DEIM Forum 2016 D5-5

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

渡　佑也† 欅　惇志†† 宮崎　純†† 中村　匡秀†††

† 東京工業大学 工学部 情報工学科　〒 152–8550 東京都目黒区大岡山 2 丁目 12-1
†† 東京工業大学 情報理工学研究科　〒 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，分散ストレージ上に蓄積された多次元のデータに対して，平均値などの集約演算値を求める範囲クエリを効率的に処理する手法を提案する．IoT (Internet of Things) 時代の到来により，センサデータをはじめとする膨大なデータが収集できるようになってきた．こうしたデータの傾向を分析して新たな知見を得ることが重要であり，そのような分析の一例として集約演算を使用するものが存在する．リレーショナルデータベース (RDB) は多次元データを効率的に扱うことができ，集約関数も標準で用意されているが，大規模データに対してスケールアウトしづらいという問題が存在する．また，分散キーバリューストア (分散 KVS) はスケールアウトしやすく設計されているが，多次元データへのアクセスはしばしばデータの全スキャンを伴い非効率的である．このように RDB と KVS は相補的な関係にある．提案手法では，両者を相互に活用することで，多次元データに対する集約演算を効率化する．評価実験として集約演算の範囲クエリを最大 96 並列で与えた結果，提案手法はパラメータを適切に設定することで，PostgreSQL や HBase を単体で用いるときよりも高いクエリスループット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，RDB，KVS

1. はじめに

近年，社会に存在する多種多様な機器がインターネットに接続されるようになってきた．これは IoT (Internet of Things) と呼ばれ，IoT 時代の到来により，センサデータなどの大量のデータが収集できるようになった．こうしたデータの傾向を分析して新たな知見を得ることが重要であり，そのような分析の一例として集約演算を使用するものが存在する．たとえば，センサデータとしてアメダス (注1) のデータを考える．このデータには，観測日時や気温のほかにセンサが置かれている場所の緯度と経度の属性が含まれているとする．ここで「ある長方形で表される地域内の過去一週間の平均気温 (移動平均)」をグラフとして表すことで気温の変動を分析したいとする．これは，日時と緯度，経度に関する三次元の範囲に含まれるデータに対して集約演算を行う範囲クエリを，日時を少しずつ移動させながら行うものであるといえる．このような範囲クエリに対する処理を実現するために，リレーショナルデータベース (RDB) [1] やキーバリューストア (KVS) [2] を用いることが考えられる．

RDB にはインデックスの概念が存在する．そのため，先のアメダスデータのような多次元データを効率的に扱うことが出来る．しかし，センサデータが継続的に入力されるようなシステムでは，短時間に大量のデータが挿入される場面に対処できる必要があるが，RDB はデータの一貫性や整合性を保ちながら書き込みリクエストを処理するため，高い挿入スループットを実現するのは困難である．また，時間の経過とともに蓄積されるデータの量は増大するため，サーバの台数を増やして大規模データや処理を分散させるスケールアウトを実現しにくい RDB の欠点が問題となってくる．

一方で，NoSQL の一種である分散 KVS では，高い挿入スループットやスケールアウトを実現しやすく，効率的に大規模データを扱うことができる．しかし，多くの KVS にはインデックスが存在しないか，存在しても単純な機能しか持たない．従って，多次元データにアクセスする際には膨大な全データをスキャンする必要があるため，極めて非効率的である．なお，本論文では，分散 KVS を単に KVS と呼ぶことにする．

このように RDB と KVS は，相補的な利点を持つ．そこで，本研究では両者を相互に利用し，多次元のデータに対して集約演算値を求める範囲クエリを効率的に処理する手法を提案する．提案手法では，データが属する多次元空間をグリッドに分割していき，グリッドごとの集約演算結果 (部分集約結果) を事前に計算しておく．範囲クエリを処理する際には，部分集約結果を可能な限り再利用して集約演算結果を求める．この際，データ量が膨大となり頻繁に更新される実データや部分集約結果は KVS で保持する．また，実データほど量は多くないが複雑な構造を持つグリッドそのものの情報を RDB で保持する．

本研究では，提案手法を PostgreSQL と HBase を用いて実装し，集約演算のスループットを従来手法，すなわち，RDB と KVS をそれぞれ単体で用いる手法と比較する実験を行うことで，提案手法の性能を明らかにする．

2. 基礎知識

本節では，提案手法が利用するリレーショナルデータベース (RDB) とキーバリューストア (KVS)，および多次元データに対する二分探索木である k-d tree について整理する．

(注1)：http://www.jma.go.jp/jp/amedas/

2.1 リレーショナルデータベース (RDB)

リレーショナルデータベース (Relational DataBase, RDB) [1] には，関係 (リレーション) やそれに対する様々な関係代数演算を SQL (Structured Query Language) を用いて簡単に記述できるという特徴がある．また，インデックスを用いることで複雑な処理を効率的に行うことができるほか，トランザクションが存在するなど一貫性を重視した構造となっている．しかしそれ故に，分散化してもスケールアウトしにくいという問題点がある．このような RDB に基づくシステムは，リレーショナルデータベース管理システム (Relational DataBase Management System，RDBMS) と呼ばれる．

PostgreSQL

PostgreSQL [3] は，オープンソースの RDBMS であり，点や長方形などを表現する幾何データ型が標準で用意されている．

PostgreSQL の幾何データ型のうち `box` 型について説明する．`box` 型は二次元平面上の長方形 (矩形) を表し，対角線の両端の二点で表現される．`box` 型をオペランドにとり真偽値を返す二項演算子として，`&&`演算子と`@<`演算子が存在する．`&&`演算子は，2 つの `box` 型に共通部分がある (接する場合を含む) ときに真を返し，`@<`演算子は，左辺が右辺に包含される (内接する場合を含む) ときに真を返す．これらの演算子は，GiST [5] インデックスとともに利用することが出来る．つまり，多数の `box` 型オブジェクトが含まれるテーブルから，ある `box` 型と共通部分を持ったり包含されたりするものを高速に列挙することが可能である．

2.2 キーバリューストア (KVS)

キーバリューストア (Key-Value Store，KVS) [2] とは，キー (key) と値 (value) が一対一に対応する構造 (この対を行と呼ぶ) を持ったデータベースのことである．この構造は，関係モデルにしたがってテーブルを定義する RDB に比べ非常に単純なものであり，膨大なデータを扱う場合でもスケールアウトしやすいという特徴がある．その一方で，データに対するアクセスは原則としてキーによる検索に限られるため，RDB が実現する柔軟で複雑な処理を KVS で行うのは難しい．

HBase

HBase [4] は KVS の一種であり，特にカラム指向型のデータベースである．HBase の行にはカラムファミリー (column family) が複数存在し，各カラムファミリー内に複数のカラム (値) が存在するという階層構造を持つ．一つのカラムファミリー内では，カラムの数やそのデータ型は柔軟に変更できる．このように一つのキーに複数の値 (カラム) を対応づけることが出来るような KVS をカラム指向型という．

また，HBase のテーブルは行の集合であり，キーでソートされている．そのため，キーの範囲検索を効率的に行うことができる．これを応用すると，キーの前方一致検索も簡単に行えることになる．たとえば，キーが “`ABC`” で始まるデータをスキャンしたいときは，キーの範囲として [“`ABC`”, “`ABD`”)(注2) を指定すればよいことになる．

(注2)：終点は含まれないことに注意する．

HBase では，テーブルを Region と呼ばれる単位に分割する．各 Region 内の行キーは連続している．データ数の増加につれて，自動的に Region が分割される．Region は，Region Server で保持される．データが増加した場合は，新たな Region Server を追加することで簡単にスケールアウトを実現できる．

2.3 k-d tree

k-d tree とは，多次元空間をトップダウン的に分割して構築される二分探索木である．分割はある軸に垂直な平面で行われ，その軸は循環的に選択される (注3)．分割点の選び方はいくつか考えられるが，データの中央値付近で分割すると平衡な k-d tree が得られる．図 1 に 2 次元平面上での空間 (平面) 分割の様子を示す．

以降では，k-d tree の各ノードが表す超直方体をグリッド (grid) と呼ぶことにする．k-d tree は二分木であるから，グリッドをビット列の番号として表現できる．あるグリッドの番号がビット列 x であるとき，それを分割して出来る二つの子グリッドの番号は，$x \oplus 0$ および $x \oplus 1$ とすることができる．ここに $\oplus$ は，ビット列の連結を表す．データ空間全体を示すグリッド (根ノード) は，1 ビットのビット列 0 で表現する．図 1 では，このビット列が表現されている．

このような番号付けを採用することで，提案手法において重要な役割を果たす次の性質が得られる．

性質 1 (グリッド番号の接頭辞性) グリッド G_d がグリッド G の子孫であるとき，グリッド G の番号はグリッド G_d の接頭辞となる．

図 1 でこの性質を確認する．図 1 (b) におけるグリッド 00 は，図 1 (c) においてグリッド 000 と 001 の 2 つの子グリッドに分割される．親グリッドの番号 00 は，ともに子グリッドの番号の接頭辞になっていることが分かる．さらに，グリッド 001 の子であるグリッド 0010，0011 も 00 を接頭辞に持つ．

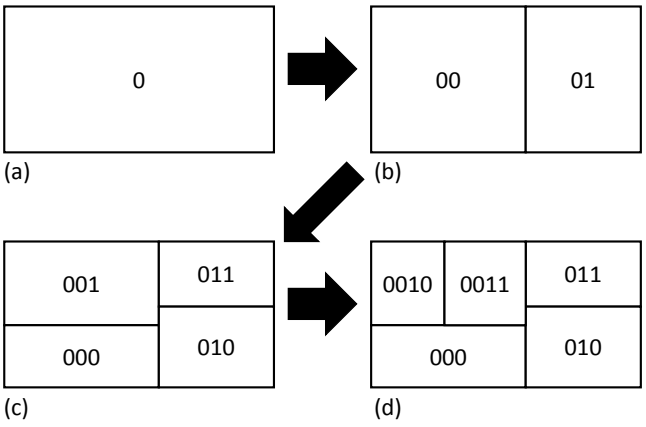

図 1　2 次元空間における k-d tree の分割の様子

(注3)：循環的とは次のような意味である．n 次元空間上の k-d tree を考えると，第 1 軸で分割したあとは第 2 軸で分割する．このように順番に分割していき，第 n 軸のあとは再び第 1 軸で分割する．

3. 関 連 研 究

3.1 部分集約法

部分集約法 [6] とは，データベースを複数のブロックに分割してブロックごとに集約演算結果を事前計算した上で，集約演算のクエリを処理する際に事前計算結果を可能な限り再利用して効率化を図る手法である．

たとえば，年齢 (age) と身長 (tall) からなるリレーション B を考える．まず，B を 3 つのブロック B_1，B_2，B_3 に分割する．このとき，3 つのブロックは互いに共通部分を持たず，それらの和集合が B に一致するように分割する．リレーション B の属性 tall に対する最大値の集約演算 $\max_{\text{tall}}(B)$ は，

$$\begin{aligned}\max_{\text{tall}}(B) &= \max_{\text{tall}}(B_1 \cup B_2 \cup B_3) \\ &= \max\left(\max_{\text{tall}}(B_1), \max_{\text{tall}}(B_2), \max_{\text{tall}}(B_3)\right)\end{aligned}$$

のように，ブロックごとに集約演算を行った結果 (部分集約結果) から求めることが出来る．部分集約法は，この部分集約結果をあらかじめ求めておくことで，実データへのアクセスを省略して集約演算を効率化する．

続いて，集約演算に選択演算が組み合わさる場合を考える．先ほどと同じリレーションに対して，15 歳以下の子供についての身長の最大値を求めたいとする．年齢 (age) が 15 歳以下であるレコードを選択する選択演算を $\sigma_{\text{age}\leqq 15}(\cdot)$ と書くことにすれば，求める最大値は，

$$\begin{aligned}&\max_{\text{tall}}(\sigma_{\text{age}\leqq 15}(B)) \\ =&\max_{\text{tall}}(\sigma_{\text{age}\leqq 15}(B_1) \cup \sigma_{\text{age}\leqq 15}(B_2) \cup \sigma_{\text{age}\leqq 15}(B_3)) \quad (1)\end{aligned}$$

と計算できる．ここで，ブロック B_1，B_2，B_3 について次の知識が与えられているとする．

- ブロック B_1 に含まれるレコードはすべて 15 歳以下である ($\sigma_{\text{age}\leqq 15}(B_1) = B_1$ が成り立つ．選択率が 100%)
- ブロック B_2 には 15 歳以下であるようなレコードは含まれない ($\sigma_{\text{age}\leqq 15}(B_2) = \emptyset$ が成り立つ．選択率が 0%)
- ブロック B_3 には 15 歳以下であるレコードもそうでない (15 歳より上の) レコードも含まれる ($\sigma_{\text{age}\leqq 15}(B_3)$ は，B_3 とも $\emptyset$ とも等しくない)

この知識により，式 (1) は，

$$\begin{aligned}&\max_{\text{tall}}(\sigma_{\text{age}\leqq 15}(B_1) \cup \sigma_{\text{age}\leqq 15}(B_2) \cup \sigma_{\text{age}\leqq 15}(B_3)) \\ =&\max_{\text{tall}}(B_1 \cup \sigma_{\text{age}\leqq 15}(B_3)) \\ =&\max(\max_{\text{tall}}(B_1), \max_{\text{tall}}(\sigma_{\text{age}\leqq 15}(B_3)))\end{aligned}$$

と変形できる．最終式の $\max_{\text{tall}}(B_1)$ は事前に計算されているため定数時間で求めることが出来る．したがって，データの全スキャンは，$\max_{\text{tall}}(\sigma_{\text{age}\leqq 15}(B_3))$ を計算するためにブロック B_3 に対してのみ行えばよく，B_1，B_2 のデータをスキャンする必要はない．

上記のような知識を得るための方法として，小山田ら [6] は軽量インデックスの利用について述べている．軽量インデックスとは，属性ごとの最大値や最小値をあらかじめ計算したものである．たとえば，ブロック B_1 に含まれるデータの年齢の最大値が $\max_{\text{age}}(B_1) = 13$ であったとすると，上記のように $\sigma_{\text{age}\leqq 15}(B_1) = B_1$ であると判断できる．

なお，部分集約法を適用することが出来るのは，最大値のほか，最小値，総和，平均値のようにデータの部分的な集約演算結果から全体の結果が求まるような集約演算に限られる．そのため，属性の濃度 (cardinality) に対しては，部分集約法を適用できない．

3.2 MD-HBase

MD-HBase [7] は，HBase を改良して多次元範囲検索を効率的に行えるようにした KVS である．

MD-HBase では，空間充填曲線と呼ばれる多次元空間内のすべての点を一筆書きでなぞる曲線を用いて，多次元データを一次元データへと変換し，その値をキーとして HBase 上にデータを挿入する．MD-HBase では，空間充填曲線として Z カーブ [8] を用いる．また，k-d tree により多次元空間を複数の領域に分割し，領域ごとのキーの最小値と最大値をインデックスとして保持する．

多次元範囲検索を行うときは，検索領域内のキーの最小値と最大値を求め，HBase に対してキーの範囲検索を行う．このとき，検索範囲外のデータまでスキャンしてしまうのを避けるために，インデックスからキーの最小値と最大値の範囲内にある領域を調べ，検索範囲と全く重ならない領域については，スキャンを省略することで範囲検索を効率化する．

MD-HBase は，インデックスを含めすべてのデータ構造を HBase 上に構築する．これは，RDB と KVS を共に利用する提案手法と異なる．また，MD-HBase において k-d tree の各領域内にあるデータのキーは連続している必要があるが，Z カーブの性質上，この条件を満たすためには領域のちょうど真ん中の点で分割しなければならない．一方，提案手法では，グリッド分割の情報を RDB で保持するため，真ん中の点以外で分割することができる．これにより，データの分布が不均一であっても木を平衡に保ちやすくなるほか，ユーザが与えるクエリに応じて柔軟に分割することも可能となる．

4. 提 案 手 法

4.1 概　　要

図 2 に示した提案手法の概念図を用いて，提案手法の概要を説明する．提案手法において，集約演算を効率的に計算する流れは次の通りである．

(1) データ空間を複数のグリッドに分割する (図 2 上側).
(2) 各グリッドについて集約演算を行った結果 (部分集約結果) を事前に計算し保持しておく．
(3) クエリ範囲 (図 2 上側の破線) に完全に包含されるグリッドについては部分集約結果を再利用してデータの全スキャンを省略しながら，集約演算結果を求める．

(1) におけるグリッド分割は k-d tree に従う．図 2 は，図 1 (d) から，さらにグリッド 0011 を分割した後の状態である．

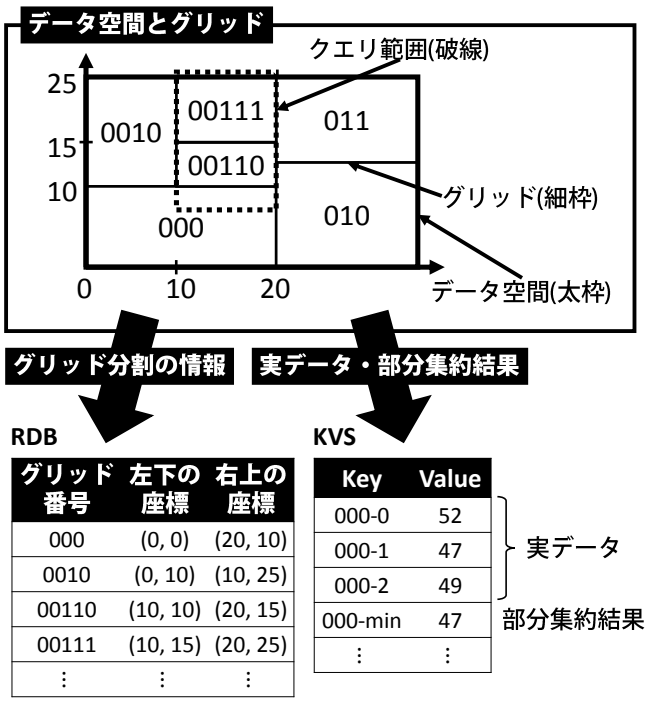

図 2　提案手法の概念図

(2) について，新しくデータが追加されたときは，部分集約結果を足し合わせるように更新する．

(3) について，図 2 に示したクエリ範囲を例にとると，これは三つのグリッド (000，00110，00111) と交わっている．このうち，グリッド 000 はクエリ範囲と一部が交わるので，データの全スキャンを行う必要があるものの，グリッド 00110 と 00111 は，クエリ範囲に完全に包含されるため，部分集約結果を参照するだけでよく，データの全スキャンは不要である．

ここで，提案手法における工夫点は，図 2 の下側に示したように RDB と KVS の双方の長所を最大限生かすようにデータを保持する点である．提案手法で必要となるデータには，グリッド分割の情報 (データ空間上に存在するグリッドの位置，大きさおよび番号) と，実データ，部分集約結果がある．

このうち，グリッド分割の情報は，グリッドが極端に小さくない限り実データに比べてデータサイズは十分小さいと考えられる．その一方で，クエリ処理を行うためにはクエリ範囲と交わるグリッドを列挙するという複雑な処理を行う必要がある．このような特徴から，提案手法では，インデックスを用いることで多次元データを効率的に扱える RDB (PostgreSQL) でグリッド分割の情報を保持することとした (図 2 の左下)．グリッド分割の情報は，グリッドが分割されない限り更新されず，そのグリッド分割もデータ挿入と比較すると十分に低頻度であるため，データベースの複製を利用することも容易である．

一方，本研究が扱うデータとしてセンサデータを想定すると，実データの量は膨大なものになると考えられる．また，絶え間なくデータが生成されるため，実データや部分集約結果は頻繁に更新される (注4)．そこで，高い挿入スループットとスケールアウトを実現しやすい KVS (HBase) でこれらの情報を保持することとした (図 2 の右下)．

このように RDB と KVS の双方の利点を活用することで，それらを単体で用いる際に発生する問題を解決する．

4.2　用語の定義

提案手法では，データを多次元空間上の点として考える．データの定義域をデータ空間と呼ぶことにし，記号 $D\,(\subset \mathbb{R}^n)$ で表す．ここに n はデータの次元数であり，データ空間は超直方体である．より正確にいえば，超直方体の次元 i の始点と終点を s_i，e_i $(s_i, e_i \in \mathbb{R},\, s_i < e_i)$ とすると，D は次のように直積で表される．

$$D = [s_1, e_1] \times [s_2, e_2] \times \cdots \times [s_n, e_n]$$

グリッドとは，データ空間を分割してできる超直方体である．グリッドの総数を m とすると，グリッド $G_1, \cdots, G_m \subseteqq D$ は，式 (2) を満たす．これは，グリッド分割が数学的な集合の分割 (どの二つのグリッドも互いに共通部分を持たず，すべてのグリッドの和集合はデータ空間全体となる分割) であることを意味する．

$$\forall i \neq j\,(G_i \cap G_j = \emptyset) \quad \text{かつ} \quad \bigcup_{i=1,\ldots,m} G_i = D \qquad (2)$$

グリッド G とクエリ範囲 $Q\left(\subseteqq D\right)$ について，G が Q と交わるとは $G \cap Q \neq \emptyset$ が成り立つことであり，G が Q に完全に包含されるとは $G \cap Q = G$ が成り立つことである．

4.3　データ構造

4.3.1　グリッド分割の情報

本節の冒頭で述べたように，グリッド分割の情報は RDB で保持し，そこに n 次元のインデックスを作成することとした．今回は，PostgreSQL を用いて実装したが，PostgreSQL には 3 次元以上のデータを扱えるインデックスが存在しないため，2.1 節で述べた `box` 型の集合で代用した．具体的には，n 次元空間上のグリッドを $\lceil n/2 \rceil$ 個の `box` 型で表現し，それらに複合インデックスを張った．

例として，$n = 5$ のときのグリッド分割の情報を保持する PostgreSQL テーブルの定義を次に示す．列 `C0`，`C1` がそれぞれ二次元分を表現し，`C2` が残りの一次元分を表現する．列 `GridID` はグリッド番号を表し，列 `NextSplittingDimension` は，k-d tree において次に分割すべき軸を表す．

```
CREATE TABLE SensorTable
  (C0 box, C1 box, C2 box,
   GridID text, NextSplittingDimension integer)
```

このテーブルに対して，グリッドの列挙を高速に行うためにインデックスを作成する．具体的には，$\lceil n/2 \rceil$ 個の `box` 型に対する複合インデックスであり，インデックスの種類は GiST である．先の例におけるインデックスの定義を次に示す．

```
CREATE INDEX SensorTableGridBoxIndex
  ON SensorTable USING GiST (C0, C1, C2)
```

(注4)：4.4 節でも述べるが，部分集約結果の計算はクエリ処理が与えられるまで遅延させることも考えられる．

4.3.2 実データ・部分集約結果

実データと部分集約結果は，分散 KVS の一種である HBase に格納する．HBase テーブルは図 3 に示したような構造をとる．この例は，図 2 の状況に対応している．

図 3 に示した行キーは，1 文字が 1 バイトを表す．行キーの先頭に記されている “?” は，0 から 255 までの 1 バイトの整数値であり，“?” の後に続くグリッド番号の先頭の次から 8 ビット分を数値として解釈した値である．ただし，ビット数が足りない場合は，下位ビットが 0 であるとみなす．なお，先頭ビットを無視するのは，それが常に 0 であるためである．

たとえば，図 3 左側の最終行の行キーは，“?00111-2” である．このときの “?” の値は，グリッド番号の先頭を除いたビット列 “0111” を 8 ビットになるよう 0 を末尾に付加して得られる “01110000” を数値として解釈した値，すなわち 112 である．

このように行キーの先頭バイトを 0 から 255 まで分散させることで，HBase テーブルを Region に分割する際にデータを分散させやすくなる．さらに，グリッド番号の大小関係と行キーの大小関係が一致するため，グリッド番号に対する接頭辞検索を簡単に行えることになる．

また，部分集約結果は親グリッドについても保持されていることに注意する．たとえば，図 3 右側を見るとグリッド 00110 と 00111 の体重の最大値が示されているが，それらの親であるグリッド 0011 の最大値も記録されている．その親についても同様である．

4.4 データ挿入

新たなデータを挿入するアルゴリズムは，次の通りである．

アルゴリズム 1 データ d $(\in D)$ の挿入

(1) データ点 d が属するグリッド番号 *gridID* を PostgreSQL テーブルに問い合わせる．
(2) HBase テーブルに「? *gridID*-*elementID*」をキーとして実データを挿入する．
(3) データ点 d が属するグリッドおよびすべての親グリッドについて，HBase 上の部分集約結果を更新する．
(4) 挿入を行った結果，グリッド内のデータ数が分割閾値 $N_{\text{threshold}}$ を超えた場合はグリッドを分割する．

(1) は，2.1 節で述べた `box` 型に対する演算子を用いる．

(3) で親グリッドに対しても更新を行うのは，4.6.1 節で述べる部分集約結果参照の効率化のためである．なお，(3) の作業は，実際にクエリが与えられて部分集約結果を参照する必要が出てくるときまで遅延させることも考えられる．

(4) の詳細は，4.5.1 節で述べる．

行キー	実データファミリ		
	年齢	身長	体重
?00110-0	32	170	65
?00110-1	40	173	71
?00110-2	25	159	47
?00110-3	26	163	50
?00111-0	30	175	70
?00111-1	33	177	81
?00111-2	34	180	83

行キー	部分集約結果ファミリ
	部分集約結果
?0-体重の最大値	97
?00-体重の最大値	90
?001-体重の最大値	83
?0011-体重の最大値	83
?00110-体重の最大値	71
?00111-体重の最大値	83

図 3　HBase テーブルの様子

4.5 グリッド分割

4.5.1 データの分布に応じたグリッド分割

4.4 節で述べたように，データの挿入によってグリッド内のデータ数が $N_{\text{threshold}}$ を超えた場合，k-d tree にしたがいグリッドを分割する．そのアルゴリズムを次に示す．

アルゴリズム 2 グリッド G の分割

(1) 分割すべき軸の番号 i $(1 \leq i \leq n)$ を PostgreSQL に問い合わせる．
(2) グリッド内の軸 i についての平均値 avg_i を HBase から取得する (これは部分集約結果として保持されている)．
(3) グリッド G 内のデータのうち，第 i 軸の値が avg_i 以下であるものから子グリッド G_{lower} を，それ以外のデータから子グリッド G_{upper} を作成する．このとき，HBase 内の実データの行キーを新たなキーに書き換える．また，G_{lower}，G_{upper} の部分集約結果を計算し，HBase に挿入する．
(4) PostgreSQL テーブルから G の情報を削除するとともに，G_{lower}，G_{upper} の情報を追加する．このとき，次に分割すべき軸の番号は $i+1$ (これが次元数 n を超えた場合は第 1 軸に戻る) とする．
(5) G_{lower}，G_{upper} に含まれるデータ数がグリッドサイズ N_{size} を超えた場合は，このアルゴリズムを再帰的に適用する．

(3) で中央値ではなく平均値で分割しているのは，中央値の計算にコストがかかるためである．

また，(5) で再帰的にアルゴリズムを適用するかを判断するグリッドサイズ N_{size} は，アルゴリズム 1 (4) における分割閾値 $N_{\text{threshold}}$ とは**異なる**．両者は，$N_{\text{size}} \leqq N_{\text{threshold}}$ なる関係を満たす．つまり，データの挿入によってグリッドサイズ N_{size} を少し超えることは許容し，ある程度のデータ数になってから分割を開始する．これによりグリッド分割の頻度を抑制できる．

4.5.2 時間軸の特別な分割

提案手法では，時間軸を特別扱いして最初のデータが挿入される前にあらかじめ分割を行っておく．具体的には，グリッドを時間軸について二等分するという操作を，グリッドの幅がパラメータ L_{time} 以下になるまで再帰的に繰り返す．このようにして出来たグリッドを初期状態とする．この分割は時間軸についてのみ繰り返し行われるため，純粋な k-d tree ではなくなるものの，提案手法における仮定は保たれる．

センサデータのような時系列データは，古いデータから順番に挿入される．こうした状況のもとで 4.5.1 節で述べたグリッド分割を行うと，木が時間軸について偏ってしまう．時間軸についての特別な分割を行うことでこの問題を解決できるほか，初期状態にグリッドが一つ (データ空間全体を表すグリッド) しか存在しない場合に比べ，データ挿入のスループットが高まると考えられる．センサデータは時間軸についておおむね一様に分布すると期待できるのでこうした分割は理にかなっている．

4.6 クエリ処理

クエリ範囲 $Q\left(\subseteqq D\right)$ が与えられたとき，Q に含まれるデータに対して集約演算を行うアルゴリズムは次の通りである．

アルゴリズム 3 クエリ範囲 Q 内のデータに対する集約演算

(1) Q と共通部分を持つグリッドを PostgreSQL テーブルから列挙する．このとき，クエリ範囲に完全に包含されるかどうかも問い合わせる．
(2) (1) で列挙されたグリッドのうち，クエリに完全に包含されるグリッドの部分集約結果を足し合わせる．
(3) (1) で列挙されたグリッドのうち，クエリと一部が交わるグリッドに含まれるデータをすべてスキャンし，クエリ範囲に含まれるものについて集約演算を行う．
(4) (2) および (3) で得られた集約演算結果を足し合わせて，最終的な結果を求める．

(2) は，HBase テーブルに記録されている部分集約結果を取得 (get) すればよい．(3) では，行キーに対する接頭辞検索により特定のグリッド内のデータのみをスキャン可能である．実際の実装では，クエリ範囲に含まれるデータだけを抽出するフィルタを独自に作成し，HBase サーバ上で実行できるようにしたほか，集約演算も HBase サーバ上で行うことでネットワーク負荷を軽減する工夫も行った．

4.6.1 部分集約結果参照の効率化

グリッドは木構造をなすため，親子関係が存在する．アルゴリズム 1 では，子に当たるグリッドだけでなく親のグリッドについても部分集約結果を保持・更新していた．これにより，クエリ処理のアルゴリズム 3 (2) を次のように効率化できる．

グリッド番号が $x \oplus 0$ および $x \oplus 1$ (x は 1 文字以上のビット列) であるグリッドがクエリ範囲に完全に包含されていたとする．この二つのグリッドの部分集約結果を参照する代わりに，それらの親であるグリッド x の部分集約結果を用いても結果は変わらない．このようにすることでデータベースへのアクセス量を削減できる．この手続きは，再帰的に適用できる．

図 2 の例では，グリッド 00110 ($= 0011 \oplus 0$) と 00111 ($= 0011 \oplus 1$) の代わりにグリッド 0011 (図 1(d) を参照のこと) の部分集約結果を参照すればよい．

この効率化は，クエリ範囲が大きく，したがってそれと交わるグリッドの数も多くなるときに有効であると考えられる．

4.6.2 データの全スキャン省略

アルゴリズム 3 (3) において，集約演算が最大値または最小値の場合，すでに得られた集約結果から全スキャンを省略できる場合がある．最小値を例に取ると，アルゴリズム 3 (2) および (3) の途中で得られた集約結果 (最小値) がグリッドの最小値より小さければ，集約結果の更新は発生し得ないので，全スキャンを省略できる．最大値についても同様である．

5. 評価実験

提案手法のパフォーマンスを評価するため，クエリ処理のスループットについて RDB と KVS をそれぞれ単体で使用したときと比較する評価実験を行った．

5.1 実験環境

10 台の計算機から構成されるクラスタ上で PostgreSQL，HBase を動作させて実験を行った．このうち，Region Server は 6 台あり，PostgreSQL が動作する計算機は HBase のクラスタには含まれていない．

用いた PostgreSQL のバージョンは 8.4，HBase のバージョンは 1.0.0 である．また，各計算機の構成は次の通りである．

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

5.2 実験で使用したデータ

5.2.1 室内気象データ

実験データの一つとして，室内の気象センサより得られた気温や湿度などからなるデータを用いた．以後，このデータを「室内気象データ」と呼ぶことにする．室内気象データは 23 個の属性を持つが，本実験においてはそのうち表 1 の 5 つの属性を用いた．室内気象データは，時刻の定義域内で 1 分ごとに記録されており，データ数は 2,032,918 件 (約 200 万件) である．

プログラム上では，5 つの属性をすべて 64 ビットの整数に変換して扱った．時刻は，定義域の始点である 2010 年 1 月 14 日 0 時ちょうどからの経過秒数に変換した．それ以外の 4 つの属性はいずれも 1000 倍することで整数に変換した．

表 1 室内気象データのうち実験に用いた属性の概要

属性名	種類	定義域
時刻	日時	[2010 年 1 月 14 日, 2014 年 4 月 11 日]
気温	小数	[0, 40]
湿度	小数	[0, 70]
照度	小数	[0, 100]
風速	小数	[0, 70]

5.2.2 拡張室内気象データ

5.2.1 節の室内気象データよりもさらに大きな規模のデータで実験を行うため，次のように室内気象データを拡張した．

まず，室内気象データのうち 2011 年から 2013 年までの 3 年分のデータ (1,492,314 件) を取り出した．このデータについて，時刻を 3 年ずつ遅らせるようにして元のデータ数の 7 倍に複製した．このようにして生成された 2011 年から 2031 年までの 21 年分の疑似的なデータ 10,446,198 件 (約 1,000 万件) を「拡張室内気象データ」と呼ぶことにする．

5.3 実験内容

拡張室内気象データ (約 1,000 万件) に対して風速の最小値と総和を求める範囲クエリを行った．このクエリ問い合わせは，Region Server でない 3 台の計算機をクライアントとし，各クライアントのスレッド数を 1 から 32 まで変化させて最大 96 並列で行ったほか，1 台の計算機でスレッド数を 1 から 8 まで変化させた問い合わせも行った．各スレッドは 50 個のクエリ問い合わせを行った．用いた範囲クエリは，風速以外の 4 つの属性を範囲とする 4 次元範囲クエリであり，クエリ範囲は次の通りである．まず，時刻の範囲は幅が 360 日であり，その開始位置は乱数を用いて定義域内で一様に移動させた．また，時刻以外の属性の範囲は次の通り固定した．

気温：[15, 35]，湿度：[20, 40]，照度：[80, 95]

提案手法でのデータ空間は，範囲クエリに用いる 4 つの属性を軸とする 4 次元空間とした．このうち，時刻の定義域は 2011 年 1 月 1 日から 2032 年 1 月 1 日であり，それ以外の 3 属性の定義域は，表 1 に準ずる．グリッド分割に関するパラメータは，グリッドサイズを $N_{\text{size}} = 50,\ 125,\ 250,\ 500,\ 1000,\ 2000,\ 4000,\ 8000$ の 8 通りに変化させ，分割閾値は $N_{\text{threshold}} = N_{\text{size}} \times 10$ とした．初期状態における分割を決定する L_{time} は，360 日とした．

データの挿入に先立って HBase テーブルを 12 個の Region に分割し，各 Region Server に 2 つの Region を割り当てた．

提案手法では，データの挿入後に HBase テーブルに対するフラッシュとメジャーコンパクションを行った．これは，提案手法では HBase テーブルに対して大量の put と delete を行うため，コンパクションを行っていないと提案手法に関係のない箇所でパフォーマンスの低下が発生する可能性を避けるためである．なお，同じ作業は比較対象の HBase でも行った．

また，比較対象として PostgreSQL と HBase を単体で用いる手法についても実験を行った．PostgreSQL では，範囲クエリの対象となる 4 つの属性に B-Tree の複合インデックスを張った上でデータを挿入し，クエリ処理を行った．HBase では，データの属性のうち時刻のバイナリ表現をキーとしてデータを挿入した．クエリ処理では，時刻に関するキーの範囲検索を行って範囲内にあるすべてのデータをスキャンし，クエリ範囲に含まれているものについて集約演算を行った．

5.4 実験結果

クエリにより選択されたデータ (クエリ範囲内にあったデータ) は平均で 104,271 件であり，選択率は 1.0%であった．

各手法におけるデータ挿入のスループットを次の表 2 に示す．「提案 (N_{size})」は，グリッドサイズが N_{size} のときの提案手法を示している．以降の図表でも同様である．表 2 におけるスループットとは，データ数を挿入にかかった時間 (秒) で割った値である．なお，提案手法と HBase(単体) におけるテーブルのフラッシュやメジャーコンパクションの時間は含まれていない．

提案手法のスループットは PostgreSQL，HBase のいずれよりも低かった．また，提案手法でグリッドサイズが大きいほどスループットが高いのは，グリッド分割の発生回数が少ないためであると考えられる．

表 2　各手法におけるデータ挿入スループット

手法	提案 (50)	提案 (125)	提案 (250)	提案 (500)	提案 (1000)	提案 (2000)
スループット	4,129	5,798	6,837	7,733	8,456	8,959

手法	提案 (4000)	提案 (8000)	PostgreSQL (単体)	HBase (単体)
スループット	9,217	9,318	45,887	63,491

各手法のクエリスループットを，最小値のクエリについて図 4 に，総和のクエリについて図 5 に示す．さらに，提案手法における各種統計情報を表 3 に示す．

これらの結果をまとめる．図 4，5 を見ると，最小値，総和の両クエリにおいて，提案手法はすべてのグリッドサイズで HBase を単体で用いた手法よりもクエリスループットは高かった．また，PostgreSQL と比較すると，$N_{\text{size}} = 8000$ のときが

表 3　提案手法における各種統計

手法	提案 (50)	提案 (125)	提案 (250)	提案 (500)	提案 (1000)	提案 (2000)	提案 (4000)	提案 (8000)
①	1,736	470	165	58	20	7	1	0
②	477	180	77	32	12	4	1	0
③	46,478	34,877	25,386	18,420	12,519	8,533	2,691	787
④	40.8%	31.0%	21.0%	14.2%	8.8%	5.5%	1.6%	0.4%
⑤	3,294	1,641	905	515	290	165	89	52
⑥	1,845	890	446	239	127	74	39	20
⑦	91,996	122,834	142,624	169,459	192,659	221,188	243,363	280,469
⑧	100.0%	100.0%	99.0%	96.6%	91.2%	83.8%	64.5%	26.4%
⑨	384,279	143,758	67,941	32,699	16,192	8,000	3,916	1,976

①：部分集約結果を再利用できたグリッド数 (4. 6. 1 節の効率化前) の平均，
②：部分集約結果を再利用できたグリッド数 (効率化後) の平均，
③：①(または②) のグリッドに含まれていた総データ数の平均，
④：再利用率 (クエリ範囲にあったデータ数に対する③の割合)，
⑤：部分集約結果を再利用できなかったグリッド数 (効率化前) の平均，
⑥：部分集約結果を再利用できなかったグリッド数 (効率化後) の平均，
⑦：⑤(または⑥) のグリッドに含まれていた総データ数の平均 (= 総和の集約演算においてスキャンを行ったデータ数)，
⑧：スキップ率 (最小値の集約演算において，⑥のグリッドのうち全データスキャンを省略できたものの割合)，
⑨：データ空間内に存在する総グリッド数．

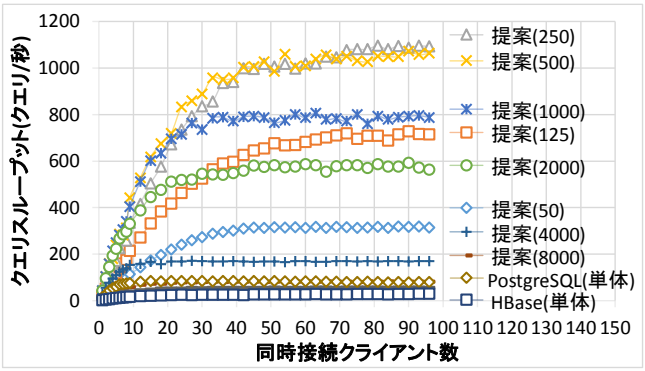

図 4　最小値のクエリスループット

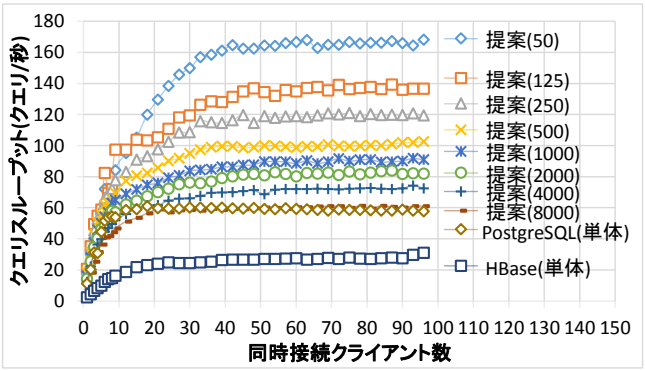

図 5　総和のクエリスループット

ほぼ同じスループットであり，それ以外では高いスループットを実現できた．同時接続クライアント数を 96 とした各クエリにおいて，提案手法が最も高いスループットを実現できたグリッドサイズと，そのときの従来手法に対するスループットの比率は次の通りである．

最小値 $N_{\text{size}} = 250$ のとき，PostgreSQL に対して 13.8 倍，HBase に対して 36.3 倍．

総和 $N_{\text{size}} = 50$ のとき，PostgreSQL に対して 2.9 倍，HBase に対して 5.4 倍．

上記結果について考察する．まず，最小値の集約演算について表 3 の⑧を見ると，グリッドサイズが小さくなるにつれてスキップ率が高くなっていることが分かる．これが高いスループットを実現できた主な理由である．このようにスキップ率が十分高い場合，クエリと一部が交わるグリッドに含まれるデータのうちスキャンするものの数は少ないので，クエリ処理時間は部分集約結果の参照時間，すなわちクエリと交わるグリッド数に支配される．表 3 の②，⑥から，その値はグリッドサイズが小さいほど多くなっている．これが原因で，グリッドサイズが小さい $N_{\text{size}} = 50,125$ では，スキップ率が 100.0%でありながらスループットがそれほど高くならなかったと考えられる．

一方，総和のクエリ処理時間はスキャンを行ったデータ数 (表 3 の⑦) に支配されるはずである．その値はグリッドサイズが小さいほど減少しており，それにつれてスループットが高くなっていることが図 5 から分かる．ここで，同時接続クライアント数 p を固定したとき，クエリ処理の応答時間が単純にスキャンを行ったデータ数 $D_{N_{\text{size}}}$ に比例すると仮定すれば，スループット T_p は $D_{N_{\text{size}}}$ に反比例することになる．したがって，$T_p \cdot D_{N_{\text{size}}} =$ (一定) が成り立つはずであるが，$p = 96$ のとき，この値はグリッドサイズが小さくなるにつれて緩やかに減少する．グリッドサイズが小さいほどクエリと一部が交わるグリッドの数は多くなるが，これは，全データスキャンを行う際に HBase テーブルに対して細かな範囲検索が多数行われることを意味する．したがって，ディスクのランダムアクセスなどのオーバーヘッドが生じて応答時間が長くなり，$T_p \cdot D_{N_{\text{size}}}$ の値が減少したと考えられる．このことから，グリッドサイズを小さくしていくとクエリスループットが減少に転じる点，すなわち最大となる点が存在すると予想される．

上記の結果から，提案手法はグリッドサイズを適切に設定することで，従来手法よりも高いクエリスループットを達成できることが判明した．

6. おわりに

本研究では，RDB と KVS を相互に利用して，多次元データに対する集約演算を効率化する手法を提案した．多次元データを複数のグリッドに分割した上で，あらかじめ計算しておいたグリッドごとの部分集約結果を再利用する．その際，膨大な量に上る実データや頻繁に更新される部分集約結果を KVS (HBase) で保持し，実データほど情報量は多くないが複雑な構造を持つグリッド分割の情報を RDB (PostgreSQL) で保持することで，両者の利点を活用して提案手法を実現した．

評価実験として，室内気象センサのデータをもとに生成した約 1,000 万件のデータに対する 4 次元の範囲クエリを同時に 96 並列で処理する実験を行った結果，提案手法はグリッドサイズを適切に設定することで，HBase を単体で用いた場合に比べ 5.4〜36.3 倍，PostgreSQL に対しても 2.9〜13.8 倍の高いクエリスループットを実現した．しかし，グリッドサイズを小さくすると実データのスキャン量は減少するものの，グリッドの数が増加するためにクエリスループットは必ずしも高くならないという問題が明らかとなった．また，データ挿入のスループットも従来手法に及ばなかった．

今後の課題としては次のようなものが挙げられる．まず，評価実験で用いた範囲クエリは，時間軸についてランダムに移動させて生成したものであったが，ユーザが与えるクエリには何らかの傾向が存在すると考えられる．その傾向に応じたグリッド分割を行うことでクエリ処理時間を短縮させることが課題の一つである．また，実際のシステムにおいて，センサデータは複数の箇所から同時に挿入される．そのような状況の下でも適切にデータ挿入がなされるよう排他制御を行うことが課題である．また，データ挿入のスループットを改善することも課題である．

謝　辞

本研究の一部は，科研費基盤研究 (B)(課題番号：26280115)，基盤研究 (B)(課題番号：15H02701) の支援による．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] Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer, “Generalized Search Trees for Database Systems”, Proceedings of the 21st VLDB Conference, pp. 562–573, 1995.

[6] 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.

[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] G.M. Morton, “A computer oriented geodetic data base and a new technique in file sequencing”, Tech.rep., IBM Ottawa, Canada, 1966.