修士論文

メモリ上の転置索引による高速全文検索システムに関する研究

A Study on High Throughput Query Processing
using Inverted Index on Main Memory

渡辺 健太郎

東京大学大学院 情報理工学系研究科 電子情報学専攻

指導教員 安達 淳

2011 年 2 月 9 日提出

概要

メモリサイズが増大し低価格化が進んだことから，メモリ上でのデータベースの運用は珍しいものではなくなった．そのようなシステムではかつてのディスク I/O ではなく，プロセッサやメインメモリのリソースがボトルネックとなる．本研究では，転置索引を用いた全文検索システムをメモリ上で運用することを前提に，圧縮データ構造の処理効率を改善することを検討する．多くの圧縮スキームにおいて重要な要素技術である Prefix Sum 処理の高速化手法ならびに，近年注目されている PForDelta 圧縮スキームの細粒度処理を可能にした改良版データ構造とその処理手法を提案する．

目次

図目次

表目次

第 1 章
序論

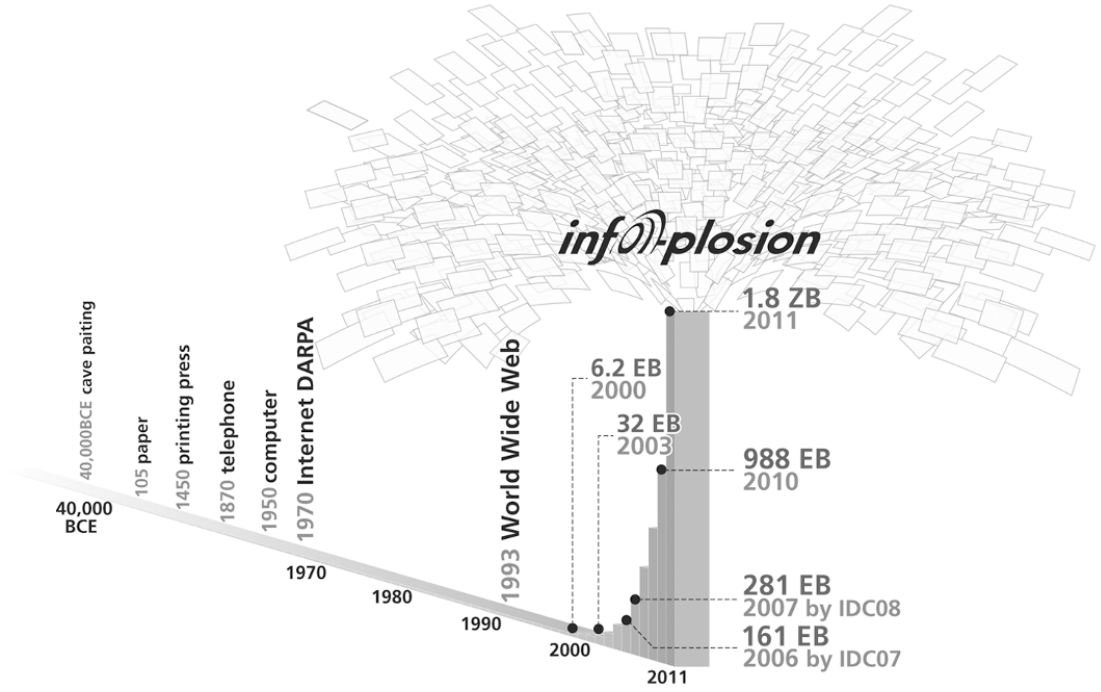

図 1.1: 情報爆発時代 (M. Kitsuregawa and T. Nishida, 2010)．2004 年時点で人類によって生み出された情報量はエクサバイト (= 10^6 テラバイト) 単位で勘定されていた．2010 年の 1 年間で生み出された情報量は 1 ゼッタバイト (= 10^9 テラバイト) を越えると言われる．

1.1 研究の背景

web 上の情報をはじめ，人類が生み出す情報量が近年，爆発的に増加してきていることは既に多くの場所で指摘されており，衆目の一致するところである (図 1.1)[17]．大規模な商用の検索エンジンではテラバイト以上のオーダーの非常に多くのデータを対象として，1 秒間に数万以上の非常に多数のユーザーからの問い合わせをリアルタイムに処理できなければならない [4]．同時にその結果の質に対する要求の水準もまたあがってきており，スパム的な文書の除去のみならず個人の嗜好にあわせた検索結果というのは単にユーザの満足度の向上にとどまらず，有効な広告戦略という観点でも重要なものになっている．

このような高いスループットと速いレスポンスを実現するために検索エンジンでは多数のサーバからなる大規模なクラスタを用いて問い合わせの処理を行っている．このようなクラスタではうまく多数のサーバに負荷を分散させることで，要求される性能を実現しているが，一方で，全体の性能を向上させるためには個々のマシンの処理性能を向上させることもまた重要である．

また，近年では，メモリサイズが増大し低価格化が進んだことから，索引データをメインメモリ上で運用することも珍しくなくなった．そのようなシステムではディスク I/O は根本的に克服され，新たな争点として空間効率というよりもむしろ問い合わせ処理の効率という面で，データの圧縮技術が注目されている．

このような背景から，本研究では web 上の検索エンジンに代表されるような，転置索引を用いた全文検索システムをメモリ上で運用することを前提としたときに，その圧縮データ構造を処理する技術を改善することを検討する．具体的には，既存の整数列データの圧縮スキームに関する処理を現代のプロセッサの命令レベル並列処理技術で効率化する *In-register Prefix Sum* を提案する．また，近年注目されている PForDelta という圧縮スキームをベースにより細粒度の処理を可能にした改良版である *Fine-grained PForDelta* を提案する．

1.2 研究の目的

現実に運用される転置索引においては，文書情報を記憶するデータ構造としてソートされた整数列構造を用いることが多い．このようなソート済み整数列データに関しては従来から様々な圧縮手法が存在した．本研究では，転置索引を用いた全文検索システムをメインメモリ上で運用することを前提に，ソート済み整数列データの圧縮構造の処理効率を改善することを目指す．

転置リストの圧縮に用いられる多くの手法において，ソート済みというデータの構造を活かし，空間効率を向上させるため整数列データを隣り合う 2 要素同士の差分として記憶する Δ-*encoding* と呼ばれる手法が重要な要素技術となっている．しかしながら，Δ-*encoding* された整数列データは要素に対する細粒度のアクセスができず，問い合わせ処理時においてオーバーヘッドが存在した．

そこでまず，Δ-*encoding* された整数列データを効率的に元の整数列データに復元する手法を提案する．このとき，本研究で注目したのは，現代のプロセッサにおいて広く実装されているベクトル演算機能である一連の SIMD 命令である．SIMD 命令を用いる利点として，処理効率の向上とともにプロセッサの実装の他に追加でコストがかかるということがない点がある．元来，SIMD 命令の典型的な適用例は浮動小数点演算アプリケーションであったが，近年整数アプリケーションにおいても SIMD 命令を用いることでパフォーマンスを改善できる例が報告されている．そこで，本研究では On-chip の SIMD 命令を用いて Δ-*encoding* された整数列データを復元する手法を述べる．

前述したように，問い合わせ処理を効率化するという観点から圧縮手法を見たとき，整数列データの個々の要素へのアクセスが可能であるということは非常に重要な要素である．そこで，空間効率と圧縮・伸張処理の高速性から近年注目されている整数列データの圧縮手法である *PForDelta* をベースに問い合わせ処理時に細粒度のアクセスを可能にするデータ構造とそれをもちいる処理手法を提案する．

1.3 本論文の構成

本論文の構成は以下のようになっている．

まず，２章で転置索引をもちいる情報検索システムとパフォーマンスを要求されるデータベースにおける圧縮手法，および現代のプロセッサで実装されているベクトル演算装置である SIMD 命令について関連研究として紹介する．

３章で本研究の共通設定と，メモリ上で転置索引を用いる全文検索システムを運用する上での争点，そして本研究で提案する手法の概要について述べる．

４章で SIMD 命令をもちいた Δ-*encoding* された整数列データの復元手法 *In-register Prefix Sum* の詳細について述べる．

５章で PForDelta 圧縮スキームをベースに細粒度の処理を可能にした改良版である Fine-grained PForDelta の詳細について述べる．

６章で本研究のまとめと今後の展望について述べる．

第 2 章
関連研究

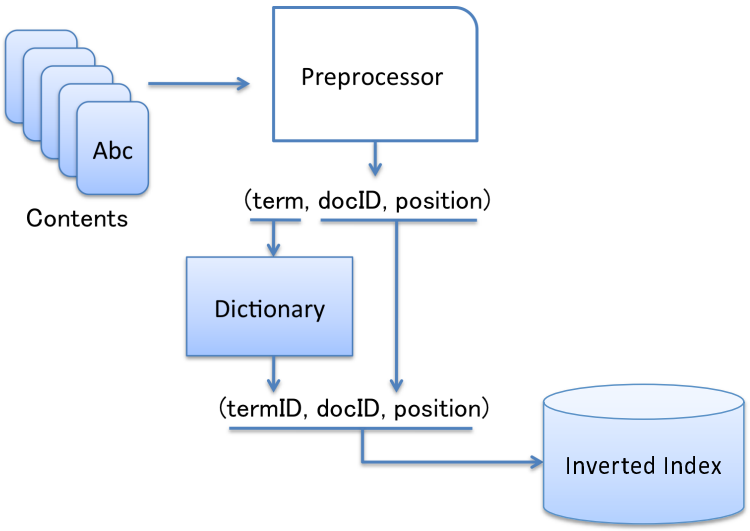

図 2.1: 転置索引生成の概要．索引付ける各文書には文書 ID(*docID*) が割り当てられ，プリプロセッサ (*preprocessor*) で前処理が行われる．出現単語のリストである辞書 (*dictionary*) を構築し，各単語に対して単語 ID(termID) を割り当てる．そして各単語 ID に対し，文書 ID のリストである転置リスト (*inverted list*) が保持される．

2.1 情報検索システム

“情報検索” とは一般に，大規模な未整理の情報の集積から，要求に合致するものを発見することである [18].

情報検索システムの代表的なものの一つとして全文検索システムがある．全文検索システムとは，文書の集合の中から，特定の文字列を含んだ文書自体，およびその文書における文字列の位置を提示するものである．

全文検索システムが対象とする検索問い合わせはおもに *Boolean retrieval query* と，*Phrase query* がある．*Boolean retrieval query* とは複数の単語をクエリ*とし，AND，OR，NOT 等の論理演算オペレータを用いて単語ごとに含むかどうかを指定した問い合わせである．*Phrase query* は複数単語からなるクエリに対して，単語の出現位置がクエリ通りとなっているような文書を発見する問い合わせである．

2.1.1 転置索引

全文検索システムの索引データはこれまでおもにディスク上に保持するものとして研究がなされてきた．問い合わせ処理に高速に対応する索引構造として転置

*クエリ (query) という表記は検索システムに対する問い合わせの際の入力文字列のことを指すものとする．

索引 (*inverted index*) が有力なものであるとされている [18][39]†．転置索引を作成する処理全体の概要を図 2.1 に示す．索引付ける各文書には文書 ID(*docID*) が割り当てられ，プリプロセッサ (*preprocessor*) で前処理が行われる．プリプロセッサでは文書の正規化が行われ，スペースやコンマピリオド，句読点の除去がなされ，大文字小文字が統一される．最終的にその文書で出現した単語のリストが生成される．すべての文書を対象とした出現単語のリストを辞書 (*dictionary*) という．辞書では各単語に対し，単語 ID(termID) が割り当てられる．転置索引自体が保持するのはそれら ID のみである．各単語 ID に対し，文書 ID のリストである転置リスト (*inverted list*) が保持され，転置リストによってその単語が出現する文書を特定することが出来る．

問い合わせ処理の際も同様にクエリの文字列を前処理し，単語に分けて単語 ID を得，索引を見るという流れになる．

このような文書単位 (*document-grained*) の索引の場合，*Boolean retrieval query* に対する問い合わせ処理は可能であるが，*Phrase query* への対応にはさらに別に文書中で単語の出現する位置 (*position*) を保持しておく仕組みが必要となる．そのためには各転置リストの文書 ID の情報の後に位置情報を保持するか，位置情報のみを別に保持するのが一般的である．

2.1.2 転置索引を用いた問い合わせ処理

転置索引を用いた問い合わせ処理は概ね次のようになる．クエリの単語に対応する転置リストがディスク上あるいはメモリ上の索引から取得され，必要であれば圧縮データを展開する．そしてリスト同士の intersection あるいはその他の Boolean filter 処理を行うことで，クエリ中の単語の条件を満たす文書 ID を決定する．文書 ID に加えて索引に保持しておく付加的な情報 (単語の出現頻度等) を用いて候補の文書に関するスコアを計算し，top-k 件の文書を出力する．

そのため，転置索引を用いた問い合わせ処理において主な処理は次の 3 つになる．

- 圧縮転置リストの展開
- リスト同士の intersection
- top-k スコア計算

†なお，近年では日本語のように形態素解析が必要な自然言語による文書や DNA 配列のような文字列データベースに対して Phrase query を高速に処理することを目指し，**接尾辞配列**(*Suffix Array*) に関する研究も盛んに行われている [19][43]．2.1.6 節も参照．

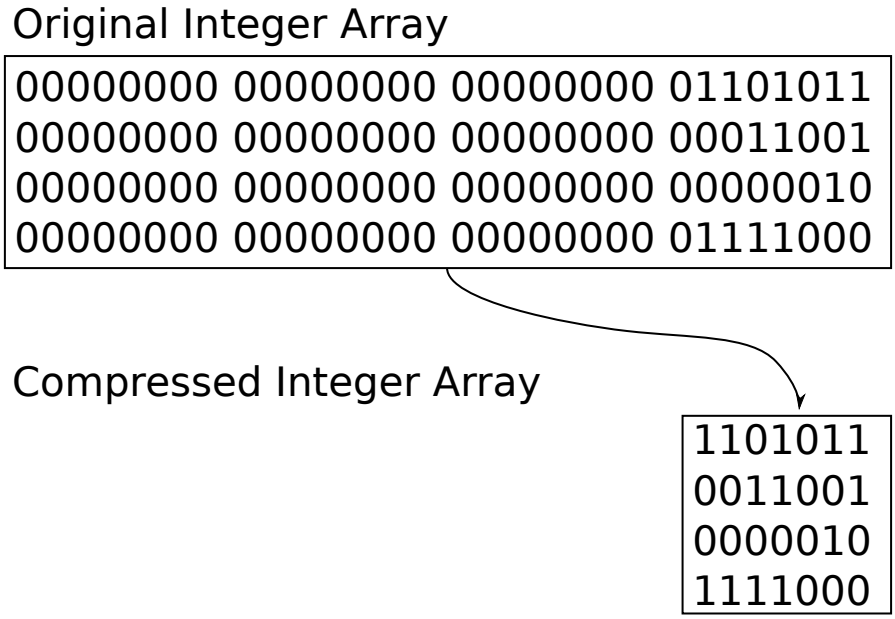

図 2.2: 固定長ビット圧縮．4 つの 32 ビット整数からなる整数列データを，整数列中の最大値 m に対する $\lceil \log_2 m \rceil = 7$ ビットで表現する．このとき，圧縮率は $7/32 \simeq 0.219$ となる．

2.1.3 転置索引の圧縮

一般に情報検索システムは大量のデータを対象とするシステムであるため，索引を保持するストレージの如何に依らず，記憶領域を効率的に利用するということは大変重要な問題である．

転置索引の圧縮の手法は過去に様々提案されてきた [39]．索引構造を圧縮して保持することを考えるとき，圧縮の効率と問い合わせ処理の効率はしばしばトレードオフの関係となる．

索引の圧縮においても各レイヤーでそれぞれの手法があるが，たとえば辞書自体のエントリを減らすという方向で *stop words* をはじめとした不要語の除去を行う手法 [6] がある．

一方，文書 ID を構成する整数列の圧縮で典型的なものでは転置リストの文書 ID を昇順にソートすることで差分のみ記憶する形にするなど，ロスレスな圧縮を用いることが多い [35]．

以下で整数列の圧縮および符号化の手法を紹介する．

固定長ビット圧縮　計算機上で整数列データを扱う際，もっとも基本的な圧縮法のひとつとなるのが固定長ビット圧縮 (*Fixed Bitwidth Compression*) である．図 2.2

に 4 つの整数からなる整数列データを固定長ビット圧縮する場合のもとのデータと圧縮後のデータを示す.

Variable-Byte Coding Variable-Byte Coding は，整数 n を複数個のバイトデータとして保持する．各バイトでは上位 1 ビットを除く下位 7 ビットで整数 n を保持する．上位 1 ビットはどこまでが一つの整数 n のデータを保持するバイトなのかを表す.

Variable-Byte Coding では整数 n を表すのに，$\lfloor \log_{128}(n) \rfloor + 1$ バイト使用する．Variable-Byte Coding は実装が比較的簡単で同種の Golomb や Rice，Gamma，Delta といった各種符号化手法と比べると展開が速いという特徴がある.

Variable-Byte Coding の欠点はどんなに小さい整数であっても必ず 1 バイトのスペースが必要になるという点である.

Simple9 Coding Simple9 Coding は近年提案された手法で，Variable-Byte Coding よりも効率よく圧縮できるのみならず，展開速度においても若干向上したものである [38]．コンセプトは 32 ビットの中に可能な限り整数を詰め込むというものである．そのために Simple9 Coding では 32 ビットを 4 ビットの状態コードと 28 ビットのデータ格納部に分けて使用する．28 ビットの部分のデータの使用法を 4 ビットの状態ビットで表現する.

具体的には [40] では 28 ビットの使い方として 9 通りが考えられていて，それぞれ 28 個の 1 ビットの整数，14 個の 2 ビットの整数，9 個の 3 ビットの整数，7 個の 4 ビットの整数，5 個の 5 ビットの整数， 4 個の 7 ビットの整数，3 個の 9 ビットの整数，2 個の 14 ビットの整数，1 個の 28 ビットの整数である.

展開は先頭 4 ビットの状態ごとにハードコーディングしておくことで高速に行われる.

状態ビットが 4 ビットあることから 28 ビットの使い方は最大 16 通り考えることができ，最大限 16 通りの使い方を実装したのが Simple16[38] である.

PForDelta Coding PForDelta(Patched Frame of Reference of Delta) Coding は本来，より一般的なデータベース，IR システムのために考案された圧縮手法である [40]．ほかに PFor や PDict という手法があり，大きな圧縮スキームのなかの一部分であるが，転置索引の圧縮に用いることが出来る部分について紹介する.

PForDelta Coding はバイト毎あるいはマシンワード毎に値を詰め込むという考

え方ではない．整数を３２の倍数個 (=32n 個) ごとにまとめて圧縮を行うというものである．たとえば 128 個の整数を圧縮する場合，PForDelta Coding の処理の流れはまず，対象の 128 個の整数の 9 割以上よりも大きくなるような整数 2^b の b を決定する．そして 128 個の整数を $128 \times b$ ビットの領域に割り当て，さらに b ビットに入りきれなかった数のためのスペースも割り当てる (入りきらない数を *exception* という)．$128 \times b$ ビット中の *exception* の場所には次の *exception* の $128 \times b$ ビット中での位置を示す数を保持する．実際の *exception* の値は $128 \times b$ ビットの後ろに記憶する．

なぜこのような形でデータを保持するのかというと，現実のデータベースや IR システムでデータの展開をなるべく高速に行うために，現在広く普及しているスーパースカラプロセッサの特性を活かすことを目指しているためである．具体的なポイントとしては圧縮と展開の両段階で if-then-else の条件分岐をなくしている点である．また，いずれの段階においてもデータの依存性がないので完全に loop-pipeline 化をすることができ，out-of-order 実行が可能となって高い IPC(Instruction Per Cycle) 効率を実現できるという点である．

PForDelta Coding の展開は二つのステップで行われる．最初のステップでは $128 \times b$ ビットの値を 128 個の整数列への展開を行う．このとき，ビットをコピーする展開関数を各 b の値に対して個別に用意することで，実行時の分岐を減らし高い効率を実現する．次のステップでは *exception* の場所を linked list をウォークすることで実際の値で置換する．

PForDelta Coding のポイントは 9 割の整数に関してはループ中で分岐がない高速な展開を可能にしている点である．一方，Variable-Byte Coding はすべての値に対して値の終端を調べるために分岐が 1 回以上存在する．また，Simple9 Coding では 1 ワード (=4 バイト) 毎に分岐が存在し，平均して 6 〜 7 個の値毎に分岐が存在することになる．

Rice Coding　Rice Coding は古くからある bit-aligned な圧縮手法で，Rice Coding では整数 n を商と余の二つの部分に分けて保持する．整数列を圧縮する場合まず最初にパラメータ b を選ぶ．b は $b = 0.69 \cdot m$(m は対象の整数列の平均値) にするとよいとされている [39]．整数 n に対して $q = \lfloor n/b \rfloor$，$r = n \mod b$ として q と r を記憶するというものである．

圧縮効率に関して優れている一方，展開の速さに関しては近年の手法に対して劣るという実験結果が報告されている [38].

表 2.1: 代表的な整数列データ圧縮手法の比較．性能に関しては [38] を参考にした．

手法	圧縮率	伸張速度	Reference
固定長ビット圧縮	低	高	Westman ら [33]，Sanders ら [24] 等
Varyable-Byte Coding	低	中	Sanders ら [24]，Zhang ら [38] 等
Simple9	中	高	Anh ら [2]
PForDelta	高	高	Zukowski ら [40]
Rice Coding	高	低	Zobel ら [39]，Zhang ら [38]

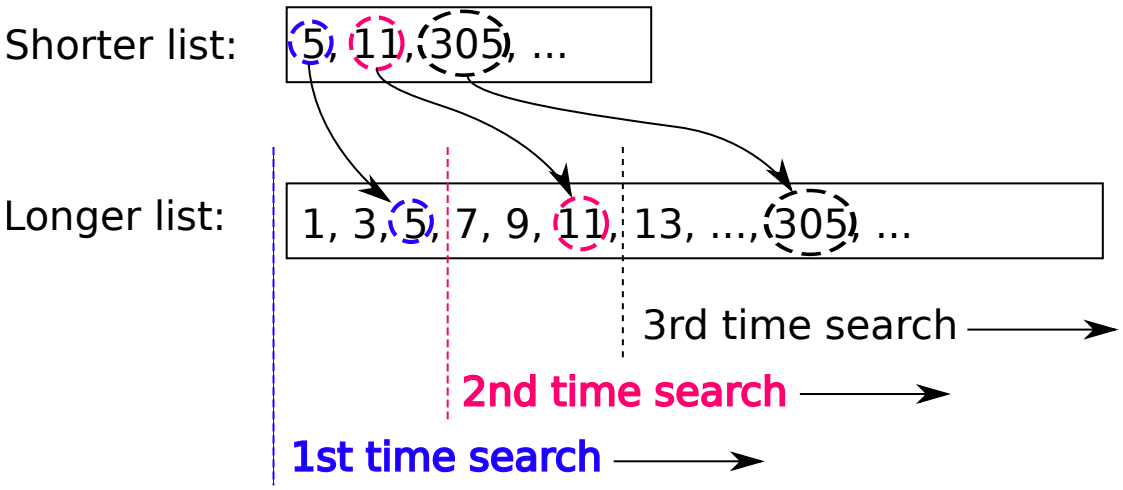

図 2.3: ソートされた二つの整数列のバイナリサーチ型 intersection．ソートされているため，サーチを行うごとに探索対象となる整数列が短くなる．

以上をまとめたものを表 2.1 に示す．

2.1.4 Intersection アルゴリズム

本節では，転置リストの Intersection に用いられるアルゴリズムについて，代表的なものを紹介する．大きく 3 つに分類され，線形探索型と二分探索型，そして階層構造をもちいるものがある．

線形探索型 ふたつのリストを先頭から順にスキャンして一致する要素を取り出すというものである．Δ-*encode* された転置リストを扱う場合もオーバーヘッドが少ないという利点がある．時間効率は $O(m+n)$ であり，二つのリストの長さが近いときにはもっとも高速な手法のひとつとなる [3][24].

バイナリサーチ型 (二分探索型) ふたつのリストのうち短い方の各要素について長い方のリストをバイナリサーチすることで両方のリストで一致した要素を取り出すというものである．時間効率は基本的に $O(m\log(n))$ である．このとき，各リストがソートされていることを仮定すれば，さらに $O(m\log(n/m))$ となる (図 2.3)[15]. 二分探索を用いる手法はリスト中の要素へのランダムアクセスが前提となるため，リストが圧縮して保持されている場合，各値毎に展開が出来ないとリスト全体を展開しなければならず，作業領域，処理時間上のオーバーヘッドが生じることがある．

階層構造を用いた intersection 階層構造を用いる intersection では，転置リストの圧縮においてデータ構造を階層的に保持し，そのデータ構造を陽に利用したサーチ／ intersection を行う．必然的に短いリストより，長いリストにおいてより効果があるということになる．一般に，上位のデータ構造は下部のデータ構造に保持されたリストへのポインタと値の範囲を保持し，下部のデータ構造では値を圧縮して保持することで，圧縮による記憶領域の効率と同時に intersection 操作の実行時にはバイナリサーチを用いることを可能にしている．

階層的なデータ構造を用いる方法は線形探索型でも用いられることがあり，その場合リスト中の要素に何個か先の要素へのポインタを保持しておき，なるべく無駄な比較を減らすというものである．このような intersection アルゴリズムを用いたメインメモリ上の検索システムが Strohman らによって [29] で提案されている．

2.1.5 top-k スコア計算

Web サーチエンジンのような用途ではユーザからのクエリに適した 10 件の文書を最終的に出力することになる．これは転置リスト同士の intersection を取った結果の文書の集合に対して，スコア計算のための関数 (*scoring function*) を適用することで計算される．スコア計算のための関数はこれまでさまざま提案されており，典型的なものでは単語の文書中の，あるいは索引が対象とする全文書中の出現頻度や文書長，文書中の出現位置を用いるものがある．代表的なものとしては TF・IDF や Okapi の BM25[23] がよく知られている．

検索の効率化を図るため，用いるスコア計算のための関数が決まっている場合は転置リスト中の文書 ID の並びを文書 ID によるソートではなく，そのスコア計算に特化して top-k 件の計算を速くする手法もある [39].

本研究では特定のスコア計算のための関数によらない一般的な問い合わせ処理の高速化のための手法を検討する．

2.1.6 メモリ上のテキストデータの検索システム

メモリ上のテキストデータの検索システムの研究においては，近年転置索引を用いる手法の他に，部分文字列の探索処理に対応するため接尾辞配列 (*Suffix Array*) を用いた索引構造の研究が盛んに行われている [19]．接尾辞配列は索引の容量が大きくなってしまう点が欠点であったが，簡潔データ構造を用いて容量を小さくしながら，検索速度を維持する手法が提案されている [43]．接尾辞配列を用いた索引は *self-indexing* な索引付けが可能で，最終的に文書を出力する際に別に文書本体のデータを用意しなくても済むという点は大きな利点である．

一方，転置索引をメモリ上で運用することを前提にした研究もまた近年行われている [30][31]．ここでは問い合わせとして Boolean retrieval query と Phrase query が扱われており，英語の自然言語で書かれた文書を対象とした実験では Phrase query においても接尾辞配列による実装と比較しうる処理速度を実現している．また，これらの研究で扱われている全文検索システムをベースにしたものが，現実に SAP の TREX という分散データベースシステムにおいて利用されている．

2.2 Light-Weight Database Compression

データの圧縮という概念は，今日非常に一般的なものになっている．日常エンドユーザがつかうものだけでも，幾つか例を挙げると例えば，音楽や画像，動画といったマルチメディアデータの扱いにおいてはデータの圧縮は必要欠くべからざるものになっている．それだけでなく，データをアーカイブにして保存する際や，ネットワークを通してやり取りをする場合には代表的なものとしては，UNIX の *gzip* や Microsoft Windows の *zip* といったコマンドをもちいてデータを圧縮することが多い．

こうした圧縮には二つの利点が存在する:

- 空間効率を向上させることで，メインメモリやディスクといったストレージをより有効に活用できるようにすること
- ディスクストレージやネットワークといった，I/O バンド幅の使用量を減らし，結果として I/O がボトルネックとなるアプリケーションの性能を改善すること

もちろん，逆に副作用も存在する．圧縮されたデータを実際に利用する際にはもとのデータの形式に戻さねばならず，その処理上のオーバーヘッドが CPU-bound なアプリケーションにおいては非常に大きな影響を及ぼすこともあり得る．

Westmann らは [33] において次のような例を挙げて説明している．圧縮なしで I/O コストが 1 分程度，CPU コストが 30 秒程度の問い合わせ処理を考える．このとき，I/O 処理と CPU の処理が並行して行われるものとすると，全体の処理時間はおそらく 1 分程度になると考えられる．圧縮ありの場合，I/O のコストを 30 秒以内にすることはおそらく可能であるが，一方で複雑な圧縮手法を用いると CPU コストの方が 1 分以上かかってしまう，というものである．このように何も考えずに圧縮を行った結果として，トータルでの処理時間が圧縮しない場合と比較して長くなってしまうということは，決して非現実的な話ではない．

そこで，Westmann らは *Light-Weight Compression*(*LWC*) という圧縮スキームを提唱した．根本的な着眼点は，処理の軽い圧縮手法，すなわち伸張処理においてオーバーヘッドが少ない手法を使ってデータを圧縮することで，問い合わせの処理のレスポンスタイムを改善しうるということである．さらに，圧縮手法において圧縮率という観点はストレージの大容量化と低価格化がすすめば，次第に薄くなり，むしろレスポンスタイムをどれだけ改善できるかの方に焦点が移るであろうことを指摘している．

2.2.1 LWC の条件

LWC という圧縮スキームが満たすべき要件は大きく次の二つである:

- 圧縮・伸張処理が高速であること
- 圧縮・伸張処理が細粒度に行えること

これらの要件の重要性は Westmann らの [33] 以前にも指摘されており，たとえば関係データベースの圧縮に関する Goldstein らの研究 [12] では，gzip は伸張処理が I/O コストに比して *heavy* であるためデータベースの圧縮には不向きであることが示されている．さらに近年 Zukowski らもまた [40] において，現代のアーキテクチャで各圧縮手法の圧縮率と処理速度の比較を行っているが gzip，bzip2 等は圧縮率が一般に優れているものの，圧縮・伸張処理時間の両面で難があることを示している[‡]．

また，細粒度であることの利点は，伸張処理の高速さに資するだけでなく，問い合わせ処理時の作業領域が小さくて済むという点も大きい．

[‡]gzip は Lempel-Ziv アルゴリズム (LZ77) とハフマン符号を用いており，bzip2 はブロックソート法 (Burrows-Wheeler 変換) と MTF(Move-To-Front) 法，ハフマン符号を用いている．

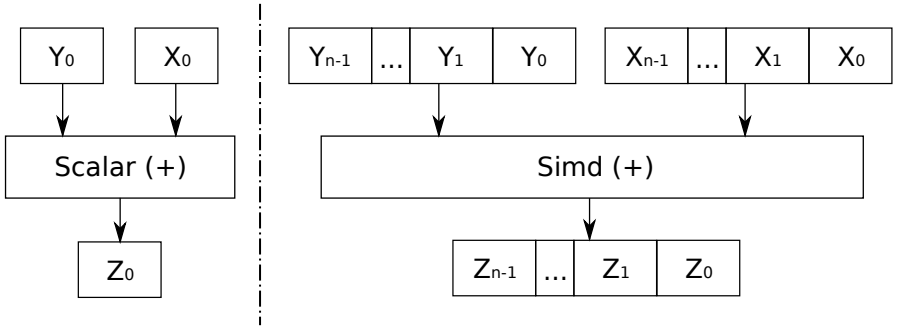

図 2.4: SIMD(Single Instruction Stream-Multiple Data Stream) 計算モデル．ベクトル変数同士の演算と見なすことができ，同時に複数個の同一型のデータに対して同一の演算を施した結果を得ることができる．

2.3 On-chip SIMD 命令

現代のプロセッサは汎用命令，FPU 命令の他に，単一の命令で複数のデータ単位を処理する SIMD 命令を実装している．SIMD(Single Instruction Stream-Multiple Data Stream) という概念は Flynn[10] によって分類された計算モデルのひとつであり，図 2.4 に SIMD の処理のモデルを示す．ここでは，演算はベクトル同士の演算と見なすことができ，同時に複数個の同一型のデータに対してそれぞれ同一の演算を施した結果を得ることができる．

1990 年代後半以降に市場にあらわれたプロセッサ製品はほぼ何らかの形で SIMD 命令をサポートしており，例としては現代の Intel の Xeon や AMD の Phenom といったプロセッサは *Streaming SIMD Extensions*(SSE) 命令セットをサポートしていることが挙げられる．SSE 命令セットでは 128 ビットのレジスタと対応する操作が実装されており，特に 3D グラフィクス，音声認識，画像処理，科学計算といった用途でパフォーマンスを向上させることを念頭に開発された [1]．SIMD 命令の利用による性能の改善は，プロセッサが SIMD 命令を実装していれば他に何も追加コストがかからないという点が大きな特色である．

2.3.1 SSE(Streaming SIMD Extensions) 命令

ここでは，代表的なプロセッサレベルでの SIMD 命令の実装である SSE(Streaming SIMD Extensions) 命令について紹介する．SSE は Intel が開発した CPU の SIMD 拡張命令セットであり，8 本の 128 ビットレジスタを使用する．

図 2.5 に SSE における SIMD レジスタ (XMM レジスタ) の概要を示す．図 2.6 に

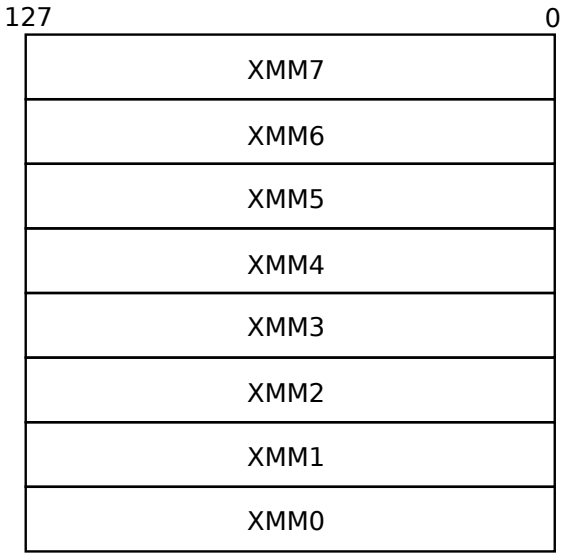

図 2.5: SSE 拡張命令セットにおける SIMD レジスタ (XMM0〜7).

XMM レジスタの具体的な使用例を示す．XMM レジスタでは 16 個の 8 ビットデータ，または 8 つの 16 ビットデータ，あるいは 4 つの 32 ビットデータ，そしてあるいは 2 つの 64 ビットデータがパックされたものとして SIMD 演算が可能である．

また，2011 年には Intel から新たな SIMD 拡張命令セット AVX を実装した Sandy Bridge が市場投入され，256 ビットの SIMD レジスタが利用可能である．

SSE の利用　現実に SSE を用いてソフトウェアを高速化するためにはおもに 4 つの手段が存在する．第一の手段は，インラインアセンブラにより直接プログラム中の SIMD 命令で置き換えたい部分をプログラマが変更するというものである．第二の手段は，アセンブラではなく，C-intrinsic 組み込み関数と呼ばれる C 言語のライブラリ関数を利用するというものである．第三の手段は，C-intrinsic 関数以外のライブラリを利用するというものである．そして，第四の手段はコンパイラによる自動変換を利用するというものである．

可能であれば，第四のコンパイラによる自動変換が最も簡便でコストも少ないと言えるが，現状においては SIMD 命令を使用しない通常のソースコードから，自動で必要な SIMD 命令を検知，使用するということは，十分には実現されていない．そのため，プログラム中で SIMD 命令を利用する際は現時点では二番目の C-intrinsic 関数を用いるということが一般的なものとなっている．

表 2.2 に代表的な SIMD 整数算術命令の命令 (ニーモニック) と対応する C-intrinsic 関数を示す．これらは 8 ビットごと，16 ビットごと，32 ビットごとのそれぞれ加算

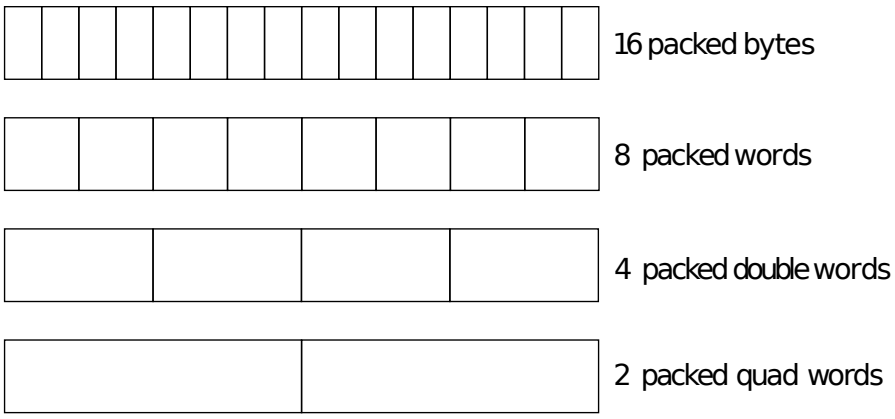

図 2.6: XMM 整数レジスタの使用例．XMM レジスタでは 16 個の 8 ビットデータ，または 8 つの 16 ビットデータ，あるいは 4 つの 32 ビットデータ，そしてあるいは 2 つの 64 ビットデータがパックされたものとして SIMD 演算が可能である．

表 2.2: 代表的な SIMD 整数算術命令の命令 (ニーモニック) および対応する C-intrinsic 関数．8 ビットごと，16 ビットごと，32 ビットごとの加算，減算命令．

分類		ニーモニック	対応する C-intrinsic 関数
算術命令	加算	PADDB	__m128i _mm_add_epi8 (__m128ia, __m128ib)
		PADDW	__m128i _mm_add_epi16 (__m128i a, __m128i b)
		PADDD	__m128i _mm_add_epi32 (__m128i a, __m128i b)
	減算	PADDB	__m128i _mm_add_epi8 (__m128ia, __m128ib)
		PADDW	__m128i _mm_add_epi16 (__m128i a, __m128i b)
		PADDD	__m128i _mm_add_epi32 (__m128i a, __m128i b)

命令，減算命令である．同様に，表 2.3 に代表的な SIMD 論理命令の命令 (ニーモニック) および対応する C-intrinsic 関数を示す．これらは 128 ビットレジスタ同士の AND および OR 演算，そして 16 ビットごとの値，32 ビットごとの値，64 ビットごとの値それぞれに対する左右シフト命令である．また，表 2.4 に代表的な SIMD 整数比較命令の命令 (ニーモニック) および対応する C-intrinsic 関数を示す．これらは 8 ビットごと，16 ビットごと，32 ビットごとの $a = b$ および $a > b$ 比較命令である．また，表 2.5 に代表的な SIMD 転送命令の命令 (ニーモニック) および対応する C-intrinsic 関数を示す．これらはメモリからレジスタへの転送命令およびレジスタ間，レジスタからメモリへの転送命令である．

表 2.3: 代表的な SIMD 論理命令の命令 (ニーモニック) および対応する C-intrinsic 関数．128 ビットレジスタ同士の AND および OR，16 ビットごとの値，32 ビットごとの値，64 ビットごとの値それぞれに対する左右シフト命令．

分類		ニーモニック	対応する C-intrinsic 関数
論理命令	AND	PAND	__m128i _mm_and_si128 (__m128i a, __m128i b)
	OR	POR	__m128i _mm_or_si128(__m128i m1, __m128i m2)
	左シフト	PSLLW	__m128i _mm_slli_pi16(__m128i m, int count)
		PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
		PSLLQ	__m128i _mm_slli_si64(__m128i m, int count)
	右シフト	PSRLW	__m128i _mm_srli_epi16 (__m128i m, int count)
		PSRLD	__m128i _mm_srli_epi32 (__m128i m, int count)
		PSRLQ	__m128i _mm_srli_epi64 (__m128i m, int count)

表 2.4: 代表的な SIMD 整数比較命令の命令 (ニーモニック) および対応する C-intrinsic 関数．8 ビットごと，16 ビットごと，32 ビットごとの $a = b$，$a > b$ 比較命令．

分類		ニーモニック	対応する C-intrinsic 関数
比較命令	等しい	PCMPEQB	__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
		PCMPEQW	__m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
		PCMPEQD	__m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
	$a > b$	PCMPGTB	__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
		PCMPGTW	__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)
		DCMPGTD	__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)

2.3.2 On-chip SIMD 命令の利用例

SIMD 命令の典型的な利用例は音声，動画像アプリケーションにおける浮動小数点数演算であると言える．しかしながら，整数演算の用途においても，近年 SIMD 命令を適用することでパフォーマンスを向上させることを目的とした研究が盛んに行われている．これらについていくつか例を表 2.6 にまとめる．

表 2.5: 代表的な SIMD 転送命令の命令 (ニーモニック) および対応する C-intrinsic 関数．メモリからレジスタへの転送命令およびレジスタ間，レジスタからメモリへの転送命令．

分類		ニーモニック	対応する C-intrinsic 関数
転送命令	メ → レ	MOVDQA	__m128i _mm_load_si128 (__m128i *p)
		MOVDQU	__m128i _mm_loadu_si128 (__m128i *p)
	レ → メ・	MOVDQA	void _mm_store_si128 (__m128i *p, __m128i a)
	レ → レ	MOVDQU	void _mm_storeu_si128 (__m128i *p, __m128i a)

表 2.6: On-chip SIMD 命令を適用することで整数アプリケーションにおけるパフォーマンスを向上させることを目的とした近年の研究の例．

分類	処理	Reference	発表年
ソート	マージソート	Chhugani ら [7]	2008
	マージソート	Satish ら [26]	2010
	ソーティングネットワーク	Furtak ら [11]	2007
サーチ・スキャン	k 分 (*k*-ary) サーチ	Schlegel ら [27]	2009
	固定長ビット圧縮データのスキャン	Willhalm ら [34]	2009
	二分木探索	Kim ら [16]	2010
圧縮・伸張	固定長ビット圧縮データの伸張	Willhalm ら [34]	2009
	固定長ビット圧縮データの圧縮	Schlegel ら [27]	2010

第 3 章
共通設定と提案手法の概要

3.1 本研究の共通設定

本研究では，索引本体が主記憶上に保持されていることを前提とする．あるいは補助的に二次記憶装置を用いる場合でも，キャッシュが有効に機能し，ディスクアクセスのレーテンシを隠蔽できる場合を考えるものとする．ディスクアクセスがボトルネックとなる場合はプロセッサレベルの効率化の影響は非常に限定的なものとなる．

本研究の対象は，転置リストデータの圧縮データ構造と圧縮処理，および圧縮データ構造を用いたクエリ処理である．ここではまず転置リストを表現する整数列データと，それら整数列データに施されるべき処理の具体的な内容を設定する．

3.1.1 整数列

本研究で扱う転置索引を構成する転置リストは n 個の整数の順序付き集合 S として表現されるものとする．この S を整数列 (*Integer Array*) という．n 個の整数要素からなる S は各要素 a_k によって

$$S = \{a_1, a_2, \cdots, a_n\}, \quad a_k \in \mathcal{Z}(1 \leq k \leq n) \tag{3.1}$$

と表すことが出来る．具体的に S は計算機上では整数配列として実装されるものとし，このとき，圧縮等の付加的な処理をしないとすると現代の計算機では $32n$ ないし $64n$ ビットの記憶領域を占める場合が一般的である*．

3.1.2 整数列に対する処理

2.1.2 節で見たように転置索引を用いた問い合わせ処理は主に以下の 3 つである．

- 圧縮転置リストの展開
- リスト同士の intersection
- top-k スコア計算

*なお整数列の表現については，本研究のような整数の配列による表現の他に，$a_1, a_2, \cdots, a_n$ 番目のビットを 1 に，それ以外を 0 にしたような M 個 (M は索引付ける文書の全体数) のビットを並べた**ビットベクトル**(*BitVector*) と呼ばれる表現を用いるものもある [20][42].

本研究ではこのうち，上の二つの処理に焦点を当てる．top-k スコア計算については具体的なシステムが対象とする文書あるいはユーザによって決定しなければならないため，本研究では特定のスコア計算手法を前提とはしないものとする．スコア計算手法を前提とすれば，その手法に特化した形でたとえば，問い合わせに関わらずあらかじめ文書ごとに計算可能なスコアを計算しておき，そのスコアで転置リストをソートしておくことで，top-k 件の最終的な出力を効率的におこなうことが可能であることが示されている [29][39] ことを記しておく．

そのため，本研究が検討を行う整数列に対する最終的な処理は *intersection* である．ここで一つの重要な前提を導入する．それは本研究が対象とする整数列の入力データおよび出力データはいずれも 0 以上の整数であり，なおかつ昇順にソートされているものとする，ということである．このとき，整数列同士の intersection という概念は以下のように定式化される．ある二つの整数列

$$S_a = \{a_1, a_2, \cdots, a_m\}, \quad a_k \in \mathcal{Z}, a_1 \geq 0,\ a_k \geq a_{k-1} (2 \leq k \leq m) \tag{3.2}$$

$$S_b = \{b_1, b_2, \cdots, b_n\}, \quad b_k \in \mathcal{Z}, b_1 \geq 0,\ b_k \geq b_{k-1} (2 \leq k \leq n) \tag{3.3}$$

が存在したとき，intersection とは集合 $S_a \cap S_b$ であり，特に

$$S_a \cap S_b = \{a_k \mid a_k \in S_b, \quad a_1 \geq 0,\ a_k \geq a_{k-1} (2 \leq k \leq m)\} \tag{3.4}$$

$$= \{b_k \mid b_k \in S_a, \quad b_1 \geq 0,\ b_k \geq b_{k-1} (2 \leq k \leq n)\} \tag{3.5}$$

という関係が成り立つ．つまり，S_a と S_b の intersection もまた 0 以上の整数からなる昇順にソートされた整数列となる．

3.2 メモリ上の転置索引を用いた問い合わせ処理の争点

3.1 節のように見ていくと，転置索引の問い合わせ処理のうち出力する文書を決定する部分は整数列データベースへの問い合わせ処理と見なすことができる．つぎに，このような大量の整数列からなるデータベースに対する問い合わせ処理システムをメモリ上で運用する際，争点となることがらについて述べる．

2.2 節で見たように，現在データベースの圧縮の研究においては大きくふたつの観点が重要なものとなっている．それは，圧縮の性能を測る上での二つの指標であり，すなわち，問い合わせに対する処理のレスポンスタイムと圧縮による空間効率の改善である．そのなかで，*Light-Weight Compression*(LWC) と呼ばれる圧縮スキーム群がレスポンスタイムを改善する上で有効なものとなることが示されている [33].

圧縮データ構造を用いることによる問い合わせ処理の高速化の可能性は [13] で詳細に検討されている．そこでは，圧縮されたデータを可能な限り圧縮されたまま保持し，最終的に本当に必要になった段階ではじめてデータを伸張するというアイデアが述べられている．結果として，圧縮による空間効率の改善のインパクトよりもむしろ，問い合わせ処理のレスポンスタイムの改善のインパクトの方が大きいとさえ言われている．

Graefe らの [13] や Westmann らの [33] の研究においては圧縮することで，如何にして最小限のオーバーヘッドで I/O のコストを小さくできるかが最大の争点であった．一方，今日のメモリ上に構築された問い合わせ処理システムにおいては，ディスク I/O はそもそもメモリ上に索引全体を保持してしまうということで克服してしまう．この場合，次に問題となるのは，規模によっては索引全体が載るようになったとはいえ，メモリのサイズがクリティカルな制約となる，ということである．そこで，またしても LWC によるデータの圧縮が大きな争点として浮上してくる．今度はディスク I/O の削減ではなく，問い合わせ処理時のオーバーヘッドを抑えつつ，空間効率を上げてメモリを最大限有効に活用するという，もともとのデータ圧縮の意味合いが大きい．とはいえ LWC であるというのは，オーバーヘッドの大きい *heavy* な圧縮スキームは問い合わせ処理のレスポンスタイム／スループットが悪くなり，そもそもメモリ上に索引を保持する意味が薄れてしまうためである．

こうして，メモリ上の転置索引を用いた問い合わせ処理の場合もまた，転置リストの圧縮，すなわち整数列データの圧縮自体の処理と圧縮されたデータを用い

た問い合わせ処理が重要な争点となることがわかる．このときさらに，メモリサイズの観点から圧縮が争点となるだけでなく，近年では，メモリとレジスタ (プロセッサ) 間の転送幅の観点からも圧縮は重要な技術とみなされている [16][26]．というのもメモリのバンド幅はプロセッサの処理能力の向上に対して，性能向上の速度が遅く，今後プロセッサのコア数がますます増加していくと，コアごとのバンド幅は横ばいか，場合によっては下降に転じる可能性も示唆されている [22] ためであり，かつてのディスク I/O と類似の状況がメモリ-プロセッサ間で再現されている．

以上のような状況の下，圧縮伸張の処理時間のオーバーヘッドと圧縮率がトレードオフの関係を成し，問い合わせ処理は究極的に CPU-intensive なものとなる．

こうしたなかで，研究の方向性は大きく二つ存在する．一つは，圧縮スキーム自体は既存のものを採用し，その処理を高速化するというものである．そのような文脈での研究がたとえば，Willhalm らの [34] や Schlegel らの [28] である．マルチコアプロセッサによる同時マルチスレッディング (*Simultaneous Multi-Threading*) を活用する研究はそれまでも盛んにおこなわれてきた一方で，これらの研究は 2.3 節で見た On-chip の SIMD 命令が十分に活用されていなかったことに着目している．具体的には，[34] では固定長ビット圧縮されたデータの伸張処理や圧縮したままでの探索処理を高速化し，[28] では固定長ビット圧縮処理自体を高速化している．On-chip の SIMD 命令を用いることで，追加コストが一切かからず純粋に同じ処理を高速化できるだけでなく，前述したように今後コアごとのメモリとの転送バンド幅がなかなか伸びないとしても圧縮データの伸張処理のコストを償却できれば，より圧縮率の高い手法にシフトしていくことでさらなる性能向上の可能性が開けてくる．

もう一つの方向性は，処理速度と圧縮率のトレードオフでよりよい均衡点を見出すというものである．すなわち，LWC の範疇で処理のオーバーヘッドを抑えつつ，圧縮率もより高い点を目指して圧縮スキーム自体を改良するというものである．したがって，この方向ではデータ構造の圧縮法のみならず伸張処理の効率もまた陽に意識してスキームを策定しなければならない．このような研究にあたるのが，Zukowski らの [40] である．[40] ではあくまで固定長ビット圧縮をベースとした上で工夫を加えることで圧縮率を高めつつ，伸張処理のオーバーヘッドを最小限に抑えている．

本研究はこれらの方向性に沿って，過去の研究をベースに，新たな手法を提案する．

3.3 提案手法の概要

本研究では，前節 (3.2 節) で述べたメモリ上のデータベースへの問い合わせ処理システムにおける争点と研究の方向性に沿って新たな手法を提案する．

3.3.1 On-chip SIMD 命令の効果的活用

まず，第一の提案手法は On-chip SIMD 命令を有効に活用し，既存の圧縮スキームの処理を高速化することを目指したものであり，4 章で詳しく議論する．

LWC において重要な圧縮手法である *Null-suppression*(または固定長ビット圧縮) は転置リストを表現する整数列の圧縮においてもよく用いられる要素技術の一つであるが (2.1.3 節参照)，ソートされた整数列の場合，整数列を隣り合う 2 要素同士の差分に変換した形で圧縮を行うことで，圧縮率を大きく向上させることができる．このような操作のことを Δ-*encode* と呼ぶ．

本研究ではこの Δ-*encode* された整数列をもとの整数列に戻すための *Prefix Sum* という処理を On-chip SIMD 命令を用いて高速化する手法，*In-register Prefix Sum* を提案する．

3.3.2 In-Memory Inverted Index のための整数列圧縮法

第二の提案手法は，前節 (3.2 節) で述べたメモリ上のデータベースへの問い合わせ処理システムにおける研究の方向性の二番目に当たるもので，圧縮スキーム自体の改良を目指したものであり，5 章で詳しく議論する．

近年提案され空間効率と圧縮・伸張処理効率の両面で優れていることから，転置リストの圧縮手法として注目を集めている *PForDelta* をベースに，とくに転置リストとしての問い合わせ処理の面に焦点を当て，圧縮データをより細粒度で処理することを可能にするデータ構造と，その構造を利用する処理手法を総称した，*Fine-grained PForDelta* を提案する．

第 4 章

In-register Prefix Sum: On-chip SIMD命令の効果的活用

本章では，CPU に実装された標準的な On-chip SIMD 命令をもちいた，Δ-*encode* されたデータの伸張手法について述べる．

まず，4.1 節で LWC において重要な要素技術である Δ-*encode* の定義を与えるとともに，Δ-*encode* された整数列をもとの整数列に復元する処理である *Prefix Sum* 処理の定義を与える．

つぎに，4.2 節で本研究が提案する *In-register Prefix Sum* における並列計算のモデルを説明する．

4.3 節では，過去に存在した，On-chip 実装の命令を用いるのではなく，マルチプロセッサにおいて Prefix Sum を SIMD 処理する手法と *In-register Prefix Sum* との比較を行い，相違点を述べる．

4.4 節では，提案手法を実装する上で必要な On-chip SIMD 命令，および注意点となるキャッシュの効率的な利用法について議論する．

4.5 節で，提案手法を実際に実装して行った実験の概要と，その結果，および考察について述べる．

4.6 節で，本章の結論を述べる．

4.1 Δ-encode と Prefix Sum 処理

Δ-encode Δ-*encode* とは，整数列から整数列への写像であり，具体的にはある整数列 S

$$S = \{a_1, a_2, \cdots, a_n\}, \quad a_k \in \mathcal{Z}(1 \leq k \leq n)$$

から

$$S' = \{a_1, a_2 - a_1, a_3 - a_2, \cdots, a_n - a_{n-1}\} \tag{4.1}$$

への写像で，このとき各要素は

$$a_1 \rightarrow a_1 \tag{4.2}$$

$$a_k \rightarrow a_k - a_{k-1} \ (2 \leq k \leq n) \tag{4.3}$$

のように変換される．

整数列データが昇順にソートされ，なおかつ重複した値が存在しないとすると，Δ-encode を適用した際，各要素の値は 1 以上の整数となり，固定長ビット圧縮の

圧縮率を高めることができる．

Prefix Sum 処理　Δ-*encode* された整数列は Prefix Sum 処理を施すことで元の整数列の内容に戻る．一般に Prefix Sum 処理とは以下のように，ある演算子 $\oplus$ と n 個の要素からなる整数列 A から整数列 A' を得る処理のことである [5].

$$A = \{a_0, a_1, \cdots, a_{n-1}\} \tag{4.4}$$

$$A' = \{a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})\} \tag{4.5}$$

Δ-*encode* されたデータの復元には演算子として加算 + を用いる．

4.2 並列計算のモデル

提案手法では，ワード数 n のレジスタに対して，$\log n$ 回のシフトと加算でデータを展開する．最初に n 個のデータをレジスタにロードする．最初の加算では，ロードしたデータを 1 ワード分シフトさせた別のレジスタを用意し，これらの間で加算を行う．２回目の加算では，最初の加算の結果を２ワード分シフトさせ同様に加算を行う．

このようにして，１回の iteration 内の k 回目の加算では $k-1$ 回目の加算の結果を 2^{k-1} ワードシフトしたものとの加算を行うことになる．

各 iteration 間では前の iteration のレジスタの最上位ワードの内容を次の iteration に渡していく．対象となる整数列の要素数がレジスタのワード数の倍数ではない場合，最後に余った要素については別途処理を行わなければならない．

図 4.1 にレジスタのサイズが 128 ビットの場合の，レジスタ 1 本のデータの展開の様子を示す．

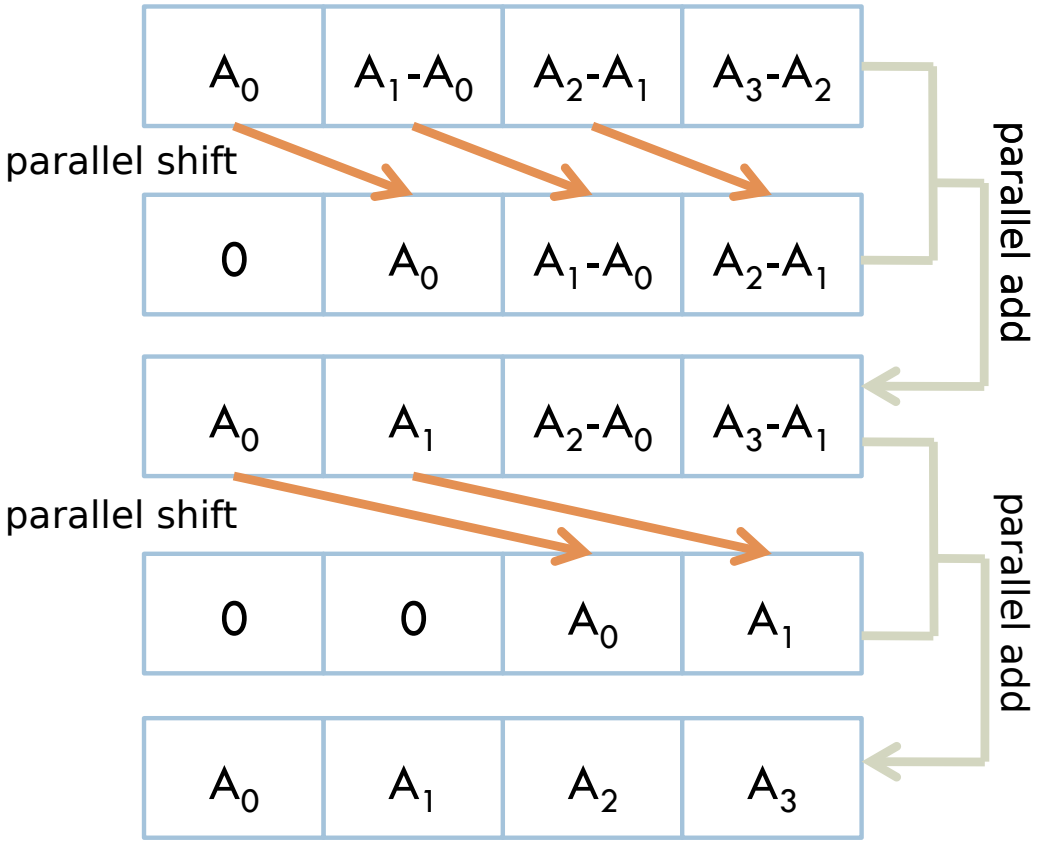

図 4.1: 128 ビットレジスタ・32 ビット整数列の Prefix Sum 処理の様子

4.3 マルチプロセッサによる並列Prefix Sum処理との相違点

過去に非常に多量のプロセッサを相互に接続し大規模データを処理することを念頭に研究された並列プロセッサシステムにおいて 4.2 節で述べた Prefix Sum の並列処理のモデルと類似するモデルによる処理手法が存在した (Hillis ら [14]).

このような並列プロセッサシステムにおいては，Prefix Sum 処理はおもに並列 radix sort の処理の一部という性格が強い [5].

[14] では入力と同じサイズの数のプロセッサを並べて並列処理を行っていたため，iteration が深くなるごとにアイドル状態のプロセッサが増えてしまう．そのため，[5] ではアイドル状態のプロセッサを出さない，reduce と sweep の 2 段階の操作による Prefix Sum 処理を提案している．近年の GPGPU を用いた radix sort の研究ではみな reduce-sweep 型 Prefix Sum が用いられている [26][25][9]．しかしながら，GPGPU における Prefix Sum および Hillis らや Blelloch の Prefix Sum の研究ではプロセッサと共有メモリとの通信が自由に行えることを仮定していたことに注意する．

一方，本研究における On-chip SIMD 命令を用いた Prefix Sum 処理は Hillis らと類似のモデルを用いる．なぜなら，Blelloch の reduce-sweep 型 Prefix Sum では不連続な位置のメモリに頻繁にアクセスが生じ，いわゆる *gather*, *scatter*[*] と呼ばれるメモリアクセス操作が必要となるが，これらの命令は依然として On-chip の SIMD 命令では効果的な実装が存在しないためである．

また，On-chip SIMD 命令を用いた高速化が可能なのはそもそも，一度に複数個のデータに対して同じ処理を施すことができるからであり，特定の状況では一度の演算で幸運なことに余分におつりが来る，というようなものである．On-chip SIMD 命令においてレジスタの一部が計算をしていないからといって，無理にレジスタ幅を目一杯利用しようとして，処理自体が複雑になってしまえば本末転倒となってしまう．あくまで，処理の自然な流れの上で利用できる際に処理の SIMD 化を図ることが肝要である．

[*] gather，scatter とはそれぞれ不連続なアドレスからの読み込み，書き込みのことである．

4.4 実装

本研究では，提案手法のモデル (図 4.1) を SSE2 命令セットを用いて実装した．プログラム中ではインラインアセンブラではなく，C-intrinsic 関数を用いた．加算演算の Prefix Sum 処理を SIMD 命令を用いて処理する際，必要となる SIMD 命令は以下である (2.3 節も参照)．

- 128 ビット単位のレジスタ間，レジスタ-メモリ間の転送命令
- 128 ビットレジスタにおける 32 ビット整数要素毎の加算命令
- 128 ビットレジスタにおけるバイト単位シフト命令

4.4.1 キャッシュの効率

SIMD 命令の利用による高速化は処理における命令レベルの並列性 (ILP) を向上させるものとして捉えられる [16]．メモリアクセス時のキャッシュミスによる遅延は数百 CPU サイクルにのぼるため，処理の対象となるデータを含めてアプリケーションが使用する領域の大きさがラストレベルキャッシュ(LLC) のサイズと同じオーダに達する場合，メモリアクセスについて考慮することは有益である．

SSE 命令ではレジスタからメモリへの書き込みにおいてキャッシュラインを経由するかどうかで異なる命令が存在する．たとえば，XMM レジスタからメモリへの書き込みにはアライメントが揃っている場合は通常，MOVDQA 命令が使用されるのに対し，後続の処理でしばらく使用しないテンポラルなデータの書き込みに対しては MOVNTDQ 命令の使用が推奨されている [1]．後者のようなストリーミング・ストア命令を使用することでキャッシュをフラッシュする影響を最小限に抑えることができる．

したがって入力データのサイズによって，メモリの書き戻しに適切な方法を選択することでキャッシュラインの利用を最適化することが可能であると考えられる．

表 4.1: 実験環境の概要

System Parameters		
	Xeon	Core2Duo
Model number	X5492	T8300
Core clock speed	3.4GHz	2.4GHz
Number of cores	4	2
L1 Data Cache	32KB	32KB
L2 Data Cache(Shared)	6MB	3MB
Front-side Bus Speed	1600MHz	800MHz
Memory Size	64GB	4GB
Operating System	CentOS	MacOS X
Compiler	gcc 4.4.0	gcc 4.0.1

4.5 実験と考察

Δ-*encode* された整数列の展開において，前節で説明した In-register Prefix Sum を用いて SIMD 命令を利用した場合と SIMD 命令を利用しない場合でそれぞれ処理時間を計測し，比較を行った．いずれも C 言語で実装し，SIMD 命令の使用には C-intrinsic 組み込み関数を用いた [1].

実験は，Intel の Xeon(X5492) および Core2Duo(T8300) プロセッサを用い，それぞれ Red Hat Linux 上，Mac OSX 上で行い，コンパイルにはいずれも GCC を用いた．Xeon は L1 データキャッシュ32KB，2 コア共通 L2 データキャッシュ6MB という構成であり，一方 Core2Duo は L1 データキャッシュ32KB，2 コア共通 L2 データキャッシュ3MB という構成である．その他，実験環境の概要を表 4.1 にまとめる．

SIMD 命令はそれぞれのプロセッサで，MMX，SSE，SSE2，SSE3 をサポートし，In-register Prefix Sum に必要な命令を実行できる．最適化のオプションはいずれのアルゴリズムも-O3 でコンパイルを行った．オペレーティングシステムのノイズによって外れ値が生じることがあるため，それぞれの実験は 150 回実行し，その結果のプロセッサの消費サイクル数の中央値を用いて比較を行った．

提案手法の整数列の長さに対する大域的な性質を調べるため，$2^7(= 128)$ から $2^{25}(= 33554432)$ 個までの 2 のベキ乗個の整数列を対象に Prefix Sum 処理の性能を示す．図 4.2 に SIMD 命令を利用した場合としない場合の処理時間の比較結果を示す．グラフは横軸が処理した整数列の長さ，縦軸が速度向上比を表しており，値が大きい方がより速度が改善していることを示す．

整数列が短い場合，L1 キャッシュから L2 キャッシュに載る間の範囲では MOVDQA 命令を用いた場合，Xeon，Core2Duo プロセッサの双方で最大で 2.5〜3 倍弱の性能改善が見られる．一方，L1 キャッシュサイズに載る範囲では Core2Duo では改善が見られたものの，Xeon プロセッサでは改善が見られないか，あるいは逆に遅くなるということがわかった．

整数列が長く，キャッシュに載りきらない場合，アプリケーションの途中でキャッシュとメモリのやり取りが発生する．図 4.2 中で LLC に載るラインを示しているが，処理対象のデータが LLC に載らない場合は MOVDQA 命令を用いた In-register Prefix Sum による速度向上は限定的なものとなる．一方，MOVNTDQ 命令を用いた場合はメモリへの書き戻しでキャッシュラインに後続処理で使わないデータが残るという非効率性が除かれ，入力データが LLC に載らない大きさでも安定した速度の向上が見られる．

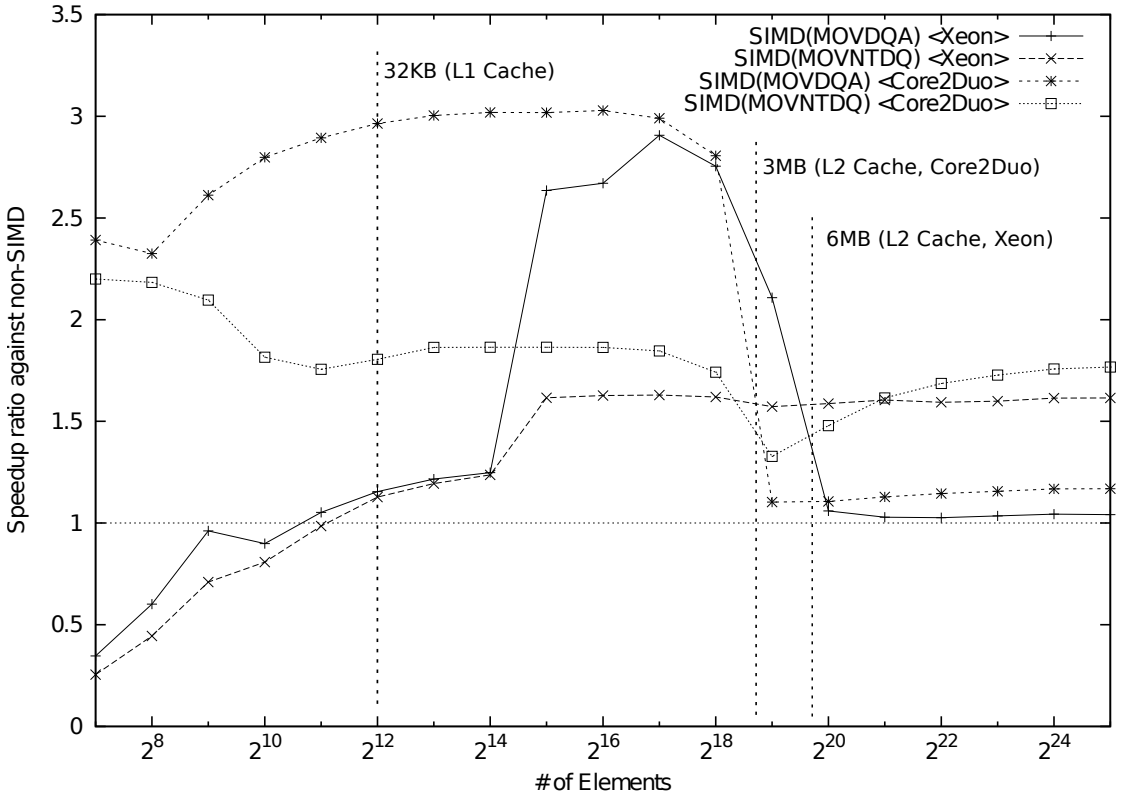

図 4.2: Xeon・Core2Duo プロセッサでレジスタ-メモリ間の転送に MOVDQA・MOVNTDQ 命令を用いた場合の In-register Prefix Sum の速度向上比

つぎに，通常の SIMD を使わない場合と比較して性能が悪化した，Xeon における L1 データキャッシュに載る範囲についてより詳しく実験した結果を図 4.3 に示す．図 4.3 では Prefix Sum 処理において，平均で 1 要素あたり消費するサイクル数を測定したものである．実験は図 4.2 と同様，150 回の試行に対する中央値を用いている．整数列の長さの範囲は 100 個から 4092 個程度で示しているが，見てわかる通り，丁度 2^{10} の倍数の長さのとき，In-register Prefix Sum はいずれの書き込み命令を使った場合もなんらかのオーバーヘッドが生じてしまっていることがわかる．

図 4.2 で L1 データキャッシュに載る範囲において性能が悪化して見える部分はこの箇所を拾っており，2^{10} の倍数個の箇所を除くと，SIMD 命令を使わない場合と比較して，In-register Prefix Sum が有利になると言えるのは整数列の長さが 700 個程度以上になったとき，であるということがこの結果からわかる．

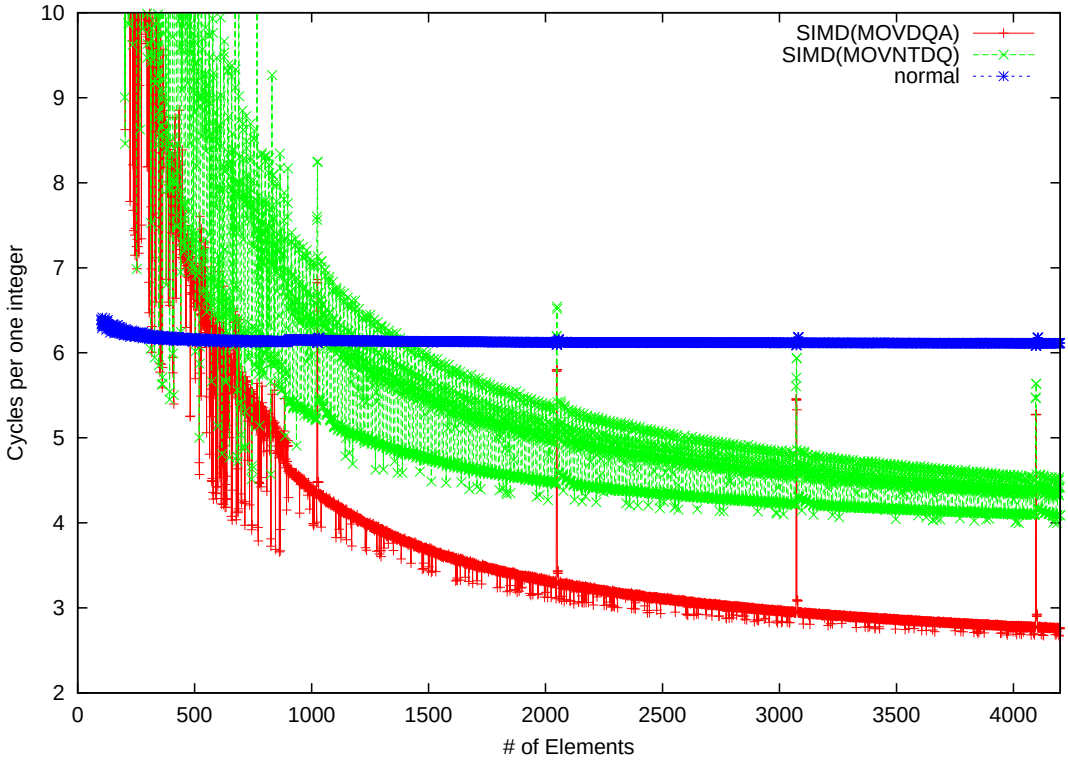

図 4.3: Xeon プロセッサでレジスタ-メモリ間の転送に MOVDQA・MOVNTDQ 命令を用いた場合および SIMD 命令を使わない場合の Prefix Sum 処理における整数 1 要素当たりの消費プロセッササイクル数.

4.6 本章の結論

本章では，On-chip SIMD 命令を用いた，高速 Prefix Sum 処理手法である *In-register Prefix Sum* を提案した．

具体的には，On-chip SIMD 命令を使用した並列計算に向けたモデルを提案し，実際に実装して，SIMD 命令を使わない一般的な場合と比較して性能を測定した．整数列の長さによって，用いる SIMD 命令を切り替えることで，LLC の範囲内で最大で 3 倍弱，LLC の範囲外においても 1.5〜1.8 倍程度の性能の向上を示し，On-chip SIMD 命令を使用することで，一般に Prefix Sum 処理を行うアプリケーションの性能が向上する可能性を示した．

なお，*In-register Prefix Sum* のモデルはレジスタのワード数が増えるほど 1 要素当たりの必要な処理の回数が減るため，今後アーキテクチャが発展し，レジスタのビット幅が増えれば，さらなる性能の向上が見込まれる．

第 5 章
Fine-grained PForDelta: In-Memory Inverted Indexのための整数列圧縮法

本章では，PFor 型データ構造をメインメモリ上の転置索引での利用を念頭にした改良版である Fine-grained PForDelta について述べる．

5.1 PForDelta

本節では，Fine-grained PForDelta のベースとなっている PForDelta データ構造の核となる概念について述べる．

5.1.1 PFor 型データ構造

PForDelta とは，Zukowski らによって [40] で提案されたデータ圧縮スキームの一部で，特に整数列のデータを対象とするものである．対象となる整数列のデータとは，PFor の場合，任意の数 n の整数の順序付き集合 S

$$S = \{a_1, a_2, \cdots, a_n\}, \quad a_k \in \mathcal{Z}(1 \leq k \leq n) \tag{5.1}$$

であり，PForDelta では，任意の数 n のソートされた 0 以上の整数の順序付き集合 S

$$S = \{a_1, a_2, \cdots, a_n\}, \quad a_k \in \mathcal{Z}, a_1 \geq 0, \ a_k \geq a_{k-1}(2 \leq k \leq n) \tag{5.2}$$

である．

この圧縮スキームの特徴は，圧縮データの伸張*を高速に行えるよう設計されたということである．具体的には，固定長のビット圧縮データの伸張からわずかの伸張にかかる時間上のオーバーヘッドだけで，圧縮率も大きく改善することに成功している [38].

ここではまず，PForDelta に用いられている以下の要素技術を順に解説する．

- Prefix-suppression(PS), または Null-suppression
- Frame-Of-Reference(FOR)
- Δ-encode

*本論文では “伸張” という言葉は，圧縮されたデータから元の形式のデータを得る行為のことを指すものとする．

Prefix-suppression(PS), または Null-suppression *Prefix Suppression* とは，データの値の集合において共通する先頭部分 (prefix) を除去することでデータ量を減らすことである [33]．このことは整数データであれば多くの場合，データの先頭から続くゼロを削るということになる．その場合，Null(=0)-suppression とも呼ばれる．この Prefix Suppression を用いた最もシンプルな整数列データの圧縮法のひとつが固定長ビット圧縮である．

Frame-Of-Reference(FOR) *Frame-Of-Reference(FOR)* とは，数値データの圧縮に用いられ，圧縮するデータの全体を単位数ごとに chunk 分割し，それぞれの chunk を一ブロックとして圧縮することを考えたとき，各ブロックのデータの中で最小の値 *min* を保持しておき，そのブロックの各データの値 a_k を

$$a_k \to a_k - min \tag{5.3}$$

とし，計算機上でこれらの値を

$$\lceil \log_2(max - min + 1) \rceil \tag{5.4}$$

ビットで表すというものである．ここに *max* は *min* 同様そのブロックのデータ中の最大の値である．

したがって，FOR は PS を包含しており，固定長ビット圧縮の一種とみなせる．

Δ-encode Δ-*encode* については既に 4.1 節で見たように，整数列から整数列への写像であり，具体的にはある整数列 S

$$S = \{a_1, a_2, \cdots, a_n\}, \quad a_k \in \mathcal{Z}(1 \le k \le n)$$

から

$$S' = \{a_1, a_2 - a_1, a_3 - a_2, \cdots, a_n - a_{n-1}\}$$

への写像で，このとき各要素は

$$a_1 \to a_1$$
$$a_k \to a_k - a_{k-1} \ (2 \le k \le n)$$

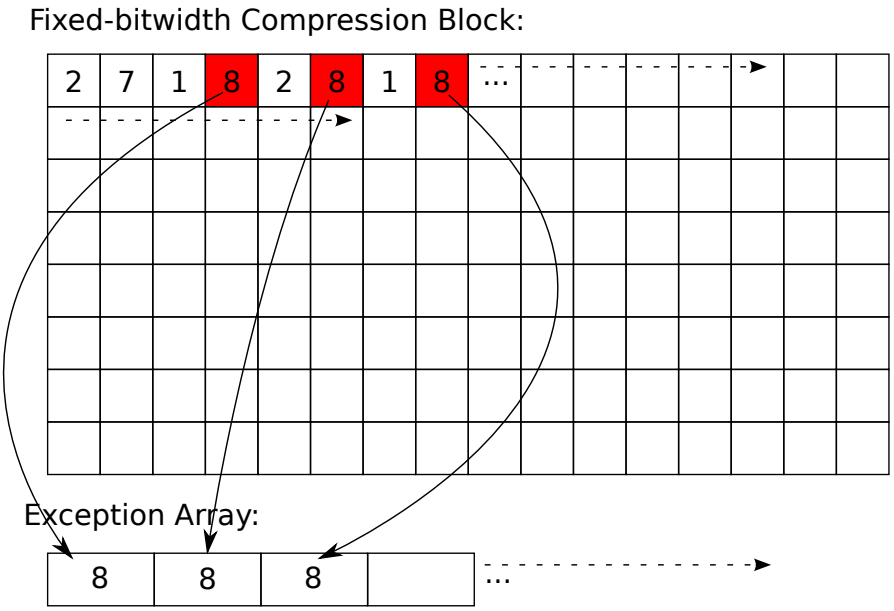

図 5.1: PFor 型データ構造における単位ブロックの圧縮の概念図.

のように変換するものであった.

整数列データが昇順にソートされ，なおかつ重複した値が存在しないとすると，Δ-encode を適用した際，各要素の値は 1 以上の整数となり，FOR やそれに類する固定長ビット圧縮の圧縮率を高めることができる.

PFor データ構造のコンセプト　FOR のような固定長ビット圧縮技術においては圧縮するデータのブロック中の要素の最大値 *max* によって圧縮率が決定される．圧縮するデータが文書 ID のような整数列である場合，文書 ID がクラスタ状の分布をしているとクラスタ間のギャップがブロック中に含まれてしまうことがあり，圧縮率は著しく悪化してしまう．そのため Sanders らは文書 ID を一様分布になるような写像を行うことで圧縮率を最適にする研究を行ったがあまりよい結果は得られていない [24].

一方，Zukowski らの *PFor(Patched Frame-of-Reference)* データ構造のコンセプトは整数列データ中で一定以上大きな値をもつ要素を**例外**(*exception*) としてブロックから追い出し，ブロック中の要素の最大値 *max* を下げることで圧縮率を上げるというものである．例外の要素の値は別に保持しておく．そのため例外の割合は全体の数%から 1 割程度となる.

このような空間効率上の工夫が一つ目のコンセプトであるが，二つ目のコンセプトは，同時に伸張時の処理上のオーバーヘッドを最小限にするような工夫であ

る．具体的には伸張する整数列の各要素についてひとつひとつ例外か否かを判定するのではなく，代わりに 2 段階の伸張ステップを考えるというものである．第一段階では，例外も含めてとにかくそのブロックのデータを誤りを含んだ状態で全部伸張してしまい，第二段階で例外の部分を正しい値で書き換える．このような二段階のステップで伸張を行うメリットは，現代のスーパースカラプロセッサでの処理を念頭に置くことで説明される．

スーパスカラプロセッサとは，逐次記述されたプログラムから動的に並列実行可能な命令を見つけ，それらを同時に実行するプロセッサである [41]．性能向上のために内部で並列処理を行っているが，プログラムの記述は逐次プログラムでよく，逐次実行の既存のプロセッサに対し，バイナリ互換を維持することが出来る．また，プロセッサの内部構造に関わらずプログラムは逐次プログラムでよいので，スーパスカラプロセッサ間でもバイナリ互換を維持することが出来る．一般に，逐次実行順ではない順番のことをアウトオブオーダ (*out of order*) という．これに対して，逐次実行順のことをインオーダ (*in-order*) あるいはプログラム順 (*program order*) という．プロセッサの中心部分である実行コアは，プログラムの実行時間を短くするためにアウトオブオーダで命令を並列に実行するが，逐次実行という外的側面を維持するために，フロントエンドとバックエンドが命令の流れをインオーダにしている．このような構成を取ることで逐次実行バイナリの命令レベルでの並列実行を実現している．

スーパースカラプロセッサは複数のステージ (*stage*) からなるパイプライン処理機構を実装することにより，*IPC*(*Instruction per Cycle*，1 プロセッサ・クロック分時間が経過する間に実行できる命令数) を 1 以上に高めるということを実現する [21]．このとき，命令のストリーム中に条件分岐が存在する場合は投機的実行を行うことになるが，分岐予測が外れた場合，分岐節の命令全部をやり直さなければならないというペナルティが発生する．そのため，パイプライン処理によって IPC を高めているような現代のスーパースカラプロセッサにおいてその性能を最大限引き出すためには，プログラム中で条件分岐の数を減らすという戦略が重要なものとなる [41].

このような戦略を前提に考えると，前述した PFor データ構造の伸張の方法の利点が明らかとなる．固定長ビット圧縮ブロックの伸張時に，例外かどうかの判定を都度おこなっていくとすると，整数列の要素数と同じ数だけ条件分岐が発生する．一方，前述の方法では固定長ビット圧縮ブロックを誤りも含めて全部伸張するステップでは条件分岐は一切存在しない．その次の例外の要素を修正するステッ

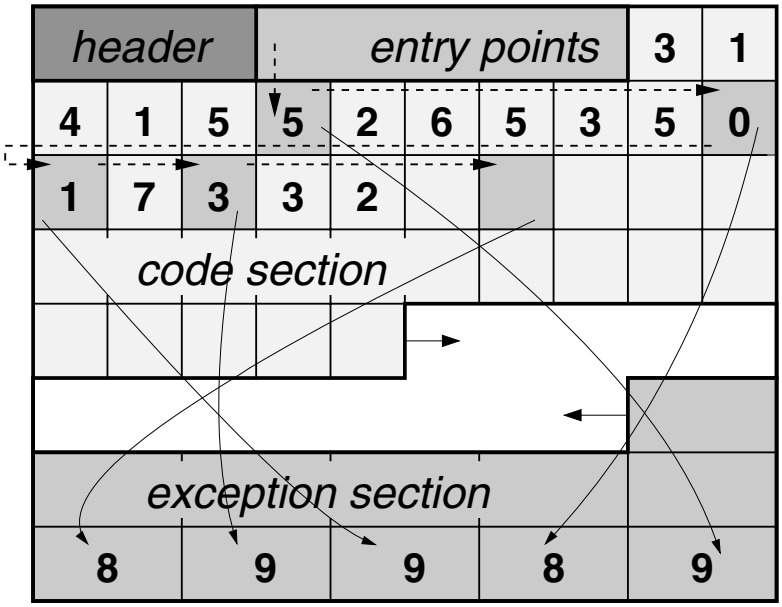

図 5.2: Zukowski らによる PFor データ構造の設計 (M. Zukowski et al., 2006, p. 59)

プではやや複雑な制御が必要となるものの全体数がもとの整数列の要素数の 1 割にも満たない程度であるので，トータルでの処理時間の上では支配的な割合とはならない．

Zukowski ら [40] における実装　図 5.2 に Zukowski らによる [40] での PFor データ構造の設計の概略を示す．この設計における特色は，固定長ビット圧縮ブロック中の例外の位置の値に，位置情報を保持するという点である．Zukowski らは例外に当たる部分も有効に活用することで，別に例外の位置情報を保持する必要を排し，空間効率を重視する設計をおこなっている．

このような設計であるため，固定長ビット圧縮ブロックのビット幅によって，例外間の取りうる距離に上限が存在してしまうという点が欠点となっている．そのため [40] では，例外同士の距離が上限を超えると，間を繋ぐ**強制例外**(*Compulsory Exception*) を設けるという方針になっている．

5.1.2　本研究以外の PFor 型データ構造の研究

前節で述べたように PFor 型のデータ構造の中でもとくに PForDelta は転置索引を用いた全文検索システムにおいて転置リストの圧縮に有効であることが複数の研究グループによって明らかにされ [38][40]，その後 PForDelta の実装上の欠点を克服することを目指した研究が行われた [37].

New-PFD [37] で提案され，Zukowski らの [40] の実装における強制例外を設けなければならなかった欠点を克服したものが *New-PFD* である．具体的には，固定長ビット圧縮ブロック中の例外の位置の値には例外の真値の下位 b ビット (b は固定長ビット圧縮ブロックのビット幅) を保持し，別に二つの配列を用意して，例外の真値の残りの上位ビットと，例外間の距離をそれぞれ保持する．さらにこれらの配列もまた，別の手法 (たとえば，*Simple16* など) で圧縮する，というものである．

Opt-PFD *New-PFD* と同じく，[37] で提案された手法で，整数列全体を通して同じビット幅を用いるのではなく，ブロックごとにビット幅を決定するというものである．圧縮率を最適化するという観点のみならず，伸張時のスピードも加味してビット幅をチューニング可能にしたという点が特徴となっている．

また，全体の圧縮率を考慮してビット幅を決定するという手法は Wan らによる [32][36] の研究においても議論がなされている．

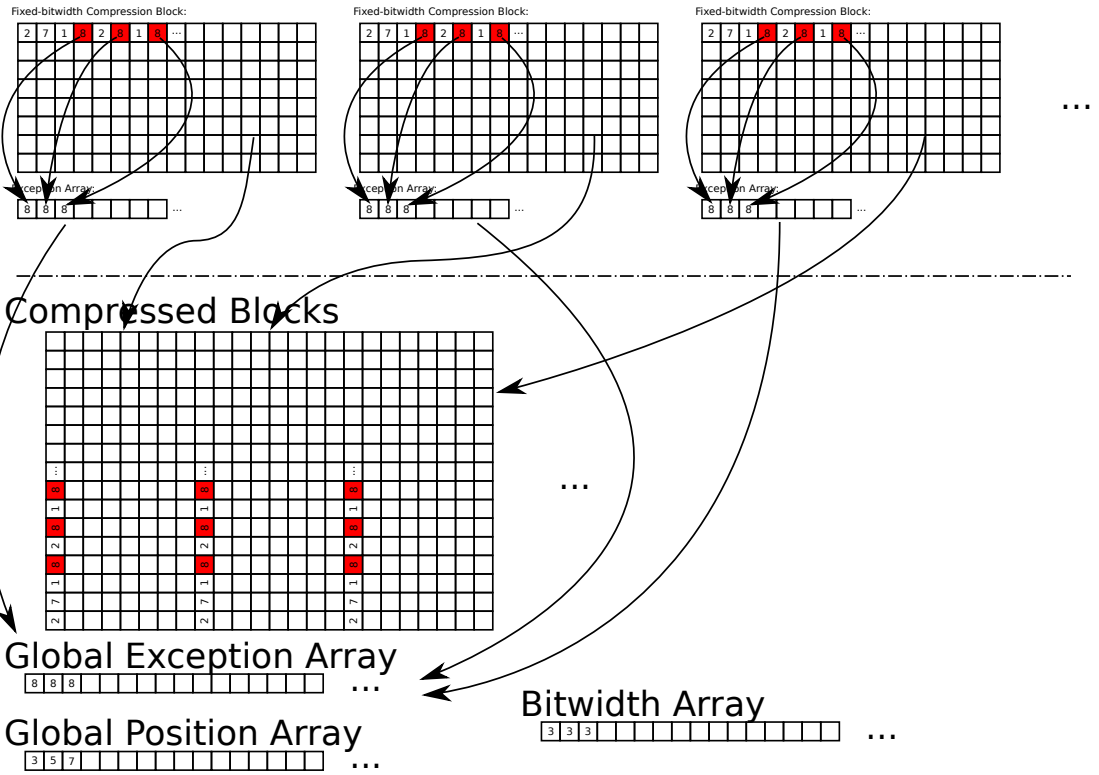

図 5.3: Fine-grained PForDelta の圧縮データ構造の概念図.

5.2 提案データ構造の詳細

5.2.1 Fine-grained PForDelta

入力の整数列を差分列に変換し，単位ブロックごとに PFor のフレームワークで圧縮することを考える．単位ブロックの大きさは，アプリケーションやハードウェアの構成に応じて 128(= 2^7) 個から 1024(= 2^{10}) 個程度の 2 のベキ乗個とする．2 のベキ乗個であるのはブロックの伸張処理において余計な端数処理を生じさせないためである．また，大きさについては索引が対応する文書数や，文書コレクションに含まれる文書の分布によって転置リストの大きさが変わることを想定している．

本研究で提案するデータ構造の概要を図 5.3 に示す．骨子は，ブロックごとに圧縮した整数列を，ブロックはブロック，例外は例外でそれぞれ整数列全体で繋がった構造にするということである．繋がったブロックを *Compressed Blocks*，例外の真値を記憶する配列を *Global Exception Array*(*GEA*) と呼称することにする．また付随して，例外の場所を記憶する配列を *Global Position Array*，また，*Compressed Blocks* の各ブロックのビット幅を記憶する *Bitwidth Array* も用意する．なお，*Global Position Array* の各要素はブロック中の位置を記憶するのでブロック長が 128 個の

場合 7 ビットで表され，1024 個とすると 10 ビットで表されるということになる．また，*Bitwidth Array* の各要素は圧縮ビット幅を記憶するので，とりうる値は 1〜32 であり 5 ビットで表される．

5.2.2 提案データ構造の圧縮アルゴリズム

ブロックの圧縮 まず 1 つのブロックを圧縮する処理から説明する．1 ブロックを処理する関数の処理の流れは以下の通り:

1. ブロック内の値を差分列に変換 (Δ-*encoding*)
2. 圧縮率が最も高くなるビット幅を決定 (5.2.4 節参照)
3. ブロック全体を 2 で決定したビット幅で固定長ビット圧縮されたブロックを生成
4. 2 で決定したビット幅でおさまらない値を持つ例外の値†，ブロック中の位置を保持した配列を生成
5. ブロックの長さ，例外の長さを出力

整数列全体の圧縮 次に，整数列全体の圧縮する処理の説明を行う．整数列から圧縮された整数列を生成する関数の処理の流れは以下の通り:

1. 整数列の長さからブロックの数を計算
2. 各ブロックを圧縮し，例外の値，位置を保持した配列を生成
3. 整数列全体でブロックの配列を生成 (*Compressed Blocks*)
4. 各ブロックの圧縮ビット幅を保持する配列を生成
5. 各ブロックの圧縮ビット幅を保持する配列を 5 ビット幅で圧縮 (*Bitwidth Array*)
6. 整数列全体の例外の値，位置を保持する配列を生成 (*Global Exception Array*, *Global Position Array*)
7. ブロックごとの圧縮 (2-6 の処理) に入らなかった端数データを例外の値，位置を保持する配列に追加

†なお，ここで例外として保持する値は差分ではなく，もとの整数列の値とする．

圧縮についてはこのような処理を行うので，各ブロックに含まれる例外の位置には例外の真値の下位のビットが保存されることになるが，本提案手法においてはこの情報は何にも使わない．このことは記憶領域の無駄になるというのは一面では真実であるが，一方で利点もある．ひとつは，Zhang らの *New-PFD*[38] と同様，強制例外を設ける必要がないという点である．それから，Zhang らの *New-PFD* のように例外の値を別の配列で保持する際上位ビットのみで済ますのではなく，真値自体を保持するのには，GEA をサーチにもちいることで細粒度のアクセスを実現するという狙いがある．

また，それだけでなく，決定的な理由として，例外の正確な数は入力によって決まるため，コンパイル時点ではわからないということがある．つまり，ブロックの伸張処理においてコンパイル時点で処理する数がわかっていないと条件分岐が頻繁に生じることになってしまうからである．

5.2.3 提案データ構造の伸張アルゴリズム

整数列全体の伸張 圧縮された整数列をすべてもとの整数列データに伸張する処理は以下の通り:

1. 各ブロックの圧縮ビット幅を保持する配列を伸張 (*Bitwidth Array*)
2. 1 で得られた情報を基に各ブロックを伸張
3. 各ブロック中の *Global Position Array* の位置の値を *Global Exception Array* の値で修正 (**例外の修正**)
4. 3 で修正した例外の位置を基点にブロックの差分列を復元
5. ブロックごとの圧縮に入らなかった端数データの書き込み

整数列データの部分伸張 整数列データの部分伸張の粒度はブロックごとの伸張と，GEA をもちいた例外間の伸張という大きく二つが存在する．

ブロックごとの伸張処理は整数列全体の伸張において伸張したいブロックのみを伸張すればよいので，とくに考慮しなければならない事柄はない．

GEA をもちいた二つの例外の間の伸張処理の流れは以下の通り:

1. GEA 中の二つの例外 Y_1, Y_2 に対応する位置を *Global Position Array* から発見する (P_{Y_1}, P_{Y_2})

2. P_{Y_1}, P_{Y_2} から *Compressed Blocks* におけるブロックレベルでの位置，ブロック内での位置を決定

3. P_{Y_1}, P_{Y_2} が含まれているブロックの圧縮ビット幅を *Bitwidth Array* から得る

4. 固定長ビット圧縮されている P_{Y_1}, P_{Y_2} が含まれるブロックの P_{Y_1}, P_{Y_2} 間の値を伸張する

5. 伸張した範囲について差分列を復元する

このようなことが可能であるのは，各 *Compressed Blocks* がそれぞれビット幅固定で圧縮されているためである．すなわち P_{Y_1}, P_{Y_2} の情報から簡単な計算でブロック中の伸張したい範囲を決定できる．伸張したい範囲が十分小さければ，ブロック全体を伸張するよりもその後の処理において効率的な処理が可能であることが期待できる．

また，GEA による例外間伸張はブロック単位および全体の伸張が例外を修正しなければならなかったのに対して，例外の修正が必要ないという点も大きな利点である．これは，伸張範囲が例外間であるので，当然その範囲に例外は存在しないためである．

5.2.4 圧縮ビット幅の決定法

提案データ構造は 5.2 節で述べた構成をしており，ブロックごとに圧縮をおこなうため，ビット幅を決定する際，圧縮率の観点から最適化が可能である．具体的には，ビット幅の選択肢は高々32 通りであるので，そのそれぞれの場合で圧縮をおこなったときブロックでのトータルで消費する記憶容量を計算し，最もその値が小さくなる値を選択すればよい．

5.2.5 ブロック長の決定法

一般にブロック長が短い方がよりデータの分布に適した圧縮が可能である．トレードオフはブロックごとの圧縮に関するメタ情報の記憶量との間に存在すると言える．

本手法の場合においてもブロックごとの圧縮に関してはブロック長が短い方が圧縮率が高くなることが期待できる．ブロック長を短くした場合，逆にコストが

増えるのは *Bitwidth Array* の数が増えるという点である．したがって，圧縮率の点から転置リスト全体の圧縮に関して，ブロック長を最適化するとすれば，ブロック長の選択肢のすべての場合で圧縮した結果必要となる記憶領域の大きさが最小になるように選べばよい．

さらに転置リスト全体でブロックごとに長さも可変とすることも考えられる．この場合，*Bitwidth Array* と同じ長さの *Bitlength Array* も用意することでさらに圧縮率を高めることも考えられる．ただし，この場合のパラメータの選択肢の数は少数ではなくなり，組み合わせ最適化が必要になるため，圧縮処理においてオーバーヘッドが巨大になると考えられる．本研究ではこのような戦略は指摘するだけに留め，扱わないものとする．

5.3 提案データ構造を用いた intersection アルゴリズム

提案データ構造を用いる intersection 処理は 2.1.4 節で述べた階層構造を用いる intersection に分類される．上部構造 (GEA) においてはバイナリサーチを行い，下部構造 (伸張した圧縮ブロック内の値) に対してはリニアサーチを行う．

5.3.1 Global Exception Array によるサーチと細粒度アクセス

ここで問題としているサーチ，探索とは，ある値 X が整数列中に存在するかどうかを調べる処理である．より具体的には，値 X が存在する場合には，その値 X の整数列中の位置 (配列の添字) を返し，値 X が存在しない場合はなんらかのマジックナンバー (–1 など) を返すというものである．

図 5.4 に提案データ構造を用いたサーチの概要を示す．骨子となるアイデアは整数列全体でブロックごとの例外の真値を繋げた GEA(*Global Exception Array*) に対して最初にサーチを行い，必要に応じて圧縮されたブロック内に降りてサーチを続けるというものである．

図 5.4 において，探索の対象となる値を X とする．まず，GEA に対してバイナリサーチを行い，$Y_1 < X < Y_2$ を満たすような隣り合う値 Y_1, Y_2 を見つける．次に，Y_1, Y_2 に対応する *Global Position Array* の値 P_{Y_1}, P_{Y_2} を見て，*Compressed Blocks* の中のブロックの位置およびブロック中の位置から X の存在する範囲を特定し，その部分を伸張する．伸張した配列の Δ-encoding を復元し，X についてリニアサーチを行う．

また，そもそも GEA のサーチの時点で値 X が見つかればその位置を返す．

5.3.2 SIMD 命令を用いたサーチ法

下部構造に対し，サーチを行う際には，SIMD 命令をあわせて活用することも可能である．具体的には，整数比較演算を複数同時におこなうことで最大でレジスタワード数 n 倍の高速化が期待できる．

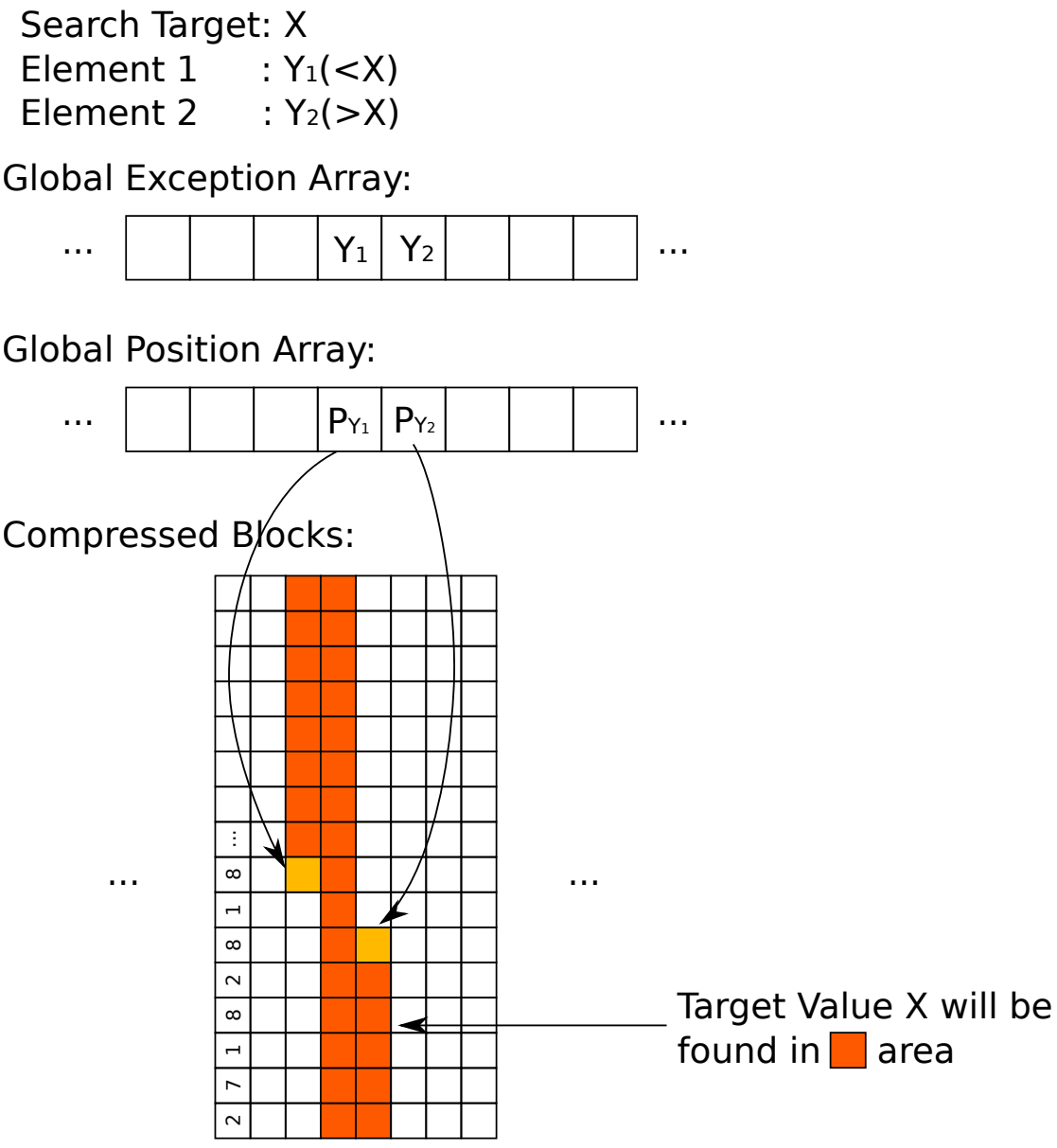

図 5.4: 提案データ構造を用いた search の概要.

5.4 実験と考察

本節では，提案手法によるサーチが細粒度で実現できるということを検証するためにおこなった実験について述べる．

Fine-grained PForDelta はデータの分布によって例外の数の割合が異なるため，なるべく現実のアプリケーションに近い形で有効性を示すべく，実験をおこなった．実験に使うデータセットには，The Text REtrieval Conference(TREC) の文書コレクション TIPSTER1 と TIPSTER3 をもちいた．これらは英語で書かれたニュース記事や，特許情報など自然文の文書である．

各データセットからもちいた文書は以下である．

TIPSTER1

- WSJ: Wall Street Journal (1987, 1988, 1989)
- AP: AP Newswire (1989)
- ZIFF: articles from Computer Select disks (Ziff-Davis Publishing)
- DOE: short abstracts from Department of Energy(DOE) publications

TIPSTER3

- SJM: The San Jose Mercury News (1991)
- AP: AP Newswire (1990)
- Patent: U. S. Patents (1983-1991)
- ZIFF: articles from Computer Select disks (Ziff-Davis Publishing)

これらのデータセットをプリプロセスし，索引付けして転置索引を生成した．プリプロセス処理は，記号類を除去して，Porter stemmer[‡]を使用してステミングし，索引付けをおこなった．

これらの文書コレクションはトータルで 820312 件の文書からなり，さらになるべく長い転置リストを生成するためこれらの文書コレクションについて 13 回同じ索引で索引付けした．

[‡]http://tartarus.org/ martin/PorterStemmer/

圧縮した整数列に対し，クエリとして与えられた整数をサーチするのにかかる時間を計測し，比較をおこなった．

クエリとなる整数は文書 ID の値の範囲 1〜10664056(= 820312×13) の範囲でランダムに選択した．

比較手法として，[24][30][31] で intersection 処理の代表例として説明されている，*Merge(Zipper)* 型と *Binary Search* 型を想定したサーチ手法と性能を比較した (2.1.4 節参照)．

まず，ランダムに選んだクエリ整数 100 個を，サーチごとに伸張する場合で提案手法と比較した結果を表 5.1 に示す．サーチする整数列としては，索引付けして生成した転置索引から 10 のベキ乗に近い長さのリストを抽出し，これを対象の整数列とした．サーチ対象の整数列を圧縮し，その状態からクエリをサーチするのにかかったプロセッササイクル数を測定した結果を示している．

実験に当たっては，100 個のクエリ (固定) を処理して計測することを 150 回繰り返し，その平均の数値を比較に使用した．

提案手法ではサーチ対象となる圧縮整数列を伸張する必要がないため，サーチを始める前に全部伸張しなければならない比較手法 (*Zipper*，*Binary Search*) と比較していずれの長さの整数列でも大きく有利になっている．また，このとき整数列の長さが約 10 倍ずつ増えるのに対して，比較手法はコストが約 10 倍程度ずつ増えていっているのに対し，提案手法では増分が 2 倍程度で留まっている．これは比較手法が整数列の長さが増えたときに受ける影響が 2 回存在する (1 回は圧縮データの伸張時，そしてもう 1 回は伸張した整数列のサーチ時) のに対し，提案手法において整数列の長さの影響を受けるのは GEA の長さが増え，最初の GEA サーチ時のみであり，下部構造である圧縮ブロック内をサーチするコストは整数列の長さによらず一定となっている．この実験結果は提案手法のそのような性質を確かに確認することが出来るものであると言える．

つぎに，クエリ，もとの整数列データは同じままで，比較手法をサーチの度に伸張するというものではなく，1 度伸張した整数列に対して連続してクエリをサーチした場合の実験結果を表 5.2 に示す．

この場合でも，比較手法が整数列の長さが増えるごとに約 10 倍程度コストが増えることがわかる．

しかし，本実験をおこなっていく中で，提案手法の課題も見つかった．それは例外を保持する構造が本稿のものは比較的重い (1 要素当たり 64 ビット) ため，空間効率の上で圧縮率を最適化した場合，例外が少なくなってしまうということで

ある．例外が少なくなってしまうと，サーチの粒度が粗くなってしまう他，最悪のケースではブロックごとに 1 つしか例外がないということになり，結局ブロックごとに代表値を集めてそれを目印にサーチをおこなうというのと変わらなくなり，手法としての面白さがなくなってしまうということも考えられる．

このような課題に対処する方法としてまず考えられるのは，*New-PFD* のように例外を保持する構造もまた別の手法で圧縮するということが考えられる．そうすれば空間効率上はいくらかの改善が期待できるがデメリットとして，サーチごとに例外の伸張から始めなければならなくなり，処理上のオーバーヘッドが避けられないという点がある．GEA は整数列本体の長さの 1 割未満であるため，それほど大きなオーバーヘッドにはならないと考えられるが，例外を保持するデータ構造の圧縮に関しても空間効率と処理効率のトレードオフが生じると言える．

表 5.1: Fine-grained PForDelta データ構造をもちいたサーチに関する実験結果．提案手法の性能を調べるため，比較手法として *Zipper* および *Binary Search* と比較をおこなった．10 のベキ乗に近い長さの整数列データをもちいて，ランダムに生成した 100 個のクエリに対するサーチにかかる時間を計測した．数値は，消費プロセッササイクル数の平均である．なお，このとき，*Zipper*，*Binary Search* 両手法については毎回圧縮データを伸張し，サーチをおこなっている．

# of Elements	*Zipper*	*Binary Search*	*Proposed Method (GEA Search)*
9997($\simeq 10^4$)	3.39E07	3.29E07	2.37E05
100022($\simeq 10^5$)	2.90E08	2.98E08	4.95E05
1000129($\simeq 10^6$)	2.84E09	2.84E09	1.16E06

表 5.2: Fine-grained PForDelta データ構造をもちいたサーチに関する実験結果．提案手法の性能を調べるため，比較手法として *Zipper* および *Binary Search* と比較をおこなった．10 のベキ乗に近い長さの整数列データをもちいて，ランダムに生成した 100 個のクエリに対するサーチにかかる時間を計測した．数値は，消費プロセッササイクル数の平均である．本実験では *Zipper*，*Binary Search* 両手法については一度だけ圧縮データを伸張し，以降はそのデータが保持されているものとしてサーチをおこなっている．

# of Elements	*Zipper*	*Binary Search*	*Proposed Method (GEA Search)*
9997($\simeq 10^4$)	1.80E06	3.41E05	2.37E05
100022($\simeq 10^5$)	1.76E07	2.98E06	4.95E05
1000129($\simeq 10^6$)	1.80E08	2.80E07	1.16E06

5.5 本章の結論

転置リストを表現する整数列の圧縮手法として近年注目されていた *Light-Weight Compression*(*LWC*) の一種である PForDelta を改良したデータ構造およびそれを用いたサーチ処理の手法を提案した.

提案手法では既存のデータ構造に対し，細粒度で圧縮データを伸張することが可能であり，その性質を利用したサーチ処理が既存のデータ構造と処理手法に比べて性能が向上したことを示した.

今後の課題は，現状では比較的重い例外を保持するデータ構造について空間効率を向上させるということである．また，その際別の圧縮法を使用する場合は例外構造の伸張処理のオーバーヘッドとのトレードオフをどのように扱うかが焦点となると考えられる.

第 6 章
結論

6.1 本論文のまとめ

本研究では，メインメモリ上の転置索引をもちいた全文検索システムのためのデータ圧縮処理技術として二つの手法を提案した．処理の内容としては，転置リストを表現する整数列データの圧縮・伸張に焦点を当て，とくにクエリ処理の効率という観点から研究を行った．

第一の手法として，既存の多くの整数列圧縮手法にもちいられていた要素技術である Δ-*encoding* された整数列を高速に復元する手法 *In-register Prefix Sum* を提案した．本手法は，現代のプロセッサに広く実装されているベクトル演算装置である SIMD 命令をもちい，追加のコストなしで処理の効率化を実現できる．整数列の長さによって，用いる SIMD 命令を切り替えることで，LLC の範囲内で最大で 3 倍弱，LLC の範囲外においても 1.5〜1.8 倍程度の性能の向上を示し，On-chip SIMD 命令を使用することで，一般に Prefix Sum 処理を行うアプリケーションの性能が向上する可能性を示した．

第二の手法として，転置リストを表現する整数列の圧縮手法として近年注目されていた *Light-Weight Compression*(*LWC*) の一種である PForDelta を改良したデータ構造およびそれを用いたサーチ処理の手法を提案した．

提案手法では既存の PForDelta 型のデータ構造に対し，細粒度で圧縮データを伸張することが可能であり，その性質を利用したサーチ処理の性能は既存のデータ構造と処理手法に比べて性能が向上したことを示した．

6.2 今後の展望

転置索引を用いた検索システムにおいて，整数列圧縮手法の展望として考えられるのは文書 ID の分布最適化に関する研究 [8] をもちいて，よりデータの分布の特徴を活かした圧縮スキームを目指すという点が挙げられる．とくに，本研究で提案した PForDelta 型データ構造は単語の出現分布が似た文書がクラスタ状になっている入力に対して効率的に圧縮できることが考えられる．このことは，索引に文書を追加することを考えるとき，追加分の文書のみに関してクラスタリングして文書 ID を割り当てることで，索引全体の最適化を必要としないながらも，有効な圧縮をおこなえることが考えられる．

また，本研究では提案手法の有効性を示すのに，同じ PForDelta 形データ構造をもちいた場合で比較をおこなったが，さらに今後は他の整数列処理手法として，ビットベクトルを並べた簡潔データ構造 [20] との比較や，あるいは全文検索システム全体での性能評価として接尾辞配列を用いた索引 [19] と性能比較をおこなっていくことが必要であると考える．

ハードウェアアーキテクチャとの協調の方向性としては，データ構造自体キャッシュライン等プロセッサハードウェアをより意識した構成にするというミクロな視点からと，システム全体で索引の配置をアーキテクチャに合致した形にするというマクロな視点からの両面において今後の研究の方向性が考えられる．

また，本稿では In-register Prefix Sum および Fine-grained PForDelta の両手法を提案し，それらの手法を用いたシステムの性能向上への可能性を示したが，さらにこれらの手法が現実に役に立つものになるためには，より現実に近いかたちで詳細な検証が必要となる．ひとつには現実に運用されている IR システムのコンポーネントの一部として組み込むことで，実際のワークロードに対するパフォーマンスを測定することである．また，ハードウェアについても現実のアプリケーションの設定において実際に性能が出るか，精細な実験が必要となるものと考えられる．

謝辞

本研究は多くの方々の暖かいご支援により，進めることが出来ました．ここに，篤く御礼申し上げます．

指導教官である，安達淳教授には，ご多忙の中にも日々熱意を持ってご指導頂きました．なかなか修士論文のテーマが固まらず，遅々として研究の進まない私を，辛抱強く見守って頂きました．また，ミーティングにおける先生の素朴な疑問に端を発する科学の議論は，情報科学の枠を超えて縦横無尽に展開され，研究者の姿勢について大変勉強になりました．

国立情報学研究所の高須淳宏教授には，お忙しい中，ミーティングの場を始め，研究の細かい点まで鋭いご指摘を頂きました．大変ありがとうございました．

研究室の博士課程の倉沢央さんには研究活動において大変お世話になりました．研究の方向性から細かい知識に至るまで，さまざまな面で勉強になったと思います．また，私が体調を崩していた際には家まで様子を見に来て頂きました．本当にありがとうございました．

研究室 OB の高橋輝さん，それから，現安達研究室メンバーの Chu Yimin さん，同期の木村光樹君，鈴木貴敦君，那小川君には，NII における日々の生活において大変お世話になりました．

また，NII の事務の久芳さん，上野さんには，書籍や備品など研究に必要な資材の購入などで迅速なサポートをして頂きました．そのほか久芳さんには私が体調を崩していた際に倉沢さんと一緒にお見舞いに来て頂きました．大変感謝しております．

最後になりましたが，学生生活をあらゆる面から支えていただいた両親，家族，友人への感謝の意を表して，謝辞とさせていただきたいと思います．

皆様，本当にありがとうございました．

参考文献

[1] IA-32 インテル®アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル. http://www.intel.co.jp/jp/download/index.htm.

[2] Vo Ngoc Anh and Alistair Moffat. Index compression using fixed binary codewords. In *Australasian Database Conference (ADC)*, pp. 61–67, 2004.

[3] Ricardo A. Baeza-Yates and Alejandro Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *String Processing and Information Retrieval (SPIRE)*, pp. 13–24, 2005.

[4] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, Vol. 23, No. 2, pp. 22–28, 2003.

[5] Guy E. Blelloch. Prefix sums and their applications. Synthesis of Parallel Algorithms. Ed. John H. Reif. pp. 35–60. Morgan Kaufmann, 1991.

[6] David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, Yoëlle S. Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 43–50, 2001.

[7] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, Vol. 1, No. 2, pp. 1313–1324, 2008.

[8] Shuai Ding, Josh Attenberg, and Torsten Suel. Scalable techniques for document identifier assignment ininverted indexes. In *INTERNATIONAL WORLD WIDE WEB CONFERENCES (WWW)*, pp. 311–320, 2010.

[9] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike J. Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *International Conference on Supercomputing (ICS)*, pp. 205–213, 2008.

[10] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, Vol. 54, No. 12, pp. 1901–1909, 1966.

[11] Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 348–357, 2007.

[12] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *International Conference on Data Engineering (ICDE)*, pp. 370–379, 1998.

[13] Goetz Graefe and Leonard D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. On Applied Computing*, pp. 22–27, 1991.

[14] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, Vol. 29, pp. 1170–1183, December 1986.

[15] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, Vol. 1, pp. 145–158, 1971.

[16] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD Conference*, pp. 339–350, 2010.

[17] Masaru Kitsuregawa and Toyoaki Nishida. Special issue on information explosion. *New Generation Computing*, Vol. 28, No. 3, pp. 207–215, 2010.

[18] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[19] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, Vol. 39, No. 1, 2007.

[20] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Algorithm Engineering and Experimentation (ALENEX)*, 2007.

[21] David A. Patterson and John L. Hennessy. コンピュータの構成と設計 第3版 成田 光彰訳. 日経BP社, 2006.

[22] Matthew Reilly. When Multicore Isn't Enough: Trends and the Future for Multi-Multicore Systems. In *HPEC*, 2008.

[23] S.E. Robertson, S. Walker, S. Jones, M.M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *Overview of the 3rd TREC Text REtrieval Conference,*

Gaithersburg, MD, D. Harman, Ed. NIST, NIST Special Publication 500-226, pp. 109–126, 1996.

[24] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Algorithm Engineering and Experimentation (ALENEX)*, 2007.

[25] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *International Parallel (and Distributed) Processing Symposium (IP(D)PS)*, pp. 1–10, 2009.

[26] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD Conference*, pp. 351–362, 2010.

[27] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. k-ary search on modern processors. In *International Workshop on Data Management on New Hardware (DaMoN)*, pp. 52–60, 2009.

[28] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *International Workshop on Data Management on New Hardware (DaMoN)*, 2010.

[29] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 175–182, 2007.

[30] Frederik Transier and Peter Sanders. Compressed inverted indexes for in-memory search engines. In *Algorithm Engineering and Experimentation (ALENEX)*, pp. 3–12, 2008.

[31] Frederik Transier and Peter Sanders. Engineering basic algorithms of an in-memory text search engine. *ACM Transactions on Information Systems*, Vol. 29, No. 1, p. 2, 2010.

[32] Jian Wan and Shengyi Pan. Performance evaluation of compressed inverted index in lucene. In *International Conference on Research Challenges in Computer Science (ICRCCS)*, pp. 178 –181, 2009.

[33] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, Vol. 29, No. 3, pp. 55–67, 2000.

[34] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, Vol. 2, No. 1, pp. 385–394, 2009.

[35] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[36] Xianghua Xu, Shengyi Pan, and Jian Wan. Compression of inverted index for comprehensive performance evaluation in lucene. *International Joint Conference on Computational Sciences and Optimization*, Vol. 1, pp. 382–386, 2010.

[37] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *INTERNATIONAL WORLD WIDE WEB CONFERENCES (WWW)*, pp. 401–410, 2009.

[38] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *INTERNATIONAL WORLD WIDE WEB CONFERENCES (WWW)*, pp. 387–396, 2008.

[39] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, Vol. 38, No. 2, 2006.

[40] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar ram-cpu cache compression. In *International Conference on Data Engineering (ICDE)*, p. 59, 2006.

[41] 安藤秀樹. 命令レベル並列処理. コロナ社, 2005.

[42] 定兼邦彦. 単純な rank/select 辞書. 電子情報通信学会技術研究報告, Vol. 106, No. 128, pp. 43–48, 2006.

[43] 田中洋輔, 小野廣隆, 定兼邦彦, 山下雅史. 高速復元可能な接尾辞配列圧縮法. 電子情報通信学会論文誌, Vol. 93, No. 8, pp. 1567–1575, 2010.

発表文献

[1] 渡辺健太郎, 高須淳宏, 安達淳, “メモリ上の全文検索システムのためのデータ構造と処理の効率化,” 第 73 回情報処理学会全国大会, 2P-9, 2011. (発表予定)