

多次元データに対する集約演算の効率化手法における データ挿入スループットの向上

渡 佑也[†] 櫛 惇志[†] 宮崎 純[†] 中村 匡秀^{††}

[†] 東京工業大学 情報理工学院情報工学系 〒152-8550 東京都目黒区大岡山 2 丁目 12-1

^{††} 神戸大学大学院 システム情報学研究科 〒657-8501 兵庫県神戸市灘区六甲台町 1-1

E-mail: [†]{watari,keyaki}@lsc.cs.titech.ac.jp, ^{††}miyazaki@cs.titech.ac.jp, ^{†††}masa-n@cs.kobe-u.ac.jp

あらまし 本研究では、分散キーバリューストア上に蓄積された多次元データに対する集約演算の効率化手法において、高いデータ挿入スループットを実現する手法を提案する。我々は、センサデータのような大規模かつ多次元のデータに対する集約演算を効率よく行う手法として、部分集約法を適用してデータのスキャン量を削減した上で、リレーショナルデータベースと分散キーバリューストアの双方を活用する手法を過去に提案した。この手法は、高いスループットで集約演算を実現するものの、既存の方式と比べデータ挿入スループットが低いという問題が存在する。本研究では、データ空間におけるインデックス更新処理を複数台のマシンで並列して行うことで、高いデータ挿入スループットを実現する。また、データ挿入および集約演算のクエリ処理の性能を従来手法と比較する評価実験を行った結果、提案手法ではクエリ処理スループットを低下させることなく、データ挿入スループットの向上を達成できることを示した。

キーワード 集約演算, 範囲クエリ, 分散キーバリューストア, リレーショナルデータベース, 挿入スループット

1. 背景

近年、ネットワーク技術の高度化により様々な機器がインターネットに接続され、膨大な量のデータが収集できるようになってきた。そのような情報を活用するための分析手段の一つとして最大値や平均値などの集約演算を用いるものが存在する。

たとえば、自動車のカーナビゲーションシステムから得られる走行データのようなセンサデータを考える。このデータに対して「ある二次元の矩形内に存在する自動車の平均移動速度」を求めることにより、その地域内の渋滞状況を知ることができる。また、同じ集約処理を時間軸をスライドさせながら行うことで、平均移動速度の時間推移をグラフにすることができる。

こうした処理を実現するためには様々な課題が存在する。まず、センサデータは頻繁に更新されるため、それを扱うデータベースは高いデータ挿入スループットを提供しなければならない。また、多次元データを効率的に扱って集約演算を行う必要がある。

このような課題に対しては、過去に多数の研究がなされてきた。西村ら [7] による MD-HBase では、多次元データを空間充填曲線を用いて一次元化することで、インデックスの存在しないキーバリューストア (KVS) [2] でも多次元データを効率的に扱う手法を提案した。また、小山田ら [9] は、データの部分的な集約結果をあらかじめ計算しておき、集約処理で可能な限り再利用する部分集約法 [8] を用いて、クエリ範囲が一次元の集約演算を効率化できることを示した。

さらに、我々は、リレーショナルデータベース (RDB) [1] と分散キーバリューストア (分散 KVS) の利点を相補的に活用し、多次元データに対する集約演算を効率化する手法 [10] を提案し

た。この手法は、部分集約法のアイデアに基づき、多次元データが属するデータ空間を複数のグリッドに分割して、グリッドごとに部分集約値を事前に計算するものである。その際、多次元データを効率的に扱うことができる RDB と大規模データに対してスケールアウトしやすく設計されている分散 KVS の利点を相補的に活用した。我々は、この手法を用いて高い集約演算のクエリ処理スループットが実現できることを示したが、既存方式に比べてデータ挿入スループットが低いという問題が明らかとなった。我々が検証実験を行った結果、データ挿入スループットの低下を引き起こす要因として、グリッド分割などのインデックス更新処理をクライアント側で行っていたために分散システム上ではコストのかかる排他制御が必要であったことと、クラスタ内での負荷分散が十分でなかったことが判明した。

本研究では、我々の従来手法 [10] におけるインデックス更新処理をサーバ上でバックグラウンドで行うよう変更して排他制御のコストを削減するほか、RDB のレプリケーションと分散 KVS のキーを改良することにより負荷分散を効率化し、データ挿入スループットを向上させる手法を提案する。さらに、データ挿入スループットと集約演算のクエリ処理スループットを従来手法と比較する。

2. 先行研究

本節では、集約演算を効率化する手法の一つである部分集約法 [8] について、2.1 節で説明する。続いて、部分集約法を多次元データに適用した上で、RDB と分散 KVS を相補的に活用することにより集約演算を効率化する我々の従来手法 [10] について、2.2 節以降で説明する。

2.1 部分集約法

部分集約法 [8] とは、データの部分的な集約結果を可能な限り再利用して全体の集約結果を効率的に求める手法のことである。

データ集合 B が存在し、それをいくつかのブロックに分割することを考える。ここでは、図 1 のように 3 個のブロック B_1, B_2, B_3 に分割する。ただし、これらのブロックは互いに共通部分を持たず、和集合が元のデータ集合 B に一致するものとする。部分集約法では、各ブロック B_i についての集約値 $f(B_i)$ を事前に計算する。ただし、集約関数 f は部分集約可能、すなわちデータの部分的な集約結果から全体の集約結果が求まるものに限る。

B に対してある選択演算 $\sigma(\cdot)$ を適用した結果の集約値 $f(\sigma(B))$ を求めたいとする。このとき、各ブロックに選択演算を適用した結果が次の通りであったとする。

$$\sigma(B_1) = B_1, \sigma(B_2) = \emptyset, \sigma(B_3) \neq B_3 \text{ かつ } \sigma(B_3) \neq \emptyset$$

ブロック B_1 はデータのすべてが選択されるため、事前に計算した値が再利用できる。また、ブロック B_2 については、データが一つも選択されないため無視できる。したがって、データのスキューン、選択演算によって一部が選択されるブロック B_3 についてのみ行えばよい。集約演算を効率化できる。

2.2 部分集約法の多次元データへの適用

我々の従来手法 [10] では、部分集約法を多次元データに適用する。まず、データを多次元空間上の点と考え、その空間を図 2 のように複数のグリッドと呼ばれる超直方体に分割する。そして、グリッドごとに集約値を事前に計算して保持しておく。この値のことを部分集約結果と呼ぶ。このように、グリッドに分割されたデータ空間とグリッドごとの部分集約結果を本稿ではインデックスと呼ぶことにする。また、データの挿入に伴って発生するグリッド分割や部分集約結果の更新処理をインデックスの更新と呼ぶ。

以下では、グリッド分割とクエリ処理について説明する。

2.2.1 グリッド分割

グリッド分割は、k-d tree [5] に従う。すなわち、データ空間のある軸に垂直な超平面でグリッドを分割する。このときに分割される軸は循環的に選択される。k-d tree では、データの中央値付近で分割することにより木が平衡化されるが、中央値の

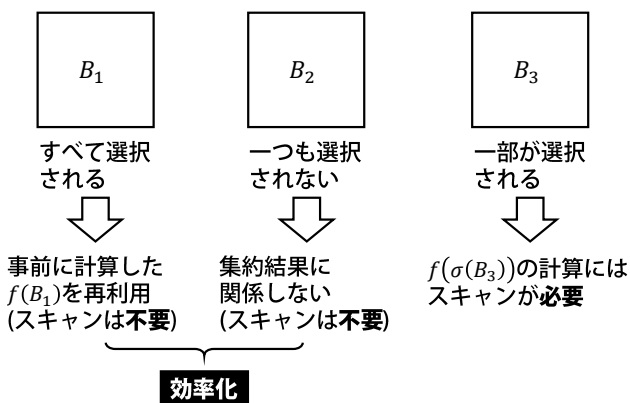


図 1 部分集約法の概要

計算にはコストがかかるため、我々の手法では、中央値の近似値として平均値を用いている。また、分割はグリッド内に含まれるデータが閾値 $N_{\text{threshold}}$ を超えた場合に開始され、データ数がグリッドサイズ N_{size} 以下になるまで再帰的に行われる。

2.2.2 クエリ処理

インデックスを利用して集約演算のクエリを処理する手順について図 2 を参照しながら説明する。図 2 においてクエリ範囲は破線で示されている。

まず、クエリ範囲と共通部分を持つグリッドを列挙する。図 2 の場合、グリッド 000, 00110, 00111 の 3 個のグリッドが列挙される。

続いて、これらのグリッド内のデータの集約値を計算する。クエリ範囲に完全に包含されるグリッド（図 2 ではグリッド 00110, 00111）については、あらかじめ計算された部分集約結果がそのまま再利用できるため、グリッド内のデータをスキューンする必要はない。一方で、クエリ範囲と部分的に共通部分を持つグリッド（図 2 ではグリッド 000）は、部分集約結果が再利用できないため、データの全スキューンが必要になる。なお、クエリと全く共通部分を持たないグリッドに対しては、部分集約結果の参照もグリッド内のデータのスキューンも必要ないため、クエリ処理においてコストがかからない。

2.3 RDB と分散 KVS の相補的活用

2.2 節で述べた手法を実現するために必要な情報は、次の 3 種類である。

- (1) データ空間内に存在するグリッドの大きさ・位置の情報
- (2) 実データ
- (3) 部分集約結果

このうち、(1) のデータサイズは実データに比べると十分小さいと考えられる。また、グリッド分割が発生しない限り更新されず、そのグリッド分割の頻度も十分低い。しかし、2.2 節で

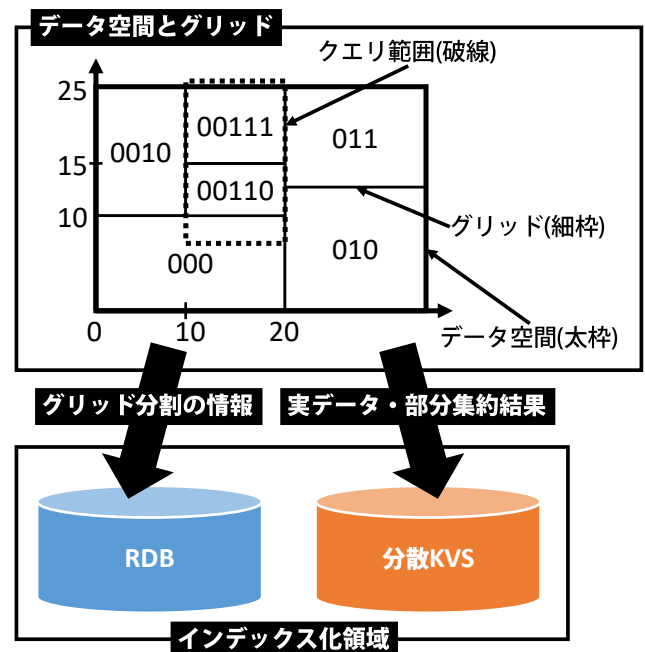


図 2 データ空間とその分割の様子

述べたクエリ処理を行うには、クエリ範囲と共通部分を持つグリッドを列挙する必要がある、この作業は複雑である。そこで本手法では、多次元データを効率的に扱うことのできる RDB で (1) を保持する。

一方で、(2) および (3) のデータサイズは膨大なものになると考えられる。また、データの更新も頻繁に発生する。そこで本手法では、大規模データに対してスケールアウトしやすく設計され、高いデータ挿入スループットを実現できる分散 KVS でこれらのデータを保持する。

このように RDB と分散 KVS の利点を相補的に活用してインデックスを構築したデータ構造を**インデックス化領域**と呼ぶことにする。我々は、RDB、分散 KVS として PostgreSQL、HBase を用いて実装を行い、それらを単体で使った場合と比べ高いクエリ処理スループットが実現できることを示した。

2.4 問題点

従来手法における問題点は、データ挿入スループットがそれまでの手法に比べて低いことにある。我々が過去に行った実験 [10] では、HBase と PostgreSQL を単体で使った場合とデータ挿入スループットを比較した。その結果、それぞれ約 7 分の 1、約 5 分の 1 以下のスループットに低下した。データ挿入スループットを向上させるためには、次の課題が挙げられる。

課題 1 インデックスの更新における排他制御のコスト

我々の従来手法におけるインデックスの更新処理は本来サーバ側で行う必要があるが、データを挿入するクライアントが行っていた。この状況の下で複数のクライアントがデータを挿入する場合、各クライアントがどのようなタイミングでインデックスの更新を行うか予測できないため、インデックスの整合性を確保するために強い排他制御が必要となる。

たとえば、クライアントがあるグリッドに対してデータを挿入している最中に、そのグリッドが分割によって削除されてはならない。従って、データの挿入中にはグリッドに対して何らかのロックを行う必要がある。また、グリッドごとの部分集約結果を更新する場合は、別のクライアントと更新が競合しないように行わなければならない。

我々がこれらの排他制御を HBase 上の CAS (compare-and-set) 操作を用いて実装し、検証実験を行ったところ、排他制御がデータ挿入時間全体の 70% 程度を占めることが分かった。分散システムにおけるアトミックな操作は本質的に高コストであり、上記で述べた排他制御がデータ挿入スループット低下の大きな要因であるといえる。

課題 2 一部のマシンへの負荷集中

従来手法では、RDB と分散 KVS の双方において一部のマシンに負荷が集中する問題が存在した。それぞれについて以下で説明する。

a) RDB (PostgreSQL)

我々の従来手法では、挿入するデータがどのグリッドに属するかについて RDB (PostgreSQL) へ問い合わせる必要があるが、PostgreSQL のサーバは一台しか存在しなかったため、この問い合わせ処理がクライアントの台数に対してスケールせず、ボトルネックになるという問題があった。

b) 分散 KVS (HBase)

HBase では、テーブルはキーに応じて複数の Region に水平分割され、Region Server 間で分散して保持される。従来手法では、キーの先頭に 0~255 の 1 バイトの整数値を付加することで、効率的に分散させることを試みていた。

このとき付加されるバイト値は、データ点が属するグリッド番号の先頭 8 ビットを数値として解釈した値^(注1)である。この方式は、付加する値の大小関係とグリッド番号を文字列として解釈した場合の大小関係が一致するため、グリッド番号に基づく接頭辞検索を行いやすいという利点がある。その反面、特定のグリッドにアクセスが集中した場合は、うまく負荷を分散させることができない。

特に本研究で扱うようなセンサデータは時系列データであるため、ある時点で挿入されるデータの時間軸の値は近いものになる。従って、挿入されるデータが一部のグリッドに集中しやすくなるため、HBase の特定の Region Server に負荷が偏ってしまうという問題がある。

3. 提案手法

2. 節で述べた我々の従来手法を元に、データ挿入スループットを向上するための改良を施す。

まず、課題 1 を解決するため、データを挿入するクライアントではなくサーバ側でインデックスの更新処理を行うよう改良する。これにより、インデックス更新のタイミングを制御しやすくなり、排他制御のコストを削減できる。詳細については、3.1 節で説明する。

また、課題 2 を解決するため、RDB (PostgreSQL) のレプリケーションと HBase のキーの改良を行った。それぞれ 3.4 節、3.2 節で説明する。

3.1 サーバ上でのインデックス更新と中間バッファの設置

課題 1 では、データを挿入するクライアントがインデックスの更新処理を行っていたためコストのかかる排他制御が必要となり、これがデータ挿入スループット低下を引き起こしていたことを述べた。そこで、提案手法では、インデックス更新処理をクライアントではなくサーバ側においてバックグラウンドで行うよう改良した。

図 3 に提案手法におけるデータ構造とデータの流れを示す。提案手法では、クライアントはインデックス化領域に直接データを挿入するのではなく、その中間に存在する**中間バッファ**へ挿入する。なお、この中間バッファは HBase 上に存在するため、データ挿入のスループット増加に対してスケールアウトさせられる。中間バッファへのデータの挿入時には、我々の手法におけるインデックスを一切更新せず HBase に対する単純な挿入を行うため、HBase 本来の高いデータ挿入スループットが発揮できると期待される。

一方、中間バッファのデータに対する集約演算は、部分集約法を適用できないため非効率となる。従って、データが中間

(注1) : 正確には、先頭ビットの次から 8 ビット分を用いる。ビット数が 8 ビットに足りない場合は、末尾に 0 を補う。

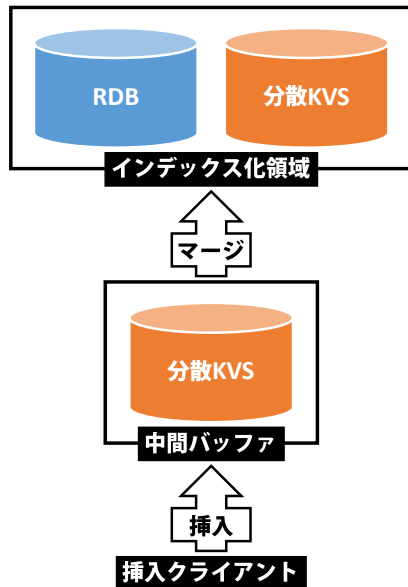


図3 提案手法におけるデータ構造とデータの流れ

バッファに挿入されれば速やかにインデックス化領域へ統合しなければならない。この統合する作業のことをマージと呼び、サーバ側で分散してバックグラウンドで行う。そのアルゴリズムは3.3節で述べる。

上述のように改良することで、クライアントから見たデータ挿入スループットが向上すると考えられる。また、サーバ側で行うインデックスの更新処理（マージ）も、そのタイミングを制御しやすくなるため排他制御のコストを削減でき、効率的に行えるようになる。

3.2 ランダムなキー接頭辞による効率的な分散

提案手法では、HBaseの特定のRegion Serverに負荷が集中することを避けるため、キーの先頭に付加するバイト値をグリッド番号に基づくものから乱数に変更した。具体的には、 $1 \leq P \leq 255$ なる P について、 $[0, P)$ の範囲で一様に分布するバイト値を付加する。 P の値としては、クラスタ内に存在するRegion Serverの台数の倍数にすることが望ましい。これは、HBaseテーブルを接頭辞に応じてRegionに分割する場合、各Region Serverに等しい個数のRegionを割り当てられるためである。なお、この状況の下でRegion Serverの台数が変化した場合、効率的な分散を維持するには、 P の値を変更してキーを再構成する必要がある。

3.3 マージ

提案手法におけるマージの流れを次の図4に示す。マージは、3.2節で述べたキーの接頭辞に基づき複数台のマシンで分散して行う。図4は、三台のマシンA、B、Cでマージを行う様子を示している。3.2節で述べたように接頭辞は乱数であるため、データの分布や挿入順によらず均一に分散できる。

マージ処理を複数台のマシンで分散させるとき、各マシンは自らに割り当てられた接頭辞を持つデータのマージを担当する。その際、 $0 \leq p < P$ の各接頭辞 p は、ちょうど一台のマシンに割り当てられなければならない。これは、同一の接頭辞が複数のマシンに割り当てられてはならず、かつ、どのマシンにも割

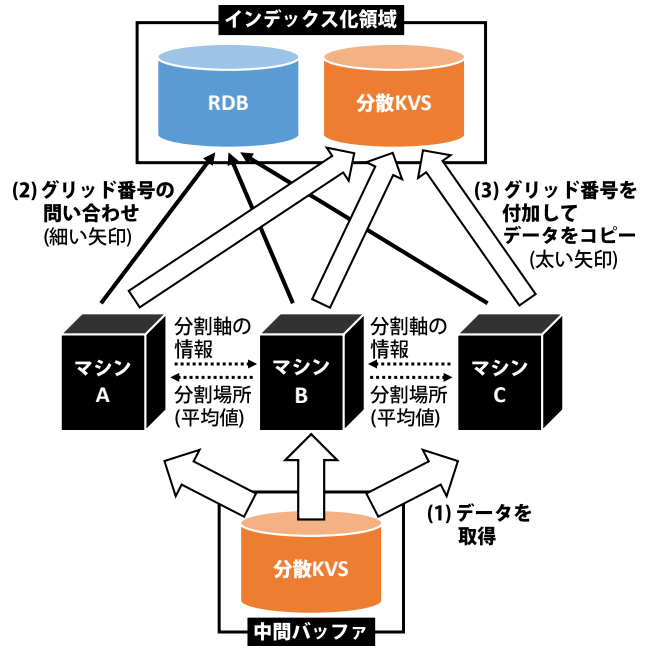


図4 マージの流れ

り当てられない接頭辞があってはならないことを意味する。

各マシンが行うマージのアルゴリズムは次の通りである。

アルゴリズム 1 マージ

- (1) HBaseの中間バッファ内に存在するデータのうち、自らのマシンに割り当てられた接頭辞を持つデータを取得する。
- (2) (1)で取得したデータが属するグリッド番号をPostgreSQLに問い合わせる。
- (3) (1)で取得したデータをHBaseテーブルにコピーする。その際、必要であればグリッド分割を行った上で、グリッド番号をHBaseのキーに付加する。
- (4) (1)で取得したデータを中間バッファから削除する。

(3)において、インデックス化領域に存在するデータ数とバッファ内のデータ数の和がグリッド分割開始の閾値 $N_{\text{threshold}}$ を超えない場合、グリッドの分割は生じない。従って、(2)で取得したグリッド番号を単純に付加すればよい。

一方で、閾値 $N_{\text{threshold}}$ を超える場合、グリッドを分割する必要がある。グリッドを分割するためには、グリッド内のデータのほかにグリッドを分割する場所の情報が必要となる。前者については、アルゴリズム1(1)で得た中間バッファ内のデータのほかに、インデックス化領域にすでに存在するデータを取得する。後者について、具体的に必要になるのは分割軸の平均値である。その計算方法を説明する。

まず、マシンのうちの任意の一台にマスタと呼ばれる役割を与える(図4では、マシンBがこの役割を持つ)。マスタは各マシンからグリッドの分割軸の総和とデータ数を受け取る。そして、その情報を元に分割軸の平均値を計算して、各マシンに通知する(図4における破線)。上記のようにして、グリッド分割を分散して行う。また、マスタは、RDBのテーブルに格納されているグリッドの情報に分割を反映する役割も持つ。

この方式において、特にマージするグリッドの数が多いとき、

マスタはボトルネックとなり得る。しかし、異なるグリッドのマージ処理は互いに独立なので、グリッドごとに別々のマシンにマスタの役割を与えることでボトルネックを回避できる。

3.4 レプリケーション

3.3 節で述べたマージ処理では、PostgreSQL へのグリッド分割の問い合わせが一度に多数発生する。この問い合わせがマージ処理におけるボトルネックとなる恐れがあるため、PostgreSQL テーブルを複製してレプリケーションさせる。PostgreSQL テーブルが保持するグリッドの情報はサイズが小さく、更新が発生するタイミングはマージ時に限られるため、レプリケーションに適している。

4. 実験

提案手法の性能を評価するため、データ挿入スループットとクエリ処理スループットを従来手法と比較した。

4.1 実験環境

17 台のマシンから構成されるクラスターで評価実験を行った。17 台すべてに HBase をインストールし、このうち 14 台を Region Server とした。また、PostgreSQL は 8 台のマシンにインストールし、マルチスタンバイ構成のレプリケーションを構築した。

用いた PostgreSQL のバージョンは 9.6.1、HBase のバージョンは 1.2.0 である。各マシンの構成を次に示す。

OS CentOS 6.7

CPU Intel Core i7-3770 (3.4 GHz, 4 コア/8 スレッド)

メモリ 32GB

HDD 2TB

4.2 実験で使ったデータ

室内の気象センサより得られたデータをもとに実験を行った。このデータは、2010 年 1 月 14 日から 2014 年 4 月 11 日まで収集した 2,032,918 件 (約 200 万件) のデータであり、時刻、気温、湿度、照度、風速の五つの属性からなる。データのうち 2011 年から 2013 年までの 3 年分のデータを抽出し、その時刻を 3 年ずつ遅らせながら 7 倍に複製することで生成した 2011 年から 2031 年までの疑似的な 10,446,198 件 (約 1,000 万件) のデータセットを実験で用いた。各属性の定義域を次の表 1 に示す。

表 1 実験で用いたデータの各属性の定義域	
属性名	定義域
時刻	[2011 年 1 月 1 日, 2031 年 12 月 31 日]
気温	[0, 40]
湿度	[0, 100]
照度	[0, 100]
風速	[0, 70]

4.3 実験内容

実験 1：クライアントが一台のときのデータ挿入スループットの計測

一台のクライアントがデータを挿入するときのデータ挿入ス

ループットを計測した。比較対象として、我々の従来手法 [10]、および HBase と PostgreSQL をそれぞれ単体で使った場合を設定した。

提案手法、および我々の従来手法では、風速以外の四つの属性で構成される 4 次元空間をデータ空間とした。また、グリッドサイズ N_{size} は、50, 125, 250, 500, 1000, 2000, 4000, 8000 の 8 通りに変化させ、分割開始の閾値 $N_{\text{threshold}}$ は $N_{\text{threshold}} = N_{\text{size}} \times 10$ とした。さらに、提案手法において Region Server 間に負荷を分散させるためのパラメータ P は $P = 14$ とした。これは、Region Server の数の倍数である。

HBase 単体の手法では、時刻をキーとしてデータを挿入した。なお、提案手法と同様にキーの先頭に $[0, P)$ の乱数に基づくバイト値を付加することで、Region Server 間の負荷分散を行った。

また、PostgreSQL 単体の手法では、風速以外の四つの属性に B-tree の複合インデックスを張った上で挿入した。その際、レプリケーションを無効にして一台のマシンに対して挿入した。

実験 2：実験 1 で挿入したデータに対するクエリ処理スループットの計測

実験 1 でデータを挿入した後、風速の総和を求めるクエリ処理スループットを計測した。

用いた範囲クエリは、風速以外の四つの属性に関する 4 次元の範囲クエリである。時刻の範囲は 360 日であり、データ空間内でランダムに移動させた。また、時刻以外の範囲は次の通り固定した。

気温：[15, 35]、湿度：[20, 40]、照度：[80, 95]

クエリは、16 台のマシンで各マシンあたり 4 スレッドで発行することで、64 並列で問い合わせを行った。

なお、提案手法、我々の従来手法、および HBase 単体の手法では、データの挿入後に HBase テーブルのフラッシュとメジャーコンパクションを行った上でクエリを処理した。

実験 3：クライアントが複数台のときのデータ挿入スループットの計測

実験 1 と同様のデータ挿入をクライアントの台数を複数台にして実施した。ただし、提案手法におけるグリッドサイズ N_{size} は、 $N_{\text{size}} = 1000$ に固定した。また、我々の従来手法は、複数台のクライアントがデータを挿入する状況に対応していないため、評価の対象外とした。

4.4 実験結果

以降では、グリッドサイズが N_{size} のときの提案手法を「提案 (N_{size})」、我々の従来手法を「従来 (N_{size})」と表記する。

実験 1：クライアントが一台のときのデータ挿入スループットの計測

クライアントが一台のときの各手法におけるデータ挿入スループットを次の図 5 に示す。また、比較対象に対する提案手法のデータ挿入スループット向上率 (比較対象を 1 としたときの提案手法のデータ挿入スループット) を表 2 に示す。

図 5、および表 2 の結果をまとめる。まず、提案手法は従来手法、および PostgreSQL を単体で用いた場合に比べ、どのグ

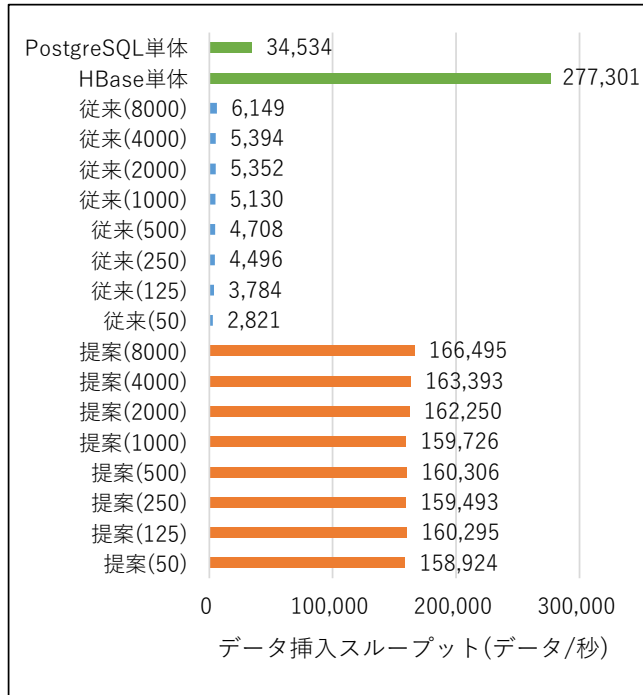


図5 クライアントが一台のときのデータ挿入スループット

表2 比較対象に対する提案手法のデータ挿入スループット向上率
(従来手法に対しては同じグリッドサイズ同士を比較)

グリッドサイズ	50	125	250	500	1000	2000	4000	8000
従来手法	56.3	42.4	35.5	34.1	31.1	30.3	30.3	27.1
HBase 単体	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
PostgreSQL 単体	4.6	4.6	4.6	4.6	4.6	4.7	4.7	4.8

リッドサイズでも高いデータ挿入スループットを実現した。その高速化率は、従来手法に対して 27.1～56.3 倍、PostgreSQL 単体を用いた場合に対して 4.6～4.8 倍である。一方、HBase を単体で用いた場合に比べると、データ挿入スループットは 0.6 倍に低下した。

また、図5から、提案手法では、グリッドサイズが小さくなるにつれてデータ挿入スループットがわずかながら低下していることがわかる。 $N_{\text{size}} = 8000$ のときに対する $N_{\text{size}} = 50$ のときのデータ挿入スループットの低下割合は 4.5% である。これは、グリッドサイズが小さい場合、グリッド分割が頻繁に発生するためであると考えられる。

実験2：実験1で挿入したデータに対するクエリ処理スループットの計測

実験1で挿入したデータに対するクエリ処理スループットを次の図6に示す。また、比較対象に対する提案手法のクエリ処理スループット向上率を表3に示す。

まず、提案手法は PostgreSQL および HBase をそれぞれ単体で使用した場合に比べ、どのグリッドサイズでも高いクエリ処理スループットを実現した。

続いて、従来手法と比較する。提案手法と従来手法のクエリ処理スループットが最も高かったグリッドサイズとそのときのスループットは、次の通りであった。

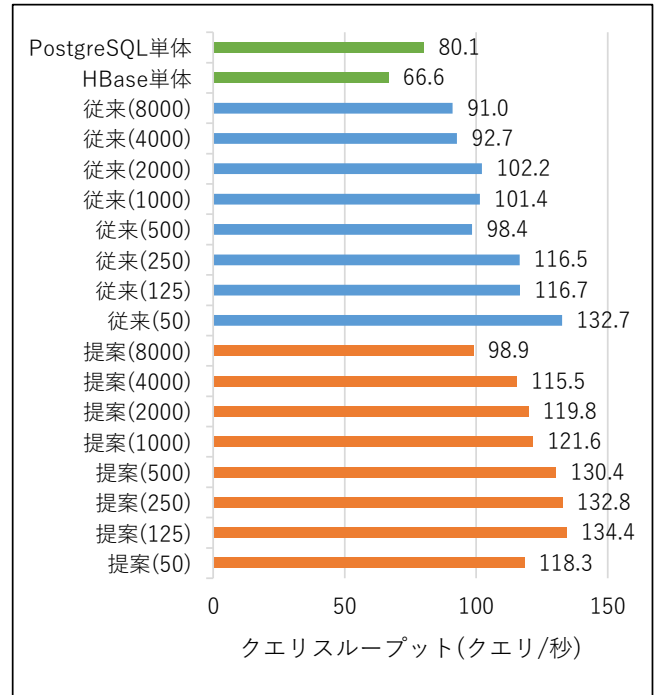


図6 実験1で挿入したデータに対するクエリ処理スループット

表3 比較対象に対する提案手法のクエリ処理スループット向上率
(従来手法に対しては同じグリッドサイズ同士を比較)

グリッドサイズ	50	125	250	500	1000	2000	4000	8000
従来手法	0.9	1.2	1.1	1.3	1.2	1.2	1.2	1.1
HBase 単体	1.8	2.0	2.0	2.0	1.8	1.8	1.7	1.5
PostgreSQL 単体	1.5	1.7	1.7	1.6	1.5	1.5	1.4	1.2

提案手法 $N_{\text{size}} = 125$ のとき、134.4 クエリ/秒

従来手法 $N_{\text{size}} = 50$ のとき、132.7 クエリ/秒

このことから、提案手法は従来手法とほぼ同等のクエリ処理性能を実現できたといえる。実験1の結果と合わせると、提案手法は従来手法に対して、クエリ処理スループットを低下させることなく、データ挿入スループットの大幅な向上を実現できたことが判明した。

実験3：クライアントが複数台のときのデータ挿入スループットの計測

挿入クライアント数を変化させた場合のデータ挿入スループットを図7に示す。また、比較対象に対する提案手法のデータ挿入スループット向上率を表4に示す。

これらの結果を議論する。提案手法は、PostgreSQL 単体を用いた場合に比べて、2.9～5.1 倍の高いデータ挿入スループットを実現した。一方、HBase を単体で用いた場合に対しては、0.6～0.7 倍のスループットに低下した。

提案手法において、このような挿入スループットの低下が発生する原因を考察するため、データ挿入中の瞬間的なスループットの推移を計測した。そのグラフを図8に示す。このグラフは、クライアント数が7台のときの実験結果であり、各クライアントの瞬間的な挿入スループットの和を示している。

図8のグラフから HBase を単体で用いた場合は、40 万デー

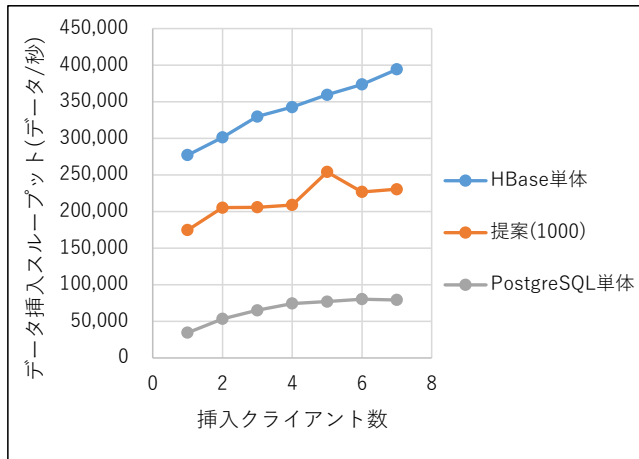


図7 クライアントが複数台のときのデータ挿入スループット

表4 比較対象に対する提案手法のデータ挿入スループット向上率

挿入クライアント数	1	2	3	4	5	6	7
HBase 単体	0.6	0.7	0.6	0.6	0.7	0.6	0.6
PostgreSQL 単体	5.1	3.8	3.2	2.8	3.3	2.8	2.9

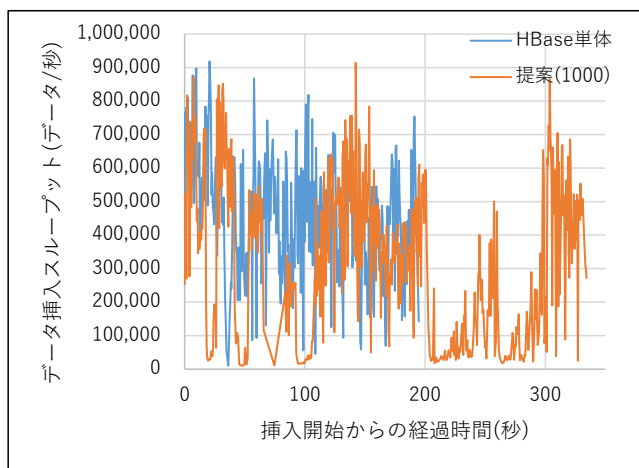


図8 クライアント数が7台のときの瞬間的なデータ挿入スループット

タ/秒前後の挿入スループットが安定して得られているのに対し、提案手法では挿入スループットが急激に低下する箇所が複数存在する。特に、挿入開始後 200～300 秒の間では、挿入スループットが 10 万データ/秒以下の時間帯が長く続いている。一方、挿入開始後 100～200 秒の間などでは、HBase 単体とほぼ同等の挿入スループットが得られている。

このことから、提案手法においてさらにデータ挿入スループットを向上させるには、一時的なスループットの低下をもたらす原因の解明が必要である。

5. まとめ

本研究では、RDB と分散 KVS を相補的に活用して多次元データに対する集約演算を効率化する我々の従来手法 [10] に対し、データ挿入スループットを向上させる手法を提案した。

提案手法では、グリッド分割などのインデックス更新処理をデータ挿入クライアントではなくサーバ上で行うよう変更する

ことで、インデックスを更新する際の排他制御のコストを削減した。また、RDB をレプリケーションさせて負荷を分散させたほか、HBase のキーに乱数による接頭辞を付加して特定の Region Server に負荷が集中する問題を解決した。

評価実験として、室内のセンサから得られたデータを元に生成した約 1,000 万件のデータを一台のクライアントで挿入した結果、我々の従来手法に対して 27.1～56.3 倍のスループット向上を達成した。その後、総和を求める 4 次元の範囲クエリを 64 並列で処理した結果、従来手法と同等のクエリ処理スループットを実現した。このことから、提案手法は、従来手法に対してクエリ処理スループットを低下させることなく、高いデータ挿入スループットを実現した。

また、1～7 台のマシンにより同じデータを挿入した結果、PostgreSQL を単体で用いた場合に対しては 2.9～5.1 倍の高いデータ挿入スループットを実現したものの、HBase を単体で用いた場合に対しては 0.6～0.7 倍のスループットに低下した。データ挿入の瞬間的なスループットを計測した結果、提案手法は、HBase 単体に匹敵するスループットを実現した時間帯があった一方で、スループットが急激に低下する時間帯が存在することが分かった。この一時的なスループットの低下が生じる原因を解明することが今後の課題である。

謝 辞

本研究の一部は、JSPS 科研費 JP26280115, JP15H02701, JP16H02908, JP15K20990 の助成を受けたものである。ここに記して謝意を表す。

文 献

- [1] 増永良文, “リレーショナルデータベース入門 [新訂版]”, サイエンス社, 2003.
- [2] Avinash Lakshman, Prashant Malik, “Cassandra - A Decentralized Structured Storage System”, ACM SIGOPS Operating Systems Review, Volume 44, Issue 2, pp. 35–40, 2010.
- [3] Korry Douglas, Susan Douglas, “PostgreSQL: a comprehensive guide to building, programming, and administering PostgreSQL databases”, SAMS publishing, 2003.
- [4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, “Bigtable: A Distributed Storage System for Structured Data”, OSDI '06, pp. 205–218, 2006.
- [5] Bentley, Jon Louis. “Multidimensional binary search trees used for associative searching”, Communications of the ACM 18.9, pp. 509–517, 1975.
- [6] G.M. Morton, “A computer oriented geodetic data base and a new technique in file sequencing”, Tech.rep., IBM Ottawa, Canada, 1966.
- [7] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, Amr El Abbadi, “MD-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services”, Distributed and Parallel Databases June 2013, Volume 31, Issue 2, pp. 289–319, 2013.
- [8] 小山田昌史, 陳テイ, 成田和世, “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”, Technical Report 19, Nov 2014.
- [9] 小山田昌史, 陳テイ, 成田和世, 荒木拓也, “PA-Proxy: SQL-on-Hadoop におけるデータ集計処理を精度の劣化なく高速化するフレームワーク”, DEIM 2015, E5-6, 2015.
- [10] 渡佐也, 櫻憊志, 宮崎純, 中村匡秀, “RDB と KVS を相互に利用した多次元データに対する集約演算の効率化”, DEIM2016, D5-5, 2016.