データ圧縮を活用した
分析クエリの最適化技術に関する研究

提出先　　大阪大学大学院情報科学研究科
提出年月　2018 年 1 月

山 室　健

発表論文一覧

1. 学術論文（国内，査読あり）

・山室 健, 鬼塚 真, 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, 情報処理学会論文誌データベース（TOD66）, Volume 8, No. 2, 2015.

・藤原 靖宏, 中辻 真, 山室 健, 塩川 浩昭, 鬼塚 真, 精度保証付き Personalized PageRank の高速化, 電子情報通信学会論文誌, Volume J97-D, No. 4, 2014.

・山室 健, 若森 拓馬, 寺本 純司, 西村 剛, 復元せずに処理可能な圧縮形式による列指向 DB の評価と考察, 日本データベース学会論文誌, DBSJ Journal, Volume 13, No. 1, 2014.

・塩川 浩昭, 山室 健, 藤原 靖宏, 鬼塚 真, SIMD 命令によるモジュラリティに基づくグラフクラスタリングの並列化, 日本データベース学会論文誌, Volume 12, No. 1, 2013.

2. 学術会議（国際，査読あり）

・Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura, Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis, Proceedings of the IEEE International Parallel & Distributed Processing Symposium, Pages 22-31, 2016.

・Takeshi Yamamuro, Makoto Onizuka, and Toshimori Honjo, Tree Contraction for Compressed Suffix Arrays on Modern Processors, Proceedings of International Conference on Database Systems for Advanced Applications, 2015.

・Takeshi Yamamuro, Makoto Onizuka, Toshio Hitaka, and Masashi Yamamuro, VAST-Tree: A Vector-advanced and Compressed Structure for Massive Data Tree Traversal, Proceedings of the 15th International Conference on Extending Database Technology, Pages 396-407, 2012.

・Yasuhiro Fujiwara, Makoto Nakatsuji, Takeshi Yamamuro, Hiroaki Shiokawa, and Makoto Onizuka, Efficient Personalized Pagerank with Accuracy Assurance, Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining, Pages 15-23, 2012.

・Takeshi Yamamuro, Yoshiharu Suga, Naoya Kotani, Toshio Hitaka, and Masashi Yamamuro, Buffer Cache De-duplication for Query Dispatch in Replicated Databases, Proceedings of International Conference on Database Systems for Advanced Applications, 2011.

3. 学術会議（国内，査読なし）

・山室 健, 鬼塚 真, 本庄 利守, LZE++：共有辞書を用いた圧縮データに対するランダムアクセスの高速化, 第 7 回データ工学と情報マネジメントに関するフォーラム（DEIM2015）, 2015（山下記念研究賞）.

・山室 健, 若森 拓馬, 寺本 純司, 西村 剛, 復元せずに処理可能な圧縮形式による列指向 DB の評価と考察, 第 6 回データ工学と情報マネジメントに関するフォーラム（DEIM2011）, 2014.

・山室 健, 鬼塚 真, 小西 史和, VAST 木における圧縮による CPU ペナルティの実験と考察, 第 5 回データ工学と情報マネジメントに関するフォーラム（DEIM2011）, 2013.

・山室 健, 鬼塚 真, 日高東潮, 山室 雅司, CPU 特性を考慮した圧縮接尾辞配列の高速化手法, 第 4 回データ工学と情報マネジメントに関するフォーラム（DEIM2011）, 2012.

・山室健, 鬼塚真, 日高東潮, 山室雅司, 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, 第 3 回データ工学と情報マネジメントに関するフォーラム（DEIM2011）, 2011.3（優秀インタラクティブ賞）.

内容梗概

近年の CPU の高度化や DRAM の低価格・大容量化を背景に，DBMS で処理を行うデータの大半がメモリに載るようになり，結果として HDD などの低速な 2 次記憶装置の入出力以外の処理（例えば CPU 内部の計算処理やメモリ参照）がボトルネックになる事例が多くなってきている．そのため DBMS で用いる要素技術では処理データがメモリ上にある前提で，CPU 内部の並列性（命令レベル並列性，データレベル並列性，スレッドレベル並列性）の考慮，処理データを CPU に転送する際に発生するメモリ参照の遅延，CPU とメモリ間のバス転送の最大帯域など様々な観点を考慮した設計が重要になってきている．

そこで本研究では DBMS の要素技術として用いられる読み込み処理，探索処理，結合操作などに焦点を当てて，メモリ上の処理を前提とした計算で発生する技術的な課題の明確化，またその課題を解決する手法に関する研究に取り組む．具体的には，DBMS 内部で用いるアルゴリズムとデータ構造における CPU の処理特性を考慮した最適化と，メモリ上のデータ表現を考慮したコストモデルの確立を行う．これらの課題に取り組むことで，大規模データをメモリ上で効率的に処理可能な DBMS の実行方式を確立することが本研究の目的である．

本論文は５章から構成され各章の内容は次の通りである．まず１章において序章として研究背景と目的について述べる．第２章では，大規模データに対して空間コストが低く CPU の実行効率が高い木構造索引を提案する．この手法の特徴は，データ圧縮を活用することで索引サイズの削減を行いながら，SIMD 命令を用いた比較処理のデータ並列度を改善している点である．具体的には分岐ノードの比較キーに対して非可逆な圧縮手法を適用することで，圧縮後の比較キーの bit 長を 8 や 16 に揃え，128bit の SIMD 命令で同時に比較可能なデータ数を向上させる．また索引全体のサイズを削減することを目的に，葉ノードには CPU に最適化された可逆圧縮手法を適用する．提案手法の有効性確認のため性能評価を行い，データ数が 2^{30} の分岐ノードのみのサイズは既存手法に対して 95%以上削減，索引全体のサイズとしては 47～84%削減できることをそれぞれ確認した．またスループット性能に関しては 2 分木に対して 6.0 倍，既存手法で最も高速な

FAST に対して 1.24 倍の性能向上を確認した．探索中の CPU とメモリ間のバスの帯域消費量に関しても，2 分木に対して 94.7%，FAST に対して 40.5%削減できることを併せて確認した．

第３章では，圧縮された文字列データに対して，圧縮データ全体を復元せずに任意の位置にある部分文字列データを高速に参照可能な圧縮表現を提案する．先行研究で提案されている長さ m の部分データ列を $O(m)$で復元可能な圧縮手法を参考に，本技術では共有辞書を用いた再帰処理の枝刈りと，復元済みデータを参照することで再帰処理を軽量なループ処理に置換する最適化を行う．提案手法の性能評価のために，自然文・Web データ・サーバログなどの特性の異なる 9 個のデータを用いた実験の結果を示し，その有効性について検証した．評価結果としてメモリ上の処理にもかかわらず 64KiB 以下の圧縮データの復元処理で 3.3～58.2 倍，64KiB より大きなデータの復元処理では最大で 1439.0 倍の大幅な改善が可能であることを確認した．

第４章では，内部の表データを列方向に分割・圧縮を行う列指向 DBMS アーキテクチャを前提として，圧縮データの復元時に発生する CPU 計算量を考慮したコストモデルを提案する．本手法では，先行研究で提案されているメモリの参照パタンの組み合わせを用いてモデル化されたコスト T_{Mem} に対して，各圧縮データ形式に対応した CPU 計算量を表現するための補正項 T_{Ins} を加算することで，先行研究で提案されているコストモデルと比べてより高い精度が実現できる点が特徴である．提案手法の導出にあたり，列指向 DBMS のプロトタイプを用いたコストモデル分析の結果を示し，その妥当性について検証する．

最後に第５章では，本研究で考案した技術を要約したのちに，今後の研究の展望に関して述べることで本論文のまとめとする．

目次

第 1 章 序章

1.1 研究背景

近年の情報社会の高度化により管理・分析する必要のあるデータは増加傾向にあり，その大量に蓄積された大規模データを効率的に扱うことのできる基礎として DBMS（Data Base Management System）を構成する要素技術の継続的な発展は非常に重要である．DBMS の中でも関係モデル [1]に基づいて設計・開発された DBMS を RDBMS（Relational DBMS）と呼び，IBM の System R [2]が世界で最初に開発された RDBMS である．現代の企業の情報管理システムにおいて広く導入されている Oracle Database や PostgreSQL[1]などの RDBMS の内部実装は，System R の要素技術の影響を大きく受けている．また DBMS 分野に限らず DBMS の要素技術は他の分野においても広く活用されている．Google が 2004 年に論文発表を行った大規模データ処理のための分散処理フレームワーク MapReduce [3]を参考に，OSS（Open Source Software）として実装された Hadoop[2]がある．Hadoop はユーザが map 関数と reduce 関数を記述することで分散処理を行うが，これらの関数の記述の困難さから，2009 年に Facebook が RDBMS においてデータの定義・操作を行うための問い合わせ言語 SQL に似た DSL（Domain Specific Language）である HiveQL を採用した Hive[3] [4]の提案を行う．Hive では，ユーザが記述した HiveQL を内部で map 関数と reduce 関数に暗に変換して分散処理を行う．そのためユーザが容易に大規模データに対して処理を記述できるため，現在でも広く利用されている．現在の Hive の実装には，HiveQL の問い合わせの最適化に System R で培われた手法，1990 年前半に研究された関係モデルの最適化を一般化する手法 [5]，列方向のデータ表現である DSM（Decomposition Storage Model） [6]に分類される ORC[4]や Parquet[5]など多くの DBMS の要素技術が活用されている．また MapReduce を一般化した分散処理モデルを採用し，分散メモリを活用した分散処理フレーム

[1] https://www.postgresql.org/
[2] http://hadoop.apache.org/
[3] https://hive.apache.org/
[4] https://orc.apache.org/
[5] https://parquet.apache.org/

ワークとして有名になった Spark[6] [7]の実装にも，RDBMS で一般的に採用される Volcano-style の実行方式 [8]や，実行計画のコード生成による最適化 [9]などの手法が積極的に活用されている．

DBMS では図 1.1 に示すように，任意の形式で記述されたユーザからの問い合わせ（例えば SQL を用いたクエリ）を，処理を行うターゲットのハードウェア上で高速に実行できるように変換（最適化）を行う．1990 年代までは HDD などの低速な 2 次記憶装置での処理時間が大半を占めていたため，主に I/O 回数を最小化するように問い合わせを最適化することが一般的だった．しかし近年の CPU の高度化や DRAM の低価格・大容量化を背景に，DBMS で処理を行うデータの大半がメモリに載るようになり，結果として入出力以外の処理（例えば CPU 内部の計算処理やメモリ参照）がボトルネックになる事例が多くなってきている．本研究ではこのような背景を前提に，DBMS の要素技術として用いられる読み込み処理，探索処理，結合操作などに焦点を当てて，メモリ上の処理を前提とした計算処理で発生する技術的な課題の明確化，またその課題を解決する手法に関する研究に取り組む．

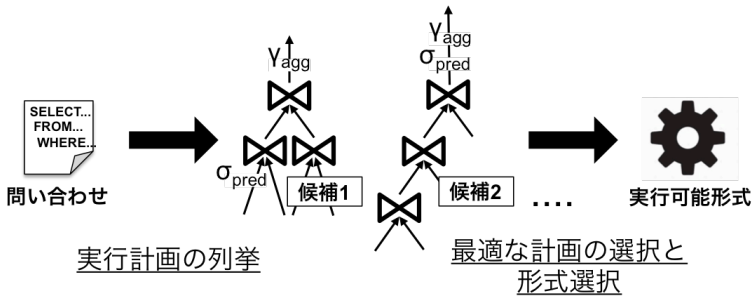

図 1.1 ユーザからの問い合わせの最適化

1.2 CPU の高度化とメモリの大容量化における DBMS の問題

1.2.1 CPU の高度化と CPU 最適化の必要性

図 1.2 に示すように 2000 年以降，CPU 内部の発熱などの影響によりクロック周波数が向上しなくなってしまったため，プログラムを記述する際に CPU の命令レベル並列性（ILP: Instruction-Level Parallelism），データレベル並列性（DLP: Data-Level Parallelism），スレッドレベル並列性（TLP: Thread-Level Parallelism）を

[6] https://spark.apache.org/

考慮した設計がより重要になってきている．大抵の CPU では 1 サイクル当たり複数の CPU 命令発行が可能で，実際に発行できた命令数が CPU の実行効率を示す指標（IPC: Instructions Per Cycle）となっている．命令列やデータ参照の依存関係などが原因で命令が複数発行できずに ILP が低下し，結果として IPC が低い値になるため，これらの依存関係を減らす考慮が必要になる．また CPU 内部の計算密度をより高めるために，1 命令で複数のデータを処理することで DLP を向上させるベクトル（SIMD: Single Instruction，Multiple Data）命令の活用も重要である．SIMD は Flynn の分類 [10]と呼ばれるコンピュータの分類体系の 1 つで，SIMD は単一の命令で複数のデータの処理（例えば，4 つの比較処理を 1 命令で処理）を行うコンピュータの分類である．近年の CPU ではこれを命令セットの一部として実装しており，最新の Intel の CPU（例えば Skylake EP/EX の Xeon）では AVX512 と呼ばれる 512bit 長の SIMD 命令が実装され，16 個の 4byte 整数が同時に評価可能である．

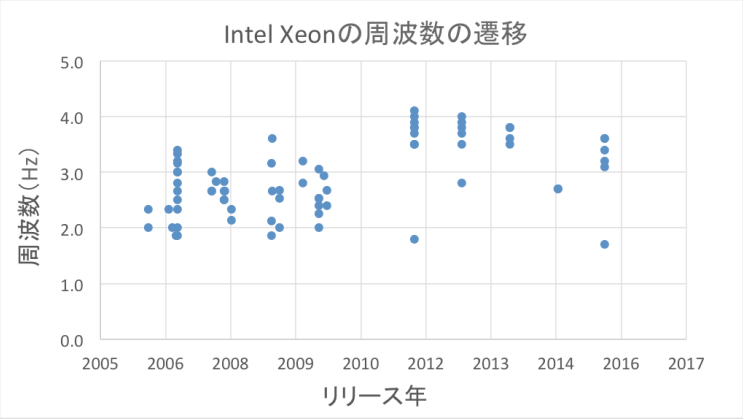

図 1.2 CPU（Intel Xeon）のクロック周波数の向上

Intel 研究所による CPU 最適化に関する報告書[7] [11]によれば，CPU キャッシュと TLB[8]（Translation Look-aside Buffer）を考慮したデータの再配置，命令の依存関係を減らすための分岐除去，SIMD 命令によるデータ並列化などの最適化を行う場合と行わない場合の性能差はメモリ上の処理にもかかわらず平均で 24 倍，最

[7] この報告書ではデータの探索やソートを含めた代表的なアルゴリズム計 11 種を用いて性能比較が行われている．

[8] 頻繁に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．

大で 53 倍だと報告されている．特に DBMS の内部処理は分岐などを多く含み命令の依存関係が複雑で IPC が非常に低いプログラムの代表である．この問題を背景に MonetDB の X100 プロジェクトでは，CPU キャッシュの効率化や SIMD 命令の活用など DBMS 内部で用いる CPU 効率の高いアルゴリズムとデータ構造を提案している [12, 13].

1.2.2 メモリの大容量化と性能ボトルネック

市販されている単体の物理サーバのメモリサイズは近年 100GiB を超えるものが一般的で，GCP（Google Cloud Platform），Microsoft Azure，AWS（Amazon Web Services）などの IaaS（Infrastructure as a Service）として提供されているサーバでは 1TiB 近いメモリを搭載したインスタンスも提供されはじめている．そのため 1.1 節で述べたように，処理するデータサイズと比較してメモリサイズが大きく，CPU 内部の計算処理やメモリ参照がボトルネックになることが多くなってきている．先行研究で指摘（図 1.3）されている通り，CPU の計算速度の向上と比較してメモリ参照速度の向上は緩やかである．そのため DBMS に代表されるようなデータ操作が非常に多いプログラムでは，CPU のロード命令がメインメモリ上のデータを参照した場合に，該当データを CPU とメモリ間のバス（図 1.4）を経由して CPU に転送する際に発生する遅延がボトルネックになることが先行研究で指摘されている [14]．そのためメモリ参照の遅延を隠蔽するために相互に依存（データ依存と制御依存）関係がなく並列実行可能な命令でプログラムを構成することは性能改善の観点で非常に重要である．

メモリ参照の遅延だけではなく，CPU とメモリ間のバスの転送帯域の狭さもデータの参照が多いプログラムでは問題になる．特に 1.2.1 節で述べた 1 命令で複数のデータを処理する SIMD 命令は，他の命令と比べて多くのデータを参照するため SIMD 命令用のレジスタ長が大きいほど転送帯域を消費することになる．そのため参照するデータのサイズを小さくすることや，そもそも処理に不必要なデータの参照を抑制することが非常に重要になる．この問題に対する DBMS 分野における代表的なアプローチは，DSM に基づく列方向のデータ表現である．分析処理を主に行う DBMS では属性（列）数が多く，行方向のデータ表現

（NSM: N-ary Storage Model）では CPU が参照しない列まで L1/L2 キャッシュにロードされてしまう傾向がある．この不必要なロードを削減してバスの転送帯域を節約するために，列方向のデータ表現は適している．このデータ表現を採用したカラム指向 DBMS アーキテクチャの先駆けである C-Store と MonetDB/X100 の研究が発表されて以降，カラム指向 DBMS に関する多くの研究や開発が行われ，現在では分析向けのミドルウェアで広く採用されている．

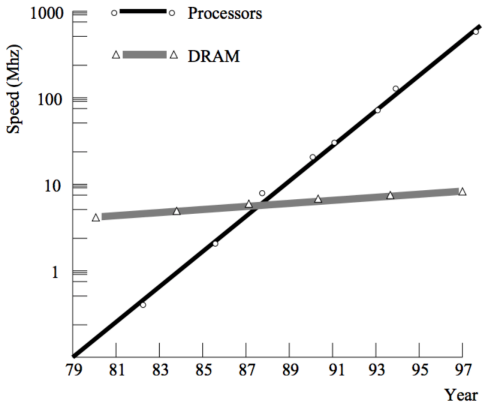

図 1.3 CPU の計算速度とメモリの参照速度（[14]の Figure1 より引用）

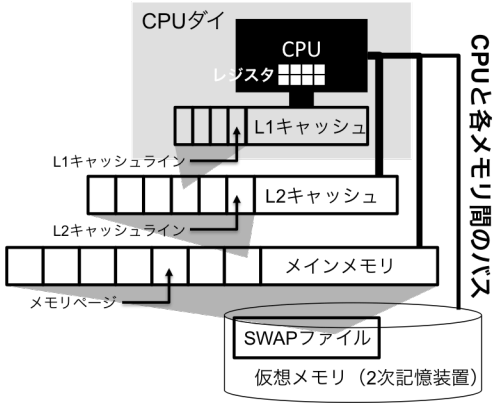

図 1.4 CPU とメモリ間のバス

1.3 研究課題と目的

本研究では近年の CPU の高度化とメモリの大容量化に対して，DBMS の内部で用いられる要素技術をこれらのハードウェアに適応させることで DBMS 上での処理を効率化し，増大する分析データを効率的に処理可能な DBMS 技術の確立を目指す．具体的には大規模データをメモリ上でコンパクトに表現しながら効率的に処理を行うため，データ圧縮を活用することで CPU 効率の高い DBMS の実行方式を検討する．

以下の課題を本研究では取り扱う．

- 課題 1: DBMS 内部で用いられるアルゴリズムとデータ構造の CPU 上での非効率性分析と解決（図 1.5 の左）
- 課題 2: データ表現（圧縮形式）を考慮した実行計画の列挙とコストモデルの確立（図 1.5 の右）

DBMS ではユーザからの問い合わせに回答するため，利用可能であれば対象のデータを高速に参照することのできる索引を活用して対象の位置を特定し，任意の圧縮形式で格納されたデータを取得する．データ規模が大きい場合，データ読み込みに近い処理がボトルネックになる傾向があるため，課題 1 としてデータの参照に関するアルゴリズムとデータ構造に着眼する．具体的には CPU の処理特性を考慮して最適化された索引構造と探索，またメモリ上の効率的なデータ圧縮形式に関して検討を行う．

1.1 節で述べたように，従来の DBMS では主に I/O 回数が最小になるように問い合わせを最適化していたが，任意の圧縮形式でデータがメモリ上に格納されている場合には CPU 内部の計算量やメモリ参照回数を考慮する必要がある．例えば入力データが 2 つあり，それぞれが任意の圧縮形式でメモリ上に配置されている場合に，選択操作の後に結合操作を行う場合と，結合操作の後に選択操作を行う場合のどちらの実行計画が良いかを DBMS が I/O 回数以外の変数を考慮することで判断する必要がある．そのため課題 2 として，圧縮形式を考慮したコストモデルの確立を行う．

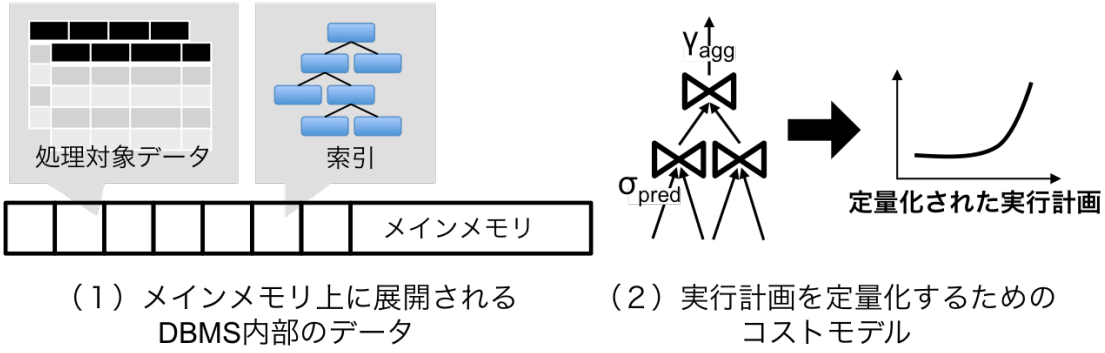

図 1.5 課題の概要

1.4 提案技術の概要

本節では課題 1 のアルゴリズムとデータ構造に関するものと課題 2 のコストモデルに関するものに分類して，上記の CPU 効率の高い DBMS の実行方式を達成するために考案した技術についてその概要を説明する．

1.4.1 アルゴリズムとデータ構造

CPUの処理特性を考慮した索引構造と探索

課題1に対して，DBMS内部でデータ探索に用いる索引構造におけるCPU上の非効率性に着眼し，大規模データに対して空間コストが低くCPUの実行効率が高い木構造索引（VAST木）を提案する．従来の研究 [16, 26, 33]では索引の探索時の分岐ノード内の比較処理をSIMD命令により並列化することで性能改善を図る提案が行われている．併せて探索中に参照するキャッシュラインとメモリページの総数を最小化するための比較キーの再配置とパディング[9]を行う．しかし，データ並列数が比較キーのbit幅に制限（128bit長のSIMD命令で比較キーが32bitの場合，最大並列数は4）される点と，データの規模が大きくなった場合にパディングによる索引サイズが肥大化する欠点がある．本技術VAST木の特徴は，データ圧縮を活用することで索引サイズの削減を行いながら，SIMD命令によるデータ並列数を高めることで性能改善を同時に実現している点である．分岐ノードの比較キーに対して，比較に必要なMSB（Most Significant Bit）以降のbitのみを取り出す非可逆な圧縮を適用することで，データサイズを削減しながらデータ並列数の向上（圧縮後の比較キーが8bitの場合，最大並列数は16）を実現する．この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で，圧縮する前に大小関係がある比較を同値と判断してしまう不正な処理が発生する場合がある．そのため，探索処理の最後にこの不正な比較が行われてかどうかを判定し，行われている場合には本来の探索位置からのズレを訂正する処理を実施する．

メモリ上の効率的なデータ表現

課題1に対して，DBMS内部で文字列型の属性データの圧縮に用いられる手法のCPU上の非効率性に着眼し，圧縮データ全体を復元せずに任意の位置にある部分文字列データを高速に参照可能な圧縮表現（LZE++）を提案する．圧縮されたデータ全体を復元せずに部分データを効率的に参照する最も単純な方法は，圧縮対象のデータ列を固定長kのブロックに分割をして各ブロックを既存の手法で圧

[9] データサイズがキャッシュラインやメモリページのサイズに満たない場合に，満たしていない残りの部分を全て 0 で埋める操作のこと．

縮を行う方法である．データ列の先頭からi番目に位置する部分データ列を復元する場合にはi/k番目のブロックのみを復元して$i\%k$番目に位置する部分データ列を参照すれば良い．より洗練された従来研究では，簡潔データ構造を用いることで長さmの部分データ列を$O(m)$で復元可能な手法（LZEnd [42]）が提案されている．簡潔データ構造はデータ構造のサイズを情報理論下限に漸近させながら，データの明示的な復元をせずに列挙や探索などの問い合わせを可能にする手法である．しかし，LZEndでは復元処理に必要な$O(m)$回の再帰処理で発生する簡潔データ構造の問い合わせ処理によるCPUペナルティ（実行命令数やキャッシュミス回数の増大）が性能上の課題となる．本技術LZE++の特徴は，共有辞書を用いた再帰処理の枝刈りと，復元済みデータを参照することで$O(m)$回の再帰処理を軽量なループ処理に置換している点である．共有辞書は入力データ列（長さN）から取り出した長さM（$M<<N$）の部分データ列である．再帰処理による復元を行っている際に，現在復元しようとしている符号が共有辞書内の部分データ列を示している場合に，辞書内のデータを参照することで以降の再帰処理の枝刈りを行う．また一定数以上のデータ列を復元した際に，復元しようとしている符号列が復元済みのデータ列を参照している場合がある．これを活用することで$O(m)$回の負荷の高い再帰処理を平均（$mK/N+1$）回（Kは圧縮後の符号の総数で，$K \leqq N$）の軽量な繰り返し処理で置換する．

1.4.2 コストモデル

課題 2 に対して，列指向 DBMS アーキテクチャで用いられるコストモデルの精度の悪さに着眼して，圧縮データの復元時に発生する CPU 計算量を考慮したコストモデルを提案する．DBMS では問い合わせを実行する前に複数の実行計画の候補から最も効率的なものを選択する必要があるため，ユーザからの問い合わせと処理対象のデータの特性から実行計画のコストを推定するためのコストモデルが必要になる．メモリの大容量化を背景に，先行研究では処理対象のデータが全て（もしくは大部分が）メモリ上にある前提で実行計画の参照パタンの組み合わせでモデル化を行い，実行コストを処理中に発生する CPU のキャッシュミス回数で見積もり行うコストモデルが提案されている．しかし圧縮された列デー

タの復元処理の計算量は高く，メモリ上の実行計画の参照パタンだけでは実行計画のコストを正確に推定することができない欠点があった．本技術では先行研究で提案されているメモリの参照パタンの組み合わせを用いてモデル化したコスト T_{Mem} に対して，各圧縮形式に対応した CPU 計算量を表現するための補正項 T_{Ins} を加算することで，従来研究に比べてより精度の高い実行計画のコスト推定を実現できる点が特徴である．補正項 T_{Ins} の算出には，各圧縮形式が 1 行の圧縮されたデータを処理するために必要な CPU サイクル数を事前に分析することで，各物理プランに入力された圧縮データ数と実際に復元・出力をした数を用いている．

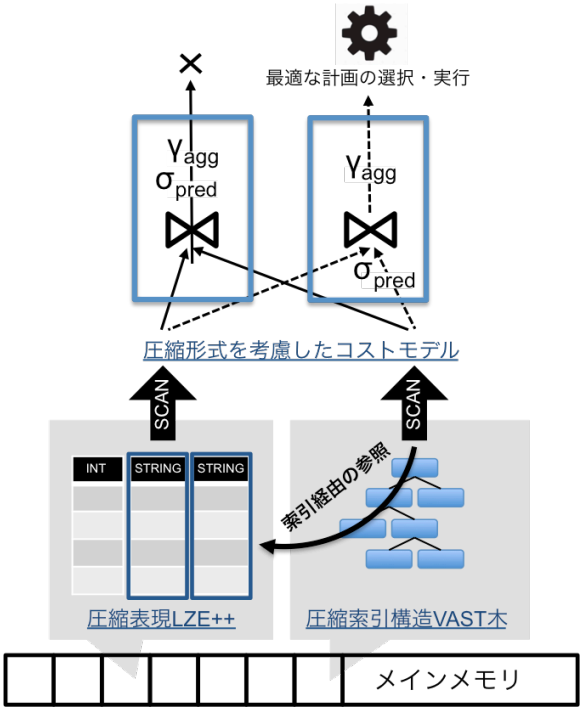

図 1.6 提案技術の関連

1.5 提案技術の関連性

本論文で考案した技術の関連性を図 1.6 に示す．DBMS を利用するユーザはまず初めに DDL（Data Definition Language）を用いることで，テーブル名，属性名，属性の型，キー制約，索引の有無などのデータ定義を行う．DBMS（ここでは列指向 DBMS を想定）はユーザから受け取ったデータ定義の情報を用いて DBMS 内部のデータ構造の初期化を行うが，ここで本提案技術の CPU 効率の高いデータ構造を活用することができる．例えばユーザが指定した属性の型に文字列（図 1.6 中の STRING）が含まれる場合，DBMS は本提案技術である文字列の圧縮表

現 LZE++を用いることで，メインメモリ上でコンパクトにデータを表現しながら効率的に部分データの参照を行うことができる．もしこの属性に対して索引が設定されている場合，索引を経由した部分データの参照を行う必要があるため，高速な部分データの参照は効率的に問い合わせ処理を行うために重要である．整数の型[10]（図 1.6 中の INT）をもつ属性に索引が設定される場合，DBMS は索引として本提案技術の VAST 木を用いることで，問い合わせ処理に必要な部分データを高速に絞り込むことが可能になる．VAST 木は整数列に対する索引構造だが，文字列の先頭 8byte を固定長整数とみなして索引を設定するなど他の手法[11]と併用することで文字列に対する索引構造として利用できるため，LZE++と VAST 木を用いて高速な文字列データの参照が実現できる．

ユーザはデータ定義の後に分析を行うための問い合わせを DBMS に対して行う．DBMS は問い合わせ内容から等価な複数の実行計画を生成し，この複数候補の中から最適な実行計画を選択して処理を行う．この際 DBMS はデータ定義にあわせて圧縮して格納されている整数や文字列などのデータ列に対して，参照パスのコストを考慮した実行計画の選択を行う必要がある．そのため本提案技術である圧縮形式を考慮したコストモデルを利用することで，より精度の高い実行計画のコスト推定を実現し，結果として最適な実行計画を選択することの可能な DBMS を構築することができる．

1.6 論文の章構成

本論文の構成を示す．CPU に最適化されたメインメモリ上に展開される DBMS 内部データに関する技術については 2，3 章で述べる．2 章に関しては索引の分岐ノードを圧縮することで SIMD 命令を用いた同時比較数を向上させる索引の構築技術について述べる．3 章では，圧縮された状態で任意の位置のデータを復元可能なメインメモリ上のデータ構造に関する復元処理の高速化技術について述べる．そして 4 章では，明示的に復元処理を行わずに処理可能な様々な圧縮

[10] 大抵のログデータには生成時刻の情報が含まれており，その時刻データは Timestamp/Date 型で保持されることが多く，これらは内部的に整数表現で格納される．

[11] 可変長データ（文字列やバイナリ）に対する処理の最適化として先頭 *n*-byte の固定長の整数を用いることは一般的で，例えば Spark では並び替え処理の最適化に利用している．

形式を用いて，列指向 DBMS でクエリを実行する際に発生するコストの分析，また CPU 計算量を考慮したコストモデルの構築に関する技術について述べる．最後の 5 章では本論文で考案した技術の要約と今後の展望についてまとめる．

第 2 章 探索の高速化

2.1 はじめに

データ集合から指定の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合に索引サイズの肥大化とIPCの低下の問題が顕在化する．この索引サイズに関する問題の具体例を示すために，ポインタを用いた 2分木の広く知られた実装とPostgreSQL v9.0.1のB+木の索引を用いて，データ数が2^{28}〜2^{32}の場合の索引サイズを表2.1に示す（表の括弧内は葉ノードを除いた分岐ノードのみのサイズを表す）．

表2.1 2分木とB+木の索引サイズ（GiB）比較

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

表からデータ数が多くなると索引サイズが10〜100GiB程度になり非常に大きくなることが分かる．索引がメモリサイズを超えると仮想メモリ上のSWAPファイルに追い出されるため，極端な性能劣化が発生する．そのため索引サイズの削減は重要な課題の1つといえる．CPUの処理特性を考慮した設計を行っていない手法のIPCは一般的に低く [11]，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 [15]されている．図2.1にCPUとしてXeon X5260を用いて，データ数を2^{22}〜2^{28}と変化させた場合の2分木の完全一致探索の実行時間内訳を示す（図中の括弧内は各実行におけるIPCを表す）．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は0.3前後であり非常に低い値であることが分かる．

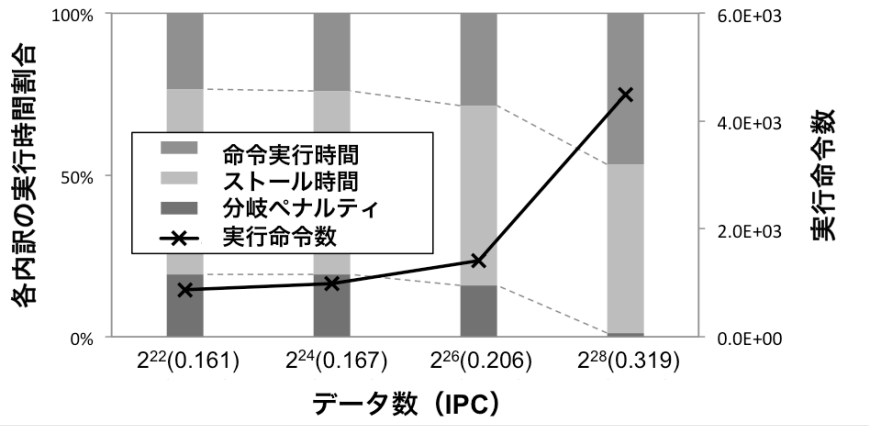

図2.1 索引対象のデータ数が2^{22}～2^{28}におけるXeon X5260を用いた実行時間内訳（命令実行時間，ストール時間，分岐ペナルティ）の比較

木構造索引におけるCPUの実行効率の課題を取り扱った最新の提案はKimらによるFAST [16]である．FASTはメモリ上の索引構造で，更新処理がないことを前提にCPUの処理特性を考慮したデータ構造設計と探索処理により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なSIMD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を大規模なデータに適用する場合には以下のような問題が顕在化する．

- 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行うが，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，データ並列数が3に制限されている（128bit長のSIMD命令で比較キーが32bitの場合）．
- 葉ノードの圧縮が取り扱われていない．

本章で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メモリ上の索引構造を前提とする．本研究では大規模なデータに対しても索引を構築できるように，圧縮技術を活用することでコンパクトなデータ構造をメモリ上で実現する．ただし複雑な復元処理が必要な圧縮手法を用いると探索性能が悪化するため，シンプルな圧縮手法を選択するこ

とで探索性能も併せて改善可能な設計を行う．そこでVAST木では SIMD命令でのデータ並列度を改善するための不可逆な圧縮を分岐ノードに適用し，CPU効率の高い可逆圧縮を葉ノードに適用する．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可能にしている．分岐ノードを複数の領域に分割して，それぞれの領域で異なる不可逆圧縮手法を適用する．木構造の分岐ノードは上部に比べて下部が大きくなるため，木構造の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．具体的には分岐ノードの比較キーに対して非可逆な圧縮手法を適用することで，圧縮後の比較キーのbit長を8や16に揃え，128bitのSIMD命令で同時に比較可能なデータ数を向上させる．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）が結果として返される．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．

本研究の学術・社会的な貢献は以下である．

- 大規模なデータに対して空間コストが低く，SIMD命令を用いた高いデータ並列度による高速な探索を実現するVAST木を提案した．本手法は木構造全体に対して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SIMD命令によるデータ並列数の改善も実現している．
- 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%以上の削減，また索引全体のサイズは47～84%の削減を実現している．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6.0倍，既存手法で最も効率的なFASTに対して1.24倍の性能向上も確認した．
- 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

バスの転送帯域の消費量に関して，データ数が2^{28}の探索処理において2分木に対して94.7%，FASTに対して40.5%の削減を確認した．

- VAST木で発生する探索のズレΔeをモデル化し，このモデルを用いて発生する探索のズレの訂正処理範囲を限定化することで，訂正処理時間の最悪値を7.1～18.5%改善した．

次の2.2節では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．2.3節で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の訂正処理などの設計に関する詳述を行い，擬似コードを併せて示す．2.4節ではVAST木のプロトタイプを用いた評価実験の結果を示す．そして2.5節では探索中に発生するΔeのモデル化を行い，そのモデルを用いた訂正処理の効率化方法に関して考察を行う．最後の2.6節と2.7節で関連研究と結論を述べる．

2.2 前提知識：SIMD 命令を用いた木構造探索

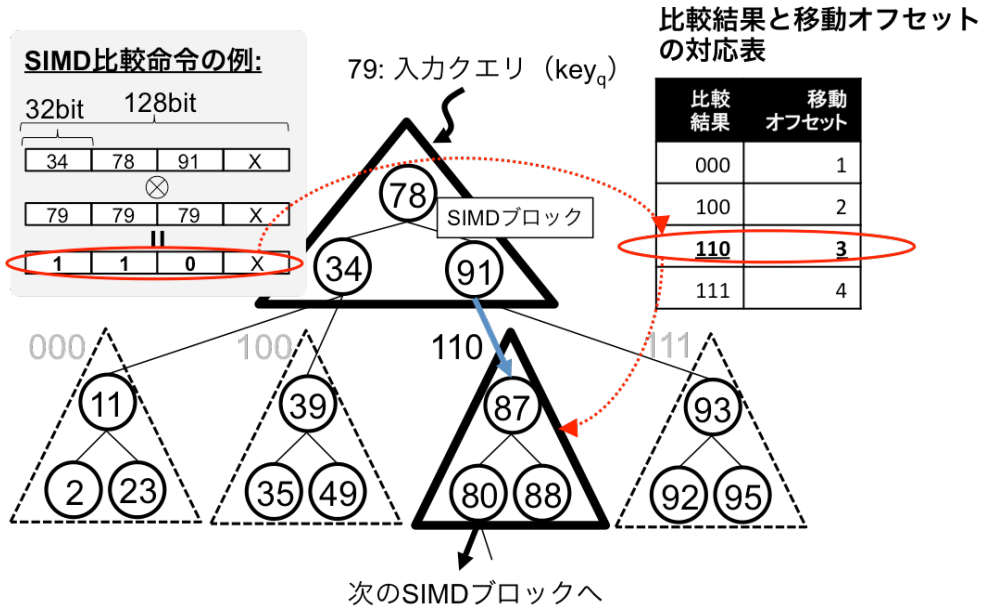

図2.2 SIMD命令による木構造探索の例

FASTにおけるSIMD命令を用いた木構造探索の概略を図2.2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128bit

のSIMD用レジスタ[12]と32bitのキーを前提に比較処理の同時実行を提案している．探索処理中に発生する分岐処理を取り除くため，FASTではSIMD命令の比較結果と移動オフセットの対応表を用いる．これにより分岐処理を積和演算に置き換え可能で，結果として制御依存をデータ依存に変換することができる．

図2.2で示した木構造は以下のように幅優先順でメモリ上に配置される．

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

角括弧内の整数列が1つのSIMDブロックを表し，下部線が図2.2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128bit長のSIMDレジスタAに[79, 79, 79, *[13]]としてロードをする．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をそれぞれ返す．そして比較結果と移動オフセットの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（32bit整数の比較キーが3個）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+12×3）だけ位置を移動させれ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2.2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L1キャッシュラインサイズ64Bに収まる．このデータ配置により，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．また上記と同様の方法で，探索中に読み込むメモリページ数を最小化するように複数のキャッシュラインブロック集合を再配置してページブロックを構成する．しかしこのブロック化により，1回のSIMD命

[12] レジスタ長が 128bit のデータ処理用の SIMD 命令として Intel の CPU には SSE 命令が，AMD の CPU には 3D NOW!が実装されている．

[13] '*'はクエリの探索に使用しないため任意の値を表している．

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（図2.2の例では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端数4B）の影響による索引サイズの肥大化が課題となる．

2.3 提案手法：VAST 木

2.3.1 データ構造設計の概要

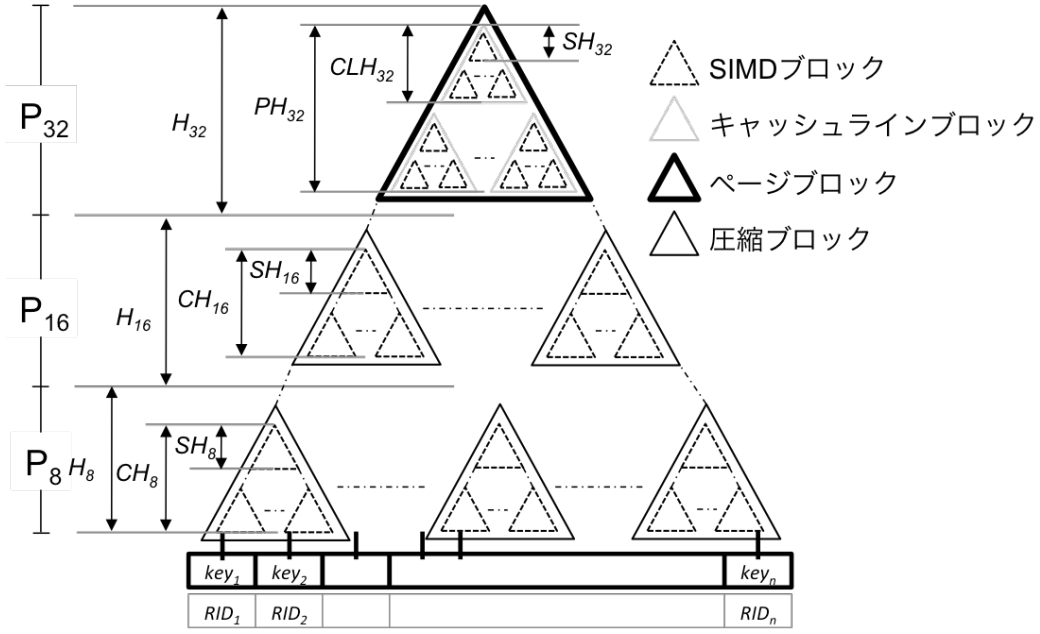

図 2.3 VAST 木の設計概要：木構造の上部 P_{32} には FAST [16]と同様のデータ構造を適用，中部 P_{16} と下部 P_8 には比較処理のデータ並列度改善のための不可逆圧縮を適用，添字の値はそれぞれの領域に含まれる比較キーの bit 長を示す

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のbit長は32か64のどちらかを想定しているが，これ以降は32bitを前提に説明を行う．図2.3にVAST木の全体構造を示す．VAST木は図のP_{nbit}で示される3つの領域（P_{32}，P_{16}，P_8）で構成され，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いる．ここで添字の*nbit*はそれぞれの領域に含まれる比較キーのbit長を示す．

高さは対応する2分木としての高さを表しており，例えば図2.3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，木構造の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2.3.1.1節と2.3.1.2節で概説する．

2.3.1.1 分岐ノードの設計概要

FAST と同様に VAST 木も図 2.3 中の SIMD ブロックは SIMD 命令を用いて比較を行う．SIMD ブロック内の比較キーは木構造の下部の領域ほど bit 長が小さいため，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[14]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に処理
- P_{16}：32bit を 16bit に圧縮，（$2^{SH_{16}}-1$）個の比較キーを同時に処理
- P_8：32bit を 8bit に圧縮，（$2^{SH_8}-1$）個の比較キーを同時に処理

P_{16} と P_8 の領域にある比較キーに対して Prefix truncation と Suffix truncation [17]を適用することで非可逆圧縮を行う．上位 bit の連続する 0 と任意の数の下位 bit を除去することで，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 hash などを用いて比較キーを圧縮する方法とは異なり，圧縮後の値を用いて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可能性がある．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 木では探索処理後に葉ノード上のデータに対して順読み込みを行うことで，正しい探索位置からのズレ Δe の訂正処理を実現する．比較キーの非可逆圧縮に関しては 2.3.2.1 節で，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に関しては 2.3.2.3 節で説明する．

図 2.3 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 [16]．P_{32} の領

[14] 64bit の場合は 4 つの領域（$P_{64}/P_{32}/P_{16}/P_8$）に分割することで本手法が適用可能である．

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木構造の下部になるほど高くなるため結果として索引サイズが肥大化してしまう．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 2.4 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 2.3.2.1 節で説明を行う．

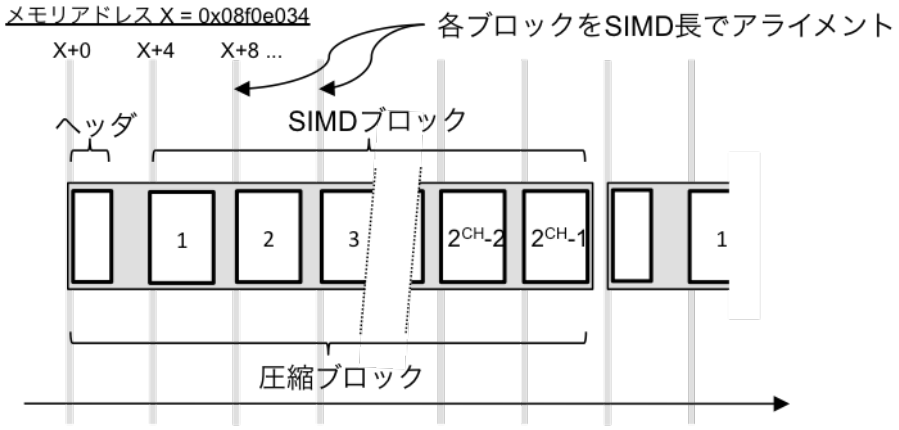

図 2.4 圧縮ブロックと SIMD ブロックのアライメントの再構成

2.3.1.2 葉ノードの設計概要

葉ノード上のデータ $key_1, key_2, ..., key_n$ は昇順の列とみなせるため，VAST 木では先行研究で提案されている昇順データ列に対する CPU 効率の高い可逆圧縮手法である P4Delta [18]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 2.3.3 節で説明を行う．この圧縮手法を用いることで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．

2.3.2 分岐ノード構造における再構成

2.3.2.1 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$V=v_1, v_2, ..., v_m$を圧縮ブロック内に含まれる昇順の比較キー列とする．比較キーの非可逆圧縮は以下の2ステップの処理を各圧縮ブロックに適用することで行う．

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）と減算してWを算出する，具体的には$W=0, v_2-v_{min}, v_3-v_{min}, ..., v_m-v_{min}$となる．
2. Wの最大値w_mのMSBから$nbit$分の範囲に対応したW内の各値のbit列を圧縮後の比較キー列とする，$nbit$の値はP_{16}の領域が16，P_8が8である．

図2.5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2.4）する．圧縮ブロック内の探索に関しては次の2.3.2.2節で説明を行う．$nbit$の値が高いほど圧縮率は高いが，圧縮前後で順序関係を保持しない比較キーの発生確率も高くなる．圧縮前後で順序関係を保持しない具体的な例は2.3.2.3節で説明する．

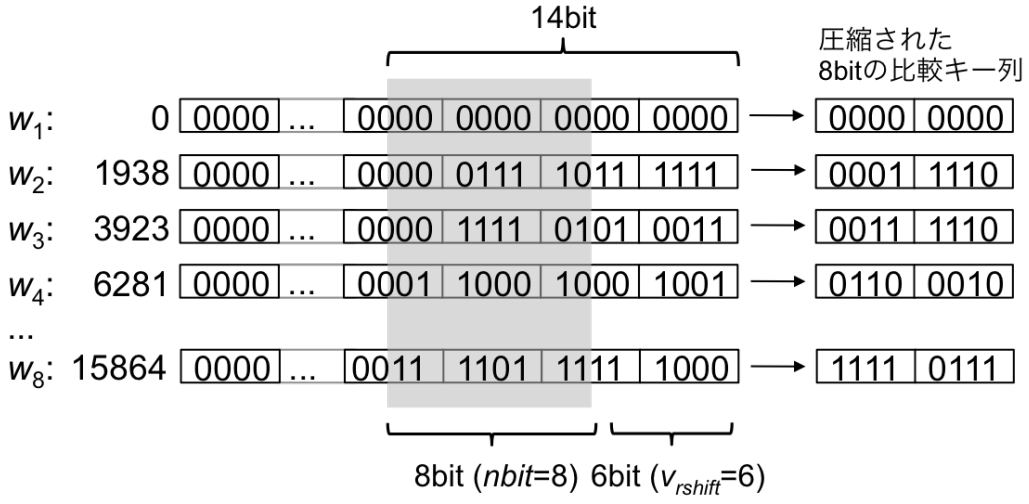

図 2.5 P_8 内の 8 つの値 $w_1, w_2, ..., w_8$ における不可逆圧縮の例

2.3.2.2 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．この節では圧縮ブロック内の探索処理を，続く 2.3.2.3 節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以下のように適用することで比較を可能にする．

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 2.2 節で説明した SIMD 命令による探索処理に従った方法で比較を行う．2.3.4.2 節では P_8 の領域における探索処理を行う擬似コードを示す．

2.3.2.3 Δe の訂正処理

まず 2.3.2.1 節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 110010010100）が 201（2 進表記で 11001001）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 110010010101）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．2.3.4.3 節で訂正処理の擬似コードを示す．

2.3.2.4 Δe の訂正処理を活用した索引サイズの削減

2.3.2.3 節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ことができることを示す．具体的には分岐ノードにおける最下部の SIMD ブロック（図 2.3 の P_8 領域の最下端）を取り除くことで分岐ノードサイズの削減を実現する．この削減により，全ての探索処理に対して最大で探索のズレ $\Delta e=2^{SH_8}-1$ が発生する．しかし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葉ノード上のデータ数は多い．そのため単体キャッシュライン内の訂正処理で完結できる可能性が高く，探索性能への影響は小さいと予想される．事前実験から性能劣化に対して圧縮効果が非常に高いことが判明したため，2.4 節の評価実験では全ての条件に対してこの手法を適用する．

2.3.3 葉ノード構造における再構成

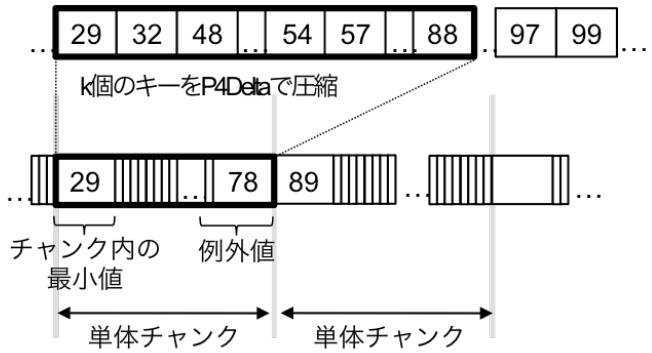

図2.6 葉ノードにおける可逆圧縮の概要：上部が圧縮前のキー列を，下部が圧縮後のキー列をそれぞれ表し，k個のデータ を1つのチャンクとして圧縮

P4Deltaを葉ノード上の昇順データ $key_1, key_2, ..., key_n$ 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．この手法は差分値 D をcodedとexceptionsの2つに分類する．大半の値（例えば90%以上の値）が表現可能なbit長 b を定め，b-bitとして圧縮して格納する値をcodedとする．一方 b-bitで表現できない b-bitより大きな値をexceptionsとして非圧縮で格納する．図2.6に葉ノードの圧縮の概要図を示す．2.3.1.2節の葉ノー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，2.3.4.3節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値は2分法（Bisection Method）を用いて決定する．チャンクのアライメントサイズS_{align}とデータのbit長L_{key}としてk値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$|S_{align}/L_{key}|$とS_{align}とする．
2. k_{middle}（$=|(k_{left}+k_{right})/2|$）個のデータを圧縮した場合に，キャッシュラインに圧縮後のデータが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$|S_{align}/L_{key}|$として圧縮前後でサイズが変わらない場合，最大値をS_{align}としてデータのbit長が1になる場合に注意してk値の有効範囲を設定する．

2.3.4 VAST 木の疑似コード

VAST木を構築する擬似コードをAlgorithm 1に，P_8の分岐ノード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に示す．

2.3.4.1 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）．

2.3.4.2 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．外部ループ処理が全て完了した際の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．

2.3.4.3 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．

2.4 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における評価を行う．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}件）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2.7に実データの差分値Dの分布を示す．

図から大半のデータが重複する偏ったデータであることが分かる．

Algorithm 1 VAST 木を構築するための疑似コード

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

評価環境がキャッシュラインサイズ64B，ページサイズ4KiB（kernel-2.6.18-194のCentOS v5.5），SIMDレジスタ128bitであることを前提に，2.3.1節で説明したアライメントの調整を行うためにVAST木の索引構造の各変数（PH_{32}，SH_{16}，CH_{16}，SH_{8}，CH_{8}）をそれぞれ（2，4，8，3，7，4，8）と設定した．本評価はXeon X5670（物理6コアで最大メモリバス転送帯域31.8GiB/s）のサーバを用いて実施した．Xeon X5670は32KiBのL1キャッシュ，256KiBのL2キャッシュ，12MiBのLLC（Last Level Cache）を搭載している．CPUの実行時間内訳はoprofilev0.9.6を用いて取得した．評価実施時にXeon X5670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Xeon X5260（物理2コアで最大メモリバス転送帯域

21.2GiB/s）を用いた．評価用コードは全てC++で記述し，gcc（-O3）のv4.1.2でコンパイルして評価を行った．

Algorithm 2 P_8 探索のための疑似コード

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

2.4.1 VAST 木の評価

2.4.1.1 索引構造の圧縮性能

表2.2に索引の分岐ノードのサイズ比較結果（図中のVAST木の括弧内はH_{32}とH_{16}の値をそれぞれ表す）を，表2.3にVAST木の葉ノードの圧縮率（図中の括弧内の値は2.3.3節で説明した手法で決められたk値を表す）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2.2から分かるとおり，全ての条件において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[15]．分岐ノードの圧縮率が高い理由は，2.3.1.1節で説明したPrefix/Suffix truncationによる比較キー圧縮とSIMDブロック単位のアライメント調整，また2.3.2.4節で説明した木構造の最下部のSIMDブロック除去によるものである．VAST木の変数（H_{32}/H_{16}/H_8）の値は索引サイズと2.3.2.3節で説明した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，それ以外の場合はH_{32}=8とH_{16}=12の条件で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．

Algorithm 3 Δe 訂正処理の疑似コード

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

[15] 表 2.2 の値は近似しているため H_{32}=0 と H_{32}=6 のパタ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．

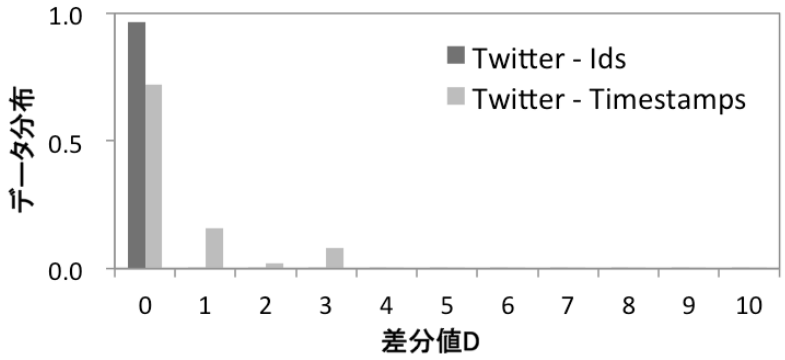

図 2.7 Twitter Public Timeline（2010 年 5 月～2011 年 4 月）の
tweet 情報の IDs と Timestamps の差分値 *D* の分布

表2.3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2.7）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2.2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．

表 2.2 索引の分岐ノードサイズ（GiB）比較

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 2.3 VAST 木の葉ノードの圧縮率

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)

索引構築時間に関してKimらによる先行研究 [16]では64M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100倍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．

2.4.1.2 VAST 木の性能評価

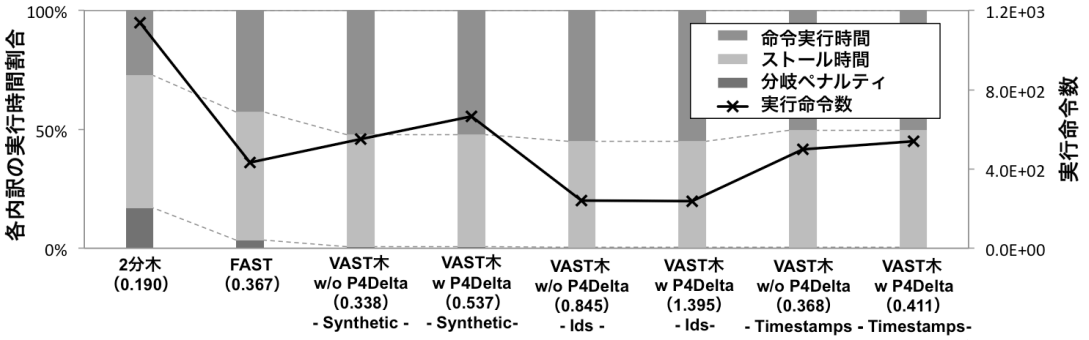

図 2.8 索引対象のデータ数が 2^{25} における完全一致探索の実行時間内訳（命令実行時間，ストール時間，分岐ペナルティ）の比較

図2.8（図中の括弧内は各実行におけるIPCを表す）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パタンの評価を行った．図からVAST木の全てのパタンで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木 w/o P4Delta）では，2.3.3.1節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木 w/P4Delta）では全てのパタンにおいて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．

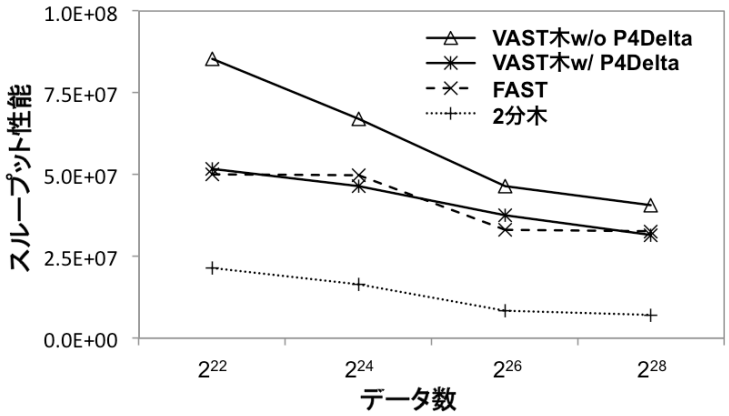

図 2.9 人工データを用いた完全一致探索のスループット性能比較

図2.9に人工データを，表2.4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木 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2.8）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木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6.0倍，FASTに対して1.24倍である．表2.4の実データを用いたスループット性能も図2.9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2.7）．この節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．

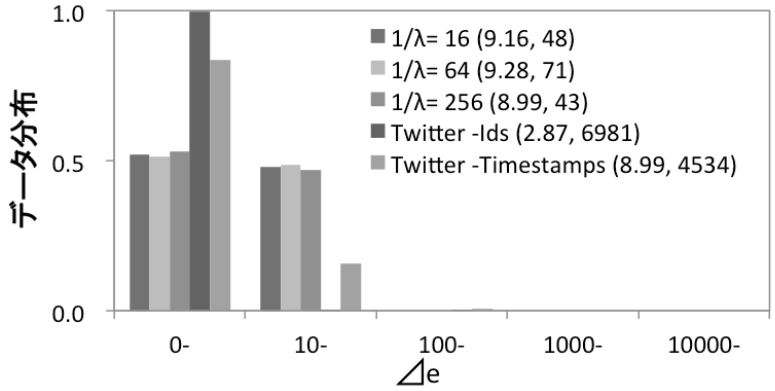

図2.10 分布の異なるデータ列の探索処理で発生する Δeの分布

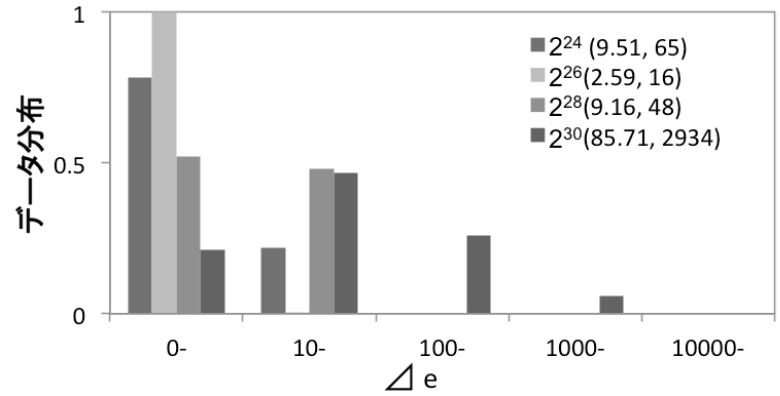

図2.11 データ数を変化（2^{24}～2^{30}個）させた場合の Δeの分布

最後に分布の異なるデータ列の探索処理で発生する Δeの分布を図2.10に，人口データの総数を変化させた場合の Δeの分布を図2.11に示す．それぞれの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． Δeが高くなるとCPUの命令数やメモリバス転送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2.10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2.3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．また実データ（図2.10）やデータ数を大きくした場合に最悪値が高くなる（図2.11）が，平均値が100以下程度に抑えられていれば図2.9と表2.4の結果から探索性能に関しては影響が小さいと判断できる．

表2.4 Twitter Public Timelineを用いた スループット性能比較（$\times 10^6$）

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

2.4.1.3 メモリバス転送帯域の消費量評価

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 [16, 19, 20]．図 2.12 に完全一致探索における平均メモリバス転送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バス転送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．

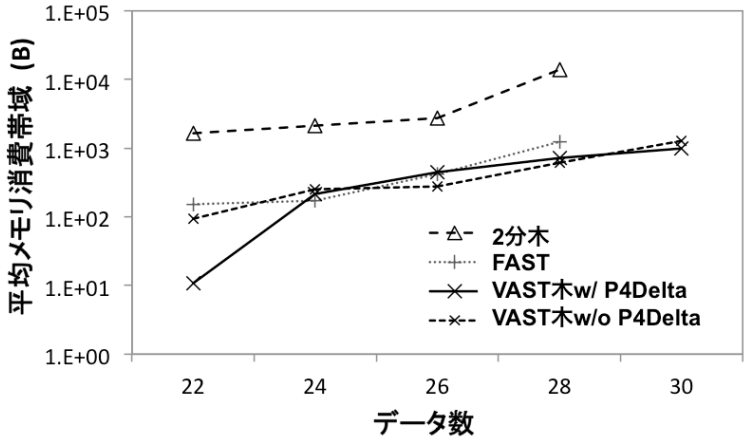

図 2.12 メモリバス転送帯域の消費量（B）比較

2.5 考察

前節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，2.5.1節でデータ分布と Δeの関係を分析してモデル化を行い，2.5.2節では最悪値 Δe（図2.10と図2.11）を考慮した訂正処理の改善方法を提案する．

2.5.1 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 2.13 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 $n_k<n_{k+1}$ 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 $key_A...key_{A+2h}$ に対応した比較キーを持つ部分木 ST_A と $key_B...key_{B+2h}$ 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 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 $n_{k+1} \leq key_q < n_{k+2}$ 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．

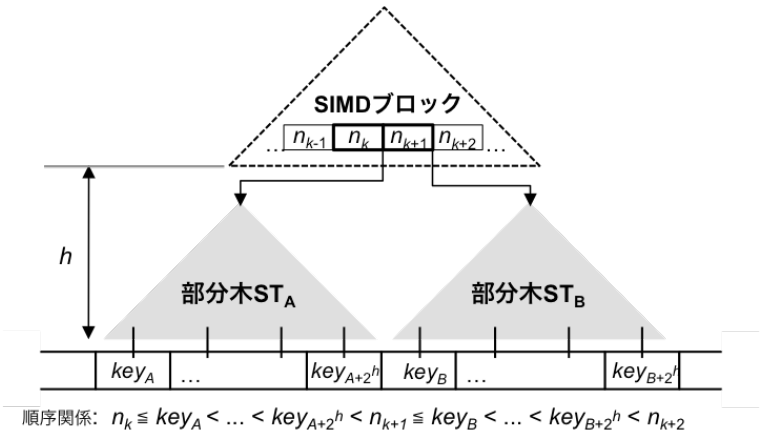

図 2.13 SIMD ブロックと葉ノード上のキー列の関係

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を p_h，探索のズレ量を de_h，木の高さを 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．

$$E = \frac{1}{H}\left(\sum_{h=1}^{H} de_h \times p_h \sum_{k=1}^{h} (1-p_{k-1})^{k-1}\right) \tag{2.1}$$

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(2.1)は以下のようになる．

$$E = \frac{1}{H_{16} + H_8} \sum_{k=1}^{\frac{H_{16}}{CH_{16}} + \frac{H_8}{CH_8}} e_k \tag{2.2}$$

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．

$$e_k = \sum_{l=1}^{\frac{CH_{16}}{SH_{16}} + \frac{CH_8}{SH_8}} de_l \times p'_k \sum_{n=1}^{l} \left(1 - p'_{n-1}\right)^{n-1} \tag{2.3}$$

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．

$$de_l = \begin{cases} 2^{H_{16}+H_8-l\times SH_{16}} & (l \le \frac{H_{16}}{CH_{16}}) \\ 2^{H_8-(l-\frac{H_{16}}{CH_{16}})\times SH_8} & otherwise \end{cases} \tag{2.4}$$

$p \times k$ 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 $p \times k$ を以下のように計算する．

$$p'_k = \frac{c_k \times r_k}{qr} \tag{2.5}$$

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．例えば圧縮処理によって下位 7bit 削除されていれば r_k は 27 となる．さらに c_k は以下のように計算する．

$$c_k = \begin{cases} 2^{H_{32}+(k-1)\times CH_{16}} & (k \le \frac{H_{16}}{CH_{16}}) \\ 2^{H_{32}+H_{16}+(k-\frac{H_{16}}{CH_{16}}-1)\times CH_8} & otherwise \end{cases} \tag{2.6}$$

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である 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．

$$r_k = \begin{cases} 2^{log_2 Y_k - 16} & (k \leq \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k > 16) \\ 0 & (k \leq \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k \leq 16) \\ 2^{log_2 Y_k - 8} & (k > \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k > 8) \\ 0 & (k > \frac{H_{16}}{CH_{16}} \ and \ log_2 Y_k \leq 8) \end{cases} \quad (2.7)$$

$$Y_k = \begin{cases} \sum_{n=1}^{H_{16}+H_8+(1-k)\times CH_{16}} X_n & (k \leq \frac{H_{16}}{CH_{16}}) \\ \sum_{n=1}^{H_8+(1-k-\frac{H_{16}}{CH_{16}})\times CH_8} X_n & otherwise \end{cases} \quad (2.8)$$

図 2.14 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 2.4 節と同様に 1/λ 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 2.5.2 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．

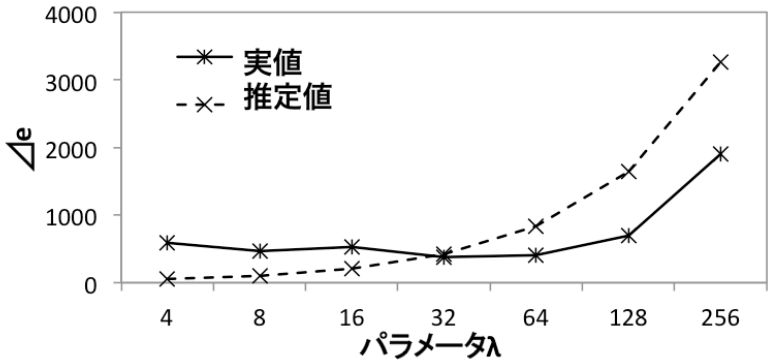

図 2.14 Δe のモデルによる推定値と実値の比較

2.5.2 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．$pos \times q$ 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 2.15 で示すように $pos \times q$ 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 2.5.1 節の分析から $range_{\Delta e}$ は以下のように計算する．

$$range_{\Delta e} = t \times \sqrt{V(E)} \tag{2.9}$$

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．

$$P\left(|E - Ex(E)| \geq range_{\Delta e}\right) \leq \frac{1}{t^2} \tag{2.10}$$

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 値を設定すればよいことが分かる．上記の確率が 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 2.5 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 で 18.5%の時間がそれぞれ節約できていることが分かる．

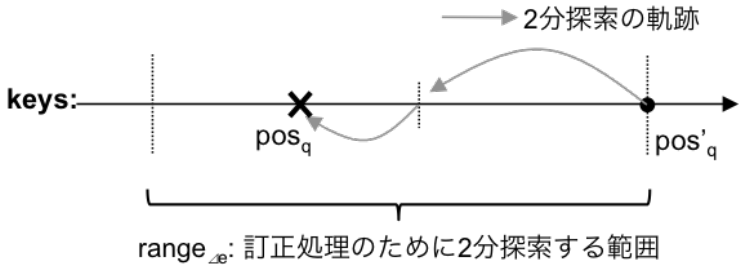

図 2.15 最悪値 Δe を考慮した 2 分探索による訂正処理の概要

表 2.5 最悪値 Δe の平均探索時間比較

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

2.6 関連研究

先行研究 [14]において DBMS 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，DBMS 技術において広く用いられる圧縮や結合操作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 [21, 22, 23, 24, 25, 26]．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 [27]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．

2.6.1 ハードウェア最適化された索引技術

T-tree [28]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 [28, 29, 30, 31]．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 [28, 29]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 [23]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バス転送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 [16, 26, 33]．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 [34, 35]されており，FAST においてもこれらの環境を用いた評価 [33]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 [36]．ハードウェアを考慮した最適化の有無でメモリ上の処理にもかかわらず平均 24，最大で 53 倍の性能差が発生するという報告 [11]があり，大規模なデータ処理で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．

2.6.2 索引構造の圧縮技術

索引構造の圧縮に関しては，DBMS 研究の黎明期に提案された Prefix/Suffix truncation や NULL suppression [17]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 [18]と VSEncoding [37]を紹介する．2.3.3 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 [38]．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形式としては任意の手法が適用可能である．

2.7 本章のまとめ

本章では大規模なデータに対して空間コストが低く，CPU の実行効率が高いメモリ上の索引構造である 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な削減に，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ながら，アルゴリズムの CPU 実行効率を高めることはデータ規模が増大傾向にある分析処理においては今後も重要な要素技術である．

第 3 章 メモリ上の効率的なデータ表現

3.1 はじめに

データのサイズ削減という観点で圧縮は基本的でかつ重要な要素技術として幅広い分野で活用されている．LZ77 は可逆圧縮手法の中で最も有名で，UNIX で用いられる gzip[16]や CPU に最適化された Snappy[17]などの代表的な実装が存在する．LZ77 では入力データ列を先頭から順番に処理し，現在符号化を行おうとしている位置から始まる部分データ列が，それ以前に出現していたかを探索する．出現している場合は，過去の出現位置を示す参照に置き換えることで圧縮を行う．

列指向 DBMS [39]や検索エンジン [40, 41]などの処理系では，圧縮技術はデータサイズ削減に加えて処理速度の改善にも活用されている．特に特定の条件を満たす部分だけを必要とする場合に，任意位置の部分データ列だけを復元（ランダム参照）する操作は時間と空間の両方の処理コストを改善可能である．例えば，以下のような事例が考えられる．

- あるサービスがユーザの日々の活動（参照時間，参照 URI，ユーザが生成したコメントなど）のデータを記録・圧縮して保管していて，サービス管理者がある期間に含まれる特定のデータ列を含むユーザ情報を抽出して分析を行いたい場合．
- サーバが日々の参照情報や負荷状況をログとして記録・圧縮して保管していて，障害は発生した場合に特定のエラー情報を含むログを探索することで原因の究明したい場合．

しかし LZ77 を用いた場合には，圧縮前のデータ列と圧縮後の符号列の位置情報に関する対応関係を保持していないため，任意位置の部分データ列だけを復元するには先頭から復元対象の部分データ列まで全てを処理する必要がある．そのため元のデータ長を N とした場合に $O(N)$の計算量が必要になる．データ長 N に依

[16] http://www.gzip.org/

[17] http://code.google.com/p/snappy/

存せずに圧縮データの効率的なランダム参照を実現するためには 2 つの異なるアプローチが考えられる．1 つ目の単純なアプローチは，圧縮対象のデータ列を固定長 k のブロックに分割をして各ブロックを既存手法で圧縮を行う方法（Block-based Compression，以下 BC）である．データ列の先頭から i 番目に位置する部分データ列を復元する場合には，i/k 番目のブロックのみを復元して $i\%k$ 番目に位置する部分データ列を返却すれば良いため計算量は $O(k)$である．この手法は最も単純であるため検索エンジンで有名な Lucene でも活用されている．2 つ目のより洗練されたアプローチは，近年提案された任意位置の長さ m の部分データ列を $O(m)$で復元可能な LZEnd [42]を用いる方法である．LZEnd は LZ77 を参考にした手法で，簡潔データ構造を用いることでランダム参照を可能にしている．簡潔データ構造は 2000 年以降に離散アルゴリズム分野で主に研究されている手法で，データ構造のサイズを情報理論的下限に漸近させながら，データの明示的な復元をせずに列挙や探索などの問い合わせを可能にする方法である．木構造や索引など様々なデータ構造において簡潔データ構造を活用する方法が既に多く提案されている [43, 44, 45, 46].

しかし上記の既存手法は以下に挙げる課題が存在する．

- BC にはよく知られたブロック長 k における圧縮率と性能のトレードオフが存在する．ブロック長 k を大きくすれば圧縮率は高くなるが，部分データ列の復元時間と空間コストが増大する．一方 k を小さくすると処理コストは小さくなるが，圧縮率は低くなる．最適な k 値は利用用途に依存するため，ユーザが最適値を決定することは一般的に難しい．
- LZEnd では再帰処理と簡潔データ構造を用いて部分データ列の復元を行うが，$O(m)$回の再帰処理で発生する簡潔データ構造の問い合わせ処理による CPU 上のペナルティ（実行命令数やキャッシュミス回数の増大）が復元性能を低下させる．

図 3.1 に CPU（Xeon X5670[18]）における参照パタンと参照速度の関係を示す事前実験の結果を，図 3.2 に圧縮していないデータの部分データ列の参照に対する LZEnd の実行命令数と L1 キャッシュミス回数の相対値を示す．図 3.2 中の横軸は参照サイズ *m*，縦軸は圧縮していないデータの同等の操作に対する相対値をそれぞれ示している．図 3.1 では参照サイズと間隔をそれぞれ変化ながら評価を行った．L2 キャッシュ（1.5MiB）を超えた場合に，最大 40 倍の差（三角実線）が発生している．この事前実験の結果や先行研究による報告 [11]により，CPU の最適化の有無でメモリ上の処理にもかかわらず 2 桁以上の性能差が容易に起こることが分かる．図 3.2 は圧縮していないデータ列の参照に対して LZEnd では 2 桁以上の実行命令数とキャッシュミス回数が発生することを示している．3.2.2 節では，これらの原因である LZEnd における再帰処理について詳述する．

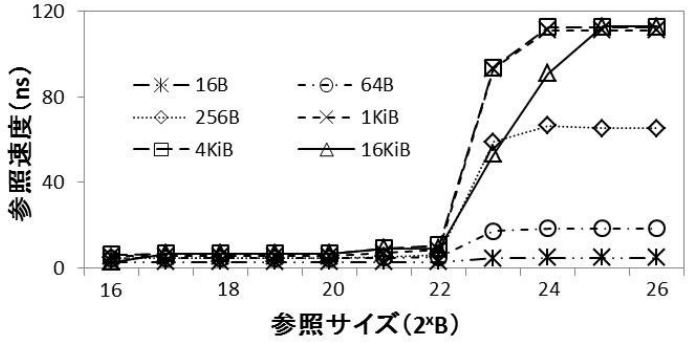

図 3.1 Xeon X5670 における参照パタンと参照速度の関係

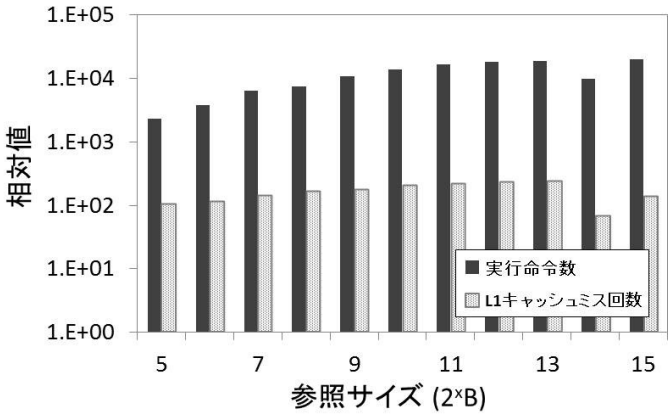

図 3.2 LZEnd における実行命令数と L1 キャッシュミス回数の相対値

[18] L1（data），L2，LL（Last Level）キャッシュはそれぞれ 129KiB，1.5MiB，12MiB である．

本研究の目的は BC のように圧縮率や性能が用途に依存するパラメータを持たない高速な復元を可能にする手法 LZE++を提案することである．LZEnd の復元処理では簡潔データ構造による複雑な問い合わせ処理で発生するメモリ上のランダム参照が低い参照性能の原因である．一方で LZ77 をベースとしている LZEnd は前提として冗長的な部分データ列の繰り返しを圧縮する手法である．そこで頻出する部分文字列データはシンプルなデータ構造として保持することで復元性能の改善を図る．具体的には頻出パタンを保持した共有辞書を用いることで LZEnd の再帰処理の枝刈りを行い，LZEnd と同等の計算量かつ高速な復元処理を実現する．さらに復元済みのデータを参照して $O(m)$回の再帰処理を軽量な繰り返し処理で置換する最適化も併せて適用する．辞書作成時間が圧縮処理のオーバヘッドとなることと，辞書サイズが復元処理時に集中して参照されることから性能に影響することが懸念される．そこで固定長 k でサンプリングした部分データ列に対して頻出パタンの列挙手法である PrefixSpan [47]を適用し，頻出語のみを共有辞書に含めることで圧縮を行う．復元しようとしている符号列が共有辞書内の部分データ列を示している場合に，以降の再帰処理を枝刈りすることができる．また一定以上のデータ列を復元した場合に，復元をしようとする符号が既に復元済みのデータ列を参照している場合がある．この復元済みのデータ列を参照することで $O(m)$回の再帰処理を平均（$mK/N+1$）回（$K \leqq N$，N は入力データ長で K は因子化のブロック数）の軽量な繰り返し処理で置換する．

本研究の学術・社会的な貢献は以下である．

- 圧縮されたデータ列に対して，用途に依存したパラメータを持たず高速なランダム参照を可能にする LZE++を提案する．共有辞書を用いた再帰処理の枝刈りと，平均（$mK/N+1$）回の軽量な繰り返し処理により LZEnd と同等の計算量かつ高速な復元処理を可能にする．
- LZEnd に対する簡単な改良で，メモリ上の処理にもかかわらず 64KiB 以下のデータの復元処理で 3.3～58.2 倍の，64KiB より大きなデータの復元処理で最大 1439.0 倍の大幅な改善が可能であることを示す．

- LZE++では最長共通接頭辞の探索範囲を限定したことで一部の条件で圧縮率が悪化するが，圧縮率とランダム参照の性能トレードオフの観点でLZEnd に対して依然としてLZE++が有利であることを示す.

3.2.1 節では LZEnd における因子化と実装を概説して，図 3.2 の原因である LZEnd の再帰処理を明示する．3.3 節では，提案手法における共有辞書の作成方法，辞書を用いた再帰処理の枝刈り方法，$O(m)$回の再帰処理を平均（mK/N+1）回の繰り返し処理に置換する方法をそれぞれ説明する．3.4 節で実験結果を示し，最後の 3.5 節と 3.6 節で関連研究と結論を述べる.

3.2 前提知識：LZEnd

LZEnd では入力データ列 $T[0...N\text{-}1]=t_0t_1...t_{N-1}$（$t_i \in \Sigma$, $0 \leqq i < N$）を因子化した $Z[0...K\text{-}1]=z_0z_1...z_{K-1}$（$K$ は因子化のブロック数で，$K \leqq N$）の計算を次の手順で行う．Σは任意のデータ集合で，文字データの場合には文字集合である．$T[0...i\text{-}1]$を $Z[0...p\text{-}1]$と因子化済みであるとして，因子化済みの列を以降では history と呼ぶ．$Z[0...q]$（$q<p$）の接尾辞と共通な最長接頭辞 $T[i...i+l\text{-}1]$を $T[i...N\text{-}1]$から探索する．結果として $T[i...i+l]$を z_p（$=z_qt_{i+1}$）とする．ここで $T[i...i+l\text{-}1]$に対応した history における z_q を source と呼ぶ．この変換処理を $T[N\text{-}1]$まで繰り返すことで因子化を完了する．LZEnd における因子化は最後の z_{K-1} を除く全ての要素が異なるため，この因子化は粗く最適（coarsely optimal）である．因子化が粗く最適である場合は以下の定理が成り立ち，符号化後のデータサイズは情報理論下限値に漸近することが知られている [48].

Theorem1. データ列の因子化が粗く最適である場合，圧縮率 $\rho(T)$ と k 次経験エントロピー $H_k(T)$ [49] との差は $|T|$ をパラメータとする式で表され，$|T|$ の値が大きくなると漸近的に 0 になる．つまり $\forall k$ に対して $lim_{|T| \to \infty} fk(|T|)=0$ s.t. $\rho(T)<=H_k(T)+f_k(|T|)$ を満たす f_k が存在する.

3.2.1 前提知識：LZEnd のデータ構造

LZEnd では因子化した Z に対して，符号列の内部表現として以下の 3 つのデータ構造を保持する.

- $c[0...K\text{-}1]$：Z の各要素における末尾記号列
- $x[0...K\text{-}1]$：参照する z_q（source）の添字列
- $B[0...N\text{-}1]$：c に格納した記号の位置を記録した bit 列

例えば $z_p=z_q t_i+1$ を考えた場合，$c[p]=t_i+1$，$x[p]=q$ として $B[i+1]$の bit を立てることで表現される．B は簡潔データ構造として表現することで，次に説明する rank と select の問い合わせを実現する．$\mathrm{rank}_B(i)$は $B[0...i]$の範囲にある bit の総数を返し，$\mathrm{select}_B(i)$は $i+1$ 番目に立てられた bit の添字を返す．rank と select を $O(1)$や $O(\log N)$で効率的に処理する手法が様々提案されている．例えば，立てられた bit が疎な B に対して NH_0に圧縮が可能な手法 [50]が存在する．結果的に z_pの長さ l は B に対して $\mathrm{select}_B(p+1)-\mathrm{select}_B(p)-1$ として計算できる．

3.2.2 LZEnd の実装

Algorithm 1 復元処理 extract(base, m) の擬似コード

```
1:  /* IN – base: start position to decompress */
2:  /* IN – m: length */
3:  /* OUT – extracted symbols in T[base...base+m-1] */
4:  if n > 0 then
5:     end = base + m - 1;
6:     r = rank_B(end);
7:     if B[end] == 1 then
8:        extract(base, m - 1);
9:        Append c[r] to an output;
10:    else
11:       pos = select_B(r) + 1;
12:       if base < pos then
13:          extract(base, pos - base);
14:          m = end - pos + 1
15:          base = pos;
16:       end if
17:       ref = select_B(x[r+1]) - select_B(r + 1) + base + 1;
18:       extract(ref, m);
19:    end if
20: end if
```

LZEnd における復元処理を Algorithm 1 に示す [42]．extract(*base*, *m*)は $T[base...base+m\text{-}1]$の部分データ列を復元することができる．$B[end]$の bit が立っている場合（7 行目），対応する位置の記号は末尾記号であるため $c[r]$を出力する（9 行目）．LZEnd では最長接頭辞 $T[i...i+l\text{-}1]$が history の接尾辞 z_q（source）に対応するように因子化を行うため，参照している source の位置を計算して extract

を呼び出すことで $T[i...i+l-1]$の復元を再帰的に行う（9～16 行目）．1 つの記号を復元するために 1 回以上の extract を呼び出すため，結果的に任意位置の長さ m の部分データ列を復元するためには $O(m)$回の extract の再帰処理が必要になる．

3.3 提案手法:LZE++

3.3.1 設計概要

この節では LZE++における共有辞書を用いた再帰処理の枝刈りと，復元済みのデータを参照して $O(m)$回の再帰処理を単純な繰り返し処理で置き換える手法の概略を説明する．$T[0...N-1]$から固定長 k でサンプリングした部分データ列に対して頻出パタン列挙法である PrefixSpan [47]を用いて圧縮する共有辞書 $S[0...M-1]$（$M<<N$）の詳細は 3.2 節で説明する．

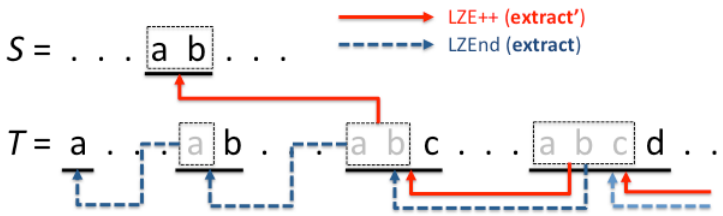

図 3.3 LZE++と LZEnd の復元処理の動作の概略

データ列の因子化の際に最長共通接頭辞が共有辞書内で探索された場合，共有辞書内の出現位置を示す参照に置き換えることで，この参照を復元する際に以降の再帰処理を枝刈りすることが可能になる．この共有辞書を用いた再帰処理の枝刈りの概要を図 3.3 に示す（灰色の記号は過去の出現位置を示す参照に置き換えられ，太字の記号は c に格納されている．図中の矢印が復元処理関数 extract の 1 回の再帰的な呼び出しをそれぞれ示している）．図中右下の abcd を LZEnd で復元する処理を考える．d は c に格納されているためそのまま出力に書き出し，abc は参照で置き換えられているため extract を再帰的に呼び出す．参照先でも同様の処理を最後まで繰り返して，最終的に 3 回の extract の再帰的な呼び出しを行う．LZE++では，共有辞書上の ab への参照が含まれているため共有辞書内のデータをそのまま出力することができる．そのため以降の再帰処理を枝刈りするこ

とができ結果的に 2 回の呼び出しで復元を完了することができる．LZE++における共有辞書を考慮した復元処理関数を以降では extract'と呼ぶ．LZEnd の再帰処理では x に記録されている参照を後方に辿る処理（ランダム参照）が繰り返し発生するため，CPU のキャッシュミス回数が増大して復元速度の遅延が発生する．LZE++では PrefixSpan で圧縮した共有辞書を用いて再帰処理を枝刈りしてランダム参照を抑制すること，さらに全ての復元処理で CPU のキャッシュサイズ程度の辞書を共有することで CPU の実行効率改善を図ることができる．

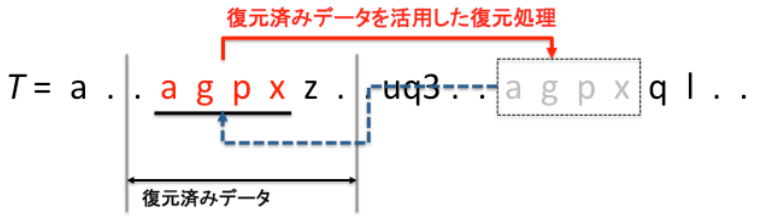

図 3.4 単純な繰り返し処理による復元方法（extractseq）の概略

一定以上のデータ列を復元した際に，復元をしようとする符号が既に復元済みの部分データ列を参照している場合がある．そこで LZE++の因子化で最長共通接頭辞の探索範囲を *wlen* に限定し，*wlen* を超えた処理は必ず復元済みのデータを参照している特性を利用して単純な繰り返しで復元処理を置き換える．この処理の概要を図 3.4 に示す．復元を行う部分データ列の先頭から *wlen* の範囲は extract’で処理を行い，*wlen* を超えた範囲のデータ列の復元処理は復元済みの部分データ列（図中では agpxz）を参照しながら復元を行う．LZE++における再帰処理を除いた復元処理関数を以降では extractseq と呼ぶ．CPU 命令を活用することで復元済みの部分データ列の位置を計算する処理の最適化も併せて行う．3.4 節では単純な繰り返しによる符号列の復元方法と CPU 命令を用いた最適化の方法に関して詳述する．

3.3.2 ランダム参照の高速化のための共有辞書

LZE++で使用する共有辞書の作成方法を示す．3.3.1 節で説明した通り，共有辞書 $S[0...M\text{-}1]$（$M<<N$）は枝刈りが効率的に行えるように頻出パタンを含みながら，CPU のキャッシュサイズ程度に小さく構築する必要がある．また共有辞書

の作成時間は圧縮処理のオーバヘッドになるため，本手法では固定長 k でサンプリングしたデータ列に対して PrefixSpan [47]を適用することで圧縮を行う．まず入力データ列 T（長さ N）から r%の領域 B（長さ L）を取得するために，位置 0 から等間隔で 0, kN/L, $2kN/L$, …と L/k 回長さ k の部分データ列をサンプリングする．PrefixSpan は入力データ列から指定された条件（長さが α 以上で β 以下，かつ頻度が ξ 回以上）を満たすパタンを高速に列挙する手法である．

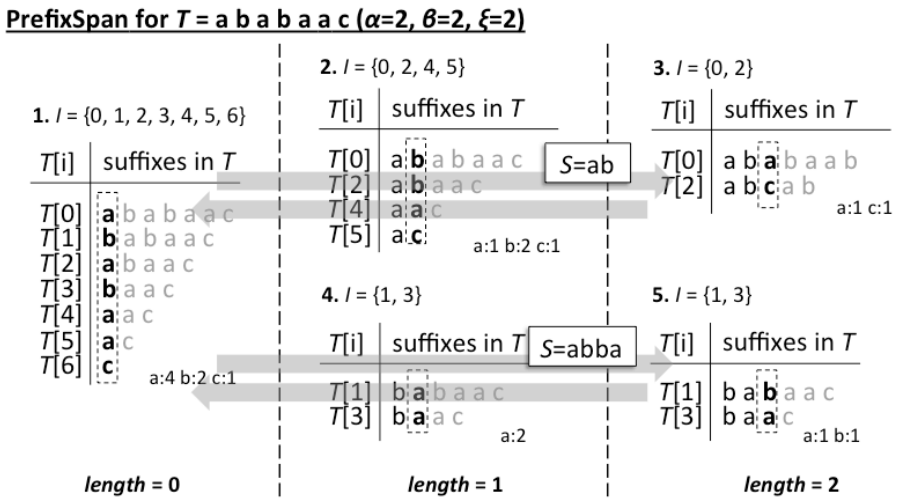

図 3.5 ababaac に対する PrefixSpan（α=2, β=3, ξ=2）の例

Algorithm 2 に PrefixSpan を用いた共有辞書を作成するための擬似コードを，図 3.5 に入力データ列 ababaac を例に共有辞書を作成する具体例をそれぞれ示す．I は入力データ列 B（長さ L）に対する接尾辞に対応した添字の集合 $B[i...L\text{-}1]$（$i \in I$）を表し，$\{0, 1, ..., L\text{-}1\}$ と初期化する．7～15 行目のループ処理において，各データ集合 Σ（$s \in \Sigma$）の要素 s 毎に深さ優先（13 行目の再帰処理）で接尾辞間の共通接頭辞の（対応する添字の）部分集合を探索する．結果的に長さ *length* が α 以上で β 以下（10 行目と 4 行目）かつ部分集合の大きさが ξ 以上（9 行目）となる部分集合 I'_s を計算する．結果として得られた $B[i...i+length]$（$i \in I'_s$）を S に追加する（11 行目）．また既に S に abcde が含まれている場合，最長共通接頭辞を探索する因子化では abcd が冗長になるため 9 行目の条件（$B[i...i+length+1] \in S$）で不要なデータ列を排除することで共有辞書のサイズ削減を行う．図 3.5 では長さが 2 以上で 3 以下（α=2, β=3）かつ出現頻度が 2 以上（ξ=2）のパタンを探索する

具体例を示す．まず接尾辞に対応した添字の集合 I を{0, 1, ..., 6}と初期化して，深さ優先で a, aa, aaa, aab, ...と探索を行う．深さ 1 の ab に対応する接尾辞の添字は{0, 2, 4, 5}で部分集合の大きさが 2 以上で，深さ 2 で条件を満たす接尾辞が存在しなくなるため ab を S に追加する．同様に ba が条件を満たすため S に追加することで S={ab, ba}を得る．最終的に得られた集合 S 内のデータを連結することで abba を共有辞書とする．

Algorithm 2 共有辞書作成 PrefixSpan(I, length) の疑似コード

```
1:  /* IN – I: indices of T and initially I = {0...N-1} */
2:  /* IN – length: length of common prefixes among suffixes */
3:  /* OUT – S: set of frequent patterns (shared dictionary) */
4:  if length > β then
5:      Return;
6:  end if
7:  while s ∈ Σ do
8:      I'_s = {i || i ∈ I ∩ T[i + length] == s};
9:      if ||I'_s || ≧ ξ then
10:         if length > α && T[i...i + length + 1] ∉ S then
11:             S = S ∪ {T[i...i + length] || i ∈ I'_s };
12:         end if
13:         PrefixSpan(I'_s , length + 1);
14:     end if
15: end while
```

3.3.3 LZE++のデータ構造

LZE++と LZEnd の因子化（3.2.1 節）の違いは最長共通接頭辞の探索で hisotry と共有辞書の両方を用いる点である．LZE++では共有辞書内の部分データ列への参照を表現するために，LZEnd と内部表現に加えて以下のデータ構造を保持する．

- S[0...M-1]：共有辞書のデータ列
- x'[0...K-1]：対応する共有辞書 S の添字列
- R[0...K-1]：共有辞書への参照を表すフラグ bit 列

例えば z_p が共有辞書を参照している場合（z_p=S[j...j+l-1]t_{i+l}），c[p]=t_{i+l}, x'[p]=j として B[i+l]と R[p]のそれぞれに bit を立てることで表現される．結果的に LZE++では復元処理のはじめに z_p が過去の source（z_q, q<p）である Z[x[p]]と，共有辞書 S[x'[p]]のどちらを参照しているか R を使って判断することで復元処理を行う．この処理に対応した簡単な改良（Algorithm 3）を，LZEnd の extract（Algorithm 1）の 17～18 行目に加えることで再帰処理の枝刈りを可能にする．

Algorithm 3 LZEnd に対する改善のためのコード（extract’）

```
1: if R[r] == 1 then
2:     ref = x’[rank_R(r) + base - pos];
3:     Append S[ref...ref + m] to B;
4: else
5:     ref = select_B(x[r + 1]) - select_B(r + 1) + base + 1;
6:     extract(ref, m);
7: end if
```

また共有辞書を用いた LZE++における因子化も LZEnd と同様に以下の定理が成り立つため粗く最適である．

Theorem2. Z[p]と Z[p']（p<p'）を考えた場合，Z[p']は次の 2 通りのいずれかの方法で因子化が行われる．最長共通接頭辞が history から探索されて Z[p]c（c∈Σ）となる場合と，共有辞書から探索されて S[i...i+k-1]c（i+k-1<M）となる場合である．前者では Z[p']は Z[p]より長く異なる要素になり，後者では最長共通接頭辞が共有辞書で探索されるため全ての history の要素と異なることが保証される．結果として最後の要素を除き互いに異なる因子が得られる．

3.3.4 復元済みのデータ列を利用した高速化

この節では extractseq における単純な繰り返しによる符号列の復元方法と CPU 命令を用いた最適化を説明する．LZE++の因子化で最長共通接頭辞の探索を *wlen* に限定して，*wlen* を超えた復元処理は必ず復元済みのデータを参照していることを利用して単純な繰り返しで復元処理を置き換える．結果として *T*[*base...base+m*-1]（*m>wlen*）を復元する際に，*T*[*base...wlen*-1]を extract' で，*T*[*base+wlen...base+m*-1]を extractseq でそれぞれ復元する．また探索を *wlen* に限定したことで，LZE++の *x* の値を絶対値ではなく最大値 log(*wlen*)-bit の相対値で表現することが可能になる．extractseq では CPU 命令を活用することで LZE++の内部表現である *B* と *R* の bit 列から高速に対応する復元済みの部分データ列の位置を計算することができる．この最適化により LZEnd で記号単位の再帰処理が，因子単位で効率的に復元することが可能になる．

Algorithm 4 復元処理 extract_seq(base, m) の疑似コード

```
1:  /* IN – base: start position to decompress */
2:  /* IN – m: length (m > wlen) */
3:  /* OUT – extracted symbols in T[base + wlen...base + m - 1] */
4:  r1 = rank_B(base + wlen);
5:  r2 = rank_R(r1);
6:  pos = select_B(r1) + 1;
7:  while pos - base < m do
8:     l = Count leading zeroes from pos in B
9:     if l != 0 then
10:       if R[r2] == 0 then
11:          r3 = r1 - x[r1] + 1;
12:          ref = select_B(r3);
13:          Copy T[ref...ref + l - 1] to an output;
14:       else
15:          ref = x'[r2++];
16:          Copy S[ref...ref + l - 1] to an output;
17:       end if
18:    end if
19:    Print c[r1++] to an output;
20:    pos += l + 1;
21: end while
```

Algorithm 4 に *T*[*base*+*wlen*...*base*+*m*-1]を extractseq で復元する処理を示す．*T*[*base*...*wlen*-1]のデータ列は extract'で事前に復元済みであることを前提とする．初期化処理として復元処理の開始位置を示す変数（*r1*, *r2*, *pos*）を計算する（5～7 行目）．7～21 行目の 1 回の繰り返しで 1 つの因子化のブロックを復元する．まず始めに現在復元しようとしているブロック長を，*B* 上の連続する 0 を数えることで計算する（8 行目）．連続する 0 を数える処理は一般的な CPU 上で広く実装されている bit 操作のための命令を活用することで効率的に処理を行うことができる．今回の評価実験で使用する環境 x64 の CPU では，64bit のレジスタに *B* 上の bit 列をロードして bsr 命令で連続する 0 を効率的に数えることができる．もし 0 が連続する場合（*l* != 0），復元済みのデータ列か共有辞書のどちらかからデータ列を復元する．復元済みのデータ列を参照している場合（*R*[*r2*] = 0），単純に参照しているデータ列を出力する（14 行目）．ここで *x* は上述したように現在位置からの相対的な位置で示される参照であるため，select$_B$ で絶対値に変換する処理が必要になる（13 行目）．共有辞書を参照している場合（*R*[*r2*] = 1），参照している共有辞書上のデータ列を出力する（16～17 行目）．最後に対応する末尾記号を出力することで，1 つのブロックの復元を完了する．この最適化に

より復元処理における $O(m)$回の再帰処理を平均（$mK/N+1$）回の軽量な繰り返しで置き換えることが可能になり，結果として以下の 3 つの改善が可能になる．

- $K/N<1$ である場合に繰り返し回数の削減
- 再帰処理の除去による CPU 効率の改善
- 1 回の繰り返しでの rank/select の呼び出し回数の削減

3.4 評価実験

この節では LZE++と LZEnd のランダム参照性能と圧縮に関する基本性能（圧縮率と圧縮時間）の評価を行う．まず 3.4.1 節で評価に用いた LZE++と LZEnd の実装に関して詳述し，3.4.2 節で実験環境について説明する．

3.4.1 評価実験で用いる実装

まず LZEnd の各内部表現のデータ構造（3.2.2 節）について説明する．c には末尾記号を格納するため 1B の配列，また x には対応する source の添字を絶対値で格納するため 4B の配列でそれぞれ表現する．bit 配列 B は疎であることから効率的に圧縮できることが知られているため [42]，rank と select の問い合わせが可能な圧縮表現である RecRank [50]を用いる．結果として LZEnd の因子化のブロック（c，x，B）の最大サイズは 41bit である．圧縮処理は LZEnd の論文を参考に Suffix Array を含むデータ構造を事前に構築することでデータ列の因子化を行う．

LZE++では c と B は LZEnd と同様の表現を用いる．x は 3.3.4 節で説明したように log($wlen$)-bit の相対値で格納するため，$wlen$ を 65,536 として 2B の配列で表現する．共有辞書のデータ列 S は 1B の配列，因子化の各ブロックが参照する共有辞書 S の添字列は絶対値で保持するため 4B の配列で表現する．3.3.2 節で説明した PrefixSpan は入力データ列 T の 1%（r=1）の領域を k=1024 の固定長でサンプリングを行った部分データ列 B に対して適用した．また PrefixSpan の各パラメータは事前実験によりランダム参照性能への影響が小さかったことから（α，β，ξ）をそれぞれ（8，256，8）と固定して評価実験を行った．LZE++の因子化のブロック（c，S，x，x'，B，R）の最大サイズは 58bits であるため，8B（64bits）以上の部分データ列を参照することができれば圧縮率に寄与できるため α=8 と

設定した．圧縮処理では共通接頭辞の探索範囲を *wlen* に限定しているため，LZ77 系の実装で一般的に用いられる hash 表を使用した．このことから最長接尾辞を探索できないため，実装上は LZE++の因子化が粗く最適（3.3.3 節）にはならない．しかし圧縮処理で LZEnd と同様の Suffix Array を用いた処理と $O(m)$回の再帰処理を軽量な繰り返しで置換する最適化を除くことで粗く最適な因子化を保証した実装が可能であるが，ランダム参照性能と圧縮時間を考慮した実装を本評価では選択した．

B と *R* の表現方法の違いで LZE++1 と LZE++2 の 2 つを評価実験で用いた．LZE++1 は LZEnd の実装と同様の RecRank を *B* と *R* の表現に適用して，3.4 節で説明した繰り返しの中に存在する 1 つの select_B の問い合わせを行った．事前実験より上記の select が復元性能への影響が大きかったため LZE++2 では *R* は LZE++1 と同様の表現を用いて，select_B の問い合わせ結果だけを事前計算する最適化を適用した．この事前計算による最適化は圧縮率と圧縮速度には影響しないことから LZE++2 の実装は 3.4.3 節でのみ使用する．

3.4.2 実行環境

評価実験では以下 9 個のテストデータを使用した．

1) DNA シーケンス（influenza と Escherichia_Coli）と自然文（einstein.de.txt と world_leaders）[19]
2) enwiki8[20]と gov2[21]
3) Hadoop[22]，Cassandra[23]，PostgreSQL[24]のログ

1)は冗長的で圧縮の行いやすい特徴をもつ圧縮手法のベンチマークでよく利用されるデータである．2)の enwiki8 は 2006 年 3 月 3 日の英語 Wikipedia から先頭 10^8B を抜き出したデータである．また gov2 は 2004 年初期の.gov ドメインの

[19] http://pizzachili.dcc.uchile.cl/repcorpus.html
[20] http://cs.fit.edu/mmahoney/compression/textdata.html
[21] http://cs.fit.edu/mmahoney/compression/textdata.html
[22] http://hadoop.apache.org/
[23] http://cassandra.apache.org/
[24] http://www.postgresql.org/

Web サイトを収集して集めた集合から 128MiB のデータをランダムにサンプリングしたものである．3)は各行が似た情報を含むレコード集合で，先頭から 128MiB を抜き出したデータである．3.4.3 節では LZE++と LZEnd のランダム参照性能を比較する．紙面の都合で全ての結果を記載することができないため，LZE++の圧縮率が最も高い PostgreSQL，中央の einstein.de.txt，最も低い enwiki8 の 3 つを選択した．これは LZE++の再帰処理の枝刈りや平均（mK/N+1）回の繰り返しによる処理が圧縮率（因子化のブロック数 K）に依存するためである．

3.4.5 節では圧縮に関する基本性能（圧縮率と圧縮速度）の比較を行う．LZEnd 以外に次の 6 つの既存手法を比較に用いた．gzip（Version1.4）は広く使用されている LZ77 の実装である．bzip2[25]（Version1.0.6）は Burrows-Wheeler 変換 [49]によるブロックソートを利用した圧縮手法である．gzip と bzip2 は圧縮率が最も高くなるオプション-9 を指定して評価を行った．lzip[26]（Version1.12）はスライド窓のサイズに制限のない LZ77（LZMA）で高い圧縮率で知られる実装である．評価では入力データ列の長さをスライド窓のサイズに設定した．Snappy（Revision73）と LZ4（Revision90）は LZ77 の高速な実装である．最後の Re-Pair[27] [51]は文法を考慮した圧縮手法で，冗長的なデータの圧縮に向いた特性をもつ．

本評価は Xeon X5670 のサーバを用いて実施した．Xeon X5670 は L1，L2，LL キャッシュにそれぞれ 129KiB，1.5MiB，12MiB を実装している．CPU の統計情報取得には perf v3.6.9 を使用した．評価用コードは全て C++で記述し，gcc（-O2）の v4.7.1 でコンパイルして評価を行った．

3.4.3 ランダム参照の性能評価

3.4.3.1 64KiB 以下のデータ復元処理

図 3.6（各図中の左部）に 2^5B～2^{16}B の範囲における LZE++と LZEnd のランダム参照性能を示す．使用したテストデータはそれぞれ PostgreSQL（左図），einstein.de.txt（中央図），enwik8（右図）である．全条件で LZE++の復元性能は

[25] http://www.bzip.org/

[26] http://www.nongnu.org/lzip/lzip.html

[27] http://www.cbrc.jp/~rwan/en/restore.html

LZEnd より高く，LZEnd に対して LZE++1 が 3.3～12.0 倍，LZE++2 が 8.1～58.2 倍の性能差である．3.4.2 節で説明したように $select_B$ の問い合わせ性能が LZE++ の復元速度に影響し，LZE++1 に対して 2.0～5.1 倍程度の差が発生している．

図 3.7 に各テストデータ T が共有辞書でどの程度置換されているか示す．上側が固定長 k でサンプリングしたデータ列 B を共有辞書として使用した場合の，下側がデータ列 B に対して PrefixSpan で圧縮を行った共有辞書の俯瞰図である．$T[0...N-1]$が正方形の左上から右下にかけてに対応し，黒い領域が共有辞書 S で置き換えられた部分を示している．つまり $T[i...i+l]$が共有辞書内の部分データ列を参照している場合には，対応する図中の領域が黒く示してある．PostgreSQL における置換割合が 44.57%から 71.32%に変化し，共有辞書が 63.8KiB で 4.83%に圧縮されている．einstein.de.txt における置換割合が 23.78%から 22.85%に変化し，共有辞書が 124.3KiB で 13.07%に圧縮されている．enwik8 における置換割合が 66.20%から 47.46%に変化し，共有辞書が 27.8KiB で 2.84%に圧縮されている．einstein.de.txt と enwik8 は置換割合が 0.93%と 18.74%と悪くなっているものの，全ての共有辞書が大きく圧縮されていることが分かる．

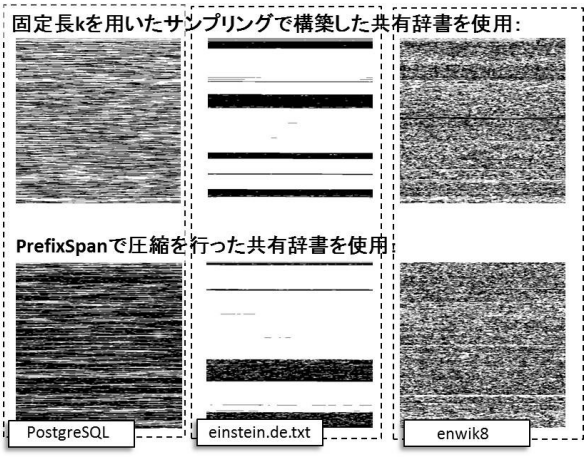

図 3.7 T 上の共有辞書で置き換えられた部分の俯瞰図

共有辞書を用いた枝刈りによる再帰処理回数と，CPU の実行効率（実行命令数と L1/LL キャッシュミス回数）の変化を図 3.8 に示す．それぞれの値は LZEnd の値で正規化した相対値で表し，図中の baseline が LZE++と LZEnd が同じ値であることを示している．再帰処理回数を削減したことで全体的な CPU の実行効率

は改善されているが，参照サイズが小さい時に LL キャッシュミスが LZEnd に対して悪化していることが分かる．しかし参照サイズが大きくなると LL キャッシュミスは相対的に改善し，最終的に 16.01%，55.59%，39.37%となる．この結果から LL キャッシュミス回数の削減は LZE++の性能の改善にあまり寄与していないことが，図 3.6 の参照サイズが小さい場合の参照性能の結果から判断できる．

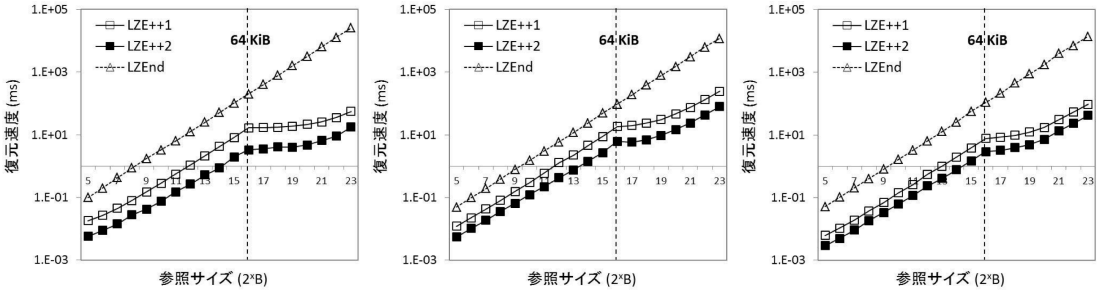

図 3.6 2^5B（32B）～2^{23}B（8MiB）における復元性能，左図が PostgreSQ のログ，中央図が einstein.de.txt，右図が enwik8 である．

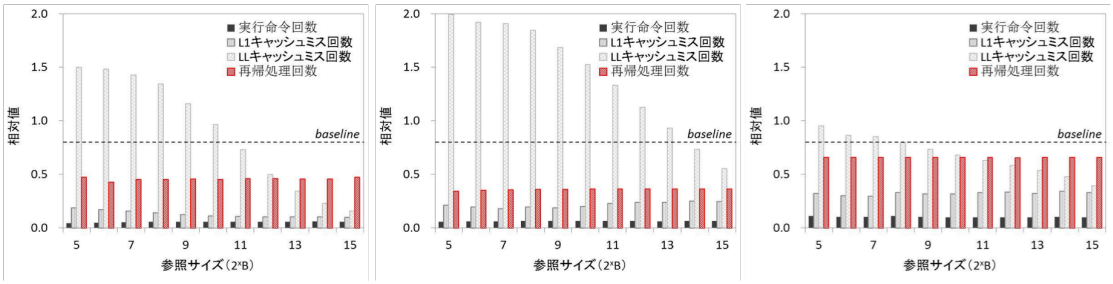

図 3.8 64KiB 以下の LZEnd に対する LZE++の実行効率の相対値

3.4.3.2 64KiB より大きなデータ復元処理

図 3.6（各図中の右部）に 2^{17}B～2^{23}B の範囲における LZE++と LZEnd のランダム参照の性能を示す．結果から明らかなように全ての条件において LZE++の復元性能は LZEnd より高く，LZEnd に対して LZE++1 が 7.3～4541.0 倍，LZE++2 が 34.5～1439.0 倍の性能差である．LZE++の 64KiB 以下の復元処理の改善と比較して非常に高く，3.3.3 節で説明した $O(m)$回の再帰処理を平均（mK/N+1）回の繰り返しで置き換えることで復元性能を大幅に改善できることが結果から分かる．

3.4.4 圧縮率と圧縮速度

表 3.1 LZE++と既存手法との圧縮率の比較

Source	Size(B)	LZE++	LZEnd	gzip-9	bzip-9	lzip	Snappy	LZ4	Re-Pair
Escherichia_Coli	112,689,515	44.91	15.9	28.91	26.98	4.35	46.91	68.21	9.60
influenza	154,808,555	13.33	5.71	11.21	6.59	0.99	38.56	53.05	3.27
einstein.de.txt	92,758,441	34.22	0.70	31.18	4.32	0.11	59.19	27.30	0.17
world_leaders	46,968,181	40.32	3.91	17.86	6.94	1.10	30.88	28.15	1.79
enwik8	100,000,000	67.49	57.42	36.52	29.01	24.78	58.35	56.98	41.05
gov2	135,268,352	66.41	45.25	32.53	27.56	18.84	51.94	49.72	31.20
Hadoop	135,268,352	10.54	9.73	5.15	2.91	2.45	12.95	8.87	4.63
Cassandra	135,268.352	**14.33**	15.85	6.18	4.28	3.17	13.85	11.08	8.43
PostgreSQL	135,268,352	**9.74**	15.89	4.65	2.76	2.57	10.98	11.32	8.83

表3.2 1MiBを復元するための要した時間（ms）の比較

Source	LZE++	LZEnd	gzip-9	bzip-9	lzip	Snappy	LZ4	Re-Pair
Escherichia_Coli	855.22	84539.0	239.32	111.47	1387.66	5.21	3.56	855.41
influenza	248.52	15620.49	88.39	188.57	891.63	4.27	2.84	498.45
einstein.de.txt	2792.74	18365.53	600.26	177.03	420.00	6.56	2.49	466.31
world_leaders	160.74	43079.97	31.26	77.69	890.91	3.35	2.68	0.45
enwik8	172.07	14611.31	66.90	108.84	886.63	58.35	56.98	41.05
gov2	152.26	27090.72	50.55	119.84	857.19	5.39	4.22	1444.45
Hadoop	205.26	14585.62	13.64	318.37	1624.11	454.80	1.71	1.01
Cassandra	171.47	15954.85	15.04	179.92	1859.69	1.94	12.40	481.16
PostgreSQL	151.62	12788.98	17.05	260.69	1552.79	1.71	1.08	470.92

この節では圧縮に関する基本性能の評価として圧縮率と圧縮速度に関して，LZEndを含めた7つの既存手法と比較したものを表3.1と表3.2にそれぞれ示す．データサイズが異なるため圧縮速度に関しては1MiBを復元するために要した時間を示す．圧縮率に関してはCassandraとPostgreSQLのログ（図中の太字）において，最長共通接頭辞の探索長を*wlen*に限定しているのにもかかわらずLZEndに対して圧縮率がそれぞれ1.52%と6.15%だけ改善している．しかし他の条件では圧縮率は低く，world_leadersで36.41%だけ悪化している．これはLZE++では圧縮処理でSuffix Arrayを用いていない点と因子化のブロックサイズが最大で17bit大きくなっている点が原因だと考えられる．他6つの既存手法を用いた評価結果は次に示す通りである．gzip，bzip2，lzip，Re-Pairのそれぞれの結果はLZEndとLZE++の圧縮率より高く，これはLZEndとLZE++がランダム参照を行うための余剰のデータ構造が原因だと考えられる．Re-Pairは冗長なDNAシーケンスと自然文に対して，

bzip2はenwik8とgov2に対して高い圧縮率を示している．SnappyとLZ4は復元速度を重視した実装であるため，圧縮率はLZEndとLZE++に近い値を示している．

圧縮速度に関してはLZE++はhash表を用いた実装であるためLZEndに対して5.9～126.8倍程度高速である．LZE++の圧縮速度はgzipと，Escherichia_Coliとeinstein.de.txtにおけるbzip2より悪い結果を示している．一方で上記の2つを除くbzip2の圧縮速度とは近い値を示している．lzipは非常に高い圧縮率である一方で，圧縮速度がLZE++と比べて非常に悪い．SnappyとLZ4の圧縮速度は全条件の中でも高く，Re-Pairはテストデータに対して一貫しない結果となった．

3.4.5 LZEnd と LZE++のトレードオフ分析

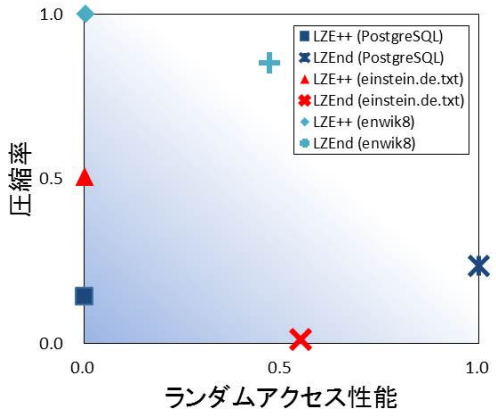

図3.9 LZEndとLZE++における圧縮率と性能のトレードオフ分析

最後に3.4.3節で使用した3つのテストデータを用いてLZEndとLZE++の圧縮率と2^{23}Bのランダム参照性能のトレードオフ分析を図3.9に示す．図中の各値は最大値を用いることで0.0～1.0に正規化し，図中の左下側が圧縮率と性能とともに良いことを意味している．図中左下（原点）からの距離という観点で比較した場合にLZE++は全ての条件でLZEndよりも良い結果となり，特にPostgreSQLの場合にはLZEndに対して圧縮率と性能の両面で圧倒的に有利であることが分かる．

3.5 関連研究

圧縮は様々な処理系 [39, 40, 41]における重要な要素技術として活用され，近年ではデータのサイズ削減だけではなく圧縮を応用した処理速度の改善手法も広く

研究されている．列指向DBMSでは表を列（属性）方向に圧縮して格納し，明示的な復元をせずに術後評価や集約操作などを直接処理可能な形式で保存することで効率化する様々な手法が提案されている [39]．これらの手法は連長圧縮（Run Length Encoding）やビットマップ圧縮（Bitmap Encoding）などの古典的な手法を応用することが多く，本研究が対象とする任意のデータ列に対して単純に適用することが難しい．また簡潔データ構造 [43, 44, 45, 46]を検索エンジンに応用する先行研究 [40]も存在するが，結果として本研究が扱っているような課題によって現段階では性能の優位性を示すことが難しいことが指摘されている．本提案手法と，LZEndに対する圧縮索引構造 [52]を応用することで，より高度で高速な処理を明示的な復元をせずに実現できる．また離散アルゴリズム分野でランダム参照が可能な文法圧縮手法 [53]が提案されているが，入力データ列Nに対して計算量が$O(logN)$であるため，本提案手法より計算量が高く実用上の課題が存在する．

3.6 本章のまとめ

処理対象のデータが増大する場合に，圧縮を用いてサイズ削減と性能改善を行うことは非常に重要である．本章では，圧縮データの長さmの部分データ列を高速に復元可能なLZE++を提案した．LZE++では共有辞書を用いた再帰処理の枝刈りと，復元済みのデータを参照して$O(m)$回の再帰処理を軽量な繰り返しで置換する最適化を行った．圧縮されたデータに対するランダム参照は非常に一般的で汎用性があるため，様々な分野において活用が可能な重要な要素技術である．

第 4 章 データ表現を考慮したコストモデル

4.1 はじめに

サービスやサーバに関する CSV，TSV，JSON などの形式で蓄積されたログデータに対して，選択操作・集約操作などの関係代数的に表現される分析処理を SQL で記述して実行することの需要は高い．SQL を DSL として実装する DBMS で採用されるデータ表現の代表として，データを行方向に配置する NSM（N-ary Storage Model）[54]と，列方向に配置する DSM（Decomposition Storage Model）[6]がある．DBMS を用いた分析処理では属性（列）が多くあり，処理中に参照される属性の数は限定的であるという傾向から DSM が採用されることが多い．代表的なものに Vertica，Vectorwise，Amazon Redshift などの製品/サービスを含め，OSS である Hive や Spark などで使用される ORCFile，Parquet の実装でも DSM を基にしたデータ配置が採用されている．DSM を前提に，圧縮技術，非同期 I/O や，ベクトル演算等を活用するアーキテクチャである列指向 DBMS である C-Store が 2005 年に提案された [55]．特に圧縮技術に関しては，ログデータの冗長性を利用して高い圧縮効果が期待できるが，クエリ実行中に圧縮された内部データを復元する必要があるため I/O だけでは無く CPU やメモリの参照コストを考慮したクエリの実行計画が列指向 DBMS における課題の 1 つとなっている．

本研究では列指向 DBMS の特徴の 1 つである明示的に復元処理を行わずに透過的に処理可能な圧縮形式 [39]に着眼して，SQL で記述されたクエリを列指向 DBMS が実行する際に発生するコストの評価と詳細な分析を行う．実行時間とコストの関係を分析することで実行計画に必要なコストモデル構築に向けた考察を行うことが本研究の目的である．先行研究 [58]では DBMS 実行時に発生するキャッシュミス回数でコストを算出する手法が提案されている．そこで本研究ではこの手法の推定精度を分析し，もし精度に問題がある場合には単純なモデルの拡張により精度を改善できないかを検討する．

取り扱う圧縮形式は列指向 DBMS で頻繁に用いられる以下３つである．

- 値毎の出現位置記録した Bit-Vector 符号化

- 順序関係を保持した辞書による符号化
- 連続値を効率的に扱う Run-Length 符号化

1 つ目の Bit-Vector 符号化では，値毎に出現位置情報である整数列を記録することで各整数列を可変長符号で圧縮する．述語評価の際には，述語の条件情報と合致する整数列のみを結果として取り出す．2 つ目の辞書による符号化では，データ内に出現する値を用いてまず辞書を作成する．そして作成された辞書内の項目番号（整数）に値を置き換え，Bit-Vector 符号化と同じように整数を可変長符号で圧縮する．述語評価の際には，辞書を作成する際に辞書内の値と対応する項目番号の順序関係を保持している特性を活用する．最後の Run-Length 符号化は，同一の値が多く連続する場合に利用される手法で，連続値を（値，繰り返し数，出現先頭位置）の組で表現することで符号化を行う．述語評価の際には，データ全体を復元せずに値部のみを線形探索することで条件情報と合致する列データを結果として取り出す.

本研究の学術・社会的な貢献は以下である.

- CPU のキャッシュミス回数でコスト T_{Mem} を見積もる手法を列指向 DBMS のプロトタイプを用いて分析を行い，このモデルだけでは発生する実行計画の優劣変化を正確に把握できないことを指摘した.
- 上記の分析結果を踏まえ，各実行計画の CPU 負荷をより詳細に分析することでコスト T_{Mem} に対して各圧縮形式に対応した CPU 計算量を表現する補正項 T_{Ins} を加算することでこの優劣変化を捉えることができるモデルを提案し，その妥当性について検証した.

4.2 節では前提とする列指向 DBMS に関する概説を行う．4.3 節では，今回評価に用いる圧縮形式に関して詳細化を行い，4.4 節では列指向 DBMS のプロトタイプを用いた評価を行うことで実行時間とコストの関係の考察を行う．最後の 4.5 節と 4.6 節で関連研究と結論を述べる.

4.2 想定する列指向 DBMS アーキテクチャ

4.2.1 列指向 DBMS の概略

本章で想定する列指向DBMSのアーキテクチャを，先行研究を参照しながら概説する．列指向DBMSでは表データを列方向に連続した領域に圧縮して格納する（図4.1）．列に対応したストレージ上の表現をC-Storeに倣いProjectionと呼称する．図4.1のidとvalueが示すように，表内の１つの列は複数のProjectionに含むことができる．例えばロード時の位置情報を維持したProjectionや，列でソートしたProjection，また適用する圧縮手法が異なるProjectionなどである．特にロード時の位置情報を維持して格納されているものはSuper Projectionと呼ばれる [56]．図4.2に示すようにProjectionは複数の行から構成される列ベクタ（Column Vector）に分割され，さらに複数の列ベクタをまとめた列ブロック（Column Block）でストレージ上に格納される．列ベクタはクエリで同時に処理される最小単位で，列ブロックはI/Oの最小単位である．更新の多いOLTPなどを想定した行指向DBMSでは1行単位でクエリの処理を行うVolcano-Style [8]に従った実装が一般的であるが，OLAPなどのような参照の多い分析処理では複数行でまとめて処理（Vector-at-a-time）を行うほうがCPUのキャッシュ効率やベクトル演算を活用する観点で有利になることが知られている [12, 56]．そのため現在のCPUのL1/L2のキャッシュサイズを考慮して列ベクタは16KiB～64KiB程度（100～1000行程度）に設定される．同様に列ブロックに関しても，OLAPの場合には処理対象のデータ領域がOLTPと比較してかなり広い領域を参照するためI/O単位である列ブロックは8MiB～32MiB程度と大きく設定されることが多い．また図4.2の各列ブロックのヘッダ（もしくはフッタ）にはブロック内に含まれるデータの最大値/最小値などの統計情報が含まれている場合があり，その統計情報を用いたクエリ実行中のI/O削減の方式も提案されている [57].

列データをProjectionとしてストレージ上に格納する際，元の行データに復元するために必要な位置情報（サロゲートキー）との組で記録する必要があり，データ肥大化の要因となる．Super Projectionでは各列ベクタにおける先頭位置情報だけを記録して，明示的に各行単位の位置情報をストレージ上に保持しないことで

対処する．処理中に位置情報が必要になった場合には，各列ベクタの先頭位置情とoffsetを用いて計算される仮想IDを用いて行データの復元が行われる．

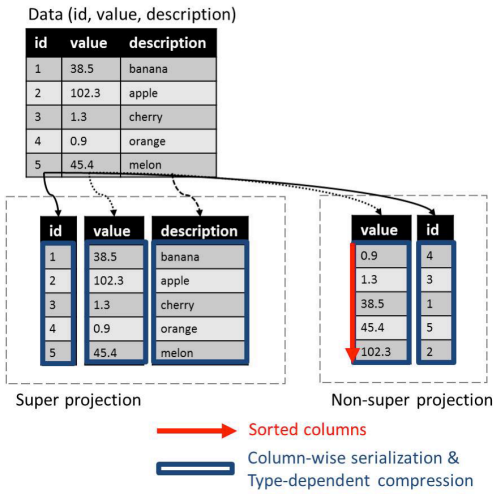

図4.1 表とProjectionの関係

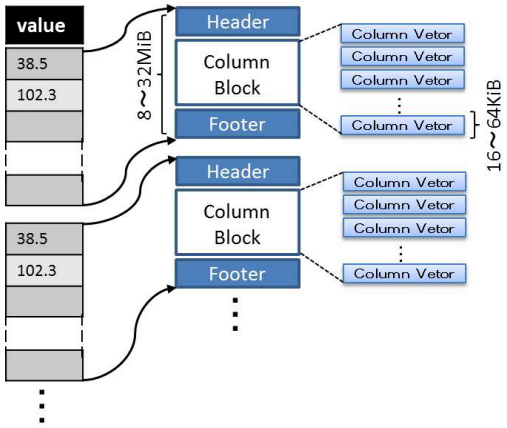

図4.2 Projectionのデータ構造

列指向DBMSでは上記の列ごとに格納されたProjectionを前提に，行指向DBMSと同様に入力されたクエリから論理プランを作成する（図4.3）．列データ毎のProjectionでストレージ上に保持されていることから位置情報を用いたInner Joinが論理プラン内で多く発生することが列指向DBMSの特徴である．図4.3では，valueとdescriptionの列データを読み取り（ColumnScan処理），値が25よりも大きいvalueと値が*abc*であるdescriptionを選択する．それらの列データの位置情報を示す仮想ID（図4.3中のvid）でInner Joinを行い，その結果とidの位置情報でさらにInner Joinをすることで最終的な結果を取得する．この論理プランに対応した複数の物理プランの中から，位置情報でのInner Joinによるコストを考慮して最も効率的なものを選択することが重要であると指摘されている [39]．次節では，論理プラン内のColumnScan処理に対応した物理プランに着眼する．

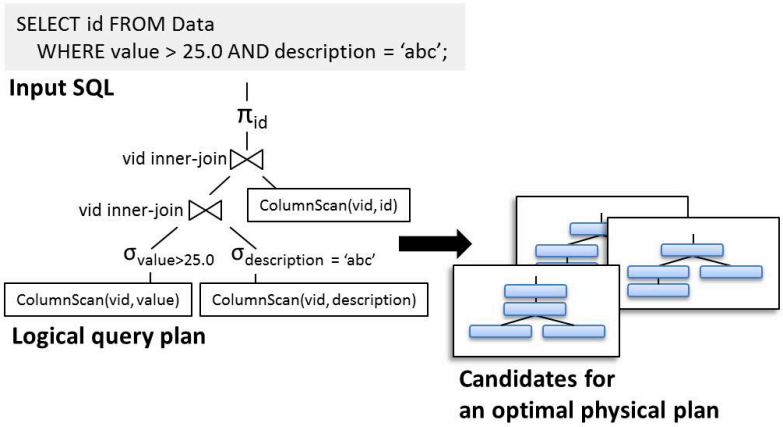

図4.3 入力クエリの最適化

4.2.2 列指向 DBMS における実行計画

論理プランにおける列データの読み取り処理はI/O処理を伴い対象となる列データをストレージから取得するSeqScan，WHERE句の条件情報を含んだ各列データの選択操作はVectorCompareFilterか，SimpleFilterに変換される．その後，選択された列データの位置情報を示す仮想IDを用いたInner JoinはPosFilterScan，PosMergeJoin，PosHashJoinのいずれかを用いて実現される．以下では本章で想定する物理プランを構成する各ノードの役割を概説する．

SeqScan

選択された列データのProjectionをメモリ上のキャッシュに読み込み，先頭の列ブロック内に含まれる列ベクタから順に上位のノードに返却する．

SimpleFilter

SeqScanから返却された列ベクタに対して，圧縮された列データをメモリ上に復元後に条件に合致する列データを抽出する．

VectorCompareFilter

SeqScanから返却された列ベクタに対して，圧縮された列データを復元せずに条件情報に合致する列データの抽出を行う．復元せずに条件情報を評価する手法に関しては4.3節で詳述する．条件に対する選択率が低く，圧縮率が高い場合にSimpleFilterに対して効率的に処理できる可能性がある．

PosFilterScan

選択された列データのProjectionに対して下位のノードから取得した結果に対応した列データの抽出と結合操作を行う．

PosLinearMerge/PosMergeJoin

下位のノードから取得した列データがSimpleFilterやVectorCompareFilterを用いて選択処理されていない場合には，PosLinearMergeで単純に先頭から結合して行データを復元する．一方選択操作が適用されている場合には，MergeJoinで結合を行うPosMergeJoinが選択される．この物理プランでは先頭から2つの列データを順に読み込み，位置情報が同じ列データで結合して行データに復元する．単純に先頭から結合操作を行うPosLinearMergeに対して，MergeJoinを行う必要のあるPosMergeJoinは処理コストが一般的に高くなる．図4.3内で用いた入力クエリを例として具体的な実行計画の一例を考える．このクエリに対応した実行計画は複数存在するが，例として図4.4の様な実行計画が考えられる．

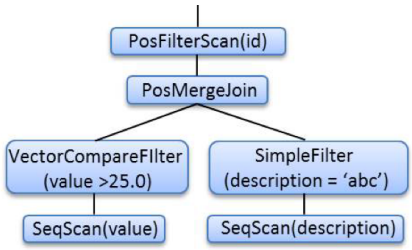

図4.4 実行計画の具体例

1) SeqScanでvalueに対応したProjectionの列ブロックを先頭から順に読み込み，VectorCompareFilterで値が25より大きい列データを選択する．
2) SeqScanでdescriptionに対応したProjectionの列ブロックを先頭から順に読み込み，SimpleFilterで値が*abc*である列データを選択する．
3) 1)と2)で選択された列データをPosMergeJoinで位置情報が同じデータを結合して行データに復元する．
4) 下位の物理プランから取得した結果と同じ位置情報を持つidの列データをPosFilterScanで読み込むことで結果を取得する．

DBMSでは1章で述べたように，主にI/O回数を最小化するようにDBMS内の処

理を抽象化した固有のコストモデルをもつ．実行計画ではストレージ上に存在する各列に対応したProjectionを考慮して，論理プランに対応した全ての実行計画の中から，論理プランを最も効率的に実行可能と判断される物理プランを探索する．探索処理には動的計画法や，探索空間が大きくなった場合には焼きなまし法などの近似手法が用いられる．次節ではVectorCompareFilterの述語評価で扱う圧縮形式に関する詳述を行う．

4.3 復元せずに評価可能な圧縮形式

この節では復元せずにWHERE句の条件情報を評価するための圧縮形式に関して説明を行う．1つ目のBit-Vector符号化は列ベクタ内に出現する値毎の位置情報を記録する符号化形式で，属性がとりうる値の集合（カーディナリティ）が小さい場合に特に有効である．一般的には，1つの値に対して集合の大きさと同じbit幅が必要になるが，集合が大きくなった場合に疎なbit列になるため本研究では位置情報をoffset値の整数で表現して実装する．2つ目の辞書式圧縮は従来の行指向DBでも広く利用される手法で，列ベクタ内に出現する全ての値を辞書として記録して，その作成された辞書内の項目番号（整数）に列ベクタ内の値を置き換えることで符号化する．辞書を作成する際に辞書内の値と対応する項目番号の順序関係を保持することで，WHERE句の条件情報の評価の際の可変長値の比較を固定長で行う最適化をしている．3つ目のRun-Length符号化（RLE）は連続値を，（値，繰り返し数，出現先頭位置）の組で置き換える符号化方式で，主にソートされているProjectionに用いられる方法である．それぞれの圧縮形式はGoogleのProtocol Buffers[28]を用いて，データの入出力を行う．以降の節では各手法の具体的な説明と実装に関して詳述する．

4.3.1 Bit-Vector 符号化

Bit-Vector符号化では列ベクタ内に出現する値毎の位置情報を整数として記録する．例えば以下の文字列からなる列ベクタを考える．

one, two, three, one, two, three, one, two, three,...

[28] https://code.google.com/p/protobuf

これを以下のように，値，値が出現する位置の情報（列ベクタの先頭を0）の可変長配列の組として記録される．

{one, 0, 3, 6, ...}, {two, 1, 4, ...}, {three, 2, 5, ...}, ...

各値の位置情報（整数列）は昇順のため，値の集合が小さい場合に前後の差分値が小さくなる特徴がある．そのため出現位置情報をそのまま記録するのではなく，差分値をVariable-Byteなどの整数圧縮手法を用いて符号化することが一般的である．このように表現された列ベクタに対して，VectorCompareFilterでは列ベクタ内のBit-Vector符号化で表された符号列の先頭の値部分を探索することで条件情報と合致する列データの位置情報を抽出する．

実装に関して

まず下記のBitmapEncodingに列ベクタ内の値の総数（numberOfValues）を記録し，その後0個以上のSymbol列を格納する．Symbolでは値（value）と出現位置情報（positions）の可変長配列で格納される．positionsには実際の位置情報の整数ではなく，前後の差分値を記録してProtocolBufferを用いて可変長符号として格納する．VectorCompareFilterでは，Symbol内のvalueを参照することで条件情報の評価を行う．この条件評価では，値が昇順（または降順）で並び替えられている場合に2分探索などの最適化が考えられるが，列ベクタはCPU内のL1/L2キャッシュに収まる程度を想定して設計するため本実装では線形探索で判定を行う．

```
message Symbol {
    requiredstringvalue = 1;
    repeateduint32positions = 2;
}
message BitmapEncoding {
    uint32 numberOfValues = 1;
    repeated Symbol = 2;
}
```

4.3.2 辞書式符号化

辞書式圧縮手法では列ベクタ内に出現する全ての値を辞書として記録して，その作成された辞書内の項目番号（整数）に値を置き換える．また辞書を作成する際に，辞書内の値と対応する項目番号の順序関係を保持するように辞書を作成する．前節の例と同様に以下の文字列からなる列ベクタを考える．

one, two, three, one, two, three, one, two, three, ...

これを以下のように，先頭に元の値の順序情報を保持した辞書を格納し，後続にその辞書に対応した項目番号である整数列（先頭が0）を格納する．

{one, three, two}, 0, 1, 0, 2, 1, 0, 2, 1, ...

辞書内の項目番号が値の順序を表しているため，VectorCompareFilterの条件情報の評価では可変長の文字列を直接比較するのではなく，処理の効率化の観点で固定長の項目番号を用いて判定を行う．

実装に関して

Bit-Vector符号化と同様に，列ベクタ内の値の総数（numberOfValues）をまず記録する．0個以上の値の順序関係を保持した辞書がdictsに記録され，その後にindexesに項目番号を表す整数が可変長符号としてProtocol Buffersを用いて格納される．VectorCompareFilterでは，dictsを参照することで条件情報を対応する項目番号に変換を行う．そして変換された条件情報を用いてindexesとの比較を実施する．

```
message DictionaryEncoding {
    uint32 numberOfValues = 1;
    repeated string dicts = 2;
    repeated uint32 indexes = 3;
}
```

4.3.3 Run-Length 符号化

RLEは連続値を効率的に符号化する手法で，一般的にはカーディナリティが低

く，ソート済みデータ列に対して適用する．前節の辞書式圧縮手法の例と同様に以下の文字列からなる列ベクタを考える．

one, one, one, one, two, two, three, three, three, ...

RLEでは（値，繰り返し数，出現先頭位置）の3つの組で記録する．そのため先ほどの例は以下のように符号化される．

{one, 4, 0}, {two, 2, 4}, {three, 3, 6}, ...

実装に関して

前述の2つの手法と同様に，列ベクタ内の値の総数（numberOfValues）をまず記録する．値を先行してvaluesに記録した後，（繰り返し数，出現先頭位置）の組を0個以上のsymbolsに書き出す．値を先に書き出す理由はVectorCompareFilterで値のみを先に線形探索を行うことで条件判定処理を効率化するためである．

```
message Symbol {
    required uint32 numberOfRepetitions = 1;
    required uint32 offset = 2;
}
message RunLengthEncoding {
    uint32 numberOfValues = 1;
    repeated string values = 2;
    repeated Symbol symbols = 3;
}
```

4.4 評価実験

前節で説明した3つの圧縮形式で格納されたデータを参照する物理プランの処理時間と実行コストの評価を行い，各圧縮形式が物理プランの実行時間に与える影響を分析する．文字列型で定義されたcol1とcol2の2列から構成される関係R(col1, col2)に対して，以下のSQLで記述されたクエリを考える．

Q1: SELECT * FROMR WHERE col1 = '*param*';

4.2.2節で説明した様に，クエリは論理プランに変換された後に，論理プランに対応した複数の実行計画の中から最適なものが選ばれ実行される．本評価では，Q1の論理プランに対応した2つの実行計画（図4.5）を前提に，条件（圧縮形式と入力データパタン）を変化させた場合の実行時間の評価を行う．

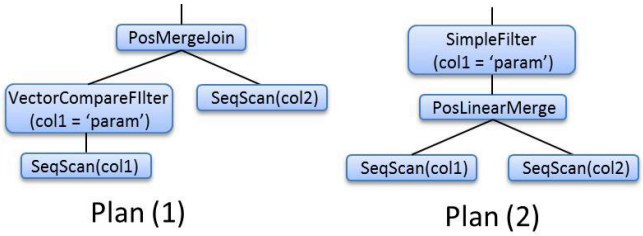

図4.5 Q1に対応した実行計画

各実行計画の実行時間に関する評価実験は，col1の圧縮形式とWHERE句の*param*で選択される行の割合（選択率）を0.001%～5.000%の間で変化させることで実施した．指定した選択率を入力クエリに参照させるために，選択率に対応したカーディナリティを持つ入力データを人工的に作成した．また作成したデータを昇順に並び替えたパタン（sorted）と，一様に分布させたパタン（uniform）の2つを入力データとして用いた．col1のProjectionには4.3節で導入した3つの圧縮形式を適用して，さらに評価の基準に用いるLZ77（lz77）と非圧縮（raw）のパタンも併せて評価を行った．透過的な圧縮形式ではないLZ77と非圧縮には，図4.5のPlan(1)においてVectorCompareFilterではなくSimpleFilterを適用する．WHERE句による述語評価が無いcol2のProjectionに関しては，単純にLZ77で圧縮されている想定で評価を行っている．列指向DBMSアーキテクチャに関する研究では，MonetDB/X100のようにCPU最適化に着眼した研究が広く行われているため，本実験に関しても評価対象のデータを全てメモリ上に置いた状態で実施した．

本評価はXeon X5670と16GiBのメモリを搭載したサーバを用いて実施した．Xeon X5670は32KiBのL1キャッシュ，256KiBのL2キャッシュと，12MiBのLLキャッシュを搭載している．4.3節で説明したように透過的な圧縮手法に関してはProtocol Buffers v2.5.0を，LZ77の実装に関してはSnappy v1.1.1を用いて列指向

DBMSのプロトタイプをC++で構築した．このプロトタイプをgccのv4.7.1（-O2）でコンパイルして評価に用いた．また実行コスト分析ではCPUプロファイラとしてperf v3.6.9を用いた．まず図4.5で示した2つの実行計画の実行時間の評価と，各条件におけるcol1の圧縮率を4.4.1節示す．その後の4.4.2節で選択率が0.001%，0.050%，0.500%の条件でのCPU内の実行命令数とキャッシュミス回数の結果を示すことで実行時間と各圧縮形式におけるコストの関係を分析する．

4.4.1 各実行計画の実行時間評価

表4.1 ストレージ上のProjection（col1）の圧縮率

圧縮手法	ソート有無	0.001%	0.005%	0.010%	0.050%	0.100%	0.500%	1.000%	5.000%
bitmap	×	0.291	0.314	0.340	0.567	0.798	1.296	1.360	1.308
	○	0.290	0.285	0.284	0.284	0.287	0.292	0.283	0.256
dict	×	1.263	1.187	1.103	0.705	0.517	0.281	0.215	0.178
	○	0.196	0.192	0.192	0.192	0.193	0.197	0.191	0.173
rle	×	-	-	-	-	-	-	-	-
	○	0.011	0.009	0.009	0.009	0.009	0.009	0.009	0.008
lz77	×	1.000	0.988	0.960	0.756	0.633	0.472	0.430	0.379
	○	0.234	0.231	0.230	0.230	0.231	0.235	0.229	0.212

sortedでの実行時間の結果を図4.6に，uniformでの実行時間の結果を図4.7にそれぞれ示す．RLE（表中のrle）はソートされていないデータに対しては用いられないため，sortedのみの結果を記載した．横軸はクエリの選択率を，縦軸が列ベクタのProjectionを非圧縮（図中のraw）で格納したデータを基準にした改善率を表し，各パタンの接尾辞の-c1が図4.5のPlan(1)に，-c2がPlan(2)にそれぞれ対応している．表4.1には各条件（圧縮形式と入力データ）におけるProjectionの圧縮率を記載した．図4.6の結果に関しては，選択率が0.001%～0.100%と低い場合に非圧縮データやLZ77（図中のlz77）に対して実行計画に関わらず全ての透過的な圧縮形式が10倍以上高速であることが分かり，特にrle-c1はメモリ上の処理にもかかわらず0.001%～0.01%の範囲で100倍近い性能差が発生している．2つの実行計画の違いによる性能差に関しては，選択率が0.001%～0.050%の低い範囲でPlan(1)がBit-Vector符号化（表中のbitmap）で3.1～4.6倍，辞書式圧縮（表中のdict）で1.8～2.2倍，RLEで1.7～2.8倍程度の差がそれぞれ発生している．しかし選択率が

0.500%前後で優劣が逆転して，最終的に選択率が5.000%まで大きくなった場合にPlan(1)の実行計画はPlan(2)に対して0.40倍程度まで性能劣化している．各Projectionの圧縮率（表4.2）の結果を踏まえた上で入力データがソートされている場合，Projectionの圧縮形式にはRLEを用いて，選択率が低い場合にはPlan(1)を，選択率が0.500%以上の場合にはPlan(2)の実行計画をそれぞれ選択した場合に最も効率的に処理できることが分かる．

データを一様に分布させた図4.7の結果においては，選択率が低い0.001%の時のdict-c1は1.6～2.8倍程度，選択率が高い場合にでもbitmap-c1とdict-c1が1.8～2.3倍程度の高速化を実現している．しかし，それ以外の条件では改善率は低く，Plan(2)の実行計画におけるbitmap-c2，dict-c2，lz77-c2では，非圧縮データと比較して同等か，それ以下の性能まで劣化している．また選択率が低い場合のbitmap-c1とdict-c1を除けば，実行計画の違いによる性能差が約10%以内で非常に小さくなっていることも併せて分かる．

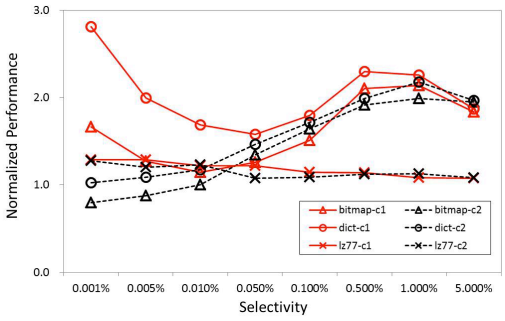

図4.6 実行計画と圧縮形式による実行時間評価（sorted）

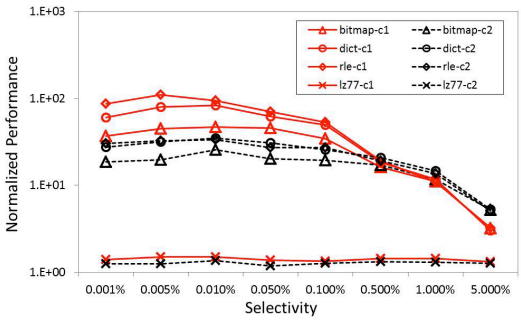

図 4.7 実行計画と圧縮形式による実行時間評価（uniform）

4.4.2 実行計画とコストの考察

図4.6の結果から選択率が低い場合にはPlan(1)の実行計画の性能が高く，0.500%以上で逆転してPlan(2)の実行性能が良くなる結果となった．4.2.2節で説明した様に列指向DBMSを含めた一般的なDBMSでは，実際にクエリを実行する前に効率的に処理可能な実行計画を選択する必要がある．そのため実行前に入力クエリと処理対象のデータから妥当な実行計画を選択するためのコストモデル構築が課題となる．処理対象のデータが全て（もしくは大部分が）メモリ上にある前提で，プラン処理をモデル化する検討は従来研究 [58, 59]で既に取り扱われている．特

にMonetDB/X100に関わるCWI [58]の研究では，物理プランを基本的な6つの参照パタンの組み合わせでモデル化し，実行コスト*T*を処理中に発生するCPUキャッシュミス回数を用いて以下のように見積もる手法が提案されている．

$$T_{Mem} = \Sigma_{i=1}^{N}(M_i^s \times l_{i+1}^s + M_i^r \times l_{i+1}^r) \quad (4.1)$$

表 4.2 キャッシュミス回数による実行コスト T_{Mem} のモデル化

読み込みパタン	概要
$s\_tra(R[,u])$	R に対する 1 回ずつの順読込み，u は行毎の読込むデータ（カラム）サイズ（図 8(a)） ※ u が指定されていない場合は u=$R.w$ とする
$r\_tra(R[,u])$	R に対する 1 回ずつのランダム読込み（図 8(b)）
演算子	概要
$a \odot b$	a と b の並行実行を表す演算子
$a \oplus b$	a と b の直列実行を表す演算子

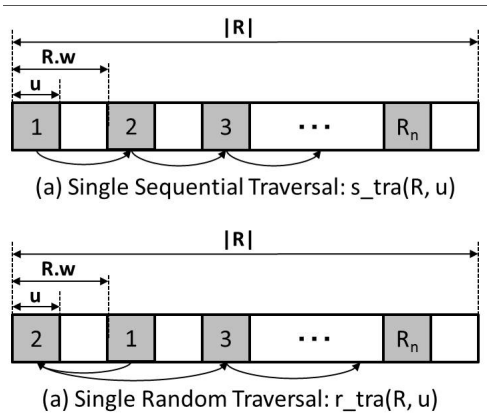

図 4.8 s_tra(R[,u])と r_tra(R[,u])

M（M^s は順読み込み，M^r はランダム参照）がキャッシュミス回数を，l（l^s は順読み込み，l^r はランダム参照）がキャッシュミス時のレイテンシを表している．これらをキャッシュ階層数 N で合計したものを実行コスト T_{Mem} とする提案である．具体的な参照パタンの一部の例と，複数の物理プランが並列に実行されている影響を考慮するための演算子を表 4.2 と図 4.8 に示す．R は処理される関係データを抽象化した構造を表し，$|R|$が全体のサイズを，$R.w$ は 1 行当たりのサイズ

をそれぞれ表す．本章では取り扱わない他の参照パタンに関しては省略するため元論文 [58]を参照していただきたい．

列指向 DBMS の場合は処理データが Projection であるため，*R* を 1 つの列だけ含む構造とみなす．これらのコストモデルを用いた場合の図 4.5 の実行計画のコストを以下のように見積もることができる．

Plan(1)の参照パタン

$$\begin{aligned} &\mathrm{U}_{col1} \leftarrow VectorCompareFilter(P_{col1}) : \\ &\quad s\_tra(P_{col1}) \odot s\_tra(U_{col1}) \end{aligned} \tag{4.2}$$

$$\begin{aligned} &\mathrm{V} \leftarrow PosMergeJoin(U_{col1}, P_{col2}) : \\ &\quad s\_tra(U_{col1}) \odot s\_tra(P^{*}_{col2}) \odot s\_tra(V) \end{aligned} \tag{4.3}$$

Plan(2)の参照パタン

$$\begin{aligned} &\mathrm{V} \leftarrow SimpleFilter(P_{col1}, P_{col2}) : \\ &\quad s\_tra(P_{col1}) \odot s\_tra(P^{*}_{col2}) \odot s\_tra(V) \end{aligned} \tag{4.4}$$

P^{*}_{col2} は P_{col1} に対して条件が合致した行データに対応する P_{col2} の列データを表す．s_tra(*R*)のキャッシュミス回数 M^{s} は以下の式で見積もられる．

$$\mathrm{M}^{s}_{i}(s\_tra(P)) = \lceil \|P\| / B_i \rceil \tag{4.5}$$

$$\mathrm{M}^{r}_{i}(s\_tra(P)) = 0 \tag{4.6}$$

B_iは階層 *i* におけるキャッシュブロックサイズ（例えば Xeon X5670 の L1 キャッシュは 64B）を表す．演算子⊙は s_tra()の場合は，並列実行による影響は無いと仮定するため単純な総和が物理プランの総キャッシュミス回数 *M* となる．このコストモデルを用いて図 4.6 の実験結果を推定したところ，選択率が 0.500%で発生している優劣変化が表現できない結果となった．そこで図 4.6 における CPU 内の統計情報（実行命令数/キャッシュミス回数/分岐ミス回数）を分析することで，選択率が 0.500%で発生する優劣変化を表現するため既存モデルを修正することを以降では検討する．

図 4.6 と図 4.7 の実験における実際の実行コストを分析するために各条件における CPU 内の実行命令数（Instructions），L1D と LL のキャッシュミス回数，分岐ミス回数（Branch misses）をそれぞれ図 4.9 に示す．上段が sorted，下段が uniform の選択率が 0.001%，0.050%，0.500%の条件における CPU 内の実行コストの棒グラフである．各実行コストの中で最もコストが高い条件を 1.0 として結果を 0.0～1.0 で正規化してプロットした．結果から sorted の条件においては Bit-Vector 符号化，辞書式圧縮，RLE 間の実行コストはほぼ同様の傾向で，Plan(1)の実行計画は選択率が高くなった場合に実行命令数とキャッシュミス回数が大幅に増大していた．一方 uniform は選択率が低い場合には Plan(2)の実行計画に対して Plan(1)の実行コストが小さいが，sorted ほどの特徴的な傾向は見られなかった．

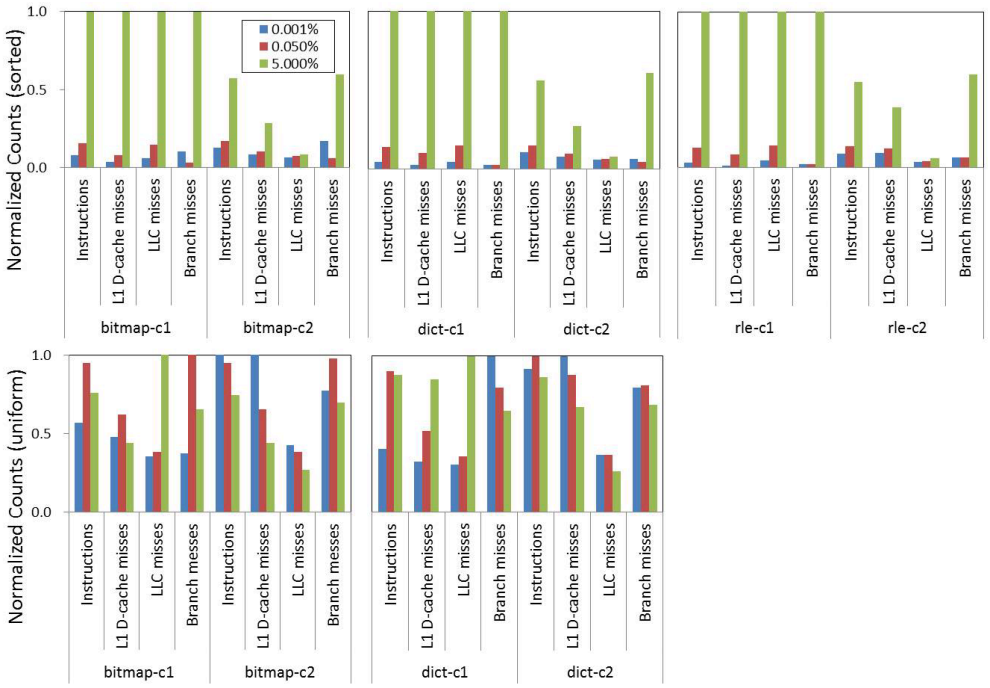

図 4.9 実行命令数/キャッシュミス回数/分岐ミス回数

この結果を踏まえ既存モデルによる実行コスト T_{Mem} に対して，実行命令数によるコストを考慮した T_{Ins} を補正項として用いることを提案する．物理プランの各ノードが処理する出力データを，復元後の列データを処理するためのコスト関数 t_proc()と，圧縮した列データを処理する場合のコスト関数 c_proc()で以下の式を用いて表わす．

$$\mathrm{c\_proc(P)} = \mathrm{P.n} \times l^c \tag{4.7}$$

$$\mathrm{t\_proc(P)} = \mathrm{P.n} \times l^t \tag{4.8}$$

P.n は *P* に含まれるデータの行数で，l^t は復元された行データを，l^c は圧縮された行データを 1 つ処理するために必要な CPU サイクル数をそれぞれ表す．この２つのコスト関数 t_proc()/c_proc()を用いて物理プランの各ノードにおける T_{ins} を以下のように計算する．

物理プランの各ノードの T_{ins} の計算

$$\mathrm{U}_{col1} \leftarrow VectorCompareFilter(P_{col1}) : \\ c\_proc(P_{col1}) + c\_proc(U_{col1}) \tag{4.9}$$

$$\mathrm{V} \leftarrow PosMergeJoin(U_{col1}, P_{col2}) : \\ t\_proc(U_{col1}) + c\_proc(P^*_{col2}) + t\_proc(V) \tag{4.10}$$

$$\mathrm{V} \leftarrow SimpleFilter(P_{col1}, P_{col2}) : \\ c\_proc(P_{col1}) + t\_proc(P_{col1}) + c\_proc(P^*_{col2}) + t\_proc(V) \tag{4.11}$$

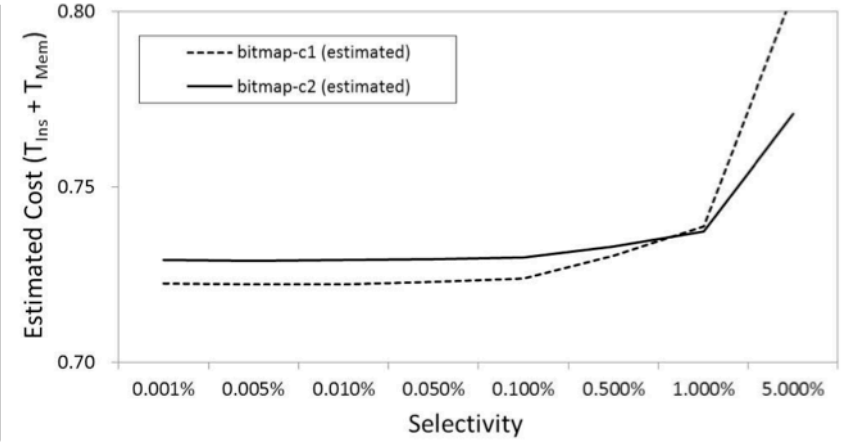

図 4.10 実行計画（Bit-Vector 符号化）のコスト推定

この検討を踏まえて T_{Ins} を T_{Mem} の補正項として構築した実行コストモデル（T_{Mem}+T_{Ins}）を用いて，図 4.6 の実験で用いた物理プランのコストを推定した結果を図 4.10 に示す．図 4.10 では Bit-Vector 符号化のみの結果を表示しているが，辞書式圧縮と RLE に関してもほぼ同等の結果が得られている．l^t を *1* として，l^c

は Bit-Vector 符号化の復元の際に実際に発生した平均的な必要 CPU サイクル数である 103.8 を用いて計算を行った．図から実行コストモデルによる推定コストで選択率 0.500%～1.000%の範囲で優劣の反転が表現できていることが分かる．そのため列指向 DBMS の選択操作の実行コストには T_{Mem} に補正項 T_{Ins} を加算することで精度を改善できる知見が得られたことが分かる．

4.5 関連研究

コストモデルは DBMS 技術において複数の候補から最適な実行計画を選択するための重要な要素技術である．DBMS の研究分野では 90 年代から最適な実行計画を選択するためのコストモデルに関する研究 [60]が行われ，様々な手法が提案されてきた．背景でも述べたとおり，90 年代は HDD のような低速な２次記憶装置に対する I/O 回数を最小化することが主な目的で，DBMS に関する変数（テーブルサイズや値のカーディナリティなど）と入出力装置に関する変数（I/O サイズや読み書きの速度など）を用いたコストモデルが一般的だった [54]．これに対して 2000 年以降はより新しいハードウェアを意識したコストモデル [58, 59, 61, 62]が提案されている．本章で取り扱っているメモリの参照パタンを用いたコストモデルに関する先行研究 [58]は，メモリ参照が DBMS のボトルネックになっていることを背景 [14]に，メモリ参照時のキャッシュミス回数からコストを推定する手法である．別の先行研究 [61, 62]では SSD，クラウドストレージ（Amazon S3 や Azure Storage など），メインメモリなどを含めた汎用的なデータの読み書きを抽象化したコストモデルを提案している．またキャッシュミス回数を用いたコストモデル [58]を用いて，列指向 DBMS の集約操作におけるコストを詳細に分析し，データ依存の変数（データサイズやカーディナリティなど）とデータ非依存の変数（集約関数や集約数など）に分割して議論することで，キャッシュミス回数を用いた集約操作のコスト推定における問題点を先行研究 [59]では指摘している．これらの先行研究はデータの読み書きに着眼したコストモデルであるのに対して，本研究では圧縮データの復元処理のような CPU 計算の多い実行計画に着眼してモデルを改善している点が先行研究とは異なる．

4.6 本章のまとめ

本章では，内部の表データを列方向に分割・圧縮を行う列指向 DBMS アーキテクチャを前提として，圧縮データの復元時に発生する CPU 計算量を考慮することでより精度の高い実行計画のコスト推定手法を提案した．先行研究では CPU のキャッシュミス回数からコスト T_{Mem} を見積もる手法が提案されているが，本章で構築した列指向 DBMS のプロトタイプを用いて分析することで，このモデルだけでは候補が複数ある実行計画の優劣の変化を正確に把握できないことが判明した．そこで各実行計画を処理させた場合に発生する CPU 負荷を実際に分析し，コスト T_{Mem} に対して各圧縮形式に対応した CPU 計算量を表現する補正項 T_{Ins} を加算することでこの優劣変化を捉えることができるモデルを考案した．ハードウェアの進化を背景に今後も DBMS の内部処理は CPU 上での処理がボトルネックになると考えられるため，CPU 内部の計算処理やメモリ参照を考慮したコスト推定技術は，現実で発生するより複雑な処理を高速に実行するための最適化技術の要として非常に重要な要素になると考えられる．

第 5 章 本論文のまとめ

本論文では CPU の高度化とメモリの大容量化における DBMS に関する問題を背景に，（1）DBMS 内部で用いられるアルゴリズムとデータ構造の CPU 上での非効率性分析と解決，（2）データ表現（圧縮形式）を考慮した実行計画の列挙とコストモデルの確立について論じた．そしてそれぞれの課題に対して，データ圧縮を活用することで大規模データをメモリ上でコンパクトに表現しながら効率的に処理を行うための CPU 効率の高い DBMS の要素技術を提案した．提案した技術についてその概要を説明する．

VAST 木: 索引の圧縮を用いたベクトル命令による大規模データの探索技術

課題（1）に対して 2 章では，大規模データに対して空間コストが低く CPU の実行効率が高い木構造索引である VAST 木を提案した．VAST 木では SIMD 命令を用いた比較処理のデータ並列度を高めるため，索引の分岐ノードに非可逆圧縮手法を適用する．また索引全体のサイズを削減することを目的に，葉ノードには CPU に最適化された可逆圧縮手法を適用する．結果的にデータ数が 2^{30} の分岐ノードのみのサイズは既存手法に対して 95%以上削減，索引全体のサイズとしては 47〜84%削減できることをそれぞれ確認した．またスループット性能に関しては 2 分木に対して 6.0 倍，既存手法で最も高速な FAST に対して 1.24 倍の性能向上を確認した．探索中の CPU とメモリ間のバスの帯域消費量に関しても，2 分木に対して 94.7%，FAST に対して 40.5%削減できることを併せて確認した．

LZE++: 圧縮されたデータの高速な参照技術

課題（1）に対して 3 章では，圧縮データ全体を復元せずに任意の位置にある部分文字列データを高速に参照可能な圧縮表現である LZE++を提案した．LZE++では共有辞書を用いた再帰処理の枝刈りと，平均（mK/N+1）回の軽量な繰り返し処理により LZEnd と同等の計算量で高速な復元処理を可能にする．メモリ上の処理にもかかわらず 64KiB 以下の圧縮データの復元処理で 3.3〜58.2 倍，64KiB より大きなデータの復元処理では最大で 1439.0 倍の大幅な改善が可能であることを確認した．

圧縮を考慮したコストモデル

課題（2）に対して 4 章では，内部の表データを列方向に分割・圧縮を行う列指向 DBMS アーキテクチャを前提として，圧縮データの復元時に発生する CPU 計算量を考慮したコストモデルを提案した．本コストモデルの導出において，先行研究から妥当な列指向 DBMS アーキテクチャに基づいたプロトタイプを構築し，様々な実行計画に対するコストの分析を行うために 6 種の物理プランと DBMS の内部データの圧縮表現 4 種の実装を行った．これらの物理プランと圧縮表現の組み合わせで実際に発生する CPU 上のコストを分析し，結果として先行研究で提案されている CPU のキャッシュミス回数から見積もられたコスト T_{Mem} だけでは実際の実行計画のコストを推定できないことが判明した．そこで本技術では各圧縮データ形式に対応した CPU 計算量を表現するための補正項 T_{Ins} を加算することを提案し，この提案したコストモデルを用いることでより精度の高い実行計画のコスト推定が実現できることを確認した．

5.1 今後の展望

CPU の高度化とメモリの大容量化に代表されるハードウェアの進化は今現在も続いており，このような高性能・大容量ハードウェア環境は広く一般的に利用されるようになってきている．例えば PasS を提供している企業の中で AWS は，2017 年 12 月現在で最も CPU コア数の多いインスタンス（CPU ソケットが 4 でコア数が 128，メモリサイズが約 2TB）を提供する企業で，アカウントさえあれば誰でも気軽に高性能な環境が利用可能である．また市販のものでも CPU コアの総数が 1000 に近い物理サーバが販売されることが徐々に多くなってきている．このようなハードウェア環境が誰でも容易に利用できる一方で，これらの環境を最大限に活用したシステムを構築するためには序章で述べたように様々なレベルの並列化や，OS・ミドルウェアの（一貫性を保持したキャッシュやロックによる）同期などに関する高度で複雑な知識を要する．そのためハードウェアの進化に対して DBMS 技術の追従が十分ではなく，今後もハードウェアを考慮したデータ構造やアルゴリズムの改善に関する研究は重要視されると考えられる．

コア数の増大と併せてメモリの大容量化や不揮発性メモリの台頭により，従来多数の物理サーバに分散する必要があった高負荷な処理が，今後は少数の高性能環境に集約されていくことが予想される．しかし現状では処理の特性（求められる I/O 性能・CPU 性能の割合や，並列化可能部分の割合など）を利用者自身が考慮して最適なハードウェアとソフトウェアの設計を行う必要がある．そのため処理の特性に対してハードウェアのリソース配置やアルゴリズム・データ構造の選択を行うなどの最適化に関する研究分野が再度注目されることが予想される．また世界的な深層学習の流行により GPU や FPGA などのコプロセッサが広く活用されるようになってきている．DBMS 分野においては以前からこれらのコプロセッサの活用に関する研究が行われてきたが，ハードウェアの導入コストに対して性能向上が高くなく，産業の分野で定着していない現状がある．そのためこれらの GPU や FPGA の流行により，深層学習以外の分野におけるコプロセッサの活用に関する研究が再び注目を浴びることが予想される．

上記は各ハードウェアに特化した最適化を行うアプローチに対して言及したものであったが，DBMS の全ての要素技術に対してこれらの最適化を行い続けることは非常に難しい．そのため相補的なアプローチとして，コンパイラ技術の活用が注目されている．これは入力された問い合わせから最適な実行計画を決定した後，その実行計画を等価な逐次プログラムに変換・コンパイルして実行することで各ハードウェアに適した処理を実現するものである．研究分野で開発された DBMS である HyPer が先駆け[29]で，Spark を代表として様々なミドルウェアで活用されている近代的なアプローチである．現在は物理プランのノード毎に逐次プログラムの雛形を事前に用意しておき，入力された問い合わせの情報を用いて逐次プログラムを生成する単純な方法を採用しているミドルウェアが多い．しかしノード毎にプログラムの雛形を用意しているため，ノードを跨いだ最適化が容易に実現できない問題がある．例えば同じキーに対して Hash Join の後に Hash Aggregation を行う場合，Hash Join を処理するために構築した Hash 表は本質的には Hash Aggregation を処理するために再利用が可能である．しかし雛形が静的に用

[29] System R がコンパイラ技術を活用した初めての DBMS であるが，当時のハードウェア環境ではコンパイル時間や動的リンクなどのオーバヘッドが高く性能向上が得られず採用されなかった．

意されている場合には Hash 表に関する逐次プログラムが Join と Aggregation の処理に対してそれぞれ生成されてしまい，コンパイラがこの再利用性を検出することは難しい．そのため，DBMS 技術とコンパイラ技術をより密に連携させるための研究が今後より注目されると考えられる．

近年スキーマを持ち構造化されたデータに対して，SQL を用いた関係代数的な処理だけではなくグラフ処理，機械学習，ストリーミング処理など様々なタスクを行う事例が増えてきている．これらの構造化データは列方向の分割・圧縮や関係代数に基づく最適化を活用できるため，DBMS 分野以外で DBMS の要素技術が活かされる良い例である．このように今後はタスクや言語などに囚われずユーザが記述した問い合わせから，ハードウェアに最適化された手法を暗黙的に活用するための DBMS の要素技術が重要視されると考えられる．このような技術進展に対して本研究の成果が礎になることを期待したい．

謝辞

本研究を進めるにあたり格別なるご指導とご高配を賜りました大阪大学大学院情報科学研究科マルチメディア工学専攻 鬼塚真教授に謹んで御礼申し上げます．研究内容に関するご指導とご鞭撻，また博士論文をご精読いただき有益なご助言をいただきました副査の原隆浩教授，下條真司教授に深く感謝いたします．また同様に副査として研究内容に有益なご助言をいただきました藤原融教授，松下康之教授に感謝いたします．本研究において研究内容に関する助言，論文の執筆に関するご指導をいただいたNTTソフトウェアイノベーションセンタ特別研究員兼大阪大学大学院情報科学研究科マルチメディア工学専攻 藤原靖宏招へい准教授に深く感謝いたします．またNTT研究所での本研究に関わる活動におきまして，学術的観点だけではなく産業的な側面に関するご指導をいただきましたNTTソフトウェアイノベーションセンタ 小西史和主席研究員，日高東潮主幹研究員，寺本純司主幹研究員，岩村相哲主幹研究員，須賀啓敏主任研究員，内山寛之主任研究員の皆様に深く感謝いたします．最後に私のこれまでの人生，そして研究生活を送る上で暖かい支援と理解をいただきました友人や家族，特に妻の典子に心から感謝いたします．

参考文献

[1] E. F. Codd, A Relational Model of Data for Large Shared Data Banks, Communications of the ACM, Volume 13 Issue 6, Pages 377-387, 1970.

[2] M. M. Astrahan et al., System R: Relational Approach to Database Management, ACM Transactions on Database Systems, Volume 1, Issue 2, Pages 97-137, 1976.

[3] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Communications of the ACM, Volume 51, Issue 1, Pages 107-113, 2008.

[4] Ashish Thusoo et al., Hive: A Warehousing Solution over a Map-Reduce Framework, Proceedings of the VLDB Endowment, Volume 2, Issue 2, Pages 1626-1629, 2009.

[5] Goetz Graefe and William J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, Proceedings of the 9th International Conference on Data Engineering, Pages 209-218, 1993.

[6] George P. Copeland and Setrag N. Khoshafian, A Decomposition Storage Model, Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Pages 268-279, 1985.

[7] Matei Zaharia et al., Spark: Cluster Computing with Working Sets, Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Pages 10-10, 2010.

[8] G. Graefe, Volcano: An Extensible and Parallel Query Evaluation System, IEEE Transactions on Knowledge and Data Engineering, Volume 6, Issue 1, Page 120-135, 1994.

[9] Thomas Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware, Proceedings of the VLDB Endowment, Volume 4, Issue 9, Pages 539-550, 2011.

[10] Michael J. Flynn, Some Computer Organizations and their Effectiveness, IEEE Transactions on Computers, Volume 21, Issue 9, Pages 948-960, 1972.

[11] Nadathur Satish, Changkyu Kim, Jatin Chhugani et al., Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?, Communications of the ACM, Volume 58, Issue 5, Pages 77-86, 2015.

[12] Peter Boncz, et al., MonetDB/X100: Hyper-Pipelining Query Execution, Conference on Innovative Data Systems Research, Pages 225-237, 2005.

[13] Marcin Zukowski, Peter Boncz, Niels Nes, and Sándor Héman, MonetDB/X100: A DBMS in the CPU Cache, IEEE Data Eng. Bull. Volume 28, Issue 2, Pages 17-22, 2005.

[14] Stefan Manegold, Peter Boncz, and Martin L. Kersten, Optimizing Database Architecture for the New Bottleneck: Memory Access, Proceedings of the 26th International Journal on Very Large Data Bases, Volume 9, Issue 3, Pages 231-246, 2000.

[15] Hankins, R.A. and Patel, J.M., Effect of Node Size on the Performance of Cache-Conscious B+-trees, Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Pages 283-294, 2003.

[16] Changkyu Kim et al., FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs, Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Pages 339-350, 2010.

[17] Rudolf Bayer and Karl Unterauer, Prefix B-trees, ACM Transactions on Database Systems, Volume 2, Issue 1, Pages 11-26, 1977.

[18] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz, Super-Scalar RAM-CPU Cache Compression, Proceedings of the 22nd International Conference on Data Engineering, Pages 59-70, 2006.

[19] Matthew Reilly and When Multicore, Isn't Enough: Trends and the Future for Multi-Multicore Systems, Proceedings of the Annual High-Performance Embedded Computing Workshop, 2008.

[20] Nadathur Satish, Changkyu Kim, Jatin Chhugani et al., Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Pages 351-362, 2010.

[21] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, et al., Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture, Proceedings of the VLDB Endowment, Volume 1, Issue 2, Pages 1313-1324, 2008.

[22] Naga K. Govindaraju, Jim Gray, et al., GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management, Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Pages 325-336, 2006.

[23] Naga K. Govindaraju et al., Fast Computation of Database Operations using Graphics Processors, Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Pages 215-226, 2004.

[24] Bingsheng He and Jeffrey Xu Yu, High-Throughput Transaction Executions on Graphics Processors, Proceedings of the VLDB Endowment, Volume 4, Issue 5, Pages 314-325, 2011.

[25] Nadathur Satish et al., Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Pages 351-362, 2010.

[26] Benjamin Schlegel et al., k-ary Search on Modern Processors, Proceedings of the 5th International Workshop on Data Management on New Hardware, Pages 52-60, 2009.

[27] Stefan Manegold, Martin L. Kersten, and Peter Boncz, Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct, Proceedings of the VLDB Endowment, Volume 2, Issue 2, Pages 1648-1653, 2009.

[28] Tobin J. Lehman and Michael J. Carey, A Study of Index Structures for Main Memory Database Management Systems, Proceedings of the 12th International Conference on Very Large Data Bases, Pages 294-303, 1986.

[29] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry, Improving Index Performance Through Prefetching, Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Pages 235-246, 2001.

[30] Jun Rao and Kenneth A. Ross, Cache Conscious Indexing for Decision-Support in Main Memory, Proceedings of the 25th International Conference on Very Large Data Bases, Pages 78-89, 1999.

[31] Jun Rao and Kenneth A. Ross, Making B+-Trees Cache Conscious in Main Memory, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Pages 475-486, 2000.

[32] Jingren Zhou and Kenneth A. Ross, Buffering Accesses to Memory-Resident Index Structures, Proceedings of the 29th International Conference on Very Large Data Bases, Volume 29, Pages 405-416, 2003.

[33] Changkyu Kim, Jatin Chhugani, Nadathur Satish et al., Designing Fast Architecture Sensitive Tree Search on Modern Multi-Core/Many-Core Processors, ACM Transactions on Database Systems, Volume 36, Issue 4, 2011.

[34] Tim Kaldewey et al., Parallel Search on Video Cards, Proceedings of the First USENIX Conference on Hot Topics in Parallelism, 2009.

[35] Jason Sewall et al., PALM: Parallel Architecture-Friendly Latch-Free Modification to B+Trees on Many-Core Processors, Proceedings of the VLDB Endowment, Volume 4, Issue 11, Pages 795-806, 2003.

[36] Wenbin Fang, et al., Database Compression on Graphics Processors, Proceedings of the VLDB Endowment, Volume 3 Issue 1-2, Pages 670-680, 2010.

[37] Fabrizio Silvestri and Rossano Venturini, VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming, Proceedings of the 19th ACM International Conference on Information and Knowledge Management, Pages 1219-1228, 2010.

[38] Hao Yan, Shuai Ding, and Torsten Suel, Compressing Term Positions in Web Indexes, Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, Pages 147-154, 2009.

[39] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, et al., The Design and Implementation of Modern Column-Oriented Database Systems, Foundations and Treands in Databaes, Volume 5, No. 3, Pages 197-280, 2013.

[40] Paolo Ferragina and Giovanni Manzini, On Compressing the Textual Web, Proceedings of the ACM International Conference on Web Search and Data Mining, Pages 391-400, 2010.

[41] Christopher Hoobin, Simon J. Puglisi, and Justin Zobel, Relative Lempel-Ziv Factorization for Efficient Storage and Retrieval of Web Collections, Proceedings of the VLDB Endowment Volume 5, Issue 3, Pages 265-273, 2011.

[42] Sebastian Kreft and Gonzalo Navarro, LZ77-like Compression with Fast Random Access, Proceedings of the 2010 Data Compression Conference, Pages 239-248, 2010.

[43] Kunihiko Sadakane, New Text Indexing Functionalities of the Compressed Suffix Arrays, Journal of Algorithms, Volume 48, Issue 2, Pages 294-313, 2003.

[44] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter, High-Order Entropy-Compressed Text Indexes, Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, Pages 841-850, 2003.

[45] Roberto Grossi and Jeffrey Scott Vitter, Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching, SIAM Journal on Computing, Volume 35, Issue 2, Pages 378-407, 2005.

[46] Paolo Ferragina and Giovanni Manzini, Opportunistic Data Structures with Applications, Proceedings of the Annual Symposium on Foundations of Computer Science, 2000.

[47] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl et al., PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth, Proceedings of the 17th International Conference on Data Engineering, Page 215-226, 2001.

[48] S. Rao Kosaraju et al., Compression of Low Entropy Strings with Lempel-Ziv Algorithms, SIAM Journal on Computing, Volume 29, Issue 3, Pages 893-911, 2000.

[49] Giovanni Manzini, An Analysis of the Burrows-Wheeler Transform, Journal of the ACM JACM Homepage archive, Volume 48, Issue 3, Pages 407-430, 2001.

[50] Daisuke Okanohara and Kunihiko Sadakane, Practical Entropy-Compressed Rank/Select Dictionary, Proceedings of the Meeting on Algorithm Engineering & Expermiments, Pages 60-70, 2006.

[51] N. Jesper Larsson and Alistair Moffat, Offline Dictionary-based Compression, Proceedings of the Conference on Data Compression, 1999.

[52] Sebastian Kreft and Gonzalo Navarro, Self-Indexing based on LZ77, Proceedings of the 22nd Annual Conference on Combinatorial Pattern Matching, Pages 41-54, 2011.

[53] Philip Bille et al., Random Access to Grammar-Compressed Strings, Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms, Pages 373-389, 2011.

[54] Raghu Ramakrishnan and Johannes Gehrke, Database Management Systems, McGraw-Hill, 2nd Edition, 2000.

[55] Mike Stonebraker et al., C-Store: A Column-Oriented DBMS, Proceedings of the 31st International Conference on Very Large Data Bases, Pages 553-564, 2005.

[56] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem, Column-Stores vs. Row-Stores: How Different Are They Really?, Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Pages 967-980, 2008.

[57] Goetz Graefe and Harumi Kuno, Fast Loads and Queries, Proceedings of International Conference on Database Systems for Advanced Applications, 2010.

[58] Stefan Manegold, Peter Boncz, and Martin L. Kersten, Generic Database Cost Models for Hierarchical Memory Systems, Proceedings of the 28th International Conference on Very Large Data Bases, Pages 191-202, 2002.

[59] Stephan Müller and Hasso Plattner, An In-Depth Analysis of Data Aggregation Cost Factors in a Columnar In-Memory Database, Proceedings of the 15th International Workshop on Data Warehousing and OLAP, Pages 65-72, 2012.

[60] S. Chaudhuri and V. R. Narasayy, Self-Tuning Database Systems: A Decade of Progress, Proceedings of the 34th International Conference on Very Large Data Bases, 2007.

[61] Ladjel Bellatreche et al., The Generalized Physical Design Problem in Data Warehousing Environment: Towards a Generic Cost Model, Proceedings of Information & Communication Technology Electronics & Microelectronics, 2013.

[62] Ladjel Bellatreche et al., How to Exploit the Device Diversity and Database Interaction to Propose a Generic Cost Model?, Proceedings of the 17th International Database Engineering & Applications Symposium, Pages 142-147, 2013.