空間効率の高いインメモリー辞書のオンライン構築

小柳 光生　吉田 一星

Online Building of Space Efficient In-memory Dictionaries

Teruo Koyanagi and Issei Yoshida

大規模なテキスト・データに対する自然言語処理では，逐次抽出される大量の語彙を格納するために，辞書と呼ばれるマップ型のデータ構造が利用される．このデータ構造には，文字列をコンパクトに格納することで，より多くの語彙をメモリー内に保持し，計算資源あたりの容量を向上したいという要求がある．本論文では，入力データを分割して構築する既存の手法を導入して，極めて空間効率の高い LOUDS トライをオンライン構築する方法を提案する．既存の手法には，データ構造数の増加に伴う検索速度低下の問題と，マージ・コスト増加の問題があるが，提案手法では，ブルーム・フィルターにより検索速度の低下を防ぎ，仮想ノードを導入してマージ・コストを抑える．実データによる実験で約 2.5 倍の性能改善を達成し，提案手法の有効性を確認した．

Data structures comprising string-value mappings called dictionaries are employed to store the large amounts of vocabulary that is sampled from large amounts of textual data by using natural language processing. To improve cost-capacity performance, it is desired that such data structures be compact. In this paper, we employ an existing online building method in order to build an extremely space-efficient data structure, called LOUDS. Existing methods have two problems: search performance degrades in proportion with the increasing number of data structures to be searched, and the additional merge costs involved in reducing the number of data structures. We provide a novel method to remove this degradation by using Bloom filters, and also have introduced virtual nodes to reduce merge costs. An application benchmark based on real data shows that our method efficiently improves end-to-end throughput by 2.5 times.

Key Words & Phrases：辞書，連想配列，キー・バリュー・マッピング，簡潔データ構造，ブルーム・フィルター，オンライン索引構築，自然言語処理
dictionary, association list, key-value mapping, succinct data structure, bloom filter, online index building, natural language processing

1. はじめに

大規模なテキスト・データに対する自然言語処理では，生成される大量の語彙を辞書に保持する必要がある．語彙は，文書処理の過程で頻繁に検索，追加されるため，それらを格納する辞書には，高い検索性能と語彙を動的に追加できるオンライン構築の能力が求められる．自然言語処理に限らず，ユーザーや商品の管理など，文字列で表される大量の ID を扱うアプリケーションでは，動的更新可能な文字列をキーとするマップ型のデータ構造が利用される．以後このようなデータ構造を指して辞書と呼ぶ．

大量の文字列をキーとして扱う場合には，文字列をコンパクトにメモリーに格納したいという要求が生じる．より少ないメモリーで，より多くの文字列を格納できる空間効率の高い辞書を用いれば，より規模の大きい語彙を，より少ない計算資源で扱えるようになる．

簡潔データ構造［1］の一種である，LOUDS（Level-Order Unary Degree Sequence）［2］［11］を用いると，理論上，極小のビット列で順序木を表現できる．これを同じ順序木であるトライ木（Trie）［9］に用いることで，極めて空間効率の高い辞書を実現できる．本論文では，LOUDS を利用したトライ木を LOUDS トライと呼ぶ．

Double Array［10］などの動的なトライ木の実装では，メモリー・アドレスに対する参照を多用し，その書き換えによって自由に構造を変更できる．しかし，n 個のノードを指す参照の表現には，最低でも $\log n$ ビットを必要

提出日：2011年5月9日　再提出日：2011年12月12日

とし，全体のサイズは O(n log n) となる．これは，$2n$ + o(n) ビットで表される LOUDS と比較すると非常に大きい．語彙を格納するトライ木の表現に LOUDS を用いることで，Double Array と比較して，4 から 10 倍の空間効率を実現した例がある［12］．

これに対し，LOUDS は参照を用いず，ビット列によって表される静的なデータ構造である．新たなノードを追加するには，追加する場所以降のビットをすべて移動しなければならない．これは，オンライン構築のコストとしては，受け入れられない大きさである．

われわれは，既存の静的データ構造に対するオンライン構築法を LOUDS トライ構築に用い，LOUDS トライのオンライン構築を実現する．以後，これをオンライン LOUDS トライと呼ぶ．

しかし，オンライン LOUDS トライには，保持する LOUDS トライ数が増えることに起因する検索性能の低下の問題と，マージに伴う再構築のコストが追加される問題がある．本論文では，これらの問題を解決して，オンライン LOUDS トライを大規模な辞書として利用可能にするため，以下の 3 つの手法の導入を提案する：

1）検索性能の低下を防ぐため，ブルーム・フィルター［4］をオンライン構築で作られる各 LOUDS トライに付加する手法
2）ブルーム・フィルター構築コストを抑えるため，LOUDS トライとブルーム・フィルターを同時構築する手法
3）マージのコストを抑えるために，2 つのトライ木をあたかも 1 つのトライ木のように扱う,仮想ノードによるマージ手法．

本論文では，2 章に提案手法で用いる既存の手法とその課題を，3 章に提案手法 1）を含むオンライン LOUDS トライの概要を，4 章にその効率を改善する提案手法 2），3）を示す．5 章で提案手法の解析的な評価を，6 章に実データを使った実験による評価を示し，7 章に本論文の結論をまとめ，今後の課題を提起する．

2. 既存の手法と問題点

提案手法で用いる既存の手法について述べ，それらに含まれる問題のうち，本論文で解決する問題を明らかにする．

2.1 トライ木

トライ木は，ノードに文字列のアルファベットを割り当て，ルートから子へノードをたどることで，文字列を検索するデータ構造である．ルートからあるノードへのパスは，その配下の文字列の共通の接頭辞となる．“cattle” に対する “cat” のように，一方のパスに完全に含まれる文字列に値を割り当てるため，どの文字とも一致しない終端を表す記号（“#”）が用いられる．

トライ木の検索効率は，アルファベットからノードを選択するコストによるため，効率よく選択できるようにノードの配置を工夫する必要がある．本論文では，子ノードはアルファベット順にソートされているとする．

2.2 トライ木の幅優先走査

幅優先走査とは，木のノードを階層ごとに左から右に取り出す走査方法である．LOUDS は，トライ木の幅優先走査によって構築される．

トライ木 T のノード n には，対応する文字があり，次の 3 つの操作が定義される．firstChild(n) は n の子ノードのうち，最も小さい文字を持つノードを返す．sibling(n) は，n と共通の接頭辞を持ち，n よりも大きい文字を持つノードのうち，最も小さい文字を持つノードを返す．alphabet(n) は，n の文字を返す．n に子ノードがない時には firstChild(n)=ϕ とし，n が兄弟の中で最大の文字を持つ時には sibling(n)=ϕ とする．ルート･ノードを r とし，その文字は，空文字 (“”) とする．

これらの操作を用いて，幅優先走査を行うアルゴリズムは，p, n をノードとすると，キュー q (要素数 |q|) を使って，visit(p, n) を呼び出す Visitor パターンとして次のように書ける：

```
q ← {r}; visit(φ, r)
while |q| > 0
  p ← q.dequeue(); n ← firstChild(p)
  while n ≠ φ
    q.enqueue(n); visit(p, n) ; n ← sibling(n)
  endwhile
endwhile
```

2.3 LOUDS トライ

LOUDS トライは，LOUDS によるトライ部に，文字列の先頭の共通部分（共通接頭辞）を格納し，残りを配列 TAIL に格納する．また，値は整数の配列 VAL に格納される．トライ部は,LOUDS を表す BASE と,リーフを表す LEAF の 2 つのビット列と，文字配列 EDGE から構成される．トライ木の性質から，各キーには1つのリーフが対応し，リーフには TAIL と VAL の値が対応付けられる．

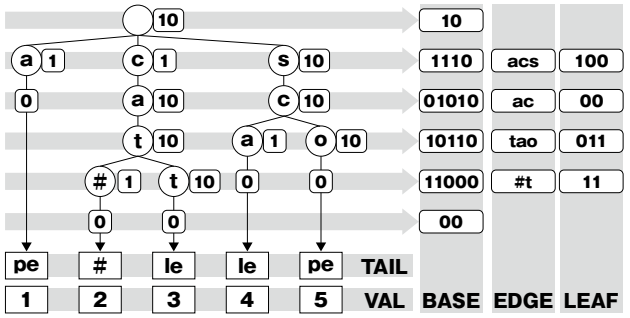

図1. LOUDSトライの構築例

図 1 に，“ape”，“cat”，“cattle”，“scale”，“scope” の 5 つのキーを格納する LOUDS トライの構築例を示す．LOUDS トライの構築には，visit(*p*, *n*) の中で次の操作を実行する．BASE に，*n* がルートの時に “01” を，また，*n* の子の数と同数の “1” に続く “0” を追加する．LEAF に,*n* がリーフの時に“1”を,それ以外の場合は“0” を追加する．EDGE には alphabet(*n*) を追加する．

LOUDS では，ノードの順位を幅優先に取り，“1” の個数を数えるインデックスを作ることで，ノードの親，子，兄弟の順位が $O(1)$ 時間で計算できる［2］［13］．EDGE には，ノードの順位に従って文字が格納されるため，ノードの順位から文字が取得できる．LEAF にも BASE と同様のインデックスを作ることで，LEAF とノードの順位から，リーフだけを数えた順位が計算できる．この順位に従って VAL に値を格納することで，特定されたリーフに対して値を返すことができる．

LOUDS は，ビット列の配置によって木構造を表すため，構造を変更するには全体を作り直す必要がある．オンラインでキーが追加されるたびに全体を作り直すことは非常に効率が悪いという課題がある（課題 1）．

2.4 オンライン構築法

静的なデータ構造をオンライン構築する仕組みが全文検索の索引で実用化されている［7］．全文検索の索引は，圧縮された文書 ID の列を含むため，一度構築されると更新できない．そのため，入力される文書を入力順に分割して索引を構築する手法が使われる．一定数の入力でバッファーが満たされるたびに差分の索引を構築し，作られた複数の索引を隠ぺいして，単一の索引であるかのように検索させる一方，マージ戦略（merge policy）と呼ばれるルールに従って，索引をマージし，その個数を制御する［3］［5］［6］．

しかし，効率の良いマージ戦略を用いても，データの増加に伴って検索すべき LOUDS トライの個数は増え，検索性能はそれに比例して低下する．検索性能の低下を緩和するには，マージの頻度を増やして，内部の LOUDS トライ数を少なく抑える方法が考えられるが，マージの頻度を増やすことで，マージに伴うデータ構造の再構築のコストが増大し，結局，全体の性能の低下につながるという課題がある（課題 2）．

2.5 ブルーム・フィルター

ブルーム・フィルター［4］は，キーに対して複数の独立したハッシュ値を取ってビット列にエンコードしたデータ構造であり，キー自身を保持せずに，そのキーが集合に含まれるかどうかを判定することができる．ただし，ハッシュ値は衝突の可能性を持つため，その応答には，存在しないキーに対して真を返す偽陽性（false positive）が含まれる．偽陽性確率はビット列のサイズとハッシュ数によって決まる［8］．

ブルーム･フィルターの例を図 2 右に示す．数字はビット番号，灰色が “1”，白が “0” を表す．左の表では，異なる素数 P_0, P_1, P_2 を用いて，各キーからハッシュ値を計算し，ブルーム・フィルターと照合している．この場合は，“scope” のハッシュ値のうち 2 つの示すビットが “0” であるため，ブルーム・フィルターの表すキー集合には “scope” は含まれない．ビットがすべて “1” である他のキーについては，それらがキー集合に含まれる可能性を示唆するに過ぎない．

Hash Codes

KEY	P_0	P_1	P_2
ape	✓14	✓22	✓14
cattle	✓ 3	✓23	✓17
scale	✓22	✓18	✓ 4
scope	× 8	×12	✓14

Bloom Filter

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

図2. ブルーム･フィルターの例

文字列 *s* のハッシュ値 *h* は，*P* を文字の種類の数に近い素数（例えば 131），s_i を *s* の *i* 番目の文字とすると，以下のように計算できる：

$h = 0;$
for $i = 0$ **to** $i < |s|$ **do**
　　$h = h * P + s_i$
endfor

この計算方法では，文字列を 1 つずつ取り出してハッシュを計算する必要があるため，LOUDS トライからブルーム・フィルターを作成すると大きなコストがかかるという課題がある（課題 3）．

2.6. 中間バッファによるマージ

2 つの LOUDS トライ T_1, T_2 をマージするには，中間バッファとして，空の動的トライ B_m を用意し，以下の手順で新しい LOUDS トライを作る.

1）T_1, T_2 からキー集合 K_1, K_2 を取り出す
2）B_m に K_1, K_2 を入力してトライ木を作る
3）B_m を走査して LOUDS トライを構築する

各トライ木からキーを取り出すには，トライ木を深さ優先走査しながら，出現する文字をつないで文字列を再構築すればよい.

この方法は，中間バッファ B_m にキーを入力した後，3）の処理が，通常の LOUDS トライ構築と同じコードを共有できる点が良い. しかし，中間バッファの分メモリーを多く使用し，かつ，走査回数が多いため，処理コストも高くなってしまう課題がある（課題 4）.

3. オンライン LOUDS トライ

前節で述べた課題 1 に対して，2.4 節のオンライン構築法を適用し，また，課題 2 に対して，LOUDS トライと同じキー集合を持つブルーム・フィルターを使って，検索時に，検索キーが含まれない LOUDS トライを除外することで検索効率を向上させる手法を提案する. 図 3 にその提案手法の概要を示す.

3.1 入力と LOUDS トライの構築

提案手法で実現する辞書では,入力（put）したキーと値の組み合わせが，即座に検索（get）に反映される必要がある. これを実現するため，検索可能な動的トライによるバッファを用いる.

入力されたキーと値の組は，バッファ B_i に格納される. 一定数のキー集合 K_i がバッファ B_i に格納されると，新たな空のバッファ B_{i+1}（以下，添え字の大きいものが新しい）が用意され，構築の間に入力される新たなキーは B_{i+1} に追加される. 続いて，B_i に対する幅優先走査によって LOUDS トライ L_i が構築される. また，K_i を網羅するブルーム・フィルター F_i が生成される. 検索時にキーが L_i に含まれるか，F_i を使ってチェックできるように，組 $<L_i, F_i>$ がリスト S に i 番目の要素として追加される. $<L_i, F_i>$ がリスト S へ追加された後，B_i は破棄される.

3.2 LOUDS トライのマージ

S には構築時間順に $<L_i, F_i>$ が格納される. マージ戦略の条件が満たされると，マージ戦略によって選択された，隣り合う組の集合 $\boldsymbol{L} = \{<L_i, F_i>, ..., <L_j, F_j>\}$ がマージされる. $\boldsymbol{L}$ に同じキーが含まれる場合には，最新の値を保持するため，最大の添え字を持つ LOUDS トライに含まれる値が保持される. また，構築時と同様にブルーム・フィルターが作られ，新たな組 $< L_{i,j}, F_{i,j} >$ を生成して $\boldsymbol{L}$ を置き換える.

マージする組を選択する方法にはさまざまな方法が考えられるが,ここでは,Log マージ戦略［3］を用いる. マージ係数 m の Log マージ戦略では，i 番目のバッファが満たされた時，任意の整数 l (l>0) に対して，$i = m^l$ ならば，バッファと最新の $l(m-1)$ 個のトライ木をマージする.

3.3 キーによる検索

検索時は，まず，バッファ B_{i+1}, B_i の順でキーを検索する. 見つからなければ，添字 j ($j<i$) の降順に LOUDS L_j を検索する. 各 L_j の検索に先立って，対応するブルーム・フィルター F_j がチェックされ，結果が偽ならば，L_j の検索はスキップされる. 結果が真であれば,キーが含まれる可能性があるので,L_j の検索を行う.

4. 提案手法のアルゴリズム

3 章のオンライン構築を導入することで，課題 3 で示したブルーム・フィルターの構築コストと課題 4 で示したマージのコストが新たに発生する. これらを軽減するため，2 つの手法を提案する. LOUDS トライとブルーム・フィルターの同時構築により，ブルーム・フィルターの生成コストを大きく削減する. また，仮想ノードによって 2 つのトライ木を 1 つのトライ木と同様に扱えるようにし，中間バッファを用いずにトライ木をマージできるようにする.

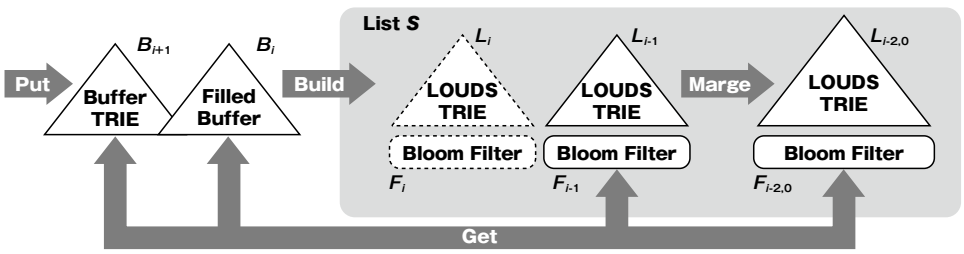

図3. オンラインLOUDSトライ

4.1 ブルーム・フィルターの同時構築

LOUDS 構築時の幅優先走査にハッシュ値の計算を組み込むことで，ブルーム・

フィルターの生成のために必要な走査を省く．これを行うために，ブルーム・フィルターに，通常の文字列 s を入力とする put(String s) とは別に，外部で計算した文字列のハッシュ値 h を直接入力する put(int h) を追加する．

初期状態のブルーム・フィルター F は，0 でクリアされたビット列からなる．ノード n には，計算途中のハッシュ値が対応付けられる．この中間ハッシュ値を，$n.h$ とする．幅優先走査にて，トライ木に含まれるすべてのキーのハッシュ値を計算し，ブルーム・フィルターを構築するアルゴリズムは次のように書ける：1）ルート・ノード r の中間ハッシュ値 $r.h$ を 0 に初期化する．2）幅優先走査の visit(p, n) で以下の計算を行う：$n.h = p.h * P$ + alphabet(n)．n がリーフなら，F.put($n.h$) を呼び出して，F にハッシュ値 $n.h$ を書き込む．

実際のブルーム・フィルターでは，1 つのキーに対して複数のハッシュ値が計算される．上記アルゴリズムで，複数のハッシュ関数を実現するには，中間ハッシュ値を配列にし，異なる P を用いて上記の計算を行えばよい．

文字列のハッシュ値にはさまざまな計算方法があるが，文字列の先頭の文字からインクリメンタルにハッシュ値を計算する方法であれば，本手法と同様に幅優先走査で全キーのハッシュ値を計算できる．

4.2 仮想ノードによるマージ

トライ木では，各ノードの持つ子ノードの列はアルファベット順にソートされていなければならない．2 つのトライ木 T_1，T_2 の共通接頭辞を持つノード $n_1 \in T_1$，$n_2 \in T_2$ をマージしたノードを $M(n_1, n_2)$ とする時，アルファベット順の条件を満たすには，n_1 の子の列と，n_2 の子の列をマージ･ソートして，$M(n_1, n_2)$ の子の列とすればよい．これを，ルートを含むすべての共通接頭辞を持つ 2 つのノードについて行うと，T_1，T_2 をマージした木 T_m が得られる．

本論文では，M を仮想ノードと呼び，上記の振る舞いを M の firstChild，sibling，alphabet の 3 つの操作に組み込むことで，中間のトライ木を作らずとも，LOUDS トライを構築する幅優先走査ができるようにする．仮想ノード M に対する 3 つの操作を n_1，n_2 の操作によって**表 1** のように定義する．なお，便宜的に任意のアルファベット α に対して α<alphabet(ϕ)，また，終端文字 # に対して # <α であるとする．

T_1，T_2 のルートを r_1，r_2 とすると，$M(r_1, r_2)$ によって，T_1，T_2 のマージ木 T_m のルート･ノードが得られる．T_m に対して 2.3 節の LOUDS トライ構築法と 4.1 節のブルーム・フィルターの同時構築を実行することで，T_m の LOUDS トライと対応するブルーム・フィルターが構築できる．

表1.仮想ノードの操作

firstChild(M(n_1, n_2)) →
M(firstChild(n_1), firstChild(n_2)) **when** alphabet(n_1)=alphabet(n_2)
(firstChild(n_1) **when** alphabet(n_1)<alphabet(n_2)
(firstChild(n_2) **when** alphabet(n_1)>alphabet(n_2)
sibling(M(n_1, n_2)) →
M(sibling(n_1), sibling(n_2)) **when** alphabet(n_1)=alphabet(n_2)
M(sibling(n_1), n_2) **when** alphabet(n_1)<alphabet(n_2)
M(n_1, sibling(n_2)) **when** alphabet(n_1)>alphabet(n_2)
alphabet(*M*(n_1,n_2)) → min(alphabet(n_1), alphabet(n_2))

5. 解析的な評価

提案手法で用いるブルーム・フィルターのサイズと，その検索性能への寄与について，解析的な手法で評価を行う．

5.1 ブルーム・フィルターのサイズ

ブルーム･フィルターは，そのサイズと精度の間にトレードオフの関係がある．キーの数を N，ブルーム・フィルターのビット列のサイズを m，使用するハッシュの個数を k とすると，m を定めた時に，ブルーム・フィルターの偽陽性確率を最小にする k を求めることができる．ハッシュ関数の衝突が独立であると仮定する時，偽陽性確率はおよそ $(1-e^{-kN/m})^k$ で表され，これを最小化するハッシュ個数は $k = (m/N) \ln 2$ である［11］．これを基に m を定めると，$m = kN/\ln 2 = 1.44kN$ が得られる．このときの偽陽性確率は $(1/2)^k$ である．

トライ木のノード数を n とすると，ブルーム・フィルターのサイズは，$1.44kN$ となり，$N < n$ から，最悪でもノードあたり $1.44k$ ビットの消費に抑えられる．ブルーム・フィルターを含まない LOUDS トライのサイズは，$O(n)$（ノードあたり BASE 2bit，LEAF 1bit，EDGE 8bit，VAL 32bit）であるため，サイズのオーダーは，ブルーム・フィルターを付加した場合でも変わらない．

一方，動的トライのサイズは，参照を用いてノードを表すことから，$O(n \log n)$ である．これは，$O(n)$ でバウンドされるブルーム・フィルター付きの LOUDS トライと比較してかなり大きい．

5.2 ブルーム・フィルターによる検索性能の改善

ブルーム・フィルターによる検索速度の改善度合いを

知るために，ブルーム・フィルターがない場合とある場合のそれぞれについて，検索キーがオンラインLOUDSトライに含まれる場合と含まれない場合に分けて計算コストを求める．LOUDSトライの個数をM，検索文字列をqとする．また，どのLOUDSトライも，重複するキーを含んでいないとする．

まず，ブルーム・フィルターがない場合，qがいずれかのキー集合に含まれていて，どの集合に含まれる確率も等しいとすると，キーが見つかるまでに検索するLOUDSトライの数の期待値は$(M+1)/2$であり，LOUDSトライを検索するコストをC_Lとすると，検索コストの期待値は$C_L(M+1)/2$である．qがどのキー集合にも含まれていない場合には，常にすべてのLOUDSトライが検索され，検索コストの期待値はC_LMである．

次に，ハッシュ個数kのブルーム・フィルター付きLOUDSトライの検索コストを求める．qがいずれかのキー集合に含まれるならば，最後に検索されるブルーム・フィルターは必ず真を返す．その前に検索されるブルーム・フィルターの偽陽性確率は$(1/2)^k$であり，この確率でLOUDSトライが検索される．従って，検索されるLOUDSトライ数の期待値は$((M+1)/2-1)(1/2)^k+1$となり，コストの期待値は$C_L((M+1)/2-1)(1/2)^k+C_L$となる．なお，ブルーム・フィルターを検索するコストC_Fは，LOUDSトライの検索コストC_Lに対して小さいため，ここでは無視する．

qがどのキー集合にも含まれていない場合，検索されるLOUDSトライの数の期待値は$M(1/2)^k$である．偽陽性確率が小さいと，この値が非常に小さいため，ここでは，ブルーム・フィルターを検索するコストCFを加味する．すべてのブルーム・フィルターが常に検索され，検索コストの期待値は$C_LM(1/2)^k+C_FM$になる．

図4は，計測して求めた$C_L(33.3\mu s)$と$C_F(1.25\mu s)$を当てはめて，検索速度を計算した結果である．Existent, Nonexistentはそれぞれブルーム･フィルターが無く，キーが集合に含まれる場合と含まれない場合，Existent/BFとNonexistent/BFはブルーム・フィルターがある場合である．LOUDSトライは全入力（各試行で一定）を等分した数の入力を保持し，バッファは空であると仮定している．横軸は，検索対象となるLOUDSトライの個数を示し，バッファは個数に含まれない．

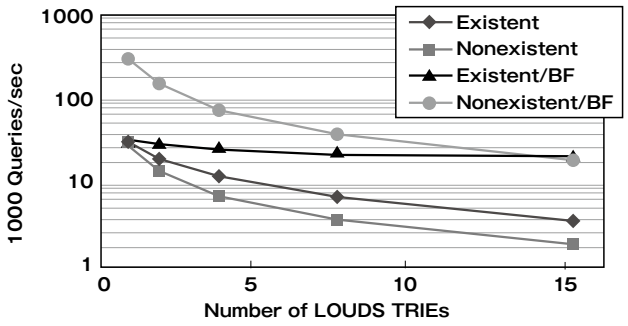

図4. LOUDSの個数に対する検索速度の変化

この結果から，検索キーがいずれかのLOUDSトライに含まれる時には，ブルーム・フィルターによって，LOUDSトライ数の増加に伴う検索速度の低下が効果的に防がれることが分かる．また，検索キーがどのLOUDSにも含まれない場合には，非常に高い性能を示すことから，新規キーによる検索が多く発生する場合には，全体の性能を高める可能性を示唆している．

6. 実験による評価

提案手法の効果を評価するために，実データを用いて実験を行い，ブルーム・フィルターを生成する空間的，時間的コストと，それに対する性能の改善の程度を調べる．実験用のデータには，NHTSA［14］が提供する自動車の不具合報告データベースから自然言語処理によって抽出された，重複を含むキーワード文字列約2億4千万件のストリームを用いる．このストリームには，約650万件のユニークなキーワード文字列が含まれる．各キーワード文字列をキーとして格納し，それぞれにユニークな整数（32bit）を値として割り当て，キーで整数値を検索する．

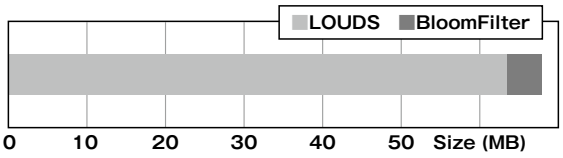

図5. LOUDSトライに占めるブルーム･フィルターのサイズ

図5は，LOUDSトライとブルーム・フィルターを650万件のキーから構築した場合のサイズを示すグラフである．LOUDSトライは1キーワードあたり9.4バイトを占める．この場合では，ブルーム・フィルターには1キーワードあたり3bit（2つのハッシュ関数）を与え，LOUDSトライのサイズに対しては,約6.8%のメモリー消費となる．

図6は，ブルーム・フィルターをLOUDSトライの構築後に生成する場合と，LOUDSトライの構築と同時に生成する場合を比較するグラフである．LOUDSトライは，約650万件のキーを含む動的トライから構築する．Build without BFはLOUDSトライのみの構築時間に，ブルーム・フィルターを単独で構築するのにかかる時

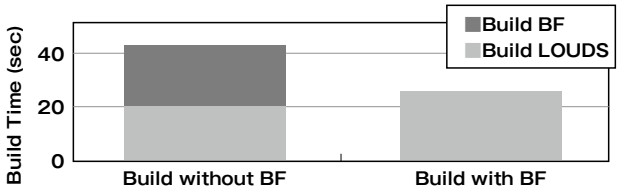

図6. LOUDSトライとブルーム･フィルターの構築時間

間を積み重ねた. Build with BF は LOUDS トライと同時にブルーム・フィルターを構築した場合の時間であり, ブルーム・フィルターの構築時間も含む.

LOUDS トライを構築する過程でブルーム・フィルターを同時に作る場合には, ブルーム・フィルターの構築時間は,LOUDS トライの構築時間の約 24% になる. 一方, 両者を別々に生成した場合, ブルーム・フィルターの生成時間は LOUDS トライ単体の構築時間の約 113% になる . この差は, 1 回の幅優先走査で複数の構築処理を同時に行う本手法の有効性を示している .

図 7 は, それぞれ約 320 万件のキーを含む 2 つの LOUDS トライを 1 つにマージする際の時間を測定した実験の結果である. Intermediate buffer は,中間バッファに動的トライを用いる場合, Virtual node は, 仮想ノードを用いたマージの場合を示している. この場合,仮想ノードによって, 性能が約 25%改善されている.

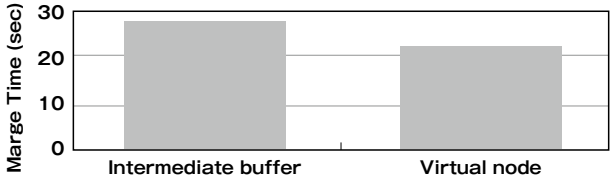

図7. 中間バッファと仮想ノードの性能差

図 8 は, 提案手法によってオンライン構築した LOUDS トライに対して, すでに含まれるキーと, 含まれないキーをそれぞれ検索した場合のスループットを比較するグラフである. キーの有無を分けるために, 650 万件のキーのうちの半分をストアに入力して実験を行った. すでに存在するキーの検索では, いずれも LOUDS トライに対する検索が発生するため, スループットにはほとんど差がない. 一方,キーが存在しない時,ブルーム･フィルターがない場合には, すべての LOUDS トライを検索する必要があるのに対して, ブルーム・フィルターがある場合には, LOUDS トライの検索がスキップされるため, スループットの差が顕著に現れる.

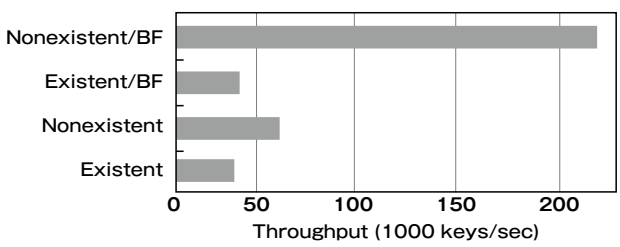

図8. ブルーム･フィルターによる検索性能の向上

図 9 は, 重複を含む約 2 億 4 千万件の実験データ全体を使って, 実際の辞書構築の過程を再現した実験の結果である. この過程では, 入力されたキーワードを辞書から検索し, そのキーワードが辞書に含まれていなければ, 新しい語彙として辞書に追加する. 図の Without BF はブルーム・フィルターなしの場合, With BF はブルーム・フィルターありの場合のスループットである. 新規キーは全入力の約 2.56% を占め, トランザクションの 97.44% が検索のみを含み, 新規キーの 2.56% が辞書に含まれないキーの検索と, そのキーの書き込みを含んでいる .

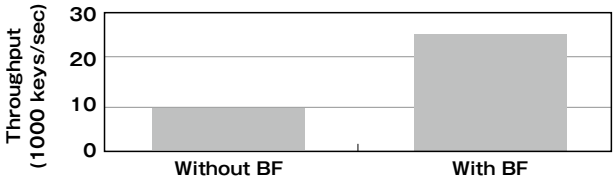

図9. 実際の辞書構築による性能比較

この結果から, 本手法が, オンライン LOUDS トライの性能改善に大きな効果があることが分かる. 検索の性能を大幅に上げる新しいキーの割合は約 2.5% と決して高くないが, 構築時に大きなコストを上乗せすることなく, LOUDS トライに対応するブルーム・フィルターを作成し, 約 2.5 倍の改善を得た.

7. まとめ

本論文では, オンラインでデータの追加が可能な, 文字列をキーとする空間効率の高い辞書の実装方法を提案した. 既存のオンライン構築法を LOUDS トライの構築に応用し, LOUDS トライの構築・マージと同時に, そのキー集合に対応するブルーム・フィルターを生成することで, 構築時のオーバーヘッドを抑えながら, 検索性能を高めた. この提案手法は, 新しいキーを追加しながら利用する場合に効果的であり, 2.5% 程度の新規キーを含む入力に対して, 約 2.5 倍の性能改善が認められた.

本手法と同様にして, アルゴリズムを幅優先走査に

設計しなおすことで，ほかにも有益な操作をLOUDSトライの構築と同時に実行することができる．紙面の制約で省いたが，最小接頭辞トライを構築する際のTailの判別なども，幾つかかの操作を追加することで，同様に設計し，効率よく処理することができる．

本手法によって，LOUDSトライがオンラインで利用できるようになったが，テキスト・データの解析に利用するには，まだ最適化の余地を残している．今後は，これを応用した,より効果的なデータストアを開発し,発表したい．

謝辞

本研究を進めるにあたり，多くの方に協力していただきました．特に，本研究のきっかけを作り，素晴らしい動的トライの実装を提供してくれた，海野裕也さんと，研究を時間的にも資金的にも支援してくれた所属長の濱田誠司さんには最大の感謝を申し上げます．

参考文献

［1］ Guy Joseph Jacobson: "Succinct static data structures," PhD thesis,Carnegie Mellon University,Pittsburgh,PA,USA, AAI8918056,（1988）.
［2］ Guy Joseph Jacobson: "Space-efficient static trees and graphs," In Proceedings of the 30th Annual Symposium on Foundations of Computer Science（SFCS '89）,IEEE Computer Society, Washington,DC,USA,pp.549-554,（1989）.
［3］ Stefan Buttcher,and Charles L.A.Clarke: "Indexing time vs. query time: trade-offs in dynamic information retrieval systems," In Proceedings of the 14th ACM international conference on Information and knowledge management, CIKM'05,pp.317-318,New York,NY,USA,（2005）.
［4］ Burton Howard Bloom: "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM,Vol.13,pp. 422-426,（July 1970）.
［5］ Nicholas Lester,Alistair Moffat,and Justin Zobel: "Fast on-line index construction by geometric partitioning," In Proceedings of the 14th ACM international conference on Information and knowledge management,CIKM'05,pp.776-783,New York,NY, USA,（2005）.
［6］ Ruijie Guo,Xueqi Cheng,Hongbo Xu,and Bin Wang: "Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree," In Proceedings of the sixteenth ACM conference on Conference on information and knowledge management,CIKM'07,pp.751-760,New York,NY,USA,（2007）.
［7］ The Apache Software Foundation: "Apache Lucene," http://lucene.apache.org/
［8］ Li Fan,Pei Cao,Jussara Almeida,and Andrei Z.Broder: "Summary cache: a scalable widearea web cache sharing protocol," IEEE/ACM Transactions on Networking（TON）, Vol.8,pp.281-293,（June 2000）.
［9］ Edward Fredkin: "Trie memory," Communications of the ACM, Vol.3,pp.490-499,（September 1960）.
［10］ Jun-ichi Aoe: "An Efficient Digital Search Algorithm by Using a Double-Array Structure," IEEE Transactions on Software Engineering,Vol.15,Issue 9,pp.1066-1077,（September 1989）.
［11］ O'Neil Delpratt,Naila Rahman and Rajeev Raman: "Engineering the LOUDS Succinct Tree Representation," Lecture Notes in Computer Science,Volume 4007/2006,pp.134-145,（2006）.
［12］ 岡野原大輔: "大規模コーパスを扱うためのツール群," NLP若手の会第3回シンポジウム,言語処理学会,（September 2008）.
［13］ Dong Kyue Kim,Joong Chae Na,Ji Eun Kim,and Kunsoo Park: "Efficient implementation of rank and select functions for succinct representation," In Experimental and Efficient Algorithms Lecture Notes in Computer Science,Vol. 3503/2005,pp. 125-143,（2005）.
［14］ National Highway Traffic Safety Administration: http://www.nhtsa.gov/

日本アイ・ビー・エム株式会社
大和ソフトウェア開発研究所
ソフトウェアエンジニア

小柳 光生 Teruo Koyanagi

［プロフィール］
1999年，日本IBM入社．東京基礎研究所配属．アプリケーションサーバーやデータベースの高速化に関する研究に従事．2008年より同社大和ソフトウェア開発研究所．IBM Content Analyticsを中心にディスカバリー製品の開発を担当．製品の性能解析・高速化に従事．

日本アイ・ビー・エム株式会社
東京基礎研究所
主任研究員

吉田 一星 Issei Yoshida

［プロフィール］
2001年，日本IBM入社．2003年より同社東京基礎研究所．主任研究員．テキスト・マイニング・システム IBM TAKMIの設計開発を担当．データベース・検索・並列処理に興味を持ち，計算量の解析・実装の高速化・テキスト分析への応用に従事．