An Adaptive Algorithm for Splitting Large Sets of Strings and Its Application to Efficient External Sorting

Tatsuya Asai† **Seishi Okamoto**† **Hiroki Arimura**‡

† Knowledge Research Center, Fujitsu Laboratories Ltd.
4–1–1 Kamikodanaka, Nakahara-ku, Kawasaki 211–8588, Japan
Email: asai.tatsuya@jp.fujitsu.com
Email: seishi@labs.fujitsu.com

‡ Graduate School of Information Science and Technology, Hokkaido University
Kita 14–jo, Nishi 9–chome, Sapporo 060–0814, Japan
Email: arim@ist.hokudai.ac.jp

Abstract

In this paper, we study the problem of sorting a large collection of strings in external memory. Based on adaptive construction of a summary data structure, called *adaptive synopsis trie*, we present a practical string sorting algorithm DistStrSort, which is suitable to sorting string collections of large size in external memory, and also suitable for more complex string processing problems in text and semi-structured databases such as counting, aggregation, and statistics. Case analyses of the algorithm and experiments on real datasets show the efficiency of our algorithm in realistic setting.

Keywords: external string sorting, trie sort, semi-structured database, text database, data splitting

1 Introduction

Sorting strings is a fundamental task of string processing, which is not only sorting the objects in associated ordering, but also used as a basis of more sophisticated processing such as grouping, counting, and aggregation. With a recent emergence of massive amount of *semi-structured data* (Abiteboul *et al.* 2000), such as plain texts, Web pages, or XML documents, it is often the case that we would like to sort a large collection of strings that do not fit into main memory. Moreover, there are potential demands for flexible methods tailored for semi-structured data beyond basic sorting facilities such as statistics computation, counting, and aggregation over various objects including texts, data, codes, and streams (Abiteboul *et al.* 2005, Stonebraker *et al.* 2005).

In this paper, we study efficient methods for sorting strings in external memory. We present a new approach to sorting a large collection of strings in external memory. The main idea of our method is to generalize distribution sort algorithms for collections of strings by the extensive use of *trie* data structure. To implement this idea, we introduce a synopsis data structure for strings, called a *synopsis trie*, for approximate distribution, and develop an adaptive construction algorithm for synopsis tries for a large collection of strings with the idea of adaptive growth.

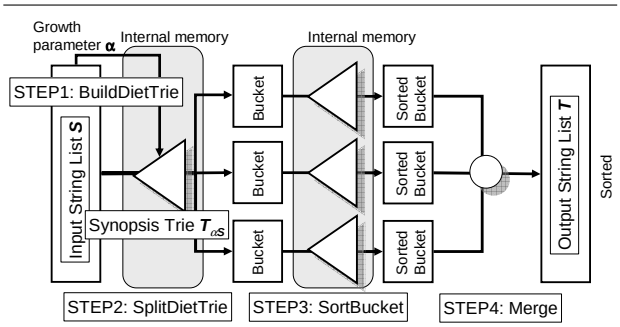

Figure 1: The architecture of an external sorting algorithm DistStrSort using adaptive construction of a synopsis trie

Based on this technique, we present an external string sorting algorithm DistStrSort. In Fig. 1, we show an architecture of our external sorting. This algorithm first adaptively construct a small synopsis trie as a model of string distribution by a single scan of the input data, splits input strings into a collection of buckets using the synopsis trie as a splitter, then sort the strings in each bucket in main memory, and finally merge the sorted buckets into an output collection.

By an analysis, we show that if both of the size of the synopsis trie and the maximum size of the buckets are small enough to fit into main memory of size M, then our algorithm DistStrSort correctly solves the external string sorting problem in $O(N)$ time within main memory M using $3N$ sequential read, N random write, and N sequential write in the external memory, where N is the total size of an input string list S. Thus, this algorithm of $O(N)$ time complexity is attractive compared with a popular merge sort-based external string sort algorithm of $O(N \log N)$ time complexity. Experiments on real datasets show the effectiveness of our algorithm.

As an advantage, this algorithm is suitable to implement complex data manipulation operations (Ramakrishnan and Gehrke 2000) such as

group-by aggregation, statistics calculation, and functional join for texts.

1.1 Related works

In what follows, we denote by $K = |S|$ and $N = ||S||$ be the cardinality and the total size of an input string list $S \in (\Sigma^*)^*$.

A straightforward extension of quick sort algorithm solves the string sorting problem in $O(LK \log K)$ time in main memory, where L is the maximum length of the input strings and $N = O(LK)$. This algorithm is efficient if $L = O(1)$, but inefficient if L is large.

Arge *et al.* presented an algorithm that solves the string sorting problem in $O(K \log K + N)$ time in main memory (Arge *et al.* 1997). The String B-tree is an efficient dynamic indexing data structure for strings (Ferragina and Grossi 1999). This data structure also has almost optimal performance for external string sorting in theory, while the performance degenerates in practice because of many random access to external memory.

Sinha and Zobel presented a trie-based string sorting algorithm, called the *burst sort*, and show that the algorithm outperforms many of previous string sorting algorithms on real datasets (Sinha and Zobel 2003). However, their algorithm is an internal sorting algorithm, and furthermore, the main aim of this paper is proposal of an adaptive strategy for realizing an efficient external string sorting.

Although there are a number of studies on sorting the suffixes of a single input string (Bentley and Sedgewick 1997, Manber and Myers 1993, Manzini and Ferragina 2002, Sadakane 1999), they are rather irrelevant to this work since they are mainly incore algorithms and utilize the repetition structure of suffixes of a string.

The adaptive construction of a trie for a stream of letters has been studied since 90's in time-series modeling (Laird, Saul 1994), data compression (Moffat 1990), and bioinformatics (Ron, Singer, Tishby 1996). However, the application of adaptive trie construction to external sorting seems new and an interesting direction in text and semi-structured data processing. Recently, (Stonebraker *et al.* 2005) showed the usefulness of distributed stream processing as a massive data application of new type.

1.2 Organization

This paper is organized as follows. In Section 2, we prepare basic notions and definitions on string sorting and related string processing problems. In Section 3, we present our external string sorting algorithm based on adaptive construction of a synopsis trie. In Section 4, we report experiments on real datasets, and finally, we conclude in Section 5.

2 Preliminaries

In this section, we give basic notions and definitions on external string sorting.

2.1 Strings

Let $\Sigma = \{A, B, A_1, A_2, \ldots\}$ be an *alphabet* of letters, with which a *total order* $\leq_\Sigma$ on Σ is associated. In particular, we assume that our alphabet is a set of integers in a small range, i.e., $\Sigma = \{0, \ldots, C-1\}$ for some integer $C \geq 0$, which is a constant or at most $O(N)$ for the total input size N. This is the case for ASCII alphabet $\{0, \ldots, 255\}$ or DNA alphabet $\{A, T, C, G\}$.

We denote by Σ^* the set of all finite strings on Σ and by ε the empty string. Let $s = a_1 \cdots a_n \in \Sigma^*$ be a string on Σ. For $1 \leq i \leq j \leq n$, we denote by $|s| = n$ the *length* of s, by $s[i] = a_i$ the *i-th letter*, and by $s[i..j] = a_i \cdots a_j$ the *substring* from i-th to j-th letters. If $s = pq$ for some $p, q \in \Sigma^*$ then we say that p is a *prefix* of s and denote $p \sqsubseteq s$.

We define the lexicographic order $\leq_{\text{lex}}$ on strings of Σ^* by extending the total order $\leq_\Sigma$ on letters in the standard way.

2.2 String sorting problem

For strings $s_1, \ldots, s_m \in \Sigma^*$, we distinguish an ordered list $(s_1, \ldots, s_m)$, an unordered list (or multiset) $\{\!\{s_1, \ldots, s_m\}\!\}$, and a set $\{s_1, \ldots, s_m\}$ (of unique strings) each other. The notations $\in$ and $=$ are defined accordingly. We use the intentional notation $(\ s \mid P(s)\ )$ or $\{\ s \,|\, P(s)\ \}$. For lists of strings S_1, S_2, we denote by $S_1 S_2$ ($S_1 \oplus S_2$, resp.) the concatenation of S_1, S_2 as *ordered lists* (as *unordered lists*, resp.).

An input to our algorithm is a *string list* of size $K \geq 0$ that is just an ordered list $S = (s_1, \ldots, s_K) \in (\Sigma^*)^*$ of possibly identical strings, where $s_i \in \Sigma^*$ is a strings on Σ for every $1 \leq i \leq K$. We denote by $|S| = K$ the *cardinality* of S and $||S|| = N = \sum_{i=1}^{K} |s_i|$ the *total size* of S. We denote the set of all unique strings in S by $uniq(S) = \{\ s_i \,|\, i = 1, \ldots, K\ \}$. A string list $S = (s_1, \ldots, s_n)$ is *sorted* if $s_i \leq_{\text{lex}} s_{i+1}$ holds for every $i = 1, \ldots, n-1$.

Definition 1 The *string sorting problem* (STRSORT) is, given a string list $S = (s_1, \ldots, s_K) \in (\Sigma^*)^*$, to compute the sorted list $\pi(S) = (s_{\pi(1)}, \ldots, s_{\pi(K)}) \in (\Sigma^*)^*$ for some permutation $\pi : \{1, \ldots, K\} \to \{1, \ldots, K\}$ of its indices.

We can extend our framework for more complicated string processing problems as follows.

Definition 2 The *string counting problem* (STRCOUNT) is, given a string list $S = (s_1, \ldots, s_K) \in (\Sigma^*)^*$, to compute the histogram $h : uniq(S) \to \mathsf{N}$ such that $h(s)$ is the number of occurrences of the unique string s.

Let $(\Delta, \circ)$ be a pair of a collection Δ of *values* and a corresponding associative operation $\circ : \Delta^2 \to \Delta$. For example, $(\mathsf{N}, +)$ and $(\mathsf{R}, \max)$ are examples of $(\Delta, \circ)$. A *string database* of size $K \geq 0$ over $dom(D)$ is a list $D = (\ (s_i, v_i)\,|\, i = 1, \ldots, K\ ) \in (\Sigma^* \times \Delta)^*$ of pairs of a string and a value. We denote by $uniq(D) = \{\, s_i \,|\, (s_i, v_i) \in D \,\}$ the set of unique strings in S. For $(\Delta, \circ)$, the *aggregate* on s w.r.t. $\circ$, denoted

Algorithm DistStrSort:

Input: A string list $S = (s_1, \ldots, s_K) \in (\Sigma^*)^*$ and a maximal bucket size $0 \leq B \leq M$.

Output: The sorted string list $T \in (\Sigma^*)^*$.

1. Determine a growth parameter $\alpha \geq 0$. Build an adaptive synopsis trie $\mathcal{T}_{\alpha,S}$ for S in main memory with the growth parameter α and a bucket size B. (BuildDietTrie in Fig. 4)
2. Split the input list S in external memory into partition of buckets $S_1, \ldots, S_d$ of size at most B ($d \geq 0$) by using $\mathcal{T}_{\alpha,S}$, where $S_i \preceq S_{i+1}$ for every $i = 1, \ldots, n-1$. (BuildDietTrie in Fig. 5)
3. For every $i = 1, \ldots, K$, sort the bucket B_i by arbitrary internal sorting algorithm and write back to B_i in external memory. (BuildDietTrie in Fig. 6)
4. Return the concatenation $T = B_1 \cdots B_K$ of sorted buckets.

Figure 2: An outline of the external string sorting algorithm with adaptive splitting DistStrSort

by $\mathtt{aggr}_\circ(s)$, is defined as $\mathtt{aggr}_\circ(s) = \circ\,(\,v \,|\, (s,v) \in D\,) = v_1 \circ \cdots \circ v_m$, where $(s_1, v_1), \ldots, (s_m, v_m) \in D$ is the list of all pairs in D with $s_i = s$.

Definition 3 The *string aggregation problem* w.r.t. operation $(\Delta, \circ)$ (STR-AGGREGATE$(\Delta, \circ)$) is, given a string database $S = (\,(s_i, v_i) \,|\, i = 1, \ldots, K\,) \in (\Sigma^* \times \Delta)^*$, to compute the pair $(s, \mathtt{aggr}_\circ(s))$ for every unique string $s \in \mathit{uniq}(D)$.

The above string counting and string aggregation problems as well as string sorting problem can be efficiently solved by a modification of trie-based sorter. In this paper, we extend such a trie-based sorter for an external memory environment.

2.3 Model of computation

In what follows, we denote by $K = |S|$ and $N = ||S||$ the cardinality and the total size of the input string list S. We assume a naive model of computation such that a CPU has random access to a main memory of size $M \geq 0$ and sequential and random access to an external memory of unbounded size, where memory space is measured in the number of integers to be stored. Though the external memory can be partitioned into blocks of size $0 \leq B \leq M$, we do not study a detailed analysis of I/O complexity. In this paper, we are particularly interested in the case that the input size does not fit into the main memory but not so large, namely, $N = O(M^2)$.

3 Our Algorithm

3.1 Outline of the main algorithm

In Fig. 2, we show the outline of our external string sorting algorithm DistStrSort using an adaptive construction of synopsis trie for splitting, which is a variant of distribution sort for integer lists. In Fig. 1 shows the architecture of DistStrSort.

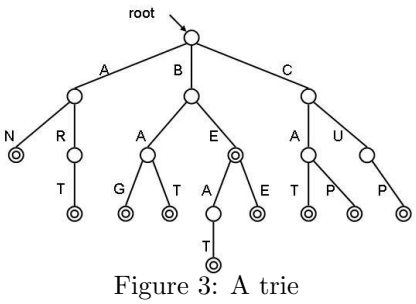

Figure 3: A trie

The key of the algorithm is Step 2 that splits the input list S into buckets $S_1, \ldots, S_d$ of at most B by computing an ordered partition defined as follows. For string lists $S_1, S_2 \in (\Sigma^*)^*$, S_1 *precedes* S_2, denoted by $S_1 \prec S_2$, if $s_1 <_{\mathrm{lex}} s_2$ for every combination of s_1 in S_1 and s_2 in S_2.

Then, an *ordered partition* of a string list S is a sequence of string lists $S_1, \ldots, S_d$ such that

(i) the sequence is a partition of S as unordered list, i.e., $S = S_1 \oplus \cdots \oplus S_d$,

(ii) S_i precedes S_{i+1}, i.e., $S_i \preceq S_{i+1}$, for every $i = 1, \ldots, n-1$, and

Furthermore, an ordered partition of a string list S satisfies *maximum bucket size* B if

(iii) $|S_i| \leq B$ holds for every $i = 1, \ldots, n$.

It is easy to see that if we can compute an ordered partition of an input string list S of maximum size $B \leq M$ in main memory of size M and if we can sort bucket of size B in $O(B)$ main memory then the algorithm DistStrSort in Fig. 2 correctly solves the string sorting problem in main memory M (Theorem 7).

However, it is not easy to compute a good partition with maximum block size B using only a limited main memory for a large input data S that does not fit into main memory. Hence, we take an approach of computing an approximate answer using adaptive computation in the following sections.

3.2 A Synopsis Trie

A *synopsis trie* for a string list $S = (s_1, \ldots, s_K) \in (\Sigma^*)^*$ is the trie data structure $\mathcal{T}$ for a subset of prefixes of strings in $\mathit{uniq}(S)$ defined as follows.

A *trie* for a set S of strings on Σ (Fredkin 1960) is a rooted tree $\mathcal{T}$ whose edges are labeled with a letter in Σ. In Fig. 3, we show an example of a trie for a set $S = \{\mathtt{AN}, \mathtt{ART}, \mathtt{BAG}, \mathtt{BAT}, \mathtt{BEAT}, \mathtt{BEE}, \mathtt{CAT}, \mathtt{CAP}, \mathtt{CUP}\}$ of strings.

We denote by $\mathcal{T}$ the set of vertices of $\mathcal{T}$ and by $L(\mathcal{T}) \subseteq \mathcal{T}$ the set of its leaves. Each vertex $v \in \mathcal{T}$ represents the string $\mathit{str}(\ell) \in \Sigma^*$, called the *path string*, obtained by concatenating the labels on the unique path from the root `root` to v in $\mathcal{T}$. Let $\mathit{Str}(\mathcal{T}) = \{\, \mathit{str}(\ell) \,|\, \ell \in L(\mathcal{T}) \,\}$ be the set of strings represented by the leaves of $\mathcal{T}$. The vertices of a trie $\mathcal{T}$ are often labeled with some additional information.

Algorithm: BUILDDIETTRIE

Input: A string list $S = (s_1, \ldots, s_K)$, a maximum bucket size $M \geq B \geq 0$, and a growth parameter $\alpha > 0$.

Output: A synopsis trie $\mathcal{T}_{\alpha,S}$ for S.

1. $\mathcal{T} := \{\texttt{root}\}$;
2. For each $i := 1, \ldots, K$, do the following:
   (a) Read the next string $s = s_i$ from the external disk;
   (b) For each $j := 1, \ldots, |s|$, do:
   - $y := \texttt{goto}(x, s[j])$; (Trace the edge c_j.)
   - If $y = \perp$ holds, then:
     - If $\texttt{count}(x) \geq \alpha$ holds, then: Create a new state y; $T := T \cup \{y\}$; $\texttt{count}(y) := 1$; $\texttt{goto}(x, c_j) := y$;
     - Else: **break** the inner for-loop and **goto** the step 2(a);
   - Else, $x := y$ and $\texttt{count}(y) := \texttt{count}(y) + 1$;
3. Return $\mathcal{T}$;

Figure 4: An algorithm for constructing a synopsis trie

If a trie $\mathcal{T}$ satisfies that $Str(\mathcal{T}) = S$ then $\mathcal{T}$ is said to be a *trie for* S. The trie $\mathcal{T}$ for a string list S can be constructed in $O(N)$ time and $O(N)$ space in the total input size $N = ||S||$ for constant Σ (Fredkin 1960).

Then, a *synopsis trie* for S is a trie $\mathcal{T}$ for S which satisfies that for every string $s \in uniq(S)$, some leaf $\ell \in \mathcal{T}$ represents a prefix of s, i.e., $str(\ell) \sqsubseteq s$. Note that a synopsis trie for S is not unique and the empty trie $\mathcal{T} = \{\texttt{root}\}$ is a trivial one for any set S.

Given a string list S, a synopsis trie $\mathcal{T}$ can store strings in S. For every leaf $\ell \in L(\mathcal{T})$, we define the sublist of strings in S belongs to ℓ by $\texttt{list}(\ell, S) = (\, s \in S \mid str(\ell) \sqsubseteq s \,)$ and the count of the vertex by $\texttt{count}(\ell, S) = |\texttt{list}(\ell, S)|$. The total space required to implement a trie for S is $O(||S||)$ even if we include the information on `count` and `list`. On the incore computation by a trie, we have the following lemma.

Lemma 1 *We can solve the string sorting, the string counting, and the string aggregation problems for an input string list S in $O(N)$ time using main memory of size $O(N)$, where $N = ||S||$.*

We will extensively use this trie data structure in the following subsections in order to implement the computation at Step 1, Step 2, and Step 3 of the main sorting algorithm in Fig. 2.

3.3 STEP1: Adaptive Construction of a Synopsis Trie

In Fig. 4, we show the algorithm BUILDDIETTRIE for constructing an adaptive synopsis trie.

Although the empty trie $\mathcal{T} = \{\texttt{root}\}$ obviously satisfies the above condition, it is useless for computing a good ordered partition. Instead, the goal is adaptive construction of a synopsis trie $\mathcal{T}$ for S satisfying the following conditions:

(i) $\mathcal{T}$ fits into the main memory of size M, i.e., $\texttt{size}(\mathcal{T}) \leq M$.

(ii) For an input string list S, $\texttt{count}(\ell, S) \leq B$ for a parameter $B \leq M$

The adaptive construction is done as follows. For an alphabet $\Sigma = \{c_0, \ldots, c_{s-1}\}$, each state v of the automaton is implemented as the list $(\texttt{goto}(v, 0), \ldots, \texttt{goto}(v, s-1))$ of s pointers. For an alphabet of constant size, this is implemented as an array of pointers. At the creation of a new state v, these pointers are set to $NULL$ and a counter $\texttt{count}(v)$ is set to 1.

When we construct a synopsis trie $\mathcal{T}$, the algorithm uses a positive integer $\alpha > 0$, called the *threshold for growing*. This threshold represents a minimum frequency for extending a new state to the current edge.

The algorithm firstly initializes a trie $\mathcal{T}$ as the trie $\mathcal{T} = \{\texttt{root}\}$ consisting of the root node only. When the algorithm inserts a string $s \in S$ into $\mathcal{T}$, it traces the corresponding edge to w starting from `root` by `goto` pointers. Each state v in the synopsis trie $\mathcal{T}$ has an integer $\texttt{count}(v)$ incremented when a string w reaches to v. These counters represent approximate occurrences of suffixes of input strings.

When there are no states the current state v transfers to, the synopsis trie tries to generate a new state. However, the algorithm does not permit the trie to extend a new edge unless the counter $\texttt{count}(v)$ of v is more than the given threshold α. If $\texttt{count}(v)$ exceeds α, the trie generates a new state, and attach it to the current state with a new edge.

Lemma 2 *Let S be an input string list of total size N. The algorithm* BUILDDIETTRIE *in Fig. 4 computes a synopsis trie for S in $O(N)$ time and $O(|\mathcal{T}|)$ space.*

At present, we have no theoretical upper bounds of $|\mathcal{T}|$ and the maximum value of $\texttt{count}(\ell)$ according to the value of growth parameter $\alpha \geq 0$. Tentatively, setting $\alpha = cB$ for some constant $0 \leq c \leq 1$ works well in practice.

3.4 STEP2: Splitting a String List into Buckets

Let $BID = \{1, \ldots, b\}$ be a set of *bucket-id*'s for some $b \geq 0$. Then, a *bucket-id assignment* for $\mathcal{T}$ is a mapping $\beta : L(\mathcal{T}) \to BID$ that assigns bucket id's to the leaves. Let $(\mathcal{T}, BID)$ be a synopsis trie $\mathcal{T}$ whose leaves are labeled with the bucket-id assignment β.

A bucket-id assignment β for $\mathcal{T}$ is said to be *order-preserving* if it satisfies the following condition: for every leaves ℓ_1, ℓ_2, if $str(\ell_1) \leq_{\text{lex}} str(\ell_2)$ then $\beta(\ell_1) \leq \beta(\ell_2)$.

Algorithm SplitDietTrie:

Input: A string list $S = (s_1, \ldots, s_K)$ (in external memory), a synopsis trie $\mathcal{T}_{\alpha,S}$ for S (in internal memory), and a positive integer $M \geq B \geq 0$.

Output: An ordered partition $B_1 \oplus \cdots \oplus B_b$ for S.

1. //Compute a bucket-id assignment β
   - Let $N' = \sum_{\ell \in L(\mathcal{T}_{\alpha,S})} \mathtt{count}(\ell)$; $B' = BN'/N$; $k = 1$; $A = 0$;
   - For each leaf $\ell \in L(\mathcal{T}_{\alpha,S})$ in the lexicographic order of $str(\ell)$, do:
     − If $(A + \mathtt{count}(\ell) \leq B')$, then $\mathtt{bucket}(\ell) := k$ and $A := A + \mathtt{count}(\ell)$;
     − Else, $k := k + 1$ and $A := 0$;
   - $b = k$;
2. //Distribute all strings in S into the corresponding buckets.
   - For each $k = 1, \ldots, b$, do: $B_i := \emptyset$;
   - For each string $s \in S$, do:
     − Find the leaf $\ell = \mathtt{vertex}(s, \mathcal{T}_{\alpha,S})$ reachable from the root by s;
     − Put s to the k-th bucket B_k in external memory for $k = \beta(\ell)$.

Figure 5: An algorithm SplitDietTrie for splitting an input string list

By the definition of a synopsis trie above, we know that for every string $s \in uniq(S)$, there is the unique leaf $\ell \in L(\mathcal{T})$ such that $str(\ell) \sqsubseteq s$. We denote this leaf by $\mathtt{vertex}(s, \mathcal{T}) = \ell$. Then, the bucket-id assigned to s is defined by $\mathtt{bucket}(\mathcal{T}, \beta, s) = \beta(\mathtt{vertex}(s, \mathcal{T}))$. Given an input string list $S \in (\Sigma^*)^*$, $(\mathcal{T}, \beta)$ defines the partition

$$PART(S, \mathcal{T}, \beta) = S_1 \oplus \ldots \oplus S_b$$

where $S_k = (\ s \in S \mid bid(\mathcal{T}, \beta, s) = k\ )$ holds for every bid $k \in BID$. Then, the *maximum bucket size* of $(\mathcal{T}, BID)$ on input S is defined by $\max\{\ S_k \mid k = 1, \ldots, b\ \}$. Clearly, $S_1 \oplus \ldots \oplus S_b = S$ holds.

Lemma 3 *If β is order-preserving then $PART(S, \mathcal{T}, \beta)$ is an ordered partition.*

In Fig. 5, we show an algorithm SplitDietTrie for splitting an input string list S into buckets $PART(S, \mathcal{T}, \beta) = B_1 \oplus \ldots \oplus B_b$. We can easily see that the bucket-id assignment computed by algorithm SplitDietTrie is always order-preserving. Furthermore, we have the following lemma.

Lemma 4 *Let S be an input list and $\mathcal{T}$ be a synopsis trie with bucket-id assignment for S. Suppose that $|\mathcal{T}| \leq M$. Then, the algorithm* SplitDietTrie *of in Fig. 5, given S and $\mathcal{T}$, computes an ordered partition of S in $O(N)$ time and $O(|\mathcal{T}|)$ space in main memory, where $N = ||S||$.*

We have the following lemma on the accuracy of the approximation.

Algorithm SortBucket:

Input: A bucket $B \in (\Sigma^*)^*$.

Output: A bucket $C_i \in (\Sigma^*)^*$ obtained from B by sorted in $\leq_{\mathrm{lex}}$.

1. Build a trie $\mathcal{T}_B$ for the bucket B in main memory;
2. Traverse all vertices v of $\mathcal{T}_B$ in the lexicographic order of path strings $str(v)$ and output $str(v)$.

Figure 6: An algorithm SortBucket for sorting each bucket

Lemma 5 *Let $B' \geq 0$ $(0 \leq B' \leq M)$ and $k \geq 1$ be any integers. Let $\mathcal{T}$ be the synopsis trie for S. If the trie $\mathcal{T}$ on S satisfies $0 \leq \mathtt{count}(v) \leq \frac{1}{k}B'$ for any vertex $\ell \in L(\mathcal{T})$, then the resulting ordered partition $S = B_1 \oplus \cdots \oplus B_b$ $(b \geq 0)$ satisfies that $\frac{k-1}{k}B \leq B_i \leq B$ holds for any bucket B_i $(1 \leq i \leq m)$.*

3.5 STEP3: Internal Sorting of Buckets

Once an input string list S of total size N has been split into an ordered partition $S_1 \oplus \ldots \oplus S_b = S$ of maximum bucket size $B \leq M$, we can sort each bucket B_i in main memory by using any internal memory sorting algorithm. For this purpose, we use a *internal trie sort* which is described in Fig. 6.

Lemma 6 *The algorithm* SortBucket *of Fig. 6 sorts a bucket B of strings in $O(N)$ time using main memory of size $O(N)$, where $N = ||B||$ and Σ is a constant alphabet.*

3.6 STEP4: Final concatenation

Step 4 is just a concatenation of already sorted buckets $B_1, \ldots, B_b$. Therefore, this step requires no extra cost.

3.7 Analysis of the algorithm

In this subsection, we give a case analysis of a practical performance of the proposed external sorting algorithm assuming a practical situations for such an algorithm. For the purpose, we suppose the following condition:

Condition 1 *the synopsis trie computed by* BuildDietTrie *and* SplitDietTrie *on the input string list S and the choice of a growth parameter α satisfies the following conditions:*

- *The synopsis trie fits into main memory, that is, $\mathtt{size}(\mathcal{T}) \leq M$.*
- *The maximum bucket size of the ordered partition computed by* SplitDietTrie *at Step 2 does not exceed M.*

For the above condition to hold, we know that at least the input S satisfies that $N = O(M^2)$. Note that at this time, we only have a heuristic algorithm

Table 1: Parameters of the datasets

Dataset	*data A*	*data B*
File size	67MB	1.1GB
Number of records (with duplications)	140×10^4	2290×10^4
Number of records (no duplications)	22×10^4	109×10^4
Average of record length	45.0	44.9
Variance of record length	45.1	19.7
Maximum record length	58	58
Minimum record length	15	15

for computing a good synopsis trie without any theoretical guarantee for its performance. Finally, we have the following result.

Theorem 7 *Suppose that the algorithms* BuildDietTrie *and* SplitDietTrie *satisfy Condition 1 on the input S and $\alpha \geq 0$. The algorithm* DistStrSort *correctly solves the external string sorting problem in $O(N)$ time within main memory M using $3N$ sequential read (Step 1–Step 3), N random write (Step 2), and N sequential write (Step 3) in the external memory.*

From Theorem 7 and Lemma 1, we can show the following corollary using modified version of DistStrSort algorithm.

Corollary 8 *Let $(\Delta, \circ)$ be any associative binary operator, where operation $\circ$ can be computed in $O(1)$ time. Suppose that the algorithms* BuildDietTrie *and* SplitDietTrie *satisfy Condition 1 on the input S and $\alpha \geq 0$. There exists an algorithm that solves the string counting (STR-COUNT), and the string aggregation problems (STR-AGGREGATE$(\Delta, \circ)$) for an input string list S in $O(N)$ time using main memory of size M in external memory.*

4 Experimental Results

In this section, we present experimental results on real datasets to evaluate the performance of our algorithms. We implemented our algorithms in C. The experiments were run on a PC (Xeon 3.6GHz, Windows XP) with 3.25 gigabytes of main memory.

We prepared two datasets *data A* and *data B* consisting of cookie values from Web access logs. The parameters of the datasets are shown in Table 1.

4.1 Size of Synopsis Trie

First, we studied sizes of synopsis tries constructed by the algorithm BuildDietTrie by varying the growth parameter $\alpha = 100, 1000$, or 10000. Table 2 shows the number of states of the constructed synopsis trie for each α on the *data A*. We can see that the bigger α is, the smaller the size of the constructed synopsis trie becomes.

Table 2: Sizes of the constructed synopsis tries

Growth parameter α	100	1,000	10,000
Number of states	426,209	92,768	5,030

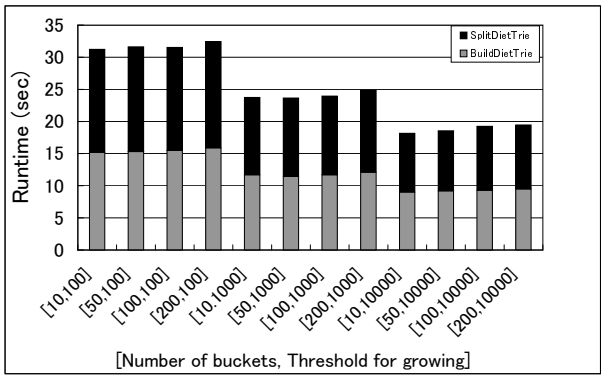

Figure 7: Running time of BuildDietTrie and SplitDietTrie

4.2 Running Time

4.2.1 BuildDietTrie and SplitDietTrie

Next, we measured the running times of the subprocedures BuildDietTrie and SplitDietTrie on the *data A*, by varying the number of buckets and the growth parameter. Fig. 7 show the results. The horizontal axis in the figure represents pairs of the number of buckets and the growth parameter. The vertical axis represents the running time.

The results indicate that the running time is shorter for larger growth parameters. Thus, a smaller synopsis trie can be constructed in a reasonable computational time.

4.2.2 DistStrSort

Finally, we studied the running time of the algorithm DistStrSort. Fig. 8 shows the running times on the *data B* of DistStrSort and the GNU sort 5.97. The running time of our algorithm scales linearly with the size of data and is faster than the GNU sort for larger data sizes.

5 Conclusion

In this paper, we presented a new algorithm for sorting large collections of strings in external memory. The main idea of our method is first splitting a set of strings to some buckets by adaptive construction of a synopsis trie for input strings, then sort the strings in each bucket, and finally merge the sorted buckets into an output collection. Experiments on real datasets show the effectiveness of our algorithm.

In this paper, we only made empirical studies on the performance of adaptive strategy for construction of a synopsis trie for data splitting. The probabilistic analysis of the upper bounds of $|\mathcal{T}|$ and the maximum

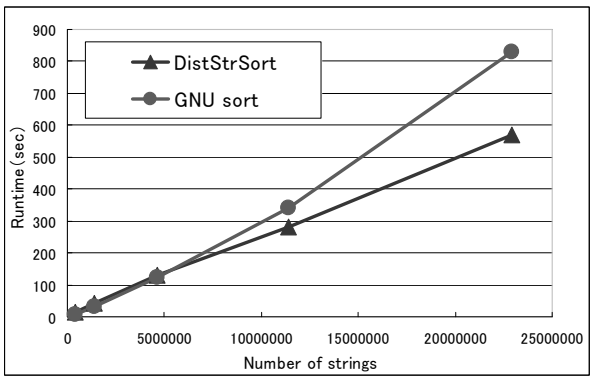

Figure 8: Running times of DistStrSort and GNU sort

value of `count`(ℓ) concerning to the value of growth parameter $\alpha \geq 0$ is an interesting future research.

We considered only the case that the input size is at most the square of the memory size, and then presented a two-level external sorting algorithm. It is a future research to develop a hierarchical version of our algorithm for inputs of unbounded size. A stream-based distributed version of distribution string sort algorithms is another possible direction.

References

S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey, S. Ceri, W. B. Croft, D. J. DeWitt, *et al.*, The Lowell database research self-assessment, C. ACM, 48(5), 111–118, 2005.

S. Abiteboul, P. Buneman, D. Suciu, *Data on the Web*, Morgan Kaufmann, 2000.

L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, On Sorting Strings in External Memory, Proc. the 29th Annual ACM Symposium on Theory of Computing (STOC'97), 540–548, 1997.

J. Bentley, R. Sedgewick, Fast Algorithms for Sorting and Searching Strings, Proc. the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97), 360–369, 1997.

P. Ferragina, R. Grossi, The String B-tree: A New Data Structure for String Search in External Memory and Its Applications, J. ACM, 46(2), 236–280, 1999.

E. Fredkin, Trie Memory, C. ACM, 3(9), 490–499, 1960.

P. Laird and R. Saul, Discrete sequence prediction and its applications, *Machine Learning*, 15(1): 43–68, 1994.

U. Manber, E. W. Myers, Suffix Arrays: A New Method for On-Line String Searches, *SIAM J. Comput.*, 22(5), 935–948, 1993.

G. Manzini and P. Ferragina, Engineering a Lightweight Suffix Array Construction Algorithm, Proc. the 10th European Symposium on Algorithms (ESA'02), 698-710, 2002.

A. Moffat, Implementing the PPM data compression scheme, *IEEE Trans Communications* COM-38(11): 1917–1921, 1990.

R. Ramakrishnan, J. Gehrke, *Database Management Systems*, McGraw-Hill Professional, 2000.

D. Ron, Y. Singer, and N. Tishby, The power of amnesia: learning probabilistic automata with variable memory length, *Machine Learning*, 25(2-3): 117– 149, 1996.

K. Sadakane A Fast Algorithms for Making Suffix Arrays and for Burrows-Wheeler Transformation, Proc. the 8th Data Compression Conference (DCC'98), 129–138, 1999.

R. Sinha, J. Zobel, Efficient Trie-Based Sorting of Large Sets of Strings, Proc. the 26th Australasian Computer Science Conference (ACSC'03), 2003.

M. Stonebraker, U. Cetintemel, "One Size Fits All": An Idea Whose Time Has Come and Gone, Proc. the IEEE 21st International Conference on Data Engineering (ICDE'05), keynote, 2–11, 2005.