VAST 木: 木構造索引の圧縮を用いたSIMD 命令による 大規模データ探索の高速化

山室　健† 鬼塚　真† 日高　東潮† 山室　雅司†

† NTT サイバースペース研究所, 239-0847 神奈川県横須賀市光の丘 1-1
E-mail: †{yamamuro.takeshi,onizuka.makoto,hitaka.toshio,yamamuro.masashi}@lab.ntt.co.jp

あらまし 大規模データに対する範囲探索を効率的に処理可能な木構造索引 (VAST 木) を提案する．範囲探索可能な索引として木構造の 2 分木や B+木が広く用いられてきたが，これらの手法を大規模なデータに適用した場合，索引サイズと実行命令数の増大，実行効率の悪化が課題になることが判明した．そこで VAST 木では，木構造索引に対して不可逆な圧縮の適用，さらに近年注目されている CPU 内の SIMD 命令を活用することで従来手法の課題を解決する．この VAST 木のプロトタイプを作成し，2 分木と比べて索引の内部比較キーの総サイズを最大 1.04%と大幅に削減し，CPU 実行効率の指標である IPC を 0.246 から 0.433 に改善，実行命令数を 0.514%に削減可能であることを示す．
キーワード 大規模データ, 分析処理，木構造索引探索, SIMD 命令, 圧縮

VAST-Tree: Vector-Advanced Structure for Massive Data Tree-Traversal

Takeshi YAMAMURO†, Makoto ONIZUKA†, Toshio HITAKA†, and Masashi YAMAMURO†

† NTT Cyber Space Laboratories, 1-1 Hikarino-o-ka, Yokosuka Kanagawa, 239-0847 Japan
E-mail: †{yamamuro.takeshi,onizuka.makoto,hitaka.toshio,yamamuro.masashi}@lab.ntt.co.jp

Abstract We propose a compact and efficient index structure for range queries on massive data. There are widely-used and well-known approaches for these queries: the binary-tree and the B+tree. Unfortunately, we find that these techniques on massive data lead to three shortcomings; for one thing, the size of these indices is too large to fit in regular main memories, and the others are that inefficient processing on CPUs causes low instructions per cycle (IPC), and the number of instructions increases dynamically. Our state-of-the-art index technique provides a solution of these shortcomings by employing a compressed and SIMD-friendly structure. We implement its prototype and demonstrate that the index size was decreased up to 99.53% compared to the binary-tree. And also, it improved IPC to 0.433 from 0.246 and needed instructions was held down to 0.514%.
Key words Massive Data, Analyasis, Tree Traversal, SIMD, Compression

1. 研 究 背 景

近年，Google や Facebook を中心とした大規模データの分析処理に関わる研究が積極的に取り組まれている．サービスやシステムによって日々出力される大量のログなどのデータに対して行う処理は様々存在するが，最初の処理としてある特定部分のデータを抽出する操作を行うことが多いと言われている [1]．例えば，あるサーバ管理者がホスト A の最近 3 日間のアクセス数の度数分布を知りたいと思い，大量のサーバログに対して，この結果を取得するための処理を投入する場合を考える．データ全体に対して 3 日分のログの量が相対的に少ない場合には，ログ内に含まれるタイムスタンプ相当の属性を利用して，この部分的なデータを先行して抽出したほうが効率が良い場合が多い．タイムスタンプに限らず，相対的に少ないデータ量の属性を利用して部分的な抽出を行う処理を高速化する手法として，従来から範囲探索が可能なデータ構造である 2 分木や B+木のような木構造索引が広く利用されている．

サービスやサーバが 1 日出力するログの平均総データ量を約 10GiB 程度と考えた場合，3ヵ月の累積データ量は約 1TiB である(注1)．単体の平均ログ長を 0.1〜100KiB と仮定した場合，ログ件数が 2^{26}〜2^{32} 程度になる．この 3ヵ月分のログのタイムスタンプ列に対して，従来の木構造の索引を作成した場合の木構造索引サイズを表 1 に示す (B+木は PostgreSQL v9.0.1 の索引サイズの実測値である)．表を見てわかる通り，索引対象の総数が多い場合では，索引サイズが 10〜100GiB 程度になる．

(注1)：現在の HDD 容量や，Web 向けのサービスを行っている企業の公開情報を参考にデータ規模を設定した．

一般的に利用されるサーバのメモリサイズは 10GiB 前後のものが多いことから考えると，これは非常に大きいと言える．比較的メモリ上に残りやすいデータ構造である索引がメモリ量を超えた場合，極端な処理性能の悪化の原因となる可能性があるため，索引サイズは性能の重要な指標である．

	26	28	30	32
2 分木	1.38(0.63)	5.5(2.5)	22.0(10.0)	88.0(40.0)
B+木	1.4	5.6	23.5	89.7

表 1　2 分木と B+木の索引サイズ評価 (GiB，索引総数は 2^x で表記，括弧内は比較キーの総サイズ)

一方，近年のメモリ量拡大 [5] の傾向や，SSD に代表される高速な I/O デバイスの登場により，共有メモリ型マルチコア CPU 環境におけるデータベース内処理の CPU とメモリ間のボトルネックに着眼する研究 [6] が増えてきている．索引に関する研究も同様で，近年の索引に対する処理のボトルネックは I/O ではなく，CPU とメモリ間で発生することが大半で，この課題を取り扱った研究は多い [2] [8] [13]．この背景を前提に，2^{22}〜2^{28} 個のデータに対する 2 分木をメモリ上に作成し，範囲探索を行った場合の CPU 実行比率と実行命令数の評価結果を図 1 に示す (索引総数の括弧内に CPU の実行効率の指標値を表す IPC(注2) を示す)．前例の約 1TiB で 3ヵ月分のデータに対して 3 日分のログを抽出する操作を想定して，全体の 3%の範囲探索を行う．この評価に用いたサーバの CPU は Xeon X5260(コア数 2，最大メモリバンド帯域 10.7GiB/s) で，メモリサイズは 16GiB である．この結果から以下の 2 点に着目する．

（1） ストール時間が実行時間の 50%以上を占有

（2） CPU 内の実行命令数が索引総数に対して大幅に増加

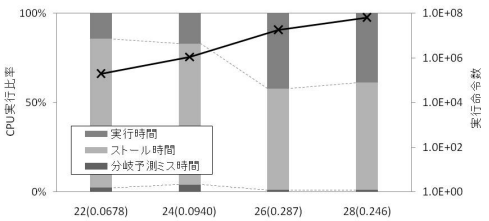

図 1　2^{22}〜2^{28} の 2 分木における CPU 実行比率評価 (括弧内は IPC)

(1) に関しては，木構造索引内の探索と範囲探索処理において発生するキャッシュや TLB ミス(注3)が原因だと考えられる．また (2) は，データ規模の拡大と共に探索する範囲も大きくなり，CPU 内で処理される命令数は飛躍的に増大する．しかし，(1) の影響で範囲探索処理の CPU 実行効率は非常に悪く，IPC の値が 0.0678〜0.287 と非常に低い値になっているため，実行命令数の増大に対して応答速度が悪化することが懸念される．

さらに図では示していないが，探索範囲を 3%より小さくした場合に，分岐予測ミス時間の割合が大きくなる傾向があることも判明している．これは処理全体の中における木構造索引内の処理の割合が大きくなり，ここでの比較処理に対する CPU 内の投機的な分岐予測処理が大量に発生していることが原因だと考えられる．単体探索に限った話になるが，分岐予測ミス時間の課題に対しては既に Intel の Changkyu K. ら [2] によって索引探索から分岐処理を取り除くことで対処され，さらに複数の比較処理を CPU の 1 サイクルで同時実行することが可能な SIMD 命令を活用することで処理の高速化を行う FAST が提案されている．しかし，単体探索に関する研究であるため範囲探索の議論はなく，(1) と (2) の課題は取り扱われていない．これらはデータ規模に比例して顕著になることが予想されるため，大規模なデータ処理に索引を適用する場合には考慮しなければならない重要な課題である．

さらに近年の CPU のコア数増大と共に，単体のコアで使用可能なメモリバンド帯域が年々減少 [4] 傾向にある．これは従来研究においても指摘 [2] [3] され，これらの研究では消費するメモリバンド帯域がアルゴリズムの重要な指標になっていると主張している．

本研究では，大規模なデータに対する木構造索引がメモリ上に載るように不可逆な圧縮を適用し，さらに CPU とメモリ間処理の最適化を考慮することでマルチコア CPU 環境上で効率的に範囲探索を処理可能な VAST 木を提案する (Section 3)．この VAST 木は，木構造内の探索で偽陽性的 (False Positive) な処理を許し，サイズが大きく，参照量の多い木構造索引下端部の比較キーを積極的に圧縮することで，背景で説明した課題解決を図る．圧縮の際，SIMD 命令で実行可能なより小さい単位の比較整数サイズに調整することで，命令の同時実行数を改善し，命令数の増大を抑制する．Section 4 では，VAST 木のプロトタイプを作成し，事前評価で挙げた課題がどの程度解決できているかについての分析を行う．Section 5 では，VAST 木が消費するメモリバンド帯域と，他の従来手法とを比較分析することで，マルチコア CPU 環境上における適応性を考察する．

2. 事前準備: SIMD 命令を利用した索引木構成

VAST 木では，木構造内の比較キーを圧縮することによる索引サイズの削減，ストール時間の低減と，さらに SIMD 命令を活用による実行命令数の抑制を扱う．しかし分岐予測ミス時間に関しては，従来通り探索から分岐処理を取り除くことで対処 [2] するため，ここでは事前準備として SIMD 命令を利用した索引木構成について概説する．

図 2 に SIMD 命令を利用した索引木の構成例を示す．SIMD 命令を適用することで，木構造索引の上部から複数の比較キーを同時に処理し，次に探索を行うべき部分木の決定を行うことで探索を進める．SIMD 命令によって同時に比較される部分木を，SIMD ブロックと呼び，図中では点線の三角形で表されている．SIMD ブロック間の移動には，SIMD 命令の結果ビット列を利用する (図 2 右上の SIMD レジスタ演算例)．SIMD ブロック単位の幅優先で比較キーの配置し，現在の位置と次の位

(注2)：Instructions Per Cycle の略で，CPU 内 1 サイクル当たりの実行完了命令数を表す．

(注3)：Translation Look-aside Buffer の略で，良く利用する論理アドレスと物理アドレスの対応関係をキャッシュしている CPU 内の機構である．

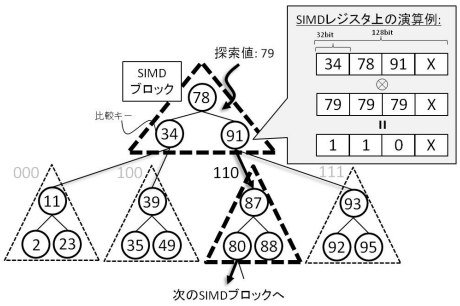

図 2　SIMD 命令を利用した木構造索引構成

置のオフセットを計算することで探索アドレスの移動を可能にする．このようにすることで，加算処理だけで木構造索引内の移動が実行できるようになり，分岐予測が発生しないように構造化されている．図 2 における SIMD ブロックは，メモリ上では以下のように幅優先配置されている (括弧で閉じられた比較キーが SIMD ブロックを表している) (注4)．

[34, 78, 91], [2, 11, 23], [35, 39, 49], [80, 87, 88], . . .

次に，このように構造化された比較キーの集合の探索方法について説明する．図 2 の太点線三角形であらわされた部分，また上記のメモリ上の配置例において下線で表された部分が今回の例の探索で処理される比較キーである．まず左端 3 つの比較キーを探索値 79 と SIMD 命令で同時に比較すると，結果ビット列として 1，1，0 が得られる (図 2 右上の SIMD レジスタ演算例)．ここで使用しているのは，被比較値以上であれば 1 を，未満であれば 0 を返却する SIMD 命令である．1，1，0 は次の SIMD ブロックの左から 3 番目に該当するため，以下の計算を行うことで次に探索を行うメモリ上のアドレスに移動する (図 2 における 2 つ目の太点線三角形への移動に対応)．current_idx は，現在の探索を行っている部分木の間順における先頭比較キーのメモリ上のアドレスを表している．

$$current\_idx \leftarrow current\_idx + 3 * 3 * 4$$

34 の探索アドレスを 0，つまり current_idx の初期値を 0 とする．SIMD ブロック内の比較キーの数は 3 であるため，結果ビット列から計算した 3 (2 段目左端 11, 2, 3 の SIMD ブロックを 1 として，3 番目の SIMD ブロックを 3 とする) と乗算することで，現在探索を行っている 34 と次に探索を行うべき部分木の先頭比較キー 80 との，比較キーのオフセット個数が計算できる．さらに各比較キーのサイズは 4 バイト (ここでは比較キーを 32bit の整数と想定) であるため，これを乗算することで次の部分木に移動するために加算しなければいけないメモリ上のオフセットアドレス値が計算できる．この計算の結果，次に探索を行うアドレスは 36 となり，2 個目の下線部先頭の 80 の探索アドレスに移動することになる．同様に SIMD 命令で 80, 87, 88 を同時に比較することで処理を継続し，探索を行うアドレスが索引サイズを超えた段階で処理を完了させる．

従来手法では索引対象 (32bit の整数，これ以降同様の条件を想定) と同じ整数サイズの比較キーを持つように構造化されるため，図中の比較キーは全て 32bit である．そのため現状利用可能な 128bit 長の SIMD 命令では，最大 4 つの同時比較が可能である．しかし SIMD 命令は 8bit なら 16 個，16bit なら 8 個の同時比較が可能である．そこで本研究では，圧縮後の比較キーのサイズを 8bit，16bit に揃えることで同時比較数の向上を図る．次節では，本提案の不可逆な圧縮手法について説明する．

(注4)：SIMD レジスタへのロード命令は，レジスタ上の格納順序 (図の例では 34, 78, 91) でメモリ上に連続配置されていない場合，複数のロード命令に分解されてしまうため，SIMD ブロック内の比較キーはメモリ上で間順に並べている．

3. 提案手法: VAST 木

3.1 VAST 木のデータ構造設計

まず VAST 木の設計方針に関して説明する．VAST 木は，従来手法において比較キーの数が多くサイズが大きくなりがちで，範囲探索処理の際に連続的に参照される索引下端部分の領域を積極的に圧縮するように構造化する．こうすることで索引サイズを削減しつつ，参照するデータ量を減少させ，CPU のストール時間を短くすることを可能にする．

まず索引対象の整数列を 32bit と想定して，図 3 のように 2 分木を以下の 3 領域に分割する．

- H_{32} の領域，比較キーを無圧縮 (32bit) で扱う部分木集合
- H_{16} の領域，比較キーを 16bit で圧縮する部分木集合
- H_8 の領域，比較キーを 8bit で圧縮する部分木集合

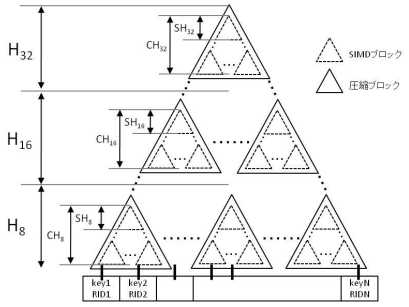

図 3　VAST 木の概要

この例では整数列を 32bit と想定しているため 3 領域に分割されているが，64bit を想定する場合 4 領域 ($SH64$, SH_{32}, SH_{16}, SH_8) となり，最上位の H_{64} が無圧縮という扱いになる．これ以降は索引対象を 32bit の整数列と想定し，3 領域分割を前提に話を進める．SIMD ブロック (点線の三角形) と圧縮ブロック (実線の三角形) の 2 つのまとまりで，各領域は 1 つの部分木を構成している．前者の SIMD ブロックは 1 回の SIMD 比較命令によって同時に実行される比較キーの集合，圧縮ブロックはこの後説明する圧縮手法を適用する単位である．各領域の高さはそれぞれ SH_x と，CH_x で表している ('x' には所属する領域の圧縮 bit 数が記載され，例えば H_{32} における SIMD ブロックの高さは SH_{32} となる)．

従来研究 [2] から，木構造探索の処理中に通過するキャッシュライン数と，メモリ上のページ数が性能に大きく影響すること

が分かっている．そのため VAST 木の各領域における CH の値は，探索処理を実行するサーバのキャッシュラインサイズと，メモリのページサイズを考慮して決定される．例えば，128bit の SIMD レジスタである場合，SH_{16} で同時実行される比較キー数は 15 となり，単体 SIMD ブロックサイズが 30B となる．ここで，CH_{16} を SH_{16} の 2 段分とした場合は圧縮ブロックのサイズが 510B，3 段分とした場合は 8190B となる．使用するサーバのページサイズを 4096B とした場合，これに収まる SH_{16} の 2 段分を CH_{16} に設定することになる．

また図 3 の H_{32}，H_{16}，H_8 は，任意に設定されるパラメータ値である．圧縮している下位の領域 H_{16} と H_8 では，Section 3.4 で説明するようにペナルティとして探索の際に偽陽性的な比較処理の誤りが発生する．そのため，これらのパラメータ値は比較処理の誤りの影響を考慮しながら性能分析する必要がある．

3.2 木構造索引の圧縮手法

前節の構造設計で述べたように，圧縮は圧縮ブロック単位で行う．H_8 の圧縮ブロック内に，8 つの比較キー (V_1〜V_8) がある場合を想定し，図 4 を用いて具体的な圧縮方法を説明する．以下の 4 つ処理を 8 つの比較キーに適用することで，圧縮ブロックを作成する．

（1） 圧縮ブロック内の最小値で，全ての比較キーを減算

（2） 最大値の Prefix 8bit と同位置にある全比較キー内の 8bit 列を取得

（3） (1) で取得した最小値と，(2) で取得した最大値の Prefix 位置を圧縮ブロックのヘッダに格納

（4） (3) のヘッダに続き，(2) で取得した 8bit 列を格納

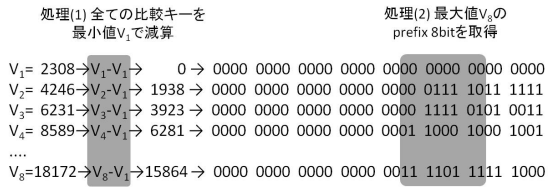

図 4 比較キーの圧縮概要

図の場合では，まず全ての比較キー値を最小値 V_1(=2308) で減算する．減算後の最大値 V_8 の 7〜14bit が 8bit の Prefix となるため，全ての比較キーの 7〜14bit 目を取得し，他の bit を削除することで不可逆な圧縮を行う．この後，取り出された比較キー bit 列と，最小値と，Prefix の位置を合わせて保存することで，1 つの圧縮ブロックとして出力する．この圧縮された比較キーと同時に保存された情報は，探索の際に利用する．

3.3 VAST 木構築処理の実装

前節で説明した VAST 木の構築を行うための疑似コードを Algorithm 1 に示す．まず VAST 木構築処理 (10〜24 行) に先立ち，入力データである索引対象データから 2 分木を作成 (9 行) する．構築処理は 3 つの繰り返し処理から構成され，一番外側の処理 (10〜24 行) で各領域 ($H[1] = H_{32}$，$H[2] = H_{16}$，$H[3] = H_8$) 毎の処理を行い，中段の処理 (11〜23 行) で圧縮ブロックの構築 (17〜19 行) とファイルへの書き出し (20〜22 行) を行い，一番内側の処理 (13〜16 行) で事前構築した 2 分木からの比較キーの取り出し (14 行) と並び替え (15 行) を行う．2 分木からの比較キー取り出し処理 (14 行) では，入力値 (i，j，k) を利用して，現在の処理対象となっている SIMD ブロックに対応した部分木の比較キーを ($2^{SH[i]} - 1$) 個取り出し，幅優先で比較キーを並べる．この処理を圧縮ブロック内の SIMD ブロック数だけ繰り返し (13〜16 行)，さらに SIMD ブロック単位で幅優先で並び替え (15 行)，圧縮ブロック内の比較キー配列を作成する．これが作成された後に，図 4 に従い配列内の最小値と最大値の Prefix 位置を取り出し (17〜18 行)，比較キー配列の圧縮を行う (19 行)．圧縮処理後は，圧縮に利用した最小値と Prefix 位置を圧縮ブロックのヘッダに書き出し (20〜21 行)，圧縮後の比較キー配列を書き出して (22 行)，単体圧縮ブロックの処理を完了させる．

Algorithm 1 VAST 木構築の疑似コード

```
 1: /*
 2: key_size: 索引対象の整数データ数
 3: keys[key_size]: 索引を作成する整数データ列
 4: region_num: 分割領域数，ここでは H32，H16，H8 の 3 領域
 5: H[region_num]: 各領域の高さを表すパラメータ H 値
 6: CH[region_num]: 圧縮ブロックの高さを表すパラメータ CH 値
 7: SH[region_num]: SIMD ブロックの高さを表すパラメータ SH 値
 8: */
 9: binary_tree = build_tree(keys[]);
10: for i ← 1 to region_num do
11:   for j ← 1 to H[i]/CH[i] do
12:     comp_array[] = {};
13:     for k ← 1 to CH[i]/SH[i] do
14:       simd_array[] = extract_tree(binary_tree, SH[i], i, j, k);
15:       push_array(simd_array[], comp_array[]);
16:     end for
17:     minval = get_minval(compress_array[]);
18:     prefix = get_prefix(compress_array[]);
19:     comp_array[] = comp_keys(comp_array[], minval, prefix);
20:     write_file(minval, file_p);
21:     write_file(prefix, file_p);
22:     write_file(comp_array[], file_p);
23:   end for
24: end for
```

3.4 圧縮木構造索引の探索処理と実装

VAST 木の探索処理は，以下 3 段階で行う．

（1） Tree Traversal Phase

（2） Validation Phase

（3） Range Scan Phase

(1) では，SIMD 命令を利用して木構造索引内の複数比較処理を同時に実行しながら探索を行う．各圧縮ブロック内で比較を行う前処理として，ヘッダに保存された情報 (最小値と，使用する Prefix 位置) を利用することで，その圧縮ブロック内で比較可能な値に探索値を変換する．H_8 における圧縮ブロックの探索処理の概要を図 5 に，疑似コードを Algorithm 2 に示す．探索したい値を “684942” とした場合，圧縮ブロック内の

ヘッダに含まれる最小値で減算し，指定される Prefix 位置を取り出すと “79” になる (10〜12 行)．この圧縮ブロック内では “79” の値を用いるため，これを SIMD 用レジスタに転送し (14 行)，探索処理を行う (16〜21 行)．探索処理は図 2 で説明したように，SIMD 比較命令を利用して結果ビット列を取得し (18 行)，このビット列から次の探索アドレスの計算 (19〜20 行) を行い，圧縮ブロック内の探索処理を継続する．圧縮ブロック内の探索が完了した後は，次の圧縮ブロックの探索アドレスを計算 (22〜23 行) し，探索位置を移動する．

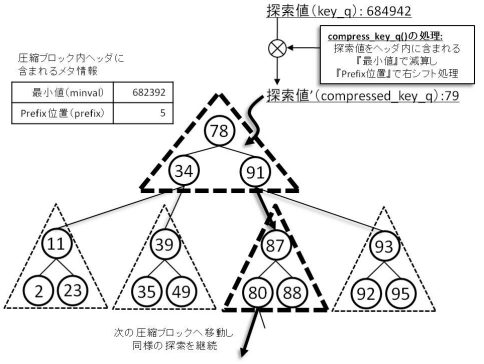

図 5　$H_8(SH_8 = 2,\ CH_8 = 4)$ における
圧縮ブロック内探索の例

Algorithm 2 H_8 における探索処理の疑似コード

```
1: /*
2: simd_offset: 圧縮ブロック内の SIMD ブロックの差分位置
3: compress_offset: 木構造索引内の圧縮ブロック部分木の差分位置
4: total_simd_size[x]: 高さ x における SIMD ブロックの総サイズ
5: total_compress_size[x]: 高さ x における圧縮ブロックの総サイズ
6: per_simd_size: 単体 SIMD ブロックサイズ
7: per_compress_size: 単体圧縮ブロックサイズ
8: */
9: for i ← 1 to H8/CH8 do
10:    minval = get_minval(current_idx);
11:    prefix = get_prefix(current_idx);
12:    compressed_key_q = compress_key_q(key_q, minval, prefix);
13:    current_idx += sizeof(minval) + sizeof(prefix);
14:    simd_key_q = simd_load8(compressed_key_q);
15:    simd_offset = 0, top_idx = current_idx;
16:    for j ← 1 to CH8/SH8 do
17:       simd_ckey = simd_load128(current_idx);
18:       simd_result = simd_compare8(simd_key_q, simd_ckey);
19:       simd_offset = simd_offset * 2^SH8 + calc_offset(simd_result);
20:       current_idx = top_idx + simd_offset * per_simd_size + total_simd_size[j];
21:    end for
22:    compress_offset = compress_offset * 2^CH8 + simd_offset;
23:    current_idx = compress_offset * per_compress_size + total_compress_size[H32 + H16 + i];
24: end for
```

しかし，Section 3.1 の構造設計で説明したように，圧縮手法によって削除された下位ビットの影響で，偽陽性的な誤った比較が発生する場合がある (図 6)．例えば，比較キー “3220”(2 進数で 110010010100) を Prefix 位置 4 で圧縮した場合，圧縮後の値は “201” である．ここで探索値が “3219” の場合，この探索値を Prefix 位置 4 で圧縮すると 201 となってしまい，正しい比較処理ができなくなってしまう．この誤った処理の修正は，(1) で行わず，索引の最下端まで処理が完了した後に (2) で行う．(2) では，索引対象の実値と探索値を比べ，目的位置まで順読み込みを行うことで対処する．そのため，探索のズレ Δw を少なくするように，H_{32}，H_{16}，H_8 を設定する必要がある．(2) の後に，要求された範囲の探索を (3) で行う．索引の下端部のデータはメモリ上の連続領域に並んでいるため，(2) と (3) は SIMD 命令を利用して SH_8 の SIMD ブロック内に含まれる比較キーの同時比較を行うことで高速化を図る．

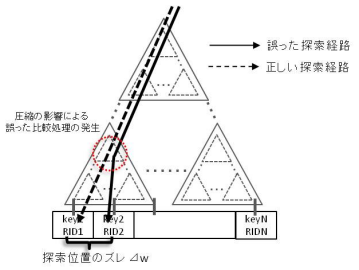

図 6　索引の圧縮による誤った比較処理

3.5 Validation Phase を利用した索引サイズの削減

ここでは，(2) の処理を利用した索引サイズの削減方法を説明する．図 7 に示す様に，VAST 木下端で SIMD ブロック内の比較キーの個数が 2^{SH_8} 以下になる場合がある．この端数の比較キーに対して，2^{SH_8} 以下で SIMD 命令の比較処理を適用することも考えられるが，背景で説明した通り，大規模なデータでは索引サイズが問題になるため，ここでは 2^{SH_8} 以下の比較キーの SIMD ブロックを索引構築の対象とせずに削除する．これを行うことで，探索のズレ Δw を最大で 2^{SH_8} 個大きくなり，(2) の処理量が増大する．しかし応答速度が，この処理の有無で大きく変化しないため，これを用いて評価実験を行う．

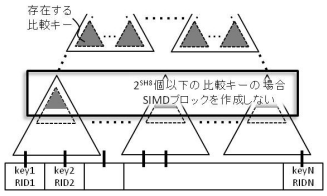

図 7　2^{SH_8} 個以下の比較キーの扱い

4. 評 価 実 験

評価で用いた VAST 木のパラメータ値を表 2 に示す (各値は 2 分木における高さに対応)．使用した OS は CentOS v5.5 で，kernel-2.6.18-194.26.1 である．評価用 CPU のキャッシュラインサイズが 64B，SIMD レジスタが 128bit で，kernel 指定のページサイズが 4096B であることを参考にパラメータ値を設

定した．また，索引対象のデータはポアソン分布 λ に従うイベントのタイムスタンプ (32bit 整数) 列を想定するが，整数列であれば本手法は適用可能である．特に記述が無ければ $\frac{1}{\lambda} = 8$ で評価を行っている．

パラメータ名	値	補足
SH_{36}	2	SIMD 同時比較命令数は 3
CH_{36}	4	内部に含まれる総比較キー数は 15
SH_{16}	3	SIMD 同時比較命令数は 7
CH_{16}	6	内部に含まれる総比較キー数は 63
SH_8	4	SIMD 同時比較命令数は 15
CH_8	8	内部に含まれる総比較キー数は 255

表 2　今回の評価の用いた VAST 木の各パラメータ値

性能評価を行う際に用いたサーバの CPU は Xeon X5670(コア数 6，最大メモリバンド帯域 51.2GiB/s) である．oprofile の対応 CPU の関係で CPU のプロファイリングを利用する評価においては，X5260(コア数 2，最大メモリバンド帯域 10.7GiB/s) を用いている．評価に利用した oprofile の Version は 0.9.6 である．全ての実験においてメモリサイズが 16GiB のサーバを用い，各パラメータ H_{32}，H_{16}，H_8 を変化させて作成した VAST 木と，索引サイズが 16GiB 以下の従来手法をオンメモリ上で比較評価する．

4.1　VAST 木の性能評価

まず，各パラメータ値における VAST 木と従来手法の索引サイズを表 3 に示す．ここでの索引サイズは，図 3 下部の索引対象の (key, RID) を除いた値である．H_8 の値は，$x-(H_{32}+H_{16})$ によって定まる．表を見てわかる通り，2^{30} 個の VAST 木 ($H_{32}=8$, $H_{16}=6$) の索引サイズにおいて，FAST の 0.146%，2 分木の 1.04%まで大幅に削減できていることが分かる．

	H_{32}	H_{16}	24	26	28	30
VAST 木	4	6	0.00587	0.00587	0.564	0.564
VAST 木	4	12	0.00153	0.141	0.141	0.376
VAST 木	8	6	0.0353	0.0353	0.0939	0.0939
VAST 木	8	12	0.00989	0.0245	0.0245	2.26
FAST	-	-	0.251	1.25	1.25	64.25

表 3　索引内部の比較キー総サイズ評価 (GiB，索引総数は 2^x で表記)

また各パラメータ値と索引サイズの関係に関して，圧縮率の高い H_8 の領域が最も大きい VAST 木 ($H_{32}=4$，$H_{16}=6$) の索引サイズが最小になっていないことも分かる．これは，SIMD 命令によって同時に比較できる個数以下の最下端の比較キーを索引対象としない最適化による影響である (Section 3.4)．この影響により，探索のズレ Δw が最大で 16 増加するが，この後の探索位置の平均ズレ Δw の評価結果 (図 10) から判断すると，この増加量は相対的に小さいため，性能への影響は少ないと考えることができる．

次に VAST 木 ($H_{32}=8$，$H_{16}=6$) の CPU 実行比率と実行命令数を図 8 に示す．従来手法の課題で挙げたストール時間と，それに伴う IPC の悪化の課題に関しては，FAST に対しては多少劣るものの，2 分木に対してはストール時間割合を 60.22%から 46.8%まで抑制し，IPC を 0.246 から 0.433 まで改善できている．一方，実行命令数は 2 分木に対して 0.514%まで，FAST に対して 1.93%まで大幅に削減できているため，IPC の向上と併せて応答速度の改善が期待できる．これを評価したものを図 9 に示す．VAST 木のパラメータ値にもよるが，従来手法と比較して 1〜2 桁程度の応答速度改善ができていることが分かる．

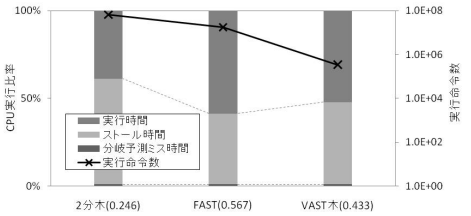

図 8　2^{28} における CPU 実行比率の評価 (括弧内は IPC)

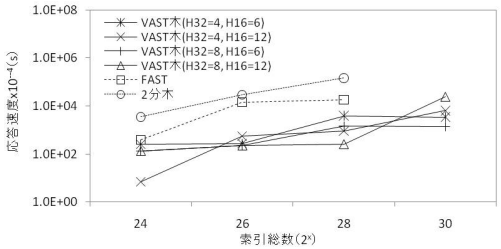

図 9　範囲探索応答性能の評価 (探索範囲 3%)

最後に，索引対象の分布に対する探索のズレ Δw の評価結果を図 10 に，探索範囲の変化に対する応答性能の評価結果 ($H_{32}=8$，$H_{16}=6$) を図 11 に示す．図 10 の分布に対するズレ Δw の変化に関しては，索引上端の H_{32} の無圧縮の比較キーの領域が大きいほうが，Δw が小さくなっていて，上端におけるズレが平均的な Δw 値に大きく影響していることが分かる．今回の評価では 32bit の整数範囲の制約で索引総数 2^{28}，$\frac{1}{\lambda}=128$ までしか評価ができなかったが，索引対象の分布が歪んでいる場合には Δw が大きくなると予想されるため，今後 64bit の整数列における評価と共に分析が必要だと考える．図 11 の評価結果からは，データ規模が大きく探索範囲が 0.1〜10%で従来手法に対して優位であることが分かる．しかし VAST 木の Validation Phase の影響で，単体探索 (範囲探索が 0 の箇所) に関しては，FAST と比べ 2〜3 桁応答速度が劣っていることも併せて分かる．

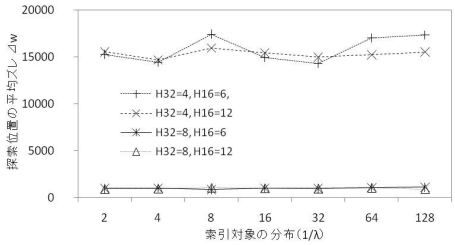

図 10　2^{28} における索引対象の分布変化に対する Δw 評価

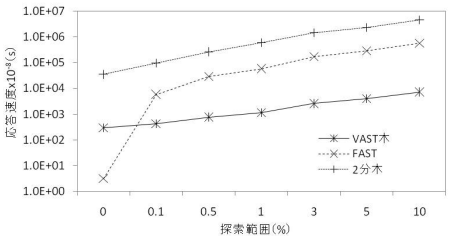

図 11　2^{28} における探索範囲の変化に対する応答性能評価

5. VAST 木の考察

本稿における考察観点は以下 2 点である．

- H_{32}，H_{16}，H_8 に対する最悪値と平均値 Δw の考察
- VAST 木の消費メモリバンド帯域の考察

5.1 H_{32}，H_{16}，H_8 における最悪値と平均値 Δw

VAST 木のパラメータ H_{32}，H_{16}，H_8 を，探索の際に発生するズレ Δw の最悪値と平均値性能の観点から考察する．まず最悪値は H_{32} に依存する．最上位から H_{32} までの比較キーは無圧縮で管理されているため，探索のズレが発生しない．そのため，H_{32} 以降 (H_{16} と H_8 の領域) において発生するズレの和が最悪値のズレ Δw となる．H_{32} が与えられた場合の，最悪値 W_{worst} は以下の式で定められる (2 分木の高さを H と表す)．

$$W_{worst}(H_{32}) = \sum_{k=1}^{H-H_{32}} 2^k \qquad (1)$$

例えば，2^{30} における 3%の探索範囲の 5%は約 53,687,091 で，この値を許容する最悪値とした場合，$W_{worst}(H_{32}) \leqq 53687091$ を満たす最大の H_{32} は 12 となる．しかし，H_{32} 以降の領域で全て探索を誤る確率は非常に低く，この最悪値は厳密な値ではないと考えられる．索引対象の分布で最悪値や，この後説明する平均値が大きく変化し，VAST 木の性能に大きく影響すると予想されるため，今後はより詳細な分析を行う予定である．

次に平均値に関しては，H_{16} と H_8 によって決定される．これらのパラメータ値によって決定される領域では，各比較キーの下位ビットを削除することで圧縮しているため探索のズレが発生する．ただし H_{16} は H_8 の領域に比べ，表現ビットが多いため探索のズレが発生する確率が相対的に少ない．一方 H_8 は確率は高いが，表現ビットが少ないため圧縮率が高い特性がある．これを評価したものを図 12 に示す．この評価では，索引総数を 2^{30}，$H_{32} = 12$ を用い，各パラメータ値の括弧内は索引サイズ (GiB) を表している．

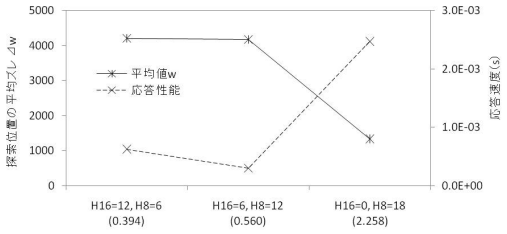

図 12　H_{16} と H_8 の変化に伴う各評価

評価結果を見ると，ここでも Section 3.4 の最適化の影響で，必ずしも H_8 の領域が大きい VAST 木 ($H_{16} = 0$，$H_8 = 18$) の索引サイズが最小になっていないことが分かる．また Δw が最も大きいもの ($H_{16} = 12$，$H_8 = 6$) が最悪の応答性能にならず，逆に Δw が小さいもの ($H_{16} = 0$，$H_8 = 18$) も最良の応答性能になっていないことから，最適なパラメータ値の決定を Δw の推定値からのみでは決定できず，CPU の特性や索引サイズまで考慮しなければいけないことが分かる．この結果の場合，索引サイズの優先度が高ければ $H_{16} = 12$，$H_8 = 6$ を，応答性能の優先度が高ければ $H_{16} = 6$，$H_8 = 12$ のパラメータ値を採用すれば良いことが分かる．

従来研究で提案されている CPU 特性を考慮した探索性能の推定手法 [7] や，Δw の最悪値と平均値，索引サイズなどを考慮して機械的にパラメータ値を決定する手法を現在検討中である．

5.2 VAST 木のメモリ帯域消費量の考察

近年，CPU のマルチコア化が進む背景で処理が消費するメモリバンド帯域が重要視されている [2] [3]．これはコア数に対する使用可能なメモリバンド帯域が年々減少傾向にあることから，過度に処理がメモリバンド帯域を消費してしまうと，データ転送可能上限に先に達してしまい，マルチコア CPU の性能を十分に発揮できないことが原因である．本手法の VAST 木は，索引の下端部を圧縮したことにより，範囲探索に必要なメモリ参照量を削減し，結果として消費メモリバンド帯域が減少していることが予想される．oprofile を用いて分析した 2^{28} における各手法の 1 回の範囲探索に消費するメモリバンド帯域を図 13 に示す．この評価に用いた VAST 木のパラメータ値は，各索引総数において索引サイズが最小になる値を用いている．この結果から従来手法と比較して，2^{24}〜2^{28} で 2 桁程度の大幅な消費メモリバンド帯域の節約が確認でき，索引総数が 2^{28} で 2 分木に対して 0.271%，FAST に対して 1.67%まで削減できている．この結果から考えて，今後メモリバンド帯域の制約が厳しくなると予想される CPU 環境上に対して，従来手法と比較してより優れていると判断できる．

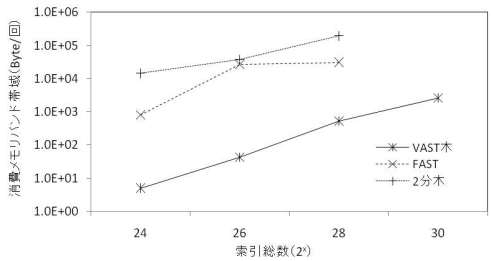

図 13　各手法における消費メモリバンド帯域の評価

参考性能値として，コア数の多い Xeon X5760 上でのスループット性能 (単位時間当たりに完了した範囲探索処理数) の評価結果を図 14 に示す．この CPU は oprofile に対応していないため，メモリバンド帯域がボトルネックになっているかについては今回分析できなかった．しかし，図 13 の単体範囲探索に消費するメモリバンド帯域を，図 14 の単位時間のスループット値で等倍しても，Xeon X5760 の最大メモリバンド帯域

(51.2GiB/s) を超えていないことから，今回の評価環境では，全ての手法が全ての測定値において，マルチコア CPU の性能が最大まで利活用できていると推測される．VAST 木は従来手法と比較して実行命令数も削減 (図 8) できていることから，メモリバンド帯域がボトルネックにならないような状況下においてもスループット性能の改善が期待でき，この測定境においては 2^{24}〜2^{28} の範囲で約 100 倍程度の向上が確認できる．

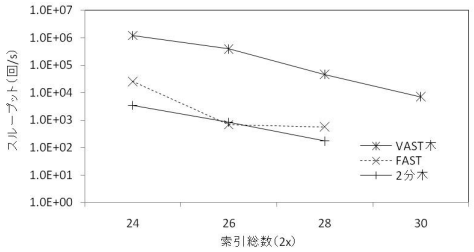

図 14　コア 6 の Xeon X5760 上でのスループット性能評価

6. 関 連 研 究

2002 年頃以降，CPU の SIMD 命令を用いた DB の性能向上に関する研究は盛んに行われている [2] [3] [8] [9] [10] [11] [12] [13]．探索の索引高速化に関する従来研究 [2] [8] [13] は存在するが，これらの研究は全て単体探索に着眼するもので，大規模なデータに対する範囲探索を行う処理の高速化に SIMD 命令を適用したものは本研究が初めてである．ハードウェアを利活用した DB の性能向上化という観点で言えば，GPU を用いた従来研究 [14] [15] [16] [17] も多い．現状の GPU と CPU はメモリ空間が異なるため，処理に先立って必要なデータを GPU 上のメモリに転送する必要がある．しかし，この転送処理時間が全体の 15〜90%程度占有されてしまっているという報告 [14] があるように，決定的な DB の性能向上には至っていないのが現状である．この課題に関しては，Intel や AMD が積極的に取り組んでいる CPU と GPU の統合チップの開発(注5)により解決すると予想されるため，今後はこれらのハードウェアを利用した研究がますます重要になると考えられる．

7. 結　　論

本研究では，大規模なデータの中から相対的に小さい範囲を探索するために木構造索引を使用する状況下で，索引下端部の圧縮と SIMD 命令を適用することにより，従来手法ではメモリサイズを超えてしまう規模への適用と，さらにメモリ上に乗るような環境においても性能改善を可能とする VAST 木の提案を行った．本手法では 2 分木や FAST などの従来技術と比較して，索引サイズの縮小化，CPU 実行効率の改善，さらに実行命令数の大幅な削減を可能にした．これらの改善の結果，索引がメモリ上に乗るような状況下でも，範囲探索の応答性能が従来手法に対して 1〜2 桁の改善が可能であることを合わせて示した．考察においては単体の範囲探索が消費するメモリバンド帯域を評価し，従来手法に対して大幅に消費量を節約できていることを示した．この分析により，メモリバンド帯域の制約が厳しくなる今後の CPU 動向においても，従来手法と比べ優れていることを明らかにした．

今後は，継続的に以下の課題に着手する予定である．

- 図 3 下部の索引対象 (key, RID) の圧縮検討
- Δw の最悪値と平均値の厳密な解析手法の検討
- 機械的な最適パラメータ値 (H_{32}，H_{16}，H_8) の決定
- より大規模なデータ (2^{32} 以上の 64bit 整数列) での評価

(注5)：Intel は 2011 年 1 月に初の GPU 内蔵 CPU である Sandy Bridge をリリースし，AMD も年内中に市場投入予定である．

文　　献

[1] Spyros B. et al. “A comparison of join algorithms for log processing in MapReduce”, Proc. of the 2010 international conference on Management of data, 2010.

[2] Changkyu K. et al. “FAST: fast architecture sensitive tree search on modern CPUs and GPUs”, Proc. of the 2010 international conference on Management of data, pp. 339-350, 2010.

[3] Nadathur S. et al. “Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort”, Proc. of the 2010 international conference on Management of data, pp. 351-362, 2010.

[4] M. Reilly “When multicore isn ’t enough: Trends and the future for multi-multicore systems”, HPEC’08, 2008.

[5] M. Stonebraker et al. “The End of an Architectural Era (It’s Time for a Complete Rewrite)”, VLDB’07, 2007.

[6] Peter Boncz et al. “Database Architecture Optimized for the new Bottleneck: Memory Access”, VLDB’99, 1999.

[7] Richard A. Hankins, and Jignesh M. Patel “Effect of node size on the performance of cache-conscious B+-trees”, Proc. of the 2003 international conference on Measurement and modeling of computer systems, 2003.

[8] J. Zhou et al. “Implementing database operations using simd instructions”, Proc. of the 2002 internatinal conference on Management of data, 2002.

[9] Kenneth A. Ross “Efficient hash probes on modern processors”, IEEE 23rd International Conference on Data Engineering, 2007.

[10] J. Chhugani, et al. “Efficient implementation of sorting on multi-core SIMD CPU architecture”, Proc. of the VLDB Endowment, Vol. 1, Issue. 2, 2008.

[11] T. Willhalm et al. “SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units”, Proc. of the VLDB Endowment, Vol. 2, Issue. 2, 2009.

[12] B. Schlegel et al. “Fast Integer Compression using SIMD Instructions”, Proc. of the Sixth International Workshop on Data Management on New Hardware, 2010.

[13] B. Schlegel et al. “kary search on modern processors”, Proc. of the Sixth International Workshop on Data Management on New Hardware, 2009.

[14] W. feng et al. “Database Compression on Graphics Processors”, Proc. of the VLDB Endowment, 2010.

[15] N. Govindaraju et al. “Fast computation of database operations using graphics processors”, Proc. of the 2004 international conference on Management of data, 2004.

[16] N. Govindaraju et al. “GPUTeraSort: high performance graphics co-processor sorting for large database management”, Proc. of the 2006 international conference on Management of data, 2006.

[17] B. He et al. “Relatinal joins on graphics processors”, Proc. of the 2006 international conference on Management of data, 2008.