrelevant in this case because sorting is done on only one level. Obviously, XSort sorts less, and should complete in less time than NEXSORT. However, XSort does not lend itself well to solving the structural merge problem.

XQuery [4] includes an `order by` clause that allows results of queries (or subqueries) to be output in a specific order. This clause sorts the sequence of result XML fragments, but does not recursively sort the fragments themselves. Nevertheless, with the help of DTD's, "head-to-toe" XML sorts can be expressed in XQuery using explicitly specified nested `order by` clauses. Our work provides an efficient algorithm for processing such queries.

3. NEXSORT

On the highest level, NEXSORT consists of two phases: sorting and output. We first describe the general idea behind NEXSORT and then present the detailed algorithm in Section 3.1. Section 3.2 discusses possible extensions and optimizations to the algorithm.

In the sorting phase, NEXSORT scans the input document (in its natural depth-first order) and sorts complete subtrees of sufficient sizes. A subtree rooted at a given element encompasses the element together with all its descendents. During the scan, NEXSORT detects complete subtrees and decides whether to sort them. The decision is based on whether it is worth allocating disk block(s) to store the sorted result, which we call a *sorted run*. Specifically, NEXSORT sorts a subtree only if its size is no less than the *sort threshold*, t, a parameter tunable by the user. Typically, t should be no less than the block size in order to avoid uneconomical partial block I/O's.

Once a subtree has been sorted and written to disk as a sorted run, NEXSORT replaces the subtree with just its root element (with no descendents) and records in this element a pointer to the disk location of the sorted run. This operation is illustrated in Figure 2 for the subtree rooted at element Z. The operation effectively "collapses" an entire subtree into a single element, reducing the number of elements to be processed later by the algorithm. The sorted run will not be accessed again until the output phase.

Conversely, if the size of a subtree is below the sort threshold, NEXSORT simply leaves it alone and keeps scanning the input until a sufficiently large subtree is detected. This large subtree, which includes the small subtree that we have left alone, will then be sorted. This situation is illustrated in Figure 2 for the subtree rooted at Y, which is sorted as part of a large subtree rooted at X. Assuming that the maximum fan-out of the input document tree is k, we know that any subtree we sort must have size less than kt, because otherwise one of its child subtrees must have size no less than t and would have been sorted already.

Conceptually, NEXSORT processes the input document bottom-up, collapsing subtrees into their roots until only the root of the entire tree remains. At this point, we have a collection of sorted runs on disk, connected together by pointers (one for each subtree sort) into a tree as shown in Figure 3. In the output phase, NEXSORT simply performs a depth-first traversal of this tree to generate the final sorted document.

Intuitively, both NEXSORT and external merge sort follow a divide-and-conquer approach, breaking a large input into smaller pieces to sort first. However, external merge sort has to work to merge the results of smaller sorts. In con-

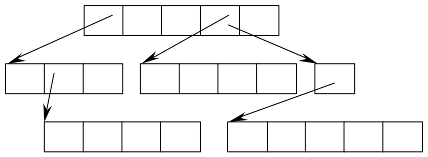

Figure 3. Tree of sorted runs.

trast, NEXSORT does not need to do any additional work on sorted subtrees (except outputting them in output phase). This difference is the primary source of NEXSORT's performance improvement over external merge sort.

3.1. Algorithm

Before presenting the algorithm, we first describe the three data structures involved.

- A *data stack* stores the elements to be sorted in the sorting phase.
- A *path stack* records, for the element currently being scanned in the sorting phase, the start locations of this element and its ancestors on the data stack. This information is used to compute the size of subtrees.
- An *output location stack* is used in the output phase to implement recursive processing of the tree of sorted runs.

Since these stacks may become quite large during execution (potentially larger than internal memory), we implement them as external-memory data structures, capable of paging blocks in and out of internal memory as needed. To bound the number of the I/O's, we assume a no-prefetch paging policy, i.e., a block of stack in external memory will not be paged in unless something on this block needs to be popped. In order to guarantee the worst-case performance (analyzed in Section 4.2), we assume that at least two blocks of internal memory are allocated to the path stack, and at least one block each is allocated to the data and output location stacks.

The detailed algorithm is in Figure 4. Pseudo-code for the sorting phase closely follows the earlier discussion. The loop on Line 2 can be implemented using a simple event-based XML parser (e.g., SAX [3]). Depending on the actual size of the subtree, sorting on Line 11 may use either an internal-memory algorithm or an external-memory algorithm, e.g., internal-memory recursive sort or key-path external merge sort discussed in Section 1, respectively.

Pseudo-code for the output phase may seem more complicated than necessary because it implements recursive tree traversal manually using a stack. The reason is that we wish to control I/O's explicitly in the rare case that the call stack grows bigger than the internal memory. This explicit control allows for a more rigorous analysis in Section 4.2.

3.2. Extensions and Optimizations

For simplicity of presentation and ease of analysis, the algorithm presented in Figure 4 omits many possible extensions and optimizations, which we now describe below.

Graceful degeneration into external merge sort. Clearly, if the input XML document is simply flat (i.e., with only two levels, the root and its children), we cannot expect NEXSORT to do better than external merge sort. However, we can at least make NEXSORT degenerate gracefully into external merge sort. The version of NEXSORT in Figure 4 does not have this property. When sorting a flat XML document, NEXSORT scans and pushes the entire document onto the data stack, only to pop the document out again for sorting. The initial pass is basically wasted. Fortunately, it is not difficult to modify NEXSORT to remove this overhead. The idea is to relax the requirement of sorting only complete subtrees. Whenever an incomplete subtree has filled internal memory, we sort it in internal memory and create an *incomplete sorted run*, instead of continuing to scan until the subtree is complete. Unlike regular sorted runs for complete subtrees, which do not need to be accessed again until the output phase, incomplete sorted runs for the same subtree must be merged to produce a regular, complete sorted run. Effectively, we have incorporated the first step of creating initial sorted runs for external merge sort into the loop of Line 2. With this optimization, NEXSORT completes with the same number of passes as external merge sort for flat input documents.

Depth-limited sorting. A feature mentioned in Section 1 but missing from the algorithm in Figure 4 is that of depth-limited sorting, useful under conditions where sorting XML "from head to toe" would be overkill. This feature allows a user to specify a depth at which to stop recursive sorting. Subtrees below this depth are treated as atomic units and left internally unsorted, although they are still sorted relative to the remainder of the document. Depth-limited sorting comes with the expectation that the user is particularly aware of the DTD or the contents of the XML documents. When merging two XML documents, for example, the user may know a depth below which no overlap of information is possible. Performing depth-limited sorting in this case can prevent a good amount of irrelevant sorting.

It is straightforward to modify NEXSORT to support depth-limited sorting. Let the root be at level 1. Suppose d is the sorting depth limit, i.e., subtrees rooted below level d do not need to be sorted. Before Line 9, we compute d_s, the level of the subtree being considered, as the length of the path stack plus 1. On Line 9, we add to the sorting condition an additional check that $d_s \leq d + 1$. This check avoids breaking up subtrees below the sorting depth limit unnecessarily. Furthermore, the sorting algorithm on Line 11 needs to support depth-limited sorting as well. To be specific, it

```
// Sorting phase:
Initialize both data and path stacks to empty;                                          (1
Loop until end of the input XML:                                                        (2
    Read a unit of XML data (i.e., a start tag, and end tag, or a piece of text);       (3
    Push the data onto data stack;                                                      (4
    If a start tag was encountered:                                                     (5
        Push its location on the data stack onto the path stack;                        (6
    If an end tag was encountered:                                                      (7
        Pop location $l$ from path stack;                                               (8
        If the difference between the current location of data stack and $l$ is greater than $t$, or
        if $l = 1$ (we have finished scanning the input):                               (9
            Pop the subtree (starting from location $l$) from data stack;               (10
            Sort this subtree and write the result in a sorted run;                     (11
            Push the root element of the subtree back onto data stack,
                together with a pointer to the sorted run;                              (12
// Output phase:
Initialize output location stack with entry $(s, 0)$, where
    $s$ is the pointer to the root sorted run (found on the top of data stack);         (13
While output location stack is not empty:                                               (14
    Pop ($s$, $l$) from output location stack;                                          (15
    Start after location $l$ in the sorted run pointed to by $s$, loop until end of the sorted run:   (16
        Read a unit of XML data;                                                        (17
        If a pointer $s'$ to another sorted run was encountered:                        (18
            Push ($s$, the current location within the sorted run) onto output location stack;   (19
            Go to Line 16 with $s = s'$ and $l = 0$;                                    (20
        Output the data just read;                                                      (21
```

Figure 4. Algorithm NEXSORT.

only needs to sort up to the top $d + 1 - d_s$ levels of the subtree. If $d_s = d + 1$, no sorting is needed but the subtree is still written to disk, ensuring that we do not "carry" large subtrees along that increase the costs of higher-level sorts.

Complex ordering criteria. So far, we have assumed simple ordering criteria that can be evaluated for each element using its tag name and/or attribute values. More complex ordering criteria often arise and may need to be evaluated in the context of subtrees, e.g., order employee elements by `personalInfo/name/lastName`. In general, if an ordering expression for an element can be computed from its ancestors and/or a single pass over its subtree using constant space, then NEXSORT can be adapted to evaluate this expression. During the sorting phase, we can augment the path stack with copies of the elements along the current path, which provide the necessary context for evaluating expressions based on ancestors. To handle expressions over subtrees, we can further augment the path stack with the expressions to be evaluated for the elements along the current path. We evaluate these expressions while processing the subtree. Once we finish evaluating an expression on the path stack, we replace it by its result. When the end tag of any element e is encountered, we will have processed all the data necessary for evaluating e's ordering expression, and its result will be found on the path stack. This result can be pushed onto the data stack with the end tag and used for sorting.

The above approach does not work if an element e's ordering expression cannot be evaluated with a single pass over e's subtree (e.g., median of `employee/salary`), or if the ordering expression references data other than e's descendents and ancestors (e.g., an XPath expression that follows `IDREF`'s). We plan to investigate such ordering expressions as future work.

XML compaction techniques. XML is not a particularly compact data representation format. A document usually contains many repeated occurrences of labels such as tag and attribute names, which can be compressed in some manner to reduce the size of input for sorting. For example, each unique string can be converted to an integer before sorting and back during output. The availability of a DTD can greatly simplify this conversion.

Also, labels inside end tags can be eliminated since they merely repeat the same information in matching start tags. In fact, we can eliminate end tags altogether if we keep level numbers with start tags. Specifically, when we encounter a start tag while scanning the input XML document, we push the tag onto the data stack together with the level number of the element (which can be determined from the current length of the path stack). End tags in the input still trigger sorting decisions, but are not themselves pushed onto the data stack. During the output phase, end tags can be recovered using the intuition that in a series of start tags, any transition from a start tag on level l_1 to a start tag on the same or a higher level l_2, where $l_2 \leq l_1$, must have $l_1 - l_2 + 1$ end tags in between to close elements on lower levels. To be able to output these end tags, we maintain a structure similar to the path stack, which records the tag names and level numbers of unclosed open tags during output. Upon seeing the transition from a start tag on level l_1 to a start tag on level $l_2 \leq l_1$, we pop the top $l_1 - l_2 + 1$ entries from the stack and generate their corresponding end tags.

4. I/O Analysis

We now present the I/O complexity analysis of NeXSort and the XML sorting problem in general. The bounds are given in terms of the number of I/O's (or disk accesses), the primary metric for measuring the performance of external-memory algorithms. We use the standard notation of the external memory model [5], which includes the following parameters:

- N: number of elements in the input XML document;
- B: number of elements that can fit in one block;
- M: number of internal memory blocks available.

In addition, to capture roughly the "flatness" of the input XML document, we use the following parameter:

- k: maximum fan-out of the input XML document, i.e., the maximum number of children for any element.

To simplify the analysis, we assume that all elements have equal size. For a non-leaf element, we define its size to be the total size of its start and end tags. This size includes the element's label name, attribute names, and attribute values; it does not recursively include the size of the children. The assumption of equal-sized elements does not limit the applicability of our analysis at all. The reason is that we can consider, for the purpose of analysis, any large element X as a chain of smaller, equal-sized elements $X_0, X_1, X_2, \ldots$, where each element (X_i) is the only child of the previous element (X_{i-1}). NeXSort would still work in the same way. Furthermore, N now correctly captures the total size of the input document, and k remains the same as the original document (since the chain we introduced has a fan-out of 1 and therefore cannot increase the maximum global fan-out). Note that leaf elements with long text contents can be treated in the same way for the purpose of analysis.

4.1. Lower Bound for Sorting XML

We now establish a lower bound for sorting XML in external memory, using the comparison model of computation. First, we determine the number of possible outcomes of sorting an XML document. Next, we derive the minimum of I/O's required by any algorithm to pick the correct sorting outcome among all possible ones.

The number of possible outcomes of sorting a flat file of N elements is $N!$. However, an XML document with the same size may have far fewer possible sorting outcomes, since any legal ordering of the elements must preserve all parent-child relationships in the original document. With the following two lemmas, we determine the number of possible sorting outcomes for an XML document with N elements and a maximum fan-out of k. Lemma 4.1 shows that a particular XML structure (one that tries to reach the maximum fan-out as much as possible) offers the maximum number of possible sorting outcomes. This structure would be the one that an adversary would pick for any XML sorting algorithm. Lemma 4.2 then computes the number of possible sorting outcomes for this structure.

Lemma 4.1 An XML document with N elements and a maximum fan-out of no more than k has the maximum number of possible sorting outcomes if at most one element has neither 0 nor k children. □

Proof: Suppose, on the contrary, this document has two elements, X and Y, with x and y children respectively, where $0 < x \leq y < k$. The number of possible sorting outcomes for this document is $P \cdot (x!) \cdot (y!)$, where P is the contribution from the rest of the document structure (or more specifically, the product of factorials of all other elements' fan-outs). We construct a new document from the original one as follows. If $x = 1$, we promote X's only child to take X's place, and then make X a child of Y. If $x > 1$, there must exist a child of X that is not Y or Y's ancestor; we transfer this child (together with the subtree rooted at this child) from X to Y. In either case, the new document still has N elements, and its maximum fan-out is still no more than k. The number of possible sorting outcomes for the new document is $P \cdot (x-1)! \cdot (y+1)!$. Compared with the original document, the number has increased by a factor of $\frac{y+1}{x} > 1$, which contradicts the assumption that the original document has the maximum number of possible sorting outcomes. □

Lemma 4.2 The number of possible sorting outcomes for an XML document described in Lemma 4.1 is $(k!)^{\lfloor (N-1)/k \rfloor} \cdot ((N-1) \bmod k)!$. □

Proof: It is easy to see the total number of possible outcomes is the product of factorials of the all fan-outs in the document tree. The sum of all fan-outs is equal to the total number of edges in the tree, which is $N - 1$ for any tree. For the tree described in Lemma 4.1, all non-zero fan-outs are equal to k except one. Therefore, there are $\lfloor (N-1)/k \rfloor$ fan-outs that are equal to k and possibly one that is equal to $(N-1) \bmod k$. Taking the product of factorials of all these fan-outs, we get $(k!)^{\lfloor (N-1)/k \rfloor} \cdot ((N-1) \bmod k)!$. □

Next, we derive the minimum number of I/O's required for any algorithm to choose the correct sorting outcome from all possible ones. First, we analyze how many possible outcomes an algorithm can possibly distinguish after performing a given number of I/O's. The following lemma and its proof technique are both from [5], where they were used in deriving a lower bound for sorting flat files.

Lemma 4.3 (Aggarwal and Vitter [5]) After reading T blocks from external memory, any algorithm can generate at most $(B!)^{N/B}\binom{M}{B}^T$ possible outcomes. □

Proof: Consider an input of one block containing B elements from external memory. If these B elements were previously output together during an earlier output, we may assume that their relative ordering is already known (because all comparisons among them would have been carried out while they were in internal memory). Before the input, there are at most $M - B$ other elements in internal memory, and their relative ordering is also known. After the input, the algorithm can determine the ordering of all M elements in internal memory, giving rise to $\binom{M}{B}$ possible outcomes. If the B elements were not previously output together (i.e., they are read for the first time), then there is an additional factor of $B!$ for possible outcomes; this case arises only once for each of the N/B input blocks. To summarize, the total number of possible outcomes that can be generated after reading T blocks is $(B!)^{N/B}\binom{M}{B}^T$. □

Combining Lemmas 4.2 and 4.3, we obtain the following lower bound for XML sorting. Because of limited space, we only describe the crux of the proof; the complete, rigorous proof is in the full version of this paper [17].

Theorem 4.4 In the worst case, the minimal number of I/O's required to sort an XML document with N elements and a maximum fan-out of k is

$$\Theta\left(\max\{\frac{N}{B}, \frac{N}{B}\log_{\frac{M}{B}}\frac{k}{B}\}\right).$$

□

Proof sketch: According to Lemma 4.2, the number of possible outcomes in the worst case is roughly $(k!)^{N/k}$. According to Lemma 4.3, the number of I/O's, T, required to generate this number must satisfy the following inequality:

$$(B!)^{N/B}\binom{M}{B}^T \geq (k!)^{N/k}.$$

Taking logarithms of both sides and solving for T, we get

$$T \geq \frac{\frac{N}{k}\ln(k!) - \frac{N}{B}\ln(B!)}{\ln\binom{M}{B}}.$$

Since $\binom{M}{B} \leq \frac{M^B}{B!}$, we have

$$T \geq \frac{\frac{N}{k}\ln(k!) - \frac{N}{B}\ln(B!)}{B\ln M - \ln(B!)} = \frac{N}{B} \cdot \frac{\frac{1}{k}\ln(k!) - \frac{1}{B}\ln(B!)}{\ln M - \frac{1}{B}\ln(B!)}.$$

Using Stirling's formula, we can simplify the above to

$$\Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{k}{B}\right).$$

Finally, since any algorithm must read the entire input at least once, the bound should be no less than $\frac{N}{B}$. This check ensures that the bound is meaningful when k is small. □

4.2. Performance of NEXSORT

We now turn to the I/O analysis of NEXSORT with a sort threshold of $t \geq B$ given an input XML document with N elements and a maximum fan-out of k. Because of the hierarchical structure of XML, the I/O pattern of NEXSORT is significantly more complicated than that of regular external merge sort. NEXSORT needs to perform I/O's not only for sorting per se, but also for bookkeeping, such as paging of various stacks. For example, since the path stack and the output location stack may get arbitrarily deep for a tall XML tree, parts of them may need to be paged in and out of the internal memory as needed. A careful analysis of their associated costs is not trivial and in fact is quite a stimulating task.

In the following, we first provide a detailed breakdown of the I/O's incurred by NEXSORT, together with an overview of the analysis. We then present the worst-case I/O complexity of NEXSORT in Theorem 4.5, followed by a comparison with the theoretical lower bound derived in Section 4.1. Finally, we present the lemmas and proofs that support Theorem 4.5.

During the sorting phase, NEXSORT incurs the following I/O costs:

- *Reading the input*. This cost is obviously $O(N/B)$.
- *Sorting subtrees* (excluding costs related to operations on the data stack but including the cost of writing out sorted runs). The total number of I/O's spent

on subtree sorts is $O(\frac{N}{B}\log_{\frac{M}{B}}\lceil\frac{\min\{kt,N\}}{B}\rceil)$ according to Lemma 4.9, to be presented later in this section. This cost is the most significant component in the overall cost of NEXSORT.

- *Paging the data stack*. This cost is $O(N/B)$, as we will see in Lemma 4.10.
- *Paging the path stack*. This cost is also $O(N/B)$, according to Lemma 4.11.

During the output phase, NEXSORT incurs the following I/O costs:

- *Reading blocks in sorted runs*. Although each sorted-run block may be read multiple times, the total number of accesses is still $O(N/B)$ according to Lemma 4.12.
- *Paging the output location stack*. This cost is $O(N/t)$, as we will see in Lemma 4.13.
- *Writing the output*. This cost is obviously $O(N/B)$.

Combining these results, we have the following theorem:

Theorem 4.5 In the worst case, the total number of I/O's that NEXSORT uses to sort an XML document with N elements and a maximum fan-out of k is

$$O\left(\frac{N}{B}+\frac{N}{B}\log_{\frac{M}{B}}\lceil\frac{\min\{kt,N\}}{B}\rceil\right),$$

where t is the sort threshold used by NEXSORT. □

Proof: Follows directly from the analysis above and Lemmas 4.9– 4.13. □

With the natural choice of sort threshold $t = B$, the bound above becomes $O(\frac{N}{B}+\frac{N}{B}\log_{M/B}\min\{k,N/B\})$. Compared with the theoretical lower bound given in Theorem 4.4, the only real difference is the arguments to the logarithm function (k versus k/B). Although these arguments differ by a factor of B, the two bounds differ only by a constant factor if $k \geq B^\alpha$ or $M \geq B^\alpha$ for some constant $\alpha > 1$. The reason is the following. If $k \geq B^\alpha$, $\log k = \frac{1}{\alpha-1}\log(k^\alpha/k) \leq \frac{1}{\alpha-1}\log(k^\alpha/B^\alpha) = \frac{\alpha}{\alpha-1}\log(k/B)$. If $k < B^\alpha$ and $M \geq B^\alpha$, both $\log_{M/B} k$ and $\log_{M/B}(k/B)$ are bounded by constants, so both bounds become $\Theta(N/B)$. More specifically, $\log_{M/B} k < \log_{B^\alpha/B} B^\alpha = \frac{\alpha}{\alpha-1}$, and $\log_{M/B}(k/B) < \log_{B^\alpha/B}(B^\alpha/B) = 1$.

Before presenting the lemmas and their proofs in support of Theorem 4.5, we need some additional notation. Suppose that NEXSORT performs a total of x subtree sorts with $s_1, \ldots, s_x$ elements each, where $B \leq t \leq s_i < \min\{kt,N\}$ for $i = 1,\ldots,x$, as discussed in Section 3. The first three lemmas provide some useful facts that are used in other proofs.

Lemma 4.6 $\sum_{i=1}^{x} s_i = N-1+x$. □

Proof: Follows directly from the observation that the i-th subtree sort has the effect of collapsing s_i elements into one, and that NEXSORT completes when all N elements have been collapsed into one. □

Lemma 4.7 $x = O(N/t) = O(N/B)$. □

Proof: Since $s_i \geq t$, $N-1+x = \sum_{i=1}^{x} s_i \geq xt$, which implies that $x \leq \frac{N-1}{t-1}$. Because $t \geq B$, we have $x = O(N/t) = O(N/B)$ for any reasonable value of B (i.e., greater than 1). □

Lemma 4.8 The total number of blocks taken by all sorted runs is $\sum_{i=1}^{x}\lceil s_i/B\rceil = O(N/B)$. □

Proof: $\sum_{i=1}^{x}\lceil s_i/B\rceil < \sum_{i=1}^{x}(s_i/B+1) = \frac{N-1+x}{B}+x$ according to Lemma 4.6, which in turn is $O(N/B)$ according to Lemma 4.7. □

Lemma 4.9 The total number of I/O's spent by NEXSORT on subtree sorts is $O(\frac{N}{B}\log_{\frac{M}{B}}\lceil\frac{\min\{kt,N\}}{B}\rceil)$. □

Proof: Consider the x subtree sorts. The i-th subtree sort requires $O(\lceil\frac{s_i}{B}\rceil\log_{\frac{M}{B}}\lceil\frac{s_i}{B}\rceil)$ I/O's. Therefore, the total number of I/O's incurred by subtree sorts is

$$\begin{aligned}
&\sum_{i=1}^{x} O\left(\lceil\frac{s_i}{B}\rceil\log_{\frac{M}{B}}\lceil\frac{s_i}{B}\rceil\right)\\
=\;& O\left(\sum_{i=1}^{x}\left(\lceil\frac{s_i}{B}\rceil\log_{\frac{M}{B}}\lceil\frac{\min\{kt,N\}}{B}\rceil\right)\right)\\
=\;& O\left(\left(\sum_{i=1}^{x}\lceil\frac{s_i}{B}\rceil\right)\log_{\frac{M}{B}}\lceil\frac{\min\{kt,N\}}{B}\rceil\right)\\
=\;& O\left(\frac{N}{B}\log_{\frac{M}{B}}\lceil\frac{\min\{kt,N\}}{B}\rceil\right). \quad \text{(Lemma 4.8)}
\end{aligned}$$

□

Lemma 4.10 The total number of I/O's spent by NEXSORT on paging the data stack is $O(N/B)$. □

Proof: We focus on counting the number of page-in's, since the number of page-out's cannot be more than this number (every block that is paged out is eventually paged in). A block of the data stack is paged in only in two cases: (1) the block needs to be read and sorted as part of a subtree sort, and (2) the block needs to be written with the pointer to a sorted run. For the first case, the total number of I/O's cannot be more than $\sum_{i=1}^{x}(\lfloor\frac{s_i}{B}\rfloor+2) \leq 2x+\sum_{i=1}^{x}\frac{s_i}{B} = 2x+\frac{N-1+x}{B}$ (Lemma 4.6). For the second case, the total number of I/O's cannot be more than x. Therefore, the grand total is no more than $3x+\frac{N-1+x}{B} = O(N/B)$ (Lemma 4.7). □

Lemma 4.11 Assume that NeXSort devotes two blocks of internal memory to the path stack, and each block can hold B path elements. The total number of I/O's spent by NeXSort on paging the path stack is $O(N/B)$. □

Proof: Again, we focus on counting the number of page-in's, since the number of page-out's cannot be more than this number. A block p of the path stack is paged in only when we have encountered the end tag of the element (let us call it e) pointed to by the B-th (last) path element on p. Otherwise, there is no need to access p. Furthermore, with two blocks of internal memory devoted to the path stack, p cannot have been paged out (and therefore do not need to be paged in) unless e has at least one descendent more than distance B away. Therefore, the total number of pages-in's is equal to the number of what we call *fringe elements* in the input document. Assuming that the root element is at level 1, a fringe element is a level-(iB) element ($i \geq 1$) with at least one descendent more than distance B away. Note that for each fringe element at level iB we can choose $B - 1$ descendents by following the path to the descendent that is more than distance B away, and stopping right above level $(i + 1)B$. This set of $B - 1$ descendents do not include fringe elements, nor descendents chosen for other fringe elements. Hence, the total number of elements in the input document must be no less than B times the number of fringe elements, which implies that the number of fringe elements cannot be more than N/B. □

Lemma 4.12 The total number of I/O's spent by NeXSort on reading blocks in sorted runs is $O(N/B)$. □

Proof: Consider a block b in a sorted run. Suppose that the number of pointers (to other sorted runs) found in b is $p(b)$. The number of times that b is accessed is $1 + p(b)$, because NeXSort remains on the same block unless a pointer is encountered or the block is finished. Therefore, the total number of accesses to sorted-run blocks can be obtained by summing $1 + p(b)$ over each sorted-run block b. The result is the total number of sorted-run blocks plus the total number of pointers found in sorted-run blocks. The first term is $O(N/B)$ according to Lemma 4.8. The second term is the number of sorted runs (x) minus one (the root run), which is $O(N/B)$ according to Lemma 4.7. Hence, the total is $O(N/B)$. □

Lemma 4.13 The total number of I/O's spent by NeXSort on paging the output location stack is $O(N/t)$. □

Proof: In the worst case, the number of I/O's incurred by paging is no more than the number of push and pop operations. We focus on counting the number of pushes, since it should be the same as the number of pops. Note that NeXSort only pushes a location to the output location stack when a pointer to a sorted run is encountered. Therefore, the total number of pushes is the total number of pointers to sorted runs, which is the number of sorted runs (x) minus one (the root run). According to Lemma 4.7, this number is $O(N/t)$. □

5. Experimental Results

To evaluate the effectiveness of NeXSort we have implemented both NeXSort and external merge sort using *TPIE* (***T**ransparent **P**arallel **I**/O **E**nvironment* [6]). TPIE supports explicit control and detailed accounting of I/O operations, as well as easy adjustment of the amount of main memory available to application programs. Our experiments are carried out on an 800 MHz Pentium III computer running Linux 2.4.21 with 1 GB of RAM. We use TPIE to set the application memory to be smaller than this amount in all experiments. The size of one memory block is 64 KB.

We implement some of the XML compaction techniques in Section 3.2, including compression of tag names and elimination of end tags, for both NeXSort and external merge sort. We have not implemented the optimization that allows NeXSort to degenerate gracefully into external merge sort. Thus, we expect NeXSort to perform worse than external merge sort for inputs that are nearly flat. Overall, we have not hand-optimized our code. Coupled with the overhead of our runtime environment (a cost of its flexibility) and low-end hardware, our code may not deliver good absolute running times, but the relative running times compared with external merge sort still clearly demonstrate the benefits of NeXSort.

We generate our test data using the IBM alphaWorks XML Generator [2] and an XML generator that we developed. The IBM generator allows us to specify height and maximum fan-out for the document to be generated. The fan-out of each element is a random number between 1 and the specified maximum. Our custom generator allows us to specify the exact fan-out for each level, giving us more precise control over the shape and the size of the generated document. All test data has an average element size of about 150 bytes.

Effect of sort threshold. Recall that NeXSort only sorts a subtree if its size is no less than the sort threshold t. Our asymptotic analysis in Section 4.2 suggests that it is best to set t as small as possible (but no less than the block size) in order to reduce the logarithmic term in the bound. However, our intuition suggests that a somewhat larger threshold might yield better performance by cutting down on the overhead of doing a large number of smaller sorts, while still avoiding large external-memory sorts that require extra passes. Experimental results (not shown here

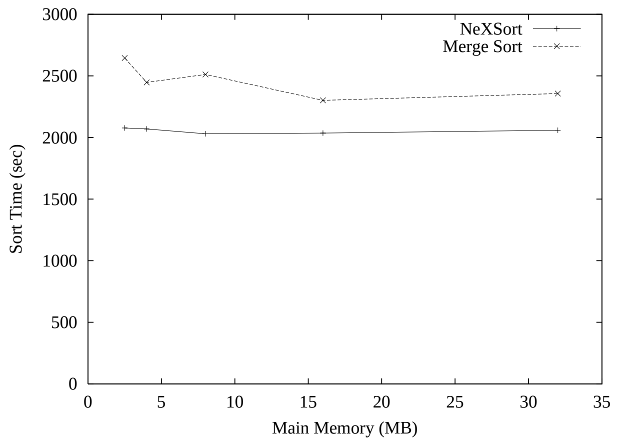

Figure 5. Effect of main memory size.

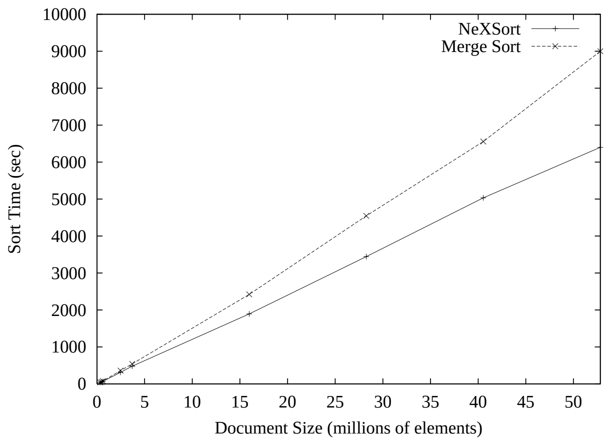

Figure 6. Effect of input size with constant maximum fan-out.

due to space constraints) confirm our intuition. When the threshold is small, there is a significant amount of overhead caused by many small sorts. When the threshold becomes too large, performance begins to degrade because NEXSORT is sorting large subtrees with multiple levels using external merge sort, which fails to take advantage of the structure within subtrees. Details can be found in the full version of this paper [17]. For the following experiments, we set the threshold to be roughly twice the block size, which works well for most inputs.

Effect of main memory size. In this experiment, we run NEXSORT and external merge sort with a range of main memory sizes, with the dual purposes of comparing the performance of the two algorithms and studying the relative impact of main memory on each of them. The input document is the same one used in the previous experiment. The running times are graphed in Figure 5. Overall, external merge sort is slower than NEXSORT by 13% to 27%. As memory decreases, NEXSORT running time increases only marginally. In contrast, external merge sort running time increases more dramatically, especially when decreased memory forces additional passes. This result demonstrates an important difference between the two algorithms. When fanouts are small, NEXSORT is not very dependent on main memory size, because it performs few sorts that actually require all of memory. External merge sort, however, always needs as much memory as possible.

Effect of input size with constant maximum fan-out. Next, we use our custom generator to construct a series of XML documents with increasing sizes from 33 MB to 7.9 GB. Meanwhile, the maximum fan-out is capped at 85 to ensure that the input exhibits enough “hierarchicalness” and does not become array-like as it grows in size. We run NEXSORT and external merge sort with 3 MB of main memory and plot the running times in Figure 6. As the size of the input increases, the running time of NEXSORT increases roughly linearly, confirming our performance analysis of NEXSORT in Section 4.2, which derives a logarithmic multiplicative factor of $\log_{M/B}\lceil kt/B\rceil$ that is independent of the input size in this case. On the other hand, Figure 6 gives a slight indication that the running time of external merge sort increases superlinearly. The jump is noticeable where the input reaches 1 million elements, which is roughly when the sort goes from two-pass to three-pass, and where the input reaches 50 million, when the sort becomes four-pass. This result is expected because external memory merge sort has a logarithmic multiplicative factor of $\log_{M/B} N/B$, which does depend on the input size.

Effect of input tree shape. In this experiment, we change the tree shape of the input document while keeping its size roughly constant. We use our custom generator to construct five input documents with the number of tree levels ranging from 2 to 6. Within each document, the fan-out is near uniform across all elements. Table 2 summaries the characteristics of these documents. Figure 7 plots the running times of NEXSORT and external merge sort with 4 MB of main memory. As the input tree grows taller, external merge sort performs slightly worse because of the overhead of generating and comparing longer key paths. Note that the two-level input is essentially a flat file. As discussed earlier, we expect our implementation of NEXSORT to perform worse than merge sort in this case since we have not implemented the optimization that allows NEXSORT to degenerate into external merge sort. However, when the “hierarchicalness” of input reaches a critical level (4 in this case), the performance of NEXSORT significantly improves due to the decreased maximum fan-out. Between the critical levels, NEXSORT shows little improvement (and in fact occasionally slight degradation) in performance. There is an explanation for this interesting phenomenon. Namely, increased tree height does not necessarily translate into smaller subtree sorts; instead, the fan-out may become just small

Height	Fan-out for each level (starting from root)	Size (number of elements)
2	3000000	3000001
3	1733, 1733	3005023
4	144, 144, 144	3006865
5	41, 41, 42, 42	3037609
6	19, 19, 20, 20, 20	3040001

Table 2. Input document shapes.

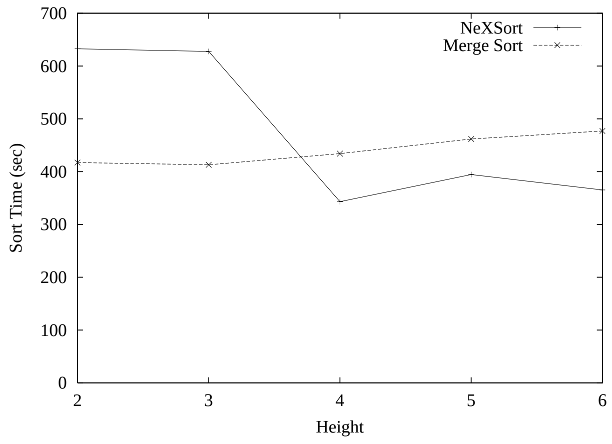

Figure 7. Effect of tree shape.

enough to force a subtree to be sorted as part of its parent, which could mean that NEXSORT is sorting larger subtrees than before.

6. Conclusion and Future Work

We have presented NEXSORT, an I/O-efficient XML sorting algorithm that leverages the hierarchical nature of XML to achieve better performance than regular external merge sort. We have also derived a lower bound for sorting XML, and shown that NEXSORT performs within a constant factor of it. Our experimental results demonstrate that NEXSORT outperforms external merge sort, except when the input XML is close to a flat file of elements. While discussed in the context of XML, our results apply to any type of nested data in general.

We conjecture that the constant-factor difference between the theoretical lower bound and the performance of NEXSORT can be made smaller when $k < B$ and M is small. In this case, the dominating cost is not sorting but permuting the input to generate the output. On one hand, we will try to improve the lower bound by considering the cost of permutation in external memory; on the other hand, we will try to improve the upper bound by further optimizing NEXSORT. We also plan to generalize NEXSORT to support more complex ordering criteria.

References

[1] Document Object Model (DOM). http://www.w3.org/DOM.

[2] IBM alphaWorks XML generator. http://www.alphaworks.ibm.com/tec/xmlgenerator.

[3] SAX: Simple API for XML. http://www.saxproject.org.

[4] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery.

[5] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[6] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickeremesinghe. TPIE user manual and reference. http://www.cs.duke.edu/TPIE.

[7] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML toolkit for scalable XML stream processing. In *PlanX*, 2002.

[8] P. Buneman, A. Deutsch, and W. C. Tan. A deterministic model for semi-structured data. In *Proc. of the 1999 Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.

[9] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, Madison, Wisconsin, June 2002.

[10] T. Fiebig and G. Moerkotte. Evaluating queries on structure with extended access support relations. In *Proc. of the 2000 Intl. Workshop on the Web and Databases*, Dallas, Texas, May 2000.

[11] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[12] L. Wegner K. Kuspert, G. Saake. Duplicate detection and deletion in the extended NF^2 data model. In *Proc. of the 1989 Intl. Conf. on Foundations of Data Organization and Algorithms*, 1989.

[13] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[14] J. McHugh and J. Widom. Query optimization for XML. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, pages 315–326, 1999.

[15] G. Saake, V. Linnemann, P. Pistor, and L. Wegner. Sorting, grouping and duplicate elimination in the advanced information management prototype. In *Proc. of the 1989 Intl. Conf. on Very Large Data Bases*, pages 307–316, 1989.

[16] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *The VLDB Journal*, 10(2–3):133–154, 2001.

[17] A. Silberstein and J. Yang. NeXSort: Sorting XML in external memory. Technical report, Duke University, July 2003. http://www.cs.duke.edu/dbgroup/papers/2003-sy-nexsort.pdf.

[18] K. Tufte and D. Maier. Aggregation and accumulation of XML data. *IEEE Data Engineering Bulletin*, 24(2):34–39, 2001.

[19] K. Tufte and D. Maier. Merge as a lattice-join of XML documents, 2002. http://www.cs.wisc.edu/niagara/papers/paper608.pdf.