情報処理学会論文誌　データベース　Vol.8　No.2　30–42 (June 2015)

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

山室 健[1,a)]　鬼塚 真[2,b)]

受付日 2014年12月21日, 採録日 2015年4月7日

概要：大規模なデータに対して空間コストが低く，CPU の実行効率が高い on-memory の木構造索引（VAST 木）を提案する．2 分木や B+ 木に代表される索引は基本的でかつ重要なデータ構造として幅広く活用されている．しかしデータの大規模化に対して索引サイズの肥大化や CPU の実行効率低下の問題が顕在化する．特に CPU の実行効率の改善には，並列化・分岐処理・キャッシュ構造など様々な技術的な課題を考慮する必要がある．VAST 木ではデータ並列度が高い比較処理を行うために分岐ノードには不可逆な圧縮手法を，葉ノードには CPU に最適化された可逆な圧縮手法をそれぞれ適用する．この設計により索引全体のデータ削減を行いながら実行効率の改善も同時に実現する．データ数が 2^{30} の分岐ノードのサイズは既存手法に対して 95%以上削減，また索引全体のサイズで 47〜84%削減した．データ数が 2^{28} の探索では，スループット性能は 2 分木に対して 6 倍，既存手法で最も効率的な FAST に対して 1.24 倍の性能向上を確認した．探索処理中のメモリ帯域の消費量も 2 分木に対して 94.7%，FAST に対して 40.5%削減している．

キーワード：木構造探索，SIMD 命令，圧縮

Vast-Tree: A Vector-advanced and Compressed Structure for Massive Data Tree Traversal

Takeshi Yamamuro[1,a)]　Makoto Onizuka[2,b)]

Received: December 21, 2014, Accepted: April 7, 2015

Abstract: We propose a compact and efficient on-memory index structure, called VAST-Tree, for massive data sets. Several indexing techniques such as binary trees and B+trees are widely-used in the database community. Unfortunately, we find that these techniques suffer two major shortcomings when applied to massive data sets; for one thing, their indices are so large that they could overflow regular main memory. For another, they suffer from a variety of penalties (e.g., conditional branches and cache misses). VAST-Tree applies lossy compression to keys in branch nodes and processor-friendly compression to keys in leaf nodes in order to reduce the size of indices. This lossy compression can also improve data parallelism for key comparisons in branch nodes. Compared to current alternatives, VAST-Tree can compress branch nodes by more than 95% and the overall index size by 47∼84% for 2^{30} keys. With 2^{28} keys, VAST-Tree has roughly 6.0-times and 1.24-times throughput and saves the memory consumption by more than 94.7% and 40.5% as compared to binary trees and state-of-the-art FAST, respectively.

Keywords: Tree traversal, SIMD, compression

1　NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center, Musashino, Tokyo 180–8585, Japan
2　大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
a)　yamamuro.takeshi@lab.ntt.co.jp
b)　onizuka@ist.osaka-u.ac.jp

1. 研究背景

データ集合から指定された条件に合致した部分集合を高速に探索するため 2 分木や B+ 木に代表される索引は基本的でかつ重要なデータ構造として幅広く活用されている．これらのデータ構造は索引対象のデータ数が多くなった場

表 1　2 分木と B+ 木の索引サイズ（GiB）比較．括弧内は分岐ノードのサイズを示す

Table 1　Total sizes (GiB) of binary trees and B+trees. The values in parentheses show only the sizes of branch nodes.

データ数	2^{28}	2^{30}	2^{32}
2 分木	5.5 (2.5)	22.0 (10.0)	88.0 (40.0)
B+ 木	5.6	23.5	89.7

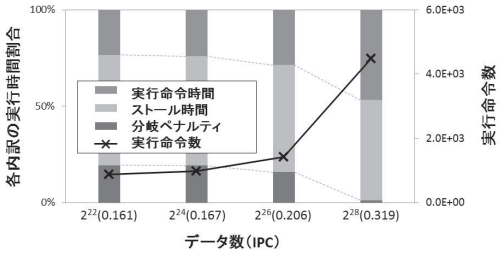

図 1　索引対象のデータ数が 2^{22}～2^{28} における Xeon5260 を用いた実行時間内訳（命令実行時間，ストール時間，分岐ペナルティ）比較．括弧内は各実行における IPC を表す

Fig. 1　Ratios of execution time and counts of instructions for exact-match scans on a Xeon X5260 processor. The values in parenthesis indicate IPC in the processor.

合に索引サイズの肥大化と CPU の実行効率（CPU の 1 サイクルあたりの実行命令数）の低下の問題が顕在化する．**表 1** にデータ数が 2^{28}～2^{32} に対する 2 分木と B+ 木の索引サイズを示す．2 分木は広く知られた一般的な方法で実装を行い，B+ 木は PostgreSQL v9.0.1 の索引構造を用いた．データ数が多くなると索引サイズが 10～100 GiB 程度になり，現在市販されているサーバのメモリサイズと比較しても非常に大きいことが分かる．索引がメモリサイズを超えると極端な性能劣化が発生するため，索引のサイズ削減は重要な課題の 1 つといえる．

CPU を考慮した設計を行っていない手法の実行効率は一般的に低く [1]，木構造索引に関しても既存研究で分岐処理やキャッシュ・TLB [*1] ミスによるストール時間が探索処理の実行効率を下げていることが報告 [9] されている．**図 1** に Xeon5260 を用いたデータ数が 2^{22}～2^{28} の 2 分木の完全一致探索の実行時間内訳を示す．分岐処理によるペナルティとストール時間が内訳の約 60%～80%を占めており，結果的に実行効率を示す IPC（Instructions Per Cycle）は 0.3 前後であることが分かる．

木構造索引における CPU の実行効率（命令あたりに必要な CPU サイクル数）の課題を取り扱った最新の提案は Kim らによる FAST [12], [13] である．FAST は on-memory で更新がない処理を前提に，CPU を考慮したデータ構造設計と探索処理を行うことで分岐処理によるペナルティやキャッシュミスを改善している．また 1 つの命令で複数のデータを処理可能（SIMD: Single Instruction Multiple Data）なベクトル命令を活用することで比較処理の高速化も実現している．しかし，FAST を規模の大きなデータに適用する場合に以下のような課題が存在する．

- 効率化のためにキャッシュラインやページに対する内部構造のアラインメント調整をするが，データ規模が大きくなった場合に多量のパディング領域による索引サイズの肥大化が発生する．
- SIMD 命令で分岐ノード内の比較キーの同時処理を行うが，最大のデータ並列数が 3 に制限されている．
- 葉ノードの圧縮が取り扱われていない．

本研究の目的は大規模なデータに対して空間コストが低く，探索における比較処理の並列数を改善した圧縮木構造索引である VAST 木を提案する．VAST 木では FAST と同様に，2 次記憶装置を考慮しない on-memory の索引構造を前提としている．VAST 木は分岐ノードに SIMD 命令でのデータ並列度を改善する不可逆な圧縮を，葉ノードに復元処理の際の分岐処理（if-then 命令）の除去やキャッシュを考慮したデータ配置による CPU 最適化を行った可逆圧縮をそれぞれ適用する．この設計により索引全体のデータ削減を行いながら同時に実行効率の改善を実現する．分岐ノードを複数の領域に分割して，各領域ごとに異なるパラメータの不可逆圧縮手法を適用する．木構造における分岐ノードは上部に比べて下部が大きくなるため，下部に位置する領域ほど圧縮率が高くなるように設計することで，分岐ノード全体のサイズが小さくなるように構成を行う．また圧縮後の比較キーの bit 長を 8 や 16 にあわせることで SIMD 命令で同時比較が可能なデータ並列数を改善する．上記の不可逆な圧縮手法はデータ削減と実行効率改善を同時に実現する一方，圧縮前後で正しい順序関係を保持しない場合がある．具体的には圧縮する前に大小関係のある値を圧縮後に同値と判定する誤った比較（不正な比較処理）が発生する．結果的にこの不正な比較処理で，本来探索されるべき正しい探索位置とは異なる誤った探索位置（探索のズレ）が返される．この不正な比較処理によって発生する正しい探索位置からの探索のズレの量を Δe と定義する．VAST 木では分岐ノードの処理後に得られる葉ノード上の位置から順読み込みを行い，探索のズレの訂正処理に行う．

本研究の貢献は以下である．

- 大規模なデータに対して空間コストが低く，探索における比較処理の並列数を改善した VAST 木を提案する．本手法は木構造の各ノードに適した圧縮手法を適用することで索引サイズを削減しながら，同時にデータ並列度の高い SIMD 比較処理やアライメントの調整によって実行効率も改善する．
- 人工データと Twitter Public Timeline（2010 年 5 月～2011 年 4 月）の実データを用いて，VAST 木と既存

*1　Translation Look-aside Buffer の略，よく参照する論理ページと物理ページの対応関係をキャッシュする CPU 内の機構である．

手法との圧縮率や性能比較を行った．データ数が 2^{30} の VAST 木の分岐ノードのサイズは既存手法に対して 95%以上削減，また索引全体のサイズで 47〜84%削減した．データ数が 2^{28} の探索では，スループット性能は 2 分木に対して 6 倍，既存手法で最も効率的な FAST に対して 1.24 倍の性能向上を確認した．

- CPU コア数の増加に対して線形的に性能改善を行うために重要なメモリ帯域の消費量を，データ数が 2^{28} の探索処理において 2 分木と FAST に対してそれぞれ 94.7%，40.5%だけ削減した．
- VAST 木で発生する探索のズレをモデル化することで Δe の分布を分析する．またこのモデルを活用することで最悪値での訂正処理を 7.1〜18.5%改善した．

次の 2 章では FAST で用いられている SIMD 命令を活用した分岐ノードの比較処理を概説する．3 章で VAST 木の各ノードに適用する圧縮手法，分岐ノードの不可逆圧縮を活用した SIMD 命令によるデータ並列度の改善，Δe の訂正処理など設計に関する詳述を行い，擬似コードをあわせて示す．4 章では VAST 木のプロトタイプを用いた評価結果を示し，5 章で探索中に発生する Δe のモデル化や探索で発生するメモリ帯域の消費量変化を考察する．最後の 6 章と 7 章で関連研究と結論を述べる．

2. 前提知識：SIMD 命令を用いた木構造探索

FAST における SIMD 命令を用いた木構造探索の概略を図 2 に示す．まず索引内の比較キーを図中の三角形で示す部分木にまとめる．この部分木を SIMD ブロックと呼び，1 つの SIMD 命令で同時に処理される最小単位である．FAST では 128 bit の SIMD 用レジスタ[*2]と 32 bit のキーを前提に比較処理の同時実行を提案している．探索中の分岐処理を取り除き CPU 実行効率を高めるために，SIMD 命令の比較結果と移動オフセットの対応表を用いて分岐処理を積和演算に置き換えて制御依存をデータ依存に変換する．

図 2 は以下のように，幅優先順でメモリ上に配置される．

$$\underline{[34, 78, 91]}, [2, 11, 23], [35, 39, 49], \underline{[80, 87, 88]}, \ldots$$

括弧の整数列が SIMD ブロックを表し，下部線が図 2 中の太線の SIMD ブロックと対応している．‘79’ を探索対象の入力クエリとした場合，まず初めに ‘79’ を SIMD レジスタ A に（79, 79, 79, *）としてロードをする（‘*’ は任意の値を表している）．次に一番左の SIMD ブロックを SIMD レジスタ B に（34, 78, 91, *）としてロードを行い，SIMD レジスタ A と B の比較処理を行うことで比較結果（1, 1, 0, *）を SIMD レジスタ C に出力する．SIMD レジスタ A 内の値が大きければ 1 を，そうでなければ 0 を返す．その

*2 128 bit のデータ処理用の SIMD 命令として Intel の CPU では SSE 命令が，AMD の CPU では 3DNOW!が実装されている．

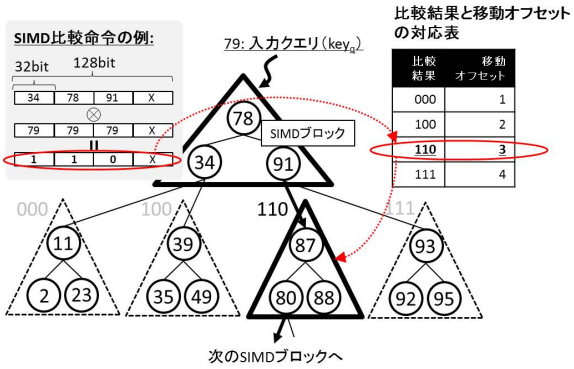

図 2　SIMD 命令による木構造探索の例．レジスタに SIMD ブロック内の比較キー列を読み込み，同時比較を行う．事前計算した対応表から，次に比較するべき SIMD ブロックへ移動する

Fig. 2　An example of tree traversal with SIMD instructions. This instance shows that 128 bit SIMD registers can load four 32 bit integers, and three keys can be compared simultaneously.

後，比較結果と移動オフセット対応表を用いて次に処理するべき SIMD ブロックへ移動する．各 SIMD ブロックは 12B（3 個の 32 bit 整数の比較キー）で，図から比較結果（1, 1, 0, *）に対応した値は 3 である．結果的に処理した SIMD ブロックの 12B と，比較結果から算出したオフセット 36B を加算して処理中の位置から 48B（12B + 12B × 3）だけ位置をずらせばよいことが分かる．

FAST では葉ノードまでの探索に必要な読み込みキャッシュライン数を削減するために，図 2 で説明した 2 段の SIMD ブロックがキャッシュラインに収まるようにブロック化する．SIMD ブロック内の比較キーの数を 3，SIMD ブロックを 2 段とした場合に比較キー集合の総サイズが（SIMD ブロックが 12B で，5 つの SIMD ブロックがあるため）60B となり，結果として一般的な CPU のキャッシュラインサイズ（64B）に収まる．この構成により，2 段の SIMD ブロックの探索に必要な読み込みキャッシュライン数は必ず 1 となる．また上記と同様に，複数のキャッシュラインブロック集合が 1 つのページに収まるようにブロック化している．しかしこのブロック化により，1 回の SIMD 命令による比較キー数が 3 に制限されている点と，ブロック化の際に発生するパディング（2 段の SIMD ブロックを 64B のキャッシュラインに収めた場合の差分 4B）の影響による索引サイズの肥大化が課題となる．

3. 提案手法：VAST 木

3.1 データ構造設計の概要

VAST 木では分岐ノードと葉ノードにそれぞれ異なる圧縮手法を適用することで，索引全体のデータ削減を行いながら同時に実行効率の改善を実現する．分岐ノードには SIMD 命令でのデータ並列度を改善する不可逆な圧縮を，葉ノードには復元処理の際の分岐処理の除去やキャッシュ

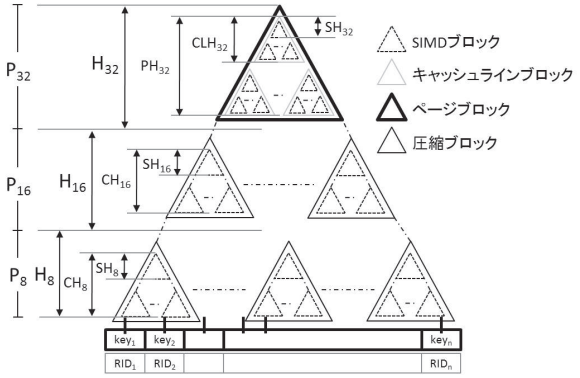

図 3　VAST 木の概要．上部 P_{32} は FAST [12], [13] と同様の構造を適用（非圧縮部）して，中部 P_{16} と下部 P_8 には不可逆な圧縮手法を適用する（圧縮部）．葉ノードに対しては可逆の圧縮手法を適用することで木構造全体のデータサイズ削減を図る

Fig. 3　An overview of VAST-Tree. The top layer (P_{32}) of VAST-Tree uses FAST techniques, and VAST-Tree compresses the middle and the bottom layers (P_{16} and P_8) of the trees. The heights of these layers are H_{32}, H_{16} and H_8, respectively. Moreover, keys in leaf nodes (key1, key2, ..., keyN) are compressed by using lossless compression.

を考慮したデータ配置で実行効率を改善した可逆な圧縮をそれぞれ適用する．分岐ノード内の比較キーは 32 bit か 64 bit であることを想定して，これ以降は 32 bit を前提に説明を行う．**図 3** に VAST 木の全体構造を示す．図中の H_{nbit}，SH_{nbit}，CLH_{nbit}，CH_{nbit} はそれぞれの領域の高さを表しており，$nbit$ はそれぞれの領域に含まれる比較キーの bit 長を示す．木の高さは 2 分木相当であり，たとえば図 2 の SIMD ブロックの高さは 2 である．木構造の分岐ノードは上部に比べて下部が大きいため，下部に位置する領域ほど圧縮率が高くなるように設計する．分岐ノードと葉ノードの設計をそれぞれ 3.1.1 項と 3.1.2 項で概説する．

3.1.1　分岐ノードの設計概要

FAST と同様に VAST 木も図 3 中の SIMD ブロックを SIMD 命令で同時比較する．SIMD ブロック内の比較キーは下部の領域ほど bit 長が小さいことから，データ並列数が 3 に限られている FAST とは異なり SIMD 命令で同時比較可能な数を改善している．VAST 木は具体的に以下 3 つの領域[*3]で構成される．

- P_{32}，FAST と同様の構造を適用，$(2^{SH_{32}}-1)$ 個の比較キーを同時処理．
- P_{16}，32 bit の比較キーを 16 bit に圧縮し，$(2^{SH_{16}}-1)$ 個の比較キーを同時処理．
- P_8，32 bit の比較キーを 8 bit に圧縮し，$(2^{SH_8}-1)$ 個の比較キーを同時処理．

P_{16} と P_8 領域の比較キーに “prefix truncation” と “suffix truncation” [2] を適用する．上位 bit の連続する 0 と，

*3　64 bit の場合は $P_{64}/P_{32}/P_{16}/P_8$ の 4 つの領域で構成する．

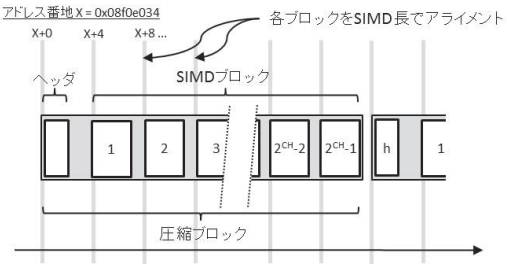

図 4　圧縮ブロックと SIMD ブロックのアライメントの再構成

Fig. 4　Alignment of compression blocks and SIMD blocks.

下位 bit を除去することで非可逆な圧縮を行い，0 ではない有意な上位 bit のみを新たな比較キーとする．これによりハッシュ手法などを用いて比較キーを圧縮する方法とは異なり，上位 bit を用いたキーの比較を行えるようにしている．しかし下位 bit を除去しているため，圧縮前後で正しい順序関係が保持できず探索中に不正な（圧縮前に大小関係のある値を，圧縮後に同値と誤って判定する）比較処理が行われる場合がある．この不正な比較処理によって発生する正しい探索位置からの探索のズレの量を Δe と定義する．VAST 木では，正しい探索位置からのズレ Δe を探索処理後に葉ノードを順読み込みすることで訂正を行う．比較キーの非可逆圧縮に関しては 3.2.1 項で，不正な比較処理と正しい探索位置からのズレ Δe の具体例と訂正処理を 3.2.3 項で説明する．

図 3 中の各ブロックのキャッシュラインやページに対するアライメントの調整は，性能改善のために重要であると先行研究で指摘されている [12]．P_{32} の領域は FAST に倣い，キャッシュラインブロック内の SIMD ブロックをキャッシュラインの先頭から連続位置に配置し，末尾の SIMD ブロックサイズ未満の領域はパディングで埋める．ページブロック内の領域に関しても同様の再構成を行う．FAST では索引全体に対して上記のアライメントの調整を行うが，この再構成によって発生するパディング領域の影響が索引の下部になるほど高くなるため索引サイズが肥大化する．そこで VAST 木では再構成と索引サイズのトレードオフに着眼して，P_{16} と P_8 の領域では**図 4** に示すように各ブロックを SIMD 長のみでアライメントすることでパディングの影響を最小化するように設計する．図中の圧縮ブロックのヘッダに関しては 3.2.1 項で説明を行う．

3.1.2　葉ノードの設計概要

葉ノード上のデータ（$key_1, key_2, \ldots, key_n$）は昇順の列と見なすことができるため，VAST 木では P4Delta [26] を活用する．葉ノードの探索処理においては任意位置のデータを高速に参照する必要があるため，k 個ずつのデータ列を 1 つのチャンクとして圧縮を行うことで任意位置の要素の復元を行いやすくしている．葉ノードの圧縮方法は 3.3 節で説明を行う．圧縮により単体キャッシュラインにより多くの比較キーを格納できることから，探索のズレの訂正

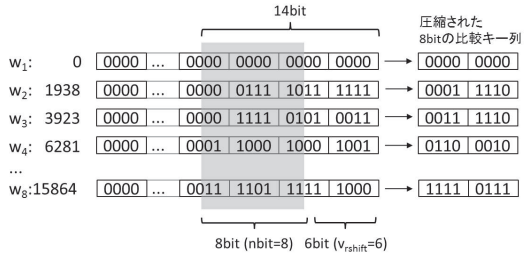

図 5　P_8 内の 8 つの値（$w_1, w_2, \ldots, w_8$）における不可逆圧縮の例
Fig. 5　An example of the lossy compression on the eight values ($w_1, w_2, \ldots, w_8$) of P_8.

処理において参照が必要なキャッシュライン数を少なくすることが可能である．

3.2　分岐ノード構造における再構成

3.2.1　比較キーの不可逆圧縮

VAST 木では B+の分岐ノードの圧縮に用いられる “prefix truncation” と “suffix truncation” を適用する．これらは比較キー間で共通した先頭と末尾の bit 列を除去することで圧縮を行う既存手法である．ここで $v_1, v_2, \ldots, v_m$（V）を圧縮ブロック内に含まれる昇順の比較キー列とする．比較キーの圧縮は以下のように行う．

(1) V の各値を V 中の最小値（v_{min}，昇順であるため $v_{min} = v_1$ となる）と減算することで W を算出する．具体的には $W = 0, v_2 - v_{min}, v_3 - v_{min}, \ldots, v_m - v_{min}$ となる．

(2) W の最大値（w_m）の左端の ‘1’ から $nbit$ 分の範囲に対応した W 内の各値の bit 列を取り出して，圧縮後の比較キー列とする．$nbit$ の値は $\mathrm{P}_{16} = 16$，$\mathrm{P}_8 = 8$ である．

図 5 に P_8 の領域での比較キー圧縮の例を示す．v_{min} と図中の v_{rshift} の値は圧縮ブロック内探索の際に利用するため，圧縮ブロックのヘッダに記録（図 4）する．圧縮ブロック内の探索に関しては次の 3.2.2 項で説明を行う．

$nbit$ の値が高いほど圧縮率が高くなるが，一方で圧縮前後で順序関係を保持しない比較キーの出現確率が高くなる．圧縮前後で順序関係を保持しない具体的な例は 3.2.3 項で説明する．

3.2.2　圧縮ブロック内の探索処理

VAST 木における探索処理は圧縮ブロック内の SIMD ブロックを比較処理しながら，探索対象のキーを含む葉ノードまでのたどるべき圧縮ブロックを算出する．この項では圧縮ブロック内の探索処理を，続く 3.2.3 項では探索のズレの訂正処理をそれぞれ説明する．圧縮ブロック内の比較キー列は不可逆な圧縮手法で変換されているため，そのままでは探索対象の入力クエリ key_q と比較することができない．そこで key_q を処理中の圧縮ブロック内の比較キーと同様の変換を行うことで，比較処理を行う．具体的な変換処理は以下のように行う．

(1) v_{min} と v_{rshift} を圧縮ブロックのヘッダから取り出す．

(2) key_q から v_{min} の値を減算して v_{rshift} だけ右シフトすることで変換後の入力クエリ key'_q を取得する．

上記のように変換することで取得した key'_q と処理中の SIMD ブロックを 2 章で説明した SIMD 命令による探索処理に従った方法で比較を行う．3.4.2 項で P_8 の領域における探索処理を行う擬似コードを示す．

3.2.3　Δe の訂正処理

まず 3.2.1 項の比較キーの不可逆圧縮によって変換前後で順序関係を保持しない具体例を示す．P_8 の領域において比較キー “3220（2 進表記で <u>1100 1001</u> 0100）” が “201（1100 1001）” に変換された場合を考える．ここで探索対象の入力クエリが “3219（2 進表記で <u>1100 1001</u> 0101）” である場合，変換後の値が “201” となる．このとき圧縮前に大小関係のある値を圧縮後に同値と判定する誤った比較（不正な比較処理）が発生する．葉ノード上の正しい探索位置を pos_q，不正な比較処理によって発生する pos_q からのズレの量を Δe と定義する．つまり VAST 木の分岐ノード内の比較処理を完了した直後の探索位置は $key_{pos_q+\Delta e}$ で表す．そのため訂正処理は Δe を 0 にして，目的のキーである key_{pos_q} を取得することである．3.4.3 項で訂正処理の擬似コードを示す．

3.2.4　Δe の訂正処理を活用した索引サイズの削減

前の 3.2.3 項の訂正処理の機構を活用することで，VAST 木の分岐ノードをさらに圧縮する．具体的には，分岐ノードにおける最下部の SIMD ブロック（図 3 の P_8 領域の最下端）をすべて取り除く．この再構成により，すべての探索処理に対して最大 2^{SH_8-1} のズレが発生する．しかし以下の理由から探索性能への影響は小さいことが予想される．

- SH_8 の値が小さければ，訂正処理は単体キャッシュライン内の処理で完結できる可能性が高い．
- また葉ノードは圧縮されているため，単体キャッシュラインに含まれるデータ数は多い．

予備実験から性能劣化に対する圧縮効果が高いことが判明したため，次の 4 章の評価実験ではすべてのパターンに対して上記を適用する．

3.3　葉ノード構造における再構成

P4Delta [26] を葉ノード上の昇順データ（$key_1, key_2, \ldots, key_n$）に対して適用する方法を説明する．P4Delta は差分値を圧縮する方法で，差分値 d は $d[1] = key[1]$，$d[i] = key[i] - key[i-1]$（$1 < i \leq n$）と定義する．この手法は圧縮する値を “coded” と “exceptions” の 2 つに分類する．大半の値（e.g.，90%以上）が表現可能な bit 長 b を定め，b-bit で格納する値が “coded” である．一方 2^b 以上の値は “exceptions” として非圧縮で格納する．

図 6 に葉ノードの圧縮の概要図を示す．3.1.3 項の葉ノー

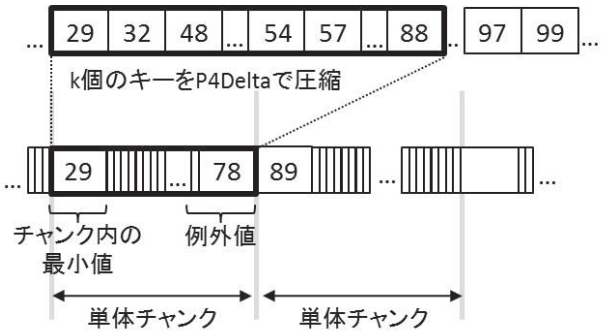

図 6　葉ノードにおける可逆圧縮の概要．上部が圧縮前のキー列で，下部が圧縮後を表す．任意位置のデータを参照できるように，k 個ずつの列を 1 つのチャンクとして圧縮する

Fig. 6　Compression of keys in leaf nodes. Upper array is a uncompressed list of keys, and lower array is a compressed one. VAST-Tree compresses k consecutive keys into a single chunk.

ドの設計概要で説明したように任意位置のデータを高速に参照するため，図で示すように k 個ずつの列を 1 つのチャンクとして圧縮を行うことで任意位置の要素の復元を行いやすくしている．それぞれのチャンクは内部の最小値（昇順であるため先頭の値）と k 個の圧縮されたキー（“coded” と “exceptions”）が含まれる．最小値はチャンクの先頭に格納することで，訂正処理の際に不必要な復元処理を行わないようにしている．訂正処理中のチャンクの扱いに関しては，3.4.3 項の訂正処理の擬似コードで説明を行う．

チャンクを適切ににアライメントすることで，訂正処理中で必要なキャッシュライン数を改善することが可能になる．k は 2 分法（Bisection Method）を用いて決定する．チャンクのアライメントのサイズとキーの bit 長をそれぞれ $nbit_{algn}$，$nbit_{key}$ として，k の決定は以下のように行う．

(1) k_{left} と k_{right} の初期値をそれぞれ $\lceil nbit_{algn}/nbit_{key} \rceil$ と $nbit_{algn}$ とする．
(2) $\lceil (k_{left} + k_{right})/2 \rceil$（$k_{middle}$）個のデータを圧縮した場合に，キャッシュラインに収まるかを確認する．
(3) もし収まれば k_{right} に k_{middle} を，そうでなければ k_{left} に k_{middle} を設定する．
(4) k_{left} と k_{right} が等しくなるまで (2) と (3) を繰り返す．
(5) k_{left}（もしくは k_{right}）の値を k とする．

最小値を $\lceil nbit_{algn}/nbit_{key} \rceil$ として圧縮前後でサイズが変わらない場合を，最大値を $nbit_{algn}$ としてキーの平均 bit 長が 1 になる場合を想定して k の有効範囲を設定する．

3.4　VAST 木の疑似コード

VAST 木を構築する擬似コードを Algorithm 1 に，分岐ノード（P_8）を探索する擬似コードを Algorithm 2 に，Δe を訂正処理する擬似コードを Algorithm 3 にそれぞれ示す．

Algorithm 1 VAST 木を構築するための疑似コード

```
1: /*
2:  * keys[]: 索引対象のキー列
3:  * height: 変換済みの索引の高さ
4:  * nbit: 比較キーの bit 長
5:  * H_nbit: P_nbit における非圧縮部/圧縮部の高さ
6:  * CH_nbit: P_nbit における圧縮ブロックの高さ
7:  * SH_nbit: P_nbit における SIMD ブロックの高さ
8:  * bytes_output: 構築された VAST 木
9:  */
10: height = 0
11: // keys から P_32 領域の FAST を構築
12: build_fast(keys, H_32, bytes_output)
13: height = H_32
14: for nbit in {16, 8} do
15:   for i ← 1 to H_nbit/CH_nbit do
16:     for j ← 1 to 2^height do
17:       // P_nbit における (i, j) 位置の部分木を構成する
18:       // 2^CH_nbit 個の比較キーを抽出
19:       keys_ext[] = extract_cb_keys(keys, CH_nbit, i, j)
20:       // 取り出した比較キー列から圧縮ブロックを構築
21:       keys_cmp[] = lossy_compress(nbit, keys_ext[])
22:       v_min = extract_v_min(keys_ext[])
23:       v_rshift = extract_v_rshift(keys_ext[])
24:       build_cb(keys_cmp[], v_min, v_rshift, SH_nbit, bytes_out)
25:     end for
26:     height = height + CH_nbit
27:   end for
28: end for
```

3.4.1　VAST 木の構築

VAST 木の構築（Algorithm 1）は，まず P_{32} の領域における比較キーの抽出と FAST のデータ構造の構築を行う（12 行目）．VAST 木における P_{16} と P_8 の領域の構築は 15〜27 行目における内外の 2 つのループで構成される．外部ループは H_{nbit}/CH_{nbit} 回実行され，内部ループは処理中の高さにおける圧縮ブロック数（2^{height}）回実行される．1 回の内部のループ内処理（17〜24 行目）で圧縮ブロックを 1 つ構築する．処理している圧縮ブロックに含まれる比較キーを索引対象のキー列 $keys[]$ から抽出（19 行目）し，圧縮ブロックを構築する（21〜24 行目）．

3.4.2　分岐ノードの探索

P_8 の領域の探索（Algorithm 2）は 10〜29 行目における H_8/CH_8 回実行される外部ループと，21〜26 行目における CH_8/SH_8 回実行される内部ループで構成される．1 回の内部のループ内処理で SIMD ブロックを，1 回の外部ループ内処理で圧縮ブロックをそれぞれ 1 つ探索する．圧縮ブロックの探索ではまずヘッダから最小値 v_{min} と右シフト値 v_{rshift} を取り出し，探索対象の入力クエリ key_q に対して内部の比較キーと同様の変換を行う（13〜15 行目）．次の内部ループ処理での SIMD ブロック内の探索は，変換後の入力クエリ key'_q と処理中の SIMD ブロックを比較処理して，次に処理するべきブロックの位置を算出する（23〜25 行目）．木の高さと累積データサイズの対応表 $size_{tree}[]$

と，比較結果と移動オフセットの対応表 $lookup[]$ を用いることで，分岐処理を用いず SIMD 比較命令の結果から積和演算のみで木構造の探索が可能になる．外部ループ処理が

Algorithm 2 P_8 探索のための疑似コード

```
1: /*
2:  * key_q: 探索対象の入力クエリ
3:  * pos_c: 処理中の圧縮ブロックを示す位置情報
4:  * pos_s: 処理中の SIMD ブロックを示す位置情報
5:  * lookup[]: 比較結果と移動オフセットの対応表（図 2）
6:  * size_cb8: P_8 における単体圧縮ブロックのサイズ
7:  * size_sb8: P_8 における単体 SIMD ブロックのサイズ
8:  * size_tree[]: 木の高さ（0〜31）と累積データサイズの対応表
9:  */
10: for i ← 1 to H_8/CH_8 do
11:     // 処理中の圧縮ブロック内の比較キーと
12:     // 同様の変換を key_q に適用
13:     v_min = get_v_min(pos_c)
14:     v_rshift = get_v_rshift(pos_c)
15:     key'_q = (key_q - v_min) >> v_rshift
16:     // 圧縮ブロック内の SIMD ブロック内の比較キーと
17:     // key'_q を比較することで，次に処理するべき
18:     // 圧縮ブロックの位置（pos_c）を算出
19:     pos_s = 0
20:     cur_pos_c = pos_c
21:     for j ← 1 to CH_8/SH_8 do
22:         // pos_c の 8 つの比較キーと key'_q をベクトル命令で比較
23:         res = compare_simd8(key'_q, pos_c)
24:         pos_s = pos_s × 2^(SH_8) + lookup[res]
25:         pos_c = cur_pos_c + pos_s × size_sb + size_tree[SH_8 + j]
26:     end for
27:     pos_c = cpos × 2^(CH_8) + spos
28:     pos_c = cpos × size_cb + size_tree[H_32 + H_16 + i]
29: end for
30: // Δe を含んだ探索結果の位置情報を返す
31: return pos_c
```

Algorithm 3 Δe 訂正処理の疑似コード

```
1: /*
2:  * key_q: 探索対象の入力クエリ
3:  * pos_q: Algorithm 2 の結果（Δe を含んだ位置情報）
4:  * k: 単体チャンク内のキー数
5:  */
6: loop
7:     // 各チャンクの先頭値（チャンク内の最小値）を取得
8:     v_min = get_v_min(⌈pos_q/k⌉)
9:     pos_q = pos_q - k
10:    if v_min < key_q then
11:        break
12:    end if
13: end loop
14: keys_chunk[] = decompress_p4delta(⌈pos_q/k⌉)
15: for i ← 1 to k do
16:    if key_q == keys_chunk[i] then
17:        // 探索結果の位置情報（pos_q - Δe）を返す
18:        return pos_q + i
19:    end if
20: end for
21: // 未発見のエラーを返す
22: return -1
```

すべて完了した際の pos_c の値が，分岐ノード探索直後の Δe を含んだ葉ノード上の位置である．

3.4.3　Δe の訂正処理

訂正処理（Algorithm 3）では，6〜13 行目で正しい結果を含んだチャンクを探索して，14〜22 行目で探索されたチャンクを復元して正しい位置情報を返す．前半の処理では，チャンク先頭に含まれた値を抽出することで，その値が入力値 key_q より小さくなるチャンクを探索する．探索されたチャンクを復元（14 行目）して，入力クエリに対応したデータが含まれれば正しい位置情報（16〜19 行目）を，そうでなければエラー（22 行目）を返す．

4.　評価実験

評価実験では人工データ（‘*Synthetic*’）と実データの両方を用いることで異なるキー分布での評価を行う．人工データでは $1/\lambda$ のパラメータで決められるポアソン分布に従うデータを用いた．特に明示しない場合には $1/\lambda$ を 16 に設定して評価を行った．一方実データには Twitter Public Timeline（2010 年 5 月〜2011 年 4 月）の 36,068,948 件（約 2^{25}）の tweet を用いて，内部に含まれる IDs と $Timestamps$ の属性を抽出して使用した．IDs はユーザを示す識別子を，$Timestamps$ は tweet を投稿した時刻をそれぞれ表す整数値である．**図 7** に実データの差分値 d の分布を示す．図から明らかにように，大半のデータが重複するような偏ったデータであることが分かる．

VAST 木の索引構造における各変数（SH_{32}，CLH_{32}，PH_{32}，SH_{16}，CH_{16}，SH_8，CH_8）は評価環境がキャッシュラインサイズ 64B，ページサイズ 4 KiB（kernel-2.6.18-194 の CentOS v5.5），SIMD レジスタ 128 bit であることを前提に 3.1 節で説明したアライメントの調整を行うために（2，4，8，3，7，4，8）と設定した．本評価は Xeon5670（物理 6 コアで最大メモリ帯域 31.8 GiB/s）のサーバを用いて実施した．Xeon5670 は 32 KiB の L1 キャッシュ，256 KiB の L2 キャッシュ，12 MiB の LLC（Last Level Cache）を

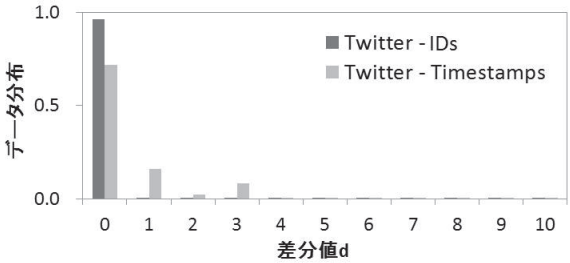

図 7　Twitter Public Timeline（2010 年 5 月〜2011 年 4 月）の tweet 情報の IDs と $Timestamps$ の差分値 d の分布

Fig. 7　Distributions of the first 10 d-gaps for realistic data used in our experiments. These data were collected from May, 2010 to Apr., 2011 from the Public Timeline of Twitter.

搭載している．CPU の実行時間内訳は oprofile v0.9.6 を用いて取得を行った．評価実施時に Xeon5670 上で oprofile をサポートしていなかったため，実行時間内訳の評価のみ Xeon5260（物理 2 コアで最大メモリ帯域 21.2 GiB/s）を用いた．評価用コードは C++ で記述し，gcc v4.1.2 の “-O3” でコンパイルして評価を行った．

4.1 VAST 木の評価

4.1.1 索引構造の圧縮性能

表 2 に索引の分岐ノードのサイズ比較結果を，**表 3** に VAST 木の葉ノードの圧縮率をそれぞれ示す．分岐ノードのサイズはデータ分布に依存しないため，人工データのみを使用した．表 2 から分かるとおり，すべての条件において VAST 木の圧縮率は高く，データ数が 2^{30} で VAST 木は他の手法と比べて 95%以上の削減を実現している[*4]．この分岐ノードの高い圧縮率は 3.1.1 項で説明した “prefix truncation” と “suffix truncation” による比較キー圧縮と SIMD ブロックによるアライメント調整によるパディング除去に加えて，3.2.4 項で説明した最下部の SIMD ブロックの除去により実現している．VAST 木の変数（$H_{32}/H_{16}/H_8$）の値は索引サイズと 3.2.3 項で説明した Δe とのトレードオフを考慮して設定する必要がある．予備実験の結果からデータ数が 2^{24} のときは $H_{32} = 8$ と $H_{16} = 6$ が，そうでない場合は $H_{32} = 8$ と $H_{16} = 12$ が Δe の値が低く探索性能が高い結果となったため，以降の評価実験ではこれらの値を使用した．

表 3 に VAST 木の葉ノードの圧縮率とチャンク内に含まれる比較キー数 k を示す．葉ノードの圧縮率はデータ分布に依存するため，人工データと実データの両方を用いて評価を行った．偏った分布（図 7）の場合，またチャンクのアライメントのサイズを大きくした場合に圧縮率が改善することが分かる．表 2 の結果とあわせて，VAST 木ではデータ数が 2^{30} の場合に索引全体のサイズを 47〜84%削減している．チャンクをキャッシュラインにアライメントした場合に探索性能が最も高くなったため，以降の評価実験ではこの値を使用した．

索引構築時間に関して Kim らによる先行研究 [12] では 64 M 個のデータに対する索引構築時間が 0.1 秒以下とあるが，VAST 木では分岐ノード内の比較キー圧縮や，葉ノード圧縮の 2 分法を用いた最適化により同数の人工データに対する索引構築で 100x 以上の構築時間を要する．そのため FAST が想定している短い構築時間を活用した索引データ更新のための再構築を行うことができず，データの更新が少なく参照量が非常に高い場合にのみ VAST 木の適用は向いている．

4.1.2 VAST 木の性能評価

図 8 に索引対象のデータ数が 2^{25} における完全一致探索の実行時間内訳を示す．2 分木と FAST は性能に対するデータ分布の影響がないことから，データ数が 2^{25} の人工データ（Synthetic）を評価に用いた．VAST 木は葉ノード圧縮の有無で 2 パターンの評価を行った．図から VAST 木のすべてのパターンで 2 分木に対してストール時間と分岐ペナルティが 72.8%と 50%程度まで改善され，結果として IPC が改善されていることが分かる．葉ノードを圧縮しない VAST 木（“VAST-Tree w/o P4Delta”）では，3.3.1 項で説明した分岐ノード内の比較処理のデータ並列数改善とアライメントの再構成によって，実データを用いた評価で IPC が FAST と比較して改善されている．人工データを用いた評価では FAST に対して IPC が低くなっているが，これは Δe の訂正処理による読み込みキャッシュライン数増大の影響だと考えられる．葉ノードを圧縮した VAST 木（“VAST-Tree w P4Delta”）ではすべてのパターンにおいて Δe の訂正処理の改善，CPU 実行効率の高い P4Delta の復元処理によって IPC の値が改善されていると考えられる．

図 9 に人工データを，**表 4** に実データをそれぞれ用いたスループット性能の評価結果を示す．葉ノードの圧縮をした VAST 木（“VAST-Tree w P4Delta”）は索引サイズの観点で利点はあるが，データ数が 2^{24} と 2^{28} の条件で IPC が高いにもかかわらず FAST に対して性能が劣ることが分かる．これは Δe の訂正処理改善のための P4Delta の適用において，復元処理に必要な実行命令数が増大（図 8）

表 2 索引の分岐ノードサイズ（GiB）比較．VAST 木の括弧内の値は H_{32} と H_{16} をそれぞれ表している

Table 2 Total branch node sizes (GiB) of VAST-Tree, binary trees, and FAST. The values in parentheses show H_{32} and H_{16}. The branch nodes of VAST-Tree are obviously highly compressed by using the lossy compression.

データ数	2^{24}	2^{26}	2^{28}	2^{30}
VAST 木 (0, 6)	0.00449	0.00449	0.130	0.130
VAST 木 (8, 0)	0.00225	0.0186	0.0186	0.519
VAST 木 (8, 6)	0.00449	0.00449	0.130	0.130
VAST 木 (8, 12)	0.00248	0.00337	0.00337	0.251
FAST	0.252	1.25	1.25	64.3
2 分木	0.156	0.625	2.50	10.0

表 3 VAST 木の葉ノードの圧縮率．括弧内の値は 3.3 節で説明した手法で決められた k の値

Table 3 Compression ratios of leaf nodes (VAST-Tree).

アライメントサイズ	64B	128B	256B
$1/\lambda = 16$	.142(113)	.133(240)	.129(497)
$1/\lambda = 64$	.225(71)	.219(151)	.206(311)
IDs	.219(71)	.211(151)	.199(321)
Timestamps	.500(32)	.320(100)	.285(311)

*4 表 2 の値は近似しているため $H_{32} = 0$ と $H_{32} = 6$ のパターンで圧縮率が同値になっているが実際は多少異なる．索引の上部（H_{32}）は下部に対して比重が小さいため，構造変化によるデータサイズの影響が小さく非常に近い値になっている．

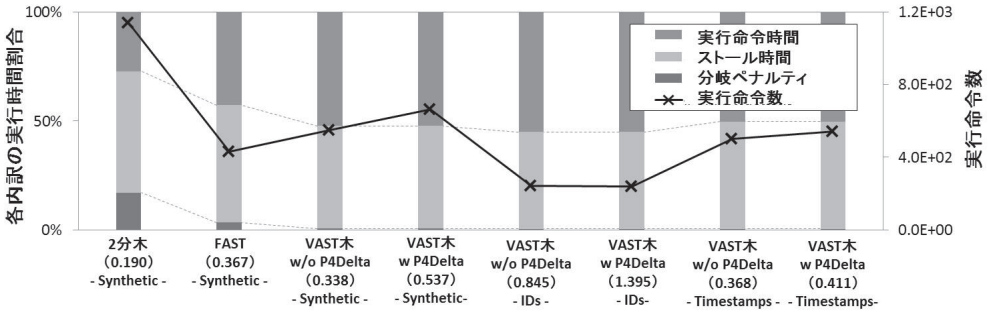

図 8　索引対象のデータ数が 2^{25} における完全一致探索の実行時間内訳（命令実行時間，ストール時間，分岐ペナルティ）比較．括弧内は各実行における IPC を表す

Fig. 8　Ratios of execution time and counts of instructions with the exact-match scans of three techniques: binary trees, FAST, and VAST-Tree. The total number of keys is 2^{25}, and the values in parenthesis indicate IPC in each technique.

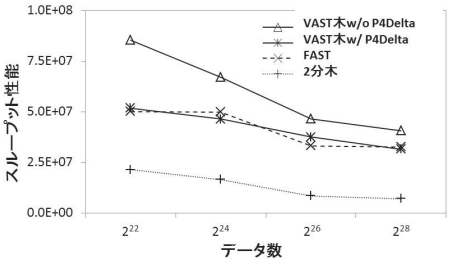

図 9　人工データを用いた完全一致探索のスループット性能比較．圧縮を用いた索引のサイズ削減を実現しながら VAST 木は FAST とほぼ同等か，それ以上の性能を実現している

Fig. 9　Throughput with exact-match scans as determined from synthetic data sets.

表 4　Twitter Public Timeline の tweet 情報を用いた完全一致探索のスループット性能比較（$\times 10^6$）

Table 4　Throughput ($\times 10^6$) with exact-match scans by using realistic data sets.

使用データ	*IDs*	*Timestamps*
2 分木	9.05	左と同値
FAST	34.3	左と同値
VAST 木 w P4Delta	67.7	31.4
VAST 木 w/o P4Delta	78.9	41.1

していることが原因だと考えらえる．一方で，葉ノードの圧縮を行わない VAST 木（“VAST-Tree w/o P4Delta”）は FAST に対して性能改善を実現している．データ数が 2^{28} の探索では，スループット性能は 2 分木に対して 6 倍，FAST に対して 1.24 倍である．表 4 の実データを用いたスループット性能も図 9 とほぼ同様の傾向が得られている．*IDs* のスループット性能が *Timestamps* よりも高い理由は，*IDs* のデータの偏りの高さが影響していると考えられる（図 7）．この項の性能評価の結果により，葉ノードの圧縮の適用は IPC の改善になるがスループット性能の改善にはならないことが判明した．そのため現実的な実装においては VAST 木の索引サイズがサーバのメモリに対し

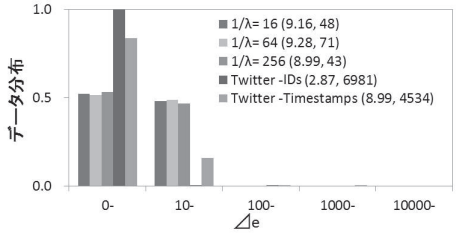

図 10　データ数が 2^{28} の探索処理での Δe 分布．括弧内の左値が平均値を，右値が最悪値をそれぞれ示している．99%以上の探索における Δe が 100 以下であることが分かる

Fig. 10　Distributions of the errors caused by the use of our lossy compression. The evaluation used synthetic data with 2^{28} keys. The left values in parenthesis indicate the averaged amounts of errors, and the right values show the worst errors in each condition.

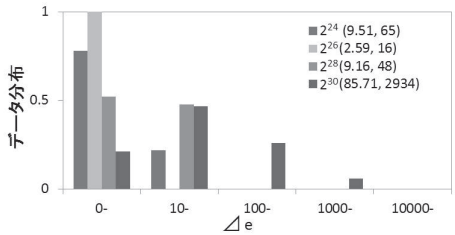

図 11　人工データの総数を変化させた場合の Δe 分布．圧縮率が高く不正な比較が行われる確率の高い H_8 の値が大きくなることで，Δe の平均値・最悪値が高くなっている

Fig. 11　Distributions of the errors for the various numbers of keys in leaf nodes (synthetic data). The error amounts get bigger as the number of keys increases, because the maximum error amounts widen due to the increase of H_8.

て十分余裕がある場合には性能向上の観点で葉ノードの圧縮を行わず，そうではない場合には P4Delta を適用して索引サイズを圧縮する選択が適している．

最後に探索処理中に発生する Δe の分布を**図 10** と**図 11** に示す．それぞれの図中の括弧に Δe の平均値と最悪値を

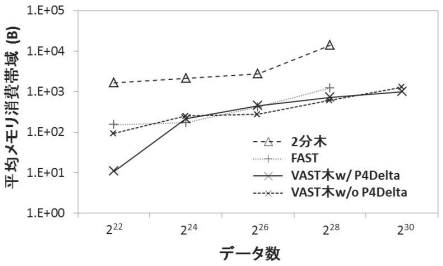

図 12　完全一致探索における平均メモリ帯域の消費量（B）比較．2^{28} 個のデータ探索において 2 分木に対して 94.7%，FAST に対して 40.5%の削減を達成している

Fig. 12　Comparison of averaged memory bandwidth (B) for a single tree search consumed by all the techniques. VAST-Tree achieves the saving of memory bandwidth consumption as compared to binary trees and FAST because of the conditional branch-free and compressed structure.

あわせて示す．Δe が高くなると CPU の命令数やメモリ帯域の消費量に影響がでることから非常に重要な指標であるが，図 10 の結果から 99%以上の探索における Δe が 100 以下であることが分かる．表 3 の結果とあわせて訂正処理に必要なキャッシュラインは 1〜2 程度で探索性能へのペナルティは小さいことが分かる．実データ（図 10）やデータ数を大きくした場合（図 11）に最悪値が高くなるが，平均値が 100 以下程度に抑えられていれば図 9 と表 4 の結果から探索処理の平均性能に関しては影響が小さいと判断できる．

4.1.3　メモリ帯域の消費量評価

先行研究で CPU コア数の増加に対して線形的に性能改善するには，メモリ消費量を抑えることが重要であると指摘されている [12], [16], [20]．**図 12** に完全一致探索における平均メモリ帯域の消費量を示す．2 分木は分岐処理における投機的なメモリ参照の影響でメモリ帯域の消費量が高く，FAST と VAST 木は分岐命令を含まないため 2 分木に対して値が低い．結果的に 2^{28} 個のデータ探索で 2 分木に対して 94.7%，FAST に対して 40.5%の削減を達成している．

5.　考察

前章の評価実験の結果から VAST 木の性能はデータ分布に依存するため，5.1 節でデータ分布と Δe の関係を分析してモデル化を行い，5.2 節では最悪値 Δe（図 10 と図 11）を考慮した訂正処理の改善方法を提案する．

5.1　Δe のモデル化

まず初めに分岐ノード内で発生する不正な比較処理の回数を分析する．**図 13** で SIMD ブロックと葉ノード上のキー列の関係を示す．SIMD ブロック内の比較キーを n_k とした場合，比較キー列は昇順であるため $n_k < n_{k+1}$

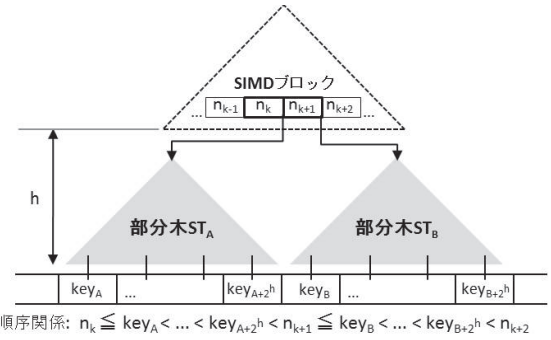

図 13　SIMD ブロックと葉ノード上のキー列の分析．図中の高さ h の ST_A と ST_B の部分木下のキー数は 2^h である

Fig. 13　An analysis of the errors on the ordering of keys under a certain SIMD block. The number of keys under these sub-trees (ST_A and ST_B) is assumed to be 2^h.

の関係がある．SIMD ブロックまでの高さを h とした場合，SIMD ブロック以下には key_A〜key_{A+2^h} に対応した比較キーを持つ部分木 ST_A と key_B〜key_{B+2^h} に対応した比較キーを持つ部分木 ST_A がそれぞれ存在する．ここで $n_k \leq key_q < n_{k+1}$ の順序関係を持つ探索対象の入力クエリを key_q を考える．もし図中の SIMD ブロック内で不正な比較処理が発生して $n_{k+1} \leq skey < n_{k+2}$ と判断されてしまった場合，探索処理は ST_A ではなく ST_B に移動する．ST_B 内の比較キー集合は必ず key_q 以上の値になるため，最終的に探索処理は key_B を返す．この分析により分岐ノード内の不正な比較処理はたかだか 1 回しか発生しないことが分かる．

上記の分析結果から幾何分布を用いて，最上部から連続する複数の正しい比較処理と続く 1 回の不正な比較処理で Δe のモデル化を行う．高さ h における不正な処理が発生する確率と探索のズレ量，木の高さをそれぞれ p_h，de_h，H とする．探索のズレの総量を表す確率変数 E は以下のように計算する．

$$E = \frac{1}{H}\left(\sum_{h=1}^{H} de_h \times p_h \sum_{k=1}^{h} (1-p_{k-1})^{k-1}\right) \tag{1}$$

実際に不正な比較処理は P_{32} ではなく P_{16} と P_8 の領域で発生するため，式 (1) は以下のようになる．

$$E = \frac{1}{H_{16}+H_8} \sum_{k=1}^{\frac{H_{16}}{CH_{16}}+\frac{H_8}{CH_8}} e_k \tag{2}$$

e_k は高さ k における各圧縮ブロックにおける探索のズレ量を表し，以下のように計算する．

$$e_k = \sum_{l=1}^{\frac{CH_{16}}{SH_{16}}+\frac{CH_8}{SH_8}} de_l \times p'_k \sum_{n=1}^{l} \left(1-p'_{n-1}\right)^{n-1} \tag{3}$$

de_l は高さ l における SIMD 比較命令によって発生する探索のズレ量を表し，以下のように計算する．

$$de_l = \begin{cases} 2^{H_{16}+H_8-l\times SH_{16}} & (l \le \frac{H_{16}}{CH_{16}}) \\ 2^{H_8-(l-\frac{H_{16}}{CH_{16}})\times SH_8} & otherwise \end{cases} \quad (4)$$

p'_k は高さ k における圧縮ブロック内における不正な比較処理の発生確率を表す．圧縮ブロック内の値は同じ bit 長で圧縮されているため，不正な比較処理が発生する確率は同じものとして，p'_k を以下のように計算する．

$$p'_k = \frac{c_k \times r_k}{qr}, \quad (5)$$

ここでは qr が葉ノードの範囲（$key_n - key_1$），c_k が高さ k における比較キーの総数，r_k が高さ k の比較キーが圧縮処理で削られた範囲の量をそれぞれ表す．たとえば圧縮処理によって下位 7 bit 削除されていれば r_k は 2^7 となる．さらに c_k は以下のように計算する．

$$c_k = \begin{cases} 2^{H_{32}+(k-1)\times CH_{16}} & (k \le \frac{H_{16}}{CH_{16}}) \\ 2^{H_{32}+H_{16}+(k-\frac{H_{16}}{CH_{16}}-1)\times CH_8} & otherwise \end{cases} \quad (6)$$

r_k は明らかに探索対象のデータ分布に依存する．ここで入力データの差分値 d を表す確率変数を X とする．高さ k の（P_{nbit} の領域に存在する）圧縮ブロックに対応した葉ノード上の比較キー列における d の合計値が 2^{nbit} を超えた場合，r_k が 0 より大きな値になる．そのため d の合計値を Y_k とした場合に r_k は以下のように計算する．

$$r_k = \begin{cases} 2^{log_2 Y_k - 16} & (k \le \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k > 16) \\ 0 & (k \le \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k \le 16) \\ 2^{log_2 Y_k - 8} & (k > \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k > 8) \\ 0 & (k > \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k \le 8), \end{cases} \quad (7)$$

$$Y_k = \begin{cases} \sum_{n=1}^{H_{16}+H_8+(1-k)\times CH_{16}} X_n & (k \le \frac{H_{16}}{CH_{16}}) \\ \sum_{n=1}^{H_8+(1-k-\frac{H_{16}}{CH_{16}})\times CH_8} X_n & otherwise \end{cases} \quad (8)$$

図 14 に上記のモデルを用いた Δe の推定値と実値の比較結果を示す．評価には 4 節と同様に $1/\lambda$ のパラメータで決められるポアソン分布に従う人工データを用いた．λ の値が小さいときと大きいときに多少の乖離はあるが，全体の傾向は推定できていることが分かる．次の 5.2 節では上記の結果を用いた最悪値 Δe の訂正処理の改善方法を説明する．

5.2　訂正処理の改善

訂正処理を順読み込みではなく，Δe の推定値と 2 分探索を用いることで改善する方法を説明する．pos'_q を探索直後の Δe を含んだ葉ノード上の位置で，正しい位置は pos_q とする．**図 15** で示すように pos'_q の位置から pos_q まで $range_{\Delta e}$ の範囲の 2 分探索を行うことで訂正処理を行う．そのため高い確率で pos_q が $range_{\Delta e}$ 内に含まれるように $range_{\Delta e}$ を設定する必要がある．前の 5.1 節の分析から $range_{\Delta e}$ は以下のように計算する．

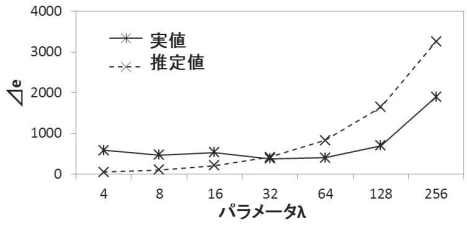

図 14　Δe のモデルによる推定値と実値の比較
Fig. 14　Estimation of the errors incurred by VAST-Tree.

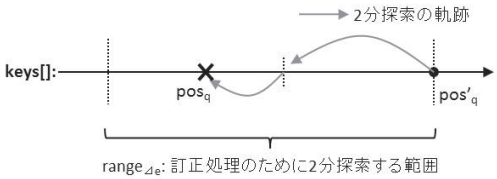

図 15　最悪値 Δe を考慮した 2 分探索による訂正処理の概要
Fig. 15　An overview of our proposed optimization technique for binary searches so as to minimize the penalty of the worst errors.

表 5　最悪値 Δe の平均探索時間比較（CPU 内の $rdtsc$ 値）
Table 5　Hardware-based timestamp counts of a single tree search with the worst error.

訂正手法	順読み込み	改善手法	改善率（%）
IDs	557285	523514	7.1
$Timestamps$	551297	449340	18.5

$$range_{\Delta e} = t \times \sqrt{V(E)} \quad (9)$$

ここで t は変数，$V(E)$ は E の分散をそれぞれ表す．pos_q が $range_{\Delta e}$ 内に含まれる確率は t に対する Chebyshev 不等式を用いて以下のように表される．

$$P\left(|E - Ex(E)| \ge range_{\Delta e}\right) \le \frac{1}{t^2} \quad (10)$$

ここで $Ex(E)$ は E の期待値を表す．上記の結果より pos_q が $range_{\Delta e}$ 内に含まれる確率を考慮して t の値を設定すればよいことが分かる．上記が確率が 99%以上になるように値を設定して以降の実験を行った．**表 5** に実データを用いて最悪値 Δe の場合の平均探索時間比較を示す．CPU 内のハードウェアカウンタである $rdtsc$ を用いて時間の取得を行った．結果的に IDs で 7.1%，$Timestamps$18.5%の時間がそれぞれ節約できていることが分かる．

6.　関連研究

先行研究 [3] においてデータベース処理のボトルネックが CPU/メモリであることが指摘されて以降，データベース技術において広く用いられる圧縮や結合などに代表される重要な処理を近代的なハードウェア上で最適化するための手法が数多く提案されている [5], [7], [8], [10], [19], [20].

近年のハードウェア最適化に関わる研究動向に関しては文献 [15] が詳しい．ここでは特にハードウェア最適化された索引技術と索引構造の圧縮技術の関連研究を概説する．

6.1　ハードウェア最適化された索引技術

T-tree [14] がメモリ上で最適化された索引構造として提案された後，キャッシュ構造を考慮した B+ 木向けの最適化手法が様々提案された [4], [9], [17], [18]．それらの手法の主な着眼点は，キャッシュミスを最小化するための B+ 木の分岐ノードサイズの検討であった．2 つの先行研究 [4], [9] が分岐ノードをキャッシュラインより多少大きく設定することで B+ 木の処理性能を改善できるという同様の指摘を行った．これは分岐ノードをキャッシュラインより小さく設定すると，TLB ミスによるペナルティが顕在化して性能を悪化させることが理由である．また異なる研究アプローチとして，キャッシュミスによる遅延を隠ぺいするための “query buffering” の手法も提案 [25] された．これは同じ分岐ノードを参照したクエリを一定時間保持して同時に処理することで，ノードの参照回数を最小化する手法である．結果的に参照するキャッシュライン数が減り，キャッシュミスによる遅延とメモリ帯域の消費量を改善する．これらは主にキャッシュ構造に着眼した手法であり，本研究で対象にしている分岐除去やアライメントの最適化など他の CPU の実行効率に関わる分析はない．

SIMD 命令を用いた探索処理の効率化手法は 2000 年代後半以降で提案された手法である [12], [13], [21]．その中でも Kim らによる FAST は最も効率的で，キャッシュ構造/アラインメント/分岐除去/データ並列などを用いることで CPU の実行効率を改善している．しかし研究背景で指摘したように，大規模なデータに対する探索処理に関して FAST は，データ構造のアライメント調整による索引サイズ肥大化などの技術的な課題がある．本研究はハードウェア最適化と索引サイズにおけるトレードオフに初めて着眼しており，そのうえで圧縮による索引サイズの削減や SIMD 命令による比較処理のデータ並列数改善を提案した．

より近代的な手法として GPU や x86 互換の Intel® MIC（Many Integrated Core）などに代表されるメニーコア環境における探索処理の改善手法も提案 [11], [22] されており，FAST においてもこれらの環境を用いた評価 [13] が行われている．しかし現在のアーキテクチャ上の制約により，ホスト側のデータを GPU 側に転送する処理が必要になり，その時間が処理全体の 15%から 90%を占めるという先行研究の報告がある [6]．ハードウェアを考慮した最適化の有無で平均で 24x，最大で 53x の性能差が発生するという報告 [1] があり，大規模なデータ処理のおいてはますますこれらの最適化が重要になることが考えられる．

6.2　索引構造の圧縮技術

索引構造の圧縮に関しては，データベース研究の黎明期に提案された “prefix/suffix truncation” や “NULL suppression” [2] が有名であるが，これらの手法は索引サイズ削減のみに着眼した提案である．本提案手法は “prefix/suffix truncation” を分岐ノードに適用することで，索引サイズを削減しながら同時に CPU の実行効率を改善する点が異なる．

葉ノードのデータは昇順の列と見なすことで近代的な CPU に最適化された様々な圧縮手法が適用可能である．様々な既存手法の中から，ここでは特に効率的な P4Delta [26] と VSEncoding [23] を紹介する．3.3 節で説明した P4Delta は主に転置インデックスの要素技術として広く用いられている [24]．一方で VSEncoding は圧縮の際に必要なパラメータを動的計画法で最適化する手法で，圧縮率に優れた手法である．VAST 木の葉ノードにおいては，探索と Δe の訂正処理で任意位置のデータを高速に参照する必要があるため，圧縮率と復元性能を損なわずに上記の要件を満たしやすい P4Delta を採用した．しかし，上記の要件を満たすことができれば VAST 木と葉ノードに適用する圧縮手法は独立しているため今後提案されるより高効率な圧縮手法を容易に適用することが可能である．

7.　結論

本論文では大規模なデータに対して空間コストが低く，CPU の実行効率が高い on-memory の索引構造である VAST 木を提案した．VAST 木は木構造内の分岐ノードに SIMD 命令でのデータ並列数向上を可能とする不可逆な圧縮手法を，葉ノードに CPU に最適化された可逆な圧縮手法をそれぞれ適用することでデータ削減と実行効率改善を同時に実現した．分岐ノードの不可逆圧縮は索引サイズの大幅の削減に，葉ノードの圧縮は VAST 木の探索ズレ Δe の訂正処理の際の CPU 実行効率改善に寄与している．最後に Δe をモデル化することで訂正処理をさらに改善する方法を考察して提案を行った．データ構造における空間コストを抑えつつ，アルゴリズムの CPU 実行効率を高めることは規模が増大傾向にあるデータ処理においては今後も重要な要素技術である．

参考文献

[1] Kim, C. et al.: Closing the Ninja Performance Gap, Intel Labs Technical Report (2013).

[2] Bayer, R. and Unterauer, K.: Prefix B-trees, *ACM Trans. Database Systems*, Vol.2, pp.11–26 (1977).

[3] Boncz, P.A., Manegold, S. and Kersten, M.L.: Database Architecture Optimized for the New Bottleneck: Memory Access, *Proc. VLDB*, pp.54–65 (1999).

[4] Chen, S., Gibbons, P.B. and Mowry, T.C.: Improving index performance through prefetching, *SIGMOD Record*, Vol.30, No.2, pp.235–246 (2001).

[5] Chhugani, J. et al.: Efficient implementation of sorting on multi-core SIMD CPU architecture, *Proc. VLDB*, Vol.1, pp.1313–1324 (2008).

[6] Fang, W. et al.: Database compression on graphics processors, *Proc. VLDB*, pp.670–680 (2010).

[7] Govindaraju, N.K., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management, *Proc. SIGMOD*, pp.325–336 (2006).

[8] Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M. and Manocha, D.: Fast computation of database operations using graphics processors, *Proc. SIGMOD*, pp.215–226 (2004).

[9] Hankins, R.A. and Patel, J.M.: Effect of node size on the performance of cache-conscious B+-trees, *Proc. SIGMETRICS* (2003).

[10] He, B. and Yu, J.X.: High-throughput transaction executions on graphics processors, *Proc. VLDB Endowment*, Vol.4, pp.314–325 (2011).

[11] Kaldewey, T. et al.: Parallel search on video cards, *Proc. HotPar* (2009).

[12] Kim, C. et al.: FAST: Fast architecture sensitive tree search on modern CPUs and GPUs, *Proc. SIGMOD*, pp.339–350 (2010).

[13] Kim, C. et al.: Designing Fast Architecture Sensitive Tree Search on Modern Multi-Core/Many-Core Processors, *ACM Trans. Database Systems*, Vol.9, No.4 (2011).

[14] Lehman, T.J. and Carey, M.J.: A Study of Index Structures for Main Memory Database Management Systems, *Proc. VLDB*, pp.294–303 (1986).

[15] Manegold, S., Kersten, M.L. and Boncz, P.: Database architecture evolution: Mammals flourished long before dinosaurs became extinct, *Proc. VLDB Endowment*, Vol.2, pp.1648–1653 (2009).

[16] Matthew, R.: When Multicore Isn't Enough: Trends and the Future for Multi-Multicore Systems, *HPEC* (2008).

[17] Rao, J. and Ross, K.A.: Cache Conscious Indexing for Decision-Support in Main Memory, *Proc. VLDB*, pp.78–89 (1999).

[18] Rao, J. and Ross, K.A.: Making B+-trees cache conscious in main memory, *Proc. SIGMOD*, pp.475–486 (2000).

[19] Satish, N., Harris, M. and Garland, M.: Designing efficient sorting algorithms for manycore GPUs, *Proc. IPDPS*, pp.1–10 (2009).

[20] Satish, N. et al.: Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort, *Proc. SIGMOD*, pp.351–362 (2010).

[21] Schlegel, B. et al.: k-ary search on modern processors, *Proc. DaMoN*, pp.52–60 (2009).

[22] Sewall, J. et al.: PALM: Parallel Architecture-Friendly Latch-Free Modification to B+Trees on Many-Core Processors, *Proc. VLDB* (2011).

[23] Silvestri, F. and Venturini, R.: VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming, *Proc. CIKM* (2010).

[24] Yan, H., Ding, S. and Suel, T.: Compressing term positions in web indexes, *Proc. SIGIR*, pp.147–154 (2009).

[25] Zhou, J. and Ross, K.A.: Buffering accesses to memory-resident index structures, *Proc. VLDB*, pp.405–416 (2003).

[26] Zukowski, M., Heman, S., Nes, N. and Boncz, P.: Super-Scalar RAM-CPU Cache Compression, *Proc. ICDE*, pp.59–71 (2006).

山室 健

NTT ソフトウェアイノベーションセンタ研究員．2008 年上智大学大学院理工学研究科博士前期課程修了，修士（工学）．DBMS のコア技術，およびハードウェア特性を考慮した探索/圧縮アルゴリズムの研究に従事．日本データベース学会会員．

鬼塚 真　（正会員）

1991 年東京工業大学工学部情報工学科卒業．同年日本電信電話株式会社入社．2000～2001 年ワシントン大学客員研究員，2010～2014 年日本電信電話株式会社特別研究員，2012～2014 年電気通信大学客員教授，現在，大阪大学大学院情報科学研究科教授．博士（工学）．大規模グラフデータの分散データ処理に関する研究開発に取り組んでいる．2004 年情報処理学会山下記念賞，2008 年データベース学会上林奨励賞等受賞．電子情報通信学会，日本データベース学会，ACM 各会員．

（担当編集委員　堀井 洋）