DBSJ Japanese Journal, Vol.13-J, No.1
October 2014

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

Column-oriented Databases Directly Operating on Compressed Data: Evaluation and Discussion

山室 健♡ 若森 拓馬♢ 寺本 純司♠ 西村 剛♣

Takeshi YAMAMURO Takuma WAKAMORI
Junji TERAMOTO Go NISHIMURA

蓄積されたログデータに対する分析処理は，数多い属性（列）から限られた部分だけを参照する傾向があることから，**I/O** 量削減を目的としたデータの列配置や，列構造の利点を活かした圧縮を特徴とする列指向 **DB** が広く採用されている．列指向 **DB** では圧縮された内部データをクエリ実行中に復元する必要があるため，**I/O** だけではなく **CPU** やメモリ参照のコストを考慮したクエリの実行計画が重要となる．本研究では列指向 **DB** の特徴の **1** つである透過的な圧縮手法を用いたクエリの述語評価に着眼する．列指向 **DB** のプロトタイプを用いて圧縮された列データに対するクエリの実行時間の評価を行い，実行コストを分析することで実行計画に必要なコストモデルの考察を行う．

Column-oriented databases, which exploit columnar storage models and compression so as to mitigate I/O overheads, are widely-used to process an amount of software logs because they typically access the limited portion of attributes in practical use-cases. These databases need to unpack compressed data when running queries, and it is significant to take into accounts not only I/O costs, but also CPU and memory access ones for plan optimizations. The objective of this paper is to evaluate running time to process compressed data with columnar formats on our prototype, and discuss what is a reasonable cost model for column-oriented databases.

1. 研究背景

蓄積されたサービスやサーバのログデータ（CSV/TSV や JSON）に対して，選択・集約などの関係代数的に表現される分析処理を SQL 形式で記述し実行することの需要は高い．SQL を用いる RDBMS で採用されるストレージモデルの代表として，データを行方向に配置する NSM (N-ary Storage Model) [5] と，列方向に配置する DSM (Decomposition Storage Model) [3] の 2 つがある．RDBMS 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．DSM を前提に，圧縮技術，非同期 I/O，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 DB の C-Store が 2005 年に提案された [4]．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率が期待できるが，圧縮された内部データをクエリ実行中に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実行計画が課題の 1 つとなる．

本研究では列指向 DB 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 [7] に着眼して，SQL で記述されたクエリの述語評価を行う際に発生する実行コストの評価を行う．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．取り扱う圧縮形式は列指向 DB でよく用いられる以下の 3 つの形式である．

- 値毎の出現位置を記録した 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

次節 2. では前提とする列指向 DB に関する概説を行う．節 3. では，評価に用いる上記 3 つの圧縮形式に関して詳細化を行い，節 4. では列指向 DB 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．

2. 想定する列指向 DB アーキテクチャ

2.1 列指向 DB の概略

本論文で想定する列指向 DB のアーキテクチャを，関連文献を参照しながら概説する．列指向 DB 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 1）．列に対応したストレージ上の表現を C-Store [4] に倣い’Projection’と呼称する．図 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 [1]．図 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 DB では 1 行単位でクエリの処理を行う Volcano-Style [5] に従った実装が一般的であるが，参照の多い分析処理（OLAP）では複数行でまとめて処理（Vector-at-a-time）するほうが 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 [6][2] ことが知られている．

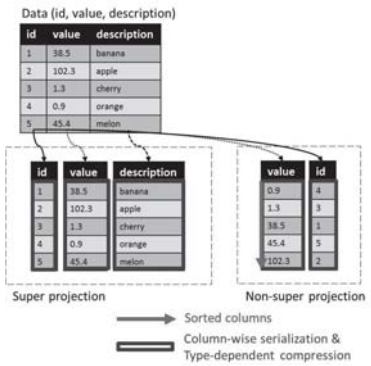

図 1: 表と Projection の関係

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，

♡ 正会員 日本電信電話株式会社 yamamuro.takeshi@lab.ntt.co.jp
♢ 正会員 日本電信電話株式会社 wakamori.takuma@lab.ntt.co.jp
♠ 正会員 日本電信電話株式会社 teramoto.junji@lab.ntt.co.jp
♣ 正会員 日本電信電話株式会社 nishimura.go@lab.ntt.co.jp

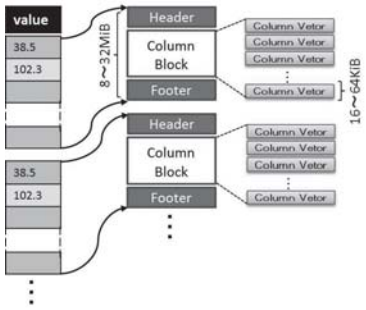

図 2: Projection のデータ構造

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報と offset を用いて計算される仮想 ID（*vid*）を用いて行データの復元が行われる．

列指向 DB では上記の列ごとに格納された Projection を前提に，行指向 DB と同様 [10] に入力されたクエリから論理プランを作成する（図 3）．上記のような形式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 InnerJoin が論理プラン内で多く発生することが列指向 DB の特徴である．図 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（*vid*）で InnerJoin を行い，その結果と *id* の位置情報でさらに InnerJoin 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 InnerJoin 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 [7]．次節では ColumnScan 処理を行う物理プランに着眼する．

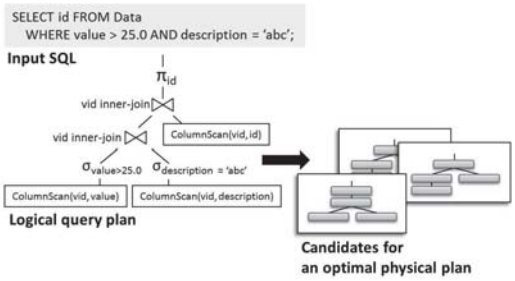

図 3: SQL 形式クエリとプラン

2.2 列指向 DB における実行計画

本論文で想定する各物理プランの役割を以下で概説する．

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物理プランに返却する．

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ものを抽出する．

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評価を行う．復元せずに条件情報を評価する手法に関しては次節 3. 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，下位の物理プランから取得した結果に対応した列データの抽出と結合を行う．

PosLinearMerge/PosMergeJoin
: 下位の物理プラン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．一方選択処理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．先頭から順に処理を行う PosLinearMerge に対して，MergeJoin を行う PosMergeJoin は処理コストが高くなる傾向がある．

論理プランにおける列データの読み取り処理は I/O 処理を伴いながら対象となる列データをストレージから取得する SeqScan に，WHERE 句の条件情報を含んだ各列データの選択処理は VectorCompareFilter/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用いた InnerJoin は PosFilterScan/PosLinearMerge/PosMergeJoin を用いて実現される．

図 3 で用いた以下のクエリを考える．

SELECT *id* FROM *Data*
WHERE *value* > 25.0 AND *description* LIKE 'abc';

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 の様なプランが考えられる．

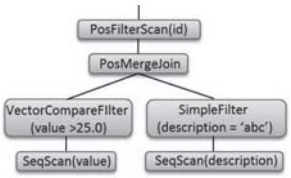

図 4: 実行計画の例

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 を用いて取得する．

DBMS は CPU と I/O 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．

次節では本論文で評価を行う VectorCompareFilter を実現する圧縮形式に関して詳述する．一般的な DBMS に実装されているデータは様々だが，本論文では特に文字列型（SQL 標準における CHARACTER 型）を扱う．

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である 3 つの手法を説明する [4][7]．1 つ目の BitVector 符号化（*bitmap*）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（*dict*）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 [7]．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ある．

本論文の評価では Google の Protocol Buffer [1] を用いて，圧縮されたデータの入出力を行う．以下では，各手法の具体的な説明と，Protocol Buffer を用いた実装に関して詳述する．

3.1 Bit-Vector 符号化

bitmap 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, . . .

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, . . . } . . .

各値の位置情報である整数列は昇順であるため，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 *bitmap* 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．

Protocol Buffer を用いた実装に関して

まず 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 は，値（value）と出現位置情報（positions）が可変長配列として格納される．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 Protocol Buffer 経由で可変長符号として書き出す．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．この処理では値が昇順（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 [4] 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されるため，本実装では線形探索で判定を行う．

```
message Symbol {
  required string value = 1;
  repeated uint32 positions = 2;
}
message BitmapEncoding {
  uint32 numberOfValues = 1;
  repeated Symbol = 2;
}
```

3.2 辞書式符号化

dict 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．3.1 節の *bitmap* の例と同様の以下の文字列からなる列ベクタを考える．

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

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, 2, 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，固定長の項目番号を用いて判定を行う．

Protocol Buffer を用いた実装に関して

bitmap 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維持した辞書が dicts に記録され，その後に indexes に項目番号の整数が可変長符号として書き出される．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する．その後，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;
}
```

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，本研究ではカーディナリティが低く，ソート済みデータに対して用いる．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}, . . .

Protocol Buffer を用いた実装に関して

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の際の読み込みパタン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. 評価実験

前節で導入した 3 つの圧縮形式を用いた場合の実行時間と，各圧縮手法の実行コストの評価を行い，各圧縮手法のコストが実行時間に与える影響を考察する．文字列型で定義された *col*1 と *col*2 の 2 列から構成される関係 *R*(*col*1, *col*2) に対して，以下の SQL で記述されたクエリを考える．

Q1: SELECT ∗ FROM *R* WHERE *col*1 = '*param*';

表 1: ストレージ上の Projection（*col*1）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

節 2.2 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 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．

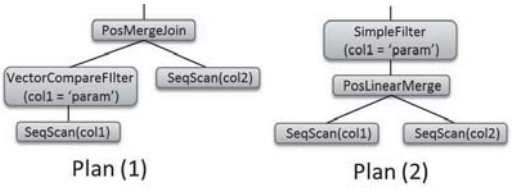

図 5: Q1 に対応した実行計画

各実行計画の実行時間に関する評価実験は，*col*1 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行った．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ソートしたパタン（*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．*col*1 の Projection には節 3. 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 *lz*77 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 *lz*77 と *raw* に関しては，図 5 の Plan(1) において VectorCompareFilter ではなく SimpleFilter を適用した．WHERE 句による述語評価が無い *col*2 の Projection に関しては，単純に *lz*77 で圧縮されている想定で評価を行っている．列指向 DB の研究では，MonetDB/X100 [6] のように I/O の最適化を前提として CPU 最適化の研究が広く行われているため，本実験に関しても I/O を含んだ評価は行わず，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 Xeon5670 と 16GiB のメモリのサーバを用いて実施した．Xeon5670 は 32 KiB の L1cache，256 KiB の L2cache と，12 MiB の LLC (Last Level Cache) を搭載している．列指向 DB のプロトタイプは，Snappy v1.1.1 [2] と Protocol Buffer v2.5.0 を用いて C++で記述した．作成したプログラムは GCC の v4.7.1 で-O2 でコンパイルを行い評価に用いた．また実行コスト分析で CPU におけるパフォーマンスカウンタ値を取得するために，perf の v3.6.9 を用いた．

まず節 4.1 で図 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 *col*1 の圧縮率を示す．その後の節 4.2 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考察する．

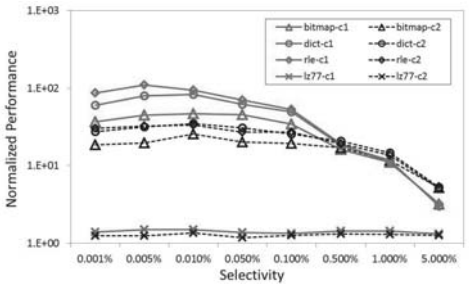

図 6: 実行計画と圧縮形式による実行時間評価（*sorted*）

4.1 各実行計画の実行時間評価

sorted での実行時間の結果を図 6 に，*uniform* での実行時間の結果を図 7 に示す．*rle* はソートされていないデータに対する効果が小さいため，*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 を非圧縮で格納した *raw* を基準にした改善比率を表し，各パタンの接尾辞の'-c1' が図 5 の Plan(1) に，'-c2' が Plan(2) にそれぞれ対応している．表 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 6 の結果に関しては，選択率が 0.001%～0.100%と低い場合に *raw* や *lz*77 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 x10 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関わらず 0.001%～0.01%の範囲で x100 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 *bitmap* で x3.1～x4.6，*dict* で x1.8～x2.2，*rle* で x1.7～x2.8 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 x0.40 程度まで性能劣化している．各 Projection の圧縮率（表 2）を考慮した場合，入力データがソートされている Projection を *rle* で圧縮して，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) をそれぞれ選択した場合に効率が良くなることが分かる．

データを一様に分布させた図 7 の結果においては，選択率が低い 0.001%の時の *dict*-c1 は x1.6～x2.8 程度，選択率が高い場合にでも *bitmap*-c1 と *dict*-c1 が x1.8～x2.3 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/*lz*77-c2 では，*raw* 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．

4.2 実行計画とコストの考察

図 6 の結果では，選択率が低い場合は Plan(1) の実行計画のほうが性能改善が高く，0.500%以上で逆転して Plan(2) のほうが性能改善が高くなる結果となった．節 2.2 で説明した様に列指向

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

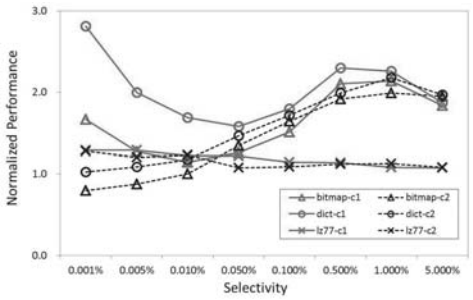

図 7: 実行計画と圧縮形式による実行時間評価（*uniform*）

DB を含めた一般的な DB 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 [8][9] で既に取り扱われている．特に MonetDB に関わる CWI [8] の研究では，物理プランを基本的な 6 つの読み込み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．

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

M が（M^s は順読込み，M^r はランダム読み込みの）キャッシュミス回数を，l 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} とする考えである．具体的な読み込み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 2 と図 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．これらの詳細に関しては本論文では省略するため，元論文 [8] を参照していただきたい．

表 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 [8]

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

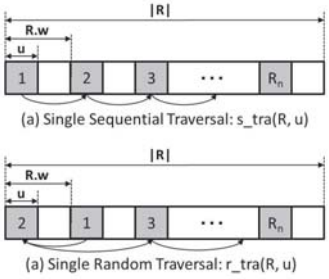

図 8: $s\_tra(R[,u])$ と $r\_tra(R[,u])$ の概要（[8] より引用）

列指向 DB の場合は処理データが Projection であるため，R に列を 1 つ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 5 の実行計画のコストは以下のようになる．

Plan(1)′s Access Pattern

$U_{col1} \leftarrow VectorCompareFilter(P_{col1})$:
$\quad s\_tra(P_{col1}) \odot s\_tra(U_{col1})$
$V \leftarrow PosMergeJoin(U_{col1}, P_{col2})$:
$\quad s\_tra(U_{col1}) \odot s\_tra(P^*_{col2}) \odot s\_tra(V)$

Plan(2)′s Access Pattern

$V \leftarrow SimpleFilter(P_{col1}, P_{col2})$:
$\quad s\_tra(P_{col1}) \odot s\_tra(P^*_{col2}) \odot s\_tra(V)$

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．

$$M_i^s(s\_tra(P)) = \lceil \|P\| / B_i \rceil$$
$$M_i^r(s\_tra(P)) = 0$$

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 Xeon5670 の L1cache は 64B）を表す．演算子 $\odot$ 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 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 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 6 と図 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），キャッシュミス回数（L1 D-cache misses と LLC misses）と，分岐ミス回数（Branch misses）を図 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 *bitmap/dict/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) のコストが小さいが *sorted* ほどの傾向は見られなかった．

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検討した．各物理プラン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 $c\_proc()$ で以下のように表わすことで補正項 T_{Ins} とすることを考える．

Physical Plan′s T_{Ins}

$U_{col1} \leftarrow VectorCompareFilter(P_{col1})$:
$\quad c\_proc(P_{col1}) \oplus c\_proc(U_{col1})$
$V \leftarrow PosMergeJoin(U_{col1}, P_{col2})$:
$\quad t\_proc(U_{col1}) \oplus c\_proc(P^*_{col2}) \oplus t\_proc(V)$
$V \leftarrow SimpleFilter(P_{col1}, P_{col2})$:
$\quad c\_proc(P_{col1}) \oplus t\_proc(P_{col1}) \oplus c\_proc(P^*_{col2}) \oplus t\_proc(V)$

$$t\_proc(P) = P.n \times l^t$$
$$c\_proc(P) = P.n \times l^c$$

$P.n$ 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．この検討を踏まえて構築した実行コストモデル T_{Ins}+T_{Mem} を用いて図 6 の実験を推定した結果を図 10 に示す．図 10 では *bitmap* のみの結果を表示しているが，*dict* と *rle*

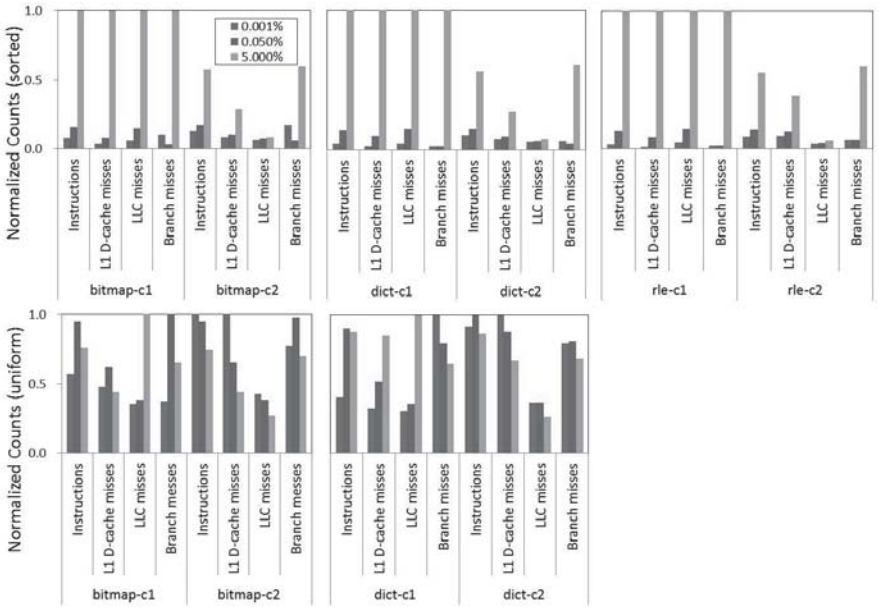

図 9: 図 6 と図 7 における実行命令数/キャッシュミス回数/分岐ミス回数

に関してもほぼ同等の結果が得られている．l^r を 1 として，l^c は *bitmap* の復元の際に実際に発生した CPU サイクルである 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 DB の選択処理のコストには T_{Mem} に補正項 T_{Ins} を加えることで改善ができることが判明した．

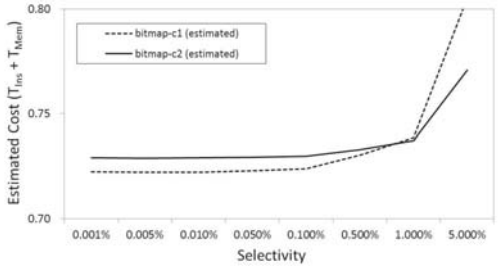

図 10: 実行計画（*bitmap*）のコスト推定

5. おわりに

本論文では，蓄積されたデータを列指向 DB で分析を行う場合に発生する，圧縮された列データに関するクエリ処理の課題を取り扱った．先行文献で提案されている復元せずに直接的にクエリの述語を評価可能な 3 つの圧縮形式を実装し，実行計画の違いによる実行時間の評価，各物理プランのコストが実行時間に与える影響の考察を行った．今後も継続して，Join や Sort などの物理プランに対するコストモデルの検証を行い，より汎用的な列指向 DB のコストモデル構築に取り組む予定である．

[文献]

[1] Daniel J. Abadi et al. “Column-stores vs. row-stores: how different are they really?”, Proc. of SIGMOD, 2008.

[2] Marcin Zukowski et al. “DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing”, Proc. of DaMoN, 2008.

[3] George P. Copeland and Setrag N. Khoshafian “ A decomposition storage model”, Proc. of SIGMOD, 1985.

[4] Mike Stonebraker et al. “C-store: a column-oriented DBMS”, Proc. of VLDB, 2005.

[5] R. Ramakrishnan and J. Gehrke “Database Management Systems”, McGraw-Hill, 2 edition, 2000.

[6] P. A. Boncz et al. “MonetDB/X100: Hyper-Pipelining Query Execution”, Proc. of CIDR, 2005.

[7] Daniel Abadi, et al. “The Design and Implementation of Modern Column-Oriented Database Systems”, Foundations and Trends in Databases, 2013.

[8] Stefan Manegold et al. “Generic Database Cost Models for Hierarchical Memory Systems”, Proc. of VLDB, 2002.

[9] Stephan Muller et al. “An In-Depth Analysis of Data Aggregation Cost Factors in a Columnar In-Memory Database”, Proc. of DOLAP, 2012.

[10] Hector Garcia-Molina et al. “Database Systems: The Complete Book”, Prentice Hall; 2 edition, 2008.

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

若森 拓馬 **Takuma WAKAMORI**
日本電信電話株式会社 NTT ソフトウェアイノベーションセンタ研究員．2013 年横浜国立大学大学院工学府博士前期課程修了，修士（工学）．分散処理基盤の高速化アルゴリズムの研究に従事．日本データベース学会会員．

寺本 純司 **Junji TERAMOTO**
NTT ソフトウェアイノベーションセンタ勤務．日本データベース学会会員．

西村 剛 **Go NISHIMURA**
NTT ソフトウェアイノベーションセンタ勤務．日本データベース学会会員．