

文字列をキーとするインメモリデータストアのための
コンパクトなデータ構造の研究

2017年 3月

小柳 光生

文字列をキーとするインメモリデータストアの ためのコンパクトなデータ構造の研究

小柳 光生

システム情報工学研究科

筑波大学

2017年3月

概要

World Wide Web (WWW), ソーシャルネットワーク, 交通事故情報など, テキストデータから文字列が抽出される時, その出現頻度分布が少数高頻度のものと多数低頻度のものに分かれる例が多くある. 本論文では, そのような出現頻度分布に大きな偏りのある文字列をキーとしてインメモリデータストアに格納するための手法について述べる. その特徴は, 高速なデータ構造とコンパクトなデータ構造の2種類を組み合わせ, コンパクトでありながら高いスループットを実現した点にある. 具体的には, 高速なハッシュテーブルに少数高頻度のキーを, コンパクトな Level-Order Unary Degree Sequence によるトライ (LOUDS トライ) に多数低頻度のキーを格納する.

LOUDS トライはコンパクトであるが, 静的なデータ構造であるためデータを追加できない問題がある. 本研究ではログ構造マージ木と類似の手法によってこの問題を解決する. この手法を Online LOUDS Trie Version 1 (OLT1) と呼ぶ. OLT1 では, 入力されるキーと値を二分探索トライによるバッファに蓄え, それが満たされたたびに LOUDS トライを作成する.

OLT1 では文字列キーを検索する時, LOUDS トライのリストを順に調べる必要がある. リストが長くなると検索が遅くなるため, 複数の LOUDS トライを1つにマージしてリストを短く保つ. しかし, 頻繁なマージは逆に性能を低下させる. 本研究では各 LOUDS トライにブルームフィルタを付加することでこの問題を解決する.

OLT1 で高いメモリ効率を達成するにはマージ時に中間バッファの使用を避ける必要がある. しかし中間バッファを使わないマージのアルゴリズムは複雑である. 本研究では, 仮想ノードという概念を導入し, 複数の LOUDS トライを仮想的に1つの LOUDS トライとして扱えるようにする. これにより, 簡潔なアルゴリズムでマージを可能にし, さらにブルームフィルタの構築も並行で行うようになる.

本研究では, ハッシュテーブルとオンライン LOUDS トライという2種のデータ構造を組み合わせる. また, オンライン LOUDS トライの中では, 二分探索トライを LOUDS トライを構築するためのバッファとして用いる. これらのデータ構造に対してどのようにメモリを配分するかを最適化問題としてオンラインで解くことは難しい. 本研究ではこの問題を自律的なメモリ割り当てにより解決する. この手法を, Online LOUDS Trie Version 2 (OLT2) と呼ぶ. OLT2 では, 残された使用可能メモリのサイズを見ながら, 二分探索トライ, および LOUDS トライに割り当てるメモリを決定する. 残りのメモリは全てハッシュテーブルに割り当て, 全体の性能を向上させる.

本研究では, Java 言語により提案した方式に従いインメモリデータストアのためのコンパクトなデータ構造を実現した. 実現したデータ構造は, 全文検索エンジン Apache Lucene に組み込んで利用可能である. 実現したデータ構造に, 3つのデータセット (交通事故情報, 日本の家電商品等に関する掲示板, および英国のニュースサイト) から抽出したデータを入力し性能を測定した. 性能測定に用いたプログラムは, 全文検索エ

ンジンのために索引をつくるアプリケーションを想定したものである。その結果、提案手法 OLT1 では、既存の代表的な手法であるハッシュテーブルやダブルアレイトライよりもコンパクトであり、より多くのキーと値を保持できることを確認した。さらに、提案手法 OLT2 では、ダブルアレイトライよりも高いスループットを得ることができた。これにより、提案手法 OLT2 は、限られたメモリでハッシュテーブルに入りきらないような多くの件数のキーを保持しなければならない時に、これら既存のどの手法よりも高いスループットを実現できることを確認した。

目 次

第 1 章 はじめに	1
第 2 章 関連する研究	9
2.1 インメモリデータストア	9
2.2 ハッシュテーブル	10
2.3 木構造	10
2.4 簡潔データ構造	12
2.5 ブルームフィルタ	13
2.6 ログ構造マージ木	13
2.7 フラッシュストレージ上のキーバリューストア	14
2.8 Apache Spark	16
2.9 Succinct	16
2.10 ログ構造によるメモリ割り当ての効率化	17
2.11 データ圧縮によるメモリ効率の改善	17
第 3 章 想定するアプリケーションと実現するインメモリデータストアの要件	19
3.1 テキスト解析システム	19
3.2 Apache Lucene によるテキスト解析システムの問題	21
3.3 本研究で実現するインメモリデータストアの要件	22
第 4 章 オンライン LOUDS トライ構築手法 OLT1	25
4.1 OLT1 の基本概念	25
4.2 コンパクトなデータ構造の性能と選定	28
4.3 オンライン LOUDS トライ	29
4.3.1 バッファリングと構築	31
4.3.2 検索	31
4.3.3 マージ	32
4.3.4 トライ木の幅優先走査	32
4.3.5 LOUDS トライの構築	33
4.3.6 二分探索トライによるバッファトライの実装	34

4.3.7	バッファトライのサイズとオンライン LOUDS トライの性能	35
4.3.8	マージ時に一時的に使用されるメモリ	35
4.4	ブルームフィルタによる検索の高速化	36
4.4.1	ブルームフィルタのサイズ	37
4.4.2	ブルームフィルタの性能	37
4.4.3	モデルによるブルームフィルタの検索性能の見積もり	37
4.4.4	ブルームフィルタのハッシュ値の計算方法	40
4.5	仮想ノードによる LOUDS トライのマージ	40
4.5.1	中間バッファを用いない複雑なマージ	41
4.5.2	仮想ノードによるマージ	42
4.5.3	仮想ノードの操作	43
4.6	オンライン LODUS トライ単体の性能	44
4.7	高速なデータ構造と組み合わせた OLT1 全体の動作	44
4.8	モデルによる OLT1 の性能評価	46
4.8.1	スループットのモデル	46
4.8.2	OLT1 が全ての文字列をハッシュテーブルに格納できる範囲	51
4.8.3	OLT1 と他のデータ構造および二次記憶を用いる方法との比較	51
4.8.4	一様分布に対する OLT1 の性能	54
4.8.5	ステップ分布に対する OLT1 の性能	54
4.8.6	OLT1 の有効な範囲	57
4.9	マルチスレッド化に関する考察	58
第 5 章	自律的メモリ割り当て機能を持つオンライン LOUDS トライ構築手法 OLT2	60
5.1	OLT2 の自律的メモリ割り当てアルゴリズム	60
5.2	Put 操作	62
5.3	Get 操作	63
5.4	変換モードへの遷移条件とメモリ見積もり	63
第 6 章	評価実験	67
6.1	実験環境	67
6.2	実験方法	68
6.3	評価実験で用いる使用可能メモリ量	69
6.4	オンライン LOUDS トライ単体の性能	69
6.5	OLT1 の評価実験	70
6.6	オンライン LOUDS トライ構築の観測	72
6.7	OLT1 の使用可能メモリとスループットの関係	74

6.8 OLT1 によるより大きなデータの処理	78
6.9 バッファトライのサイズとオンライン LOUDS トライの性能	78
6.10 オンライン LOUDS トライのメモリ内訳	80
6.11 ブルームフィルタによる検索性能の向上	81
6.12 アプリケーションを想定したブルームフィルタの効果測定	84
6.13 ブルームフィルタ構築時間の計測	86
6.14 OLT2 の評価実験	88
6.15 自律的メモリ割り当ての観測	88
6.16 OLT2 の使用可能メモリとスループットの関係	94
6.17 OLT2 によるより大きなデータの処理	94
6.18 キャッシュ置き換えアルゴリズムに関する考察	95
第7章 おわりに	97
謝辞	100
参考文献	101
付録A 既存のインメモリデータストア	110
付録B Apache Spark	113
公表論文のリスト	114

図 目 次

1.1	3つのデータセットから抽出された文字列の出現頻度分布(累積度数)	3
1.2	メモリが限られている場合, 格納すべき文字列の数とスループット最大のデータ構造の関係(従来方式)	6
1.3	メモリが限られている場合, 格納すべきキーの数とスループット最大のデータ構造の関係(提案方式)	7
3.1	テキストデータから抽出されたフレーズの例(NHTSA)	20
3.2	Apache Lucene の Taxonomy Writer Cache インタフェース	22
4.1	高速なデータ構造とコンパクトなデータ構造の組み合わせ方法	26
4.2	従来手法: ハッシュテーブル	27
4.3	従来手法: ダブルアレイトライ	27
4.4	提案手法: OLT1(2種類のデータ構造の組み合わせ)	27
4.5	オンライン LOUDS トライ	31
4.6	LOUDS トライの構築例	33
4.7	二分探索トライの例	34
4.8	バッファサイズとスループットの関係	35
4.9	ブルームフィルタ付きオンライン LOUDS トライ	36
4.10	LOUDS トライの個数と検索速度の関係(モデル)	39
4.11	オンライン LOUDS トライ構築手法 OLT1	45
4.12	ジップ分布の累積相対度数	53
4.13	格納する文字列数に対するスループット(ジップ分布)	53
4.14	一様分布の累積相対度数	55
4.15	格納する文字列数に対するスループット(一様分布)	55
4.16	ステップ分布の累積相対度数	56
4.17	格納する文字列数に対するスループット(ステップ分布 $T = 0.1$)	56
4.18	格納する文字列数に対するスループット(ステップ分布 $T = 0.4$)	56
4.19	OLT1 のマルチスレッド化	59
5.1	自律的メモリ割り当て機能を持つオンライン LOUDS トライ構築手法 OLT2	61

5.2 OLT2 の自律的なメモリ割り当てのステップ	62
6.1 OLT1 の格納文字列数とスループットの関係 (NHTSA 2.5 億件, 89MB)	73
6.2 OLT1 の格納文字列数とスループットの関係 (kakaku.com 15 億件, 269MB)	73
6.3 OLT1 の格納文字列数とスループットの関係 (reuters.com 10 億件, 101MB)	73
6.4 OLT1 の使用メモリ量の推移 (NHTSA 2.5 億件, 89MB)	75
6.5 OLT1 のスループットの推移 (NHTSA 2.5 億件, 89MB)	75
6.6 OLT1 の使用メモリ量の推移 (kakaku.com 15 億件, 269MB)	76
6.7 OLT1 のスループットの推移 (kakaku.com 15 億件, 269MB)	76
6.8 OLT1 の使用メモリ量の推移 (reuters.com 10 億件, 101MB)	77
6.9 OLT1 のスループットの推移 (reuters.com 10 億件, 101MB)	77
6.10 OLT1 の使用メモリ量の推移 (Wikipeida-en 618 億件, 8GB)	79
6.11 バッファトライのサイズとオンライン LOUDS トライ単体の性能の関係 (x はバッファサイズが 4 万個のときのスループット)	80
6.12 オンライン LOUDS トライ単体のメモリ内訳	82
6.13 LOUDS トライに格納するキー数とブルームフィルタが占める割合	82
6.14 ブルームフィルタ付き LOUDS トライのハッシュ多重度と検索性能	83
6.15 LOUDS トライの個数と検索速度の関係 (実験結果)	84
6.16 ブルームフィルタによる LOUDS トライ構築と検索の高速化 (NHTSA)	85
6.17 LOUDS とブルームフィルタの並行構築による LOUDS 構築の高速化	86
6.18 LOUDS とブルームフィルタの並行構築による LOUDS 構築と検索の高速化	87
6.19 OLT2 の格納文字列数とスループットの関係 (NHTSA 2.5 億件, 89MB)	89
6.20 OLT2 の格納文字列数とスループットの関係 (kakaku.com 15 億件, 269MB)	89
6.21 OLT2 の格納文字列数とスループットの関係 (reuters.com 10 億件, 101MB)	89
6.22 OLT2 の使用メモリ量の推移 (NHTSA 2.5 億件, 89MB)	91
6.23 OLT2 のスループットの変化 (NHTSA 2.5 億件, 89MB)	91
6.24 OLT2 の使用メモリ量の変化 (kakaku.com 15 億件, 269MB)	92
6.25 OLT2 のスループットの変化 (kakaku.com 15 億件, 269MB)	92
6.26 OLT2 の使用メモリ量の変化 (reuters.com 10 億件, 101MB)	93
6.27 OLT2 のスループットの変化 (reuters.com 10 億件, 101MB)	93
6.28 OLT2 の使用メモリ量の推移 (Wikipeida-en 618 億件, 使用可能メモリ 8GB)	95

表 目 次

1.1	ジップの法則に従う 3つのデータセット	3
4.1	ハッシュテーブルの必要メモリサイズと put と get のスループット	30
4.2	動的トライ木の必要メモリサイズと put と get のスループット	30
4.3	簡潔順序木によるトライ木 (C++) の必要メモリサイズと put と get のスループット	30
4.4	ブルームフィルタの必要メモリサイズと put と get のスループット	38
4.5	仮想ノードの操作	43
4.6	ハッシュテーブル, 静的 LOUDS トライ, およびオンライン LOUDS トライの使用メモリ量とスループット (put, get)	44
4.7	想定するデータストアとパラメータ	48
6.1	評価実験で用いる使用可能メモリ量	69
6.2	提案したコンパクトなデータ構造オンライン LOUDS トライと既存の手法の使用メモリ量とスループットの比較	70
6.3	OLT1 の使用メモリ量とスループットの関係	74
6.4	英語版 Wikipedia から作成したデータセット	79
6.5	OLT2 の使用メモリ量とスループットの関係	94
A.1	代表的なインメモリデータストアの使用メモリ量とスループット	112
B.1	Apache Spark の使用メモリ量とスループット	113

第1章 はじめに

近年, 大規模な文字列データを解析する需要が高まっている. World Wide Web (WWW), ソーシャルネットワーク, 小売りのバスケットデータ, 交通事故情報など, 多くの分野で文字列からなる大規模なデータが作り出されている [46]. 例えば、Google や Bing などの大手検索サイトに索引される Web ページ総数は 2015 年 9 月時点には 470 億件と推定されている [81]. マイクロブログサービスには 2015 年には毎日 5 億件のメッセージが投稿された [76]. 小売り大手 Wal-Mart のデータベースには, 買い物客のバスケットデータが 1 時間あたり 100 万件追加されている [46]. Wikipedia は Web 上で編集されている巨大な百科事典であり, そのページ総数は 2015 年に 3,600 万件を超えた [79]. 米国高速道路安全局 (NHTSA) [57] の収集する交通事故データベースには, 毎年約 3 万件の事故データが追加されている. これらのデータは自然言語による情報であり, 計算機上では文字列として処理される.

このような大規模な文字列データの解析にはデータストアが重要な役割を果たす. そのようなデータを解析するときには, 一度, 何らかのデータストアに格納し, 検索や集計を行えるようにする. 文字列データの解析では, アプリケーションの要求に特化した様々な種類のデータストアが実装され利用されてきた. たとえば小売のバスケットデータの解析では表のカラムごとにスキャンして集計結果を得る処理が行われることから, Amazon RedShift [3] のような列指向データベースが使用される. Apache Lucene [22] のような全文検索エンジンでは, 文書番号に関する集合演算を高速に行う必要があるため, 転置インデックスのような集合演算に特化した値の格納方法が用いられている. XML や JSON のような半構造データを扱う場合には, MongoDB [51] のような文書指向データベースが利用される. 文字列の重複排除やカウンティングでは, 多数の文字列を格納して, 指定された文字列に対応する値を高速に取り出す機能に特化したキーバリューストアがよく用いられる. 本研究では文字列をキーとし, `get(key)` と `put(key, value)` の操作を提供するキーバリューストアを扱う.

本研究では次のようなテキスト解析アプリケーションの要求に特化したキーバリューストアを実現することを目指す. テキスト解析では, 索引を作成する際, 分析対象とするキーワードを抽出するルールを与える. このルールは分析ごとに変更される. そして, 変更のたびに索引が再作成される. 抽出された文字列は, キーバリューストアに一時的に格納され, 索引作成時に参照される. この時, すでに導入されている計算機を繰

り返し利用し, その主記憶のサイズは固定で拡張しない.

本研究では, 具体的には以下のようなアプリケーションの要求に特化したキーバリューストアを実現する:

操作のインターフェースと意味 格納 (put) された文字列が即座に取得 (get) 可能になる, 動的更新が可能である.

キーと値の長さ キーが長く (数十バイト), 値が短い (4 バイトの整数).

キーの出現頻度 キーの出現頻度が少数の高頻度のものと多数の低頻度のものに偏っている. ただし, どのキーの出現頻度が高いかは事前には不明である.

上記の動的更新の要求は, 重複排除を行うアプリケーションの要件から来ている. 重複排除では, すべての文字列を保持する辞書が作成され, 各文字列に一意の番号が割り当てられる. 文字列に一意の番号を割り当てるには, 出現するすべての文字列に対して, すでに割り当たが行われたかどうかを調べる get の操作を行う. まだ割り当たが行われていない新たな単語であった時に限り, 新しい番号を割り当てる操作である put が行われる.

本研究で想定するアプリケーションでは, キーの型には文字列型やバイト配列型が用いられる. また, 番号を格納するために, 値の型には 4 バイトの整数などのキーと比較して短いものが用いられる. このとき, 全体のサイズに対してキーの占める割合が最も大きいため, コンパクトさを考える際には, キーを格納するメモリ効率を向上することが最も重要である.

本研究で想定するアプリケーションでは, データストアには重複を含む文字列が繰り返し get されるが, その出現頻度は文字列によって異なる. 冒頭にあげた多くの文字列データで, これら文字列の出現頻度が大きく偏っていることが知られている. 本研究で扱うアプリケーションでは, 文字列の出現頻度が少数の高頻度のものと, 多数の低頻度のものに分かれることを仮定する. そのような出現頻度分布の 1 つにジップの法則 [89] に従う分布 (ジップ分布) がある.

実際の文書から抽出される文字列の出現頻度分布は非常によくジップの法則に従う. 例えば, 表 1.1 に示すデータから抽出した文字列の出現頻度分布 (累積度数) を図 1.1 に示す. この図の横軸は文字列を出現頻度順に並べた際のランク, 縦軸は出現頻度の累積度数を表している. それぞれの単位は, 横軸は最大ランク (重複を除いた文字列の数) を 1 とする比率, 縦軸は全文字列の出現数の合計 (重複を含む入力文字列数) を 1 とする比率である. これらのデータはそれぞれ性質も抽出ルールも異なるが, そのような条件の違いにかかわらずほぼジップの法則に従った出現頻度分布を示す. 表 1.1 のデータセットの内容は以下の通りである:

NHTSA 米国高速道路安全局 (National Highway Traffic Safety Administration) の収集

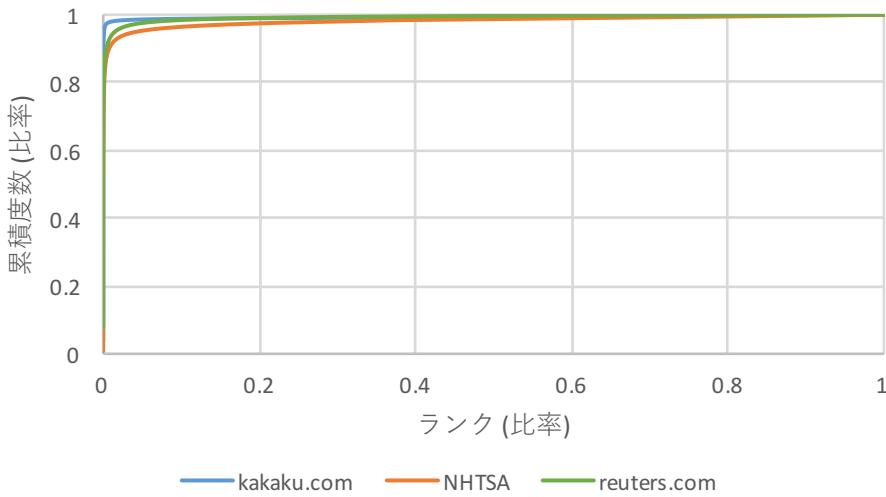


図 1.1: 3 つのデータセットから抽出された文字列の出現頻度分布(累積度数)

した交通事故データベース [57] から自然言語処理によって抽出されたキーワードの列.

kakaku.com 大規模情報コンテンツ時代の高度 ICT 専門職業人育成事業 (DCON) [75] により提供された、日本の家電商品等に関する掲示板の HTML データから形態素解析によって抽出された単語の列.

reuters.com 同じく DCON によって提供された、英国のニュースサイトの HTML データから空白を区切りとして抽出された単語の列.

本論文では、このような出現頻度分布を示す文字列データを格納することに特化したコンパクトなデータ構造を提案する。このデータ構造は、主記憶上に全てのデータ

表 1.1: ジップの法則に従う 3 つのデータセット

	NHTSA	kakaku.com	reuters.com
抽出された文字列の総数	2.5 億件	15 億件	10 億件
抽出されたデータサイズ	6.4 GB	9.4 GB	7.2 GB
抽出された文字列の平均長	29.0 バイト	6.69 バイト	7.38 バイト
重複を除いた文字列の数	650 万件	1930 万件	640 万件
重複を除いたデータサイズ	222 MB	340 MB	114 MB
重複を除いた文字列の平均長	35.5 バイト	18.4 バイト	18.6 バイト
言語	英語	日本語	英語

を格納するキーバリューストアとして利用できる。本論文では、キーバリューストアのうち主記憶上に全てのデータを格納するものをインメモリデータストアと呼ぶ。インメモリデータストアは主記憶の持つ高いランダムアクセス性能を背景に非常に高いスループットを実現できる。しかし、計算機の搭載する主記憶のサイズには制限があるため、一定以上の数の文字列はインメモリデータストアには入りきらない。

既存のインメモリデータストアには、Memcached [49], Redis [71]、および Infinispan [10] などがある。いずれもハッシュテーブルを内部のデータ構造として用い、高いスループットで動作する。しかしながら、それらのメモリ効率は高くない。例えば、表 1.1 のデータセットを格納した場合、それらのデータサイズの約 1.5 倍から 7 倍のメモリを使用する。詳しくは付録 A に示す。

インメモリデータストアは、全文検索エンジンライブラリ Apache Lucene [22] でも利用されている。Apache Lucene は主記憶上にハッシュテーブルを作成して辞書を格納する。しかし、ハッシュテーブル全体を主記憶に保持できないほど文字列の件数が多い場合は、二次記憶に辞書を置き、ハッシュテーブルは Least Recently Used (LRU) アルゴリズムによるキャッシュとして使用する。ただし、二次記憶を利用すると性能は著しく低下する。

同様の例に、Apache Spark [88] がある。Apache Spark は Resilient Distributed Dataset (RDD) [87] と呼ばれる抽象データ構造を用いて MapReduce [15] などの処理を行う分散システムである。これらの処理を高速に行うため、RDD は計算結果を主記憶に保持することができる。しかし、計算結果がメモリに入りきらない時には、格納するデータの一部、もしくは全部を二次記憶に保持する。ただし、二次記憶を利用すると性能は著しく低下する。

本論文では、1 台の計算機上で動作するインメモリデータストアのメモリ効率について議論する。格納できる文字列の数を増やすために分散インメモリデータストアを用いることが考えられるが、本論文では議論しない。ただし、分散インメモリデータストアを用いる場合でも、計算機あたりのメモリ効率の重要さは変わらない。キーの集合を分割して複数の計算機に分散すれば、1 台の計算機に搭載できる主記憶量に制限されず、使用する計算機台数に応じたキーの数を格納できるようになる。このとき、メモリ効率の低いインメモリデータストアを使用すると、計算機 1 台あたりに格納できるキーの数が少なくなり、必要な計算機台数は多くなる。これに対し、仮にメモリ効率が 10 倍のインメモリデータストアを用いれば、計算機台数は 10 分の 1 に減り、コストは 10 分の 1 になる。これは、Amazon Elastic Computing Cloud (EC2) [73] や Google Compute Engine [29] など、多くのクラウドコンピューティングサービスが、インスタンス台数に比例した料金体系になっていることから、計算機資源をクラウドコンピューティングサービスから調達する場合でも成り立つ。

一般に、インメモリデータストアを実現するデータ構造はコンパクトさとスループットの高さのトレードオフになっている。高速さが求められるインメモリデータ

ストアの実装には、高速な検索が可能なハッシュテーブルが最も多く使われている [16–18, 44, 45, 49, 62, 71]。文字列をキーとして扱う分野ではトライ木 [24] もよく使われる [40] [86]。トライ木は文字列の接頭辞を共有する構造を持つので、一般にハッシュテーブルよりもメモリ効率が高いが、検索性能はハッシュテーブルに及ばない。トライの実装として、枝の選択に二分探索を用いる二分探索トライや、ダブルアレイトライ [5] などがあげられる。

データを格納する際、メモリ効率を上げるためにハフマン符号などのエントロピーコード化や、Lempel-Ziv [90] などの辞書式圧縮が使われることもある。しかし、これらの手法では圧縮したまま検索(get) ができない。実際に、データ構造を分割してそれぞれ圧縮し、get のたびに伸長する手法を用いると、スループットは著しく低下する。

メモリ効率が高く、検索も可能な文字列の格納手段として、トライ木の実装に簡潔データ構造 [31] を用いる方法がある。簡潔データ構造は理論上最小限に近いビット数で表すことができる静的データ構造である。順序木を表す簡潔データ構造(簡潔順序木)には、Level-Order Unary Degree Sequence (LOUDS) [32] や括弧表記 (Balanced Parentheses, BP) [53], Depth-First Unary Degree Sequence (DFUDS) [8] などがある。簡潔順序木を用いることで、データ構造としては理論上最小限に近いメモリ使用量で文字列のキーと値を格納するトライ木を実現することができる。例えば、Google IME [30] では日本語変換辞書のサイズを小さく抑えるために LOUDS を用いている。圧縮技術とは異なり、簡潔データ構造に対する操作はデータを元の形式に展開することなく実現できる。このため、簡潔データ構造は圧縮技術と比較して高速な検索が可能である。

簡潔データ構造は、ハッシュテーブルや動的トライ木に検索速度は及ばないが、高いメモリ効率を実現できる技術である。しかし、格納すべきデータが全て揃うまでデータ構造を作成できないため、動的更新を求める本研究の用途には使用できない。動的簡潔順序木 [68] と呼ばれる動的更新を可能にする手法が提案されている。しかし、動的簡潔順序木の基となる BP は LOUDS などの他の簡潔順序木と比較すると get の性能が低い。そのため、動的簡潔順序木も同様に get の性能が低く、高いスループットが求められるインメモリデータストアのデータ構造には適さない。

このように、従来スループットとメモリ効率の関係はトレードオフになっている。この様子を図 1.2 に示す。図の横軸は格納する文字列の数、縦軸はスループットを表している。格納すべき文字列の数が少ない場合はハッシュテーブルが使用できるが、文字列の数が多い場合はメモリに入りきらなくなるので、スループットを犠牲にして、ダブルアレイトライのような、よりコンパクトなデータ構造を選択する。ただし、ダブルアレイトライでもハッシュテーブルの 2 倍程度の文字列の数しか格納できない。そのため、文字列の数がその上限を超える場合はインメモリデータストアを利用することができない。その場合、主記憶は LRU などのアルゴリズムを用いてキャッシュとして利用し、データを二次記憶に格納することになる。その結果スループットは著しく低下する。

本研究では、入力される文字列が、少数の非常に出現頻度の高い文字列と、多数の非

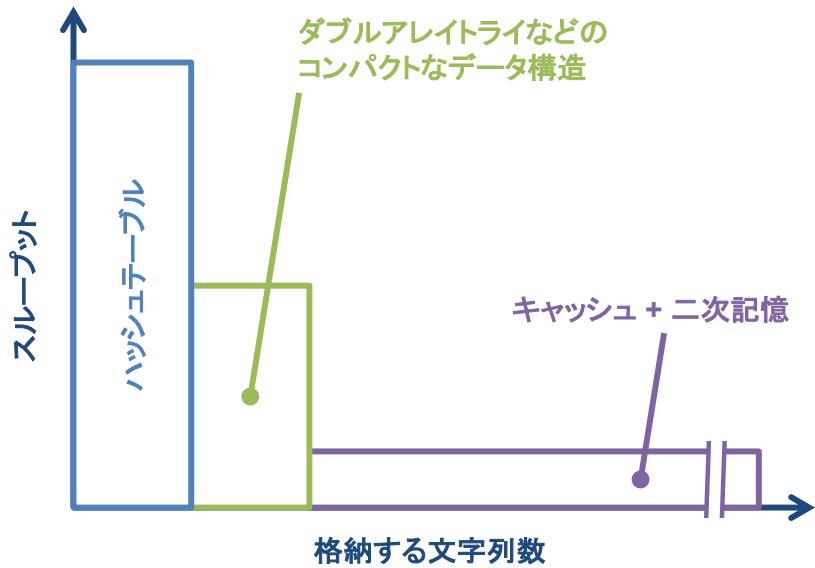


図 1.2: メモリが限られている場合, 格納すべき文字列の数とスループット最大のデータ構造の関係(従来方式)

常に出現頻度の低い文字列の 2 通りに分けられることを利用する. 具体的には, 少数の非常に出現頻度の高い文字列を非常に高速なデータ構造に, 多数の出現頻度の低い文字列を, 非常にメモリ効率の高い, コンパクトなデータ構造に格納する手法を提案する.

この手法により, 図 1.2 は, 図 1.3 に書き換えられる. 高速なデータ構造とコンパクトなデータ構造の割合を格納すべき文字列の数に応じて決めることで, 全ての文字列を主記憶に保持できる範囲を広げる. そして, 高速なデータ構造により多くのメモリを割り当てることで, より高いスループットを実現する. 文字列の数がコンパクトなデータ構造でもメモリに格納できないほど多い場合は, 提案手法も利用できなくなる.

本研究では, 高速なデータ構造としてハッシュテーブルを使用する. ハッシュテーブルは高速だがメモリ効率の低いデータ構造である.

本研究では, コンパクトなデータ構造として, LOUDS を用いてトライ木を実現した LOUDS トライを用いることを提案する. LOUDS トライは一度構築すると変更できない静的データ構造である. そのため, 本研究では, プログラムの実行中に, 新しい文字列と値の組を含む LOUDS トライを繰り返し構築する方法を提案する. これによりあたかも LOUDS トライに新しい文字列と値の組を追加できるかのように見せかける. 本研究では, この構築手法を LOUDS トライのオンライン構築と呼ぶ. LOUDS トライのオンライン構築では, 追加される文字列と値をバッファに蓄え, LOUDS トライを作成する.

本研究では, ハッシュテーブルと LOUDS トライを用いる手法の全体を Online LOUDS

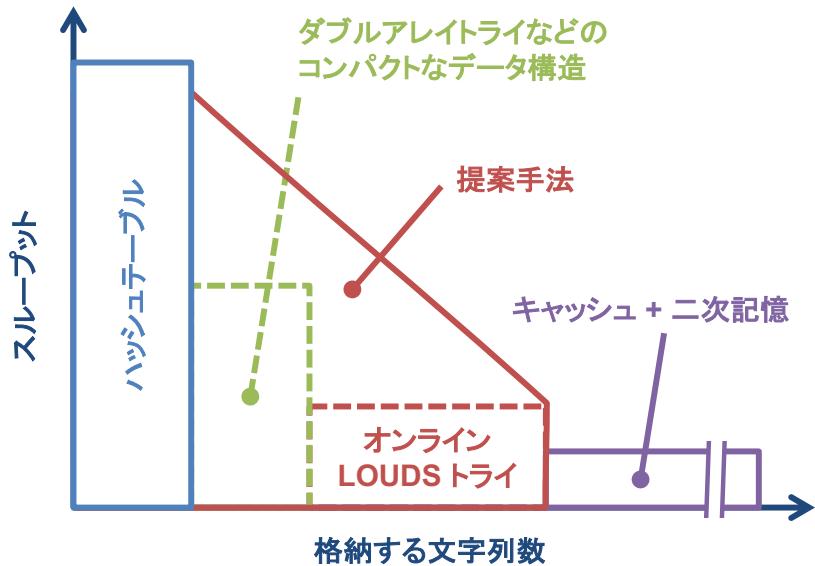


図 1.3: メモリが限られている場合、格納すべきキーの数とスループット最大のデータ構造の関係(提案方式)

Trie Version 1 (OLT1) [37] と呼ぶ。OLT1 では、LOUDS トライ構築に必要な幅優先走査に適した二分探索トライをバッファに用いる。順次構築される LOUDS トライはリストに追加され、検索時には、そのリストの新しいものから順に検索して最初に見つかった要素が返される。リストが長くなると検索時間が長くなる。そこで、OLT1 ではリスト中の LOUDS トライをログ構造マージ木 [58] と同様の手法でマージする。従来のログ構造マージ木とは異なり、OLT1 では、データの書き込み先が主記憶であり、書き込まれる木構造が LOUDS トライである。

OLT1 ではマージ回数を減らすことも性能向上には重要である。ただし、既に述べたように、LOUDS トライを格納するリストが長くなると検索性能が落ちるため、単により多くの LOUDS トライを一度にマージすることでマージ回数を減らす方法では性能が低下してしまう。この問題を解決するため、OLT1 では各 LOUDS トライに対応したブルームフィルタ [9] を作成して、高い確率で求める文字列が含まれる LOUDS トライだけを選択して検索できるようにする [37]。

LOUDS トライをマージする単純な方法は、対象のトライ木からすべてのキーを取り出し、もう一度、二分探索トライなどの中間バッファに追加して全体を作り直すことである。しかし、この単純な方法はすべてのキーをメモリ使用量の大きい中間バッファに格納する必要があり、全体として大きなメモリ領域を必要とする。しかし、中間バッファを使わずに、マージされるすべてのトライ木を順にたどりながら、新しい LOUDS トライを構築するアルゴリズムは複雑である。

OLT1 では複数のトライ木を仮想的に合成された 1 つのトライ木として扱う手法(仮

想ノード)を用いることで、簡潔なアルゴリズムでマージを可能にする[38]。仮想ノードによって、中間バッファを使わずに、单一のトライ木から LOUDS トライを作成するアルゴリズムと全く同じものを使用して LOUDS トライをマージすることができる。仮想ノードによって、マージ後の LOUDS トライに対して、適切なサイズのブルームフィルタを作成するのも、单一のトライ木から作成するアルゴリズムと全く同じものが使える。OLT1 では、仮想ノードを利用してマージと同時にブルームフィルタを生成し、マージ時間の短縮も実現する。

OLT1 では、LOUDS トライのほか、ハッシュテーブル、LOUDS トライ構築の中間バッファとして使用する二分探索トライが用いられる。しかし、それらのデータ構造に対する最適なメモリ割当は入力データに依存しているため、オンラインで最適化問題として解くことは難しい。

この問題を解決するために、本研究では、OLT1 に加えて、与えられた使用可能メモリを見ながら自律的にメモリを配分する手法を提案する。この手法を Online LOUDS Trie Version 2 (OLT2) [39] と呼ぶ。OLT2 は、残り使用可能メモリ量から、中間バッファに割り当てるメモリ量を決定し、残りをハッシュテーブルに割り当てる。ハッシュテーブルがいっぱいになったら、その内容を中間バッファに格納し、ハッシュテーブルを破棄する。ハッシュテーブルの空き領域に LOUDS トライを作成し、中間バッファを破棄する。

本研究では、提案手法に基づき、Java 言語によりインメモリデータストアのためのコンパクトなデータ構造を実現した。実現したデータ構造は、全文検索エンジン Apache Lucene に組み込んで利用可能である。実現したデータ構造に、表 1.1 に示した 3 つのデータセットを入力し性能を測定した。性能測定に用いたプログラムは、Apache Lucene で索引をつくるアプリケーションを想定したものである。その結果、提案手法 OLT1 では、既存の代表的な手法であるハッシュテーブルやダブルアレイよりもコンパクトであり、より多くのキーと値を保持できることを確認した。この時、ブルームフィルタのメモリ使用量やスループットの向上を確認した。さらに、提案手法 OLT2 では、既存手法であるダブルアレイよりも高いスループットを得ることができた。これにより、提案手法 OLT2 は、限られたメモリでハッシュテーブルに入りきらないような多くの件数のキーを保持しなければならない時に、これら既存のどの手法よりも高いスループットを実現できることを確認した。

本論文の構成を以下に示す。2 章では、関連する研究を示す。3 章では、提案方式の典型的なアプリケーションとして、Apache Lucene を用いたテキスト解析システムと、求められるインメモリデータストアの要件を示す。4 章では、LOUDS トライのオンライン構築を実現する提案手法 OLT1 について述べる。5 章では、自律的にハッシュテーブル、LOUDS トライ、および二分探索トライのメモリ配分を決める提案手法 OLT2 について述べる。6 章では、提案手法の効果を検証する実験を行う。7 章では、本論文の結論をまとめる。

第2章 関連する研究

この章では、本研究に関連する研究について述べる。

2.1 インメモリデータストア

既存のインメモリデータストアは様々な形態で提供されている。大きくは、アプリケーションのプログラムに組み込んで使用するものと、独立したサーバ、もしくはクラスタを構成してプロセス間通信によって入出力を行うものに分類される。

Memcached [49] は独立したサーバとして動作するインメモリデータストアである。データベースの負荷を低減するために、複数の Web サーバから利用される共有キャッシュの用途で多く用いられている。内部で使用される主なデータ構造はハッシュテーブルである。Brad Fitzpatrick によって 2003 年に開発が始まった。Memcached には、キーが存在しないときにだけキーと値を書き込む機能が提供されている。また、Spymemcached [70] などのライブラリにより、非同期で書き込みを行う API が提供されている。これらを組み合わせることによって、クライアント側で呼び出しの帰りを待つことなく重複排除を実装できる。

Redis [71] は独立したサーバとして動作するインメモリデータストアである。高可用性クラスタ (Redis Sentinel) やデータ分散クラスタ (Redis Cluster) として構成できる。複数の Web サーバから利用される共有キャッシュの用途で多く用いられている。内部で使用される主なデータ構造はハッシュテーブルである。値にリストやマップなど、いくつかのデータ構造を使用できる。Salvatore Sanfilippo によって 2009 年から開発が始まった。Pipeline と呼ばれるコマンドを連続して処理する API が提供されており、複数のキーに関する処理をまとめてサーバに送信して処理できる。また、Redis にもキーが存在しないときにだけキーと値を書き込む機能があり、Pipeline と組み合わせることで、クライアント側でキーごとの処理の帰りを待つことなく重複排除を実装できる。クライアントライブラリには Jedis [42] などがある。

Infinispan [10] は、Java による分散インメモリデータストアである。独立したサーバかクラスタ、もしくは、アプリケーションに組み込まれた形態で使用できる。クラスタ構成ではレプリケーションやデータ分割などの機能が提供される。アプリケーションサーバである Wildfly [61] に含まれており、アプリケーションサーバの共有キャッ

シユとして使用される。内部で使用される主なデータ構造はハッシュテーブルである。JBoss Cache の後継として Red Hat によって 2009 年から開発が始まった。Infinispan をアプリケーションサーバに組み込んでインプロセスのキャッシュとして構成した場合には、ネットワークのオーバーヘッドを回避できるため、サーバクライアント型の場合と比較してより高いスループットを得られる。

本論文で提案するデータ構造は、機能として `put` と `get` を提供し、アプリケーションのプログラムに組み込んで使用されるインメモリデータストアの内部実装として使用されることを想定している。本研究では、上記のインメモリデータストアが内部実装として使用するハッシュテーブルよりもコンパクトなデータ構造を提案する。

2.2 ハッシュテーブル

キーと値の関係を保持するデータ構造は、プログラミング言語では、連想配列やマップ、辞書などと呼ばれる。その実装にはハッシュテーブルが最もよく使用される。キーバリューストアの多くでも内部の実装にハッシュテーブルが用いられている [10, 16–18, 44, 45, 49, 62, 71]。そのほか、テキスト解析システムで使用される Apache Lucene [22] も内部でハッシュテーブルを用いている。

ハッシュテーブルは特に検索速度が速いが、スパースな配列によって実現されるため一般にメモリ効率が低い。カッコーハッシュ [59] やホップスコッチハッシュ [28] では、複数のハッシュ表を用いて衝突確率を下げることでデータ構造の密度を上げてメモリ効率を改善している。しかし、本研究の対象である文字列キーが長いアプリケーションでは、キーのサイズが全体のサイズの多くを占めるため、これらの改善がメモリ効率に寄与する割合は小さい。

本研究では、高速なデータ構造として、スパースな配列によって実現されるハッシュテーブルの実装を用いる。

2.3 木構造

ハッシュテーブルと並び、連想配列を実現するためによく使用されるデータ構造に平衡二分探索木がある。たとえば平衡二分探索木のひとつである赤黒木は Linux カーネル [21] の実装などで広く使われている。平衡二分探索木はキーを木構造に順序に応じて配置しながら、二分木の左右のノード数のバランスを保つ手法である。平衡二分探索木は範囲検索を効率よく実現でき、最悪時の検索性能に優れる。バランスを保つためのノードの移動が必要なこと、文字列がキーの時は検索時に各枝で文字列どうしの比較を繰り返す必要があることなどにより、キーが長い場合は後述のトライ木や 2.2 節で述べたハッシュテーブルの方が検索性能が高い。

文字列をキーとして扱う場合はトライ木 [24] が使われることもある。例えば、形態素解析エンジンである MeCab [40] ではトライ木の実装の 1 つであるダブルアレイトライ [5] が使われている。トライ木は文字列の接頭辞を共有する構造を持つので、一般にハッシュテーブルや平衡二分木よりもメモリ効率が高い。そのようなトライの例として、二分探索トライやダブルアレイトライがあげられる。

トライ木は枝に文字を対応付け、木の根から順に、キーのアルファベットに対応する枝を見つけながらノードをたどることで検索を行う。この枝を探索する操作の速さによってトライ木の検索速度が決まる。例えば、各ノードにキーに使用するアルファベットと同数の要素を持つ配列を用意し、アルファベットと配列の添え字を対応付け、配列の値に枝の情報を格納すれば、あるノードにおける枝の探索が定数時間で実行できる。したがってキーを検索する時間はキーの長さに比例した時間になる。これに対し、ハッシュテーブルの検索も、文字列キーのハッシュ値の計算にはキーの長さに比例した時間がかかるため、キーが長い場合、キーの長さに比例した検索時間となる。ハッシュテーブルにはハッシュ値の衝突がある場合に検索効率が落ちる問題があるため、衝突が頻発する場合はハッシュテーブルの方がトライ木よりも検索速度が遅い場合がある。通常は再ハッシュなどにより、ハッシュ値の衝突は低い確率に抑えられるため、ハッシュテーブルの方がトライ木よりも高速に検索できる。

ダブルアレイトライ [5] は枝の構造に配列を用いるトライ木の実装の一種である。ダブルアレイトライでは枝の構造に単純な配列を用いる方法のメモリ効率の悪さを改善する。枝を配列で表すと各ノードの文字から枝の探索は定数時間になり検索が高速になる。しかし、枝の分岐が少ないノードでは多くの空の要素を持つことになりメモリ効率が悪い。ダブルアレイトライではこれらの配列の空の要素の部分を他の配列の要素として使用することでメモリ効率を高める。重ね合わせた配列は 1 本の配列となり、この配列だけでは各要素がどのノードに属するか分からない。そのため各要素がどのノードに属するかを表すためにもう 1 本の配列が作成される。これによりメモリ効率が高く検索の速いデータ構造が得られる。

トライ木のメモリ効率を高める Directed Acyclic Word Graph (DAWG) [13, 84] が提案された。DAWG [13] は、トライ木を決定性有限オートマトン (DFA) の一種として見てノード数を最小化したデータ構造である。トライ木と同様に文字列を格納し、高速な検索が可能である。トライ木の実装では共通するキーの接頭辞を最小接頭辞木に格納し、それ以降の部分文字列を TAIL と呼ばれる別のデータ構造に格納する。通常のトライ木では TAIL は配列だが、TAIL のデータ構造を別のトライ木やハッシュテーブルにすることで、共通の部分文字列をまとめてノード数を削減できる。

本論文で提案するデータ構造は、ダブルアレイトライよりも多くの文字列を格納し、同じ文字列の数を格納する場合には、より高いスループットを実現する。

2.4 簡潔データ構造

簡潔データ構造は理論的に最小限に近いビット数で表されるデータ構造の総称である [31]. 簡潔データ構造は、データ圧縮とは異なり、伸長を必要とせずに様々な操作を行うことができる。そのような操作を実現するため、完備辞書 [36] などの全体のサイズと比較して小さなサイズの索引を附加する。この索引のサイズは簡潔データ構造に格納する要素数 n が大きいときに漸近的に定数とみなせる。このように、 n が大きいとき漸近的に定数とみなせる関数を $o(n)$ と表記する。例えば、ノード数 n の順序木を表す最小限のビット数は $2n$ であり、その簡潔データ構造は $2n + o(n)$ ビットを必要とする。このとき、ノードの数 n が大きくなると、順序木の簡潔データ構造のサイズは漸近的に $2n$ に近づいていく。

主な順序木の簡潔データ構造には LOUDS (Level Order Unary Degree Sequence) [32], BP (Balanced Parenthesis) [53], DFUDS (Depth-First Unary Degree Sequence) [8] がある。これまでに、順序木の他、順列 [52] やグラフ [20] などの簡潔データ構造が提案された。簡潔順序木 LOUDS, BP, および DFUDS によるトライ木の C++ 言語による実装がある [85]。

LOUDS は簡潔データ構造であり、 n 個のノードを持つ順序木を $2n + 1$ ビットで表現する。これに加えて $O(\log n)$ ビットの完備辞書 [36] を持つことで、あるノードの最初の子、隣のノード、アルファベットを取り出す操作を定数時間で実現できる [60]。LOUDS によるトライの実装である LOUDS トライを用いると、理論上最小限に近いメモリ使用量で文字列を格納できる。LOUDS トライは、静的データ構造であるため、一度構築するとデータを追加することができない。

順序木について、更新を含むより豊富な操作の実現方法も研究されている。動的簡潔順序木 [68] [69] は、セグメントに分割された BP と、分割されたセグメントを検索する手法を組み合わせることで、多くの操作を $O(\log n \log \log n)$ 時間で実行でき、かつ動的更新も均一時間で $O(\log n)$ で行える簡潔データ構造である。BP は木の枝を深さ優先順で並べたビット列であるため、共通接頭辞を持つ部分木が隣接したビット列に現れる。動的簡潔順序木では、このことを利用して接頭辞を共有する部分木を分割し、その単位で再構築することで動的更新を実現している。分割された部分木はその部分木に含まれるノード番号の最大値と最小値を保持する区間木の一種である区間最大最小木 [63] の葉にリンクされ、効率よくノードの検索が行われる。

LOUDS トライの検索性能と比較して、BP によるトライ木の検索性能は著しく低い。よって、BP の派生である動的簡潔順序木は BP と同様に検索性能が低い。そのため、動的更新が求められるケースでも検索性能に優れる LOUDS トライの使用が望まれるが、幅優先走査で作成される LOUDS のビット列は共通接頭辞を持つ部分木に分割することができないため、動的簡潔順序木と同様の更新方法は使用できない。

提案手法では、動的簡潔順序木とは異なり、2.6 節で説明するログ構造マージ木と同

様の方式を用いて LOUDS トライをオンライン構築する。これにより、より性能の良い LOUDS トライを動的更新が必要なインメモリデータストアの内部実装として使用できるようになる。

2.5 ブルームフィルタ

ブルームフィルタ [9] 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。ブルームフィルタの実装ではまず “0” で初期化されたビット列を用意する。キーを追加すると時、そのキーに対して複数の独立したハッシュ値を計算し、それらが示す位置のビットを “1” にする。キーの存在を調べる時、同様にハッシュ値を計算し、その位置のビットがすべて “1” ならば存在する、1つでも “0” ならば存在しないと判定する。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。ハッシュ値の個数を k 、キーの個数を N とすると、偽陽性確率を最小の $(1/2)^k$ にするビット列の長さは約 $1.44kN$ である [19]。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。しかし、保持するキーの数 N に比例した長さのビット列をあらかじめ確保しなければ精度の高いふるいとして働くかない問題がある。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。このとき、各 LOUDS トライのキーの個数に応じたサイズのブルームフィルタを作成する手法を提案する。

2.6 ログ構造マージ木

本研究ではログ構造マージ木 (Log Structured Merge Tree, LSM-Tree) [34, 58, 72] で用いられている手法と類似の手法を用いてインメモリデータストアを実現する。ログ構造マージ木は検索が必要なデータに対して高速に追記が可能なデータ構造である。

ログ構造マージ木では、HDD (Hard Disk Drive) や SSD (Solid State Drive) のようなブロックデバイスに、高いスループットでデータを書き込むために主記憶上に木構造のバッファを設け、そのバッファが使用可能メモリ使い切るたびに二次記憶に書き込みを行う。この際、複数の木構造をマージするアルゴリズムが必要になる。そのため、ログ構造マージ木が構築するデータ構造としては、マージが比較的容易でブロックデバイスでの利用に適した B 木 [7] やその亜種がよく使用される。

ログ構造マージ木のマージアルゴリズムである Stepped Merge アルゴリズム [33] は、全部の木を一度にマージせず、世代ごと（もしくはサイズごと）に段階的にマージする

ことで、ノード数 n のとき、定期的に全体をマージする場合に必要であった $O(n^2)$ 回の入出力を $O(n \log n)$ 回に減らす。BigTable [12], LevelDB [14], Apache Cassandra [41], HBase [6] など、近年のデータストアで用いられているログ構造マージ木は Stepped Merge アルゴリズムを利用している。

Stepped Merge アルゴリズムでは、二次記憶上に木構造のリストを保持し、検索時は全ての木を検索した結果を合わせて応答する。そのため、リストが長くなると検索性能は落ちる。これを避けるため、主記憶上に作成したブルームフィルタを用いて不要な検索を避ける手法が使われる [2, 14]。

本研究では LOUDS トライに適したバッファリング、マージ手法、およびブルームフィルタ構築手法を提案する。これにより、ログ構造マージ木と類似の手法で LOUDS トライのオンライン構築を行う。提案手法は、構築する対象が LOUDS トライであり、メモリ上で全ての操作が行われることがログ構造マージ木とは異なる。LOUDS トライを効率よく構築するため、バッファに幅優先操作が高速な二分探索トライを用いるなど、ログ構造マージ木にはない特徴を持つ。

2.7 フラッシュストレージ上のキーバリューストア

キーバリューストアにおいて、SSD などのフラッシュストレージの寿命の長期化や効率的な利用のために、主記憶上にメモリ効率の高い索引を構築する技術が研究されている [4, 16, 17, 44, 82]。これらの手法では、主記憶上にハッシュテーブルを配置し、ログ構造によってフラッシュストレージへの書き込みを行う。これによってフラッシュストレージへの書き込み回数が大幅に削減される。

このとき作られるハッシュテーブルを索引として主記憶に保持し、フラッシュストレージ上のブロックの探索に利用すると検索を高速化できる。また、ブルームフィルタを用いてブロックに対する不要な検索を防ぐ [16] などの工夫も行われている。このような手法を評価する場合、書き込みや検索の性能が考慮されることはもちろんだが、索引が主記憶上に作成されるため、アプリケーションと共有する主記憶にいかに負担をかけないかも重要な観点になる。

SILT (Small Index Large Table) [44] は「マルチストア」と呼ばれる 3 つのデータ構造からなる索引を持つ。入力されたキーと値の組みは、最初に、主記憶にカッコーハッシュによる索引を持ち、フラッシュストレージには高速に追記できるログ構造を持つ LogStore に格納される。一定数データが入力されると、LogStore をハッシュ順にソートすることで主記憶上のポインタを排除してコンパクト化した HashStore が作られる。HashStore は静的データ構造であり、生成後変更されない。さらに HashStore が一定数つくられると、圧縮された木構造による索引を持つ SortedStore にマージされる。SortedStore では全てのキーがアルファベット順にソートされている。このように、ロ

グ構造マージ木と類似したバッファリングとコンパクト化の繰り返しによって主記憶上の索引のメモリ効率が高く保たれる。主記憶上の索引により、キーあたり平均約 1.01 回のフラッシュストレージへのアクセスで値を取り出すことができる。

ログ構造を用いてよりコンパクトなデータ構造に変換する SILT の手法は提案手法 OLT1, および OLT2 と類似している。しかし、SILT ではフラッシュストレージ上の 1024 バイトのブロックにキーと値を保持し、キーを主記憶に保持しない。主記憶上の索引はハッシュ値やトライ木の最小接頭辞木にあたる部分だけを格納する。索引のメモリ使用量が小さく抑えられているのはこのためである。データサイズのほとんどはこの 1024 バイトのブロックであり、本研究のように、キー自身を主記憶にいかにコンパクトに格納するかについては考慮していない。

また、提案手法と比較するとスループットに 20 倍以上の差がある。このスループットの差の要因には以下があげられる：

- 検索あたり 1.01 回のフラッシュストレージへのアクセスがあること
- SortedStore のキーはハフマン符号によって圧縮されているため検索時に伸長が必要であること
- SortedStore の構築にはソートが必要であり、その計算時間は $O(n \log n)$ であること
- HashStore とのマージは固定数のキーごとに行われており、その回数は全体で $O(n^2)$ であること

LSM-Trie [82] は、二次記憶を前提とし、トライ木とブルームフィルタを索引として主記憶に配置するデータストアである。本研究と同様に値の占める割合が全体のデータサイズに対して小さいことを前提としている。

LSM-Trie は、主記憶上のトライ木の各ノードから、二次記憶上のハッシュテーブルを指す構造を持つ。トライ木はキーのハッシュ値の最小接頭辞木であり、各ノードのハッシュテーブルには、そのノードの表す接頭辞で始まるハッシュ値を持つキーが格納される。ハッシュテーブルが既定のサイズに達すると、子ノードにバケットを移動し、そのノードのハッシュテーブルは空にする。この操作は、LevelDB や SILT のコンパクト化に伴うデータ構造の再構築と比較すると、更新あたりの書き込み回数が少ない。

検索時はトライ木をルートから葉に向かって、順にハッシュテーブルを調べてキーを探索する。また、各ノードにブルームフィルタを持ち、キーが含まれていないハッシュテーブルのアクセスをほとんどスキップできる。これらの仕組みにより、キーあたり約 1.01 回の二次記憶の読み取りでキーから値を取得できる。ただし、全てのブルームフィルタを主記憶に配置すると、最下層のブルームフィルタが使用メモリの大部分を占めることになる。そのため、LSM-Trie では、トライ木の最下層のブルームフィルタ

を二次記憶に配置する (Bloom Cluster). これらの仕組みにより, 約 2 回強の二次記憶へのアクセスで値を取得できる.

LSM-Trie が使用する技術は本研究の使用する技術と類似しているが, 使用メモリ量の問題に対する解法が二次記憶を前提としている点が大きく異なる. LSM-Trie の使用メモリ量が少ない主な理由は, ブルームフィルタのほとんどを二次記憶に持つためであり, 主記憶に配置されるトライ木自身のコンパクトさには言及していない. そのため, LSM-Trie の手法は純粋なインメモリデータストアには有効ではない.

2.8 Apache Spark

Apache Spark [88] は, MapReduce [15] のような分散処理を行うための処理系である. データを主記憶にキャッシュする機能を持つ. データが主記憶に入りきらない場合にはデータの一部や全体を二次記憶上に配置することもできる. 2014 年にカリフォルニア大学バークレイ校の AMPLab で開発が始まった.

Apache Spark は Resilient Distributed Dataset (RDD) [87] と呼ばれる読み取り専用の抽象データ構造を用いることを特徴とする. RDD は map や reduceByKey のようなデータ変換の操作を提供し, 操作の呼び出しの結果別の RDD を生成する. こうして生成される RDD は, 変換によって生成されるべきデータの代わりに, 適用された変換の履歴を保持できる. これにより, データの変換処理を結果を取り出すまで遅延することができる. 障害によって失われたパーティションを元のデータから再計算できるため, 高い耐障害性を実現できる.

Apache Spark は本論文でいうインメモリデータストアとは異なるが, RDD の背後にインメモリデータストアを持ち, 生成されたデータを主記憶に保持するオプションを提供している. RDD の persist メソッドに MEMORY_ONLY を指定して呼び出すことで, データを主記憶上に保持する. また, 同様に MEMORY_ONLY_SER オプションを指定して呼び出すことで, バイト配列にシリアル化されたコンパクトな形式で主記憶に保持できる. しかしながら, それでも本研究で実現するデータ構造よりもメモリ効率は低い. 詳しくは付録 B に示す.

2.9 Succinct

Apache Spark や Tachyon [43] に組み込まれるインメモリデータストアとして, Suffix Array [27, 64–67] の派生を用いる Succinct [1] が提案されている. Suffix Array は文字列の接尾辞の集合を効率よく格納するデータ構造であり, パターンマッチング等の操作を提供する. Suffix Array により, コンパクトなデータ構造でありながら, データを展開することなく効率の良いクエリを実現できる. Succinct では, 効率よく Suffix Array

を作成するため, 3段階 (LogStore, SuffixStore, SuccinctStore) のマルチストア構成を取る. このようなログ構造の派生は, 2.6 節にあげる多くのデータストアや 2.7 節で述べた SILT や LSM-Trie に共通する.

Succinct は, パターンマッチング等の複雑な操作を提供するデータ構造であるのに対して, 本研究では, 単純なキーバリューストアを提供する. さらに, 本研究では, 利用可能なメモリが多い時には高速, 少ない時には速度を犠牲にして自動的にコンパクトになるようなインメモリデータストアの実装方法を示す.

2.10 ログ構造によるメモリ割り当ての効率化

インメモリデータストアのメモリ割り当てを効率化するためにログ構造を用いた研究がある [62]. 従来の malloc によるメモリ割り当ては, データを削除した際にメモリ断片化を引き起こす問題があった. データを再配置してこの問題に対処できるが, より短い停止時間で再配置を行うにはより大きな空きメモリが必要であるというトレードオフになっていた. この研究では, ハッシュテーブルの実装に必要なメモリ割り当てを一定サイズのセグメントごとにまとめて行えるようにログ構造を用いる. これにより, セグメントを単位として局所的に再配置を実行し, それぞれの停止時間を短くできる. 本研究では, ハッシュテーブルおよびオンライン LOUDS トライに必要に応じてメモリを割り当てる必要があるため, このような単純な手法だけではメモリ割り当ての問題を解くことができない.

2.11 データ圧縮によるメモリ効率の改善

データ圧縮技術は, 使用メモリ量の削減や, ディスクアクセス低減による性能の改善にも利用できる. 特に, ページフォールトを低減するために, オペレーティングシステムのバーチャルメモリの機能と組み合わせ, ページ単位でデータを圧縮する手法が研究されている. そのような手法の一つとして, メモリ中に格納されているアドレスを効率よく圧縮する手法が提案されている [80]. また, 繰り返し現れるパターンに基づくものや, ゼロ部分の削除なども提案されている [50]. これらの手法は, 圧縮アルゴリズムをページに一般的に見られる特性に特化しているため, 本研究の文字列を格納する目的には適さない.

Nakar らの手法 [54] では, プログラムのメモリアクセスの局所性が実行のフェーズによって変わることに着目し, フェーズによってどのページを圧縮するかを選択している. 本研究では文字列の格納を一貫して行うアプリケーションを対象としているため, このような手法も適用できない.

Yang ら [83] は、組込みシステム向けに主記憶に圧縮されたスワップ領域を割り当てるなどを提案している。この手法では圧縮アルゴリズムとして bzip2, zlib, LZRW1-A, LZO (Lempel-Ziv-Oberhumer [55]), および RLE を比較し、圧縮率は低いが高速な LZO を選択している。LZO はオンラインで使用するために、圧縮率はそれほど高くないが、展開速度が速いことを特徴とする。LZO と似たアルゴリズムに Snappy [74] がある。

本研究のアプリケーションである文字列の重複排除では、読み出しが多く、オンラインでデータの展開を繰り返すため、圧縮するデータの粒度が細かい。しかし、LZO や Snappy のような圧縮アルゴリズムは細粒度では有効でない。したがって、本研究ではデータ圧縮技術の使用は避け、コンパクトな構造のまま検索が可能な簡潔データ構造を用いる。

第3章 想定するアプリケーションと実現するインメモリデータストアの要件

本章では、想定するアプリケーションと実現するインメモリデータストアの要件について述べる。まず、本研究の主要なアプリケーションの1つであるテキスト解析システムについて述べる。次に、Apache Lucene を用いてテキスト解析システムを実装する場合に発生する問題について述べる。最後に、その問題を解決するインメモリデータストアに求められる要件を示す。

3.1 テキスト解析システム

本研究で想定する主要なアプリケーションの1つはテキスト解析システムである。テキスト解析とは、自然言語で書かれたテキストデータを解析し、有益なデータを抽出することである。テキスト解析の対象となるデータは、Web やソーシャルネットワーク、あるいは企業のナレッジベースに蓄積されている大量のテキストデータである。その多くは自然言語で書かれており、データ量は数 GB から数 TB にのぼる。

テキスト解析システムの処理は大きく以下の3つのフェーズに分けられる：

1. 発生源からデータを取得する処理
2. データから自然言語解析などによってキーワードを抽出する処理
3. 抽出されたキーワードから索引を作成する処理

上記フェーズの2番目で抽出したキーワードを格納するために、文字列をキーとするインメモリデータストアが使用される。インメモリデータストアには大量のキーワードが入力されるため、その性能やメモリ効率が全体の性能に大きく関わる。

テキスト解析は、ファセット検索機能を持つ全文検索エンジンにより実現されることが多い。ファセット検索とはユーザに対してあらかじめ有用な絞り込みのための条件を提示する検索である。たとえば、Amazon.com のような電子商業サイトでは、シス

```
SERVICE BRAKES, HYDRAULIC:FOUNDATION COMPONENTS:DISC:ROTOR  
1993  
MERCURY  
FORD MOTOR COMPANY  
GRAND ISLAND  
RWD  
AUTO  
GRAND MARQUIS  
2MELM75W3PX  
SERVICE BRAKES, HYDRAULIC:ANTILOCK  
WHEELS  
VOLKSWAGEN OF AMERICA, INC  
WVWDB4504LK  
LONG BEACH  
CORRADO  
VOLKSWAGEN  
FUEL SYSTEM, GASOLINE:DELIVERY:HOSES, LINES/PIPING, AND FITTINGS  
DAIMLERCHRYSLER CORPORATION
```

図 3.1: テキストデータから抽出されたフレーズの例 (NHTSA)

テムによって、あらかじめ商品カテゴリやブランド等の絞り込みに有用なフレーズが提示される。ユーザは提示されたフレーズを選択することで商品の絞り込みを行うため、検索のためのキーワードを知っておく必要がない。この有用なフレーズをファセットと呼ぶ。ファセット検索機能を持つ全文検索エンジンとしては、Apache Lucene がある。

テキスト解析システムをファセット検索機能を持つ全文検索エンジンで実現する場合、解決すべき課題がある。それは、フレーズの数が膨大になることである。通常の電子商業サイトではフレーズの数は数万件である。テキスト解析システムでは、フレーズを人間が与えるのではなくルールに基づき自動的に抽出する。ルールにより、数 100 万から 1 億件のフレーズが自動的に追加されることがある。例えば、医学文献データベースである MEDLINE [56] を解析する MedTAKMI [77] では以下のようなルールが使用された：

- 遺伝子、プロテイン、化学名などの固有名詞
- 「A inhibits B」 や 「A activates B」 など、上記固有名詞を関連づけるフレーズ

MedTAKMI では約 1100 万件の生物医学系論文のアブストラクトから抽出された約 27 万個のフレーズが使われた。その他に、同様のシステムでは、NHTSA [57] を対象としたテキスト解析では約 70 万件の事故情報データベースから抽出された約 640 万個のフレーズが使われた。図 3.1 に NHTSA から抽出されたフレーズの例を示す。各行が 1 つのフレーズを意味する。

3.2 Apache Lucene によるテキスト解析システムの問題

Apache Lucene ではファセットの索引を効率よく作るために、ファセットごとにユニークで連続した ID (ファセット ID) を付与している。ファセットの索引は文書ごとにソートされたファセット ID のリストになる。

ファセット ID を用いる手法は、索引サイズや検索速度の面で優れているが、索引作成時にファセットと ID の対応表を作成し、検索時にはそれを引いてファセットを復元しなければならない。Apache Lucene は、ID とファセットの対応を格納するために TaxonomyIndex と呼ばれる索引を作成する。索引作成時には、ファセットから ID を特定する処理が頻繁に繰り返されるため、ディスク上に作られた TaxonomyIndex を引くと効率が悪い。Apache Lucene ではディスクアクセスを減らすため TaxonomyWriterCache と呼ばれるキャッシュ機構が用いられる。TaxonomyWriterCache としては、全件をメモリに格納するインメモリデータストア (Cl2oTaxonomyWriterCache) と、Least Recently Used (LRU) アルゴリズムにより出現頻度の高いファセットだけをメモリに格納するもの (LruTaxonomyWriterCache) の 2 種類が用意されている。

TaxonomyWriterCache のインターフェースを、図 3.2 に示す。このインターフェースは、単純なキーバリューストアとしてのメソッドを持つ。`put(key, value)` は、文字列 (ファセット) のキー (key) とその ID (正の整数) を値 (value) として引数に取り、キーバリューストアに登録するメソッドである。登録に成功すると `true` を返す。`get(key)` は、文字列 (ファセット) をキー (key) としその ID を値として返すメソッドである。`clear()` と `close()` は、キーバリューストアの内容を全て破棄するメソッドである。`close()` の場合、以後の `put()` や `get()` の操作を受け付けなくなる。`isFull()` は、キーバリューストアがフルかどうかを返すメソッドである。もし、フルであれば、キャッシュに入りきらないキーがあるとみなし、キーが TaxonomyWriterCache で見つからなかった場合に、ディスク上の TaxonomyIndex も検索する。

Apache Lucene は、TaxonomyWriterCache を使い、次のような手順で索引を構築する。

1. 各文書を解析し、ファセットとなる文字列を抽出する。
2. 各文字列を、TaxonomyWriterCache から `get` する。そのとき `null` が返される、すなわちその文字列が初めて出てくるものであれば、ID を割り当て、`TaxonomyWriterCache` に `put` する。
3. 各文書に文字列の ID を関連づける。
4. 各文書に関連付けられた文字列をソートし、圧縮してディスク上に索引として書く。
5. ディスク上の TaxonomyIndex に文字列とその ID を書く。

```

package org.apache.lucene.facet.taxonomy.writercache;

import org.apache.lucene.facet.taxonomy.FacetLabel;
import org.apache.lucene.facet.taxonomy.directory.DirectoryTaxonomyWriter;

public interface TaxonomyWriterCache {

    public void close();

    public int get(FacetLabel categoryPath);

    public boolean put(FacetLabel categoryPath, int ordinal);

    public boolean isFull();

    public void clear();

}

```

図 3.2: Apache Lucene の Taxonomy Writer Cache インタフェース

Apache Lucene でテキスト解析システムを実装する場合, テキスト解析システムが扱うファセットの数に注意する必要がある. 通常のファセット検索では, Amazon.com のような電子商業サイトに見られるナビゲーション機能のように, 多くても数万のフレーズをファセットとして使用できれば十分であるが, テキスト解析システムの抽出するフレーズは単語の組み合わせからなり, ルールに使用するパターンにより, 数 100 万から 1 億件の異なるフレーズをファセットとして出力する.

Apache Lucene の Cl2oTaxonomyWriterCache はハッシュテーブルを用いた実装であるため, テキスト解析システムで利用するには大きなメモリが必要になる. また, LRU を用いた場合には, キーの数が多くなり, メモリに格納できない数になった時点で, ディスクアクセスが発生し始める. 我々の実験では, LRU を使った場合, 同じ文書量に対する索引構築時間はインメモリデータストアを使った場合の 2 倍以上になることがあった. このときの文書数は 100 万件, キー数(ファセット数)は 2100 万件であった.

3.3 本研究で実現するインメモリデータストアの要件

文字列データを解析する際, データに含まれる文字列を抽出し, その重複を排除して採番する処理が多用される. 文字列の重複を排除して番号を割り当てる際, すべての文字列を 1 つずつ保持する辞書が作成される. 文字列に一意の番号を割り当てるには, 出現するすべての文字列に対して, すでに割り当てが行われたかどうかを調べる `get(key)` の操作を行う必要がある. 文字列がすでに登録されていれば, 検索結果は割り

当てられた番号となる。まだ登録されていない新たな単語の場合、検索結果は空となり、新しい番号を割り当てる操作 $\text{put}(key, value)$ が行われる。

本研究では、キーバリューストアに保存する値として短い固定長の整数だけを想定する。数十億件の文字列を処理する場合でも、重複排除やカウンティングの処理では固定長の整数で十分対応できる。また、可変長の長いデータが必要であったとしても、オーバーフロー領域へのインデックスを保持することで対応できる。

キーとして格納される文字列の長さはそれぞれ異なり、平均では数十バイトに及ぶ。値は割り当てた番号を保持する 4 バイトの固定長整数である。したがって、ナイーブな方法で格納すると、キーを格納するためのメモリ領域が値を格納するメモリ領域よりもはるかに大きくなる。本研究では値は常に 4 バイトの固定長整数とし、主にキーである文字列の格納方法について議論する。

データストアに入力される文字列は、データ分析中に重複して繰り返し現れ、その出現頻度は文字列によって異なる。本研究で扱うアプリケーションでは、文字列の出現頻度が少数の高頻度のものと、多数の低頻度のものに分かれることを仮定する。そのような出現頻度分布となる例にジップの法則 [89] がある。ジップの法則は、出現頻度順位 k の要素が全体に占める割合が $1/k$ に比例するという経験則である。ジップの法則に従う分布をジップ分布と呼ぶ。ジップの法則はもともと文書中に現れる単語の出現頻度に関して発見された法則であり、ジップ分布は文字列データ解析ではよく見られる分布である。他にも様々な事象がジップの法則に従うことが発見されている。例えば、都市の人口のランクと都市の人口の関係 [26] や、遺伝子の発現数のランクとその遺伝子の発現数の関係 [25]、ソーシャルネットワークなどのスケールフリーネットワーク [78] のノードのリンク数のランクとリンク数の関係、Web サイトのアクセス数のランクとアクセス数の関係 [11] などの頻度分布がジップの法則に従うことが知られている。

本研究では、次の要件を満たすインメモリデータストアを実現する。

- 文字列のキー k と整数の値 v を取るマップ型のインターフェースを持つ。その操作には、キーに値を対応付ける $\text{put}(k, v)$ と、キーを指定して値を取り出す $\text{get}(k)$ がある。ハッシュテーブルと置き換え可能である。アプリケーションに組み込んで利用される。
- キーが長く、値は短い固定長の整数 (Apache Lucene では 4 バイト) である。必要なメモリの大部分はキーである文字列が占める。
- get の引数 k の分布は、高頻度少数のキーと低頻度多数のキーに分かれ、どのキーがどちらに分類されるかは事前に不明である。
- put されたキーと値の組は、直ちに get に反映されなければならない。

- get されたキーがマップに存在しなければ、そのキーは直後に put される。
- キーの数、および、平均のキーの長さを実行に先立ち知ることができる。
- 制限されたメモリで動作し、実行中に、現在利用しているメモリ、および残りのメモリを知ることができる。

本研究では、このような要件を満たしながら、次のような性能を持つインメモリデータストアを実現する。

- 高いメモリ効率。ハッシュテーブルやダブルアレイトライではメモリに入りきらないような多数のキーを保持できる。
- 高いスループット。ダブルアレイトライのような既存のメモリ効率が高いデータ構造よりも高いスループットを実現する。

Apache Lucene が提供する Cl2oTaxonomyWriterCache では、Wikipedia 全文書など、1000 万件を超える文書量を処理するために必要な TaxonomyWriterCache のメモリは 10GB を超えてしまう。そのため、搭載メモリ量の少ない(16GB 程度)安価なシステムでは、そのような文書を処理することができない。本研究では、そのような大量の文書の処理を安価なシステムでも可能にする。また、テキスト解析で使用するハードウェアは簡単に変更できないことが多い。ルールの変更や文書数の増加によって格納すべきキー数は増加する。同じ使用可能メモリ量でより多くのキーを格納できるインメモリデータストアが望まれる。

テキスト解析システムではしばしばルールを更新して文書全体の処理をし直すが、この処理時間は短いほど良い。たとえば、全体の処理に数日かかれば、ルールを更新するのはせいぜい週に 1 度となる。全体の処理時間が数時間であれば、毎日ルールを更新することも可能になる。全体の処理時間は、利用するインメモリデータストアの性能に依存している。したがって、インメモリデータストアは高いスループットのものが望まれる。

本研究で実現するインメモリデータストアは、図 3.2 に示した Apache Lucene の TaxonomyWriterCache と置き換え可能である。その他に、次のようなアプリケーションで利用可能である。

- 重複排除と索引生成。Apache Lucene と類似の全文検索エンジンでは必須の処理である。
- カウンティング。MedTAKMI のようなテキスト解析や、小売のマーケティングで使用されるバスケット分析など、共起して現れる事象を発見するための処理でよく使用される。

第4章 オンライン LOUDS トライ構築 手法 OLT1

本章では Online LOUDS Trie Version 1 (OLT1) について述べる.

4.1 OLT1 の基本概念

OLT1 では, 文字列を少數の高頻度の集合と多数の低頻度の集合に分け, それぞれ, 高速なデータ構造とコンパクトなデータ構造に格納する. コンパクトなデータ構造によってできる余剰な使用可能メモリを全て高速なデータ構造に割り当てることができるため, コンパクトなデータ構造のメモリ効率が高いほど, より多くの文字列が高速なデータ構造に格納されてスループットが向上する.

使用可能メモリ量を M とする. 高速なデータ構造に割り当てるメモリの量を決めるには, コンパクトなデータ構造が使用するメモリ量 M_L を見積もる必要がある. OLT1 では, 使用可能メモリ量 M からコンパクトなデータ構造が使用するメモリ量の見積り M_L を引いた残りを, 高速なデータ構造に割り当てるメモリ M_H とする (式 4.1).

$$M_H = M - M_L \quad (4.1)$$

図 4.1 に OLT1 で用いるデータ構造の組み合わせ方法の概要を示す. 新しく `put` された文字列は, はじめ高速なデータ構造に格納される. 多数の文字列が追加されて, 高速なデータ構造の使用メモリ量が定められた上限に達すると, LRU などのキャッシュ置き換えアルゴリズムによって選択された, 一部の文字列がコンパクトなデータ構造へと移される. キャッシュ置き換えアルゴリズムによっては, 文字列の一部が両方のデータ構造に重複して保持されるため, ある文字列に対する読み出しに対して, 最近書き込まれた値を返すには, 読み出しも書き込みも必ず高速なデータ構造から行う.

以下では, 簡単な数を用いて, 提案手法に期待される効果を説明する. 入力に占める文字列の出現回数には大きな偏りがあるとする. ここでは, 出現頻度上位 25% の文字列が, 全体の出現の 99% を占めるとする. そのような出現頻度分布で入力される文字列を重複を除いてメモリに格納する. 各文字列を保持するのに必要なメモリ量は頻度分布のランクによらないとする.

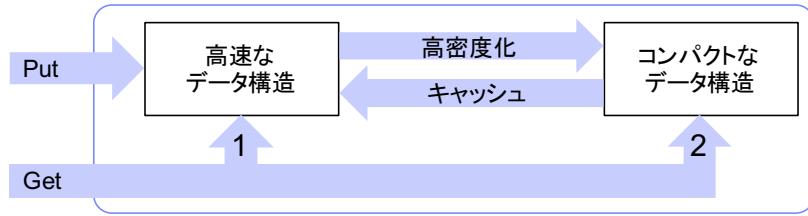


図 4.1: 高速なデータ構造とコンパクトなデータ構造の組み合わせ方法

まず、比較対象として、単一のデータ構造を用いた場合の問題を図 4.2、および図 4.3 に示す。それぞれ、上の矩形は入力される文字列とその出現頻度分布を抽象化して表しており、矩形の左端に近いほど出現頻度が高く、右端に近いほど出現頻度が低い文字列を表す。

高速なハッシュテーブルを用いれば、高いスループットで文字列を検索・格納できるが、図 4.2 のように、文字列の数が多い場合には全てをメモリに格納しきれない。ダブルアレイトライはハッシュテーブルよりもメモリ効率が高いため、図 4.3 のように、同じ使用可能メモリにより多くの文字列を格納できるが、メモリには余剰がある。スループットはハッシュテーブルの 3 分の 2 程度にとどまる（ハッシュテーブルのスループットを 1.00 とした時 0.66 程度）。

このように、単一のデータ構造をデータストアに用いると、与えられた使用可能メモリを余らせるか、もしくは超過して入りきらないことがある。

次に、OLT1 のメモリ使用量とスループットの例を図 4.4 に示す。高速なデータ構造としてハッシュテーブルを用い、コンパクトなデータ構造として LOUDS トライを用いる。ハッシュテーブルと比較して、LOUDS トライのスループットはおよそ 0.05 (20 分の 1)、使用メモリ量は 4 分の 1 程度である。OLT1 では、全ての文字列を LOUDS トライに格納し、残りの使用可能メモリを全て使って、出現頻度の高い文字列をハッシュテーブルに格納するとする。これにより、入力の 99% がハッシュテーブルで処理される。これらの数値から OLT1 のスループットを計算すると $1/(0.99/1 + 0.01/0.05) \sim 0.84$ となる。これは従来手法であるダブルアレイトライのスループット 0.66 よりも高い。

出現頻度分布に大きな偏りがある時、このようなデータ構造の組み合わせによって従来手法よりも高いスループットが得られることが分かる。また、入力される文字列の数によってハッシュテーブルのサイズが調整されるため、メモリの余剰が少なく、より多くの文字列を格納できる。

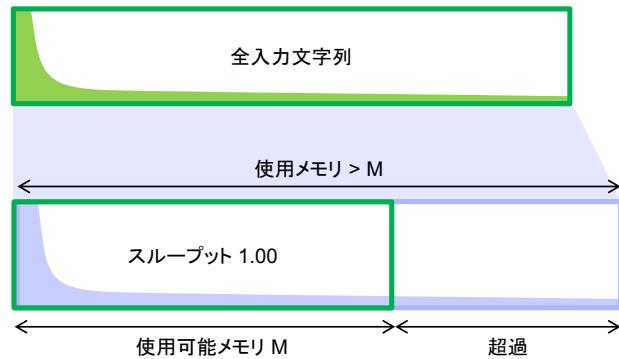


図 4.2: 従来手法: ハッシュテーブル

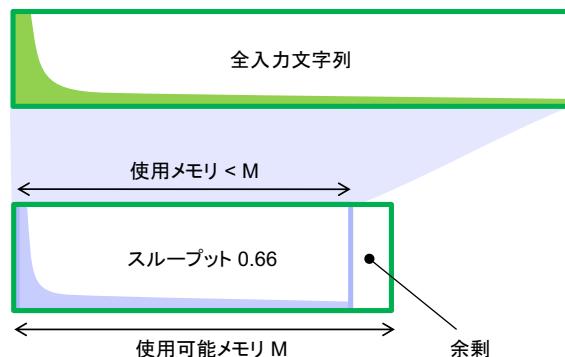


図 4.3: 従来手法: ダブルアレイトライ

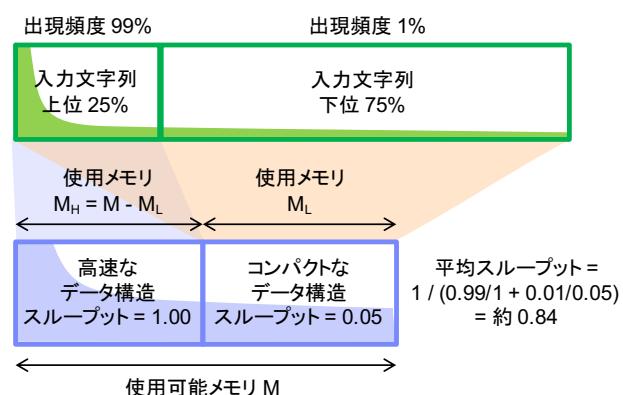


図 4.4: 提案手法: OLT1 (2種類のデータ構造の組み合わせ)

4.2 コンパクトなデータ構造の性能と選定

OLT1 の性能は、入力される文字列の出現頻度分布の偏りの大きさと、使用するコンパクトなデータ構造のメモリ効率、およびスループットにより決定される。本節では、様々なデータ構造の実際のメモリ使用量とスループットを測定した結果をもとに、OLT1 で使用するデータ構造を選定する。計測環境に使用したハードウェア、およびソフトウェアを以下に示す：

- Intel Core i7-3770 3.40GHz 4Core 6MB L3 Cache
- PC3-12800 16GB DDR3 SDRAM 1600Mhz
- SATA 6Gb/s ADATA SSD SX900 512GB
- SATA 6Gb/s WDC 2TB HDD Intellipower 64MB Cache
- Windows 8.1 Pro 64bit OS for x64 based processor
- Java(TM) SE Runtime Environment (build 1.8.0_40-b25)
- Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)

重複排除などのアプリケーションを想定し、各データセットに含まれるキーを重複を除いて格納した場合のメモリ使用量(size、単位は MB)と、重複を含まないキー集合を追記する場合の全体のスループット(put、単位は百万/秒)、キーを格納したハッシュテーブルに対して重複を含むデータセットのキーワードを get した際の全体のスループット(get、単位は百万/秒)を計測した。データセットには表 1.1 の NHTSA, kakaku.com、および reuters.com の 3つを用いた。これらは図 1.1 に示すようにジップの法則に従う頻度分布を持つ。

表 4.1 に、Java の標準ライブラリで提供されている `java.util.HashMap` (Java HashMap), Apache Lucene [22] で使用されている `CompactLabelToOrdinal` (CompactL2O), 本研究で実装したコンパクトなハッシュテーブル (Koyanagi HT), 文献 [18] を再現したカッコーハッシュの実装 (Cuckoo HT) の 4 つのハッシュテーブルの実装の必要メモリサイズと追加/検索の性能を示す。

表 4.1 の put の処理には、メモリ割り当てのほか、ハッシュ値の衝突確率を低く保つためにハッシュ表を伸長してキーを配置しなおす再ハッシュの処理時間が含まれている。このため、put の処理は get よりもスループットが低い。CompactL2O は文字コードの内部表現に Java で標準的に使用されている 16 ビットのエンコーディングを用いているためサイズが大きい。Koyanagi HT と Cuckoo HT は UTF-8 を使用しているため、特にキーが ASCII 文字の多い英文から抽出されるときメモリ効率が高い。Java

HashMap では、通常文字列をキーとする場合は `java.lang.String` クラスを使用するが、この実験では他のデータ構造とキーの型を揃えるために UTF-8 エンコーディングしたバイト配列を用いた。本研究では、最も高速な Koyanagi HT を OLT1 の高速なデータ構造として用いる。

表 4.2 に、ダブルアレイトライの実装 (DA Trie), 二分探索トライの実装 (BS Trie), ダブルアレイトライの接尾辞をハッシュテーブルを使用してまとめる実装 (DA DAWG), 二分探索トライの接尾辞をハッシュテーブルを使用してまとめる実装 (BS DAWG) の 4 つの動的トライ木の実装の必要メモリサイズと追加/検索の性能を示す。

ダブルアレイトライは、表 4.1 に示したハッシュテーブルと同程度の高速な検索 (get) が可能である。一方、キーを追加するときに更新されたノードを格納する場所を探索する必要があるため `put` のスループットは低い。二分探索トライはノードの格納方法が単純で無駄が少ないため、メモリ効率と `put` のスループットではダブルアレイトライよりも優れている。この実験結果から動的データ構造では DAWG にする効果が小さいことがわかる。

本研究では動的トライをバッファとして用い、幅優先走査することによって LOUDS トライを構築する。NHTSA のキーを全て格納した幅優先走査のスループット (秒あたりの訪問ノード数) は、ダブルアレイトライでは毎秒 1100 万ノード、二分探索トライでは毎秒 1200 万ノードであった。したがって、本研究では二分探索トライを LOUDS トライ構築のためのバッファとして使用する。このバッファをバッファトライと呼ぶ。

簡潔順序木 LOUDS, BP, および DFUDS によるトライ木の C++ 言語による実装がある [85]。この実装を用いてこれらの簡潔順序木のサイズと検索性能を調べた。その結果を表 4.3 に示す。

サイズと `get` の性能については表 4.1, 4.2 と同様の項目を、`put` の性能については、重複を含まないキーの集合をバッファし、簡潔データ構造に変換する際のスループット (単位は百万/秒) を計測した。データセットは表 1.1 に掲載したものを使用した。OS には CentOS 6.5 Linux 2.6.32-431.17.1.el6.x86_64 SMP を、コンパイラには Gcc 4.4.7 を用いた。

これらの結果から、この利用方法で最もスループットが高く、使用メモリが少ない簡潔順序木は、LOUDS トライであることが分かる。また、BP トライは検索性能が他の 2 つと比較して著しく低い。この結果を受け、本研究では簡潔データ構造として LOUDS トライを使用する。

4.3 オンライン LOUDS トライ

本節では、本研究で提案するコンパクトなデータ構造、オンライン LOUDS トライの構築法について述べる。提案手法では、バッファトライに決まった数の文字列が入力さ

表 4.1: ハッシュテーブルの必要メモリサイズと put と get のスループット

	NHTSA			kakaku.com			reuters.com		
	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)
Java HashMap	1062	1.71	2.53	2357	0.48	3.55	844	1.99	4.16
ComapctL2O	592	1.19	1.97	1019	1.67	2.44	376	1.34	2.60
Koyanagi HT	345	1.55	3.22	764	1.88	4.63	237	1.99	4.65
Cuckoo HT	317	0.99	2.91	652	1.13	4.59	209	1.23	4.14

表 4.2: 動的トライ木の必要メモリサイズと put と get のスループット

	NHTSA			kakaku.com			reuters.com		
	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)
DA Trie	145	0.94	2.26	447	1.14	4.25	168	1.11	3.86
BS Trie	126	0.91	1.03	368	1.30	2.04	141	1.43	1.69
DA DAWG	143	0.83	2.08	463	1.08	4.07	164	0.91	3.74
BS DAWG	125	0.86	1.02	385	1.27	2.16	138	1.24	1.75

表 4.3: 簡潔順序木によるトライ木 (C++) の必要メモリサイズと put と get のスループット

	NHTSA			kakaku.com			reuters.com		
	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)
LOUDS Trie	39	2.19	0.66	122	1.99	1.48	46	1.85	1.30
BP Trie	46	2.41	0.17	144	2.30	0.12	54	2.09	0.12
DFUDS Trie	47	2.34	0.54	144	2.27	1.23	55	1.97	1.04

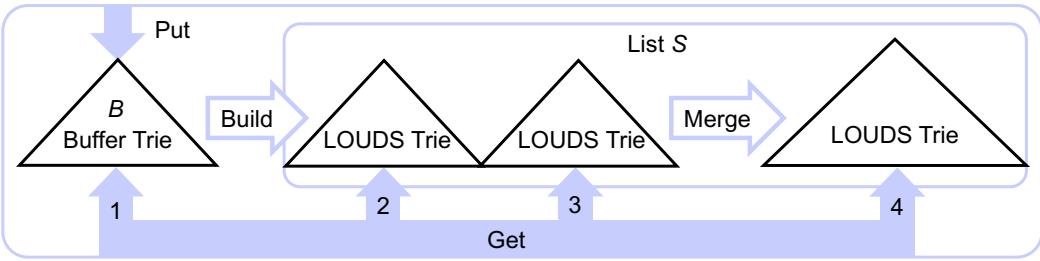


図 4.5: オンライン LOUDS トライ

れるごとに LOUDS トライを構築する。その概要を図 4.5 に示す。

オンライン LOUDS トライへの入力は、文字列のキーと整数の値の組みを要素とする列である。バッファトライのサイズの上限を w とする。 S は LOUDS トライを保持するリストとする。

4.3.1 バッファリングと構築

入力したキーと値の組みはバッファトライ B に格納される。この B は二分探索トライを用いて実装される動的なトライ木であり、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。

w 個のキーと値の組みが B に格納されると、 B に対する幅優先走査によって LOUDS トライが構築され、リスト S に追加される。その後 B を空にする。

全体をコンパクトにするために、 w は全体のサイズに対して十分小さくなるように設定する。ただし、バッファトライのサイズを小さくしすぎると性能に影響がある。バッファトライのサイズの設定方法については 4.3.7 節で詳しく述べる。

4.3.2 検索

複数のデータ構造からなるオンライン LOUDS トライには、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。

まず、バッファトライ B でキーを検索する。ここで見つからなければ、リスト S の新しい順に LOUDS トライを検索する。キーが見つかれば、その値を検索結果として返す。最後まで見つからなければ、結果がないことを示す \emptyset を返す。

4.3.3 マージ

リスト S がマージされることにより, 新しい LOUDS トライが作られる. 最新の値を保持するため, B と S の間でキーが重複する場合には B の値が保持される. S に同じキーが含まれる場合には最新の LOUDS トライに含まれる値が保持される. S の内容はマージによって作られた LOUDS トライによって置き換えられる.

このマージ方法では, 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 $\lfloor N/(wf) \rfloor$ となる.

4.3.4 トライ木の幅優先走査

バッファトライから LOUDS トライを構築するには, バッファトライを幅優先走査する必要がある. また, マージの際も LOUDS トライの幅優先走査が行われる.

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. したがって, ノード n に対して最初の子を返す $\text{firstChild}(n)$, 隣のノードを返す $\text{sibling}(n)$, そして, トライ木のノードに割り当てられた文字を返す $\text{alphabet}(n)$ 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, $\text{firstChild}(n)$ 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, $\text{sibling}(n)$ 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, $\text{firstChild}(n) = \emptyset$ とし, n が共通接頭辞に対して最大の文字を持つとき, $\text{sibling}(n) = \emptyset$ 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>):

```
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 $\text{visit}(p, n)$ 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, および, マージが, $\text{firstChild}(n)$, $\text{sibling}(n)$, $\text{alphabet}(n)$, および, $\text{visit}(p, n)$ を定義することによって実現できることを示す.

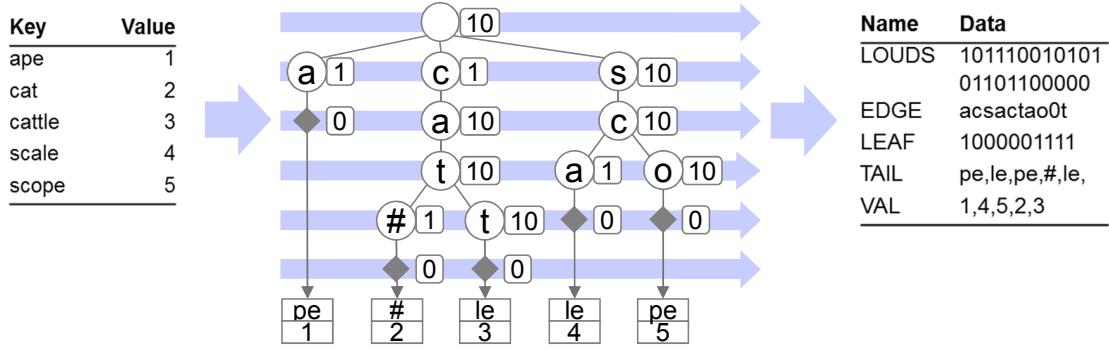


図 4.6: LOUDS トライの構築例

4.3.5 LOUDS トライの構築

本研究で用いる 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 LOUDS 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, 各文字列にはひと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 トライを構築する際には次の操作を実行する. まず, ビット列 “10” をビット列 LOUDS に追加する. 続いて, 4.3.4 項に示した幅優先走査を実行し, $\text{visit}(p, n)$ で, n の子と同じ個数の “1” と続く “0” をビット列 LOUDS 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 $\text{alphabet}(n)$ 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. 図 4.6 に, 文字列 “ape”, “cat”, “cattle”, “scale”, “scope” を含む LOUDS トライの構築例を示す.

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取ることで, ノードの親, 子, 兄弟の順位が計算できる [32]. これらの計算は, LOUDS ビット列をブロック単位に分割し, 2 段階で “1” の個数を保持する完備辞書と呼ばれるインデクス [36] を作ることで行われる. 完備辞書による順位の計算は $O(1)$ 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 LOUDS 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.

LOUDS トライを構成する各データのサイズは計算で求めることができ, そのビット数はノードの数 n に漸近的に比例する. トライ木に含まれるキーの数 N はノード

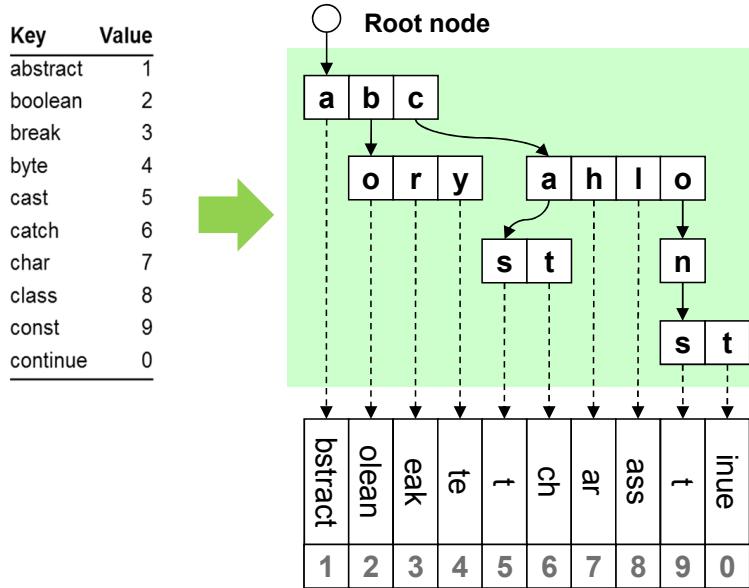


図 4.7: 二分探索トライの例

の数 n 以下である。順序木の簡潔データ構造である LOUDS は $2n + 1$ ビットを消費する。また, LEAF はノードの数と同数であるため n ビット, EDGE はノードと同数の 1 バイト文字を格納するため $8n$ ビットを消費する。キーに対応する値を格納する VAL はキーの数 N に比例する。値は 32 ビットの整数であるため, $32N$ ビットを消費する。LOUDS と LEAF に対して完備辞書が作成される。2.4 節で述べたように, これらのサイズは n が大きいときには定数とみなせるサイズであるため, 本研究では, トライ木のノードやキーあたりのメモリサイズの計算では無視する。以上から, LOUDS トライのノードあたりのビット数は 43 ビット以下である。

4.3.6 二分探索トライによるバッファトライの実装

バッファトライの主な目的は LOUDS トライを構築するために動的にトライ木を作成することである。図 4.5 に示したように, LOUDS トライに組み込む文字列の集合は, 一度動的に構築可能なトライ木に格納し, そのノードを幅優先走査してビット列を作成する。幅優先操作は 4.3.4 項で示した方法で行う。

LOUDS トライを構築する幅優先走査を高速に行うため, 各枝のアルファベットを連続したメモリ領域に保持するとよい。2.3 節で述べたように, 二分探索トライはアルファベットを連続したメモリ領域に保持するため, 幅優先探索を高速に行うことができる。したがって, 本研究ではバッファトライの実装に二分探索トライを用いる。

図 4.7 に, “abstract”, “boolean”, “break”, “byte”, “cast”, “catch”, “char”, “class”, “const”,

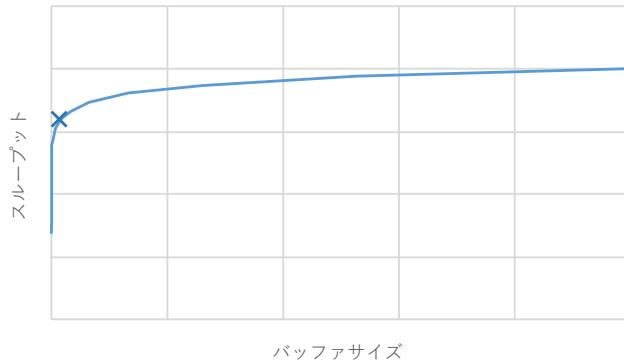


図 4.8: バッファサイズとスループットの関係

および“continue”的 10 個のキーを含む二分探索トライの例を示す。この例ではトライ木の葉に TAIL と値を格納している。文字を格納するノードは“abc”, “ory”, “ahlo”, “st”, “n”, および“st”, の 6 つがあり、格納される文字はノードごとにアルファベット順にソートされている。検索キーの文字を各ノードで探索するときに二分探索アルゴリズムが用いられる。

4.3.7 バッファトライのサイズとオンライン LOUDS トライの性能

オンライン LOUDS トライでは、バッファトライのサイズが性能に影響を与える。バッファトライのサイズを大きくすると、より大きな、より少数の LOUDS トライが構築される。このとき、検索すべき LOUDS トライ数は減り、マージ回数も少なくなる。よって、性能は高くなるが、コンパクトなデータ構造を実現するという目的とは反する。一方、バッファトライのサイズを小さくすると、より小さな、より多くの LOUDS トライが構築される。したがって、検索すべき LOUDS トライ数とマージ回数が増える。よって、性能は低くなるがメモリ使用量は抑えられる。

バッファサイズとスループットの関係は、概ね図 4.8 のようになる。図の横軸はバッファサイズを表し、縦軸はスループットを表す。バッファサイズを大きくしていくと、スループットは向上するが、その効果は次第に緩やかになる。一方、バッファサイズを小さくしていくと、ある所で急激にスループットが低下する。そこで、本研究では、この急激にスループットが低下する直前のバッファサイズを用いる(図の x 付近)。

4.3.8 マージ時に一時的に使用されるメモリ

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、一時的に全体の 2 倍に近い量のメモリが使われる。本章と 6 章の計測にはこの一時的に使用されるメ

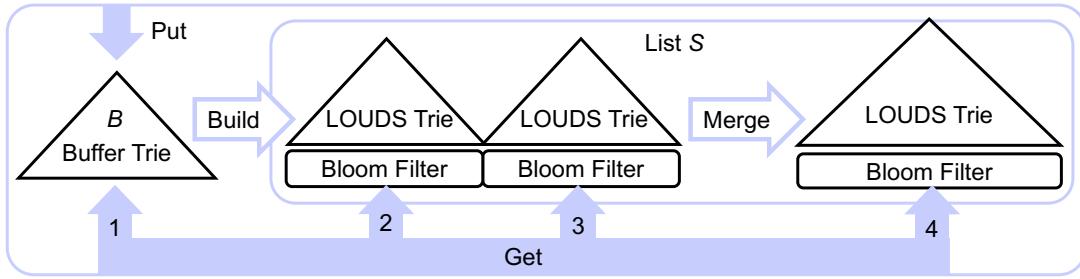


図 4.9: ブルームフィルタ付きオンライン LOUDS トライ

モリは含まれていない。しかし、システムが一時利用可能なメモリを残していない場合には、これも加味して OLT1 のメモリ割り当てを決定する必要がある。これにより、文字列を使用可能メモリに格納しきれない場合、問題になる。

この問題を回避するために、OLT1 では、マージできないほど大きくなった LOUDS トライを、マージ対象から除く手法を取る。マージから除くサイズは手動で設定する。

5 章で述べる OLT2 は、マージが発生しないアルゴリズムであるため、この問題を根本的に解決している。

4.4 ブルームフィルタによる検索の高速化

LOUDS トライを 4.3 節に示した手法でオンラインで構築する場合、LOUDS トライ数の増加に伴って検索性能が低下する。マージによって LOUDS トライの数を抑えることができるが、頻繁なマージには大きな計算時間がかかる。

本研究では、各 LOUDS トライにブルームフィルタを付加し、検索キーを含まない LOUDS トライの検索を事前にスキップすることで、LOUDS トライ数の増加に伴う検索性能の低下を抑える。この改善によって、検索性能を落とさずにマージの頻度を抑えることが可能になる。図 4.9 にブルームフィルタを含む提案手法の概要を示す。

バッファトライ B から LOUDS トライが構築されるとき、同時に、 B の含むキーの集合に対応するブルームフィルタが生成される。また、 S がマージされるとき、同時に、 S の含むキーの集合に対応するブルームフィルタが生成される。LOUDS トライと対応するブルームフィルタは組みにして管理される。

検索時は、LOUDS トライの検索に先立って、対応するブルームフィルタがチェックされ、結果が偽ならば LOUDS 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 LOUDS トライの検索を行う。

4.4.1 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。キーの数を N , 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。Li Fan ら [19] によると, 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。これを基に m を定めると, 式 4.2 が得られる。

$$m = \frac{k}{\ln 2} N \approx 1.44kN \quad (4.2)$$

このときの偽陽性確率 p は, 式 4.3 で表される。

$$p = \left(\frac{1}{2}\right)^k \quad (4.3)$$

本論文では, 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 4.2 によってキー数に対するビット列のサイズを決定する。また, 式 4.3 により偽陽性確率 p を求め, 検索性能を見積もる。

上記の式に従ってブルームフィルタのサイズを決定すると, 実行する CPU によっては剩余の計算が重たくなることがある。そこで本研究では, 実行環境に応じて次の 2 種類の実装を選択して利用する。

実装 1 ビット列のサイズを式 4.2 をそのまま用いる。

実装 2 ビット列のサイズを式 4.2 で求めた値よりも大きい, 最も小さな 2 のべき乗にする。ハッシュ値の余剰を 1 回のビット演算で計算できる。

4.4.2 ブルームフィルタの性能

表 4.4 にブルームフィルタの必要メモリサイズと追加/検索の性能を示す。ブルームフィルタの実装 1 を用い, ハッシュ多重度は 8 とした。この計測結果では, ブルームフィルタの検索は表 4.1 に示したハッシュテーブルのうち, 最も高速な Koyanagi HT よりもさらに高速である。また, そのサイズは表 4.3 に示した LOUDS トライよりも小さい。本研究では, ブルームフィルタを LOUDS トライの検索の高速化に用いる。

4.4.3 モデルによるブルームフィルタの検索性能の見積もり

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。その効果を知るために、まず、ブルームフィルタが無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検索時間を探る。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、キーは一度だけ出現し、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無い場合、 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 トライを検索する時間を C_L とすると、検索時間の期待値は $C_L(M+1)/2$ と計算できる。

q がどのキー集合にも含まれていない場合には、常に全ての 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。従って、検索時間の期待値は $C_L M$ である。

次に、ハッシュ多密度 k のブルームフィルタ付き LOUDS トライの計算時間を求める。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。従って、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、検索時間の期待値は $C_L((M+1)/2 - 1)(1/2)^k + C_L$ となる。なお、ブルームフィルタを検索する検索時間 C_F は、実測で約 $0.30\mu s$ (表 4.4 NHTSA より) であり、LOUDS トライの検索時間 C_L (約 $10.0\mu s$, 表 4.6 NHTSA より) に対して小さいため、ここでは無視している。これらの計測値は、以下の環境で測定した:

- 2.2GHz Dual core Opteron 275 × 2, 2nd cache 2MB
- PC3200 RAM 4GB
- 750GB SATA 7200rpm × 2

表 4.4: ブルームフィルタの必要メモリサイズと put と get のスループット

	NHTSA			kakaku.com			reuters.com		
	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)
Bloom Filter	9.0	1.95	3.31	26	2.50	4.71	8.8	2.40	5.24

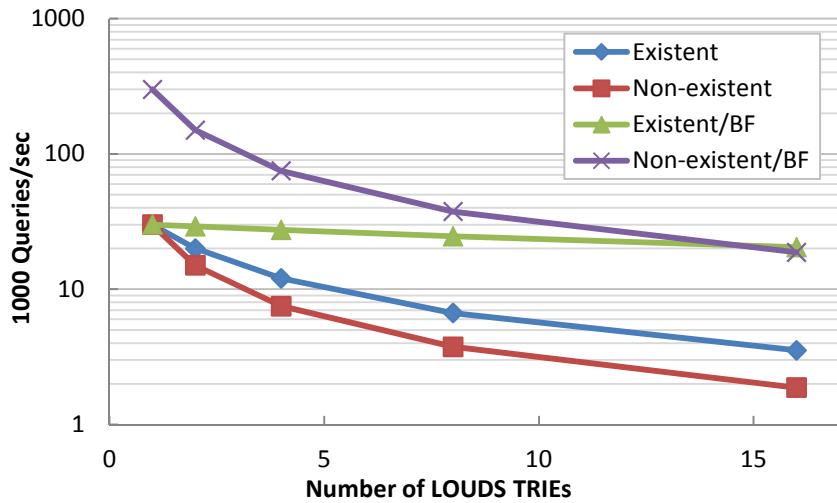


図 4.10: LOUDS トライの個数と検索速度の関係(モデル)

- Windows 2003 Server Standard x64 Edition Service Pack 2
- Java version 1.7.0, IBM J9 VM (JIT enabled, AOT enabled).

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検索時間 C_F を加味する. 常に全てのブルームフィルタが検索されるため, C_{FM} が検索時間の期待値に加算され, 検索時間の期待値は $C_L M(1/2)^k + C_{FM}$ となる.

図 4.10 は, ここまで求めた計算式に, 計測して求めた C_L と C_F を当てはめて, 検索時間の逆数, すなわち, 検索速度を取ってグラフを描いた結果である. Existant, Non-existent はそれぞれブルームフィルタが無く, キーが集合に含まれる場合と含まれない場合, Existant/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. 縦軸は検索速度をログスケールで示す.

このモデルから, 検索キーがいずれかの LOUDS トライに含まれる時には, 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の低下が抑えられることが期待される. 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.

4.4.4 ブルームフィルタのハッシュ値の計算方法

本節では幅優先走査によってブルームフィルタを生成する方法を示す。本手法は LOUDS トライを構築する際の幅優先走査に組み込むことができる。

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 ← 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。

構築する LOUDS トライについて、“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 $\text{visit}(p, n)$ 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 が構築する LOUDS トライの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。

4.5 仮想ノードによる LOUDS トライのマージ

この節では、提案する LOUDS トライの構築とマージのアルゴリズムを示す。また、それらと同時に実行されるブルームフィルタの構築方法についても述べる。

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、一度、両者のキーを中間バッファとなるトライ木 B_m に格納し、 B_m を走査して LOUDS トライを構築すればよい。

しかし, この方法では, 中間バッファ B_m に大きな量のメモリが必要になり, LOUDS トライによるメモリ節約の効果を打ち消してしまう. また, 全てのキーをそれぞれの LOUDS トライから取り出した後, 中間バッファとなるトライ木 B_m を走査する時間が必要になる. しかし中間バッファを使わないマージのアルゴリズムは複雑になる.

本研究では, 仮想ノードによって, 複数の LOUDS トライを仮想的に 1 つの LOUDS トライとして扱えるようにし, 簡潔なアルゴリズムでマージを可能にする. さらに, 4.4.4 項に示した手法を, マージの走査に組み込み, マージと同時にブルームフィルタを生成する.

4.5.1 中間バッファを用いない複雑なマージ

仮想ノードを用いたマージ手法を説明する準備として, 本項では, 問題となる中間バッファを用いない複雑なプログラムの例を示す.

中間バッファを用いずに, 複数のトライ木をマージするには, マージする複数のトライ木から同時にノードを取り出し, それらをソート順を考慮して比較しながら走査を行う必要がある. そのため, 複数のトライ木をマージするプログラムは非常に複雑になる.

そのアルゴリズムを以下に示す. このアルゴリズムで, T_1, T_2 はマージする 2 つのトライ木, n_1, p_1 , および r_1 は T_1 のノード, n_2, p_2 , および r_2 は, T_2 のノードを表す. また, $\langle n_1, n_2 \rangle$ は 2 つのトライ木から取り出したノード n_1, n_2 の組みを表す.

Require: $r_1 \leftarrow \text{root node of } T_1, r_2 \leftarrow \text{root node of } T_2$

```

 $q \leftarrow \{\langle r_1, r_2 \rangle\}$ 
visit( $\langle \emptyset, \emptyset \rangle, \langle r_1, r_2 \rangle$ )
while  $|q| \neq 0$  do
     $\langle p_1, p_2 \rangle \leftarrow q.\text{dequeue}()$ 
    if  $p_1 \neq \emptyset \wedge p_2 \neq \emptyset$  then
        if  $\text{alphabet}(p_1) = \text{alphabet}(p_2)$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \text{firstChild}(p_1), \text{firstChild}(p_2) \rangle$ 
        else if  $\text{alphabet}(p_1) < \text{alphabet}(p_2)$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \text{firstChild}(p_1), \emptyset \rangle$ 
        else if  $\text{alphabet}(p_1) > \text{alphabet}(p_2)$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \emptyset, \text{firstChild}(p_2) \rangle$ 
        end if
    else if  $p_1 \neq \emptyset \wedge p_2 = \emptyset$  then
         $\langle n_1, n_2 \rangle \leftarrow \langle \text{firstChild}(p_1), \emptyset \rangle$ 
    else if  $p_1 = \emptyset \wedge p_2 \neq \emptyset$  then
         $\langle n_1, n_2 \rangle \leftarrow \langle \emptyset, \text{firstChild}(p_2) \rangle$ 

```

```

end if
while  $n_1 \neq \emptyset \vee n_2 \neq \emptyset$  do
     $q.\text{enqueue}(\langle n_1, n_2 \rangle)$ 
    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  $\text{alphabet}(n_1) = \text{alphabet}(n_2)$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \text{sibling}(n_1), \text{sibling}(n_2) \rangle$ 
        else if  $\text{alphabet}(n_1) < \text{alphabet}(n_2)$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \text{sibling}(n_1), n_2 \rangle$ 
        else if  $\text{alphabet}(n_1) > \text{alphabet}(n_2)$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle n_1, \text{sibling}(n_2) \rangle$ 
        end if
        else if  $n_1 \neq \emptyset \wedge n_2 = \emptyset$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \text{sibling}(n_1), \emptyset \rangle$ 
        else if  $n_1 = \emptyset \wedge n_2 \neq \emptyset$  then
             $\langle n_1, n_2 \rangle \leftarrow \langle \emptyset, \text{sibling}(n_2) \rangle$ 
        end if
    end while
end while

```

4.5.2 仮想ノードによるマージ

本研究では、マージされる 2 つの LOUDS トライをあたかも 1 つの LOUDS トライであるかのように操作できるようにする。このような操作を実現するために、2 つの LOUDS トライに共通するノードをまとめる仮想的なノードを導入する。本論文では、このノードを仮想ノードと呼ぶ。仮想ノード $M(n_1, n_2)$ とは、2 つのトライ木の共通の接頭辞を持つノード 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、通常のノードと同等のインターフェースを提供するため、 M によってマージされたトライ木に対して、キーの検索や幅優先走査を単独のトライ木と同様に行うことができる。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。

4.5.1 項で示した 2 つのトライ木をマージしながら幅優先走査を行うプログラムは 35 ステートメントであった。このプログラムでは 2 つのトライ木を扱ったが、任意の数のトライ木をマージするプログラムはさらに複雑な記述を要する。提案手法の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、

4.3.4 項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. これは, 任意のトライ木をマージする場合でも同様である. 仮想ノードが通常のノードと同等のインターフェースを提供するため, 幅優先走査, LOUDS トライの構築, および, ブルームフィルタの構築プログラムは, マージ木か否かにかかわらず全て共通化できる.

4.5.3 仮想ノードの操作

4.3.4 項で述べたように, LOUDS トライを走査するためには $\text{firstChild}(n)$, $\text{ sibling}(n)$, および $\text{alphabet}(n)$ という 3 つの基本操作を提供すればよい. これに加えて, トライ木としての条件を満たすために, 全てのノードの子の列が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 4.5 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 4.3.5 項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.

表 4.5: 仮想ノードの操作

$\text{firstChild}(M(n_1, n_2))$	$\rightarrow M(\text{firstChild}(n_1), \text{firstChild}(n_2))$
	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 $\text{alphabet}(n_1) > \text{alphabet}(n_2)$
$\text{alphabet}(M(n_1, n_2))$	$\rightarrow \min(\text{alphabet}(n_1), \text{alphabet}(n_2))$

4.6 オンライン LOUDS トライ単体の性能

ここでは、4.3 節、4.4 節、および 4.5 節で説明した、コンパクトなデータ構造としてのオンライン LOUDS トライ単体の性能を計測する。4.2 節で示した他のデータ構造と比較するために、Java で実装を行い、同じ条件と手法で計測を行う。

また、4.3.4 項および 4.3.5 項の手法を用いて実装した静的な LOUDS トライの性能も計測する。静的な LOUDS トライのメモリ使用量にはバッファに必要なメモリは含まない。

表 4.6 にその計測結果を示す。オンライン LOUDS トライ単体（オンライン）と静的な LOUDS トライ（静的）と比較すると、使用メモリ量は約 123% (NHTSA) から約 126% (kakaku.com)、また、put 性能は約 33% (kakaku.com) から 84% (reuters.com)、get 性能は約 97% (kakaku.com) から約 113% (NHTSA) である。

オンライン LOUDS トライ単体は、LOUDS トライのサイズに加えてブルームフィルタやバッファトライの使用メモリが追加されるため、静的な LOUDS トライよりも使用メモリ量は多くなる。ブルームフィルタの効果により、get に関しては静的な LOUDS トライの性能に対して大きな差がない。LOUDS トライのマージなどにより、put 性能は静的な LOUDS トライよりも低くなる。

同様にハッシュテーブルと比較すると、使用メモリ量は約 21% (reuters.com) から約 35% (NHTSA) である。put 性能は 13% (reuters.com) から 21 % (NHTSA) である。get 性能は 3.5% (NHTSA) から 6.5% (kakaku.com) である。

4.7 高速なデータ構造と組み合わせた OLT1 全体の動作

4.1 節で述べたように、OLT1 では高速なデータ構造とコンパクトなデータ構造を組み合わせて用いる。図 4.11 に OLT1 全体の概要を示す。

表 4.6: ハッシュテーブル、静的 LOUDS トライ、およびオンライン LOUDS トライの使用メモリ量とスループット (put, get)

	NHTSA			kakaku.com			reuters.com		
	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)	size (MB)	put (M/s)	get (M/s)
LOUDS トライ									
静的	60	0.70	0.10	184	0.94	0.31	65	0.31	0.24
オンライン	73	0.28	0.11	233	0.31	0.30	82	0.26	0.23
Koyanagi HT	345	1.55	3.22	764	1.88	4.63	237	1.99	4.65

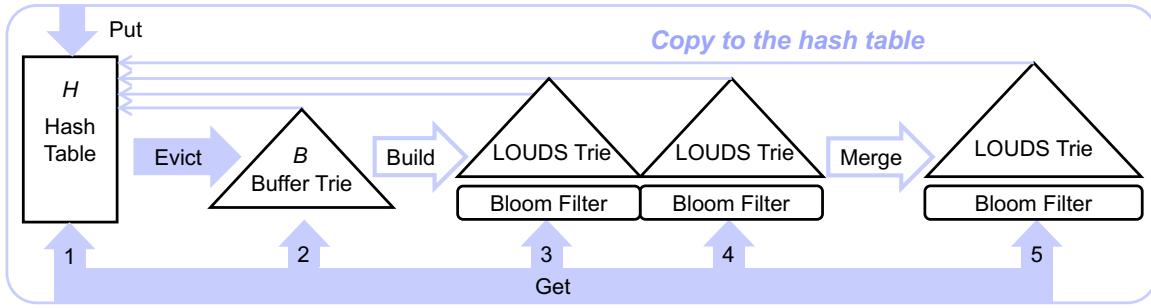


図 4.11: オンライン LOUDS トライ構築手法 OLT1

高速なデータ構造としてはハッシュテーブルを用いる。コンパクトなデータ構造としては 4.3 節から 4.5 節で述べたブルームフィルタ付きのオンライン LOUDS トライを用いる。ハッシュテーブルはオンライン LOUDS トライのキャッシュとして動作する。ハッシュテーブルのサイズは、4.1 節式 4.1 に示したように、使用可能メモリ量 M をちょうど使い切るように決定する。

ハッシュテーブルを組み合わせた OLT1 全体の検索手順は以下のようになる。まず、ハッシュテーブルを検索する。ここでキーが見つかれば、その値を検索結果として返す。見つからなければ、4.4 節に示した手順に従って検索を行う。キーが見つかれば、キーと値の組をハッシュテーブルに格納し、値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。

また、ハッシュテーブルを含む OLT1 全体に対する文字列と値を格納する手順は以下のようになる。検索による結果が \emptyset であった時、入力された文字列と値の組は、新規エントリとしてハッシュテーブルに格納される。ハッシュテーブルに文字列と値の組を格納した結果、ハッシュテーブルのサイズが事前の見積りに基づく M_H に近づくと、キャッシュ置き換えアルゴリズムによってハッシュテーブルから文字列が排除される (Evict)。新規エントリが排除された場合には、4.4 節の手順に従ってバッファに格納され、一定の入力ごとに LOUDS トライに変換・マージされる。

本研究では、キャッシュ置き換えアルゴリズムとして擬似 LRU を用いる。これは、最近使用されたキーにフラグを立て、キーを排除するときにフラグのないものを選択するものである。

本研究の想定するアプリケーションのように、出現頻度分布に大きな偏りのあるデータセットを格納する場合には、擬似 LRU のような単純なアルゴリズムでも高いヒット率が得られる。このような場合、2Q [35] や ARC [48] などの複雑なキャッシュ置き換えアルゴリズムを用いるよりも、擬似 LRU のようなメモリ効率が良く、計算負荷の小さなアルゴリズムが適している。

ハッシュテーブルに割り当てるメモリのサイズは、4.1 節で述べたように、利用可能メモリのサイズから LOUDS トライで必要なメモリサイズを差し引くことで決定す

る。3章で述べたように、本研究が対象とするアプリケーションは、繰り返し実行されることが多い。その場合、LOUDSトライで必要なメモリサイズは、直前の実行結果から推計することができる。最初に実行する時には、ハッシュテーブルには保守的なサイズのメモリを割り当てる。その他に、ランダムサンプリングでLOUDSトライを構築し、必要なメモリサイズを見積もる方法も考えられる。

4.8 モデルによるOLT1の性能評価

本節では、OLT1のスループットを計算するモデルを示し、それを用いてOLT1のスループットを見積もる。OLT1のスループットは使用するデータ構造の性質と処理するデータセットの性質の両方が関係して決まる。この節では、次の2つの関係をモデルを用いて明らかにする：

1. 組み合わせるデータ構造の性質とスループットの関係
2. 入力文字列の出現頻度分布とスループットの関係

これらの関係は、格納する文字列の数に対してスループットを返す関数で表される。この関数はデータ構造やデータセットに関するパラメータを取る。この関数にOLT1と比較対象となるデータ構造のパラメータ、および、人工的な出現頻度分布を持つデータセットのパラメータを当てはめ、格納する文字列の数を変化させた時のスループットの変化を比較する。これによりOLT1の性能を調べる。

4.8.1 スループットのモデル

本項で説明するスループットのモデルは以下の要素を入力パラメータとする：

1. 高速なデータ構造のメモリ効率(キーあたりの平均使用メモリ量) m_H
2. コンパクトなデータ構造のメモリ効率(キーあたりの平均使用メモリ量) m_L
3. 高速なデータ構造の平均スループット θ_H
4. コンパクトなデータ構造の平均スループット θ_L
5. 処理するデータセットの出現頻度分布 $f(k)$ (k は文字列の出現頻度順のランク)
6. 処理するデータセットに含まれる重複を含む文字列の数(入力文字列数) N

7. 処理するデータセットに含まれる重複を除いた文字列の数(格納する文字列の数) n

8. 使用可能メモリ量 M

出現頻度分布 $f(k)$ に従う重複を含む N 個の文字列のデータセットを重複を除きながらデータストアに格納する。データストアに最終的に格納される重複を含まない文字列の数が n である。格納する文字列の数 $n = \max k$, 入力文字列数 $N = \sum_{k=1}^n f(k)$ である。

各データ構造のキーあたりの平均使用メモリ m_H および m_L は格納する対象のデータセットによって異なる値になるため、データセットごとに見積もる必要がある。4.2節では、データ構造のスループットをputとgetに分けて論じた。この節では、それらを結合した平均スループット θ_H および θ_L を用いる。これらの平均スループットも格納する対象のデータセットによって異なるため、データセットごとの見積もりが必要である。さらにこれら平均スループットの値は、重複排除の処理から、対象のデータセットのgetの回数(N)とputの回数(n)の比率によって決まる値である。特に、コンパクトなデータ構造のget/put比は、高速なデータ構造のヒット率によって変わるため、より正確な数値を求めるには、キャッシュ置き換えアルゴリズムまで考慮した見積もりが必要である。ここでは単純化して、これらの値をあるデータセットが与えられた時のデータ構造固有の値として扱う。

上記のパラメータから以下の値を計算する:

- データ構造を組み合わせて該当データセットを処理した場合のスループット θ

$f(k)$ を変化させることによって、様々な出現頻度分布に対するOLT1のスループットを見積もることができる。また、 θ_H や θ_L を変化させることで、異なる性質を持つデータ構造を組み合わせた場合のスループットを見積もることができる。本項では、特に、格納する文字列 n を変化させた時のスループット θ の変化を見るため、他のパラメータを定数とみなし、 θ を n の関数として導出する。

計算のため、上記パラメータに加えて、以下の変数及び関数を導入する:

1. $f(k)$ の累積度数分布関数を $f_c(k)$ とする
2. $\theta_L = \nu\theta_H$ となるスループット比 ν を導入する
3. $m_L = \mu m_H$ となるメモリ効率比 μ を導入する
4. r ($r \leq 1$)を高速なデータ構造に格納する文字列の数の割合とする
5. 高速なデータ構造の使用メモリ量を M_H とする

6. コンパクトなデータ構造の使用メモリ量を M_L とする
7. 高速なデータ構造の累積処理時間を E_H とする
8. コンパクトなデータ構造の累積処理時間を E_L とする

累積度数分布はランク k までの出現頻度分布の和であるので, $f_c(k) = \sum_{x=1}^k f(x)$ が成り立つ. この式から, $k = n$ の時, $f_c(n) = \sum_{x=1}^n f(x) = N$ であるので, $f_c(n) = N$ である.

コンパクトなデータ構造の性質は, スループットの係数 ν とメモリ使用量の係数 μ によって表される. スループットの式 4.10 に異なる ν, μ の値を代入することで, LOUDS トライ以外のデータ構造をコンパクトなデータ構造として用いた場合のスループットを概算できる.

表 4.7 に比較するための 4 つの異なるデータ構造を想定したパラメータを示す. 表中の OLT1 は OLT1 を想定したパラメータを示している. Hash+DA はハッシュテーブルとダブルアレイトライの組み合わせを想定した場合のパラメータである. これらのパラメータ値は表 4.1 や表 4.2 を参考にしている. Hash+SSD はハッシュテーブルと高速な二次記憶 (例えば SSD) の組み合わせを想定した場合である. Hash+HDD はハッシュテーブルと低速な二次記憶 (例えば HDD) の組み合わせを想定した場合である. 4.2 節に示した, 本研究の実験環境では, SSD は約 4793 IOPS, HDD では約 122.8 IOPS (Crystal Disk Mark 5.1.2 x64) であった. 一方, ハッシュテーブルは表 4.1 より, 秒あたり数百万回のアクセスが可能である. これらの数値を参考に, Hash+SSD, Hash+HDD の ν 値を $1/1000$, および, $1/10000$ とした. また, 二次記憶を用いる場合はコンパクトなデータ構造に主記憶を割く必要がないため, 格納する文字列の数にかかわりなく, 全ての使用可能メモリをハッシュテーブルに割り当てることができる. これはモデルでは $\mu = 0$ とすることで表される.

高速なデータ構造の文字列あたりのメモリ使用量は m_H であるので, $n \leq M/m_H$ の範囲では, 文字列を全て高速なデータ構造に格納できる. 提案手法ではコンパクトなデータ構造にもメモリを割り当てる必要があるため, 高速なデータ構造に全ての文字

表 4.7: 想定するデータストアとパラメータ
スループット ν メモリ効率 μ

	スループット ν	メモリ効率 μ
Hash+DA	1/2	1/2
OLT1	1/20	1/4
Hash+SSD	1/1000	0
Hash+HDD	1/10000	0

列を格納できる範囲はこれよりも狭くなる。高速なデータ構造に全ての文字列を格納する時に提案手法のスループットは最大となり、その値は高速なデータ構造のスループットと同じ θ_H になる。

高速なデータ構造に全文字列を格納することができない範囲では、コンパクトなデータ構造に全ての文字列が格納され、高速なデータ構造には、残り使用可能メモリを使って出現頻度の高い文字列が格納される。高速なデータ構造に格納されていない文字列の検索はコンパクトなデータ構造で行われる。高速なデータ構造に格納される文字列の数の割合 r から、高速なデータ構造に格納される文字列の数は rn である。文字列あたりのメモリ使用量は m_H があるので、高速なデータ構造のメモリ使用量 M_H は以下のように計算される：

$$M_H = rn \cdot m_H \quad (4.4)$$

コンパクトなデータ構造には n 個の文字列全てが格納されるため、コンパクトなデータ構造の使用メモリ量 M_L は以下のように計算される：

$$M_L = n \cdot \mu m_H \quad (4.5)$$

使用可能メモリ量 M は固定であるので、4.1 節の式 4.1 で示したように、高速なデータ構造の使用メモリ量 M_H は、全ての文字列をコンパクトなデータ構造に格納した後の残り使用可能メモリである。式 4.1 を以下に示す：

$$M_H = M - M_L$$

この式 4.1 に式 4.4 と式 4.5 を代入すると、以下の式 4.6 が導かれる：

$$rn \cdot m_H = M - n \cdot \mu m_H \quad (4.6)$$

これを r について解くと以下の式 4.7 が得られる：

$$\begin{aligned} r &= \frac{M - n \cdot \mu m_H}{n \cdot m_H} \\ &= \frac{M}{n \cdot m_H} - \mu \end{aligned} \quad (4.7)$$

使用可能メモリ量 M と高速なデータ構造の文字列あたりのメモリ使用量 m_H と係数 μ は定数であるので、式 4.7 から r は文字列の数 n の関数である。

次に、高速なデータ構造に格納される文字列の数 rn を用いて OLT1 のスループット θ を求める。そのためには、文字列の出現頻度分布 $f(k)$ を与えて、高速なデータ構造における文字列の発見回数(ヒット回数)を求める必要がある。このヒット回数に基づ

き, 組み合わせるデータ構造のそれぞれの性能 θ_H, θ_L の合算から, OLT1 全体のスループット θ を計算する.

最も高頻度の文字列を高速なデータ構造に格納する理想的なキャッシュ置き換えアルゴリズムを仮定すると, 高速なデータ構造のサイズ rn と等しいランクの累積出現頻度数がヒット回数であるとみなせる. 与えられた出現頻度分布 $f(k)$ に対する累積度数分布は $f_c(k)$ であるので, ランク rn の累積度数は $f_c(rn)$ である. これを高速なデータ構造のヒット回数とみなす. 高速なデータ構造単体のスループットを θ_H とするとき, 上記ヒット回数における高速なデータ構造の累積処理時間 E_H は以下の式 4.8 で表される:

$$E_H = f_c(rn)/\theta_H \quad (4.8)$$

コンパクトなデータ構造へのアクセスは高速なデータ構造にヒットしなかった場合に発生し, その累積度数は $N - f_c(rn)$ で表される. したがって, コンパクトなデータ構造部分の累積処理時間 E_L は以下の式 4.9 で表される:

$$E_L = (N - f_c(rn))/\theta_L \quad (4.9)$$

式 4.8 と式 4.9 から, OLT1 の平均スループット θ は以下の式 4.10 によって計算される:

$$\begin{aligned} \theta &= \frac{N}{E_H + E_L} \\ &= \frac{N}{f_c(rn)/\theta_H + (N - f_c(rn))/\theta_L} \end{aligned} \quad (4.10)$$

$N = f_c(n)$ であり, かつ, 式 4.7 から r は n の関数であるので, 上記式 4.10 から θ も n の関数である. 言い換えると, 出現頻度分布の累積度数分布 $f_c(k)$ が分かっているデータを, 固定の使用可能メモリ M に OLT1 を用いて格納する時, そのスループット θ は, 入力される文字列の数 n および出現頻度分布 $f(k)$ で表される. これにより, 様々な出現頻度分布を持つデータに対する OLT1 の性能の特性は, 横軸に入力文字列数 n をとり, 縦軸にその時のスループット θ をとるグラフによって表すことができる.

式 4.7, および, 式 4.10 は, コンパクトなデータ構造のメモリ効率を表すパラメータ μ とスループットを表すパラメータ ν を含んでいる. これらを変更することによって, OLT1 のハッシュテーブルと LOUDS トライの組み合わせに限らず, 様々なデータ構造を組み合わせて用いた場合のスループットを見積もることができる.

4.8.2 OLT1 が全ての文字列をハッシュテーブルに格納できる範囲

次の式 4.11 のように、式 4.7 の左辺 r を 1 として n について解くと、提案手法において全ての文字列を高速なデータ構造に格納できる文字列の数 n を求めることができる：

$$\begin{aligned} 1 &= \frac{M}{m_H \cdot n} - \mu \\ 1 + \mu &= \frac{M}{m_H \cdot n} \\ (1 + \mu) \cdot n &= \frac{M}{m_H} \\ n &= \frac{1}{1 + \mu} \cdot \frac{M}{m_H} \end{aligned} \tag{4.11}$$

ここで、式 4.11 に OLT1 の場合を当てはめ、OLT1 が全ての文字列をハッシュテーブルに格納できる最大の n を求める。表 4.6 から、LOUDS トライのメモリ使用量はハッシュテーブルのメモリ使用量の約 1/4 であるため $\mu = 1/4$ とする。このとき、式 4.11 は次のように計算できる：

$$\begin{aligned} n &= \frac{1}{1 + 1/4} \cdot \frac{M}{m_H} \\ &= \frac{4}{5} \cdot \frac{M}{m_H} \\ &= 0.8 \cdot \frac{M}{m_H} \end{aligned} \tag{4.12}$$

ここで、 M/m_H はハッシュテーブルを使用して使用可能メモリ M に格納できる文字列の最大数である。ここから、OLT1 のハッシュテーブルに全ての文字列を格納できるのは、ハッシュテーブルに格納できる文字列の最大数の 80% 以下、すなわち $n \leq 0.8 \cdot M/m_H$ の範囲である。この範囲では OLT1 とハッシュテーブルのスループットは同様とみなすことができるが、 $0.8 \cdot M/m_H < n \leq M/m_H$ の範囲では、OLT1 では LOUDS トライへのアクセスが発生するため、ハッシュテーブルを単体で用いた方が高速になる。

4.8.3 OLT1 と他のデータ構造および二次記憶を用いる方法との比較

本項では、項で求めた式 4.10 を用いて、OLT1 と他の 3 つの手法を比較する。いずれも、高速なデータ構造にはハッシュテーブルを仮定する。それぞれ、コンパクトな

データ構造として, オンライン LOUDS トライを用いる OLT1, ダブルアレイトライを用いる Hash+DA, 二次記憶として SSD を用いる Hash+SSD, および, HDD を用いる Hash+HDD の 4 つの場合について計算を行い比較する. それぞれのデータ構造を表すパラメータ μ および ν の組み合わせには, 表 4.7 の値を用いる.

まず, OLT1 で想定している偏りのある出現頻度分布の典型例として, ジップの法則に基づく出現頻度分布(ジップ分布)を取り上げる.

n 個の文字列の出現頻度分布がジップの法則に基づくとき, ランク k の文字列の出現頻度 $f(k)$ は式 4.13 のように表される. ここで s は出現頻度分布の偏りの大きさを表す係数であり, 数字が大きいほど偏りが大きくなる.

$$f(k) = \frac{1/k^s}{\sum_{x=1}^n (1/x^s)} \quad (4.13)$$

この時, 累積度数分布 $f_c(k)$ は式 4.14 となる.

$$f_c(k) = \frac{\sum_{x=1}^k (1/x^s)}{\sum_{x=1}^n (1/x^s)} \quad (4.14)$$

図 4.12 に $s = 1.00, 1.25, 1.50$ の場合のジップの法則に基づく出現頻度分布を示す. 横軸は頻度順のランク(全文字列数に占める順位の割合), 縦軸は累積度数(全出現数に占める割合)を表す. 図 4.12 と図 1.1 を比較すると, 実際のデータセットは $s = 1.5$ に近い出現頻度分布であることが分かる. したがって, 比較には $s = 1.5$ の場合を用いる.

図 4.13 に, ジップ分布の場合の文字列数に対するスループットのグラフを示す. このグラフは, n を変化させ, $k = rn$ を式 4.14 に代入して計算し, その結果を式 4.10 に代入して計算した結果をプロットしたものである. 横軸の値は, ハッシュテーブルを使用した時に使用可能メモリ M に格納できる最大の文字列の数 (M/m_H) を 1 とした時の入力文字列数 n を表す. 縦軸の値は, ハッシュテーブルのスループット θ_H を 1 とした時の各データストアのスループット θ を表す.

高速なダブルアレイトライとの組み合わせを想定した Hash+DA の場合は, ハッシュテーブルに入りきらない範囲でも高いスループットを維持するが, $n = 2$ の周辺でハッシュテーブルへのメモリ割当がほとんど 0 になるため急激にスループットが低下する. また, $n > 2$ の範囲ではメモリ不足のため使用できない. OLT1 を想定した場合には $n < 2$ の範囲の性能が Hash+DA の場合を下回るもの, メモリに入りきらなくなる $n = 4$ に近づくまでは全体に性能の低下は緩やかである. Hash+HDD の場合も Hash+SSD と同様に $n = 1$ 付近で性能の大きな低下が見られる. Hash+SSD の場合は, ハッシュテーブルとの性能差が大きいため, ハッシュテーブルに入りきらない $n = 1$ 付近で性能が急激に低下する. しかし, ハッシュテーブルへのメモリ割当があるため, n が大きい範囲では性能の低下が緩やかである.

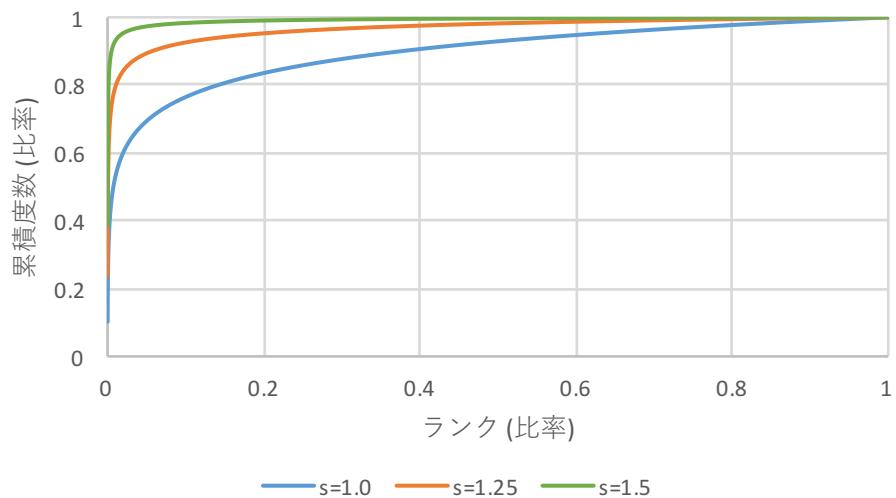


図 4.12: ジップ分布の累積相対度数

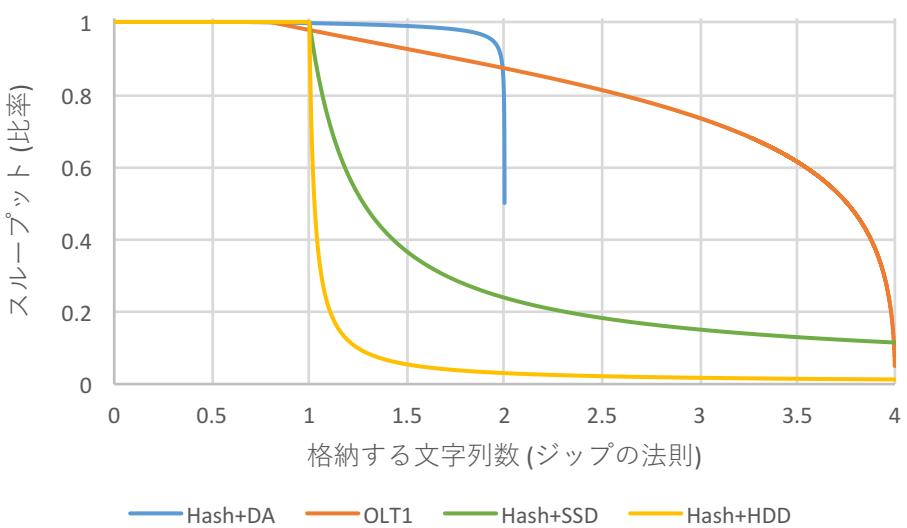


図 4.13: 格納する文字列数に対するスループット (ジップ分布)

4.8.4 一様分布に対する OLT1 の性能

OLT1 は、出現頻度分布の偏りが少ない場合に性能が低いことが予想される。そのような出現頻度分布となる典型的な確率分布には一様分布がある。一様分布では、どの文字列も出現確率が同じである。したがって、出現頻度分布はランクにかかわらず同じ度数とする。これは、 C を定数として次の式 4.15 で表される：

$$f(k) = C \quad (4.15)$$

図 4.14 に一様分布の累積度数分布を示す。縦軸と横軸の値は図 4.12 と同様である。

図 4.15 は、4.8.3 項と同様にして求めた、一様分布の場合の文字列数 n に対するスループット θ のグラフである。文字列の出現頻度分布に偏りがないため、どの組み合わせであっても、ハッシュテーブルに入りきらない境界付近で大きな性能の低下が見られる。いずれも格納する文字列数が増えると性能がコンパクトなデータ構造の性能に漸近する。これは偏りのない出現頻度分布一般に見られる性質である。そのような出現頻度分布では、ダブルアレイトライに入りきらず、使用可能メモリに格納できる範囲で、OLT1 が最も高速なデータストアとなる。

4.8.5 ステップ分布に対する OLT1 の性能

表 4.7 の組み合わせの中で、格納する文字列の数 n が大きい時に、OLT1 が最も高速なデータストアでないような出現頻度分布を人工的に作ることができる。本項では、そのような出現頻度分布を示すことで、OLT1 の有効な範囲をより明確化する。

OLT1 は全ての文字列を主記憶に格納するため、ハッシュテーブルのサイズは n が大きくなるにつれ小さくなり、全ての高頻度の文字列をハッシュテーブルに格納できなくなる。この時でも、二次記憶に文字列を格納する手法では、全ての高頻度の文字列をハッシュテーブルに格納可能なことがある。このような時に、OLT1 のスループットが二次記憶を用いる場合よりも低くなる可能性がある。

以下では、そのような例を人工的に作成する。階段状に高頻度の文字列と低頻度の文字列に分かれている出現頻度分布を考える。そのような出現頻度分布をここではステップ分布と呼ぶ。ステップ分布は 2 つの定数 $C_H, C_L (C_H \gg C_L)$ と閾値 T を用いて式 4.16 のように表される。

$$f(k) = \begin{cases} C_H & (k < T) \\ C_L & (k \geq T) \end{cases} \quad (4.16)$$

ここでは、 $C_H = 1000 \cdot C_L, T = 0.1$ とする。図 4.16 にステップ分布の累積度数分布を示す。縦軸と横軸の値は図 4.12 と同様である。

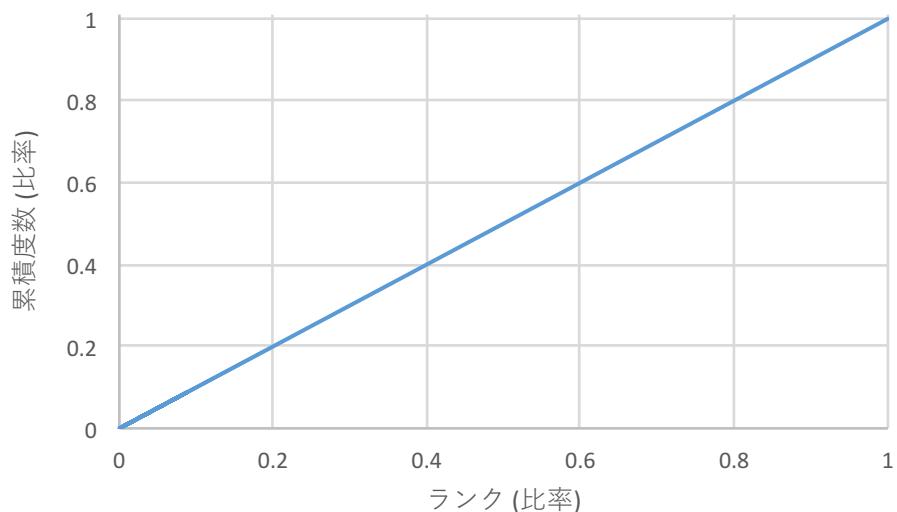


図 4.14: 一様分布の累積相対度数

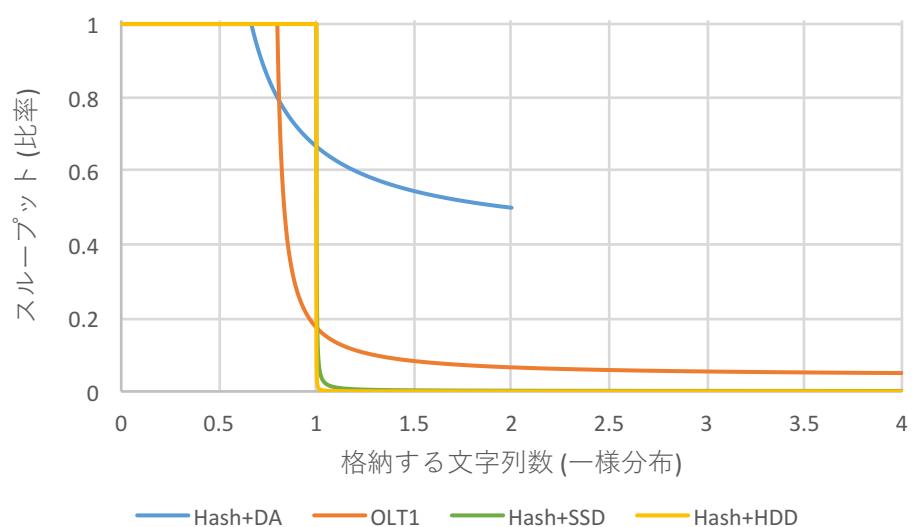


図 4.15: 格納する文字列数に対するスループット (一様分布)

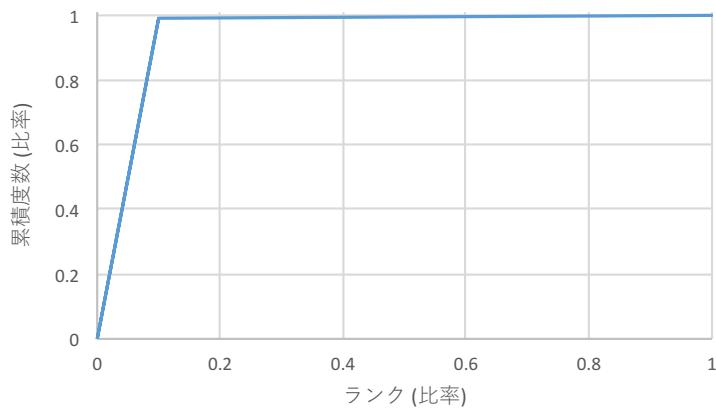


図 4.16: ステップ分布の累積相対度数

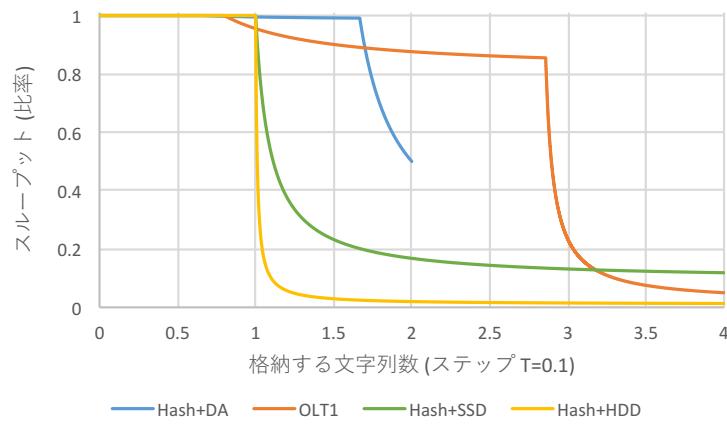


図 4.17: 格納する文字列数に対するスループット (ステップ分布 $T = 0.1$)

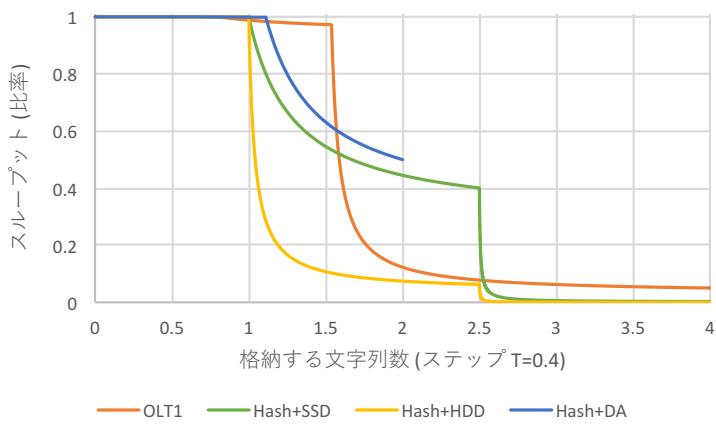


図 4.18: 格納する文字列数に対するスループット (ステップ分布 $T = 0.4$)

図 4.17 は、式 4.16 で示したステップ分布の場合の文字列数に対するスループットのグラフである。この図では、Hash+DA と OLT1 は格納する文字列数によって 3 つの場合に分けられる。第一の場合は全文字列がハッシュテーブルに入る場合であり、性能はハッシュテーブル自身と同じ $\theta = 1$ となる。第二の場合は、全文字列はハッシュテーブルに格納されていないが、高頻度の文字列は全てハッシュテーブルに格納される場合である。この時はハッシュテーブルが高いヒット率を維持するため、大きな性能の低下は見られない。第三の場合は、高頻度の文字列がハッシュテーブルに入りきらない場合である。この時は、一様分布と同様に、ハッシュテーブルのヒット率が大きく低下して性能がコンパクトなデータ構造の性能に漸近する。

OLT1 で第三の場合が発生する理由は、コンパクトなデータ構造に使用可能メモリが圧迫され、ハッシュテーブルのサイズが小さくなつたからである。Hash+SSD や Hash+HDD では、コンパクトなデータ構造の使用メモリ量が 0 であるため、ハッシュテーブルに高頻度の文字列を格納できる範囲が広い。したがって、OLT1 では高頻度の文字列がハッシュテーブルに入りきらないが、Hash+SSD ではそれができる範囲では Hash+SSD が最も高速なデータストアとなることがある。

T を大きくすると、ステップ分布の高頻度の文字列の数が増える。図 4.18 は $T = 0.4$ の時のステップ分布の文字列数に対するスループットのグラフである。この分布に含まれる高頻度の文字列は、与えられた使用可能メモリを全て使ってもハッシュテーブルに格納しきれないため、図中のすべてのデータ構造で第三の場合が現れる。この図では、OLT1 の第三の場合はおよそ $n \geq 1.54$ の範囲である。Hash+SSD および Hash+HDD の第三の場合はおよそ $n \geq 2.50$ の範囲である。グラフの交点から、およそ $1.58 \leq n \leq 2.52$ の範囲では Hash+SSD のスループットが最も高く、およそ $2.52 < n$ の範囲では OLT1 のスループットが最も高くなる。

4.8.6 OLT1 の有効な範囲

OLT1 が高い効果を発揮するのは、ジップ分布のように、高頻度少数の文字列と低頻度多数の文字列からなる入力が与えられたときである。この時、文字列の出現頻度分布の偏りが大きければ大きいほど、ハッシュテーブルのメモリあたりのヒット率が高くなり、より高いスループットになる。偏りが小さい、あるいは、一様分布のように偏りのない出現頻度分布の場合は、ハッシュテーブルに入りきらない範囲で全体のスループットがコンパクトなデータ構造のスループットに漸近する。LOUDS トライが二次記憶よりも高いスループットを持つとき、OLT1 は偏りのない出現頻度分布の時でも二次記憶よりも高速である。

図 4.17 や図 4.18 のように、人工的に OLT1 のスループットが最良でない場合を作ることができる。ステップ分布で、使用可能メモリを全て使って高頻度の文字列を全てハッシュテーブルに格納でき、その時の二次記憶の性能が LOUDS トライよりも高

いとき, かつ, OLT1 では高頻度の文字列を全てハッシュテーブルに格納しきれないとき, OLT1 が使用できるが最良でない範囲が存在する. したがって, OLT1 が最も高い性能を発揮する範囲は, 出現頻度分布が高頻度少数の文字列と低頻度多数の文字列に大きく偏っている場合に限られる. 偏りが少ない場合には他のデータ構造の組み合わせや二次記憶を使用した手法がより高い性能を発揮する可能性がある.

以下に, モデルによって, OLT1 と, 他のデータ構造, および二次記憶を用いる方法の性能を比較した結果をまとめる:

- 入力がジップ分布の場合と一様分布の場合の結果を比較すると, OLT1 は出現頻度分布が高頻度少数の文字列と低頻度多数の文字列に大きく偏っている場合に高い性能になる. 偏りがない場合には LOUDS トライの性能に近い.
- LOUDS トライの代わりにダブルアレイトライを利用する方法も良い方法である. ただし, 扱える文字列数は LOUDS トライの半分程度となる.
- 入力の偏りがない場合(一様分布)でも, OLT1 は二次記憶(SSD や HDD)を使う方法よりも広い入力文字列数の範囲で高い性能を発揮する.
- OLT1 よりも Hash+SSD が高い性能を示すような入力分布を人工的に作成することができる.

4.9 マルチスレッド化に関する考察

4.7 節で示した OLT1 は, 単体では並行に呼び出されることを前提としていない. ただし, データを複数のシャードに分割することでマルチスレッド化できる. 図 4.19 は, 入力を文字列キーごとに 4 つのシャード S_0, S_1, S_2, S_3 に分割する場合を示している. この図では, OLT1 を 4 つのコアを備えたシステムで処理することを想定し, シャードあたり 1 つのスレッドが割り当てられる最も単純なケースを示している. 各スレッドには均等にメモリが割り当てられる.

シャードあたり 1 つのブロッキングキュー, および 1 つの OLT1 が割り当てられる. 入力される文字列は, その文字列のハッシュ値をシャード数 4 で割ったあまりと同じ添字のシャードに割り当てられる. 入力を受け取るスレッド T_R は文字列からシャードの添字を計算し, シャードごとに作られるバッファに文字列を格納する. バッファがいっぱいになると, バッファは該当のシャードに割り当てられたブロッキングキューに格納される. ブロッキングキューへの格納をバッファごとに行う理由は, キューのロック取得回数を減らして衝突を少なくするためである.

シャード S_i に割り当てられたスレッド T_i はキュー Q_i から文字列を取り出し OLT1 に格納する. 文字列キーのハッシュ値によって分割しているので, 各シャードの OLT1 には互いに重なりのない特定の文字列キーの集合が格納される.

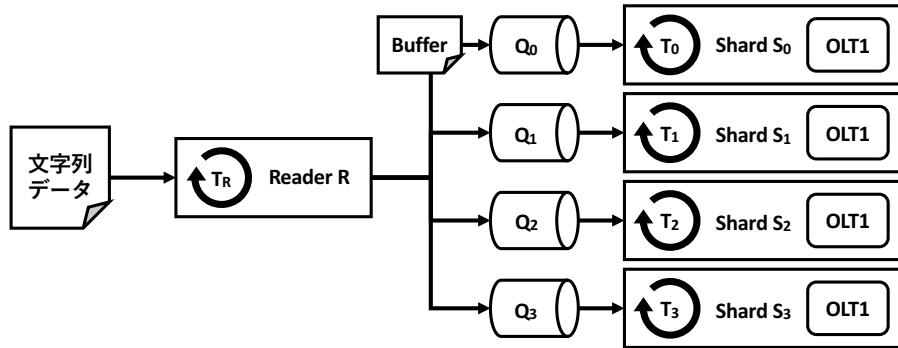


図 4.19: OLT1 のマルチスレッド化

シャードの分割に用いるハッシュ関数がよく一様に分布しているとき, 各 OLT1 に格納される文字列キーの個数は同程度になる. ハッシュ値の分布が文字列の長さに対しても一様であれば, それぞれ同程度のメモリ使用量になる.

しかしながら, 本研究で対象とするデータセットは、入力に大きな偏りがある. この場合, 各シャードへの入力文字列数には偏りが現れる. こうなると, 最も入力文字列数の多いスレッドが全体の実行時間を決定するようになる. 実際にこのような単純な方法を実装し実験した所, 偏りが観測された.

その結果, 4 CPU コアのシステムで性能が 2 倍程度にしかならなかった. 実際のシステムではインメモリデータストア以外の処理も存在するので, この高速化でも有用性がある. インメモリデータストア単体でさらに高速化するために, スレッド数を, CPU コア数以上に増やす方法も考えられる. しかしながら, この場合はメモリも細分化され, OLT1 単体の性能が低下してしまう.

この問題を解決するには, スレッドに文字列を分割する時に, 単純にシャード数で割った余りを使うのではなく, 負荷が均等になるように分割すればよい. しかしながら, そのような分割方法を実行前に知ることはできない. 実行中にスレッドの負荷を観測し, 動的にスレッドを分割する方法も考えられる. しかし, 単純にスレッドを分割すると, そのスレッドに割り当てたメモリを分割することになり, 明らかにメモリ割り当ての効率がよくない. 本来は, 他のスレッドも含めてメモリ割り当てを見直し, 負荷が高い所にメモリを割り当てなければならない. しかし, このようなスレッド間のメモリの再割り当てを実装することは, 難しい問題を含んでいる.

以上のことから, 本研究では, スレッドの分割方法やメモリ割り当ての最適化は今後の課題とし, 単一スレッドでの性能を評価する.

第5章 自律的メモリ割り当て機能を持つオンライン LOUDS トライ構築手法 OLT2

4 章では, LOUDS トライのオンライン構築手法 OLT1 について述べた. OLT1 では, LOUDS トライのほか, ハッシュテーブル, LOUDS トライ構築の中間バッファとして使用する二分探索トライが用いられる. しかし, それらのデータ構造に対して, どのようにメモリを配分するかは入力データに依存しているため, オンラインで最適化問題として解くことは難しい. 本研究では, 与えられた使用可能メモリに対して自律的にメモリを配分する手法を提案する. この提案手法を Online LOUDS Trie Version 2 (OLT2) [39] と呼ぶ. この章では, OLT2 について述べる.

5.1 OLT2 の自律的メモリ割り当てアルゴリズム

図 5.1 に OLT2 の概要を示す. OLT2 は, OLT1 と同様に, 高速なデータ構造としてハッシュテーブル, コンパクトなデータ構造としてバッファトライとブルームフィルタ付き LOUDS トライを持つ.

OLT1 では, 4.1 節で示した方法, および, 4.3.7 項で示した方法にしたがって, 利用者がメモリ割り当てを手動で行っていた. OLT2 はメモリ割り当てを自律的に行う. 自律的なメモリ割り当ては, 以下に示す動作モードを切り替えるアルゴリズムによって行われる:

ハッシュモード 初期状態. put されたキーと値をハッシュテーブルに入れる. 当初の残り使用可能メモリを R_{init} とする. ハッシュテーブルが次第に拡大し, 残り使用可能メモリ R が減少する. R が R_{init} の $X\%$ になった時, バッファモードに遷移する.

バッファモード put されたキーと値をバッファトライに入れる. バッファトライが次第に拡大し, 残り使用可能メモリ R が少なくなる. R が LOUDS トライを作成するのに最小限のサイズになった時, 変換モードに遷移する.

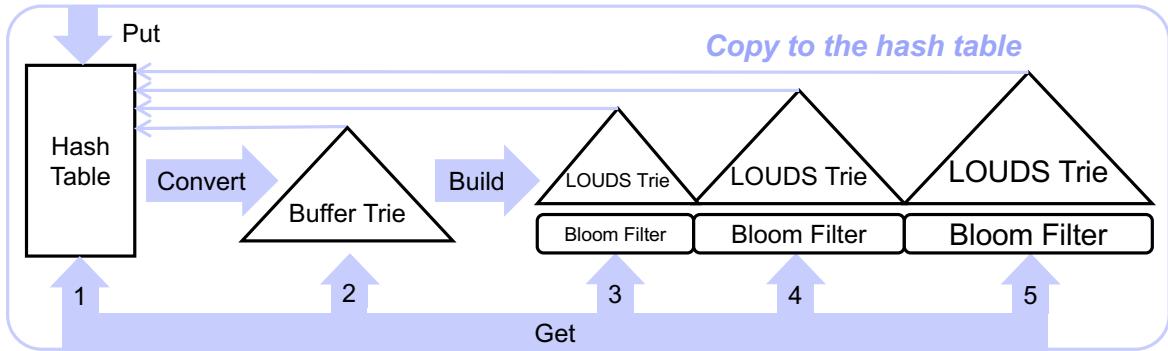


図 5.1: 自律的メモリ割り当て機能を持つオンライン LOUDS トライ構築手法 OLT2

変換モード ハッシュテーブルの内容をバッファトライに移動し, ハッシュテーブルのメモリを開放する. 次に, バッファトライから LOUDS トライとブルームフィルタを作成し, バッファトライのメモリを開放する. 最後に, ハッシュモードに遷移する.

OLT2 で上記のパラメタ X は, ハッシュテーブルとバッファトライのメモリ効率を勘案して決める. この値は, 使用可能メモリにバッファトライを割り当て, ハッシュテーブルに含まれる全てのキーと値をバッファトライに入れたとしても, 必ず全てのキーが格納できるような値に設定する. 現在の所, 余裕を見て 50 % に設定している. OLT2 はモードを切り替えるために使用メモリ量を一定の時間間隔で監視する. 使用メモリ量の計測に含まれるガーベージコレクションの頻繁な実行を避けるために, 現在の所, 計測間隔は 1 秒としている.

図 5.2 は, OLT2 のモード切り替えを伴う動作ステップの概要を表している. 図の各ステップの正方形全体は使用可能メモリを表し, 各領域は以下に示すように各データ構造を表している:

- 白色の領域は残り使用可能メモリを表す
- 青色の領域はハッシュテーブルに割り当てられたメモリを表す
- 黄色の領域はバッファトライの消費するメモリを表す
- 緑色の領域は LOUDS トライの消費するメモリを表す

図中のステップ 1 から 6 の説明を以下に示す:

1. ハッシュテーブルが入力キーを格納して拡大する (ハッシュモード).

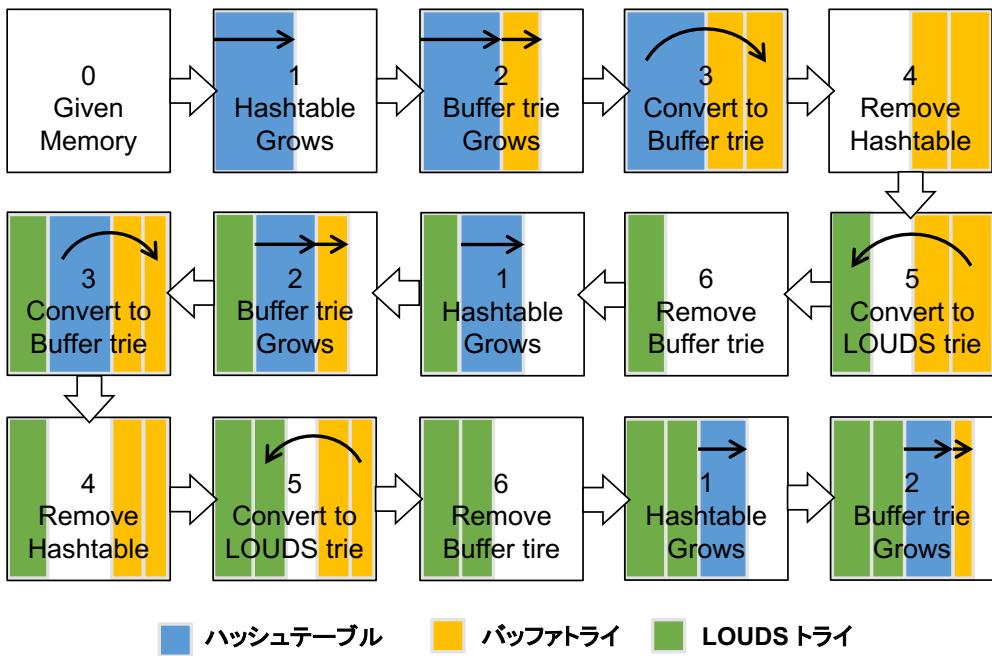


図 5.2: OLT2 の自律的なメモリ割り当てのステップ

2. バッファトライが入力キーを格納して拡大する(バッファモード). ハッシュテーブルもキャッシュとして拡大する.
3. 残り使用可能メモリがLOUDS トライへの変換に必要なサイズに近づくとOLT2は変換モードに遷移する. ハッシュテーブルをバッファトライに変換する.
4. ハッシュテーブルをメモリから削除する.
5. バッファトライを LOUDS トライに変換する.
6. バッファトライをメモリから削除する. ハッシュモードに戻る.

これらの6つのステップにより, OLT2へ入力される文字列キーは, ハッシュテーブルとバッファトライを経て LOUDS トライへ変換される. すべてのデータが処理されるまでこれらのステップが繰り返される.

5.2 Put 操作

ハッシュテーブルを H , バッファトライを B , バッファトライ B から変換される LOUDS トライを L , L に添付されるブルームフィルタを F , LOUDS トライのリストを S とする. ハッシュテーブル H の重複を除くすべてのキーと値を B にコピーした

際に同じキーと値の集合が B 上で消費するメモリ量の見積もりを $E(H \rightarrow B)$ とする。また、バッファトライ B から LOUDS トライ L を作るために必要となる使用メモリ量の見積もりを $E(B \rightarrow L)$ とする。 $E(B \rightarrow L)$ には、 L とそのブルームフィルタ F のサイズの他、 B を幅優先走査するために必要なキューのメモリが含まれる。 $E(H \rightarrow B)$ と $E(B \rightarrow L)$ を見積もる方法については 5.4 節で述べる。

ハッシュモードでは、入力されたキーと値のペアはハッシュテーブル H に格納される。バッファモードでは、ハッシュテーブル H にキーが含まれない場合のみ、バッファトライ B にキーと値が格納される。もし、すでにハッシュテーブル H にキーが存在する場合、 H に値が書き込まれる。

5.3 Get 操作

OLT2 の内部では、キーの変更やキャッシュへの移動によって、同じキーが複数保持されていることがある。したがって、キーに対して、最近格納された値を返すためには、決められた順序でデータ構造を検索する必要がある。

キーは、まず、最初にハッシュテーブル H で検索される。もし、キーが H で見つかれば、そのキーに関連付けられた値が返される。もし、キーが H に含まれていなければ、次にバッファトライ B が検索される。もし、キーが B で見つかれば、そのキーに関連付けられた値が返される。もし、キーが B に含まれていなければ、次に LOUDS トライのリスト S に含まれる LOUDS トライ L がリストの最後から降順に検索される。LOUDS トライが検索される前に、ブルームフィルタ F によってキーの存在が確認される。もし、 F が偽を返したら、 L の検索はスキップされる。そうでないなら、 L が検索される。キーがバッファトライか LOUDS トライで発見された場合には、そのキーと値の組がハッシュテーブルにコピーされ、重複を示すマークが付与される。そして、発見された値が返される。この手順でキーが見つからなかった場合は、キーが存在しないことを示すために、 \emptyset が返される。重複を示すマークはハッシュテーブルからバッファトライに変換する際に使用される。マークを持つ要素はバッファトライへの変換から除外される。

5.4 変換モードへの遷移条件とメモリ見積もり

変換モードではハッシュテーブル H とバッファトライ B に含まれるデータをすべて LOUDS トライ L に変換する。変換には H から B へ、 B から L への 2 つの段階が含まれるため、OLT2 は 2 つの遷移条件を持つ。 H から B への遷移条件(遷移条件 1)、 B から L への遷移条件(遷移条件 2)を以下に示す:

遷移条件 1 残り使用可能メモリの量が使用メモリ見積もり $E(H \rightarrow B)$ に近づいた時

遷移条件 2 H から B への変換後, 残り使用可能メモリの量が使用メモリ見積もり $E(B \rightarrow L)$ に近づいた時

遷移条件 1 を判定する式を **遷移条件判定式 1** とする. 遷移条件 2 の判定は H から B への変換前に行われるため, $E(H \rightarrow B)$ の見積もり誤差による判定誤りが含まれる. そのため, H を削除し, 使用メモリ量を測定した後, もう一度遷移条件 2 が評価される. 1 回目の遷移条件 2 を判定する式を **遷移条件判定式 2.1**, 2 回目の遷移条件 2 の判定式を **遷移条件判定式 2.2** とする. 遷移条件判定式 2.1 で使用するメモリ見積もりを $E'(B \rightarrow L)$ とする. また, その時の残り使用可能メモリの見積もりを R' とする.

キーと値のペアが追加された結果, 遷移条件 1 の判定式か遷移条件判定式 2.1 のうち, どちらか 1 つが真の場合に OLT2 は変換モードに遷移する. 変換モードでは, ハッシュテーブル H からバッファトライ B へのデータの移動が行われ, H が削除される. 遷移条件判定式 2.2 が真なら, バッファトライ B から LOUDS トライ L が作成され, S の末尾に追加される.

OLT2 では, $E(H \rightarrow B)$ と $E(B \rightarrow L)$ の 2 つのメモリ量の見積もりが一定の時間間隔で計算され, 遷移条件 1 と遷移条件 2 がチェックされる.

R を残り使用可能メモリ量, $M(H)$, $M(B)$, $M(L)$ を, それぞれ, ハッシュテーブル H , バッファトライ B , LOUDS トライ L の使用メモリ量とする. $M(L)$ には L に関連付けられたブルームフィルタのサイズを含まない. $M(F)$ を L に関連付けられたブルームフィルタ F の使用メモリ量とする.

ここで, $|H|$ と $|B|$ を, それぞれ H と B の格納するキー数, d を B もしくは S に含まれ, かつ H にも含まれる重複したキーの個数とする.

ハッシュテーブル H , バッファトライ B のキーあたりの使用メモリ量の比を定数とみなし, その定数を C_B とする. バッファトライと LOUDS トライのキーあたりの使用メモリ量の比も定数とみなし, その定数を C_L をとする. 定数 α を $\alpha|B|$ が B を幅優先走査する際に必要になる一時メモリの量となるように定める. $|H|$, d , および $|B|$ は各データ構造内で計数し, C_B , C_L , および α は事前に計測される. $M(H)$, $M(B)$, および R は実行時に計測される.

ハッシュテーブル H からは重複を除くキーがバッファトライ B にコピーされるため, 重複を除いたキーの比 $(|H| - d)/|H|$ を用いて, メモリ量の見積もり $E(H \rightarrow B)$ は以下の式 5.1 のように計算される:

$$E(H \rightarrow B) = C_B \cdot M(H) \cdot ((|H| - d)/|H|) \quad (5.1)$$

また, $E(B \rightarrow L)$ は, 以下の式 5.2 のように計算される:

$$E(B \rightarrow L) = C_L \cdot M(B) + M(F) + \alpha|B| \quad (5.2)$$

遷移条件 2 の 1 回目の判定時には、将来起こる H から B への変換によって増えるメモリ量の見積もり $E(H \rightarrow B)$ を、判定時に計測される $M(B)$ に加える必要がある。したがって、 $E'(B \rightarrow L)$ は、式 5.2 の $M(B)$ を $E(H \rightarrow B) + M(B)$ で置き換えた以下の式 5.3 で計算される：

$$E'(B \rightarrow L) = C_L \cdot (E(H \rightarrow B) + M(B)) + M(F) + \alpha|B| \quad (5.3)$$

同様に、残り使用可能メモリ量についても見積もり量を反映させる必要がある。遷移条件判定式 2.1 で使用される残り使用可能メモリの見積もり R' は、以下の式 5.4 により計算される：

$$R' = R + M(H) - E(H \rightarrow B) \quad (5.4)$$

2 つの変換を経て新しく作られる LOUDS トライに含まれるキーの数は、ハッシュテーブルのキーから重複を除いたキーの数とバッファトライにすでに格納されているキーの数の和 $|H| - d + |B|$ である。このとき、 F のハッシュ多重度が k のとき、ブルームフィルタの偽陽性確率を最小にするサイズから、 $M(F) = 1.44k \cdot (|H| - d + |B|)$ とする。

上記の見積もりには様々な誤差が含まれている。誤差によるメモリ超過を防ぐために、使用可能メモリ量に対するマージンを設ける。OLT2 のメモリ割り当てでは、実際の使用可能メモリ量からこのマージンを引いたメモリ量を上限として割り当てを行う。本研究では、必要なマージンを使用可能メモリ量 M に対して一定の割合 z とする。すなわち、マージンのメモリ量は zM である。6.14 節で行う OLT2 の評価実験では、マージン z を 5% とした。このとき、変換モードへの遷移条件を判定する 3 つの式は次のように表される：

遷移条件判定式 1 $R - E(H \rightarrow B) \leq zM$

遷移条件判定式 2.1 $R' - E'(B \rightarrow L) \leq zM$

遷移条件判定式 2.2 $R - E(B \rightarrow L) \leq zM$

ハッシュテーブル H 、バッファトライ B のいずれかに対する put の後、メモリ割り当てを含む以下の手手続きが実行される：

1. 残り使用可能メモリ R 、使用メモリ量 $M(H)$ 、 $M(B)$ が計測される
2. 遷移条件判定式 1 が偽、かつ、遷移条件判定式 2.1 が偽ならば、この手手続きを終了する

3. 遷移条件判定式 1 が真, または, 遷移条件判定式 2.1 が真ならば, 変換モードに遷移する
4. H の重複を除くすべての要素が B にコピーされる
5. H がメモリから削除される
6. 残り使用可能メモリ R , 使用メモリ量 $M(H), M(B)$ が計測される
7. 遷移条件判定式 2.2 が偽ならば, 変換モードからハッシュモードに遷移し, この手続きを終了する
8. 遷移条件判定式 2.2 が真ならば, B から LOUDS トライ L とブルームフィルタ F が作成される
9. L が S の末尾に追加され, B が削除される

第6章 評価実験

本章では、提案手法 OLT1、および、OLT2 を評価する。まず、OLT1 で用いているコンパクトなデータ構造であるオンライン LOUDS トライ単体が、既存の代表的な手法よりもコンパクトになることを示す。次に、3 章で述べたアプリケーションを用いて OLT1 全体のスループットを計測する。この結果から、OLT1 がどのようなケースで有用性があるかを明確にする。

OLT1 はブルームフィルタ、バッファトライ、ハッシュテーブルを含み、それらのサイズやブルームフィルタのハッシュ多密度などのパラメータを設定する必要がある。この章では、本研究で設定したこれらのパラメタの妥当性を実験により示す。

最後に、OLT2 の性能を示す。OLT1 の評価実験と同様に、3 章で述べたアプリケーションを用いて OLT2 のスループットを計測する。この実験によって OLT2 に有用性があることを示す。また、OLT2 のメモリ割当の様子を示し、自律的なメモリ割り当てが有効に機能することを示す。

6.1 実験環境

本章では以下に示す E1、および E2 の 2 つの計測環境で実験を行った。使用したハードウェア、およびソフトウェアを以下に示す：

- 実験環境 E1

- 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
- Java version 1.7.0, IBM J9 VM (JIT enabled, AOT enabled)

- 実験環境 E2

- Intel Core i7-3770 3.40GHz 4Core 6MB L3 Cache
- PC3-12800 16GB DDR3 SDRAM 1600Mhz

- SATA 6Gb/s ADATA SSD SX900 512GB
- SATA/600 WDC WD20EZRX 2TB x2
- Windows 8.1 Pro 64bit OS for x64 based processor
- Java(TM) SE Runtime Environment (build 1.8.0_40-b25)
- Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)

実験環境 E1 では 4.4.1 項で説明したブルームフィルタの実装 2 を用いる。それ以外ではブルームフィルタの実装 1 を用いる。

本章では 1 章の図 1.1 と表 1.1 に示したデータセットを用いて実験を行う。各データセットの名前と概要を以下に示す：

NHTSA 米国高速道路安全局の収集した交通事故データベース [57] から自然言語処理によって抽出されたキーワードの列

kakaku.com 大規模情報コンテンツ時代の高度 ICT 専門職業人育成事業 (DCON) [75] により提供された、日本の家電商品等に関する掲示板の HTML データから形態素解析によって抽出された単語の列

reuters.com 同じく DCON によって提供された、英国のニュースサイトの HTML データから抽出された単語の列

6.2 実験方法

本章の実験では、3 章で述べたアプリケーションでの使用を想定したプログラムを用いて、データ構造の性能を測定する。このプログラムは単一スレッドで実行される。入力データは実験環境で示した SSD 上のファイルに格納される。図 3.1 に示した形式と同様に、ファイルには 1 行につき 1 つの文字列が格納される。文字列をデータ構造に格納するとき、その文字列の出現順位を ID として割り当て、文字列をキー、その ID を値として保存する。プログラムでは、 N をファイルの行数、 $\text{readline}(i)$ をファイルの i 行目を読む関数、 m をデータ構造とし、 m にキー k が含まれない場合には $m.get(k)$ は \emptyset を返すとする。このとき、実験用プログラムは以下の擬似コードで表される：

```

 $c \leftarrow 0$ 
for  $i = 0$  to  $N$  do
     $k \leftarrow \text{readline}(i)$ 
    if  $m.get(k) = \emptyset$  then
         $m.put(k, c)$ 
         $c \leftarrow c + 1$ 

```

```
    end if  
end for
```

実験では、表 1.1 に示した 3 つのデータセットが入力データとして使用される。プログラム実行中か、プログラム終了後に、データ構造 m のメモリ使用量が計測される。また、このプログラムの for 文を秒あたり何回実行するかをスループットとして計測する。for 文 1 回につき 1 個のキーが処理されるため、グラフ中ではこのスループットを秒あたりに処理されるキーの個数として表示する。

6.3 評価実験で用いる使用可能メモリ量

提案手法 OLT1、および OLT2 に有用性があるのは、使用可能メモリ量が少ない場合であることが予測される。そこで、まずハッシュテーブルやダブルアレイライでは、すべてのキーを格納しきれない程度のメモリ量しかない状況での性能を測定する。含まれる文字列の数はデータセットによって決まっているため、使用可能メモリ量はデータセットごとに異なる値を使用する。

5.4 節で述べたように、OLT2 は使用可能メモリの 5% をマージンとして取るため、同じデータを格納するとき OLT1 よりもメモリ使用量が多い。そのため、各データセットを OLT2 にちょうど格納できるメモリ量を使用可能メモリ量とする。表 6.1 にデータセットごとの使用可能メモリ量を示す。

実験では、これらの固定された使用可能メモリ量に対して、データセットから格納する文字列数に応じた部分集合を取り出して入力する。これにより、OLT1 や OLT2 に対して、ハッシュテーブルに大きなメモリを割り当てるられるケースから、ほとんど割り当てられないケースまで実験対象とすることができます。

6.4 オンライン LOUDS トライ単体の性能

OLT1 では、コンパクトなデータ構造として、4.3 節で述べたオンライン LOUDS トライを用いる。本節では、このオンライン LOUDS トライ単体の性能を示す。本データ構造と既存の代表的なデータ構造に、表 1.1 に示したデータセットを入力して最終的なサイズを比較する。本データ構造はデータ入力の途中でマージに伴うメモリを消費

表 6.1: 評価実験で用いる使用可能メモリ量

	NHTSA	kakaku.com	reuters.com
使用可能メモリ量	89 MB	269 MB	101 MB

するが,一時的なものであるためここでは考慮しない. 実験には 6.2 節のプログラムを用いる. 実験環境には E2 を用いる.

表 6.2 に実験の結果を示す. この実験結果から, 同じ数のキーを格納する時, 本データ構造はハッシュテーブルやダブルアレイトライのような代表的な既存のデータ構造よりコンパクトであった. すなわち, 同じ使用可能メモリ量で, 本データ構造はそれらのデータ構造と比較してより多くのキーを格納できる.

オンライン LOUDS トライは, 理論上最小限のビット数でデータを表す LOUDS トライにほとんどのキーを格納するため, 高いメモリ効率でキーを格納できる. ただし, 本データ構造は LOUDS トライをオンライン構築するためにブルームフィルタや二分探索トライのバッファなどを使用するため, LOUDS トライそのものと比較するとメモリ使用量が大きい. 表 6.2 に, LOUDS トライに同じデータセットを格納した場合のサイズも計測して併記した(静的 LOUDS トライ). これと比較すると, 本データ構造は 30.5% から 62.2% メモリ使用量が多かった. この内訳は 6.10 節で詳しく見る.

OLT1 はブルームフィルタによって検索性能を高めている. 6.11 節ではブルームフィルタの効果を調べる.

表 6.2 の括弧内にこの実験の実行時の平均スループットを示した. OLT1 は, 他の手法と比較してスループットが低かった. 6.5 節ではハッシュテーブルを含む OLT1 全体の性能を測定する.

6.5 OLT1 の評価実験

本節では, 提案手法である OLT1 を評価する. そのため, 様々なデータストアと性能を比較する. 性能を測定したデータストアは以下の 5 種類である:

- OLT1

表 6.2: 提案したコンパクトなデータ構造オンライン LOUDS トライと既存の手法の使用メモリ量とスループットの比較

	NHTSA	kakaku.com	reuters.com
CompactL2O	591 MB (1.91)	1019 MB (2.17)	375 MB (2.92)
Koyanagi HT	345 MB (2.98)	764 MB (4.71)	237 MB (5.01)
DA Trie	124 MB (1.28)	467 MB (2.98)	150 MB (2.77)
オンライン LOUDS トライ	77 MB (0.08)	244 MB (0.24)	86 MB (0.19)
静的 LOUDS トライ	59 MB (N/A)	151 MB (N/A)	53 MB (N/A)

括弧内はスループット (million keys/sec)

- ハッシュテーブルとダブルアレイトライの組み合わせ (Hash+DA)
- ハッシュテーブル (Hash)
- ダブルアレイトライ (DA)
- ハッシュテーブルと二次記憶 (SSD) の組み合わせ (Hash+SSD)

OLT1 には、文字列を格納するために必要なメモリを除いた残り使用可能メモリを全てハッシュテーブルに割り当てる。OLT1 は格納する文字列の数が増えるに従って必要とするメモリ量が線形に増える。実験の条件から、全体の使用可能メモリ量は格納する文字列に関わらず一定であるので、ハッシュテーブルのサイズは格納する文字列の数が増えると減少し、それに伴ってスループットが減少することが予想される。

Hash+DA は、OLT1 と同じ手法を用いて、高速なデータ構造にハッシュテーブル、コンパクトなデータ構造にダブルアレイトライを配置したデータストアである。4.2 節の計測から、ダブルアレイトライはそのメモリ効率と性能がハッシュテーブルと LOUDS トライの中間に当たるデータ構造である。そのため、使用可能メモリ量が限られる場合、Hash+DA は格納できる文字列の最大数は OLT1 よりも少ないものの、文字列を全てハッシュテーブルに格納しきれない時に OLT1 よりも高速に動作することが予想される。

Hash+SSD は、二次記憶を用いたデータストアである。使用可能メモリは全てキャッシュとして利用する。キャッシュには Apache Lucene に含まれる 3 章で示した TaxonomyWriterCache のハッシュテーブルによる実装を用いる。キャッシュ置き換えアルゴリズムは Least Recently Used (LRU) アルゴリズムが使用されている。このキャッシュに文字列を格納し、メモリに格納しきれなくなると、二次記憶上のインデックスに文字列を移動する。この実験の結果は OS のキャッシュを含む性能であるため、システム全体の主記憶容量がより少いケースではさらにスループットが低い値になる可能性がある。このデータストアの性能は OLT1 が有用性を持つスループットの下限として参考に掲載した。主記憶のみを利用するインメモリデータストアと比較して、二次記憶を用いるデータストアは格納できる文字列の数が多い。従って、インメモリデータストアである OLT1 は、二次記憶を用いる Hash+SSD のようなデータストアよりも高いスループットを発揮する場合に有用性がある。

データセットには表 1.1 に示した NHTSA, kakaku.com, および reuters.com の各データセットを用いる。使用可能メモリには表 6.1 に示した値を使用する。各データセットに対して設定された実験方法には 6.4 節と同じく 6.2 節のプログラムを用いる。

ハッシュテーブルの実装には表 4.1 に示した Koyanagi HT を用いる。実験環境には予備実験と同様に E2 を用いる。

本実験の結果をデータセットごとに図 6.1 (NHTSA), 図 6.2 (kakaku.com) および図 6.3 (reuters.com) に示す。グラフの横軸は格納する文字列の数(単位は百万)を、縦軸は

スループット(単位は百万/秒)を表している。この実験では、使用可能メモリ量が制限されているため、格納すべきキー数が多い領域ではハッシュテーブルやダブルアレイトライなど使用できないデータストアがある。グラフの線が途中で途切れているのはこのためである。

この実験結果から、OLT1はハッシュテーブルやダブルアレイトライよりもスループットが低いが、より多くのキーを格納できることが確認された。OLT1は、実験に使用したどのデータセットでも、二次記憶を使用する Hash+SSD データストアよりも高いスループットを示した。したがって、OLT1は、格納すべき文字列数に対して使用可能メモリ量が少なく、ダブルアレイトライなどの他のデータ構造では主記憶に格納しきれない場合に有用性がある。

また、Hash+DAは、データセットによっては、ダブルアレイトライ自身やOLT1よりもスループットの高い範囲があることが確認された。しかし、その範囲は広くではなく、reuters.comのように、出現頻度分布により極端な偏りのあるデータセットではほとんど効果がない。

6.6 オンライン LOUDS トライ構築の観測

本節では、OLT1の実際の動作の様子について詳しく見る。6.5節の実験で、各データセットの全文字列を格納した場合のメモリ使用量の推移を図 6.4(NHTSA), 図 6.6(kakaku.com), および、図 6.8(reuters.com)に示す。グラフの横軸は経過時間(単位は秒)，縦軸は使用メモリ量(単位はMB)である。

OLT1では、4.3.7節で述べた方法でバッファトライにメモリを割り当てる。この実験では、バッファトライへ4万件の文字列を保持できるだけのメモリを割り当てた。また、4.1節で述べた方法でハッシュテーブルにメモリを割り当てる。ハッシュテーブルには、利用可能なメモリから最終的にオンライン LOUDS トライ単体が全てのキーを保持するために使うメモリを差し引いた残りのメモリを割り当てる。この実験では、NHTSAでは7.7万件、kakaku.comでは19万件、reuters.comでは36万件の文字列を保持できるだけのメモリをハッシュテーブルに割り当てた。これらの図のようにデータストアの使用メモリ量を時系列で見ると、最終的にメモリを使い切っていることがわかる。しかし、まだ格納する文字列の数が少ない時点では使用可能メモリが多く残されていることがわかる。

図 6.5 は図 6.4 と対応するスループットの推移を示している。この図は、1秒ごとに区切ったウィンドウ内の平均スループットをプロットしたものである。実験開始後、LOUDS トライの増加に伴って大きくスループットが低下するが、その後、LOUDS トライのマージによって LOUDS トライの個数が一定数以上増加しなくなると、巨視的にはスループットの減衰は緩やかになる。マージが発生するたびに一時的に処理が停

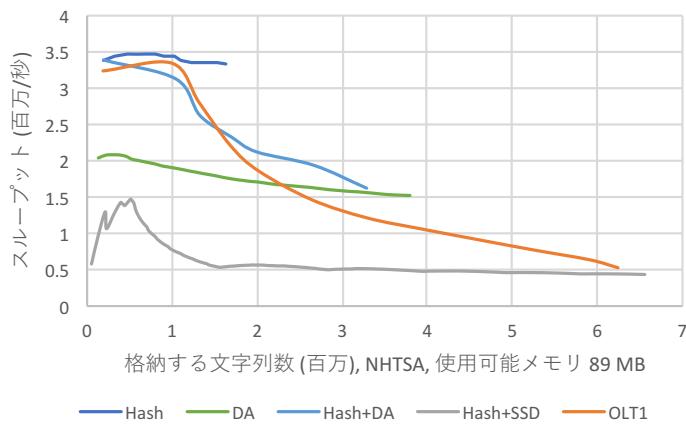


図 6.1: OLT1 の格納文字列数とスループットの関係 (NHTSA 2.5 億件, 89MB)

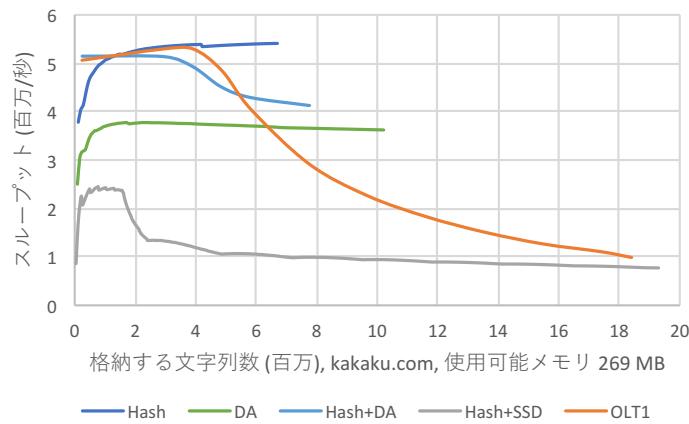


図 6.2: OLT1 の格納文字列数とスループットの関係 (kakaku.com 15 億件, 269MB)

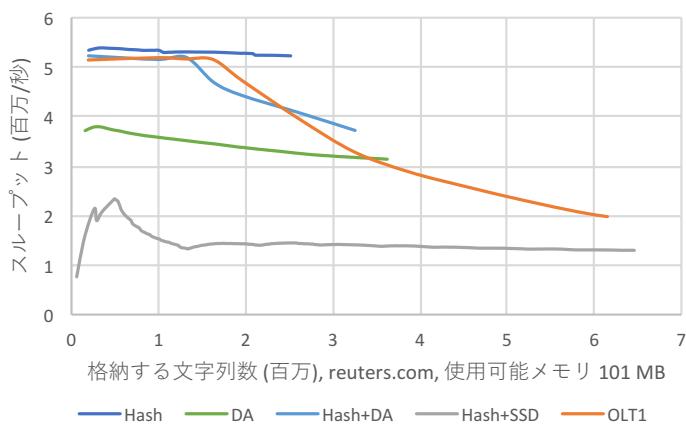


図 6.3: OLT1 の格納文字列数とスループットの関係 (reuters.com 10 億件, 101MB)

止する。

図 6.7 および図 6.9 も、それぞれ図 6.6 および図 6.8 に対応するスループットのグラフである。どのデータセットでも、バッファトライのサイズおよびマージを行う LOUDS トライの個数が同じ数に設定されているため、マージの間隔は同じ数の文字列が put されるごとに発生するが、グラフ上でのスパイクの間隔は異なっている。

6.7 OLT1 の使用可能メモリとスループットの関係

本節では、OLT1 の使用可能メモリとスループットの関係を調べる。入力データには、表 1.1 の 3 つのデータセットを用いる。実験環境には E2 を用いる。実験は、OLT1 の使用可能メモリ量を変化させ、そこに、6.2 節のプログラムを用いてデータを全て入力する。使用可能メモリ量は、表 6.1 を元に、1 倍、1.5 倍、2 倍、および 2.5 倍について計測を行った。

表 6.3 に実験結果を示す。表の各列は、データセットごとに、使用可能メモリ（メモリ）、設定したハッシュテーブルのサイズ（ハッシュ）、および平均スループット（件/秒）を表している。

OLT1 は、コンパクトなデータ構造に割り当てたメモリ以外の残り使用可能メモリを全てハッシュテーブルに割り当てる。そのため、使用可能メモリ量が多くなると、スループットは高くなる。この設定は見積もりによって手動で行ったため誤差がある。OLT1 が実際に使用したメモリは使用可能メモリの約 82% から 93% であった。

この実験では、与えられた使用可能メモリ量に対してスループットはほぼ線形になっている。ハッシュテーブルやダブルアレイトライを単体で使用した場合には、使用可能メモリが多くなると、格納できる文字列の数は増えるが、スループットが大きく増加することはない。

表 6.3: OLT1 の使用メモリ量とスループットの関係

NHTSA		kakaku.com		reuters.com	
メモリ	件/秒	メモリ	件/秒	メモリ	件/秒
89 MB	52 万	269 MB	98 万	101 MB	197 万
134 MB	93 万	404 MB	124 万	152 MB	273 万
178 MB	117 万	538 MB	171 万	202 MB	342 万
223 MB	142 万	673 MB	239 万	253 MB	422 万

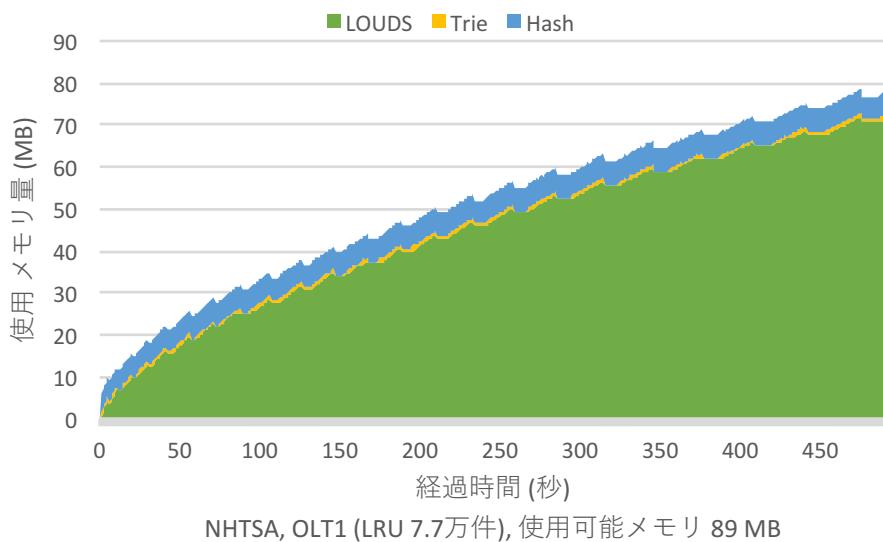


図 6.4: OLT1 の使用メモリ量の推移 (NHTSA 2.5 億件, 89MB)

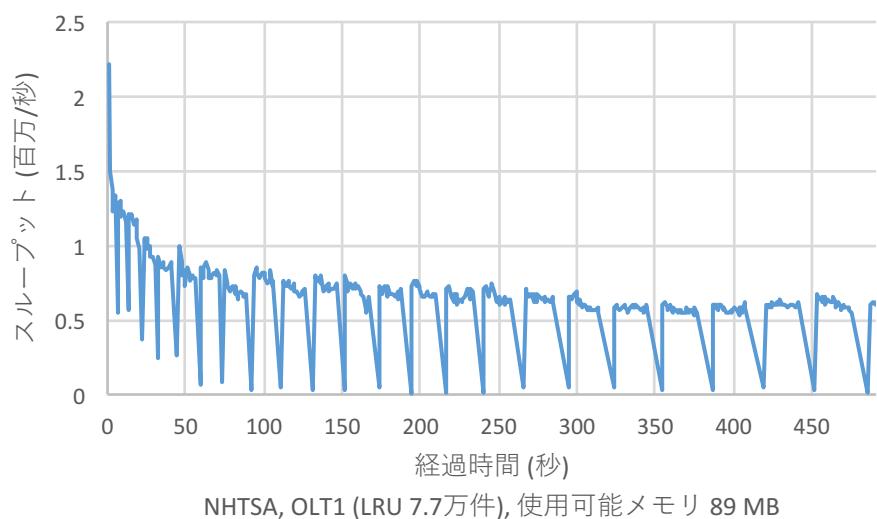


図 6.5: OLT1 のスループットの推移 (NHTSA 2.5 億件, 89MB)

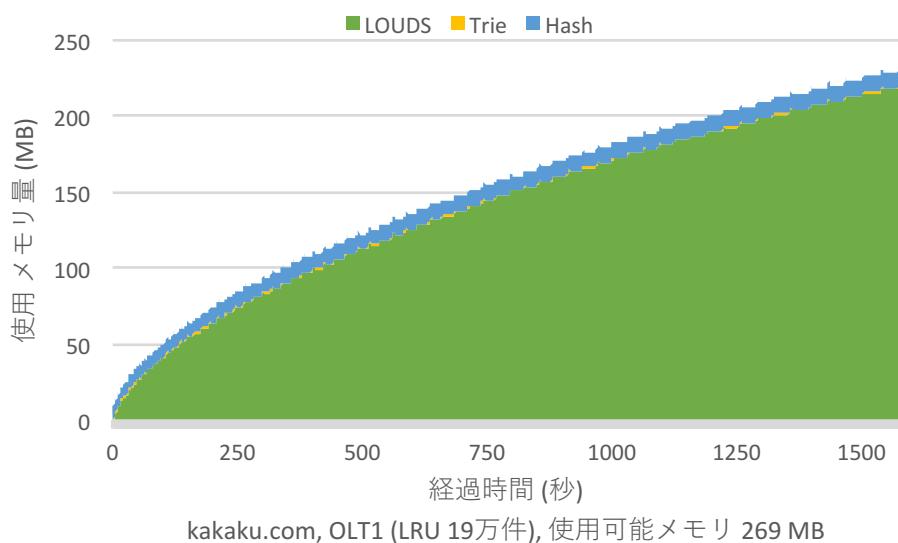


図 6.6: OLT1 の使用メモリ量の推移 (kakaku.com 15 億件, 269MB)

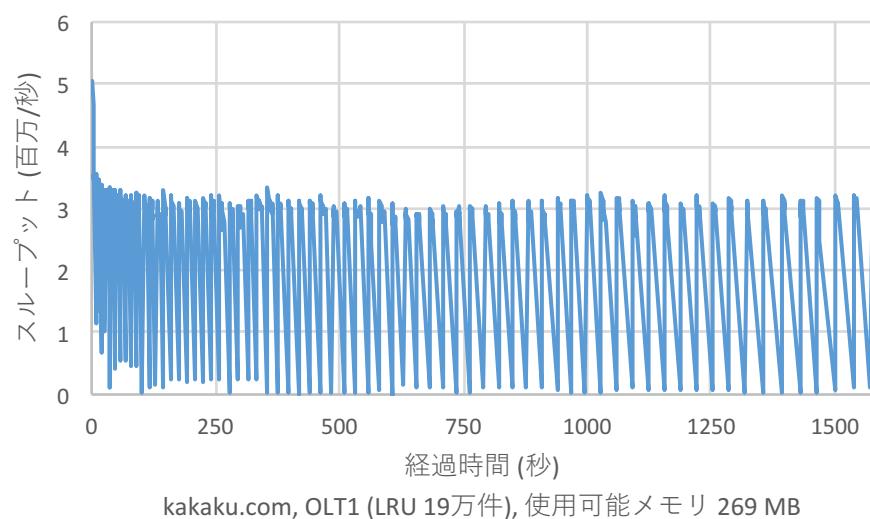


図 6.7: OLT1 のスループットの推移 (kakaku.com 15 億件, 269MB)

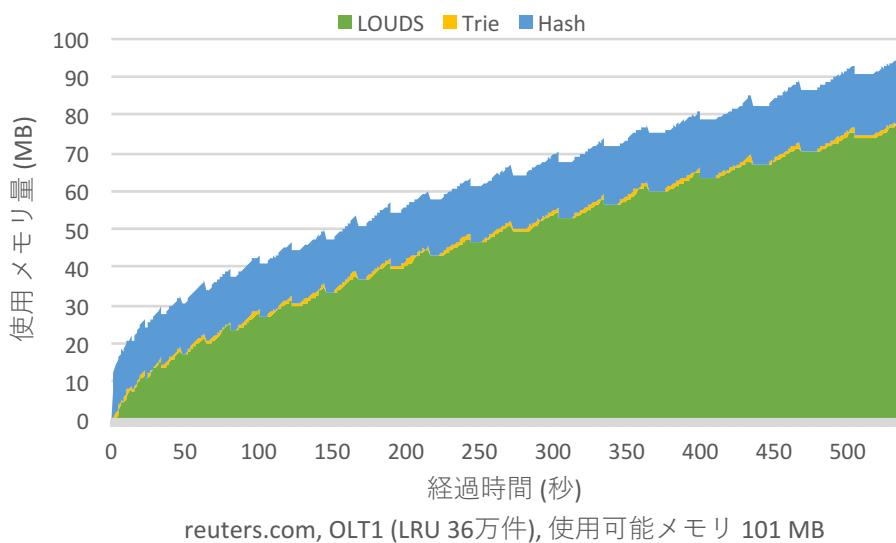


図 6.8: OLT1 の使用メモリ量の推移 (reuters.com 10 億件, 101MB)

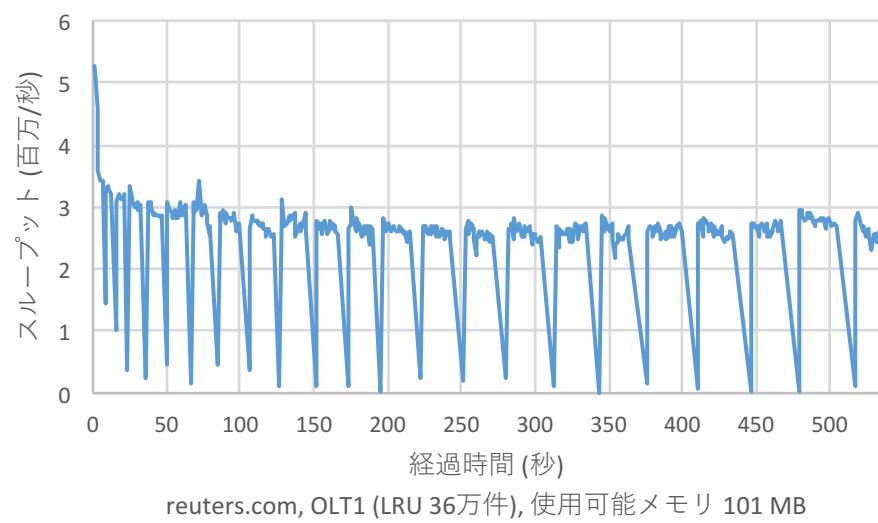


図 6.9: OLT1 のスループットの推移 (reuters.com 10 億件, 101MB)

6.8 OLT1 によるより大きなデータの処理

本節では、より大きなデータセットによる OLT1 の実験結果を示す。より大きなデータセットとして、dumps.wikimedia.org に公開されている Wikipedia [23] のアーカイブを基にしたデータを用いる。利用したデータは 2008 年 1 月時点の英語版 Wikipedia の HTML 全ページを tar により 1 つのファイルにアーカイブしたデータである。このファイルから、英数字、スペースでない Unicode 文字、'\$' および '_' からなる単語を全て抽出した。文字の判別には `Character.isJavaIdentifierPart()` を用いた。さらに、隣接する 2 単語の組み合わせを文字列として抽出した。こうして抽出したすべての文字列を含むデータセットを Wikipedia-en とする。このデータセットの詳細を表 6.4 に示す。

使用可能メモリ量を 8 GB とし、実験環境 E2 を使用してデータセット Wikipedia-en を OLT1 に格納する実験を行なった。データセット Wikipedia-en は、この使用可能メモリ量では、ハッシュテーブルやダブルアレイトライでは扱うことができない。この実験では、データセットの大きさと使用可能メモリ量から、バッファサイズを 32 万件、ハッシュテーブルのサイズを 465 万件とした。この実験のメモリ使用量の推移を図 6.10 に示す。この図の横軸は経過時間(時間)、縦軸はハッシュテーブル(Hash)、バッファトライ(Trie) および LOUDS トライ(LOUDS) の各使用メモリ量(GB) を積み上げグラフで表している。

図 6.10 の結果は、図 6.4、図 6.6、および、図 6.8 とよく似ている。この結果から、OLT1 は、ハッシュテーブルやダブルアレイトライでは扱えないような大きなデータセットについても動作することが確認された。

6.9 バッファトライのサイズとオンライン LOUDS トライの性能

4.3.7 項では、OLT1 におけるバッファトライのサイズを決定する方法について述べた。本節でその方法の妥当性を実験により示す。この実験では、バッファトライのサイズを変化させ、与えられたデータセットを格納するまでの平均のスループットを測定する。なお、バッファトライの効果を見るため、OLT1 のハッシュテーブルは無効にして計測を行う。データセットには、表 1.1 に示した 3 つのデータセットを用いる。この実験の結果を図 6.11 に示す。

実験の結果から、これらのデータセットでは、バッファトライのサイズがおよそ 4 万個以上ではスループットの変化は緩やかだが、4 万個以下ではスループットが急激に低下することがわかる。したがって、メモリ使用量を少なく抑えるにはバッファトライのサイズを 4 万個程度に設定するのが良いことがわかる。

表 6.4: 英語版 Wikipedia から作成したデータセット

Wikipedia-en	
抽出された文字列の総数	618 億件
抽出されたデータサイズ	489 GB
抽出された文字列の平均長	8.48 バイト
重複を除いた文字列の数	4.65 億件
重複を除いたデータサイズ	7.99 GB
重複を除いた文字列の平均長	18.43 バイト

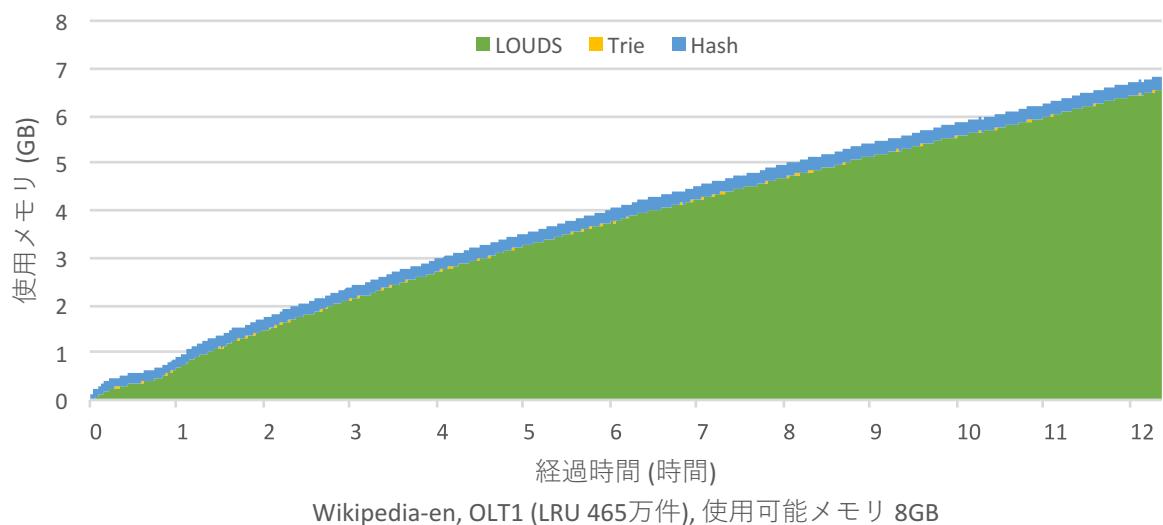


図 6.10: OLT1 の使用メモリ量の推移 (Wikipeida-en 618 億件, 8GB)

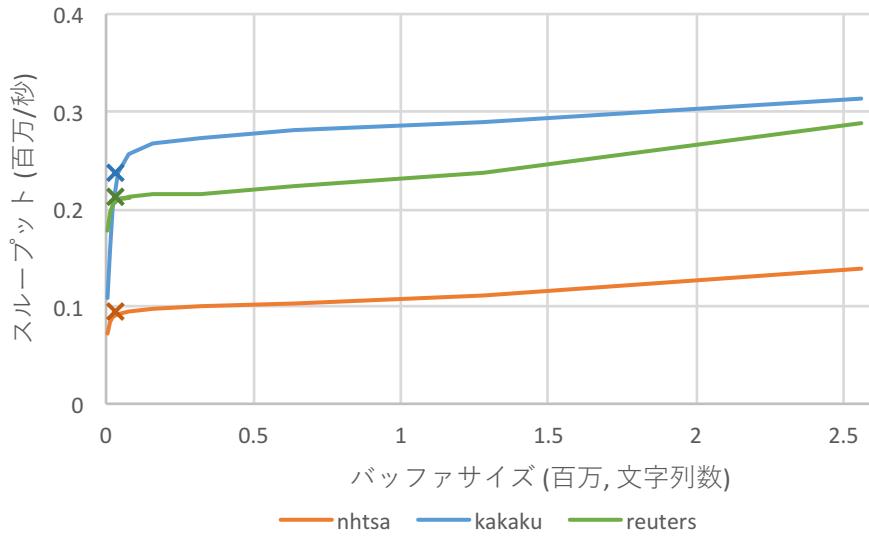


図 6.11: バッファトライのサイズとオンライン LOUDS トライ単体の性能の関係 (x はバッファトライのサイズが 4 万個のときのスループット)

6.10 オンライン LOUDS トライのメモリ内訳

本節では、6.4 節の実験で使用した、オンライン LOUDS トライ単体のメモリの内訳を測定する。

ここではキーと値のペアを格納した後の LOUDS トライとブルームフィルタのサイズを計測するため、OLT1 のハッシュテーブルは無効とし、バッファトライのサイズも測定対象外とする。表 1.1 の 3 つのデータセットを入力データとする。6.2 節の実験プログラムを使用してオンライン LOUDS トライを作成し、マージした後に使用メモリ量を測定する。ブルームフィルタの実装 1 を用いる。実験環境には E2 を用いる。

LOUDS トライには LOUDS 自身の他に、各ノードの文字を格納する EDGE や、葉ノードを表す LEAF、文字列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、LOUDS トライに対して大きな割合を占めるデータ構造が含まれる。

図 6.12 に測定したオンライン LODUS トライのメモリ使用量の内訳を示す。LOUDS のビット列は高いメモリ効率を持つため、そのサイズは全体の 4.0% (NHTSA) から 4.6% (reuters.com) に過ぎない。一方、EDGE のサイズは 14.6% (NHTSA) から 16.8% (reuters.com)、VAL は 33.5% (reuters.com) から 36.6% (NHTSA)、TAIL は 27.7% (NHTSA) から 29.7% (kakaku.com) の大きさを占める。LEAF は LOUDS と同じサイズであった。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。この実験では、4.4.1 項で述べた通り、ハッシュ多重度を k 、キーの数を N とした

とき, ブルームフィルタ (BF) に $1.44kN$ ビットが割り当てられる. その割合は 12.1% (reuters.com) から 13.2% (NHTSA) であった.

図 6.13 は, 格納する文字列数とブルームフィルタの全体に占めるサイズの割合の関係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 8 である. キーの数が多くなると他のデータ構造も共に大きくなるため, 全体に占めるブルームフィルタの割合にも大きな違いは無く, そのサイズ比はおよそ 9% から 18% であった. 各データセットの全ての文字列を格納した時のブルームフィルタを含む LOUDS トライ全体のサイズは, 68 MB (NHTSA), 210 MB (kakaku.com), および 73 MB (reuters.com) であった. ブルームフィルタ単体のサイズは, 9.0 MB (NHTSA), 26 MB (kakaku.com), および 8.8 MB (reuters.com) であった.

6.11 ブルームフィルタによる検索性能の向上

提案手法 OLT1 では検索速度の向上を狙ってブルームフィルタを付加する. 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 (Existential) 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. なお, LOUDS トライやブルームフィルタの効果を測定するため, OLT1 のハッシュテーブルは無効とする. ブルームフィルタの実装 2 を用いる. 実験環境には E1 を用いる.

4.4.3 項のモデルによる予測によれば, ブルームフィルタによる検索性能向上の効果は検索時のキーの有無によって大きく異なる. したがって, 本節の実験ではキーありの検索とキーなしの検索ができるように, もとのデータセットを以下のように構成し直して実験を行う.

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内, 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) を計測する. 計測時は検索だけを行うため, 6.2 節で述べたプログラムを変更した以下のプログラムを用いる:

```
c ← 0
for i = 0 to N do
    k ← readline(i)
    m.get(k)
end for
```

最初に, 提案手法に適した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

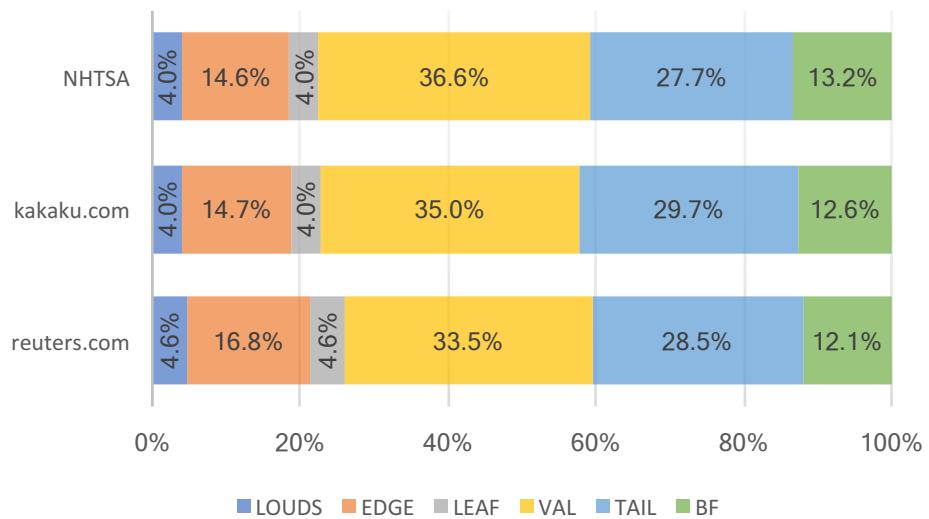


図 6.12: オンライン LOUDS トライ単体のメモリ内訳

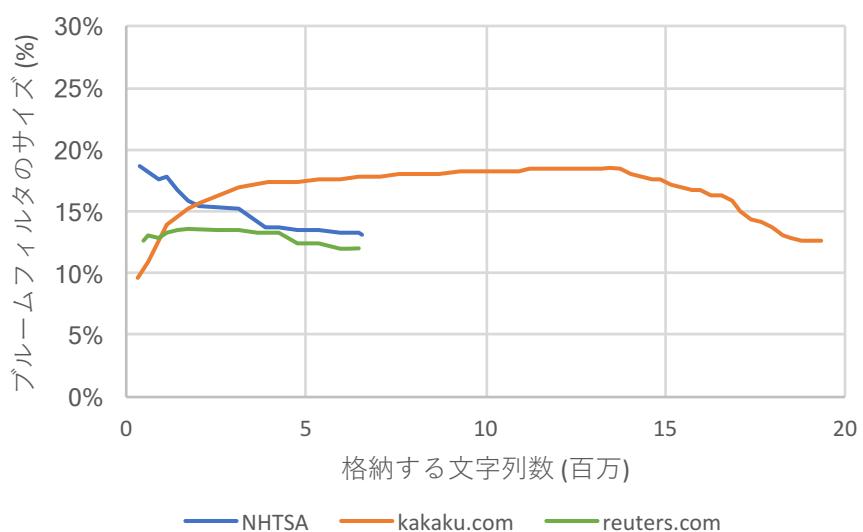


図 6.13: LOUDS トライに格納するキー数とブルームフィルタが占める割合

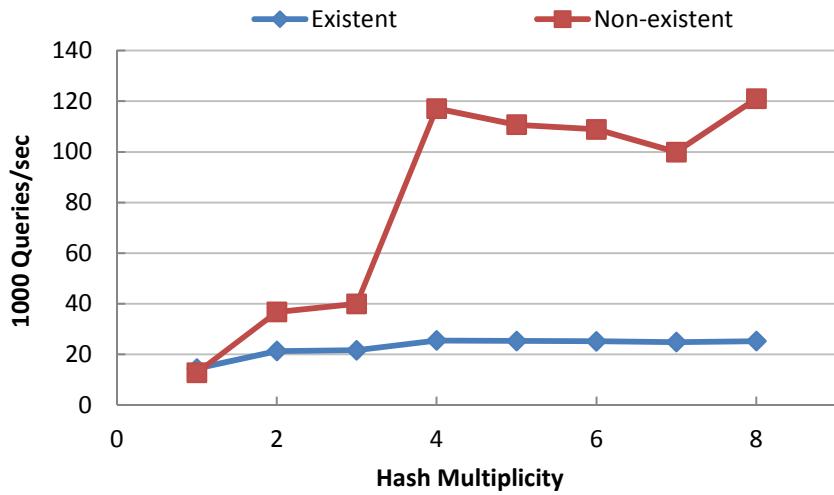


図 6.14: ブルームフィルタ付き LOUDS トライのハッシュ多重度と検索性能

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 6.14 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共に、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 2 では、ビット列のサイズを 2 のべき乗にすることで、ハッシュ値の剩余を高速に行う。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計算量が増える。図 6.14 では、同じビット数を持つハッシュ多重度 4 から 7 でスループットが下がっ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、性能は改善しなかった。

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。この実験は、4.4.3 項のモデルを検証する実験にあたる。図 6.15 は、ハッシュ多重度 4 の時に、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、バッファトライ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、バッファトライは個数に含まれない。縦軸は検索速度をログスケールで示す。

ブルームフィルタが無い場合には LOUDS トライの数が増えるに従って検索速度は

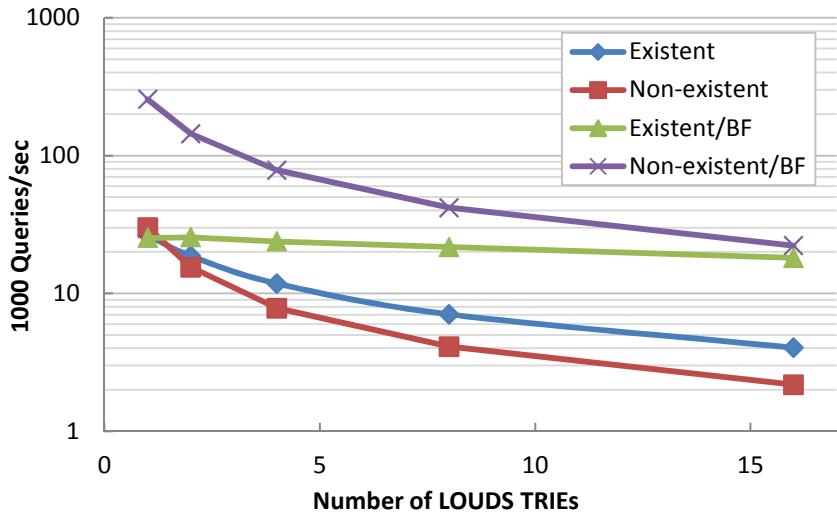


図 6.15: LOUDS トライの個数と検索速度の関係 (実験結果)

大きく低下する。一方、ブルームフィルタを付加した場合には、検索キーが含まれるときには大きな速度の低下は見られない。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高くなる。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、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 4.4.3 項で計算によって求めた図 4.10 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効果もある。

6.12 アプリケーションを想定したブルームフィルタの効果測定

次に、3 章のアプリケーションを想定し、6.2 節の実験プログラムを用いてブルームフィルタの効果を測定する。LOUDS トライやブルームフィルタの効果を測定するため、OLT1 のハッシュテーブルは無効とする。実験環境には E1 を用いる。データには NHSTA を用いる。

本節の実験では、LOUDS トライの個数、およびブルームフィルタの有無とアプリケー

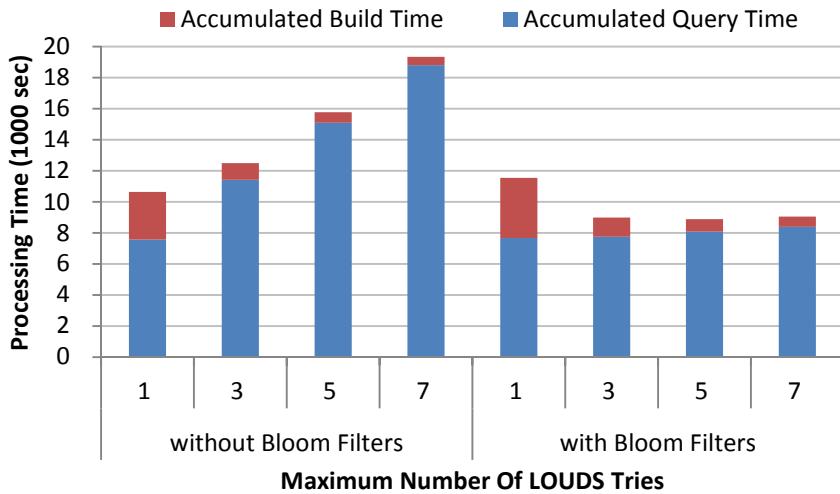


図 6.16: ブルームフィルタによる LOUDS トライ構築と検索の高速化 (NHTSA)

ションの処理時間の関係を調べる。このため、LOUDS トライの最大の個数 (Maximum Number of LOUDS Tries) を 1 から 7 に変化させて、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無いケース (without Bloom Filters) のそれについて、6.2 節の実験プログラムの完了までにかかる時間を計測する。LOUDS トライの最大個数はマージが実行される LOUDS トライの個数によって決まるため、LOUDS トライの最大個数が少ないものはマージ回数が多くなる。処理にかかる時間は、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。

6.9 節で定めたように、バッファトライのサイズはいずれもキー数で 4 万個とし、4 万個の入力ごとに LOUDS トライが構築される。

図 6.16 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。

この実験で使用した NHSTA のデータセットでは、検索と追加の両方が実行されるユニークなキーの入力は全体の約 2.56% であり、処理の 97.44% が検索のみを実行する重複したキーの入力であった。

この条件下では、ブルームフィルタが無い場合には、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には、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、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。

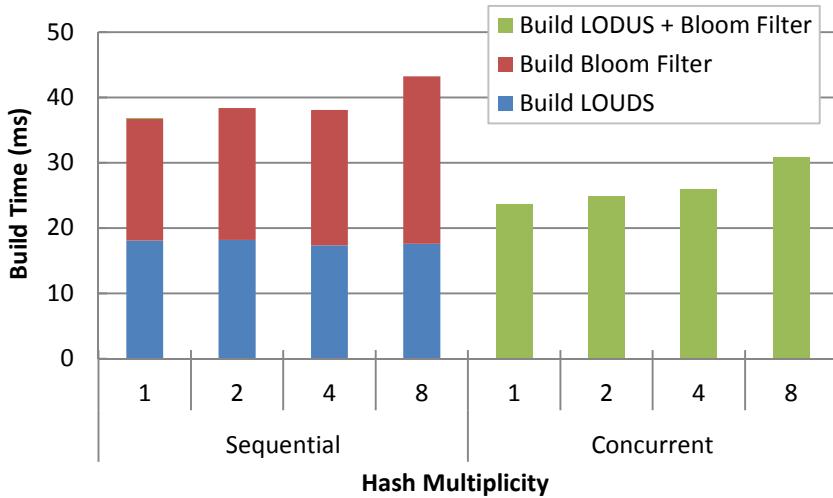


図 6.17: LOUDS とブルームフィルタの並行構築による LOUDS 構築の高速化

この実験の中でもっと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。

6.13 ブルームフィルタ構築時間の計測

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る。提案手法では、LOUDS トライを構築する幅優先走査時にブルームフィルタの生成も同時にを行うことで全体の構築時間を短縮している。この実験ではこれを確認する。

まず、予備実験として、LOUDS トライを単体で構築する際の効果を計測する。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、LOUDS トライ構築用のバッファとして使用する二分探索トライに入力して、LOUDS トライとその LOUDS トライに対応するブルームフィルタをそれぞれ 1 つ作るときの時間を計測する。実験環境には E1 を用いる。

図 6.17 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重度ごとに比較したグラフである。

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

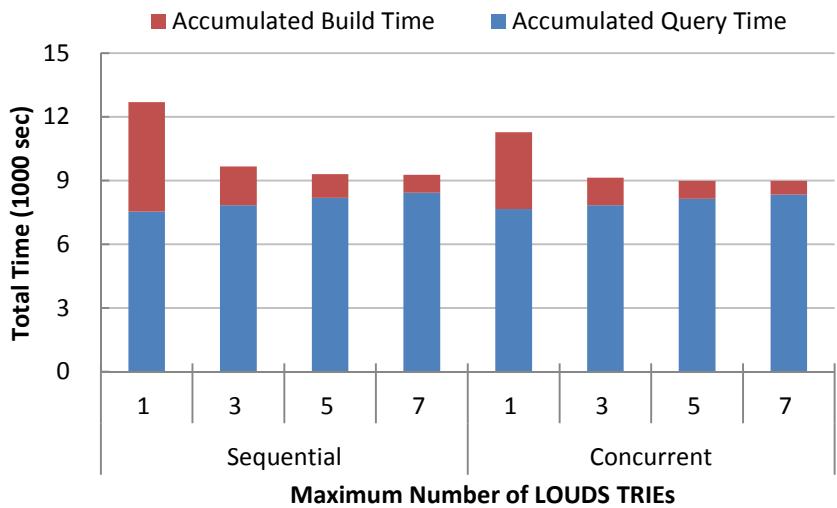


図 6.18: LOUDS とブルームフィルタの並行構築による LOUDS 構築と検索の高速化

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。

次に、アプリケーションを想定して 6.2 節の実験プログラムを用いて計測を行う。この実験には、NHTSA のデータセットに含まれる 2.4 億件のキーワードのストリームの全体を使用する。ここでは、LOUDS トライやブルームフィルタの効果を測定するため、OLT1 のハッシュテーブルは無効とする。

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に、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 6.18 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。

この実験で使用した NHSTA のデータセットでは、検索と追加の両方が実行されるユニークなキーの入力は全体の約 2.56% であり、処理の 97.44% が検索のみを実行する重複したキーの入力であった。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.

6.14 OLT2 の評価実験

本節では, 提案手法である OLT2 を実験により評価する. この実験は, 6.5 節で行った実験と共に実験環境, 実験手法, およびデータセットを用いる. 比較対象として用いる, OLT1, ハッシュテーブル (Hash), ダブルアレイトライ (DA), それらの組み合わせ (Hash + DA), およびハッシュテーブルと二次記憶の組み合わせ (Hash+SSD) による各データストアの計測データも 6.5 節に示したものと同じものである.

使用可能メモリ量を表 6.1 で示した値に設定する. OLT2 ではハッシュテーブル, およびバッファトライのサイズは使用可能メモリ量から自動的に計算されるため, OLT1 のように事前に割り当てる必要はない. OLT2 の自律的メモリ割り当てに必要なメモリ使用量の測定は 1 秒ごとに行う. また, バッファモードへの切り替えは, 残り使用可能メモリが当初の残り使用可能メモリの 50% になった時に行う. 5.1 節で述べたマージンは 5% とする.

図 6.19, 図 6.20 および図 6.21 は, 本節の実験結果をデータセットごとにグラフに示したものである.

この実験も 6.5 節で行った実験と同様に, 各手法には使用できる限界の文字列数がある. この実験結果では, ハッシュテーブルに入りきらない領域では, 比較対象のデータストアの中で OLT2 が最もスループットが高い. また, 同じ使用可能メモリ量の制限のもとでは, 格納できる文字列数が OLT1 と同様に多い. したがって, OLT2 は, 格納すべき文字列数に対して使用メモリ量が少なく, ハッシュテーブルが使用できない場合に有用性がある.

図 6.20 では文字列数が少ない時でもハッシュテーブルよりもスループットが高い範囲がある. これは, OLT2 が使用可能メモリ量をあらかじめ知っているため, ハッシュテーブルのサイズを適切に設定できるためである. これによって, 再ハッシュの回数が押さえられる. このように, 高いスループットの範囲では, 再ハッシュの回数が平均スループットに影響を及ぼす場合がある.

6.15 自律的メモリ割り当ての観測

本節では, OLT2 の自律的なメモリ割り当ての様子を詳しく見るため, 6.14 節の実験で, OLT2 が各データ構造にどのようにメモリを割り当てたかを時系列に沿って観

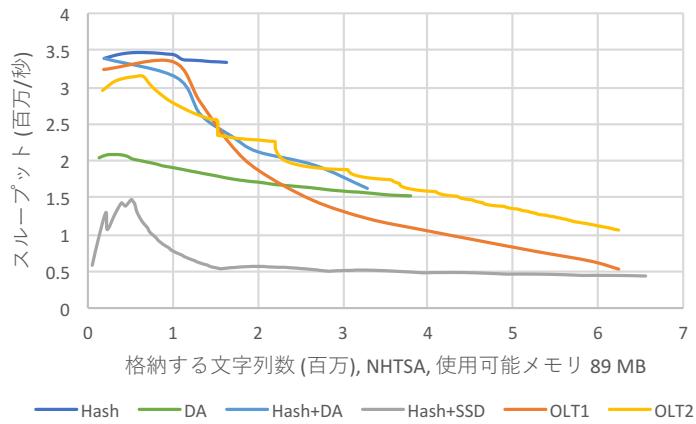


図 6.19: OLT2 の格納文字列数とスループットの関係 (NHTSA 2.5 億件, 89MB)

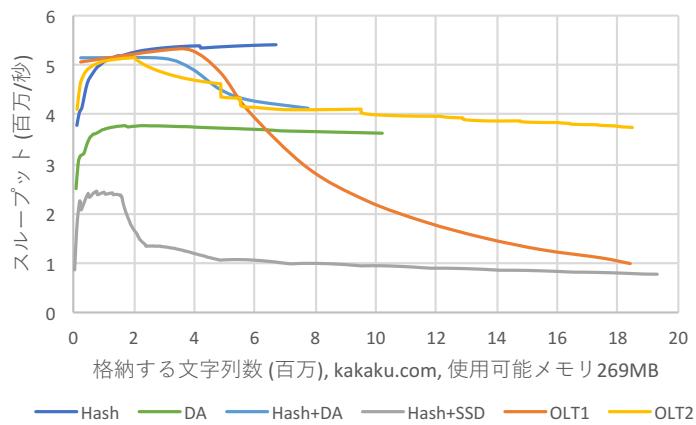


図 6.20: OLT2 の格納文字列数とスループットの関係 (kakaku.com 15 億件, 269MB)

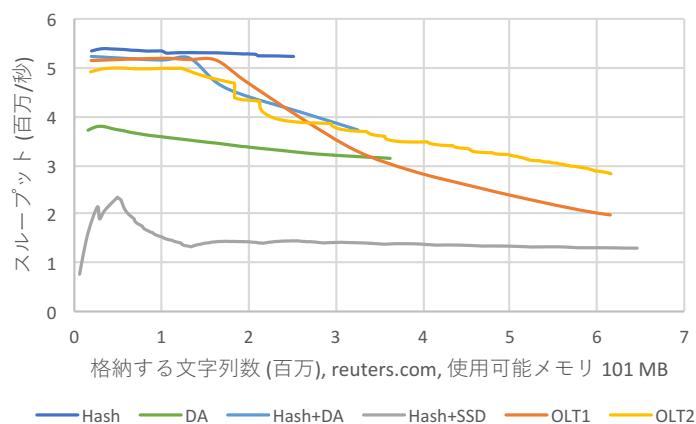


図 6.21: OLT2 の格納文字列数とスループットの関係 (reuters.com 10 億件, 101MB)

測する。図 6.22, 図 6.24, および図 6.26 に、それぞれ、NHTSA, kakaku.com, および reuters.com の場合の実験中のメモリ使用量の推移をデータ構造ごとに示す。

図中のメモリ使用量が下がる箇所ではデータ構造の変換が起こっている。NHTSA のケースでは、LOUDS トライが 3 番目にメモリ使用量が下がる箇所で作成されている。それまでのメモリ使用量が下がる箇所では、ハッシュテーブルから二分探索トライへの変換が行われている。メモリ使用量の下がる直前に見られる一時的なメモリ使用量の増加は、データ構造の変換のために一時的に必要なメモリ使用を表している。ハッシュテーブルからバッファトライへの変換時は、一時的に両方のデータ構造でデータを保持するため、一時的に多くのメモリが使用される。また、LOUDS トライを作成するときには、両方のデータ構造でデータを保持することに加えて、幅優先走査を行うキューなどの補助的なデータ構造もメモリを使用する。そのため、構築されるデータ構造よりも大きなサイズのメモリが必要になる。LOUDS トライ構築に一時的に使用されたメモリは構築後にすぐに解放され、ハッシュテーブルなどに割り当てられる。

6.6 節で観測した OLT1 のメモリ使用量の推移では、いずれもメモリ使用量は時間とともに増加し、全ての文字列を格納した時点が最もメモリ使用量が多い。一方、OLT2 では開始後の早い時間にハッシュテーブルによって使用可能メモリいっぱいまでメモリが使用される。処理中の残り使用可能メモリがハッシュテーブルに割り当てられることで、メモリが効率よく使用されることがわかる。

図 6.23 は、OLT2 でデータセット NHTSA を処理した場合のスループットの変化を表している。図中のスループットの値は、計測された毎秒の平均を示している。スループットの値は実験開始時、すべてのキーがハッシュテーブルで処理されるため最も高く、OLT2 がバッファモードに切り替わったところで減少する。データが LOUDS トライに変換されるとキャッシュがなくなるため、いったんスループットは大きく低下する。その後、キャッシュが満たされるにつれて性能が回復する。

図 6.25 と、図 6.27 は、それぞれ kakaku.com と reuters.com を OLT2 で処理した場合のスループットの変化を表している。いずれも、NHTSA のケースと同様の形状を描いている。ハッシュテーブルが破棄された後のスループット回復の傾きは、データセットにより異なる。

6.6 節で観測した OLT1 のスループットの推移と比較すると、キーがハッシュテーブル以外のデータ構造に格納され始めた後のスループットの減衰が OLT2 の方が緩やかである。これは、ハッシュテーブルに割り当てられるメモリ量が増えてキャッシュヒット率が向上したことによる。ほとんどの使用可能メモリが LOUDS トライに割り当たられる実験の終わり近くでは、ハッシュテーブルのサイズは同程度であり、スループットは同程度の値になる。

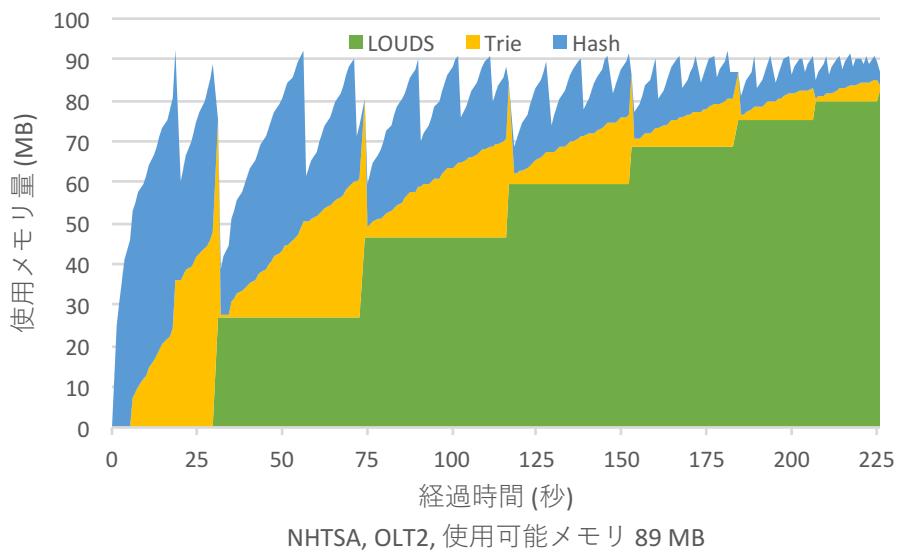


図 6.22: OLT2 の使用メモリ量の推移 (NHTSA 2.5 億件, 89MB)

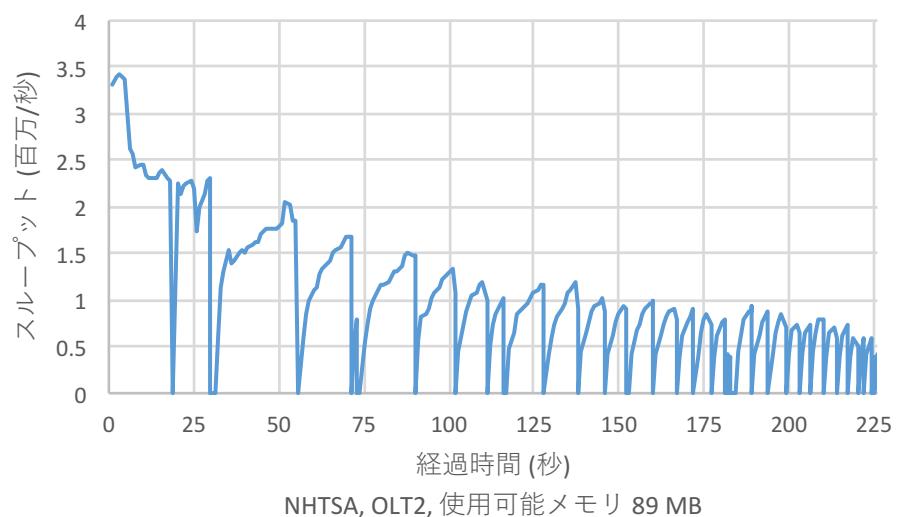


図 6.23: OLT2 のスループットの変化 (NHTSA 2.5 億件, 89MB)

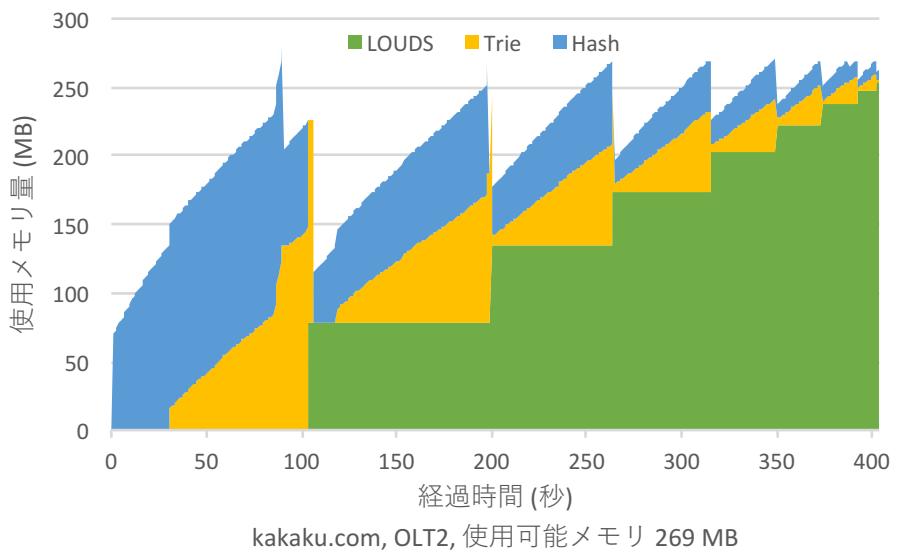


図 6.24: OLT2 の使用メモリ量の変化 (kakaku.com 15 億件, 269MB)

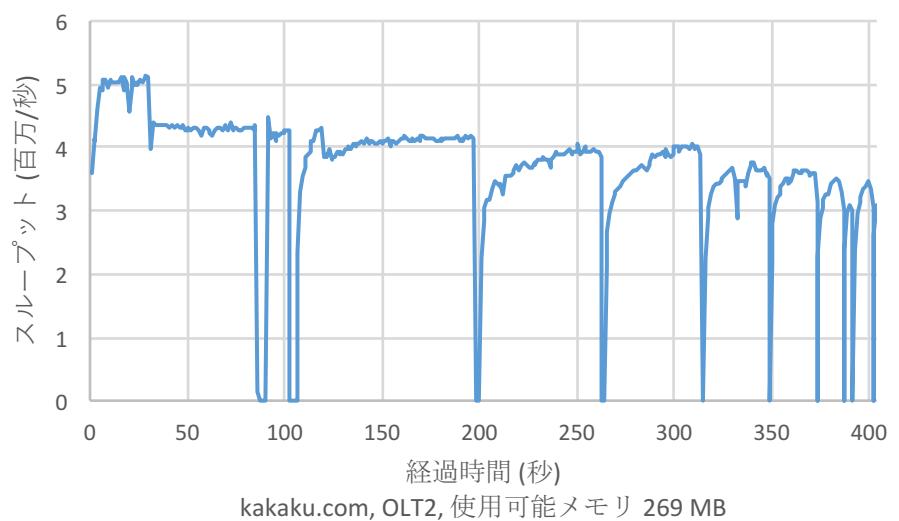


図 6.25: OLT2 のスループットの変化 (kakaku.com 15 億件, 269MB)

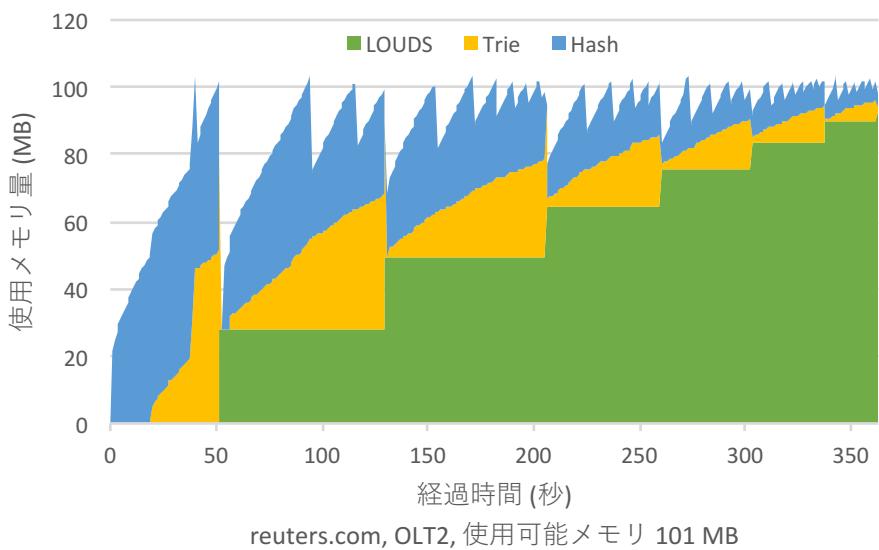


図 6.26: OLT2 の使用メモリ量の変化 (reuters.com 10 億件, 101MB)

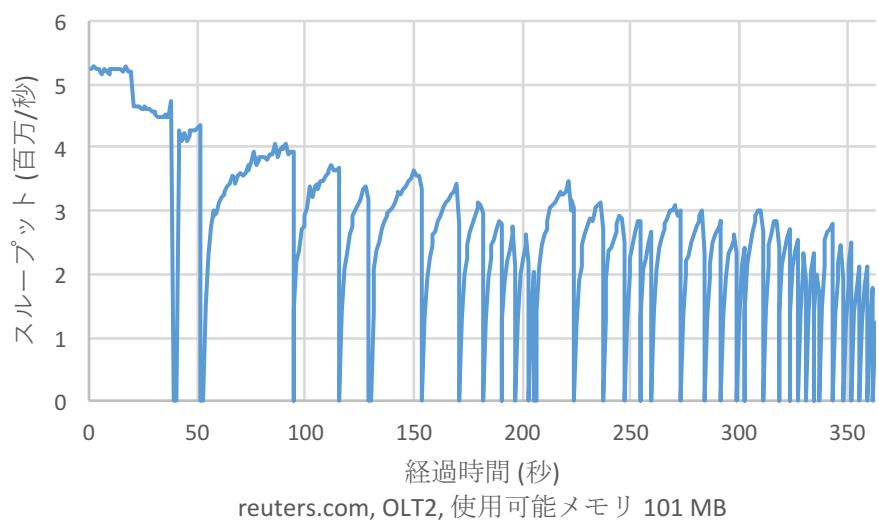


図 6.27: OLT2 のスループットの変化 (reuters.com 10 億件, 101MB)

6.16 OLT2 の使用可能メモリとスループットの関係

本節では、6.7 節と同様に、OLT2 の使用可能メモリとスループットの関係を調べる。入力データには、表 1.1 の 3 つのデータセットを用いる。実験環境には E2 を用いる。実験は、OLT2 の使用可能メモリ量を変化させ、そこに、6.2 節のプログラムを用いてデータを全て入力する。使用可能メモリ量は 6.7 節と同様に設定した。

表 6.5 に実験結果を示す。表の各列は、データセットごとに、使用可能メモリ（メモリ）と平均スループット（件/秒）を表している。

OLT2 は、コンパクトなデータ構造に割り当てたメモリ以外の残り使用可能メモリを全て高速なハッシュテーブルに自律的に割り当てる。そのため、使用可能メモリ量が多くなると、スループットはより高くなる。実際に OLT2 が使用したメモリは使用可能メモリの 95% から 99% であった。

全体の傾向として、同じ使用可能メモリの時、OLT2 は OLT1 よりも高速である。特に、使用可能メモリが小さいほどこの傾向が大きい。しかし、データセットによってこの差には違いがある。OLT1 に対する OLT2 のスループットの比は NHTSA では 1.4 から 2.1、kakaku.com では 1.6 から 4.3、reuters.com では 0.96 から 1.4 であった。

6.17 OLT2 によるより大きなデータの処理

本節では、6.8 節と同様に、より大きなデータセットによる OLT2 の実験結果を示す。使用可能メモリ量を 8GB とし、実験環境 E2 を使用して表 6.4 のデータセット Wikipedia-en を OLT2 に格納する実験を行なった。

この実験のメモリ使用量の推移を図 6.28 に示す。この図の横軸は経過時間（時間）、縦軸はハッシュテーブル（Hash）、バッファトライ（Trie）および LOUDS トライ（LOUDS）の各使用メモリ量（GB）を積み上げグラフで表している。

図 6.28 は、軸のスケール以外は基本的に他のデータセットを処理した場合の図 6.22、図 6.24、および、図 6.26 と同様である。アルゴリズムから、LOUDS トライ構築回数は使

表 6.5: OLT2 の使用メモリ量とスループットの関係

NHTSA		kakaku.com		reuters.com	
メモリ	件/秒	メモリ	件/秒	メモリ	件/秒
89 MB	79 万	269 MB	339 万	101 MB	244 万
134 MB	158 万	404 MB	396 万	152 MB	351 万
178 MB	182 万	538 MB	405 万	202 MB	394 万
223 MB	201 万	673 MB	401 万	253 MB	408 万

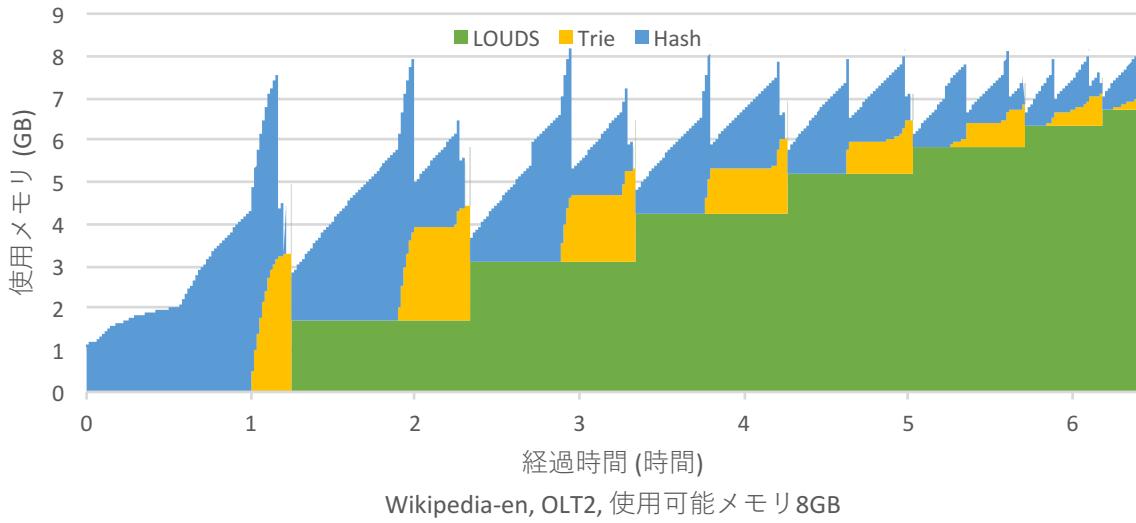


図 6.28: OLT2 の使用メモリ量の推移 (Wikipeida-en 618 億件, 使用可能メモリ 8GB)

用可能メモリ量と格納する文字列の数の比から決まる。この実験では LOUDS トライ構築回数は 7 回であり、他のデータセットを OLT2 で処理する場合と大きな差がない。

6.18 キャッシュ置き換えアルゴリズムに関する考察

本研究では、高頻度少数の文字列を高速なデータ構造に、低頻度多数の文字列をコンパクトなデータ構造に格納することによって、コンパクトでありながらできるだけ高いスループットを実現しようとしている。しかし、事前にどの文字列がより頻繁に出現するかはわからないため、正確に高頻度の文字列を選択することはできない。

どの文字列が高速なデータ構造に格納されるかは、入力データと、高速なデータ構造に格納する文字列を選択するキャッシュ置き換えアルゴリズムによる。4.8 節に示したモデルは、OLT1 における理想的なキャッシュ置き換えアルゴリズムによる性能を計算している。この理想的なキャッシュ置き換えアルゴリズムは、現実とは異なり、あらかじめどの文字列の出現頻度が高いかを知ってキャッシュを置き換えている。また、キャッシュ置き換えアルゴリズムの計算とメモリのオーバーヘッドがないものとしている。

実際には、キャッシュ置き換えアルゴリズムのオーバーヘッドは全体の性能に大きな影響を与える。そのため、できるだけオーバーヘッドの小さなアルゴリズムが良い。

Least Frequently Used (LFU) は、過去の出現頻度を数えて最も出現頻度の少ない文字列を置き換えるキャッシュ置き換えアルゴリズムである。その実現には、出現頻度のカウントや最小出現頻度の文字列を選択するため、OLT1 および OLT2 にとってオーバーヘッドが大きい。

バーへッドが大きい。出現頻度を近似的に求める Lossy Counting [47] を用いて LFU を実現すると、Least Recently Used (LRU) よりも低い性能になった。

オーバーへッドが小さいキャッシュ置き換えアルゴリズムとしては LRU が知られている。LRU の実装の中で、擬似 LRU は性能やメモリのオーバーへッドが最も小さく、OLT1 に組み込んだ際に最も高い性能になった。そのため、OLT1 では、4.7 節で述べたように、キャッシュ置き換えアルゴリズムに擬似 LRU を用いている。

OLT2 では、5.1 節で述べたように、残り使用可能メモリ量が少なくなるたびにハッシュテーブルを全てクリアする戦略をとる。すなわち、複雑なキャッシュ置き換えアルゴリズムを用いず、単に、一定期間の間、出現した文字列を全てハッシュテーブルに格納する。これによってキャッシュ置き換えアルゴリズムのオーバーへッドをなくしている。本研究の対象とするアプリケーションでは、このような単純な方法でも高い確率で高頻度の文字列を含む。擬似 LRU を用いて一定数の高頻度の文字列をハッシュテーブルに残す方式と比較すると、キャッシュ置き換えアルゴリズムなしの場合の方が高い性能になった。

第7章 おわりに

本研究では文字列をキーとするインメモリデータストアのためのコンパクトなデータ構造を実現する手法を提案した。文字列キーを主記憶に格納して重複排除やカウンティングを行う際、従来の手法では、主に高速だがメモリ効率が悪いハッシュテーブルが使われていた。そのため、メモリに入りきらない場合、二次記憶を用いるので、急激に性能が低下するという問題があった。本論文では、高速なデータ構造としてハッシュテーブルと、コンパクトなデータ構造として LOUDS トライを組み合わせて利用することを提案した。実験結果により、使用可能メモリ量が限られデータがハッシュテーブルに入りきらない場合には、提案手法が既存の手法よりも高いスループットになることを示した。

本論文の1章では、本研究の背景、解決すべき問題、および問題を解決する提案手法の概要を示した。本研究は、件数の多い文字列データに対して重複排除やカウンティングを行うために、文字列をキーとするインメモリデータストアが用いられるることを背景としている。インメモリデータストアには、二次記憶にはない高い性能が求められることに加え、使用できる主記憶の量に制限があることから、高いメモリ効率が求められる。本論文では、重複排除やカウンティングを必要とするアプリケーションの多くで文字列の出現頻度分布に偏りがあることに着目し、少数の高頻度の文字列を高速なデータ構造に、多数の低頻度の文字列をコンパクトなデータ構造に格納することで、コンパクトでありながら高い性能を発揮するデータ構造を提案した。このデータ構造はインメモリデータストアの内部実装として使うことができる。コンパクトなデータ構造には理論上最小限に近いメモリ使用量である LOUDS トライを用いる。LOUDS トライを動的なデータストアとして利用するため、本研究では以下の2つの問題を解いた：

問題1 LOUDS トライは静的なデータ構造であり値を追加できない。

問題2 LOUDS トライ、バッファ、ハッシュテーブル等のデータ構造に対する最適なメモリ割当は入力データに依存しているため、オンラインで最適化問題として解くことは難しい。

2章では関連する研究について述べた。既存のインメモリデータストアの多くは、内部のデータ構造としてハッシュテーブルを用いているものが多い。ハッシュテーブルはメモリ効率が悪いためキーの件数が多い場合には使用できない。また、簡潔データ構造

はメモリ効率が良いが検索性能は低いため、スループットの要求が高い場合には使用できない。二分探索トライやダブルアレイトライはそれらの中間に位置する。SILT や LSM-Trie はフラッシュストレージにデータを置き、主記憶に索引を持つキーバリューストアである。これらのデータ構造には、利用可能なメモリに応じてスループットを調整する機能はない。

3 章では、本研究で想定する典型的なアプリケーションの例として、Apache Lucene を用いたテキスト解析システムと、そこで求められるインメモリデータストアの要件を示した。テキスト解析システムでは、ルールによって単語の組み合わせからなるフレーズを抽出し、文字列キーとしてインメモリデータストアに保持する。そこで求められる要件は、インタフェースとして文字列をキーとする put と get を持ち、アプリケーションに組み込まれたハッシュテーブルと置き換え可能であること、キーが値に対して長く、使用メモリの大部分を占めること、キーの出現頻度分布が高頻度少数と低頻度多数に分かれること、put されたら直ちに get に反映されること、get されたキーが存在しない時に put されること、事前にキーの数、および平均の長さを知ることができること、そして、制限されたメモリで動作し、実行中にメモリの使用状況が取得できることである。

4 章では、LOUDS トライのオンライン構築を実現する提案手法 OLT1 について述べた。本研究では、1 章であげた問題 1 を解くため、二分探索トライをバッファとするログ構造マージ木に類似した手法を用いた。これにより、静的データ構造である LOUDS トライをオンライン構築できるようにした。OLT1 は、複数の LOUDS トライを作成して順にリストに追加し、リストに含まれる LOUDS トライをマージする。リストが長くなると検索対象の LOUDS トライが増えるため検索性能が低下する。これはマージを行することで解消されるが、マージは計算コストの高い処理であるため頻繁には実行できない。OLT1 では、各 LOUDS トライにブルームフィルタを付加することで、リスト長が及ぼす検索性能への影響を低減する。また、中間バッファを用いてマージを行うと、マージ時のメモリ使用量が大きくなる問題がある。

しかし中間バッファを使わないマージアルゴリズムは複雑である。本研究では、仮想ノードという概念を導入し、複数のトライ木を 1 つのトライ木として扱えるようにする。これにより、簡潔なアルゴリズムでマージを可能にし、さらにブルームフィルタの構築も並行で行うようにした。

4 章では、OLT1 が高い性能を発揮する範囲を調べるために、格納する文字列数に対する OLT1 の性能のモデルを立て、出現頻度分布と入力文字列数を変化させて性能を計算した。このモデルから、OLT1 は、出現頻度分布が高頻度少数の文字列と低頻度多数の文字列に大きく偏っている場合に高い性能になることが分かった。

5 章では、自律的に、ハッシュテーブル、LOUDS トライ、および二分探索トライへメモリを割り当てる提案手法 OLT2 について述べた。OLT2 は、1 章であげた問題 2 を解くため、残り使用可能メモリを見ながら各データ構造に自律的にメモリを割り当てる

る。OLT2では、LOUDSトライと二分探索トライが使用するメモリ以外を全てハッシュテーブルによるキャッシュに割り当てる。残り使用可能メモリ量が小さくなると、キャッシュを破棄してLOUDSトライを構築する。このアルゴリズムによって、残されたメモリに応じてデータ構造をコンパクト化し、出現頻度の高いキーをハッシュテーブルにより多く格納できるようにする。

6章では、実験により提案手法OLT1、およびOLT2を評価した。使用可能メモリのサイズを固定し、入力する文字列の数を変化させてスループットを計測する実験を行った。計測対象のデータ構造は、OLT1、OLT2、ハッシュテーブル、ダブルアレイトライ、ハッシュテーブルとダブルアレイトライの組み合わせの5種類とした。また、参考のために二次記憶と主記憶中のハッシュテーブルをキャッシュとしている手法のスループットも測定した。OLT1はダブルアレイトライよりもスループットが低いが、ダブルアレイトライではメモリに入りきらない場合に有用性がある。OLT2は、ハッシュテーブルに入りきらない場合に5種類のデータ構造の中で最もスループットが高かった。いずれも、メモリに格納できる範囲では二次記憶を用いた手法よりもスループットが高かった。この結果より、使用可能メモリ量が限られ、ハッシュテーブルに入りきらないとき、対象とするアプリケーションでは、提案手法が既存の手法よりも高いスループットになることを示した。

OLT1、および、OLT2は、使用メモリ量に応じたスループットを持つモジュールである。本研究では、外部からそのモジュールが使用可能なメモリ量が与えられた時に、モジュール内部のメモリ割り当てを扱った。1つのプログラムを走らせる時に、そのような使用メモリ量に応じたスループットを持つモジュールを複数動作させることがある。たとえば、Apache Luceneを用いたテキスト解析システムにおいても、本研究で扱ったTaxonomyWriterCacheだけでなく、検索クエリ用のキャッシュを含むIndexSearcherや、転置索引の書き込みバッファを持つIndexWriterなどもある。OLT1、および、OLT2は、実行の途中でメモリを増やすだけでなく、ハッシュテーブルに割り当てられているメモリを減らすことも可能である。今後の研究の展開として、そのようなメモリ使用量に応じたスループットを持つモジュールを複数動作させた時のメモリ割り当ての問題を考えていきたい。この時、OLT2で用いた自律的な手法が適用できるのではないかと考えている。

メモリ割り当てをさらに効率的に行うためには、Java仮想計算機やオペレーティングシステムの改変も求められる。Java仮想計算機では、モジュール単位でのメモリ割り当てとガベージコレクションの機能が求められる。オペレーティングシステムでは、データ構造の変換時に一時的に利用するメモリがあると便利である。また、OLT1やOLT2は、利用可能なメモリ使用量の増減に簡単に追随できる。今後は、そのようなメモリ使用量の増減に追随可能なアプリケーションを効率良く動作させるようなオペレーティングシステムが求められる。

謝辞

本論文の研究は、筆者が筑波大学博士課程後期システム情報工学研究科コンピュータサイエンス専攻に在籍中、新城靖准教授のご指導のもとに行われました。

新城准教授には論文の執筆にあたって非常にきめ細やかで丁寧なご指導をいただきました。この導きなしには論文の完成はありませんでした。また、海外からリモートで博士課程を進めるために大変お世話になりました。あらためて感謝の意を表します。

本論文をまとめるにあたり、建部修見教授、山本幹雄教授、中田秀基教授、天笠俊之准教授、および鈴木伸崇准教授に研究の内容に関する多くのご指摘やご指導をいただきました。また、板野肯三名誉教授、北川博之教授、および加藤和彦教授にも多大なご指導をいただきました。筑波大学の誇る第一線の先生方にご指導いただけたことに感謝いたします。

20年来の先輩である中井央准教授には筑波大学博士課程後期への入学の際相談にのっていただきました。ありがとうございました。

本研究は、筆者が日本アイ・ビー・エム株式会社ソフトウェア開発研究所に在籍中に、同東京基礎研究所の吉田一星氏、海野裕也氏、およびRaymond Rudy氏らとともに行った共同研究を元にしています。彼らのような才能あふれる研究者と一緒に仕事ができたことを誇りに思います。

また、研究開始当時上司であった濱田誠治氏率いるIBMソフトウェア開発研究所のチームには、研究を進めるうえで多大なサポートをいただきました。感謝するとともに、チームのいっそうの発展をお祈りしています。

本研究の社内誌ProVISIONへの掲載に際して杉本和敏氏にご指導いただきました。大変お世話になりました。

I would like to say thank you to Sola Ania, Roman Valiusenko and Yosuke Ozawa for revising my paper for ICICS 2014. Their proof reading was the great help to be accepted as the best research paper award.

藤森茂雄氏には本論文の文献リストの修正を手伝っていただきました。ありがとうございました。

そのほか、多くの方の支えで本研究は成り立ちました。ご支援やご指導をいたしましたすべての方々に感謝いたします。

参考文献

- [1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 337–350, Oakland, CA, May 2015. USENIX Association.
- [2] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. Storage management in AsterixDB. *Proceedings of the VLDB Endowment*, Vol. 7, No. 10, pp. 841–852, 2014.
- [3] Amazon Web Services, Inc. Amazon redshift. <https://aws.amazon.com/redshift/>. Accessed: December 28, 2016.
- [4] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI ’10*, pp. 29–29. USENIX Association, 2010.
- [5] Junichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, Vol. 15, pp. 1066–1077, September 1989.
- [6] The Apache Software Foundation. *The Apache HBase Book*, October 2010.
- [7] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, Vol. 1 Fase. 3, , September 1972.
- [8] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, Vol. 43, pp. 275–292, December 2005.
- [9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Vol. 13, pp. 422–426, July 1970.

- [10] JBoss Community Bootstrap and Awestruct. Infinispan. <http://infinispan.org>. Accessed: December 22, 2016.
- [11] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Vol. 1, pp. 126–134 vol.1, March 1999.
- [12] 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.
- [13] Maxime Crochemore and Renaud Verín. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching*, pp. 116–129. Springer-Verlag, 1997.
- [14] Jeff Dean and Sanjay Ghemawat. LevelDB. <https://code.google.com/p/leveldb/>. Accessed: August 30, 2015.
- [15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, Vol. 51, No. 1, pp. 107–113, January 2008.
- [16] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment*, Vol. 3, No. 1-2, pp. 1414–1425, September 2010.
- [17] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pp. 25–36. ACM, 2011.
- [18] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI '13*, pp. 371–384. USENIX, 2013.
- [19] 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.

- [20] Arash Farzan and J. Ian Munro. Succinct representations of arbitrary graphs. In *Proceedings of the 16th annual European symposium on Algorithms*, pp. 15–17, September 2008.
- [21] Free Software Foundation. Linux kernel. <https://www.kernel.org/>. Accessed: August 30, 2015.
- [22] The Apache Software Foundation. Apache Lucene. <http://lucene.apache.org/>. Accessed: August 30, 2015.
- [23] Wikimedia Foundation. Static html dump of wikipedia (en). https://dumps.wikimedia.org/other/static_html_dumps/current/en/wikipedia-en-html.tar.7z. Accessed: December 4, 2016.
- [24] Edward Fredkin. Trie memory. *Communications of the ACM*, Vol. 3, pp. 490–499, September 1960.
- [25] Chikara Furusawa and Kunihiko Kaneko. Zipf’s law in gene expression. *Physical review letters*, Vol. 90, No. 8, p. 088102, 2003.
- [26] Xavier Gabaix. Zipf’s law for cities: an explanation. *Quarterly journal of Economics*, pp. 739–767, 1999.
- [27] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, Vol. 35, No. 2, pp. 378–407, August 2005.
- [28] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Distributed Computing*, pp. 350–364. Springer, 2008.
- [29] Google Inc. Google compute engine. <https://cloud.google.com/compute/>. Accessed: August 30, 2015.
- [30] Google Inc. Google IME. <http://www.google.co.jp/ime/>. Accessed: August 30, 2015.
- [31] Guy J. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988. AAI8918056.
- [32] Guy J. 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.

- [33] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pp. 16–25. Citeseer, 1997.
- [34] Christopher Jermaine, Edward Omiecinski, and Wai G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, Vol. 16, No. 4, pp. 417–437, October 2007.
- [35] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pp. 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [36] Dong K. Kim, Joong C. Na Na, Ji E. 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, pp. 125–143, 2005.
- [37] 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. 情報処理学会論文誌: データベース (TOD), Vol. 4, No. 4, pp. 1–10, December 2011.
- [38] 小柳光生, 吉田一星, 海野裕也, 新城靖. 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. 情報処理学会論文誌: コンピューティングシステム (ACS), Vol. 5, No. 2, pp. 1–9, March 2012.
- [39] Teruo Koyanagi and Yasushi Shinjo. A fast and compact hybrid memory resident datastore for text analytics with autonomic memory allocation. In *In Proceedings of IEEE International Conference on Information and Computational Systems 2014 (ICICS2014)*, April 2014.
- [40] Taku Kudo. MeCab: Yet another part-of-speech and morphological analyzer. <http://taku910.github.io/mecab/>. Accessed: August 31, 2015.
- [41] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, pp. 35–40, April 2010.
- [42] Jonathan Leibiusky. Jedis. <https://github.com/xetorthio/jedis>. Accessed: December 28, 2016.

- [43] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 6:1–6:15, New York, NY, USA, 2014. ACM.
- [44] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 1–13. ACM, 2011.
- [45] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pp. 429–444. USENIX Association, April 2014.
- [46] The Economist Newspaper Limited. Data, data everywhere. *The Economist*, <http://www.economist.com/node/15557443>, February 2010. Accessed: August 30, 2015.
- [47] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pp. 346–357. VLDB Endowment, 2002.
- [48] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pp. 115–130, Berkeley, CA, USA, 2003. USENIX Association.
- [49] Memcached. Memcached. <http://memcached.org/>. Accessed: August 30, 2015.
- [50] Sparsh Mittal and Jeffrey S Vetter. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 5, pp. 1524–1536, 2016.
- [51] MongoDB, Inc. Mongodb. <https://www.mongodb.com/>. Accessed: December 28, 2016.
- [52] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Computing Research Repository*, Vol. abs/1108.1983, , August 2011.

- [53] J. Ian Munro and Venkatesh 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, october 1997.
- [54] Doron Nakar and Shlomo Weiss. Selective main memory compression by identifying program phase changes. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, pp. 96–101. ACM, 2004.
- [55] Markus F.X.J. Oberhumer. LZO real-time data compression library. <http://www.oberhumer.com/opensource/lzo>. Accessed: November 8, 2016.
- [56] U.S. National Library of Medicine. Fact sheet: Medline. <http://www.nlm.nih.gov/pubs/factsheets/medline.html>. Accessed: August 30, 2015.
- [57] U.S. Department of Transportation. National highway traffic safety administration. <http://www.nhtsa.gov/>. Accessed: August 30, 2015.
- [58] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, Vol. 33, No. 4, pp. 351–385, 1996.
- [59] Rasmus Pagh and Flemming Friche Rodlerb. Cuckoo hashing. *Journal of Algorithms*, Vol. 51, pp. 122–144, May 2004.
- [60] Naila Rahman and Rajeev Raman. Engineering the LOUDS succinct tree representation. In *In Proceedings of the 5th International Workshop on Experimental Algorithms*, p. 145. Springer, 2006.
- [61] Red Hat, Inc. Wildfly. <http://wildfly.org/>. Accessed: December 26, 2016.
- [62] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST ’14, pp. 1–16. USENIX, 2014.
- [63] 定兼邦彦. 括弧列の簡単・簡潔な表現法. 電子情報通信学会技術研究報告, Vol. 108, No. 237, pp. 33–40, October 2008. COMP2008-38.
- [64] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th International Conference on Algorithms and Computation*, ISAAC ’00, pp. 410–421, London, UK, UK, 2000. Springer-Verlag.

- [65] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pp. 225–232, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.
- [66] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, Vol. 48, No. 2, pp. 294–313, September 2003.
- [67] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, Vol. 41, No. 4, pp. 589–607, December 2007.
- [68] Kunihiko Sadakane. Dynamic succinct ordinal trees. *IEICE technical report. Theoretical Foundations of Computing*, Vol. 109, No. 9, pp. 37–41, April 2009.
- [69] 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.
- [70] Dustin Sallings. Spymemcached. <https://github.com/couchbase/spymemcached>. Accessed: December 25, 2016.
- [71] Salvatore Sanfilippo. Redis. <http://redis.io/>. Accessed: August 30, 2015.
- [72] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ACM '12, pp. 217–228, 2012.
- [73] Amazon Web Services. Amazon elastic computing cloud (EC2). <http://aws.amazon.com/ec2/>. Accessed: August 30, 2015.
- [74] Zeev Tarantov and Steinar H. Gunderson. *Snappy*. Google Inc.
- [75] 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. 『大規模情報コンテンツ時代の高度ICT専門職業人育成』事業最終報告書. http://www.cs.tsukuba.ac.jp/dcon/daicon_web2013Jul05.pdf, March 2013. Accessed: August 30, 2015.
- [76] Twitter. Twitter company facts. <https://about.twitter.com/company>. Accessed: August 30, 2015.
- [77] Naohiko Uramoto, Hirofumi Matsuzawa, Tohru Nagano, Akiko Murakami, Hironori Takeuchi, and Koichi Takeda. A text-mining system for knowledge discovery from biomedical documents. *IBM Systems Journal*, Vol. 43, No. 3, pp. 516–533, 2004.

- [78] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, Vol. 393, No. 6684, pp. 440–442, 1998.
- [79] Wikipedia. History of wikipedia. https://en.wikipedia.org/wiki/History_of_Wikipedia. Accessed: August 30, 2015.
- [80] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pp. 8–8, Berkeley, CA, USA, 1999. USENIX Association.
- [81] Worldwidewebsite.com. The size of the world wide web. <http://www.worldwidewebsite.com/>. Accessed: August 30, 2015.
- [82] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 71–82, Santa Clara, CA, Jul 2015. USENIX Association.
- [83] Lei Yang, Robert P Dick, Haris Lekatsas, and Srimat Chakradhar. Online memory compression for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 9, No. 3, p. 27, 2010.
- [84] Susumu Yata. Darts clone. <https://code.google.com/p/darts-clone/>. Accessed: August 30, 2015.
- [85] Susumu Yata. Taiju: C++ library for succinct representations of trie. <https://code.google.com/p/taiju/>. Accessed: August 30, 2015.
- [86] Naoki Yoshinaga and Masaru Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proceedings of The 25th International Conference on Computational Linguistics (COLING 2014)*, COLING 2014, pp. 1091–1102, 2014.
- [87] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2. USENIX Association, 2012.
- [88] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, Vol. 10, p. 10, 2010.

- [89] George K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.
- [90] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions On Information Theory*, Vol. 23, No. 3, pp. 337–343, 1977.

付録A 既存のインメモリデータストア

参考のため、既存の代表的なインメモリデータストアのメモリ効率を測定した。表 A.1 に、既存のインメモリデータストアを用いて重複排除を実装した場合の使用メモリ量(メモリ)と平均のスループット(件/秒)を示す。入力するデータセットには表 1.1 に示したデータセットを用いた。比較のため、表 6.3 および表 6.5 に示した OLT1 および OLT2 の使用可能メモリ量とスループットも示す。

使用したソフトウェアは、Memcached (memcached-1.4.4) [49], Redis (redis-3.2.6) [71]、および Infinispan (infinispan-8.2.5-Final) [10] である。これらのインメモリデータストアでは、内部のデータ構造にハッシュテーブルが使われている。重複排除のプログラムには、シングルスレッドで動作する 6.2 節のプログラムと同等のものを用いるが、後に説明するように、各インメモリデータストアに合わせた実装を用いている。

Memcached と Redis では、get の結果をクライアント側で同期して待つと、通信のレイテンシがスループットに含まれてしまう。組込型のデータストアともある程度比較できるように、できるだけ通信のレイテンシを隠蔽する非同期 API を用いた。これは 3 章で述べた、put した値を直ちに get に反映するアプリケーションの要求を満たしていないが、Future パターンなどを用いて同等の処理が可能である。この実験では簡単のため、重複排除のみを行う。

計測時の環境には以下を用いた：

- Intel Core i7-3770 3.40GHz 4Core 6MB L3 Cache
- PC3-12800 16GB DDR3 SDRAM 1600Mhz
- CentOS 6.8 Linux 2.6.32-431.17.1.el6.x86_64
- OpenJDK 64bit version 1.7.0_121
- Java(TM) SE Runtime Environment (build 1.8.0_40-b25)
- Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)

Memcached による重複排除の実装には, クライアントライブラリとして Spymemcached [70] を用いた. Spymemcached は Memcached の条件付き API を利用している. この API では, キーが存在しないときにだけキーと値を書き込むことができるので, 重複排除において put する前に get しなくてもよい. 使用メモリ量は, データ格納前と格納後の ps コマンドの Resident Set Size (RSS) 値の差分から求めた. サーバ, クライアント共に同一の計算機内で実行した.

Redis による重複排除の実装には, Java によるクライアントライブラリである Jedis [42] を用いた. Redis では Pipeline と呼ばれるコマンドを連続して処理する API が提供されており, 複数のキーに関する処理をまとめてサーバに送信して処理できる. また, Redis にも Memcached と同様にキーが存在しないときにだけキーと値を書き込む機能がある. Redis においても Memcached と同様にこの機能を用いて重複排除を効率良く実装できる. 使用メモリ量は, データ格納前と格納後の ps コマンドの Resident Set Size (RSS) 値の差分から求めた. サーバ, クライアント共に同一の計算機内で実行した.

Infinispan による重複排除の実装では, Infinispan をキャッシュ置き換えなしの Embedded Local キャッシュとして用いた. プログラムは 6.2 節のプログラムと同等のものを用いた. 同じ Java Virtual Machine 内で Infinispan を直接呼び出すため, ネットワークのオーバーヘッドがなく, サーバクライアント型の場合と比較して高いスループットを得られる. 使用メモリ量は, キャッシュにデータを格納した際の Java ヒープメモリの使用量の増加から求めた.

このように, これらのキーバリューストアは, ハッシュテーブルを用いているので, メモリ効率が悪い. 本研究で実装した OLT1, および, OLT2 は, 高いメモリ効率を実現していることがわかる. また, 使用可能メモリが増えるとスループットが高くなる.

表 A.1: 代表的なインメモリデータストアの使用メモリ量とスループット

インメモリデータストア	NHTSA		kakaku.com		reuters.com	
	メモリ	件/秒	メモリ	件/秒	メモリ	件/秒
memcached-1.4.4	349 MB	32 万	914 MB	36 万	345 MB	35 万
redis-3.2.6	756 MB	69 万	2136 MB	74 万	638 MB	74 万
infinispan-8.2.5-Final	861 MB	242 万	2334 MB	367 万	740 MB	393 万
OLT1	89 MB	52 万	269 MB	98 万	101 MB	197 万
	134 MB	93 万	404 MB	124 万	152 MB	273 万
	178 MB	117 万	538 MB	171 万	202 MB	342 万
	223 MB	142 万	673 MB	239 万	253 MB	422 万
OLT2	89 MB	79 万	269 MB	339 万	101 MB	244 万
	134 MB	158 万	404 MB	396 万	152 MB	351 万
	178 MB	182 万	538 MB	405 万	202 MB	394 万
	223 MB	201 万	673 MB	401 万	253 MB	408 万

付録B Apache Spark

参考のため、分析処理でよく利用される Apache Spark (spark-2.0.2-hadoop-2.7) [88] [87] のメモリ効率を測定した。測定方法は付録 A と同じである。この実験では、Apache Spark による重複排除を Scala を用いて以下のように実装した：

```
rdd.distinct.zipWithUniqueId.persist(MEMORY_ONLY_SER)
```

ここで、`rdd` は入力文字列を指す初期 RDD である。この変換では、`distinct` と `zipWithUniqueId` によって作成された RDD を `persist(MEMORY_ONLY_SER)` によってバイト配列にシリアル化された形態でメモリに維持している。この変換を単体で起動した spark-shell に実行させ、Java ヒープメモリの使用量の増加から使用メモリ量を求めた。また、データセットにこの変換を適用したときにかかった時間に入力文字列数で割ったみなし平均スループットを求めた。この実行に割り当てられたスレッド数は 4 である。

表 B.1 に結果を示す。この表の各行は、最終的な RDD の使用メモリ量（メモリ）とスループット（件/秒）を示している。この RDD を作成するために中間的な RDD がされているが、そのサイズは含まれていない。この実験では、Apache Spark はマルチスレッドで動作したため、高いスループットになった。Apache Spark の使用メモリ量は OLT1 や OLT2 の使用メモリ量の約 3.1 倍 (kakaku.com, reuters.com) から 4.7 倍 (NHTSA) になる。また、Apache Spark には、メモリ使用量を制約する機能がない。これに対して、本研究で提案した OLT1、および、OLT2 は、限られたメモリ資源の中で動作する。

表 B.1: Apache Spark の使用メモリ量とスループット

	NHTSA		kakaku.com		reuters.com	
インメモリデータストア	メモリ	件/秒	メモリ	件/秒	メモリ	件/秒
spark-2.0.2-hadoop-2.7	368 MB	589 万	772 MB	214 万	272 MB	249 万

公表論文のリスト

本論文で述べられている研究内容は以下の論文で発表済みである。

査読付き論文誌論文

1. 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. 情報処理学会論文誌データベース (TOD), Vol. 4, No. 4, pp. 1–10, 2011.
2. 小柳 光生, 吉田 一星, 海野 裕也, 新城 靖. 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. 情報処理学会論文誌コンピューティングシステム (ACS), Vol. 5, No. 2, pp. 1–9, 2012.

査読付き国際会議論文

1. Teruo Koyanagi, and Yasushi Shinjo, A Fast and Compact Hybrid Memory Resident Datastore for Text Analytics with Autonomic Memory Allocation. In Proceedings of IEEE International Conference on Information and Computational Systems 2014 (ICICS 2014), pp. 1–7, 2014. (Best Research Paper Award).

その他

1. Rudy Raymond, Teruo Koyanagi, and Takayuki Osogami, An Approximate Counting for Big Textual Data Streams. In Proceedings of 21st European Conference on Artificial Intelligence (ECAI 2014), pp. 1085–1086, 2014.
2. 小柳 光生, 吉田 一星, 空間効率の高いインメモリー辞書のオンライン構築. IBM プロフェッショナル論文 ProVISION, No. 72, pp. 89–96, 2012.

その他の査読付き論文誌論文

1. 鹿島 久嗣, 坂本 比呂志, 小柳 光生, 木構造データに対するカーネル関数の設計と解析. 人工知能学会論文誌, Vol. 21, No. 1, pp. 113–121, 2006.

その他の査読付き国際会議論文

1. Teruo Koyanagi, Kohichi Ono, and Masahiro Hori, Demonstrational interface for XSLT stylesheet generation. In Proceedings of Extreme Markup Language 2000, pp. 197–211, 2000.
2. Kohichi Ono, Teruo Koyanagi, Mari Abe, and Masahiro Hori, XSLT stylesheet generation by example with WYSIWYG editing. Proceedings of the International Symposium on Applications and the Internet (SAINT 2002), pp. 150-161, 2002.
3. Hisashi Kashima and Teruo Koyanagi, Kernels for Semi-Structured Data. In Proceedings of 19th International Conference on Machine Learning (ICML 2002), pp. 291–298, 2002.
4. Teruo Koyanagi, Yoshiaki Kobayashi, Sachiko Miyagi, and Gaku Yamamoto, Agent Server for a Location-Aware Personalized Notification Service. In Proceedings of 1st International Workshop of Massively Multi-Agent Systems (MMAS), pp. 224-238, 2004.
5. Yosuke Ozawa, Teruo Koyanagi, Mari Abe, and Liangzhao Zeng. A Hybrid Event Processing Architecture based on the Model-driven Approach for High Performance Monitoring. WS-Testing Workshop on SERVICES 2007, pp. 324–331, 2007.
6. Teruo Koyanagi, Mari Abe, Gaku Yamamoto, and Jun-Jang Jeng. Dynamic Policy Management on Business Performance Management Architecture, In Proceedings of 3rd International Conference (ICSOC 2005), pp. 539-544, 2005.
7. Mari Abe, Jun-Jang Jeng, and Teruo Koyanagi, Authoring Tool for Business Performance Monitoring and Control. In Proceedings of IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2007), pp. 107–113, 2007.
8. Mari Abe, Teruo Koyanagi, Jun-Jang Jeng, and L. An, An Environment of Modeling Business Centric Monitoring and Control Applications. In Proceedings of IEEE International Conference on e-Business Engineering (ICEBE 2006), pp. 511–515, 2006.