

# データの部分集約による 高速かつ正確なデータ集計処理の実現

小山田 昌史<sup>1,a)</sup> 陳 テイ<sup>1,b)</sup> 成田 和世<sup>1,c)</sup> 荒木 拓也<sup>1,d)</sup>

概要：データの集計処理はデータ分析やデータ整形に頻出する重要な処理である。本稿は、データウェアハウスにおける大規模データの集計処理を高速かつ正確におこなう部分集約法を提案する。部分集約法はデータを複数のブロックに分割したうえでブロックごとに集約演算結果を事前に算出し、その後の集約処理では事前に算出した集約演算結果を組み合わせることで、高速かつ柔軟に正確な処理結果を算出する。集約演算の対象となるデータが選択演算によりフィルタされる場合、ブロック毎に事前に算出した集約演算結果が利用可能かどうかをブロックのスキャンなしに判定することで、可能な限りデータ I/O と計算量を削減する。プロトタイプ実装による予備実験により、部分集約法は集約処理前の選択演算の選択率が高い場合に特に有効であることがわかった。

キーワード：集約処理，軽量インデックス，データウェアハウス，問合せ最適化

## 1. 背景

計算資源の発展に伴うデータの大規模化や蓄積の容易化により、データ分析処理の活用がますます広まっている。データ分析の基礎となる処理のひとつは、データの最大値や標準偏差などの集計値を計算するデータ集計処理である。データ集計処理は、それ自体を分析結果とするような単純な集計処理や [6]、機械学習を含んだ複雑な分析処理の入力値算出など、幅広い応用を持ち、データ分析の中心をなす。

近年、データが大規模化していることからデータ集計処理にかかる時間が長大となり、その高速化が活発な研究課題として取り組まれている [1, 7]。しかし、これら高速化技術は集約処理の際にデータをサンプリングして近似結果を算出することで高速化を実現しているため、データの集計結果が正確でなく、正確な結果が求められる場合に利用可能でない。この問題に対し、本稿は正確なデータ集計結果を高速に算出することのできる部分集約法を提案する。部分集約法はデータを複数のブロックに分割したうえでブロックごとに集約演算結果を事前に算出し、その後の集約処理では事前に算出した集約演算結果を組み合わせること

で高速かつ柔軟に正確な処理結果を算出する。

本稿の貢献は次の通りである。

- (1) 大規模データに対する正確かつ高速な集計処理を可能にする部分集約法 (*partial aggregation method*) を提案する。
- (2) 部分集約法が適用可能な集約処理に必要な特徴を部分計算可能 (*partially computable*) の概念として定式化し、部分集約法が部分計算可能な集約演算を高速化することを例示する。
- (3) 集計対象のデータが選択演算によって元のデータから選別される場合においても部分集約法が利用可能であることを示す。このとき部分集約法が利用可能であるかどうかの判定が、データベースシステムにおける演算子の選択率を推定する問題の特殊ケース (選択率=1 かどうかの判定) であることを示し、その解決策を提案する。
- (4) プロトタイプシステムを用いた実験により、部分集約法の有効性を確認する。

本稿の構成は次のとおりである。はじめに 2 節で部分集約法について述べる。つぎに 3 節で部分集約法を複雑な問合せに適用する際に必要となる部分計算結果の再利用可否判定方式について述べる。その後 4 節で提案方式の有効性を予備実験により確かめ、5 節で関連研究について紹介したあと、6 節でまとめる。

<sup>1</sup> 日本電気株式会社 グリーンプラットフォーム研究所 〒211-8666 神奈川県川崎市中原区下沼部 1753

a) m-oyamada@cq.jp.nec.com

b) t-chen@bu.jp.nec.com

c) k-narita@ct.jp.nec.com

d) t-araki@dc.jp.nec.com

## 2. 部分的集約値の利用による集約処理の高速化

本節は、大規模データに対し正確かつ高速に集計処理をおこなう部分集約法の基本アイデアを説明する。はじめに部分集約法によって高速化できる集約演算の種類について述べ、集約演算のみからなる単純な問合せの高速化について説明したあと、選択演算を含む複雑な問合せの高速化について説明する。

### 2.1 部分計算可能な集約演算

部分集約法は、以下の性質を持つ部分計算可能な集約演算を高速化する。

定義 2.1 (部分計算可能) 集約演算  $f : R(A_1, \dots, A_n) \rightarrow D$ , リレーション  $B = \bigcup_{i=1}^m B_i \in R(A_1, \dots, A_n)$  について、

$$f(B) = g(f(B_1), \dots, f(B_m))$$

となる  $g : D^m \rightarrow D$  が存在するとき、集約演算  $f$  は部分計算可能 (partially computable) であるという。また、このとき  $g$  を  $f$  のコンバイナ (combiner) と呼ぶ。なお、 $R(A_1, \dots, A_n)$  は属性  $A_1, \dots, A_n$  を持つリレーション  $R$  のスキーマである。

例えば、リレーションの属性に関する最大値の計算と和の計算は、どちらも部分計算可能な集約演算である。以下、 $B_1 = \{(1, Tom), (7, Alice)\}$ ,  $B_2 = \{(7, Jake), (17, Yuki)\}$  なるブロックから構成されるリレーション  $B \in Friends(age, name)$  を用いて、具体例を示す。

例 2.1 (属性の最大値) リレーション  $B$  の属性  $A$  の最大値  $Max_A$  は、部分計算可能である。このとき、コンバイナは要素の最大値を算出する関数  $Max$  となる。例えばリレーション  $B$  の属性  $age$  について、

$$\begin{aligned} &Max(Max_{age}(B_1) \cup Max_{age}(B_2)) \\ &= Max(7, 17) \\ &= 17 \\ &= Max_{age}(B) \end{aligned}$$

が成り立つ。

例 2.2 (属性の和) リレーション  $B$  の属性  $A$  の和  $Sum_A$  は、部分計算可能である。このとき、コンバイナは要素の和を計算する関数  $Sum$  となる。例えばリレーション  $B$  の属性  $age$  について、

$$\begin{aligned} &Sum(Sum_{age}(B_1) \cup Sum_{age}(B_2)) \\ &= Sum(8, 24) \\ &= 32 \\ &= Sum_{age}(B) \end{aligned}$$

が成り立つ。

一方、リレーションの属性の濃度 (Cardinality, 一意な値の数) を求める演算は、部分計算可能ではない。

例 2.3 (属性の濃度) リレーション  $B$  の属性  $A$  の濃度  $Cardinality_A$  は、部分計算可能ではない。コンバイナとして要素の和を計算する関数  $Sum$  を考え、リレーション  $B$  の属性  $age$  について、ブロック毎に濃度を計算し足し合わせると

$$\begin{aligned} &Sum(Cardinality_{age}(B_1) \cup Cardinality_{age}(B_2)) \\ &= Sum(|1, 7|, |7, 17|) \\ &= Sum(2, 2) \\ &= 4 \end{aligned}$$

となるが、これは真の濃度  $Sum_{age}(B) = |1, 7, 17| = 3$  と一致しない。

以下、簡単のため単純な集約演算を例にとって説明をするが、提案方式である部分集約法はあらゆる部分計算可能な集約演算を高速化できる。部分計算可能な集約演算には他に属性のヒストグラム計算などがある。

### 2.2 高速化のアイデア

提案方式である部分集約法は、上記の部分計算可能な集約演算の実行時に I/O 量と計算量を削減することで演算を高速に実行する。ここでは、

```

max_simple.sql
SELECT MAX(age)
FROM Members
```

という、*Members* テーブルの最高身長を求める問合せ `max_simple.sql` をつかい、その高速化アイデアを述べる。なお、*Members* テーブルはブロック  $B_1, B_2, B_3$  に水平分割されている、すなわち、

$$Members = B_1 \cup B_2 \cup B_3$$

であるとする。

まず、従来の RDBMS における `max_simple.sql` の処理の流れを述べる。`max_simple.sql` は *Members* テーブルの属性  $age$  の最大値を求める処理であり、その論理プラン (関係代数式) は  $Max_{age}(Members)$  である。この論理プランから生成される単純な実行プランの例を図 1-(1) に示す。図中、*Members* は二次記憶上のテーブルデータ、*Scan* はテーブルデータを読み込んでレコードデータ化するスキャン演算、 $Max_{age}$  はテーブルの  $age$  属性の最大値を算出する集約演算を意味する。ここで、従来の RDBMS では集約演算を並列化するために、*Members* テーブルを複数のブロックへと分割し、図 1-(2) の実行プランを生成することができる。これは最大値の計算が例 2.1 で示したように部分計算可能である、すなわち

$$\begin{aligned} &Max_{age}(Members) \\ &= Max_{age}(B_1 \cup B_2 \cup B_3) \\ &= Max(Max_{age}(B_1), Max_{age}(B_2), Max_{age}(B_3)) \end{aligned} \quad (1)$$

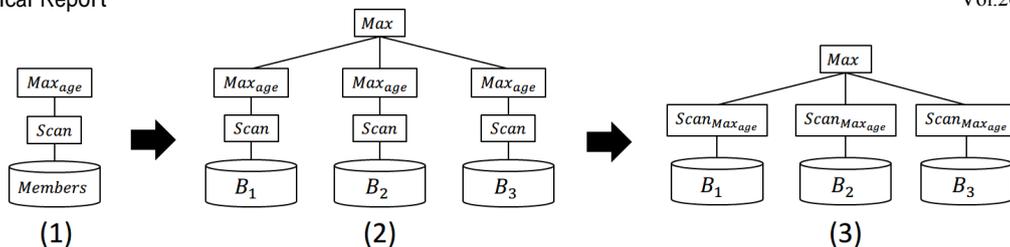


図1 max\_simple.sql の物理プラン最適化

であることによる。このプランは、各ブロック全体をスキャンして属性 *age* の最大値を計算したうえで、それら計算結果の最大値を集約結果として算出する。並列処理が可能であるが、ブロックを全スキャンするため、大量のブロックから構成される巨大なデータに対しては多量の I/O が発生することとなる。具体的には、ブロックサイズを  $B$ 、ブロック数を  $N_B$  としたとき、max\_simple.sql の I/O 量は  $BN_B$  となる。

これに対し、部分集約法はあらかじめブロック毎に属性の最大値などを部分的に計算し、ヘッダなどの特定領域へ格納しておくことで、実際の集約処理時の I/O 量を削減する。例えば max\_simple.sql に対しては、式 1 の論理プラン内で  $Max_{age}(B_1)$ ,  $Max_{age}(B_2)$ ,  $Max_{age}(B_3)$  がそれぞれブロック  $B_1, B_2, B_3$  で事前計算された属性 *age* の最大値にあたることを利用し、該当部分で事前計算された値を参照するような図 1-(3) の実行プランを生成する。図 1-(3) において  $Scan_{Max_{age}}$  はブロックのヘッダをスキャンしてブロック内の *age* 属性の最大値を参照する処理を意味する。図 1-(3) の実行プランは各ブロックのヘッダのみをスキャンするため、ブロックのヘッダサイズを  $H$  とすると発生する I/O 量は  $N_B H$  となり、一般に  $H \ll B$  であることから、部分集約法の I/O 量は従来方式よりもはるかに少ない ( $N_B H \ll N_B B$ )。

### 2.3 選択演算と集約演算からなる問合せの高速化

前小節で、単純な集約演算の高速化について述べた。本小節では、より実用的な例として選択演算の処理結果に対する集約演算の高速化について述べる。ここでは *Members* テーブルをサービスの会員とみなし、退会済みの会員の ID が *RetiredMemberID* テーブルに格納されるとしたうえで、

```

max_selection.sql
SELECT MAX(Height)
FROM Members
WHERE NOT EXISTS
  (SELECT 1
   FROM RetiredMemberID
   WHERE RetiredMemberID.ID = Members.ID)

```

なる、退会していない会員の最高身長を求める問合せ max\_selection.sql を考える。また、*Members* テーブルのブロックの具体的な内容は表 1~3 に示すものとする。

max\_selection.sql の論理プランは選択演算を  $\sigma_C$  と表記すると  $Max_{age}(\sigma_C(Members))$  となる。対応する物理プランを図 2-(1) に示す。ここで、選択演算  $\sigma_C$  について、

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$

が成り立つ [3] ことと、*Members* テーブルが複数のブロックへ分割されていること、そして最大値の計算が部分計算可能であることから、論理プランは

$$\begin{aligned}
 &Max_{age}(\sigma_C(Members)) \\
 &= Max_{age}(\sigma_C(B_1 \cup B_2 \cup B_3)) \\
 &= Max_{age}(\sigma_C(B_1) \cup \sigma_C(B_2) \cup \sigma_C(B_3)) \\
 &= Max(Max_{age}(\sigma_C(B_1)), Max_{age}(\sigma_C(B_2)), Max_{age}(\sigma_C(B_3)))
 \end{aligned} \tag{2}$$

と変形できる。変形後の論理プランに対応する物理プランを図 2-(2) に示す。このとき、前小節で述べたようにブロック毎に事前計算した最大値を再利用することで I/O 量を削減したい。しかし、このプランでは選択演算が集約演算の前に実行されるため、事前計算で最大値となったレコードが選択演算により除去され問合せ対象とならない可能性があり、事前計算した最大値を再利用することができない。

この問題に対し、提案システム部分集約法はブロックによっては選択演算によって全てのレコードが選択される、すなわち事前計算の結果が再利用できることに着目し、ブロック毎に選択演算で全てのレコードが選択されるか否かを判定し、最大限に事前計算した値を再利用しようとする。例えば *RetiredMemberID* テーブルの内容が  $\{(1), (3), (24)\}$  である、すなわち  $id = 1, id = 3, id = 24$  の会員が既に退会しているとする。このとき部分集約法は 3 節で述べる方式によりブロック  $B_2$  と  $B_3$  については選択演算で全てのレコードが選択される、すなわち退会した会員が存在しないことを突き止める。これは  $\sigma_C(B_2) = B_2, \sigma_C(B_3) = B_3$  を意味するため、部分集約法は式 2.3 の論理プランを

$$\begin{aligned}
 &Max(Max_{age}(\sigma_C(B_1)), Max_{age}(\sigma_C(B_2)), Max_{age}(\sigma_C(B_3))) \\
 &= Max(Max_{age}(\sigma_C(B_1)), Max_{age}(B_2), Max_{age}(B_3))
 \end{aligned}$$

と変形し (物理プラン: 2-(3)),  $B_2$  と  $B_3$  について事前計算した値を参照するよう物理プラン 2-(4) を生成して実行する。このときブロック  $B_1$  については事前計算した値が使いまわせないため全スキャンが必要となるが<sup>\*1</sup>,  $B_2$

\*1 選択演算の条件式を利用してブロックの読み込みをスキップすることもできるため、必ずしも全スキャンは必要でない。

表1 ブロック  $B_1$

id	height	age
min: 1	min: 164.2	min: 6
max: 4	max: 178.0	max: 51
1	178.0	8
2	172.5	51
3	152.5	6
4	164.2	18

表2 ブロック  $B_2$

id	height	age
min: 5	min: 60.0	min: 12
max: 8	max: 155.5	max: 14
5	140.0	13
6	155.5	14
7	100.5	13
8	60.0	12

表3 ブロック  $B_3$

id	height	age
min: 9	min: 100.0	min: 10
max: 12	max: 140.2	max: 19
9	100.0	15
10	125.5	17
11	134.5	19
12	140.2	10

と  $B_3$  に関してはヘッダのスキャンのみでよい。そのため、単純な実行方式と比べると I/O 量が減り、高速となる。一般に、全てのレコードが選択されるブロックの数を  $N_{\lambda=1}$  とすると、`max_selection.sql` の実行に必要な I/O 量は  $N_{\lambda=1}H + (N_B - N_{\lambda=1})B$  となる。

### 3. 部分計算結果の再利用可否判定

2.3 節で、選択演算と集約演算を含む問合せをブロック毎に事前計算した部分計算結果を利用して高速化する方式について述べた。本節では、その際に必要となるブロック毎に部分計算結果が再利用可能か否かを判定する方式について述べる。

#### 3.1 問題定義

2.3 節で、ブロックの部分計算結果を再利用するためには、そのブロック内の全てのレコードが選択演算によって全て選択されると保証されていなければならないことを述べた。そのため、本節の残りでは以下の問題に取り組む。問題 1 (選択演算の除去可否判定) ブロック  $B$  と選択演算  $\sigma_C$  が与えられたとき、ブロックのレコードを全スキャンすることなく

$$\sigma_C(B) = B$$

であるかどうかを判定する。

なお、判定結果は false-negative 性を持っていてもよい。すなわち、実際は全てのレコードが選択される ( $\sigma_C(B) = B$ ) のに、選択されないレコードがある ( $\sigma_C(B) \neq B$ ) と判定してしまってもよい。この場合、本来ブロック  $B$  においては部分計算結果を再利用できるにもかかわらずブロックをスキャンして計算し直すため性能面でペナルティとなるが、計算結果は変化しないためである。他方、判定結果の false-positive 性、すなわち実際は選択されないレコードがあるのに全てのレコードが選択されると判断してしまうことは認めない。これは、誤った部分計算結果を利用することによって計算結果が誤った値となってしまうためである。表 4 にこれらの関係をまとめる。

本節の残りでは、上記の問題 1 の解決方式を二つ説明する。

#### 3.2 軽量インデックスを利用した判定方式

まず、軽量インデックスを利用した判定方式について説明する。軽量インデックスとは、属性ごとの最大値と最小値を各ブロックのヘッダ部分に格納したものであり、二次記憶ベースのデータベースシステムにおいて、選択演算  $\sigma_C$  でスキャン対象のブロック  $B$  のスキャン処理をスキップすることができるか、すなわち  $\sigma_C(B) = \emptyset$  となるかを判定するためにつかわれてきた [8, 10, 11]。これに対し、本判定方式は軽量インデックスを  $\sigma_C(B) = B$  の判定に活用する。

例えば表 1~3 に示した各ブロックについて  $\sigma_{age>20}(B) = B$  かどうかを判定することを考える。軽量インデックスから  $B_1$  内のレコードの *age* 属性の値は [6, 51] の範囲にあることがわかるので、 $age > 20$  となるレコードを持つ可能性があり、 $\sigma_{age>20}(B_1) = B_1$  とはいえない。他方  $B_2$  については軽量インデックスの値から *age* 属性の値が [12, 14] の範囲にあることがわかり、この中に明らかに  $age > 20$  となるレコードは含まれていないことから、 $\sigma_{age>20}(B_2) = B_2$  であることがブロック内のレコードをスキャンすることなく判定できる。 $B_3$  についても同様に  $\sigma_{age>20}(B_3) = B_3$  がいえる。

軽量インデックスをつかった判定は、ブロック内部の属性の最大値と最小値の差異が大きい場合、判定の精度が。例として 2.3 節で述べた `max_selection.sql` を考える。`max_selection.sql` は *Members* テーブルの *age* 属性に関する最大値を算出するが、*Members* テーブルの各レコードについて *RetiredMemberID* テーブルのなかに *RetiredMemberID.id = Members.id* となるようなレコードがひとつでも存在するかどうかを確認し、存在する場合はそのレコードを集約対象から除外する。ここで *RetiredMemberID* テーブルのブロックの内容が {(1), (3), (24)} である、すなわち  $id = 1, id = 3, id = 24$  の会員が既に退会しているとする。このとき、*RetiredMemberID* テーブルの *id* 属性の軽量インデックスは [1, 24] となり、 $B_1 \sim B_3$  の全てのブロックで *RetiredMemberID.id = Members.id* が成立し得る (各ブロックの *id* の範囲が *RetiredMemberID* テーブルの *id* の範囲に完全に含まれる) ため、選択演算で全レコードが選択されるということができない。

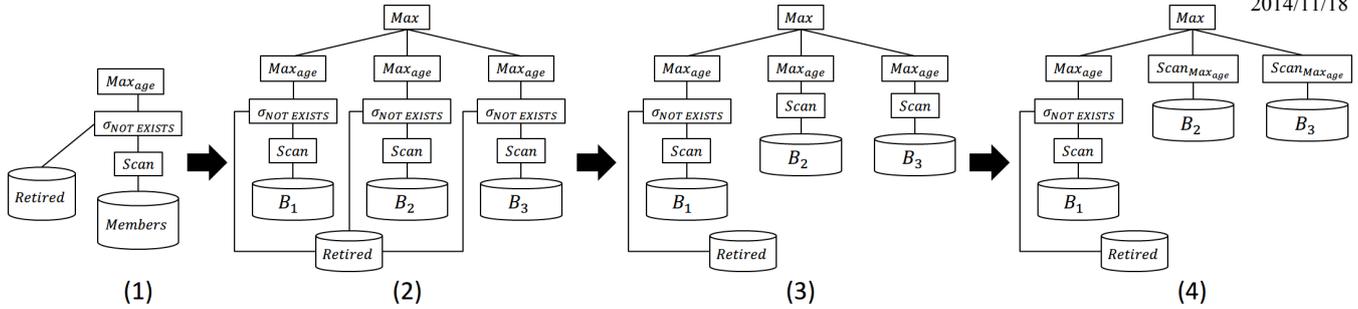


図2 選択演算と集約演算を含んだ実行プランの最適化の流れ

表4 部分計算結果の再利用可否

		実際の値	
		$\sigma_C(B) = B$	$\sigma_C(B) \neq B$
部分集約法による判定結果	$\sigma_C(B) = B$	計算結果を再利用 (positive)	N/A (false positive)
	$\sigma_C(B) \neq B$	再利用なし (false negative)	再利用なし (negative)

### 3.3 ビット列により属性値集合を近似する判定方式

次に、ビット列の利用による、ブロック内の属性値の最大値と最小値の差異に判定の精度が影響されにくい判定方式を提案する。この方式は、ブロック内の各属性の値の集合を近似した小サイズのビット列を利用し、ブロック全体のスキャンなしにブロック内の全レコードが選択されることを判定する。ビット列をブロックのヘッダ部分などに格納しておくことで、ブロックの全スキャンを避けることができる。

ビット列の算出方法と判定方式は多岐にわたる。例えば max\_selection.sql においては、Members テーブルのブロック  $B$  について  $\sigma_C(B) = B$  となるかどうかの判定が、RetiredMemberID.id = Members.id となるレコードがひとつでも存在するかどうかの判定となる。これは、次のようにして解決できる。まず、ビット列を単純な  $bit\_pos(x) = \lfloor \frac{x-1}{4} \rfloor \bmod 8$  なる関数  $bit\_pos$  とアルゴリズム 1 に示す処理から得られるような単純な量子化によって算出する。そして、ブロック  $B$  の属性  $id$  から得られたビット列と、RetiredMemberID テーブルの属性  $id$  から得られたビット列との AND を計算する。結果が 0 となった場合は RetiredMemberID.id = Members.id となるレコードはひとつも存在しないため、 $\sigma_C(B) = B$  と判定し、ブロック  $B$  に対する計算結果を再利用する<sup>\*2</sup>。

具体的例として、軽量インデックスの利用方式と同様に RetiredMemberID テーブルのブロックの内容を  $\{(1), (3), (24)\}$  とし、上記の判定方式を考える。すると RetiredMemberID テーブルの  $id$  属性について 00100001 が、Members テーブルの  $id$  属性については  $B_1$  で 00000001,  $B_2$  で 00000010,  $B_3$  で 00000100 が、それぞれ得られる。AND 結果は  $B_1$  で 1,  $B_2$  と  $B_3$  で 0 となるので、 $B_2$  と  $B_3$  につい

<sup>\*2</sup> 結果が 0 でない場合は該当するレコードが存在すると判定するが、これには false-positive 性がある。すなわち、本来は該当レコードが存在しないのに、存在すると判定ミスをしてしまうことがある。その場合、本当は再利用できる計算結果を再利用することができない。

#### Algorithm 1: 単純なビット近似

**Input:**  $A_{ij}$ , ブロック  $B_i$  における属性  $A_j$  の値の集合  
**Output:**  $A_{ij}$  のビット近似

```

1  $b \leftarrow 0$ 
2 foreach  $x \in A_{ij}$  do
3    $b \leftarrow b \mid bit\_pos(x)$ 
4 return  $b$ 

```

ては計算結果を再利用することができる。計算結果をどのブロックでも再利用できなかった軽量インデックスの利用方式と比べ、よい結果となっているといえる。

なお、ビット列をつかって集合を近似したうえで積集合を高速・高精度に計算するための試みは、Bloom Filter の応用を中心として過去に研究 [4, 5] がなされており、それらの方式を利用して精度を向上させることも考えられる。

## 4. 予備実験

本節ではプロトタイプ実装による予備実験の結果を述べ、部分集約法の有効性を確かめる。

### 4.1 実験環境

**システム:** 部分集約法のプロトタイプ実装は C++ 言語でおこない、コンパイラとしては Clang++ 3.4 を用いた。実験に用いた計算機は、プロセッサが Intel Core i5 680 (2 コア, 3.60GHz), メモリが 16GB, OS が Linux 3.13.0 である。

**テーブルデータ:** 実験では Member(height, age) なるテーブルを用いた。Member テーブルは float 型の属性 height と int 型の属性 age を持つ。どちらの属性も値が一様分布となるように人工データを生成している。テーブルの保存フォーマットには、ORC File/Parquet の構造に基づく単純化されたカラムフォーマットを実装して用いた。このフォーマットは、1 レコードを 32byte で格納する。レコードは一定数ごとにブロックへとまとめられ、各ブロックはヘッダに次のブロックへのポインタと各属性の最大値・最小値を

格納する．ヘッダのサイズは実験を通して 64byte とした．  
なお、実験ではブロックのサイズが性能に与える影響も確認した．

集約問合せ: 集約問合せとしては、一定以上の年齢の人物の最大身長を算出する下記の問合せ max\_height.sql を用いた．

```

max_height.sql
SELECT MAX(height)
FROM Members
WHERE age > ?
    
```

実験では年齢条件を指定する ? へ値を設定し、選択演算がさまざまな選択率をとるようにしている．

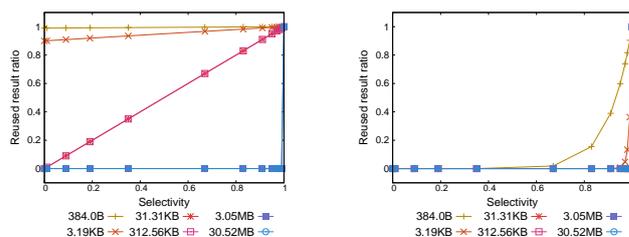
#### 4.2 ブロックのサイズと再利用率

はじめに、ファイルフォーマットにおけるブロックのサイズが部分集約法における部分集約結果の再利用率へ与える影響について議論する．

10,000,000 個のレコードからなる Member テーブルを用意し、ひとつのブロックに格納するレコード数 (= ブロックサイズ) を 10 レコードから 1,000,000 レコードまで一桁ずつ変化させ、それぞれファイルへと保存した．このとき、ブロックひとつのデータサイズは 10 レコードで 384B、1,000,000 レコードで 30.52MB となる．また、各ブロックサイズ毎に、age 属性の値でレコードをソートしたファイルと、ランダムな順番でレコードを並べたファイルの二種類を用意した．

図 3 は、各ファイルについて集約処理 max\_height.sql の選択演算の選択率を 0 から 1 まで変化させて実行した際に、部分集約法が部分集約結果を再利用できたブロックの割合を示す．図 3-(a) を見ると、レコードが age 属性についてソートされている場合、選択演算の選択率が高くなる（より多くのデータが選択演算により選択されるようになる）ほど、部分集約法が部分集約結果を再利用できるようになる割合が高まるのが分かる．これは、選択率が高くなるほど多くのデータが選択されるような場合、ブロック  $B$  において  $\sigma_C(B) = B$  となる確率が高まるためである．また、ブロックサイズが小さいほど、選択率に関係なく部分集約結果を再利用できていることもわかる．これも、ブロック内のタプル数が増加するほどブロック  $B$  において  $\sigma_C(B) = B$  となる確率が高まるためである．図 3-(b) は、レコードが age 属性についてソートされていない場合の結果をあらわす．ソートされていない場合は選択率が下がると再利用率が大幅に減少してしまうことがわかる．

これらのことから、部分集約法は選択演算の選択率が高い場合に特に有効であることが推察できる．



(a) レコードを age 属性でソートした場合 (b) レコードをランダムに格納した場合

図 3 各ブロックサイズにおいて選択演算の選択率を変化させた際の部分集約結果の再利用率の変化

#### 4.3 ブロックのサイズと実行時間の関係

次に、ファイルフォーマットにおけるブロックのサイズが部分集約法における部分集約結果の再利用率へ与える影響について議論する．

前小節と同様の状況で実験をおこない、選択演算の選択率を変化させて集約演算の実行時間を計測した．また、各処理について計測を二度連続しておこない、データが OS のページキャッシュに載っている（メモリ上にある）場合と、そうでない場合での違いを確認した．なお、全ての計測の前には echo 1 > /proc/sys/vm/drop\_caches によって OS のページキャッシュを消去している．

図 4 は、ディスク上のデータを処理した場合の実験結果を示す．ほとんどのブロックサイズでは選択率に実行時間が影響されていない．これは、ディスクの最小読み込み単位（ページサイズ、通常 4KB）が影響を与えたものと考えられる．ブロックサイズがページサイズよりも大きくない場合、部分集約法でブロックを読み飛ばしてヘッダのみを読もうとしても、同時にブロックを読み込んでしまい、部分集約法による高速化効果が小さくなってしまふ．ここから、部分集約法はブロックサイズが記憶媒体の読み込み最小単位よりも十分に大きい場合に有効であると推察できる．

図 5 は、データがページキャッシュに載った状態での実験結果を示す．データがディスクにあるときとは異なり、選択率の差（再利用率の差）が多くの方式で実行時間に影響を与えていることがわかる．これは、メモリにおける最小の読み込み単位がディスクと比べ大幅に小さいため、部分集約法の効果が大きくあらわれたためと考えられる．

### 5. 関連研究

#### 5.1 Data Skipping

テーブルを複数のブロックへ水平分割しブロック毎に軽量インデックスを持つという試みは、問合せの実行時にブロックのスキャンを省いて高速化を図る Data Skipping の領域で研究されてきた [8, 10, 11]．3.2 節で述べたように、これら Data Skipping 技術は軽量インデックスを、選択演算  $\sigma_C$  でスキャン対象のブロック  $B$  のスキャン処理をスキップすることができるか、すなわち  $\sigma_C(B) = \emptyset$  となるかを

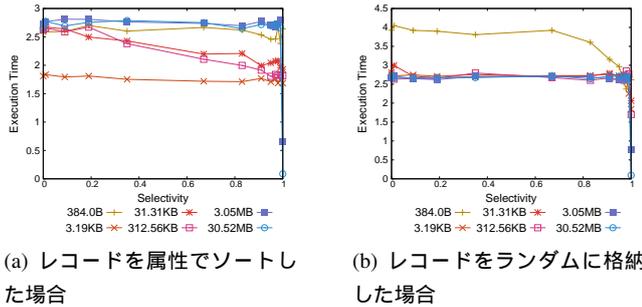


図4 各ブロックサイズにおいて選択演算の選択率を変化させた際の  
実行時間の変化（データがディスクにある場合）

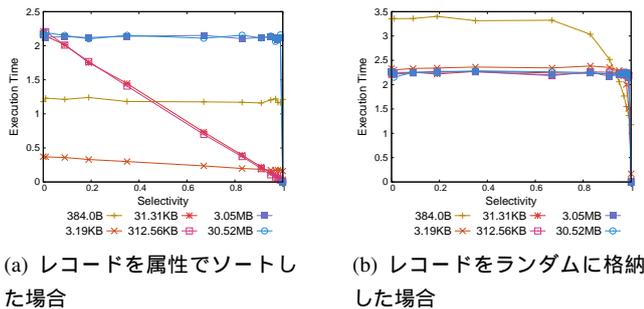


図5 各ブロックサイズにおいて選択演算の選択率を変化させた際の  
実行時間の変化（データがメモリにある場合）

判定するために利用する．これに対し，部分集約法では軽量インデックスを，対象のブロックの結果が選択演算により全て選択され部分計算結果が再利用できるか，すなわち  $\sigma_C(B) = B$  となるかの判定に活用する．Data Skipping の分野では近年ブロックのスキップ効率を高めるためのレコードの並び順を求める試み [11] があり，これは部分集約法で部分計算結果の再利用効率を高めるために利用することができると思われる．

## 5.2 集約演算の高速化

集約演算の高速化アプローチのひとつは，データのサンプリングである．BlinkDB [1] は Hive を拡張し，HDFS 上の処理対象データをサンプリングし近似結果を算出することで，集約演算を含んだ問合せへの応答時間を短縮している．BlinkDB は問合せへの応答時間と近似による誤差のトレードオフ関係をシステマティックに扱い，ユーザの要求する応答時間から，結果に生じる最大誤差を算出することができる．[7] にも同様の試みがある．

## 5.3 部分計算

部分計算の概念は，並列計算の技術に関連する．データを複数に分割し，分割されたデータを並列に処理したうえで最終的にマージする処理は，並列計算の基礎となっている [9]．このようにして並列計算ができる演算は部分計算が可能であるため，本稿で提案した部分集約法が適用できる．MapReduce [2] における Combiner も同様の考え方に

もとづく．

## 5.4 選択率の推定

3 節で取り組んだ計算結果の再利用可能性判定問題は，選択演算の選択率推定の極端な例（選択率  $\lambda = 1$  であるかどうかの判定）といえる．選択演算の一般的な選択率判定に関する試みとしては [12] があるが，この方式は部分集約法の要求する false-negative 性を有さないため，3 節で取り組んだ問題にそのまま応用することはできない．

## 6. 結論

本稿は，データウェアハウスシステムにおける大規模データの集計処理を高速かつ正確におこなう部分集約法を提案した．部分集約法はデータを複数のブロックに分割したうえでブロックごとに集約演算結果を事前に算出し，その後の集約処理では事前に算出した集約演算結果を組み合わせることで，高速かつ柔軟に正確な処理結果を算出する．集約演算の対象となるデータが選択演算によりフィルタされる場合，ブロック毎に事前に算出した集約演算結果が利用可能かどうかをブロックのスキャンなしに判定することで，可能な限りデータ I/O と計算量を削減する．プロトタイプ実装による予備実験により，部分集約法は集約処理前の選択演算の選択率が高い場合に特に有効であることがわかった．

## 参考文献

- [1] S. Agarwal *et al.* BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42. ACM, 2013.
- [2] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.
- [3] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.
- [4] D. Guo *et al.* Theory and Network Applications of Dynamic Bloom Filters. In *INFOCOM. IEEE*, 2006.
- [5] M. C. Jeffrey, J. G. Steffan. Understanding bloom filter intersection for lazy address-set disambiguation. In *SPAA*, pages 345–354, 2011.
- [6] S. Melnik *et al.* Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.
- [7] S. Nirkhivale, A. Dobra, C. M. Jermaine. A Sampling Algebra for Aggregate Estimation. *PVLDB*, 6(14):1798–1809, 2013.
- [8] V. Raman *et al.* DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB*, 6(11):1080–1091, 2013.
- [9] S. H. Roosta. *Parallel processing and parallel algorithms - theory and computation*. Springer, 2000.
- [10] L. Sun *et al.* Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, pages 1115–1126, 2014.
- [11] L. Sun *et al.* A Partitioning Framework for Aggressive Data Skipping. *PVLDB*, 7(13):1617–1620, 2014.
- [12] P. Terlecki *et al.* Filtered statistics. In *SIGMOD*, pages 897–904, 2009.