WWW 検索エンジンのためのインクリメンタルな全文検索インデックス更新方式

吉原 潤[†] 加藤和彦[††]

suffix array はテキストの接尾辞のポインタを接尾辞の辞書順に並べたもので，任意の部分文字列検索を高速に行うことができる．しかし更新のオーバーヘッドが大きく，日本語の全文検索を行う WWW 検索エンジンに suffix array を用いるには，頻繁に起こる更新によるオーバーヘッドが問題となる．本論文では suffix array の効率的な更新方式を提案する．この方式では，既存の巨大な suffix array はすぐには更新せず，差分のテキストを元に差分 suffix array を作る．検索は複数の suffix array 全てに対して行い，それらの結果をマージする．もとの suffix array と差分 suffix array をマージすることにより検索処理の際のマージオーバヘッドを軽減する．suffix array を使った WWW 検索エンジンを実装して実験を行い，提案方式の有効性を検証した．

Incremental Updating Scheme of a Full-Text Index Structure for WWW Text Search Engines

JUN YOSHIWARA[†] and KAZUHIKO KATO[††]

A suffix array is a full-text index data structure, and is an array of pointers to lexicographic-ordered suffixes of text data. A suffix array is efficient for retrieving any substring of text, but requires a lot of overhead for updating it. In general, constant update operations are required in Web search engines, so the development of an efficient updating technique for suffix arrays is an important research issue. In this paper, we propose an efficient updating scheme of suffix arrays. The scheme is based on the following two ideas. First, instead of maintaining a single large suffix array, several small differential suffix arrays are created when new text data are obtained. At the processing of a retrieval operation, all the suffix arrays are retrieved and the retrieved results are merged into one. Second, when the number or the amount of differential suffix arrays exceeds the predetermined threshold, the differential suffix arrays are merged into one. We conducted experiments using an implementation and real WWW data and we verified the effectiveness of the proposed scheme.

1. はじめに

大規模テキストに対する文字列の検索は，あらかじめ単語として切り出された単語を検索の対象とする方法と，任意の部分文字列を対象として検索する方法とに大別される．前者の処理を高速に行うための方法として，転置ファイルやシグネチャファイル等のインデックス構造を用いた方法がよく知られている．この方法は，単語の切り出しが正しく行われない場合，検索洩れが生じるという短所をもつ．特に日本語テキストの場合，単語の切り出しを行うには形態素解析が必要で，日本語辞書を利用した処理系が作成されているが，人間の介在無しには，100% 正確な形態素解析を行うことは困難である．

後者の，任意の部分文字列を検索する方法としては，suffixarray[6], suffix tree[7], SB-tree[1] などのインデックス構造を用いた方法が知られている．suffix array は，1990 年に Manber と Myers によって提案されたインデックス構造で，テキストの接尾辞を辞書順にソートして，その接尾辞へのポインタを並べた配列である．suffix array は他の 2 つよりもコンパクトな構造であり，また，二分探索によって高速に任意の部分文字列の検索ができるため，テキストが巨大になる場合にも有効である．

ここ数年，インターネット環境の急速な普及に伴い WWW が一般に広く利用されるようになり，WWW で提供される膨大な量の情報の中から求める有用な情報

† 筑波大学大学院修士課程理工学研究科
Master's Program in Science and Engineering at University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics at University of Tsukuba

を見つけ出す WWW 検索エンジンが情報収集のための重要な手段となっている[9]．WWW 検索エンジンではインデックスを作る対象となるテキストが頻繁に更新されるので，それに合わせてインデックスも更新し，常にできるだけ新しい情報を検索結果として提供する事が要求される．suffix array は，検索漏れがない全文テキスト検索技術であり，高速な検索が可能で，コンパクトなインデックス構造を持つが，静的な構造であるため，テキストが頻繁に更新される場合，それに伴う suffix array の更新にかかるオーバーヘッドが非常に大きい．このため，WWW 検索エンジンに代表される，更新の頻繁に起こる大規模なテキストに対する検索のインデックス構造として suffix array を用いるには，更新の際のオーバーヘッドが大きな問題となる．

本論文では，WWW 検索エンジンで利用することを前提として，suffix array の繰り返し的な更新を効率的に行うためのアルゴリズムを提案する．まず，一つの suffix array を直接更新する方式を提案する．次に，毎回の更新で差分のテキストから差分 suffix array を作りインクリメンタルに更新を行ない，更新時間を短縮した方式を提案する．この方式は，検索処理時に結果をマージするオーバヘッドを要する．そこで，これら二つの方式を統合し両方の利点を組み合わせると共に短所を解消した方式を提案する．

本論文は以下のように構成されている．2 章では suffix array や WWW 検索エンジンの更新などの基本概念について述べる．3 章では一つのインデックスを直接，更新する方式について述べる．4 章では，複数の suffix array を用いてインクリメンタルに更新を行なう方式について述べる．5 章では，前述の 2 つの方式を統合した方式について述べる．6 章では，提案方式を用いた WWW 検索エンジンを実装し，実験を行ない各方式の性能を比較する．最後に 7 章でまとめる．

2. 基本概念

本章では，以下に続く章を理解するために必要な基本概念として，suffix array と，それを用いた WWW 検索エンジンにおける更新処理について説明する．

2.1 suffix array

テキスト $T[1,n]$ に対して，$T[i,n], T[1,j]$ をそれぞれ T の接尾辞，接頭辞という．入力テキスト $T[1,n]$ の suffix array $A[1,n]$ とは，T の接尾辞 $s_i = T[i,n]$ へのポインタ $i\ (i = 1,\ldots,n)$ を，それが指す接尾辞の辞書順に並べた配列である．よって $T[A[i],n]$ は辞書順で i 番目になる接尾辞となる．A を用いて T を参照し配列 $T[A[1],n],\ldots,T[A[n],n]$ を考えると，これは辞書順に並んだ T の接尾辞の配列になっている．以後，この仮想の配列を $S[1,n]$ と表す．suffix array の例を図 1 に示す．

テキスト
$T =$ "abcbccab" $\Longrightarrow$ suffix array
$A = (7,1,8,2,4,6,3,5)$

T の接尾辞	ソートされた T の接尾辞
$s_1 =$ "abcbccab"	$s_7 =$ "ab"
$s_2 =$ "bcbccab"	$s_1 =$ "abcbccab"
$s_3 =$ "cbccab"	$s_8 =$ "b"
$s_4 =$ "bccab"	$s_2 =$ "bcbccab"
$s_5 =$ "ccab"	$s_4 =$ "bccab"
$s_6 =$ "cab"	$s_6 =$ "cab"
$s_7 =$ "ab"	$s_3 =$ "cbccab"
$s_8 =$ "b"	$s_5 =$ "ccab"

図 1 suffix array の例
Fig. 1 The example of a suffix array

あるパターン P がテキスト T に現れる場合，P は T のある接尾辞の接頭辞となり，この接尾辞の出現位置が P の出現位置となる．配列 $S[1,n]$ の接尾辞は辞書順に並んでいるので，P が T に現れるならば P を接頭時とする接尾辞は $S[i,j]$ に連続して現れる．このとき $A[i,j]$ の値がパターン P の全ての出現位置である．i,j は配列 S を二分探索して見つけることができる．

suffix array の構成アルゴリズムについては様々な研究がされている[6)5)8)2)]．suffix array の構成はこれまでに提案されている方法を用いて行なうこととし，本論文では言及しない．

2.2 複数のテキストに対する suffix array

suffix array は基本的には単一のテキストに対して用いられるデータ構造なので，複数のテキストに対する suffix array を作成する場合，まず複数のテキストを連結して一つのテキストを作り，それに対して suffix array を作成することになる．個々のテキストとそれらを連結して作った単一のテキストを区別するため，それぞれ要素テキスト，インデックステキストと呼ぶことにする．

ファイル名など，要素テキストを指し示す識別子をテキスト識別子と呼ぶ．要素テキスト e について，e の内容の文字列を $\mathrm{cont}(e)$，テキスト識別子を $\mathrm{id}(e)$ と表す．識別子 λ を持つ要素テキストの内容が更新された時，更新の前と後の要素テキストをそれぞれ e, e' とすると，$\mathrm{id}(e) = \mathrm{id}(e')$ だが e と e' は別の要素テキストである．

要素テキストの集合 $E = \{e_1,\ldots,e_k\}$ からインデックステキスト T を作成する手順は次のようになる．ここで，終端記号とはテキストに現れない特殊な文字である．

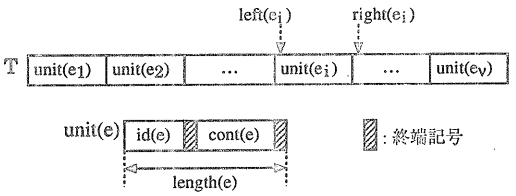

図2 インデックステキストの構造
Fig. 2 The structure of index text

(1) $1 \leq i \leq k$ について，$\mathrm{id}(e_i)$ と $\mathrm{cont}(e_i)$ のそれぞれの末尾に終端記号を付けたものを連結する．これを $\mathrm{unit}(e_i)$ とする．
(2) $1 \leq i \leq k$ の $\mathrm{unit}(e_i)$ を連結し，これをインデックステキスト T とする．

$\mathrm{unit}(e)$ の大きさを $\mathrm{length}(e)$ と表記する．インデックステキスト上の $\mathrm{unit}(e)$ の左端と右端の位置をそれぞれ $\mathrm{left}(e)$, $\mathrm{right}(e)$ と表し，これらの対 $(\mathrm{left}(e), \mathrm{right}(e))$ を $\mathrm{unit}(e)$ の位置情報と呼ぶ．ここで，$\mathrm{left}(e)$ の指す位置は $\mathrm{unit}(e)$ に含まれ，$\mathrm{right}(e)$ の指す位置は $\mathrm{unit}(e)$ に含まれない．すなわち，$\mathrm{right}(e) = \mathrm{left}(e) + \mathrm{length}(e)$ である．以後，同様に，ある領域の左端，右端の位置と言う場合，左端はその領域の内側を，右端はその領域の外側を指すことにする．インデックステキストの構造を 図2 に示す．

検索は単一のテキストの場合と同様に行い，パターンの出現位置を見つける．そこからインデックステキストを前方に走査して終端記号を見つけることで，パターンが出現した要素テキストの識別子が見つかり，また要素テキストの先頭位置からのパターンの出現位置を求めることができる．要素テキスト集合 E から構成したインデックステキスト T とそれに対する suffix array A によって，E に対する全文検索が行なえる．よって T と A の対 $I = (T, A)$ を E のインデックスと呼び，$I = \mathrm{index}(E)$ と表す．

2.3 suffix array インデックスの更新法

WWW 検索エンジンの持つインデックスを，ある要素テキスト集合に対するインデックスと区別するために，WWW 検索エンジンインデックス（WSEI）と呼ぶ．

WSEI に suffix array を用いる場合，suffix array を構成する対象となるテキストは複数の HTML ファイルであり，これらが要素テキストとなる．テキスト識別子には URL を用いる．WSEI は，これまでに収集済の要素テキスト（HTML ファイル）の集合 E のインデックス $I = \mathrm{index}(E)$ である．

HTML ファイルを全文検索する場合，HTML のタグを構成している文字列は検索の対象から除外してよい．また，日本語の文字コードの統一なども必要となる．よって，インデックステキストの構成時にこれらの前処理を行ない，それを要素テキスト e の $\mathrm{unit}(e)$ とする．

WWW 検索エンジンが最新の情報をユーザに提供できるようにするためには，Web ロボットが常に Web サイトを巡回して HTML ファイルを収集し，WSEI を定期的に更新する必要がある．WSEI の更新は要素テキストを単位として行なうことにする．

前回の WSEI の更新時刻を τ，これから行なう今回の更新時刻を $\tau'(>\tau)$ とする．現在，時刻 τ までに収集済の要素テキスト集合 E に対して作られたインデックス I があり，$\mathrm{WSEI} = I$ である．τ から τ' の間に Web ロボットの巡回によって得られた要素テキストの情報を差分情報と呼び，Δ と表記することにする．Δ によってインデックス I を更新し，時刻 τ' における最新の情報 E' に対するインデックス I' を作り，これを新たな WSEI とする．

Web ロボットが収集して来た情報は次の 3 つに分類できる．

追加 新しい要素テキスト $e(\notin E)$ を見つけ，それを持って来た．
更新 既に持っていた要素テキスト $e(\in E)$ の内容が更新されていて，更新後の内容を持って来た．
削除 既に持っていた要素テキスト $e(\in E)$ が削除されていたことが分かった．

追加，削除された要素テキストの集合をそれぞれ Δ_A, Δ_D と表す．また，更新された要素テキストの集合は更新前と更新後の物を区別して，それぞれ Δ_U^{old}, Δ_U^{new} と表す．Δ_D, Δ_U^{old} に属する要素テキストについてはそのテキスト識別子さえ判ればよい．Δ_A, Δ_U^{new} に属する要素テキストについては，そのテキストの内容も必要で，これらは Web ロボットが収集してディスク上に記録されているとする．

要素テキスト $e \in \Delta_U^{\mathrm{old}}$ が更新され $e' \in \Delta_U^{\mathrm{new}}$ となったということは，$\mathrm{id}(e) = \mathrm{id}(e')$ である e が削除され e' が追加されたとみなすことができる．$\Delta^+ = \Delta_A \cup \Delta_U^{\mathrm{new}}$, $\Delta^- = \Delta_D \cup \Delta_U^{\mathrm{old}}$ とすると，$\Delta = \{\Delta^+, \Delta^-\}$ と表せる．

E に差分情報 $\Delta = \{\Delta^+, \Delta^-\}$ を反映させた要素テキスト集合 $E \cap \overline{\Delta^-} \cup \Delta^+$ を，$E \circ \Delta$ と表すことにする．時刻 τ' の WSEI の更新では要素テキスト集合 $E' = E \circ \Delta$ に対するインデックス $I' = (T', A')$ が作られれば良い．E' を元にして初めから T', A' を構成することもできるが，E' が十分大きい場合は，suffix

array の構成に大きな時間がかかる．それよりも，既に構成されている T, A を利用し，そこに差分の情報 Δ を反映させる処理を行ない，インクリメンタルに更新を行なうようにした方が効率的である．

3. 直接更新方式

この章では，WSEI が常に一つのインデックス I を持ち，WSEI の更新のたびに I を直接更新する方式（直接更新方式）を提案する．

既にインデックス $I = \mathrm{index}(E) = (T, A)$ が作成されているとする．このとき，差分情報 $\Delta = \{\Delta^+, \Delta^-\}$ によって I を更新し，新しいインデックス $I' = \mathrm{index}(E') = (T', A')$ を作成する．この処理を $I' = DirectUpd(I, \Delta)$ と表記する．

この処理は，次の 2 つの段階に分けられる．

(1) $I^* := Delete(I, \Delta^-)$: インデックス $I = \mathrm{index}(E)$ から要素テキスト集合 Δ^- を削除し，$I^* = \mathrm{index}(E^*) = \mathrm{index}(E \cap \overline{\Delta^-})$ を作る．

(2) $I' := Append(I^*, \Delta^+)$: まず，インデックス $I^+ = \mathrm{index}(\Delta^+)$ を作る．I^* と I^+ をマージして $I' = \mathrm{index}(E^* \cup \Delta^+) = \mathrm{index}(E')$ を作る．

以下で，まずこれらの処理を説明する上で必要となるデータ構造を説明し，その後 *Delete*, *Append* のアルゴリズムを示す．

3.1 ETDB

WSEI に含まれる要素テキストに関するいくつかの情報を記録しておくために，要素テキストデータベース (ETDB) と名付けられたデータ構造を用いる．ETDB のキーは要素テキスト識別子である．

直接更新方式では，更新された要素テキストの古い方は削除されるので，ある識別子 λ を持つ要素テキストはインデックスの中に同時に一つしか存在しない．よってインデックス内にある識別子 λ を持つ要素テキスト e は一意に特定でき，これを $e = \mathrm{text}(\lambda)$ と表記する．

キー値 λ のレコード $\mathrm{Rec}(\lambda)$ は次に示すフィールドを持つ．

- pos : left(text(λ))
- len : length(text(λ))

レコード r のフィールド foo の値を r.foo と記述する．

ETDB に対する操作を次のように定義する．

- $r := ETDB\_Search(\lambda)$: キー λ に対応するレコードを探す．r は見つかったレコード，または見つからないことを表す Not Found という値．
- $ETDB\_Delete(\lambda)$: キー λ を削除する．

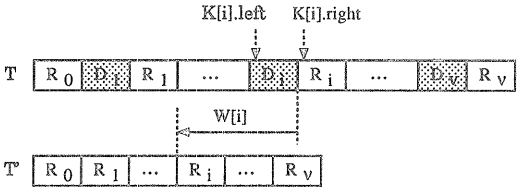

図 3 *Delete* におけるインデックステキストの処理
Fig. 3 Deletion

- $ETDB\_Update(\lambda, r)$: キー λ に対応するレコードを，新しいレコード r に更新する．

要素テキスト集合 E からインデックステキストを作成する時に，$\forall e(\in E)$ について $\lambda = \mathrm{id}(e)$ のレコードを ETDB に登録しておくことにする．またこれらの値は WSEI の更新の際に用いられ，その更新に伴いレコードの内容も更新される．

ETDB は，テキスト識別子をキーとする B^+-tree などで実装できる．

3.2 Delete(I, Δ^-) のアルゴリズム

$I' := Delete(I, \Delta^-)$ は，インデックス $I = \mathrm{index}(E) = (T, A)$ から要素テキスト集合 $\Delta^-(\subset E)$ を削除し，インデックス $I' = \mathrm{index}(E \cap \overline{\Delta^-}) = (T', A')$ を作成するアルゴリズムである．

削除すべき要素テキスト集合を $\Delta^- = \{\delta_1, \ldots, \delta_\nu\}$ とする．インデックステキスト T は，残すべき区間 $R_0, \ldots, R_\nu$ と削除すべき区間 $D_1, \ldots, D_\nu$ が交互に並んだ形 $(R_0 D_1 R_1 \ldots D_\nu R_\nu)$ となる．ここで $\forall i; D_i$ は $\mathrm{unit}(\delta_j)$ $(1 \le j \le \nu)$ のどれかである．また $\forall i; R_i$ は大きさが 0 になることもある．T 上の $R_1, \ldots, R_\nu$ を左にシフトし残すべき区間を連結した $R_0 R_1 \ldots R_\nu$ が，求める T' となる (図 3)．

そこで，まず削除すべき区間の範囲を記録した削除区間テーブル $K[1, \nu+1]$ というものを作成する．ここで，ある区間の範囲を記録するためのレコードとして，次の 2 つのフィールドを持つレコードを定義する．これを区間型レコードと呼ぶ．

- left : 区間の左端の位置
- right : 区間の右端の位置

削除区間テーブルは区間型レコードの配列で，要素 $K[i]$ は区間 D_i の範囲を記録する．また，$R_1, \ldots, R_\nu$ を左にシフトする幅を記録するシフト幅テーブル $W[1, \nu]$ も作成する．$W[i]$ は区間 R_i を左にシフトする幅である．これら 2 つのテーブルを作るアルゴリズム $(K, W) := MakeDelTable(\Delta^-)$ を図 4 に示す．ここで，n はインデックステキスト T の大きさである．

この 2 つのテーブルを用いて，T から新しいインデックステキスト T' を作る事ができる．この時，区間

アルゴリズム：*MakeDelTable*
入力：Δ^-: 要素テキスト集合
出力：K: 削除区間テーブル, W: シフト幅テーブル

```
        for i = 1,...,ν do
(1)         r := ETDB_search(id(δ_i));
(2)         K[i].left := r.pos;
(3)         K[i].right := r.pos + r.length;
(4)         if δ_i ∈ Δ_D then ETDB_Delete(id(δ_i));
        end-for
(5)     K[1,ν] を left の値によって昇順にソート;
(6)     K[ν+1].left := n+1;
(7)     W[1] = 0;
        for i = 2,...,ν do
(8)         W[i] := W[i-1] + K[i].right - K[i].left;
        end-for
```

図4 *MakeDelTable* のアルゴリズム
Fig. 4 The algorithm for *MakeDelTable*

アルゴリズム：*DelIndexText*
入力：T: インデックステキスト, K: 削除区間テーブル, W: シフト幅テーブル
出力：T': インデックステキスト

```
        for i = 1,...,ν do
(1)         x := K[i].right; y := K[i+1].left - 1;
(2)         w := W[i];
(3)         T[x-w, y-w] := T[x,y];
            T[x-w, y-w] を走査し，この区間にある
        任意の要素テキスト e について，
(4)             r := ETDB_Search(id(e));
(5)             r.pos := r.pos - w;
(6)             ETDB_Update(id(e), r);
        end-for
(7)     T[1, n-W[ν]] が T'[1,n'] となる.
```

図5 *DelIndexText* のアルゴリズム
Fig. 5 The algorithm for *DelIndexText*

$R_i(i>0)$ に含まれている要素テキストはインデックステキスト内での位置が変更されたので，これに合わせて ETDB の対応するレコードを更新する必要がある．K, W を用いて T から T' を作るアルゴリズム $T' := DelIndexText(T, K, W)$ を図5 に示す．

suffix array A から A' を作るには，区間 R_i に含まれるエントリのみを元の順序を保存して連結すれば良い．エントリ $A[j]$ がどの区間に含まれるかは，削除区間テーブルを二分探索することで求めることができる．区間 $\forall i; R_i$ に含まれる場合は残すことになる．ここでインデックステキストの更新により，T 上で区間 R_i にあったテキストは T' 上では左に $W[i]$ だけ移動している．よって $A[j]$ が R_i 内を指していた場合，$A[j]$ の値を元の値より $W[i]$ だけ小さくする．区間 $\forall i; D_i$ に含まれる場合は無視する．K, W を用いて A から A' を作るアルゴリズム $A' := DelSuffixArray(A, K, W)$ を図6 に示す．ここで，$BinSearch(K, i)$ は $K[1, \nu+1]$ を二分探索し，$K[k].\text{left} \le i < K[k+1].\text{left}$ となる k を返す関数である．

アルゴリズム：*DelSuffixArray*
入力：A: suffix array , K: 削除区間テーブル, W: シフト幅テーブル
出力：A': suffix array

```
(1)     j := 1;
        for i = 1,...,n do
(2)         k := BinSearch(K, A[i]);
            if A[i] < K[k].right then
(3)             A[j] := A[i] - W[k]; j := j+1;
        end-for
(4)     A[1, n-W[ν]] が A'[1,n'] となる.
```

図6 *DelSuffixArray* のアルゴリズム
Fig. 6 The algorithm for *DelSuffixArray*

したがって，アルゴリズム $I' = Delete(I, \Delta^-)$ は，つぎの 3 つの段階に分けられる．

(1) $(K, W) := MakeDelTable(\Delta^-)$;
(2) $T' := DelIndexText(T, K, W)$;
(3) $A' := DelSuffixArray(A, K, W)$;

3.3 Append(I, Δ^+) のアルゴリズム

$I' := Append(I, \Delta^+)$ は，既存のインデックス $I = \text{index}(E)$ と，差分の要素テキスト集合 Δ^+ に対するインデックス $I^+ = \text{index}(\Delta^+)$ をマージし，インデックス $I' = \text{index}(E \cup \Delta^+)$ を作成するアルゴリズムである．

まず，二つのインデックス $I_1 = \text{index}(E_1) = (T_1, A_1)$, $I_2 = \text{index}(E_2) = (T_2, A_2)$ をマージして一つのインデックス $I' = \text{index}(E_1 \cup E_2) = (T_2, A_2)$ を作るアルゴリズム $I' := Merge(I_1, I_2)$ を定義する．

$T'[1, n']$ は，単純に $T_1[1, n_1]$ の後ろに $T_2[1, n_2]$ を連結することで作成する．それに合わせて A' を構成す

アルゴリズム：*Merge*
入力：I_1, I_2: インデックス
出力：I: インデックス

(1) T_1 の後ろに T_2 を連結し，これを T' とする．
(2) $i := n_1$; $j := n_2$;
while true do
if suf$(T_1, A_1[i]) >_L$ suf$(T_2, A_2[j])$ **then**
(3) $A_1[i+j] := A_1[i]$; $i := i-1$;
if $i = 0$ **then exit-while**
else
(4) $A_1[i+j] := A_2[j] + n_1$; $j := j-1$;
if $j = 0$ **then exit-while**
end-if
end-while
if $i = 0$ **then**
while $j > 0$ **do**
(5) $A_1[j] := A_2[j] + n_1$; $j := j-1$;
end-while
(6) A_1 が A' となる．

図7 *Merge* のアルゴリズム
Fig. 7 The algorithm for *Merge*

るには，まず A_1 の配列を $|A_2|$ だけ大きくする．A_1 と A_2 の要素を後ろから順に比較し，大きい方を選択して A_1 の後ろから並べて行けば良い．このとき，T_2 は T_1 の後ろに連結したため，A_2 のエントリが指していた接尾辞は元の位置より $n_1 = |T_1|$ だけ後ろに移動している．したがって A_2 の要素を A_1 上にコピーする際には，元の値より n_1 だけ大きくする必要がある．$I' := Merge(I_1, I_2)$ のアルゴリズムを図7に示す．ここで，suf(T, i) はインデックステキスト T の i 番目の文字から始まる接尾辞である．$>_L$ は2つの文字列を辞書順で比較し左辺が大きいときに真となる演算子である．

T_2 に含まれていた要素テキストについて，ETDB に登録してある位置情報を更新する必要があるが，図7のアルゴリズムではそれをしていない．よって，マージ後に T_2 部分を走査してそこに含まれる要素テキストを見つけて，それらに対して ETDB のレコードの更新を行なう必要がある．しかし，T_2 の作成時に，作成後すぐにマージされて T_1 の後ろに連結されることが分かっているならば，ETDB に位置情報を登録する際に予め $|T_1|$ だけ大きくしておくことができる．その方が，一度登録したレコードを検索してまた更新するよりもコストが小さい．したがって，$I' := \text{Append}(I, \Delta^+)$ のアルゴリズムは次のようになる．

(1) $I^+ := \text{index}(\Delta^+) = (T^+, A^+)$ の作成：
ここで，T^+ の作成時に ETDB に登録する left(e) の値は実際の値に $n = |T|$ を加えたものとする．$e \in \Delta_A$ ならば新規レコードの挿入，$e \in \Delta_U^{\text{new}}$ ならばレコードの更新である．
(2) $I' := Merge(I, I^+)$;

4. インクリメンタルな更新方式

WSEI が単一のインデックス，すなわち一組のインデックステキストと suffix array から構成されていると，要素テキスト集合のわずかな変更でも巨大なインデックステキストや suffix array の全体を処理する必要があり，コストが大きすぎる．

そこで，WSEI を複数のインデックスから構成し，WSEI の更新をインクリメンタルに行なう方式を提案する．この方式では既存のインデックスは直接は更新せず，差分の要素テキストから別個のインデックスを構成していき，WSEI の一回の更新にかかる時間を軽減させる．この更新方式をインクリメンタルな更新方式と呼ぶ．

以下では，まずこの方式の基本戦略について述べ，いくつかの準備をした後，アルゴリズムを記述する．

4.1 基本戦略

初め WSEI は一つのインデックス $I_0 = \text{index}(E_0) = (T_0, A_0)$ だけを持っているとする．この後，差分情報 $\Delta_1 = (\Delta_1^+, \Delta_1^-)$ によって行なう WSEI の更新では，I_0 は全く変更せずに別のインデックス $I_1 = \text{index}(\Delta_1^+)$ を構成する．以後，同様にして $\Delta_2, \ldots, \Delta_m$ による WSEI の更新のたびに $I_2, \ldots, I_m$ を構成していく．$\Delta_1^-, \ldots, \Delta_m^-$ に属するテキストは実際に削除はせず，削除されたという情報だけを別に残しておけばよい．m 回の更新の後，WSEI は $m+1$ 個のインデックス $I_0, I_1, \ldots, I_m$ で構成される．$i(\geq 1)$ 回目の WSEI の更新で追加された差分情報 Δ_i^+ から作られるインデックス I_i を，差分インデックスと呼ぶ．これに対し，I_0 を主インデックスと呼ぶ．

この方式では，削除された要素テキストや更新された古い内容の要素テキストをインデックスの中に残したままになっている．これらの要素テキストを無効な要素テキストと呼び，それ以外のものを有効な要素テキストと呼ぶ．要素テキストが有効か無効かの判別には，後述する有効インデックス番号を用いる．検索は複数のインデックスのそれぞれに対して行ない，得られた結果の中から有効なものだけを取り出し，それらをマージして全

体の結果とする．

WSEI の更新を行なっていくに従い，インデックスの個数も増大していく．ある程度の個数になったら有効な要素テキストのみから一つの主インデックスを再構成し，再び WSEI が一つのインデックスだけを持つ状態にする必要がある．

4.2 準　　備

ここでは，インクリメンタルな更新方式の説明の準備として，それに必要な概念について述べる．

4.2.1 有効インデックス番号

I_k に含まれている要素テキスト e が有効であるとは，$k+1$ 回目以降の WSEI の更新で e が一度も，削除も更新もされていないことである．

ある識別子 λ を持つ要素テキストが更新されたり，または一度削除されて再び追加されたりすることによって，識別子が λ である複数の要素テキスト $e_1,\ldots,e_k$ がそれぞれ異なるインデックス $I_{x_1},\ldots,I_{x_k}$ に同時に存在することになる．ここで $x_1<\ldots<x_k$ とする．このとき，$e_1,\ldots,e_{k-1}$ は無効な要素テキストである．e_k は，x_k+1 回目以降の更新で削除されていなければ有効である．したがって，ある識別子を持つ複数の要素テキストの中で有効なものは高々一つしか無い．テキスト識別子 λ を持つ有効な要素テキストを $\mathrm{vet}(\lambda)$ と表す．

そこで，要素テキスト識別子ごとに有効インデックス番号というものを設定し，これによって要素テキストが有効かどうかを判定する．テキスト識別子 λ の有効インデックス番号とは，識別子 λ を持つ有効な要素テキスト $\mathrm{vet}(\lambda)$ が含まれているインデックスの番号である．λ の有効インデックス番号を $\mathrm{vin}(\lambda)$ と表す．

有効インデックス番号はテキスト識別子ごとに設定する値なので，ETDB に記録できる．また，この更新方式では同じ識別子を持つ要素テキストが複数のインデックス内に存在し得る．そこで，キー値 λ に対する ETDB のレコード $\mathrm{Rec}(\lambda)$ のフィールドを，次のように定義し直す．

- vin : $\mathrm{vin}(\lambda)$
- pos : $T_{\mathrm{vin}(\lambda)}$ 内の $\mathrm{left}(\mathrm{vet}(\lambda))$
- len : $\mathrm{length}(\mathrm{vet}(\lambda))$

インデックステキストの作成時に，pos, len と同様に vin の値も ETDB に登録することになる．ETDB に識別子 λ のエントリが無ければ，識別子が λ である有効な要素テキストが存在しないことを意味する．

4.2.2 Garbage リスト

主インデックスの再構成をする時，それぞれのインデックステキスト内に残っている無効な要素テキストを削除することになる．しかし ETDB には有効な要素テキストの情報しか格納されない．よって，各インデックス I_i に対して，そのインデックスに含まれる無効な要素テキストの位置情報を記録しておくための garbage リスト G_i を用意する．

garbage リストは区間型レコードのリストである．garbage リスト G の j 番目の要素を $G[j]$ と表記する．新しく作られたインデックスの garbage リストは空である．インデックス I_i に含まれている要素テキスト e が無効になった時，$\mathrm{unit}(e)$ の範囲 $(\mathrm{left}(e),\mathrm{right}(e))$ を garbage リスト G_i に追加する．garbage リスト G に位置情報 (l,r) を追加する手続きを $AddGarbage(G,l,r)$ と表記する．

4.3 アルゴリズム

ここでは，更新，検索，主インデックスの再構成のアルゴリズムについて述べる．

4.3.1 更新（差分インデックスの作成）

差分情報 $\Delta=\{\Delta^+,\Delta^-\}$ によって k 番目の差分インデックス $I_k=(T_k,A_k)$ を作るアルゴリズム $I_k:=MakeDiff(\Delta,k)$ について述べる．

まず，削除すべき要素テキスト $\delta(\in\Delta^-)$ に対して，ここでは実際の削除は行なわない．後でまとめて削除するように，δ を含んでいるインデックスの garbage リストに δ の位置情報を記録しておく．このとき $\delta\in\Delta_D$ ならば ETDB からキー $\mathrm{id}(\delta)$ を削除し，Δ_D に含まれる要素テキストを無効化しておく．また，差分インデックス I_k は要素テキスト集合 Δ^+ から単純に作成すれば良い．このとき，ETDB に登録する要素テキスト $e(\in\Delta^+)$ の有効インデックス番号は k である．これにより，Δ_U^{old} に含まれた要素テキストも無効になる．$I_k:=MakeDiff(\Delta,k)$ のアルゴリズムを図 8 に示す．ここで $\Delta^-=\{\delta_1,\ldots,\delta_\nu\}$ である．

4.3.2 検　　索

WSEI が持つ複数のインデックス $\{I_0,\ldots,I_m\}$ に対する検索は，まず各インデックス I_i に対して別個に行なう．このとき検索結果の中に無効な要素テキストが現れることもあるので有効かどうかの検査をし，有効なものだけを集めて I_i の検索結果 R_i とする．$R_0,\ldots,R_m$ をマージしたものが $WSEI$ の検索結果となる．

インデックス I_k に含まれ識別子 λ を持つ要素テキストが有効かを調べ，有効なら真，無効なら偽を返すアルゴリズム $b=IsValid(\lambda,k)$ を図 9 に示す．

4.3.3 主インデックスの再構成

WSEI は m 回の更新の後，$m+1$ 個のインデックス $I_0,\ldots,I_m$ を持つ．この状態から，再び WSEI が単一のインデックス $I_0'=\mathrm{index}(E_0')$ だけを持つ状態にする．ここで $E_0'=E_0\circ\Delta_1\circ\ldots\circ\Delta_m$ である．

アルゴリズム：*MakeDiff*
入力：Δ: 差分情報, k: インデックス番号
出力：I_k: インデックス

```
        for i = 1,...,ν do
(1)         r := ETDB_Search(id(δ_i));
(2)         v := r.vin; l := r.pos; r := l + r.length;
(3)         AddGarbage(G_v, l, r);
(4)         if δ_i∈Δ_D then ETDB_Delete(id(δ_i));
        end-for
```

(5) Δ^+ からインデックステキスト T_k を作る．このとき，ETDB に登録するレコードの有効インデックス番号は k である．$e \in \Delta_A$ なら新規レコードの挿入，$e \in \Delta_U^{\text{new}}$ なら既存レコードの更新である．

(6) T_k に対して A_k を構成する．

図8 *MakeDiff* のアルゴリズム
Fig. 8 The algorithm for *MakeDiff*

アルゴリズム：*IsValid*
入力：λ: テキスト識別子, k: インデックス番号
出力：b: ブール値

```
(1)     r := ETDB_Search(λ);
        if r = NotFound then
(2)         b := False;
        else
            if r.vin = k then
(3)             b := True;
            else
(4)             b := False;
```

図9 *IsValid* のアルゴリズム
Fig. 9 The algorithm for *IsValid*

$m+1$ 個のインデックス $I_0, \ldots, I_m$ から新たに一つの主インデックス $I'_0 = (T'_0, A'_0)$ を構成するアルゴリズム $I'_0 := Rebuild(I_0, \ldots, I_m)$ について述べる．

まず，garbage リスト $G[1, \nu]$ を持つインデックス I から無効な要素テキストを取り除いたインデックス I' を構成するアルゴリズム $I' := DeleteGarbage(I, G)$ を定義する．

この手続きは 3.2 節の $I' := Delete(I, \Delta^-)$ とほぼ同じで，異なるのは削除区間テーブル作成の処理だけである．削除すべき区間の位置情報は既に garbage リストに記録されている．また，I に含まれている無効な要素テキストに関する情報は既に ETDB から取り除かれている．よって $I' := DeleteGarbage(I, G)$ のアルゴリズムは，$I' := Delete(I, \Delta^-)$ のアルゴリズムにおいて図4の最初の for ループを以下のように置き換えた物となる．

```
        for i = 1,...,ν do
            K[i] := G[i];
        end-for
```

$m+1$ 個のインデックスから一つのインデックスを再構成するには，まず，$\forall i$ に対して $I_i^* := DeleteGarbage(I_i, G_i)$ を行ない，有効な要素テキストのみを含むインデックス $I_0^*, \ldots, I_m^*$ を作る．ここで I_m は最新のインデックスなので garbage リストは空であり，そのままでよい．これら全てを *Merge* によって順にマージしていき，最終的に一つのインデックス $I'_0 = \text{index}(E_0 \circ \Delta_1 \circ \ldots \circ \Delta_m) = (T'_0, A'_0)$ を作る．このとき，一般に主インデックスは差分インデックスに対して十分大きいので，まず差分インデックス同士のマージを行ない，最後に主インデックスとマージする．このとき，T'_0 内における各要素テキストの位置は以前とは変化し，また全ての要素テキストの有効インデックス番号は 0 に変わったが，ETDB の持つ情報は古いままである．よって T'_0 を走査して順にテキスト識別子を見つけ，ETDB の位置情報と有効インデックス番号を現在の物に更新する．ここで各要素テキスト有効インデックス番号は全て 0 である．図10に $I'_0 = Rebuild(I_0, \ldots, I_m)$ のアルゴリズムを示す．

5. 直接更新方式とインクリメンタルな更新方式の統合

インクリメンタルな更新方式は，直接更新方式に比べて WSEI の一回の更新にかかる時間が短いが，毎回の更新でインデックスの個数が一つずつ増え，それにしたがい検索にかかる時間も増大してしまう．直接更新方式は更新時間が大きいが，インデックスが常に一つなので検索時間は小さい．そこで，*DirectUpd* と *MakeDiff* を併用し，両方の方式の利点を組み合わせた更新方式を提案する．この方式を統合方式とよぶ．

まず．統合方式による更新の方法について述べる．この方式では，WSEI の更新を 2 通りの方法から選択することになる．よって次に，どのような順序で更新を行なっていくかについて説明する．

5.1 統合方式による更新

毎回の更新で差分インデックスを構成するのではなく，既に構成されている差分インデックスに対して直接

アルゴリズム：*Rebuild*
入力：$I_0,\ldots,I_m$: インデックス
出力：I'_0: インデックス

for $i=0,\ldots,m-1$ do
(1) $I_i^* := DeleteGarbage(I_i, G_i)$;
end-for
(2) $I_m^* := I_m$;
(3) $I_1^{(1)} := I_1^*$;
for $i=2,\ldots,m$ do
(4) $I_1^{(i)} := Merge(I_1^{(i-1)}, I_i^*)$;
end-for
(5) $I'_0 := Merge(I_0^*, I_1^{(m)})$;
(6) T'_0 を走査して見つけたすべてのテキスト識別子 λ に対し，ETDB の Rec(λ) の内容を現在の値に更新する．

図10 *Rebuild* のアルゴリズム
Fig. 10 The algorithm for *Rebuild*

更新を行なうようにすると，毎回の更新で差分インデックスを作る場合よりもインデックスの個数が少なくなり，検索時間が短くなる．また，差分インデックスの直接更新は，差分インデックスを新たに作るよりは時間がかかるが，巨大な主インデックスを直接更新するよりは高速である．

初めに WSEI には主インデックス I_0 だけがあるとする．最初の Δ_1 による更新では差分インデックス $I_1=\mathrm{index}(\Delta_1^+)$ を作る．次の Δ_2 による更新では，差分インデックス I_1 を *DirectUpd* によって直接更新し，$I_1=\mathrm{index}(\Delta_1^+\circ\Delta_2)$ とする．その後も x 回目までは I_1 を直接更新し，$I_1=\mathrm{index}(\Delta_1^+\circ\Delta_2\circ\ldots\circ\Delta_x)$ とする．$x+1$ 回目の更新では再び差分インデックス I_2 を作り，その後は I_2 に対して直接更新を繰り返す．以後も同様にして，数回おきに新規の差分インデックスを作り，それ以外の更新では最新の差分インデックスに対して直接更新を行なう．

ここで 3 章で定義した *DirectUpd* では，インデックスが常に一つで，削除すべき要素テキストが全て対象のインデックスに含まれていることを前提としていた．しかし，差分インデックス I_k を更新情報 $\Delta=(\Delta^+,\Delta^-)$ で更新しようとした場合，要素テキスト $\delta_i(\in\Delta^-)$ は I_k に含まれているとは限らない．$v=\mathrm{vin}(\delta_i)$ としたとき，$v=k$ ならば δ_i は I_k に含まれているので，Delete の処理の際に削除することになる．$v\neq k$ ならば δ_i は別のインデックス I_v に含まれているので，後の *Rebuild* の際に削除するように garbage リスト G_v に記録しておけばよい．そこで，*DirectUpd* の中の *Delete* において削除区間テーブルを作成する部分（図 4 の最初の for ループの部分）の処理を以下のように定義し直し，それを *DirectUpd* として用いる．ここで $\Delta^-=\{\delta_1,\ldots,\delta_k\}$ である．また，x は区間型レコードである．

(1) $j:=0$;
for $i=1,\ldots,k$ do
(2) $r := ETDB\_Search(\mathrm{id}(\delta_i))$;
(3) $v := r.\mathrm{vin}$;
(4) $x.\mathrm{left} := r.\mathrm{pos}$; $x.\mathrm{right} := r.\mathrm{pos}+r.\mathrm{length}$;
(5) if $\delta_i\in\Delta_D$ then $ETDB\_Delete(\mathrm{id}(\delta_i))$;
(6) if $v=k$ then $j:=j+1$; $K[j]:=x$;
(7) else $AddGarbage(G_v, x.\mathrm{left}, x.\mathrm{right})$;
end-for
(8) $\nu := j$;

その他の部分は元の *DirectUpd* と同じである．

MakeDiff, *IsValid*, *Rebuild* については，4 章で述べたものをそのまま用いる事ができる．

5.2 統合方式のスケジューリング

統合方式では，各回の WSEI の更新で行なう処理を次の 2 つから選択する．

- 新規の差分インデックスの作成
- 最新のインデックスの直接更新

更新処理のスケジュール，すなわちこれらの処理をどのような順序で行なうかは，次の 2 つのパラメータによって決定される．

- M：差分インデックスの最大の個数
- N：一つの差分インデックスの最大の大きさ

また，N の代わりに

- X：一つの差分インデックスを更新し続ける回数

を用いることもできる．初期インデックス I_0 だけが存在する状態から数回の WSEI の更新を行ない，再構成によって新たな主インデックス I'_0 を作るまでの更新処理のスケジューリングのアルゴリズム $I'_0 := WSEI\_Update(I_0,\Delta_1,\ldots)$ を図 11 に示す．主インデックスの再構成を行なった後は，I'_0 を新たな初期インデックス I_0 としてこのアルゴリズムを繰り返せばよい．

$M=0$ とすると *DirectUpd* のみで更新を行なう．つまり直接更新方式となる．また，$M\neq 0, N=0$ とすると *MakeDiff* のみで更新を行ない，これはインクリメンタルな更新方式である．すなわち，これらの方式は，統合方式のバリエーションと見ることができる．

アルゴリズム：*WSEI_Update*
入力：I_0: インデックス, $\Delta_1, \ldots$: 差分情報
出力：I_0': インデックス

(1) $m := 0, i := 0;$
while $m \leq M$ do
(2) $i := i + 1;$
if $(M > 0) \wedge (|I_m| > N \vee X = 1 \vee i \bmod X = 1)$ then
(3) $m := m + 1; I_m := MakeDiff(\Delta_i, m);$
else
(4) $DirectUpd(I_m, \Delta_i);$
end-while
(5) $I_0' := Rebuild(I_0, \ldots, I_m)$

図11 *WSEI_Update* のアルゴリズム
Fig. 11 The algorithm for *WSEI_Update*

6. 実　　験

suffix array を用いた WWW 検索エンジンを Sun Ultra 60 (CPU : Ultra Sparc 360 MHz, メモリ 640 MByte) 上に実装し，5章で述べた統合方式によってインクリメンタルに更新する実験を行なった．

6.1 実　　装

suffix array の構成には，doubling technique および numbering technique[3] を使った文献 5) で提案されているアルゴリズムを用いた．このアルゴリズムでは，大きさ n のテキストに対する suffix array をメモリ上で最悪 $O(n \log n)$ の時間で構成でき，また同じ文字列が繰り返し現れる場合でも遅くならない．初期インデックスの構成の際は suffix array が巨大になりメモリ上に入り切らないため，複数に分割して作成し Merge によってマージして一つの suffix array とした．

HTML ファイルからインデックステキストを作成する際，検索に不要な HTML のタグの除去や日本語の文字コードの統一などの前処理を行なった．各更新方式の入力となる差分情報は，HTML ファイルの URL のリストとした．Web ロボットが定期的に HTML ファイルを収集し，前回に収集した物と比較して追加，更新，削除された HTML ファイルのリストを作成した．検索結果は，まずキーワードの出現ごとに，それを含む HTML ファイルの URL とファイル内での出現位置のオフセットの対の形で求めた．同じ URL の物をまとめて URL とそこでの出現位置のリストを合わせて一件の結果とし，最終的にはこれのリストの形で表した．

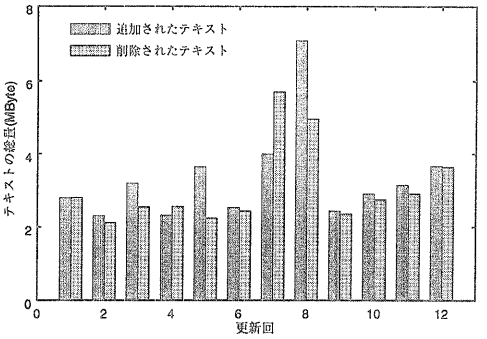

図12 1日当たりのテキストの更新量
Fig. 12 The size of updated text per one day

6.2 実験に使用したデータ

実験には，実際の Web サーバにある HTML ファイルを用いて，それに対するインデックスを作成した．インデックスを作成する対象は，is.tsukuba.ac.jp ドメインの下にある HTML ファイルのうちトップページから辿れるもの全てとした．

まず初期データとして対象 Web サーバにある全 HTML ファイルを収集した．これを E_0 とする．E_0 は合計 256.06 メガバイト (31014 ファイル，1 ファイル平均 8.75 キロバイト) であり，これから $I_0 = \mathrm{index}(E_0)$ を構成するのに 5 回の平均で 124777 秒 (およそ 3 時間 50 分) かかった．

その後 12 日間に渡って毎日，同一時刻に前日からの差分情報を収集し，それらを $\Delta_1, \ldots, \Delta_{12}$ とした．第 1 回目の更新から順に，各回の更新に 1, 2, ..., 12 と番号をつける．この番号を更新回と呼ぶ．一日当たりに追加，削除された HTML ファイルの大きさの合計を図 12 に示す．ここで，更新された HTML ファイルは，古いものを削除し新しいものを追加したとみなし，それぞれ削除，追加されたテキストの大きさに加えている．つまり，$i = 1, \ldots, 12$ についての Δ_i^+, Δ_i^- に属する要素テキストの大きさの合計である．図 12 より，毎日の更新量は全体のおよそ 1 ～ 3% 程である．

6.3 実験方法

まず予め主インデックス $I_0 = \mathrm{index}(E_0)$ を構成しておく．その後，差分情報 $\Delta_1, \ldots, \Delta_{12}$ によって更新を行なう．ここで，次のような 5 通りの更新スケジュールを考え，それぞれのスケジュールで更新を行なった．

- A : 差分インデックスは作らない (直接更新方式)
- B : 毎回，差分インデックスを作る (インクリメンタルな更新方式)
- C : 3 回おきに差分インデックスを作る．
- D : 6 回おきに差分インデックスを作る．
- E : 差分インデックスを一つだけ作り，それを更新

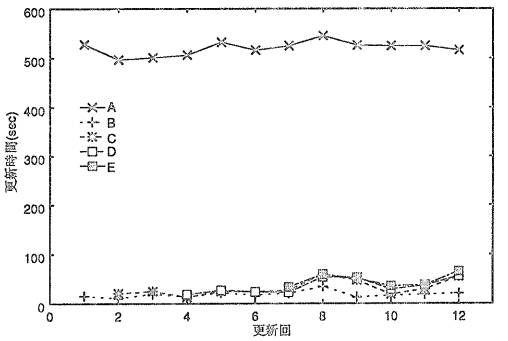

図13 更新時間
Fig. 13 The time for updating

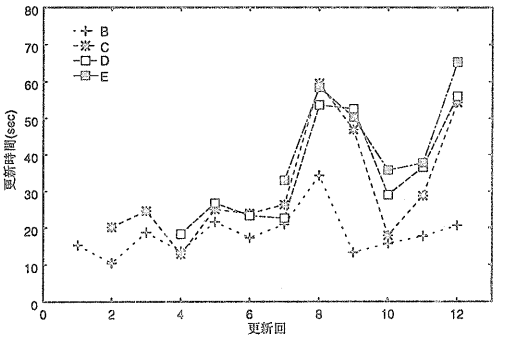

図14 更新時間 (スケジュール B, C, D, E)
Fig. 14 The time for updating (B, C, D, E)

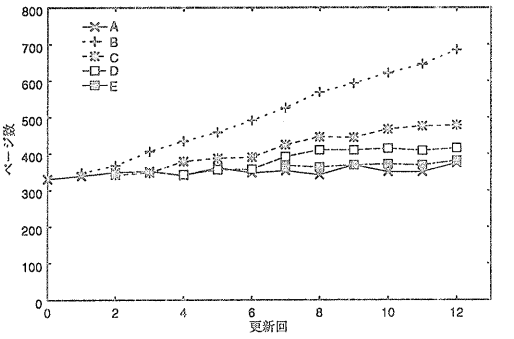

図15 検索時のメモリマップファイル入出力数
Fig. 15 The number of page faults in searching

し続ける．
これは統合方式においてパラメータ X の値をそれぞれ 0, 1, 3, 6, 12 と定めたことに相当している．

各スケジュールの毎回の更新で，以下の計測を行なった．

- その更新処理に要した時間
- その更新後の状態での検索処理における入出力ページ数☆
- その更新後の状態からの主インデックスの再構成を行うのに要した時間（A を除く）

検索はある一つの決まった単語に対して行なった．この単語に対する検索で得られた結果は 109 件から 114 件でほぼ一定である．なお，スケジュール C の 1 回目の更新はスケジュール B の処理と同じなので，C の実験では B の 1 回目の更新で作られた状態を用いて，2 回目以降からの更新処理を行なった．同様に D, E はそれぞれ C, D の処理と途中まで同じで，それぞれ 4, 7 回目以降からの更新を行なった．

計測はコールドスタートで行ない，更新と再構成は各々 1 回，検索は 3 回ずつ計測し平均を取った．ここでコールドスタートとは，主記憶上のキャッシュバッファを空にした直後に実験を開始することを意味する．

6.4 実験結果の比較

6.4.1 更新処理

更新処理に要した時間を図 13, 図 14 に示す．図 14 のスケジュール E の更新時間は差分インデックスの新規作成にかかる時間である．この時間が差分のテキストの大きさにほぼ比例していることが，図 12 との比較からわかる．またインデックスの直接更新では差分のテキストから suffix array を構成した後にそれを既存のインデックスにマージするため，差分インデックスの新規作成の場合よりマージ処理の分だけ余計に時間がかかる．このマージ処理にかかる時間は差分のテキストの大きさとマージの対象であるインデックスの大きさの和にほぼ比例している．よって全体のテキストの量に対して更新されるテキストの量が十分小さい場合，主インデックスを直接更新するよりも差分インデックスの作成および差分インデックスの直接更新をする方が更新処理にかかる時間が小さいといえる．スケジュール A での更新処理にかかるコストを 1 とした場合のスケジュール B, C, D, E でのコストの比の最大値はそれぞれ 0.063, 0.108, 0.108, 0.127 である．更新速度のみを考えればスケジュール B が最も速く，C, D, E も A よりは大幅に効率的である．

6.4.2 検索処理

検索処理時の性能を計測するために，suffix array ファイルとインデックステキストへのアクセスに伴って引き起こされた入出力ページ数を計測した．その結果を図 15 に示す．

大きさ n の一つのインデックスの二分検索には $O(\log n)$ 回の I/O が必要である．よってインデックス

☆ 更新処理に関する性能計測が実時間を対象としていたのに対し，検索処理に関する性能計測を入出力ページ数で行っているのは，実験環境の制約による．検索処理は，1 回の処理に要する時間が数秒程度と短く，実時間で計測した場合に安定性のある計測が行えなかったためである．

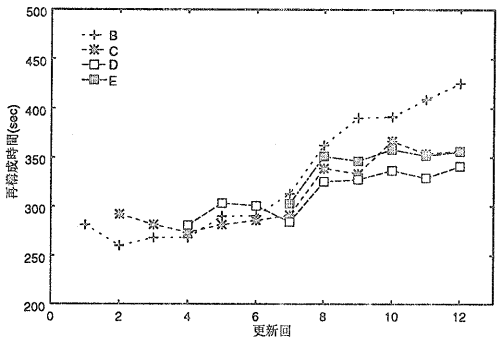

図16 再構成時間
Fig. 16 The time for rebuilding

自身の大きさの増加よりもインデックスの個数の増加の方が検索のコストにより大きく影響する．このことは図15にも現れている．スケジュールB, C, D, Eではそれぞれ1, 3, 6, 12回の更新毎に一つずつ差分インデックスが作られる．どの場合も更新に伴い検索時のページ入出力回数が増大しているが，インデックスの個数の増加速度が大きいものほどページ入出力回数の増大も急激である．スケジュールEではAと比較してほとんど差がないのに対し，スケジュールBで12回の更新後には検索にAの場合のおよそ2倍のコストがかかる．

6.4.3 再構成処理

それぞれの場合で主インデックスの再構成に要した時間を，図16に示す．再構成にかかる時間は更新の回数に伴い増大している．スケジュールC, D, Eでは差分インデックスの個数，大きさが異なるにもかかわらず，再構成かかった時間にあまり差がない．これは，主インデックスは差分インデックスに対して十分に大きく（数10倍から100倍程度），差分インデックスを一つにマージするまでの時間がそれを主インデックスにマージする時間に対して十分小さくなるためと考えられる．

また図13と図16を比較すると，主インデックスの再構成にかかるコストは一回の主インデックスの直接更新より小さいことがわかる．したがって，スケジュールB～Eにおいて数回の更新に一度必要となる再構成処理はAの更新と同程度またはそれ以下のコストであり，それ以外の通常の更新処理はAの場合の1/10程度のコストしかかからない．

6.5 考　　察

主インデックスの直接更新のみを行うスケジュールAでは図15で示されているように検索が最も高速に行えるかわりに，更新処理には他のスケジュールの場合の10倍以上の時間がかかった(図13)．一方，スケジュールEではAとほぼ同じ速度で検索を行うことができ，毎回の更新処理にかかる時間はAのおよそ1/10以下である．Eでは主インデックスの再構成を行う必要があるが，これもAでの一回の更新処理にかかる時間より小さい(図13, 図16)．以上の結果から，更新されるテキストの大きさが主インデックスの持つテキスト集合に対して十分小さいとき，差分インデックスを用いた統合方式が有効であるといえる．

統合方式を用いる場合，5.2節で述べたパラメータMとNまたはXを決める必要がある．パラメータNは一つの差分インデックスの最大の大きさである．この値を大きく取ることによって一つの差分インデックスを直接更新する回数が多くなり，差分インデックスの個数の増加を抑え，検索効率の低下を防ぐことができる．しかし，差分インデックスが大きくなることにより一回の更新処理にかかる時間が増大する．そこで許容できる最大の更新時間τ_uを設定し，更新時間がτ_uを超えない最大のNを求める．一回の更新処理の差分情報Δ^+, Δ^-のテキストサイズの和の平均をそれぞれm, n，更新対象のインデックスのサイズをkとする．このとき，直接更新を行うアルゴリズム*DirectUpdate*における各処理にかかる時間が以下のように表せるものと仮定する．ここで$f_i(x)$はxの関数，C_jは定数である．

- *MakeDelTable* … $f_1(n)$
- *DelIndexText* … $C_1(k-n)$
- *DelSuffixArray* … $kf_2(n)$
- index(Δ^+) の作成 … $f_3(m)$
- *Merge* … $C_2(k-n+m)$

このとき，以下の式が成立する．

$$\tau_u > f_1(n) + C_1(k-n) + kf_2(n) + f_3(m) + C_2(k-n+m)$$

$$k < \{\tau_u - f_1(n) + C_1 n - f_3(m) - C_2(m-n)\}/\{C_1 + f_2(n) + C_2\} \quad (1)$$

よって式(1)を満たす最大のkをNとする．また，更新されるテキストのサイズが一定であるとすると，1回の更新で差分インデックスのサイズが$m-n$ずつ増加していくことから$X = N/(m-n)$となる．

パラメータMは差分インデックスの最大の個数である．この値を大きくすることで，多大な時間を要する主インデックスの再構成処理の回数を減らすことができる．しかしMの値を大きくすると差分インデックスの個数が増え，それに伴い検索速度が低下する．そこで許容できる検索時間をτ_sと仮定する．主インデックスのサイズをm，差分インデックスのサイズをn，サイズxのインデックスの二分探索でパターンの出現位置を見つけるのに要する時間を$f(x)$，一つのパターンの出現位置からテキストを走査してテキスト識別子を見つけ有効かを検査する時間をtとする．また差分イ

ンデックスの個数を k, 検索で見つかるパターンの出現数の平均を l とする．このとき検索処理の平均時間は $f(m)+kf(n)+lt$ となり，以下の式が成り立つ．

$$\tau_s > f(m)+kf(n)+lt$$

すなわち

$$k < \frac{\tau_s - f(m) - lt}{f(n)} \qquad (2)$$

したがって式 (2) を満たす最大の k を M とすればよい．

7. おわりに

本論文では，suffix array の更新方式として，まず一つの suffix array を更新していく直接更新方式を提案し，次に差分 suffix array を作っていくインクリメンタルな更新方式を提案した．そして，より効率的な更新方式として，先の 2 つの方式を組み合わせた統合方式を提案した．この方式では，更新スケジュールを決定するパラメータによって更新処理，検索処理にかかる時間が変化する．これらのパラメータは，許容できる更新時間，検索時間に合わせて設定する．

実装を用いた実験を行ない，単一の suffix array を更新し続ける方式よりも，差分のインデックスを作ってそれを更新した方が効率的に更新を行なえることを確かめた．また，パラメータによる性能の変化を比較し，一つの差分インデックスの更新を長い期間行ない，差分インデックスの個数はあまり大きくしない方がよいことが分かった．

今後の課題としては，差分インデックスの作成，検索の処理を並列に行なうことを検討している．また，論文[4] に述べられているモバイルオブジェクト技術を用いた情報収集ロボット技術と統合して Web 検索システムを構築することを予定している．

謝辞 suffix array に関して有益な議論をして頂くと共に suffix array 作成のためのプログラムコードを提供して頂いた東京大学大学院理学系研究科の今井浩先生，および定兼邦彦氏に感謝します．また，多くの貴重な助言をして頂いた元 筑波大学大学院工学研究科（現 日立製作所中央研究所）の清水晃氏，および実験データ収集に協力して頂いた筑波大学第三学群情報学類の大久保博之氏に感謝します．

参考文献

1) Ferragina, P. and Grossi, R.: Fast String Searching in Secondary Storage: Theoretical Developments and Experimental Results, *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*, Atlanta, ACM Press, pp. 373–382 (1996).
2) Gonnet, G. H., Baeza-Yates, R. A. and Sinder, T.: New indices for text: PAT trees and PAT arrays., *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, chapter 5, pp. 66–82 (1992).
3) Karp, R. M., Miller, R. E. and Rosenberg, A.: Rapid identification of repeated patterns in strings, arrays and trees, *4th ACM Symposium on Theory of Computing*, pp. 125–136 (1972).
4) Kato, K., Someya, Y., Matsubara, K., Toumura, K. and Abe, H.: An Approach to Mobile Software Robots for the WWW, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 4, pp. 526–548 (1999). Special Issue on Web Technologies.
5) Larsson, N. J. and Sadakane, K.: Faster Suffix Sorting, Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden (1999).
6) Manber, U. and Myers, E.: Suffix arrays: A new method for on-line string searches, *Proceedings of the 1stAnnual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, pp. 319–327 (1990).
7) McCreight, E. M.: A space-economical suffix tree construction algorithm, *Journal of the ACM*, Vol. 23, No. 2, pp. 262–272 (1976).
8) Navarro, G., Kitajima, J., Ribeiro, B. and Ziviani, N.: Distributed Generation of Suffix Arrays, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97)* (Apostolico, A. and Hein, J.(eds.)), LNCS 1264, pp. 102–115 (1997).
9) Sonnenreich, W. and Macinta, T.: *Guide to Search Engines*, John Wiley & Sons, Inc. (1998).

(平成11年6月20日受付)
(平成11年9月27日採録)

(担当編集委員 河野 浩之)

吉原 潤

1976 年生．1999 年筑波大学第三学郡情報学類卒業．同年筑波大学大学院理工学研究課理工学専攻修士課程入学，現在に至る．データベースシステム，分散システムに興味を持つ．

加藤 和彦（正会員）

1962 年生．1985 年筑波大学第三学群情報学類卒業，1987 年工学修士（筑波大学大学院工学研究科），1992 年博士（理学）（東京大学大学院理学系研究科）．1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，現在に至る．オペレーティングシステム，プログラミング言語システム，データベースシステム，分散システム，モバイルオブジェクト計算に興味を持つ．情報処理学会，電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE 各会員．