情報処理学会論文誌　データベース　Vol. 4　No. 4　1–10 (Dec. 2011)

簡潔データ構造のオンライン構築とブルームフィルタによる検索性能の向上

小柳光生[†1,†2]　吉田一星[†3]
海野裕也[†4]　新城靖[†1]

簡潔データ構造は空間効率のきわめて高いデータ構造であり，その一種である LOUDS（Level-Order Unary Degree Sequence）を用いてトライ木を作れば，大量の文字列を少ない容量で格納できる．インクリメンタルにデータを追加しながら LOUDS を構築するには，差分を保持する LOUDS を複数作成して，それらを定期的にマージする方法がある．しかし，この方法では，検索すべき LOUDS の数が増え，検索速度が低下するという問題がある．本論文では，この問題を解決するために，各 LOUDS にブルームフィルタを設置する手法を提案する．提案手法では，ブルームフィルタによって検索の不要な LOUDS をスキップし，検索コストを削減する．実験により，ブルームフィルタの精度が全体の性能に与える影響を明らかにする．650 万件の語彙を含む約 2.4 億件の単語データから辞書を作成する実験を行い，提案方式の有効性を確認した．

Online Building of Succinct Data Structures and Search Performance Improvement by Bloom Filters

Teruo Koyanagi,[†1,†2] Issei Yoshida,[†3] Yuya Unno[†4]
and Yasushi Shinjo[†1]

Succinct data structures are extremely space efficient data representations. LOUDS (Level-Order Unary Degree Sequence) is a succinct data structure for trees which can be used as a TRIE to store a large number of strings. To add data incrementally into LOUDS, we can use a method to create multiple LOUDS as deltas and to merge them periodically. In this approach, there is a problem that the search performance degrades proportional to the number of LOUDS. To solve this problem, we propose a method to set bloom filters with created LOUDS, respectively. The bloom filters avoid unnecessary searches in many cases that the search key is not included in the key set of LOUDS. The efficiency of our method is confirmed by the experiment using 240 million word stream that consists of 6.5 million unique keywords.

1. はじめに

テキストマイニングやゲノム解析など，文字列で表される大量のデータを扱う分野は多い．大量のデータを扱う分野では，高い空間効率でデータを保持し，かつ，効率良く検索を行う方法が求められる．

空間効率の高いデータ構造は，データ全体を計算対象として 1 度に構築されることが多いが，インクリメンタルにデータを加えて計算結果を得ることが求められる場合がある．本論文では，インクリメンタルにデータを入力してデータ構造を構築する「オンライン構築」について議論する．

簡潔データ構造[12] は，理論的に最小限，あるいはそれに近いデータサイズで実現するデータ構造の総称であり，LOUDS（Level-Order Unary Degree Sequence）[13] は，順序木の枝を階層ごとにビット列にエンコードした簡潔データ構造である．これをトライ木[10] の実装（LOUDS トライ）として利用すると，トライ木自身も接頭辞を共有する比較的空間効率の高い構造であることも助けて，非常にコンパクトにキー文字列を格納することができる．

LOUDS は他の簡潔データ構造と同様，データ全体を対象として 1 度に構築されるデータ構造であり，1 度構築されたデータ構造に効率良く新たな文字列を追加することができない．このようなデータ構造をオンライン構築する場合には，一時的に入力データ（ここではキー文字列と値の組）をバッファに蓄え，一定数のデータが追加されるたびに，バッファされた内容からデータ構造を構築する．

この方法を検索の側から見ると，構築のたびに検索すべきデータ構造の個数が増えるため，検索時間は生成されたデータ構造の個数に比例することになり，その分検索性能が低下する．データ構造の個数を抑えるために，複数あるデータ構造を一定の戦略でマージする手法が一般的だが，構築と検索を同じタイムフレームで行うオンライン構築では，頻繁にマージを繰り返すと，そのコストが全体の性能にも影響を与える．

†1 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba
†2 日本アイ・ビー・エム大和ソフトウェア研究所
IBM Yamato Software Laboratory
†3 日本アイ・ビー・エム東京基礎研究所
IBM Research - Tokyo
†4 株式会社 Preferred Infrastructure
Preferred Infrastructure, Inc.

我々は，検索性能低下の問題を解決するために，構築される LOUDS のそれぞれに，そのキー集合に対応するブルームフィルタ[4] を付加する方法を提案する．ブルームフィルタによって，検索文字列が含まれない LOUDS を検索対象から排除し，LOUDS の個数が増えることによる検索性能の低下を防ぐ．

計算モデルによる評価と実験の両方から，LOUDS トライの個数が増えることによる検索性能の低下が，本手法によって効果的に抑えられることを示す．その際のブルームフィルタによるサイズの増大は，入力されるキー数に比例し，高い空間効率を持つ簡潔データ構造に対しては小さくないものの，LOUDS トライの全体のサイズに対しては 12%程度である．

本論文の構成を以下に示す．まず，2 章では，関連する研究を示す．3 章では，LOUDS トライのオンライン構築に求められる要件と提案手法の概要を示す．4 章では，計算モデルにより，ブルームフィルタのサイズと効果を評価する．5 章では，実データを使った実験により，本手法が検索性能を改善する効果を検証する．6 章では，本論文の結論をまとめ，今後の課題を提起する．

2. 関連する研究

ブルームフィルタ[4] は，キーに対して，複数の独立したハッシュ値をとってビット列にエンコードしたデータ構造を持ち，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の真偽を判定することができる．ハッシュ値は衝突の可能性を持つため，ブルームフィルタの応答には，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．計算するハッシュ値の最適な個数は，用意するビット列のサイズによって，計算で求めることができる[9]．以降では，ブルームフィルタのハッシュ値の個数をハッシュ多重度と呼ぶ．ブルームフィルタは，コンパクトで，検索も高速であり，目的の要素の有無を実際に調べる前にふるいにかける目的でよく利用される．本論文では，ブルームフィルタを使って，検索時に，検索キーが含まれない LOUDS トライを除外して検索を行う．

キー文字列の格納に利用するトライ木[10] は，古くから知られているデータ構造である．トライ木の枝は文字列中のアルファベットを表し，ルートからあるノードまでのパスがその配下の文字列の共通の接頭辞となる．各ノードの選択が O(1) の計算量で可能であれば，キーの検索がハッシュテーブルなどと同様に高速である．Double Array[1] のように，コンパクトで検索性能も高い実装が知られている．LOUDS のような，簡潔データ構造を使って木構造を表現することで，さらに空間効率の高い実装が実現できる．

簡潔データ構造は，理論的に最小限，あるいはそれに近いデータサイズで実現するデータ構造の総称である．たとえば，順序木の簡潔データ構造は n を木のノード数とすると，たかだか $2n + \mathrm{o}(n)$ 個のビット列で表現される．また，最小限の表現のまま，データに対する操作を効率良く実現することができる．順序木の簡潔データ構造には，LOUDS のほか，BP（Balanced Parenthesis）[18] や DFUDS（Depth-First Unary Degree Sequence）[3] などが知られている．これまでに，順序木のほかに，集合，グラフなど，様々なデータの簡潔データ構造が提案され，より豊富で，より効率の良い操作の実現方法や，応用が提案されている[22),23]．本論文では，実装が簡単でトライ木の表現に十分な機能を備えた LOUDS を用いる．

LOUDS は，順序木の枝を階層ごとにビット列にエンコードした簡潔データ構造である．他の簡潔データ構造と同様に，LOUDS も高い空間効率を実現しつつ，効率の良い操作を提供する[8),15]．自然言語処理の用途で，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[25]．

文字列をキーとし，動的に更新可能なマップは，トライ木やハッシュテーブル以外にも様々な実現手段がある．特に，近年，コンパクトさと検索性能を兼ね備えた新しいデータ構造の提案がなされている．DAWG（Directed Acyclic Word Graph）[6] は，トライ木を決定性有限オートマトン（DFA）の一種として見て，そのノード数を最小化したデータ構造であり，オンライン構築の手法も提案されている[11]．動的簡潔順序木[21] は，セグメントに分割された括弧列による木構造の簡潔表現（BP）と，BP のインデクスとして機能する区間最大最小木を組み合わせることで，多くの操作を $\mathrm{O}(\log n / \log\log n)$ 時間で実行でき，かつ動的更新も可能にしたデータ構造である．本論文の手法は，入力データを分割することで，静的なデータ構造を逐次的に構築し，追加の手段（ブルームフィルタ）によって，検索効率を担保する点で，他のアプローチとは異なっている．

本論文では，文字列を検索のキーとするマップを高い空間効率で実現する方法を提案する．この手法は，キー・バリュー型のデータストアに応用することができる．キー・バリューを扱う分散データストアには多くの実装が存在し[2),5),7),16),17),20]，高いスケーラビリティにより，数百台規模のクラスタの上で運用されているものもある．本手法を分散データストアに応用して，データ規模をスケールさせる場合には，Consistent Hashing[14] などのデータ分割の手法と組み合わせる必要がある．

3. オンライン LOUDS トライ構築

本章では，LOUDS トライをオンライン構築する場合に求められる要件と，それを効果的

に実現する提案方法について述べる．

3.1 要　　件

本論文で扱う問題は，文字列のキーと整数の値を対応付けるマップのオンライン構築である．マップの操作には，キーと値を対応付けるデータ入力（put）と，キーを指定して対応する値を取り出す検索（get）がある．できるだけ多くのデータを保持するために，少ないメモリでマップを実現しつつ，キーと値を対応付けるデータは，入力後ただちに検索結果に反映させる必要がある．また，検索もできるだけ高速に行う．入力要求と，検索要求は同じタイムフレームで発生するため，入力・検索双方のコストが全体の性能に影響を与える．

この問題は，全文検索エンジンの索引を，メインメモリ上でインクリメンタルに更新する場合を想定している．入力されるデータには一定の割合でユニークキーが含まれ，保持すべきデータ量は問題サイズに比例して大きくなる．Consistent Hashing などの分散技術と組み合わせることで，複数のマシンに分散してデータを保持するため，できるだけ空間効率の高いデータ構造を使うことが求められる．LOUDS と比較して，多くのメモリを必要とするハッシュテーブルや，動的トライ木を用いれば，より高速なマップを実現することができるが，必要になるメモリ量から換算されるマシン台数は数倍に増える．

値にユニークな整数を格納し，整数に任意の型のオブジェクトを対応付ければ，値として任意の型を扱うことができる．値の圧縮も重要な課題だが，本論文の範囲を超えるため，ここでは，値は 32 ビットの整数とし，圧縮は行わない．キーの削除には対応しないが，特定の数値を削除済みのラベルとすることで，削除を扱うことができる．

3.2 LOUDS トライ

LOUDS トライは，LOUDS を使った木構造部に最小共通接頭辞を保持し，残りの文字列をハッシュテーブルを使った TAIL と呼ぶデータ構造に格納する．木構造部は，LOUDS を格納するビット列 BASE と，リーフを示すビット列 LEAF, 各ノードの文字を格納した EDGE, 値を格納する整数の配列 VAL の 4 つのデータから構成される．トライ木の性質から，キー文字列には 1 つのリーフが対応し，値はリーフに対応して格納される．

LOUDS は，木の構造を幅優先順（level order）にビット列にエンコードしたデータ構造である．ノードは，子と同数の “1” に続く 1 つの “0” によって表され，リーフは 1 つの “0” で表される．操作の整合性のために，まず “10” が先頭に付加され，続いて，ルートから幅優先順に各ノードがビット列 BASE に格納される．LEAF は，リーフを “1”，それ以外を “0” とするビット列であり，同様に各ノードが幅優先順に格納される．

文字列に対応する値を検索するには，ルートからリーフに向かって，文字列の各文字に

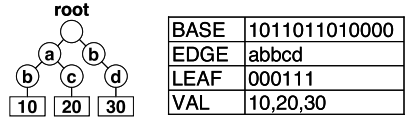

図 1　LOUDS トライ（木構造部）の構築例
Fig. 1　Example of constructs in LOUDS TRIE.

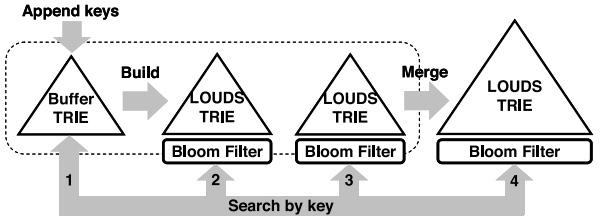

図 2　ブルームフィルタ付きオンライン LOUDS トライ構築法
Fig. 2　Online LOUDS building system with Bloom filters.

対応するノードを順に取り出す操作が必要である．LOUDS では，ノードの順位を幅優先にとることで，ノードの親，子，兄弟の順位が計算できる[13]．これらの計算は，ビット列 BASE をブロック単位に分割し，2 段階で “1” の個数を保持するインデクス[15] を作ることで，O(1) の時間で実現される．また，EDGE には，ノードの順位に従って文字を格納し，ノードの順位から文字を取得できるようにする．LEAF にも BASE と同様のインデクスを作ることで，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，対応するリーフが特定されたときにその値を返すことができる．

図 1 に，キー “ab”，“ac”，“bd” を含む LOUDS トライの例を示す．

3.3 バッファリングと構築

本論文で提案するブルームフィルタ付きオンライン LOUDS トライ構築法の概要を**図 2** に示す．

以降では，入力される一定数のキーごとにウィンドウを設定し，添え字 i でウィンドウを表す．添え字は大きいほど新しく，同じ添え字を持つデータ構造は同じウィンドウに対応して構築されたものである．K_i をそのウィンドウ i に含まれるキー集合，B_i をバッファ，L_i を LOUDS トライ，そして F_i をブルームフィルタとする．S は組 $\langle L_i, F_i \rangle$ を i の降順に保

持するリストとする．組はマージされることで，i から j までの連続する複数のウィンドウに対応する場合があるが，この場合は，$\langle L_{ij}, F_{ij}\rangle$ のように，複数の添え字によってその範囲を表す．

キーと値の組は，入力されるといったんバッファB_i に格納される．このバッファは Double Array などのアルゴリズムを用いて実装される動的なトライ木であり，キーに対する値の入出力のほか，キーの検索も提供されるため，入力したキーと値の組をただちに検索に反映させることができる．

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} 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．このとき，同時に K_i を網羅するブルームフィルタ 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 $\langle L_i, F_i\rangle$ がリスト S の先頭に追加される．$\langle L_i, F_i\rangle$ がリストへ追加された後，B_i は破棄される．

3.4 マ ー ジ

S には，3.3 節の操作によって，構築時間の降順に $\langle L_i, F_i\rangle$ が格納され，一定の戦略に従って選択される隣り合う組の集合 $\boldsymbol{L} = \{\langle L_i, F_i\rangle, \ldots, \langle L_j, F_j\rangle\}$ 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．マージは幅優先走査によってブルームフィルタの生成と同時に行われ，新たな組 $\langle L_{ij}, F_{ij}\rangle$ を生成して $\boldsymbol{L}$ を置き換える．

LOUDS トライのマージは，複数のソート済みファイルをマージする方法と同様に，すべての LOUDS トライを 1 度だけ幅優先走査することで実現している．このとき，同じ 1 度の走査で，LOUDS トライの生成とブルームフィルタの生成を同時に行うよう工夫している[24)]．

マージする組を選択する方法には様々な方法が考えられるが，ここでは，単純に一定数の組がリストに入れられたらすべてを 1 つの組にマージする戦略をとる．バッファのサイズを w，全体のキーの個数を N，1 度にマージする個数を f とすると，マージの回数は $N/(wf)$ となる．

3.5 検　　索

リスト S には同じキーが複数格納されている可能性がある．キーの検索要求に対しては，対応する値のうち，最近入力された値を応答しなければならない．これには，検索順を工夫する必要がある．

まず，バッファB_{i+1}，B_i の順でキーを検索する．ここで見つからなければ，リスト S の先頭から（新しい順に）LOUDS トライ L_j（$j < i$）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 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．

値が見つかればそれを検索結果として返す．最後まで見つからなければ，結果がないことを示す $\emptyset$ を返す．

3.6 ブルームフィルタ

ブルームフィルタを生成するには，各文字列から複数のハッシュ値を計算する必要がある．文字列を s，s_i を s の i 番目の文字，P を文字の種類の数に近い素数（たとえば 131 など）とすると，ここでは，ハッシュ値 h を以下のように計算する：

$h \leftarrow 0$
for $i = 0$ **to** $i < |s|$ **do**
　$h \leftarrow h \cdot P + s_i$
end for

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，異なる P を用いて，それぞれハッシュ値 h を計算すればよい．

ブルームフィルタを完成するには，LOUDS トライに含まれる全文字列について h を計算し，ブルームフィルタのサイズ m で剰余をとって，その値をアドレスとするビットを “1” にする．実際には，この操作を LOUDS の構築・マージと同時に行うことで効率良く実行しているが，その方法については，本論文の範囲を超えるため割愛する．

4. 計算モデルによる評価

ブルームフィルタはサイズと精度の間にトレードオフがあるため，サイズは，求める精度に対して機械的に定めることができる．検索性能は，ブルームフィルタの精度と LOUDS トライの個数によって決まる．本章では，ユニークキーの個数 N，精度を決めるハッシュ多重度 k，LOUDS トライの個数 M を変数として，ブルームフィルタのサイズと検索コストを求める．

4.1 ブルームフィルタのサイズ

ブルームフィルタはそのサイズと精度の間にトレードオフの関係がある．ブルームフィルタのビット列のサイズを m，ハッシュ多重度を k とすると，m を定めたときに，ブルーム

フィルタ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とき，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多重度は $k=(m/N)\ln 2$ である[9]．これを基に m を定めると，式 (1) が得られる．

$$m=\frac{k}{\ln 2}N\approx 1.44kN \tag{1}$$

このときの偽陽性確率 p は，式 (2) で表される．

$$p=\left(\frac{1}{2}\right)^k \tag{2}$$

キーの数を N，トライ木のノード数を n とすると，式 (1) によってブルームフィルタのサイズを定めた場合，ブルームフィルタのサイズは，$1.44kN$ であり，$N<n$ から，ブルームフィルタのサイズは，最悪でもノードあたり $1.44k$ ビット（$k=4$ では 5.76 ビット）となる．ノードあたり 43 ビット（BASE 2 bit，LEAF 1 bit，EDGE 8 bit，VAL 32 bit）を消費する LOUDS トライ全体と比較すると，12%程度であり，実用に耐えるコンパクトさである．

本論文では，簡単のため，ハッシュ多重度から逆算してブルームフィルタのサイズ m を定める．5 章の実験では，ブルームフィルタのハッシュ多重度のパラメータ k から，式 (1) によってキー数に対するビット列のサイズ比率 m/N を決定し，また，式 (2) により，偽陽性確率 p を求めることができる．

4.2　ブルームフィルタの効果

ブルームフィルタを付加することで得られる効果は，検索速度の向上である．特に，検索対象とすべき LOUDS トライの個数を絞り込むことによって，全体の LOUDS トライの数が増えた場合でも，検索速度の低下が緩やかになることが期待される．その効果の度合いを知るために，まず，ブルームフィルタがない場合とある場合のそれぞれについて，キーが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．以下では，LOUDS トライの個数を M とし，あるキー文字列 q の検索を行うとする．また，キーは 1 度だけ出現し，どの LOUDS トライにも重複は含まれていないとする．

まず，ブルームフィルタがない場合，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 トライの数は 1 個から M 個まで等しい確率で発生する．このとき，検索する LOUDS トライの数の期待値は 1 から M までの合計を M で割った数，$(M+1)M/2M=(M+1)/2$ であり，それぞれの 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 $C_L(M+1)/2$ と計算できる．

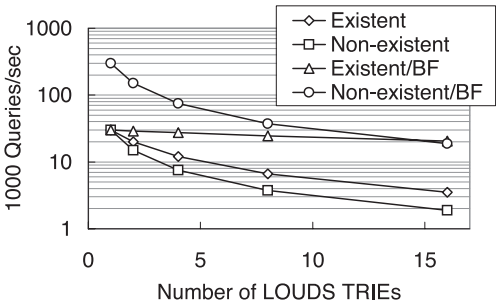

図 3　LOUDS トライの個数に対する検索速度の変化モデル
Fig. 3　Search performance model for the number of LOUDS TRIEs.

q がどのキー集合にも含まれていない場合には，つねにすべての LOUDS トライが探索され，検索される LOUDS トライの数の期待値は M と等しくなる．したがって，検索コストの期待値は C_LM である．

次に，ハッシュ多重度 k のブルームフィルタ付き LOUDS トライの計算コストを求める．ブルームフィルタがある場合には，q が含まれていないキー集合を調べる場合の偽陽性確率 p を加味する必要がある．

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は必ず真を返す．その前に検索されるブルームフィルタの偽陽性確率は $(1/2)^k$ であり，この確率で LOUDS トライが検索される可能性がある．したがって，検索される LOUDS トライの数の期待値は $((M+1)/2-1)(1/2)^k+1$ 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，ブルームフィルタを検索するコスト C_F は，実測で約 $1.25\,\mu s$ であり，LOUDS トライの検索コスト C_L（約 $33.3\,\mu s$）に対して非常に小さいため，ここでは無視している．

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，この期待値が非常に小さい場合があるため，ここでは，ブルームフィルタを検索するコスト C_F を加味する．つねにすべてのブルームフィルタが検索されるため，C_FM がコストの期待値に加算され，検索コストの期待値は $C_LM(1/2)^k+C_FM$ となる．

図 3 は，ここまでに求めた計算式に，計測して求めた C_L と C_F をあてはめて，コストの逆数，すなわち，検索速度をとってグラフを描いた結果である．Existent，Non-existent はそ

れぞれブルームフィルタがなく，キーが集合に含まれる場合と含まれない場合，Existent/BF と Non-existent/BF はブルームフィルタがある場合のキーが含まれる場合と含まれない場合である．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．

このモデルから，検索キーがいずれかの LOUDS トライに含まれるときには，提案手法によって，LOUDS トライの個数が増えた場合にも検索速度が変わらない効果が期待される．また，検索キーがどの LOUDS にも含まれない場合には，単一の LOUDS トライの検索速度を超えて，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も示唆している．

5. 評価実験

本章では，ブルームフィルタのサイズがどの程度の大きさになるかを実験で調べる．また，4.2 節の計算モデルを実験によって検証し，ブルームフィルタが，LOUDS トライの個数の増加による検索性能の低下を抑えることを確認する．

実験用のデータには，実際のアプリケーションでの利用を想定して，NHTSA [19] が提供する自動車の不具合報告データベースから言語処理によって抽出された，重複を含む平均約 27.2 文字のキーワード文字列約 2.4 億件のストリームを用いた．このストリームには，約 650 万個のユニークなキーワード文字列が含まれる．重複を除くキーワードの平均文字数は 34.5 文字である．

5.3 節では，上記ストリームのユニークキーのそれぞれに整数値を値として割り当て，実際に，提案手法の実装に入力して性能を計測する．

実験用のマシン環境には，IntelliStation APro，2.2 GHz Dual core Opteron 275×2，2nd cache 2 MB，PC3200 RAM 4 GB，750 GB SATA 7200 rpm × 2，Windows 2003 Server Standard x64 Edition Service Pack 2 を用いる．

5.1 ブルームフィルタのサイズ

ブルームフィルタのサイズを実測で調べる．データには，NHTSA のストリームから重複を取り除いたキーワード 640 万個を使用する．

LOUDS トライには，LOUDS 自身のほかに，各ノードの文字を格納する EDGE や，キーに対応する整数値を格納する VAL，最小接頭辞トライのために残りの接尾辞を格納する TAIL など，比較的大きなデータ構造が含まれる．実験で使用する 640 万個のキーワードを入力して実測すると，LOUDS のビット列はよく圧縮されているため，そのサイズは LOUDS トライ全体の 3.7%にすぎない．一方，EDGE のサイズは 14.9%，TAIL は 27.6%，整数をキーの数だけ持つ VAL は 37.2%の大きさを占める．これらの割合は，入力データに含まれる文字列の長さやバリエーションによって変化する．ブルームフィルタには，4.1 節で述べたとおり，ハッシュ多重度を k，キーの数を N としたとき，$1.44kN$ ビットが与えられる．

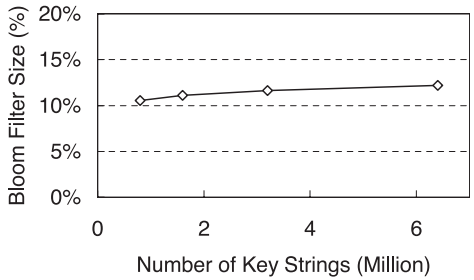

図 4　キー数の増加に対するブルームフィルタのサイズ比変化
Fig. 4　Size ratios of Bloom filters for the number of ingested keys.

図 4 は，キー数の増加にともなうブルームフィルタの全体に占めるサイズの割合の変化を示したグラフである．このときのブルームフィルタのハッシュ多重度は 4 である．キーの数が増えるにつれて VAL，EDGE，TAIL も大きくなるため，全体に占めるブルームフィルタの割合にも大きな変化はなく，そのサイズ比は約 12.2%となる．640 万個のデータを入力したときのサイズは，ブルームフィルタを含む LOUDS トライ全体が約 68 MB，ブルームフィルタ単体では 8.4 MB であった．

5.2 ブルームフィルタの効果

ブルームフィルタを付加することで得られる効果は，検索速度の向上である．特に，検索対象とすべき LOUDS トライの個数を絞り込むことによって，LOUDS トライの数が増えた場合でも，検索速度の低下が緩やかになることが期待される．以下では，キーがデータ構造に含まれている場合（Existent）と，含まれていない場合（Non-existent）に分けて，LOUDS トライの個数に対する検索速度の変化を調べる．

キーの有無を分けるために，NHTSA のストリームから重複を除いた 640 万個のキーワードのうち，320 万個のキーを入力して LOUDS トライを構築する．キーが含まれる場合の実験には，入力した 320 万個のキーワードを，キーが含まれない場合の実験には，入力しなかった 320 万個のキーワードを，それぞれ検索キーとして，検索スループット（queries/sec）

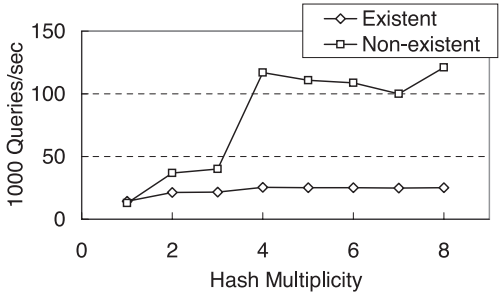

図 5　ハッシュ多重度によるブルームフィルタ付き LOUDS トライの検索性能の変化
Fig. 5　Search performance for hash multiplicity of LOUDS TRIEs with Bloom filters.

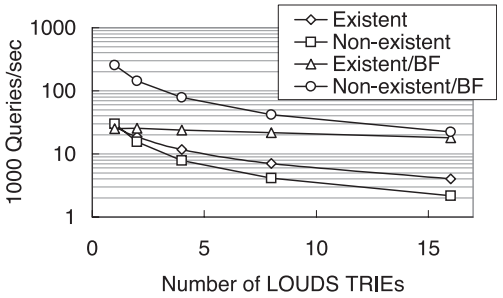

図 6　LOUDS トライの個数に対する検索速度の変化（実験結果）
Fig. 6　Search performance for the number of LOUDS TRIEs (experimental result).

を計測する．

最初に，検索に最適なハッシュ多重度を求めるため，ハッシュ多重度ごとに検索性能を調べる．対象とするブルームフィルタ付き LOUDS トライの個数は 8 個とする．**図 5** は，その結果を示すグラフである．

キーの有無にかかわらず，多重度 4 を境に大きな性能向上は見られなくなる．ハッシュ多重度を増やすと，偽陽性確率は下がり，より正確に検索すべき LOUDS トライを選択できるようになる．特に，キーが含まれないケースでは，偽陽性確率の低下とともに，ほとんど LOUDS トライに対する検索は発生しなくなる．

性能の計測に用いたブルームフィルタの実装では，ビット列のサイズを 2 のべき乗にすることで，ハッシュ値の剰余を 1 回のビット演算で実現している．これにより，ハッシュ多重度 2 から 3 と，4 から 7 は，それぞれビット列が同じサイズであった．ブルームフィルタは，ビット列のサイズが同じ場合，ハッシュ多重度を増やすと偽陽性確率が上がる．図 5 のハッシュ多重度 4 から 7 で性能が低下しているのはこのためである．ハッシュ多重度 8 の場合は，ビット列のサイズが多重度 4 から 7 の場合の 2 倍になるが，それによる偽陽性確率の改善が計算コストの上昇に対して小さいため，この実験では，性能は改善されなかった．

次に，LOUDS トライの個数が増えた場合にブルームフィルタがどのように効果を発揮するかを実験で確認する．**図 6** は，ハッシュ多重度 4 のときに，LOUDS トライの個数の増加に対して，全体の検索速度がどのように変化するかを調べた結果である．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．

ブルームフィルタがない場合には LOUDS トライの数が増えるに従って検索速度は大きく低下する．一方，ブルームフィルタを付加した場合では，検索キーが含まれるときには大きな速度の低下は見せず，また，検索キーが含まれないときには，特に LOUDS トライの個数が少ない場合に，検索速度が非常に速くなる．これは，ブルームフィルタによって，検索すべき LOUDS トライを効果的に選択できていることを示している．キーが含まれていない場合でも，LOUDS トライの数が多くなると，全体では偽陽性確率が高まり，不要な LOUDS トライの検索が発生して，その検索速度はキーが存在する場合の速度に近づいていく．これらの曲線は，計算によって求めた図 3 の曲線とよく一致する．

これらの実験結果から，提案手法によって，LOUDS トライの個数が増えた場合でも，LOUDS トライの個数が 1 つの場合に比べて，検索性能が大きく低下しないことが確認された．また，検索キーがどの LOUDS にも含まれない場合には，単一の LOUDS トライの検索速度を超えて，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場合もある．

5.3　実データによる効果の検証

最後に，辞書を構築する実際の利用形態に即した実験を行い，この例の場合について，提案手法の実用的な構成について考える．

実験には，NHTSA のデータベースから抽出した重複を含む 2.4 億件のキーワードのストリームの全体を使い，辞書にキーワードが登録されていない状態からはじめて，キーワードが辞書に登録されているかを調べ，登録されていなければ，辞書に追加する処理を行う．

ブルームフィルタが LOUDS トライの個数の変化に対してどのように効果を発揮するかを

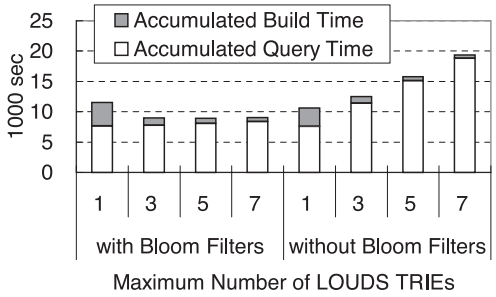

図 7　実データによる辞書構築実験の結果
Fig. 7　An experimental result of building dictionary from the real data.

調べるため，ブルームフィルタがあるケース（with Bloom Filters）とブルームフィルタがないケース（without Bloom Filters）の両方について，この実験を，マージを行う LOUDS トライの個数を変化させて実行し，処理にかかる時間を，合計の LOUDS トライの構築・マージ時間（Accumulated Build Time）と合計の検索時間（Accumulated Query Time）に分けて計測する．バッファのサイズはいずれも 40,000 とし，40,000 個の入力ごとに LOUDS トライが構築される．

図 7 は，この実験の結果をまとめたグラフである．横軸の数値は，検索対象となる LOUDS トライの最大数を表し，バッファとして用いるトライの数を含まない．

この実験では，ユニークなキーは全入力の約 2.56%であり，処理の 97.44%が検索のみを含む．したがって，LOUDS トライの構築・マージ回数は全体に対して少なく，ブルームフィルタによって，LOUDS トライの検索が省略される確率も高くはない．この条件下では，ブルームフィルタがない場合には，LOUDS トライの個数が 1 つの場合が最も実行時間が短い．構築・マージ時間は最も長いが，LOUDS トライが 3 以上の場合の検索速度の低下を埋めるほどではない．

一方，ブルームフィルタがある場合では，最も実行時間が長いのは LOUDS トライの最大数が 1 の場合であり，ブルームフィルタの効果が働く分，頻繁にマージを実行するコストが顕在化する．それ以上の場合には，構築時間の割合が小さくなるため，実行時間に大きな差はない．これらの結果から，LOUDS トライの個数が増えることによる検索速度の低下をブルームフィルタが防ぐ効果が確認できる．

この実験の中で最も実行時間が短かったのは，ブルームフィルタあり，LOUDS トライの最大数が 5 の場合であり，これは，ブルームフィルタなしの場合の最も実行時間が短い場合（LOUDS トライが 1 つの場合）に対して約 16.4%の改善となる．

この実験では，バッファの最大サイズは 1.5 MB，バッファから作られる LOUDS トライのサイズは約 0.6 MB，マージされた LOUDS トライの最大サイズは約 68 MB である．マージ時には，マージ前後の LOUDS トライが同時に存在するため，最大サイズの LOUDS トライがマージされるときにメモリ使用量が最大となる．最大 8 個の LOUDS トライがマージされる場合，最大メモリ使用量は，およそ $2 \cdot 68 + 7 \cdot 0.6 = 140.2$ MB 以下と見積もることができる．Double Array を用いた場合には，メモリ使用量は単調増加となり，その最大サイズは約 150 MB である．

6. おわりに

本論文では，インクリメンタルにデータを入力して LOUDS トライを構築する，オンライン LOUDS トライ構築法に，ブルームフィルタを組み合わせることで，差分の LOUDS トライの個数が増えた場合に検索速度が低下するという問題を解決した．

本手法は，検索時に，検索キーを含まない LOUDS を排除する効果があり，特に，検索キーがどの LOUDS トライにも含まれない場合の検索性能を大幅に向上する．この改善は，新しいキーを追加しながら検索を行う場合に効果的であり，辞書を作成する実験で，2.5%程度の新規キーを含む入力に対して，16%以上の性能の改善を確認し，本手法の有効性を検証した．

本手法によって，LOUDS を利用した空間効率の良いトライ木をオンライン構築できるようになったが，全文検索など，実際のアプリケーションで利用するには，まだ最適化の余地を残している．今後は，これを応用した，より高速で空間効率の高いキー・バリュー型データストアを開発し，発表したい．

参考文献

1) Aoe, J.: An efficient digital search algorithm by using a double-array structure, *IEEE Trans. Software Engineering*, Vol.15, pp.1066–1077 (1989).
2) Apache Software Foundation: *The Apache HBase Book* (Oct. 2010).
3) Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V. and Rao, S.S.: Representing trees of higher degree, *Algorithmica*, Vol.43, pp.275–292 (Dec. 2005).
4) Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors, *Commun. ACM*, Vol.13, pp.422–426 (July 1970).
5) Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M.,

Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A distributed storage system for structured data, *ACM Trans. Computer Systems* (*TOCS*), Vol.26, pp.4:1–4:26 (June 2008).
6) Crochemore, M. and Verín, R.: Direct construction of compact directed acyclic word graphs, *Combinatorial Pattern Matching*, pp.116–129, Springer-Verlag (1997).
7) DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's highly available key-value store, *Proc. 21st ACM SIGOPS symposium on Operating systems principles* (*SOSP '07*), pp.205–220, ACM (2007).
8) Delpratt, O., Rahman, N. and Raman, R.: Engineering the louds succinct tree representation, *Proc. 5th International Workshop on Experimental Algorithms, Lecture Notes in Computer Science*, Vol.4007/2006, pp.134–145, Springer-Verlag (2006).
9) Fan, L., Cao, P., Almeida, J. and Broder, A.Z.: Summary cache: A scalable wide-area web cache sharing protocol, *IEEE/ACM Trans. Networking* (*TON*), Vol.8, pp.281–293 (June 2000).
10) Fredkin, E.: Trie memory, *Commun. ACM*, Vol.3, pp.490–499 (Sep. 1960).
11) Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M. and Arikawa, S.: On-line construction of symmetric compact directed acyclic word graphs, *Proc. 8th International Symposium on String Processing and Information Retrieval* (*SPIRE '01*), pp.96–110, IEEE Computer Society (2001).
12) Jacobson, G.J.: Succinct static data structures, Ph.D. thesis, Carnegie Mellon University, AAI8918056 (1988).
13) Jacobson, G.J.: Space-efficient static trees and graphs, *Proc. 30th Annual Symposium on Foundations of Computer Science*, pp.549–554, IEEE Computer Society (1989).
14) Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, *Proc. 29th annual ACM symposium on Theory of computing* (*STOC '97*), pp.654–663, ACM (1997).
15) Kim, D.K., Na, J.C., Kim, J.E. and Park, K.: Efficient implementation of rank and select functions for succinct representation, *Experimental and Efficient Algorithms*, Lecture Notes in Computer Science, Vol.3503/2005, pp.125–143 (2005).
16) The kumofs Project: kumofs: Extremely fast and scalable distributed key-value store, available from ⟨http://kumofs.sourceforge.net/⟩.
17) Lakshman, A. and Malik, P.: Cassandra: A decentralized structured storage system, *ACM SIGOPS Operating Systems Review*, Vol.44, pp.35–40 (Apr. 2010).
18) Munro, J.I. and Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs, *Proc. 38th Annual Symposium on Foundations of Computer Science*, pp.118–126 (Oct. 1997).
19) National highway traffic safety administration, available from ⟨http://www.nhtsa.gov/⟩.
20) roma prj.Roma: A distributed key-value store in ruby, available from ⟨http://code.google.com/p/roma-prj/⟩.
21) Sadakane, K.: Dynamic succinct ordinal trees, IEICE technical report, Theoretical foundations of Computing, Vol.109, No.9, pp.37–41 (Apr. 2009).
22) Sadakane, K. and Navarro, G.: Fully-functional succinct trees, *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms* (*SODA '10*), pp.134–149, Society for Industrial and Applied Mathematics (2010).
23) Sato, I. and Nakagawa, H.: Succinct semi-structured data mining based on FREQT, *DBSJ Journal*, Vol.9, No.1, pp.76–81 (2010).
24) 小柳光生，吉田一星，海野裕也，新城　靖：簡潔データ構造のオンライン構築のためのブルームフィルタ構築法，Technical report，筑波大学システム情報工学研究科コンピュータサイエンス専攻 (Aug. 2011).
25) 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 (Sep. 2008).

(平成 23 年 6 月 20 日受付)
(平成 23 年 10 月 3 日採録)

(担当編集委員　定兼 邦彦)

小柳　光生（正会員）

1974 年生．1999 年筑波大学大学院理工学研究科修士課程修了．同年日本アイ・ビー・エム（株）入社．東京基礎研究所配属．アプリケーションサーバ，データベースの研究に従事．2008 年より同社大和ソフトウェア開発研究所に転属．ディスカバリ製品の開発に従事．筑波大学大学院社会人博士課程コンピュータサイエンス専攻．

吉田　一星（正会員）

1976 年生．2001 年東京大学大学院数理科学研究科修士課程修了．同年日本アイ・ビー・エム（株）入社．大和ソフトウエア開発研究所配属．データベース関連製品の開発に従事．2003 年より同社東京基礎研究所に転属．テキストマイニング，特に検索・集約処理の高速化の研究開発に従事．日本データベース学会会員．

海野　裕也

1983 年生．2008 年東京大学大学院情報理工学系研究科修士課程修了．同年日本アイ・ビー・エム（株）入社．東京基礎研究所配属．自然言語処理，テキストマイニングの基礎技術の研究に従事．2011 年株式会社プリファードインフラストラクチャー入社．研究開発部門配属．自然言語処理，機械学習の研究開発に従事．言語処理学会会員．

新城　　靖（正会員）

1965 年生．1988 年筑波大学第三学群情報学類卒業．1993 年筑波大学大学院工学研究科電子・情報工学専攻博士課程修了．同年琉球大学工学部情報工学科助手．1995 年筑波大学電子・情報工学系講師，2003 年同助教授，2004 年同大学院システム情報工学研究科助教授．2007 年同准教授．オペレーティング・システム，分散システム，仮想システム，並行システム，情報セキュリティの研究に従事．博士（工学）．ACM，IEEE，日本ソフトウェア科学会各会員．