LOUDSトライのオンライン構築のための ブルームフィルタ構築法

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極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, 大量のキー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. このとき, 各 LOUDS にブルームフィルタを添付すると, 不要な検索をスキップすることで, 検索性能を改善できる. しかし, ブルームフィルタを作成するには, 素朴な方法ではマージ処理とは別に LOUDS トライを探索する必要があり, 性能が悪い. 本論文では, LOUDS の構築・マージと同時にブルームフィルタを作成することで, ブルームフィルタの構築時間を削減する方法を提案する. 実データから抽出した 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.

Method to Build Bloom Filters for Online Building of LOUDS TRIE

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. LOUDS TRIE can be incrementally built by creating multiple LOUDS to keep deltas and merging them periodically. Setting Bloom filters with respective LOUDS improves the search performance by excluding unnecessary searches. However, it costs a substantial time to create Bloom filters by reading all key strings stored in LOUDS. In this paper, we propose a method to create Bloom filters concurrently with building and merging LOUDS. It conserves the time to build Bloom filters. The efficiency of our method is confirmed by the experiment using about 240 million word stream that consists of 6.5 million unique keywords, which are extracted from the real data.

1. はじめに

テキストマイニングやゲノム解析など, 文字列で表されるデータを大量に扱う分野では, 高い空間効率でデータを保持し, かつ, 効率よく検索を行う方法が求められる. 簡潔データ構造[9] は, 極めて高い空間効率を実現しながら, 検索も効率よく行えるため, これらのアプリケーションへの応用が期待される.

我々は, 文字列をキーとするコンパクトなマッピングを実現するため, 木構造を表す簡潔データ構造の一種である, LOUDS (Level-Order Unary Degree Sequence)[10] によるトライ木 (LOUDS トライ) の実装に着目し, LOUDS トライにインクリメンタルにデータを加えて計算結果を得られるように工夫した, LOUDS トライの「オンライン構築」の研究を行っている[20]. LOUDS は, 高い空間効率を実現するため, データ全体を計算対象として一度に構築する必要があり, 一度構築されたデータ構造に, 効率よく新たなデータ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をバッファに蓄え, 図 1 のように, 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

†1 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
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

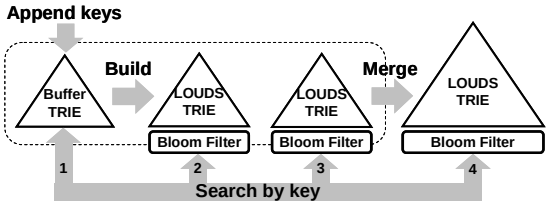


図 1 ブルームフィルタ付き LOUDS トライのオンライン構築
Fig. 1 Online LOUDS building system with Bloom filters

築する.

この方法では, 構築のたびに検索すべき LOUDS トライの個数が増え, その数に比例して検索性能が低下するため, 一定数の LOUDS トライが作られるたびにそれらをマージして, その個数を一定以下に抑える. さらに, 各 LOUDS トライに, そのトライの保持するキーセットに対応したブルームフィルタを添付して, 不要な検索を除外し, 検索の効率を上げる[20].

この論文では, 効率的なブルームフィルタの構築方法について述べる. ブルームフィルタは, 全体として良い性能を得るには, バッファから最初に作れらる LOUDS トライだけでなく, LOUDS トライをマージした結果生成される LOUDS トライにも設置する必要がある. 同じサイズのブルームフィルタであれば, 全ビットの論理和を取ることでマージできるが, 全ての LOUDS トライに, 事前に考えられる最大のブルームフィルタを設置する必要があり, LOUDS トライの数に応じてメモリ使用量が大きくなる. また, 入力されるキーの数が事前に分からない場合は, ブルームフィルタのサイズを決めることができない. そのため通常は, 空のブルームフィルタを新たに用意し, マージ後の LOUDS トライから全てのキーを取得して再構築する必要がある. しかしながら, この方法では LOUDS トライからキーを取り出すコストが大きくなるという問題があった.

この問題を解決するために, 本研究では, ブルームフィルタの構築を LOUDS トライのマージと同時に行う手法を提案する. これにより, マージ後の LOUDS トライをブルームフィルタの構築のみのために再探索するコストを削減する. LOUDS トライのマージとブルームフィルタの構築を同時に行うことは, 単純な方法ではプログラムが難解になってしまう. そこで本研究では, 仮想ノードという考え方を導入してこの問題を解決する. 仮想ノードとは, マージの途中にある 2 つの LOUDS トライをあたかも 1 つの LOUDS トライとしてデータを取り出したり探索したりできるようにしたものである. 仮想ノードの導入により, プログラムが簡素化され, LOUDS トライのマージとブルームフィルタの構築が容易に実現される.

データから抽出した 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った. その結果, 提案方式によりブルームフィルタの構築が高速化されることを確認した.

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, オンライン LOUDS トライ構築システムに求められる機能とデザインの概要を示す. 4 章では, ブルームフィルタを LOUDS トライの構築・マージと同時に生成するアルゴリズムを示す. 5 章では, 実データを使った実験により, 提案手法が構築コストを低減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[3] は, 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データ構造である. ブルームフィルタは"0"で初期化されたビット列からなり, キーが追加されるときには, そのキーに対して複数の独立したハッシュ値が計算され, それらの示す位置のビットが"1"にされる. 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[7]. 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良く利用される. 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[8] 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. 例えば, 順序木の簡潔データ構造は n を木のノー

ド数とすると, n が大きくなったとき, 漸近的に $2n$ に近づくような関数 (LOUDS では, $2n + \log n$) でバウンドされるビット数で表現される. また, 最小限の表現のまま, データに対する操作を効率よく実現することができる. これまでに, 順序木, 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[15)18)19)].

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[6)12)]. 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[21)].

以降では, キーを動的に追加できるトライ木を動的トライと呼ぶ. トライ木という場合には, 動的トライと LOUDS トライの両方を指す.

この論文では, LOUDS トライに対してブルームフィルタを設置する場合の効率的な手法について述べる.

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扱うが, 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[4)2)5)14)13)17)], 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. 本手法は, 分散データストアにおいて単一のノードの主記憶に効率的にデータを保持するために利用できる. この時には, Consistent Hashing[11)] などのデータ分割の手法と組み合わせる方法が考えられる.

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手法について述べる.

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. 出来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, データは入力後直ちに検索結果に反映させなければならない. 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にしばしば現れる. 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数億件になることもあるため, 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.

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.

本論文では, これらの要件に加え, 検索を高速化するために, ブルームフィルタを用いる. ブルームフィルタは各 LOUDS トライに設置され, ブルームフィルタを先に検索することで, 検索キーを含まない LOUDS トライを高い精度で検索対象から除外できるようになる.

検索キーが含まれない LOUDS トライを効果的に除外するためには, ブルームフィルタの偽陽性確率を一定の小さい値に抑える必要がある. これには, ビット列のサイズを入力キーの個数 N に比例したサイズにすればよい[20)]. そのため, 本論文では, LOUDS トライをマージする際に, 最適なサイズのブルームフィルタを再構築する手法を取る. 各 LOUDS トライの保持するキーの数を N_i, ブルームフィルタのサイズを決める係数を k, LOUDS トライの個数を f とすると, 本手法でのブルームフィルタのサイズの合計は以下のようになる.

$$\sum_{i=1}^{f} kN_i = k\sum_{i=1}^{f} N_i = kN \qquad (1)$$

一方, ブルームフィルタの他のマージ方法として, 全てのブルームフィルタのサイズを一定にし, 論理和によって高速にマージする手法が考えられる. この手法では, ブルームフィルタのサイズを事前に考えられる最大のキー数 $N_{\max}(\geq N)$ に基づいて決めておく必要がある. このとき, LOUDS トライの個数を f とすると, 全体のブルームフィルタのサイズは $fkN_{\max}$ となり, 本手法とのサイズの差は最低でも f 倍となる. 高速にマージが行われることで, 構築時間に対して一定の割合の改善が得られるが, LOUDS トライの個数を増やすとメモリ使用量が大きくなるため, 使用可能なメモリ量が限られる場合は, マージ回数を減らすことによる性能改善に限界がある.

本論文の手法では, ブルームフィルタのサイズが

LOUDS トライの個数によらないため, 同じメモリ量では, LOUDS トライの個数をずっと大きな値に設定することが出来る. マージ回数を減らすことによる改善は, 構築のコスト全体に及ぶため, LOUDS トライの個数を大きな値に設定できる場合は, ブルームフィルタの構築コストだけを改善する論理和によるマージ手法よりも改善が大きい. 従って, 本手法は, 論理和によってブルームフィルタをマージする手法と比較すると, 入力データ量が事前に分からない場合にも使用でき, メモリ効率が良く, メモリ使用量が限られる場合は, より高い性能を達成できる.

3.2 バッファリングと構築

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, 同じ添え字をも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 の昇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直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 に i 番目の要素として追加される. $\langle L_i, F_i \rangle$ がリストへ追加された後, B_i は破棄される.

LOUDS トライを構築するアルゴリズムを 4.2 節に示す. ブルームフィルタを生成するアルゴリズムを 4.4 節に示す.

3.3 マ ー ジ

S には, 3.2 節の操作によって, 構築時刻順に $\langle L_i, F_i \rangle$ 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 $\boldsymbol{L} = \{\langle L_i, F_i \rangle ..., \langle L_j, F_j \rangle\}$ 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.

2 つの LOUDS トライをマージするアルゴリズムを 4.3 節に示す.

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全てをひとつの組にマージする戦略を取る. バッファのサイズを w, 全体のキーの個数を N, 一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.

3.4 検　　索

同じキーが, 入力データの異なるウィンドウに出現すると, リスト S には, 同じキーが複数格納される. 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, 若い順 (リストの降順) 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.

キーが見つかればその値を検索結果として返す. 最後までキーが見つからなければ, 検索結果が空であることを示す $\emptyset$ を返す.

ブルームフィルタの効果によって, 不要な LOUDS トライの検索が排除され, LOUDS トライの個数に依存しない検索性能が発揮される[20].

4. アルゴリズム

この章では, 提案する LOUDS トライのマージ, および, ブルームフィルタの構築のアルゴリズムを示す.

LOUDS ではない 2 つの動的トライのマージは, 一方のノードを走査しながら, 他方に, 存在しないノードを追加する操作によって実現できる. 動的トライでは, 木構造をリンクによって表現し, ノードの追加を高速に行うことができる. リンク構造の書き換えによって, 容易にノードの追加が行える動的トライとは異なり, 静的なビット列からなる LOUDS に新たなノードを追加することはできない. このため, 2 つの LOUDS トライをマージするためには, それらの持つキーの和集合を求め, そのキー集合から新たな LOUDS トライを作ることが必要になる. キーの和集合を求めるための

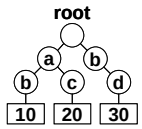
素朴な方法としては, 両者のキーを空の動的トライ B_m に入力し, B_m を走査して LOUDS トライを構築すればよい. しかしながら, この素朴な方法では, マージされたキー集合のためのブルームフィルタを作成するために, マージの後, もう一度, マージされた LOUDS トライから全てのキーを取り出す必要がある.

そこで, 本研究では, キーの和集合を求めるために動的トライを用いる代わりに, マージされる 2 つの LOUDS トライを, あたかも 1 つの LOUDS トライであるかのごとく操作できるようにする. こうして 1 つになった LOUDS トライのノードを仮想ノードと呼ぶことにする. 仮想ノード $M(n_1, n_2)$ とは, トライ木の 2 つのノード n_1, n_2 を保持し, それらをマージしたノードを表すオブジェクトである. 仮想ノード M は, マージされたノードに対して, 通常のトライ木のノードと同等の操作を提供する. たとえば, キーの検索や, 幅優先走査を行うことができる. LOUDS トライのマージは, M による仮想的なマージ木に対する幅優先走査によって, 新しい LOUDS トライを構築することで行われる. この走査の途中にキーを取り出し, ハッシュ値を計算し, ブルームフィルタを生成する.

仮想ノードを利用せず, マージしたトライ木を扱うためには, ノードの組 $\langle n_1, n_2 \rangle$ を直接扱えるように記述する必要がある. 例として, 仮想ノードを利用せずに, マージしたトライ木に対する幅優先走査を実現したプログラムを付録 A に示した. 仮想ノードを利用する場合, マージしたトライ木に対応する幅優先走査のプログラムは, 4.1 節のプログラムと同じになり, 11 ステートメントで表現できるが, 付録 A のアルゴリズムでは 35 ステートメントになり, 仮想ノードによって個々のプログラムが簡潔になることが分かる.

4.1 トライ木の幅優先走査

トライ木 T を幅優先走査するには, ノード n を訪れた後, その子のうち, 最初の子を訪れ, 続いてその兄弟を順に訪れれば良い. 従って, ノード n に対して, 最初の子を返す firstChild(n), 隣のノードを返す sibling(n), そして, トライの文字を返す alphabet(n) が実現されれば, 幅優先走査が可能になる. トライ木では, 子のノードはアルファベット順に整列しているため, firstChild(n) では, n に続くノードのうち, 文字の最も小さいノードが返され, sibling(n) では, n の次に小さい文字を持つノードが返される. n に続く文字がないとき, firstChild(n) $= \emptyset$ とし, n が共通接頭辞に対して最大の文字を持つとき, sibling(n) $= \emptyset$ とする. これらの操作を用いて, トライ木のルートノード r から, 幅優先走査を行うアルゴリズムは, n, c をノードとすると, キュー q を使って, 次のように書ける ($|q|$ は q に含まれる要素数).



BASE	1011011010000
EDGE	abbcd
LEAF	000111
VAL	10,20,30

図 2 LOUDS トライの構築例
Fig. 2 Example of constructs in LOUDS TRIE

```
q ← {r}
visit(∅, r)
while |q| ≠ 0 do
  p ← q.dequeue()
  n ← firstChild(p)
  while n ≠ ∅ do
    q.enqueue(n)
    visit(p, n)
    n ← sibling(n)
  end while
end while
```

トライ木のノード n は, **visit**(p, n) によって, その親ノード p を伴って幅優先順に走査される. 続く節では, LOUDS トライの構築, マージ, および, ブルームフィルタの生成が, firstChild(n), sibling(n), alphabet(n), および, **visit**(p, n) を定義することによって実現できることを示す.

4.2 幅優先走査による LOUDS トライの構築

LOUDS トライを構築する場合, 任意のビットを追記できるビット列 BASE と LEAF, 任意の文字を追記できる文字列 EDGE の 3 つのデータを用意し, 次の操作を実行する. まず, ビット列 “10” を BASE に追加する. 続いて, 幅優先走査を実行し, **visit**(p, n) で, n の子と同じ個数の “1” と続く “0” を BASE に追加する. 子の数が 0 の場合は “0” だけが追加される. EDGE には alphabet(n) を追加する. LEAF には, n に子が無ければ “1” をあれば “0” を追加する.

ここで, LEAF は幅優先走査順にノードがリーフかどうかを示したビット列であり, 幅優先走査の出現順に値をリスト VAL に格納したとすると, LEAF の rank が, その LEAF に対応する VAL の位置を示す. 図 2 は, キー “ab”, “ac”, “bd” を含む LOUDS トライの構築例である.

さらに, BASE, LEAF に対して 12) の方法で索引を構築する. これらの索引は, ノード数を N とすると, O(N) ビット (実際は元のビット列の 10%以下) のメモリを消費し, rank/select を定数時間で実現する.

表 1 マージノードの操作
Table 1 Functions of Merged TRIE nodes

$\text{firstChild}(M(n_1,n_2)) \rightarrow$	$M(\text{firstChild}(n_1),\text{firstChild}(n_2))$ **when** $\text{alphabet}(n_1)=\text{alphabet}(n_2)$,
	$\text{firstChild}(n_1)$ **when** $\text{alphabet}(n_1)<\text{alphabet}(n_2)$,
	$\text{firstChild}(n_2)$ **when** $\text{alphabet}(n_1)>\text{alphabet}(n_2)$
$\text{sibling}(M(n_1,n_2)) \rightarrow$	$M(\text{sibling}(n_1),\text{sibling}(n_2))$ **when** $\text{alphabet}(n_1)=\text{alphabet}(n_2)$,
	$M(\text{sibling}(n_1),n_2)$ **when** $\text{alphabet}(n_1)<\text{alphabet}(n_2)$,
	$M(n_1,\text{sibling}(n_2))$ **when** $\text{alphabet}(n_1)>\text{alphabet}(n_2)$
$\text{alphabet}(M(n_1,n_2)) \rightarrow$	$\min(\text{alphabet}(n_1),\text{alphabet}(n_2))$

こうして作られる LOUDS トライもまた, firstChild, sibling, alphabet の操作を提供する[6].

4.3 マージされたトライ木の幅優先走査

4.1 節で述べたように, トライ木では, 全てのノードの子の列はアルファベット順にソートされている必要がある.

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, n_2 をマージしたノードを $M(n_1,n_2)$ とするとき, 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, $M(n_1,n_2)$ の子の列とすれば良い. これを, ルートを含む全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.

マージソートの振る舞いを, M を使って幅優先走査に組み込むことができる, 表 1 に, n_1 と n_2 の子をマージする $M(n_1,n_2)$ に対する 3 つの操作, firstChild, sibling, alphabet を n_1, n_2 に対する操作によって定義する. なお, 便宜的に, 任意のアルファベット α に対して $\text{alphabet}(\emptyset)>\alpha$ であるとする.

T_1, T_2 のルートノードを r_1, r_2 とすると, $M(r_1,r_2)$ によって, T_1, T_2 をマージするトライ木 T_m のルートノードが得られる. T_m に対して 4.2 節のアルゴリズムを実行することで, T_1 と T_2 をマージした LOUDS トライが構築される.

4.4 ブルームフィルタの生成

幅優先走査でのハッシュ値の計算方法を求めるために, まず, キー文字列 s のハッシュ値の計算方法を定める. 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 (例えば 131) として, 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

幅優先走査で, トライ木に含まれる文字列のハッシュ値を計算するには, ノードを訪れるたびに, それぞれの文字列のハッシュ中間値 h を更新すれば良い. ただし, 全ての文字列のハッシュ値を並行して計算することになるため, 枝分かれした各ノードにハッシュ中間値を保持する変数 $n.h$ を用意する必要がある. トライ木の構造から, 共通接頭辞のハッシュ中間値は共有される.

“0” ビットで初期化された長さ l のビット列 BF を用意し, 幅優先走査の $\textbf{visit}(p,n)$ で以下の計算を行う: $n.h \leftarrow p.h \cdot P + \text{alphabet}(n)$. ただし, n がルートノードの場合は, $n.h \leftarrow 0$ とする. また, n が子を持たないなら, $n.h$ は求めるハッシュ値となり, BF の $n.h \bmod l$ 番目のビットを “1” にする. こうして作られるビット列 BF がブルームフィルタとなる.

実際のブルームフィルタでは, ひと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, 中間の計算結果 $n.h$ を配列にし, 異なる P を用いて, それぞれハッシュ値を計算すれば良い. 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逐次的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.

5. 評 価 実 験

本手法の効果を評価するために, ブルームフィルタを生成する際にかかる時間を実験によって調べ, 従来の手法と比較する. また, 検索も含む実際のアプリケーションを想定した実験を行い, オンライン構築全体に対する改善の度合いを調べる.

実験用のデータには, NHTSA[16] 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. 3 節のオンライン LOUDS トライ構築機構に, 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, キーワード文字列から割り当てられた整数値を検索する. この条件は, 文書処理によって抽出された語彙から逐次的

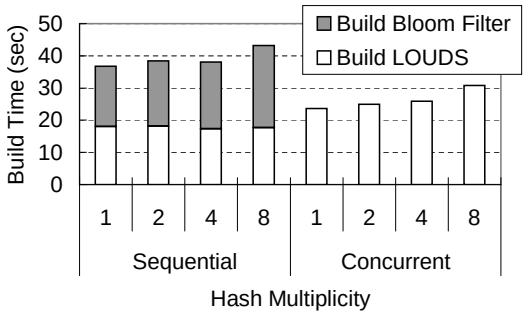


図 3 逐次的実行と並行実行の構築コスト比較
Fig. 3 Cost comparison between sequential and concurrent building

表 2 逐次的実行と並行実行の構築コスト比較 (数値), 単位:秒, #はハッシュ多重度
Table 2 Cost comparison between sequential and concurrent building, in second, # is hash multiplicity

	Sequential			Concurrent
#	LOUDS TRIE	Bloom Filter	Total Time	Total Time
1	18.06	18.67	36.73	23.65
2	18.15	20.21	38.37	24.90
4	17.34	20.71	38.02	25.86
8	17.60	25.59	43.23	30.81

に辞書を作成するアプリケーションを想定している.

実験用のマシン環境には, IntelliStation APro, 2.2GHz Dual core Opteron 275 ×2, 2nd cache 2MB, PC3200 RAM 4GB, 750GB SATA 7200rpm ×2, Windows 2003 Server Standard x64 Edition Service Pack 2 を用い, 実験用のコードは Java 1.6 で記述した.

5.1 ブルームフィルタのコスト

まず, ブルームフィルタを構築する際の時間を調べる. 実験には, NHTSA のストリームから重複を取り除いた平均長 34.5 文字のキーワード 640 万個を使用し, 3.2 節でバッファとして使用する動的トライに入力して, LOUDS トライと対応するブルームフィルタを 1 つ作るときの時間を計測する.

図 3 は, LOUDS トライの構築とブルームフィルタの作成を逐次的に実行した場合 (Sequential) の時間と, LOUDS トライを構築しながら, 同時にブルームフィルタを作成した場合 (Concurrent) の時間を, ブルームフィルタのハッシュ多重度ごとに比較したグラフである. また, 表 2 に, 図 3 の具体的な数値を示した. 各個内の数値は, 逐次的に実行した場合のブルームフィルタ構築の占める時間である.

逐次的に構築する場合と比較して, LOUDS トライの構築と同時にブルームフィルタを作成することで, ノードの走査が 1 回で済むため, 約 30%の時間短縮になる.

また, これらのデータから, ハッシュ多重度が 4 の時の各処理の割合を計算すると, ノード走査に 47%, LOUDS トライのデータ構築に 20%, ブルームフィルタの作成に 33%の時間がかかっていることが分かる. ハッシュ多重度が上がるにつれてノードあたりの計算回数が増えるため, ブルームフィルタの作成時間の割合は増加する.

5.2 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, 本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.

ブルームフィルタと LOUDS トライの同時構築によって, 全体の性能にどれくらいの影響があるかを調べるため, 同時構築しないケース (Sequential) と同時構築するケース (Concurrent) のそれぞれについて, マージを行う LOUDS トライの個数を変化させて試行する. その際, 処理にかかる時間を, ブルームフィルタと LOUDS トライの構築・マージ時間の合計 (Accumulated Build Time) と, 検索時間の合計 (Accumulated Query Time) に分けて計測する. ブルームフィルタのハッシュ多重度は 4 とした.

図 4 は, この実験の結果をまとめたグラフである. 横軸の数値は含まれる LOUDS トライの最大数を表す. また, 表 3 に, 図 4 の具体的な数値を示した.

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. 従って, LOUDS トライの構築・マージ回数は全体に対して少なく, LOUDS トライの構築時間が全体に対して占める割合も高くはない. 検索時間は, チェックすべきブルームフィルタの個数が増えるため, LOUDS トライの個数が多い方がわずかに長くなるが, 全体に大きな差はない. 一方, 構築時間は, LOUDS トライの個数によって大きく異なる.

ブルームフィルタの同時構築による効果が最も高いのは, マージ回数の最も多い LOUDS トライが 1 つのケースであり, 検索の割合の多いこの実験でも, 全体の処理時間を 11%削減した. マージ回数の少ない他のケースでも, 3%から 5%の削減を達成した.

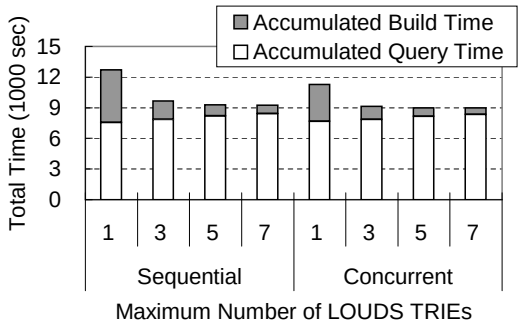


図 4 実データによる辞書構築実験の結果
Fig. 4 An experimental result of building dictionary from the real data

表 3 実データによる辞書構築実験の結果 (数値), 単位:秒 (#は LOUDS TRIE の最大数)
Table 3 An experimental result of building dictionary from the real data, in second (# means the maxium number of LOUDS TRIEs)

	Sequential			**Concurrent**		
#	Query Time	Build Time	Total Time	Query Time	Build Time	Total Time
1	7540	5151	12691	7657	3624	11281
3	7841	1815	9656	7844	1296	9140
5	8180	1114	9294	8148	838	8986
7	8426	840	9266	8344	633	8977

6. おわりに

本論文では, LOUDS トライの構築・マージと同時に, LOUDS トライに含まれるキー集合に対応するブルームフィルタを生成する手法を導入し, ブルームフィルタを効率的に生成する手法を提案した. このために, 本手法では, 仮想ノードという概念を導入した. 仮想ノードとは, マージする 2 つの LOUDS トライの対応するノードを保持し, マージされた場合と同等の操作を実現するものである. 仮想ノードを用いることで, 2 つのトライ木をひとつのトライ木と同様に, 幅優先走査したり, キーを取り出したりすることができる. これにより, マージする LOUDS トライを全て保持する中間的なデータ構造を不要にし, LOUDS トライのマージ処理と同時にブルームフィルタを構築することができるようになった.

提案手法は, Java 言語により実装した. LOUDS トライのマージとブルームフィルタの生成を行う実験では, 提案方式は逐次的に行う方式と比較して約 30%の性能が改善された. 2.4 億件の実データを用いて辞書を作成するアプリケーションでは, 3%から 11%の改善を確認した.

仮想ノードは, その他にも最小接頭辞トライを構築する処理にも利用可能であると思われる. 今後は, それを実現して仮想ノードの応用範囲を広げて行きたいと考えている. また, 本手法で実現した LOUDS トライを用いることで, 空間効率が高い分散型のキー・バリュー型データストアを開発したいと考えている.

参 考 文 献

1) Jun-ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, Vol. 15, pp. 1066–1077, September 1989.
2) Apache Software Foundation. *The Apache HBase Book*, October 2010.
3) Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Vol. 13, pp. 422–426, July 1970.
4) Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, Vol. 26, pp. 4:1–4:26, June 2008.
5) Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pp. 205–220. ACM, 2007.
6) O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the louds succinct tree representation. In *Proceedings of the 5th International Workshop on Experimental Algorithms. Lecture Notes in Computer Science*, Vol. 4007/2006, pp. 134–145. Springer-Verlag, 2006.
7) Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, Vol. 8, pp. 281–293, June 2000.
8) Edward Fredkin. Trie memory. *Communications of the ACM*, Vol. 3, pp. 490–499, September 1960.
9) Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988. AAI8918056.
10) Guy Joseph Jacobson. Space-efficient static

trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pp. 549–554. IEEE Computer Society, 1989.

11) David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pp. 654–663. ACM, 1997.
12) 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.
13) The kumofs Project. kumofs: Extremely fast and scalable distributed key-value store. http://kumofs.sourceforge.net/.
14) Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, pp. 35–40, April 2010.
15) J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of 38th Annual Symposium on Foundations of Computer Science*, pp. 118 –126, oct 1997.
16) National highway traffic safety administration, http://www.nhtsa.gov/.
17) roma prj. Roma: A distributed key-value store in ruby. http://code.google.com/p/roma-prj/.
18) Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pp. 134–149. Society for Industrial and Applied Mathematics, 2010.
19) Issei Sato and Hiroshi Nakagawa. Succinct semi-structured data mining based on FREQT. *DBSJ Journal*, Vol. 9, No. 1, pp. 76–81, June 2010.
20) 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とブルームフィルタによる検索性能の向上. 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, July 2011.
21) 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.

付 録 A

仮想ノードを利用せずに, 2 つのトライ木 $\langle T_1, T_2 \rangle$ をマージしながら幅優先走査するプログラムを以下に示す.

Require: $r_1 \leftarrow$ root nood of $T_1, r_2 \leftarrow$ root node of T_2
  $q \leftarrow \{\langle r_1, r_2 \rangle\}$
  visit$(\langle \emptyset, \emptyset \rangle, \langle r_1, r_2 \rangle)$
  while $|q| \neq 0$ **do**
    $\langle p_1, p_2 \rangle \leftarrow q$.dequeue()
    if $p_1 \neq \emptyset \wedge p_2 \neq \emptyset$ **then**
      if alphabet(p_1) = alphabet(p_2) **then**
        $\langle n_1, n_2 \rangle \leftarrow \langle$firstChild$(p_1)$, firstChild$(p_2)\rangle$
      else if alphabet(p_1) < alphabet(p_2) **then**
        $\langle n_1, n_2 \rangle \leftarrow \langle$firstChild$(p_1), \emptyset\rangle$
      else if alphabet(p_1) > alphabet(p_2) **then**
        $\langle n_1, n_2 \rangle \leftarrow \langle \emptyset,$ firstChild$(p_2)\rangle$
      end if
    else if $p_1 \neq \emptyset \wedge p_2 = \emptyset$ **then**
      $\langle n_1, n_2 \rangle \leftarrow \langle$firstChild$(p_1), \emptyset\rangle$
    else if $p_1 = \emptyset \wedge p_2 \neq \emptyset$ **then**
      $\langle n_1, n_2 \rangle \leftarrow \langle \emptyset,$ firstChild$(p_2)\rangle$
    end if
    while $n_1 \neq \emptyset \vee n_2 \neq \emptyset$ **do**
      q.enqueue$(\langle n_1, n_2 \rangle)$
      visit$(\langle p_1, p_2 \rangle, \langle n_1, n_2 \rangle)$
      if $n_1 \neq \emptyset \wedge n_2 \neq \emptyset$ **then**
        if alphabet(n_1) = alphabet(n_2) **then**
          $\langle n_1, n_2 \rangle \leftarrow \langle$sibling$(n_1),$ sibling$(n_2)\rangle$
        else if alphabet(n_1) < alphabet(n_2) **then**
          $\langle n_1, n_2 \rangle \leftarrow \langle$sibling$(n_1), n_2\rangle$
        else if alphabet(n_1) > alphabet(n_2) **then**
          $\langle n_1, n_2 \rangle \leftarrow \langle n_1,$ sibling$(n_2)\rangle$
        end if
      else if $n_1 \neq \emptyset \wedge n_2 = \emptyset$ **then**
        $\langle n_1, n_2 \rangle \leftarrow \langle$sibling$(n_1), \emptyset\rangle$
      else if $n_1 = \emptyset \wedge n_2 \neq \emptyset$ **then**
        $\langle n_1, n_2 \rangle \leftarrow \langle \emptyset,$ sibling$(n_2)\rangle$
      end if
    end while
  end while