多次元リニアハッシュファイルデータベース上での Fluent Join の並列化による高速化の実験

佃　陽平†　遠山　元道†

† 慶應義塾大学理工学部情報工学科　〒 223–8522 神奈川県横浜市港南区日吉 3–14–1
E-mail: †tsukuda@db.ics.keio.ac.jp, ††toyama@keio.ac.jp

あらまし　リレーショナルデータベースは扱うデータサイズが大きい場合、Join の実行コストが高くなることがある。この解決のため、多次元リニアハッシュファイルに基づいたリレーショナルデータベースと Fluent Join アルゴリズムを用いた研究が存在する。多次元リニアハッシュファイルとは多次元ハッシュの部分一致処理を行える点とリニアハッシュのリサイズ時にファイルが線型的に増加するため多量のデータを扱える点を組み合わせたファイル編成法であり、Fluent Join は各ハッシュ値毎の結合演算が互いに独立であり、ハッシュ値が一致する部分を繰り返し利用することで読み込むデータ量を削減できるアルゴリズムである。本研究では Fluent Join の演算が独立である特性を利用し、並列実装による高速化を行う。

キーワード　結合演算，リレーショナルデータベース, データベース技術

1　は じ め に

リレーショナルデータベースは扱うデータサイズが大きい場合、結合演算の処理時間は大きくなってしまう。近年、メモリサイズの増加によりインメモリデータベースの開発が進み、そのメモリ上に収まりきる範囲内での結合は高速に行えるようになった。しかし、衛星の観測データのようなビッグデータはメモリ上には収まりきらず、ビッグデータどうしを結合しようとすれば時間がかかってしまう。そもそもビッグデータを正規化すること自体が難しく、一般的なリレーショナルデータベースで扱うことには適していない。

ビッグデータどうしの結合を可能にするため、多次元リニアハッシュファイルに基づいたリレーショナルデータベース上で行われる Fluent Join アルゴリズムを並列に実行することを考えた。[1] 多次元リニアハッシュファイルとは Aho と Ulman によって提案された多次元ハッシュ [3] と Litwin によって提案されたリニアハッシュ [4] を組み合わせたファイル編成法である。

この多次元リニアハッシュファイル上で行われる Fluent Join は各ハッシュ値毎の結合演算が互いに独立であり、ハッシュ値が一致する部分を繰り返し利用することで全てのレコードに対して一度のみのアクセスで結合を可能にするアルゴリズムである。本研究では Fluent Join の演算が独立である特性を利用し、並列実装による高速化を見込めるのか、ベンチマークとしての実験を行った。実験を行う上で、メモリ上のデータの結合の内部動作に注目し、それぞれの並列処理を試みた。

本論文では第 2 章で従来の結合方式について、第 3 章で関連技術、関連研究について、第 4 章では Fluent Join について、第 5 章では実験、評価について、第 6 章ではまとめを記述する。

2　従来の結合方式とその問題点

現在リレーショナルデータベースで用いられている結合方式として、Nested Loop Join、Hash Join、Sort Merge Join の 3 種類が一般的である。これらの結合方式では結合演算を行うタイミングで、例えば Hash Join ならばハッシュテーブルを作成し、Sort Merge Join ならばデータのソートを行っている。これらの結合方式の実行コストが大きい理由として、ソートやハッシュテーブルの作成のような実際に結合を行う前の下準備があるということが挙げられる。さらにこの下準備はデータサイズが大きくなるほどよりコストが大きくなる。また従来の Hash Join はメモリ上にハッシュテーブルを構築するため、データサイズが大きく、メモリ上に収まりきらない場合はデータの読み書きをやり直さなくてはいけなくなる問題点もある。

この解決のため、多次元リニアハッシュファイルデータベースという、以上に述べたような下準備を行わずに済むよう最初から結合を行いやすい状態でデータを格納するデータベースと、その上で行うことができる Fluent Join が提案されている。Fluent Join はメモリの大きさなどに関わらず結合するそれぞれのデータの読み書きを一度しか行わない利点と、演算が独立である利点を持つ。

多次元リニアハッシュファイルデータベースで行われることは PostgreSQL に搭載されているパーティション機能 [6] と似ている。パーティションとはデータベース上のサイズが大きいテーブルを分割することで、問い合わせに対する検索対象削減、オーバーヘッドの回避などの利点がある。

3　関連技術, 関連研究

3.1　リニアハッシュ

リニアハッシュは Litwin によって提案された [4] ファイル編成法で、データの追加に応じてハッシュサイズが線型的に増加していく点、リサイズを行う時、ポインタが指し示すハッシュ値だけを分割し、ハッシュ値を振り直すことで、一度に多くのハッシュ値の更新をしない点が特徴である。例えば図 1 のようにハッシュサイズが 4 であるハッシュ空間を考える。それぞれのデータは 4 で割ったあまりでハッシュ値を割り振られている。このハッシュ空間では 1 つのハッシュ値に対してデータが 5 個以上格納された時分割するとする。

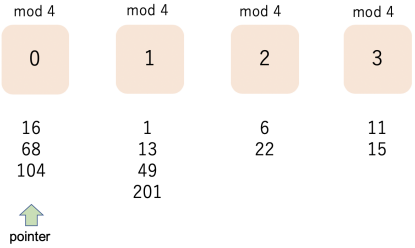

図 1　リニアハッシュの例

この状況で 5 が追加されるとする。ハッシュ値が 1 となり、ハッシュ値 1 がデータ数 5 個になったため、リサイズが行われる。最初、ポインタはハッシュ値 0 を示すのでハッシュ値 0 の空間を mod4 から mod8 に変更する。

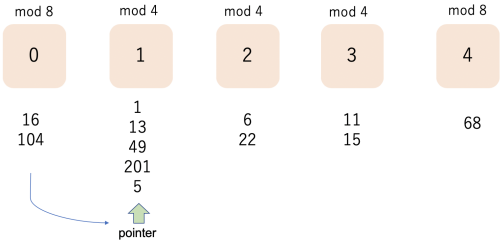

図 2　ハッシュ値 0 を分割した後の状態

この時、ハッシュ値 1,2,3 は mod4 のままで変更しないことで、一度に多くのデータのハッシングを行わないで済む。一度分割を行ったことで、ポインタはハッシュ値 1 に移動する。ハッシュ値 1 にデータが 1 個追加される、0,2,3 のいずれかに 3 個追加される、4 に 4 個追加される場合に、ハッシュ値 1 の空間がハッシュ値が 1 と 5 の空間に分割される。このようにハッシュ空間が 1 ずつ線型的に増加していく。また、ハッシュ値が変更されるデータも一部だけであるため、データサイズが大きな場合でも対応できる。

3.2　多次元ハッシュ

多次元ハッシュは Aho と Ulman によって提案された [3]、レコード 0 個以上のフィールドの値を指定する部分一致処理に適したファイル編成方法である。複数のハッシュ値で構成された固定のハッシュ長の中で、クエリで指定される確率が大きいフィールドほどより多くのハッシュ値を割り当てることで、レコードへの平均アクセス回数を減らすハッシュ法である。なおここでは、各フィールドがクエリで指定される確率はそれぞれ独立であると考える。例えば 3 のようなフィールドで構成されるテーブルを考える。このデータベースに対するクエリで、苗字フィールドが指定される確率は 0.9 である。このようなレコードが約 100 万件 ($= 2^{20}$) 格納されているとする。

フィールド	確率
苗字	0.9
名前	0.8
年齢	0.5
性別	0.1

図 3　あるデータベースとフィールドがクエリで指定される確率

このレコードを識別するために 20bit のハッシュ値を用意する。もし 20bit 全てを苗字の識別に使うとすればハッシュ値により目的のレコードか特定できるため、レコードにアクセスする回数は 1 回で済む。しかし、苗字フィールドがクエリで使われなかった時は、全てのレコードにアクセスしなくてはならない。平均すると

$$0.9 \times 1 + 0.1 \times 2^{20} \approx 100000 \tag{1}$$

約 10 万件のレコードにアクセスすることになる。ここで 20bit の使い方を 10bit を苗字フィールドの識別に、10bit を名前フィールドの識別に使うとする。この場合は苗字フィールドと名前フィールドが両方ともクエリで使用された場合はアクセスするレコードは 1 レコードで済むうえ、名前だけが指定された場合でも 10bit は確定するため、残りの 10bit のありうるパターン ($= 1024$ 件) を全て試せば良い。アクセスするレコードは 1024 件である。平均すると

$$0.72 \times 1 + 0.18 \times 1024 + 0.08 \times 1024 + 0.02 \times 2^{20} \approx 20000 \tag{2}$$

約 2 万件のレコードにアクセスすることになる。20bit 全てを苗字フィールドの識別に使うケースより、平均アクセス回数を削減できた。このようにフィールドに割り当てる bit 数を変えることレコードにアクセスする回数を変化させられる。

また部分一致処理がしやすい利点を持っている。先の例のように 10bit ずつ苗字フィールドと名前フィールドに割り当てるとする。各ハッシュ値はそれぞれ独立であるため、苗字を指定された場合、その苗字を指し示す 10bit は繰り返し再利用することができる。

4　Fluent Join

Fluent Join は関連研究で紹介したリニアハッシュと多次元ハッシュを組み合わせた多次元リニアハッシュファイルデータベース上で機能する結合演算である。

多次元リニアハッシュファイルデータベースでは多次元ハッシュのように連続した独立のハッシュ値を用いる。それぞれのハッシュ値は図 4 のように別々のハッシュ関数により生成されている。

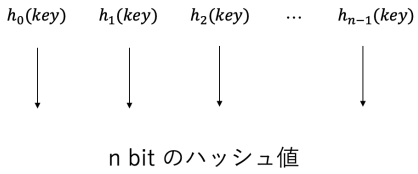

図 4　n bit のハッシュ値が作られる例

それぞれのハッシュ関数に同じキーを与えるのではなく、データベースのキー属性を与えることで複数の属性に対応したハッシュ値を作ることができる。例えば 2 つのキー属性 A,B を持つデータベースに対しては図 5 のように、m 個のハッシュ関数には A のキーを与え、n 個のハッシュ関数には B のキーを与えることで A,B 両方の値に対応するハッシュ値を構成できる。

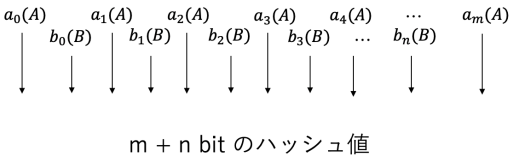

図 5　m+n bit のハッシュ値が作られる例

結合を行う上では、複数属性に跨がることで、ハッシュ値が同一である部分を再利用できるメリットを最大限に生かしやすい。例えばテーブル X のハッシュ値が属性 A で 3bit, 属性 C で 1bit の 4bit で構成されていて、テーブル Y のハッシュ値が属性 A で 3bit,B で 1bit の 4bit で構成されているとする。図 6 に示すように、X と Y で結合を行う時、属性 A のハッシュ値は X でも Y でも使えるため繰り返し利用することができる。繰り返し利用することでハッシュ値の演算の省力と、結合可能部分の検索を簡単に行うことができる。

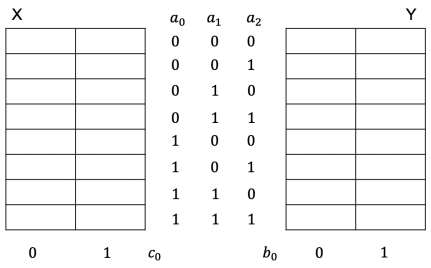

図 6　ハッシュ値が共有されるデータベース

サンプルとしてキー属性が A で、ハッシュ関数が 20 個の属性 A をキーとして受け取る関数で構成されるテーブル X と、キー属性が A,B で、ハッシュ関数が 15 個の属性 A をキーとして受け取る関数と 6 個の属性 B をキーとして受け取る関数で構成されるテーブル Y を考える。この時、図 7 に示すように A15 までは 2 つのハッシュ関数での共通部分となる。

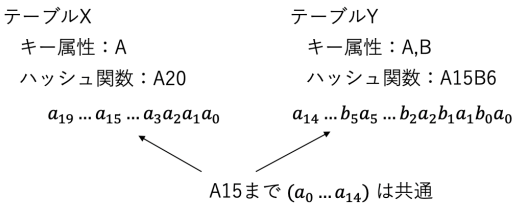

図 7　サンプルデータベースとそのハッシュ関数

この 2 つのテーブルを Fluent Join する時、まずは共通する A15 までのハッシュ値を作り、残りの A の 5bit と B の 6bit の取りうる数値を全て試して結合すれば良い。そのアルゴリズムを Algorithm1 に示す。

Algorithm 1 Fluent Join for 2 tables

```
for all possible 0-1 assignments to 15bit of A do
    fetch pages of X with remaining 5bit of A
    for all possible 0-1 assignments to 6bit of B do
        fetch pages of Y
        join X and Y
    end for
end for
```

共有できるハッシュ値は共有し、残りの部分を for 文を使うことでコストを大きく削減できる。この方法は 2 テーブルの結合に限らず、3 テーブル以上の結合も可能にする。

5　実　　験

今回実験を行う上で、メモリ上で取ってきたデータどうしを結合させる内部的な動作に注目し、その動作について並列化を試みた。

ディスクからメモリにデータを取ってきた前提で、そのデータの結合を行う。並列処理のために OpenMP [5] を導入する。OpenMP は並列プログラミングで広く利用される API で、スレッド並列なプログラムを簡潔に実装できる。なお、実行環境は以下の通りである。

表 1　実行環境について

CPU	Intel(R) Xeon(R) CPU E5-2698 v3
メモリ	130 GB
OS	Ubuntu 18.04.2 LTS
コンパイラ	gcc (Ubuntu 4.8.5-4ubuntu8) 4.8.5
コンパイラオプション	–fopenmp, -std=C99
物理 CPU 数	32

ただし現実的には 130GB 全てを DBMS が使えるわけではない。そこで、8GB が DBMS が使える量と仮定する。その場合、メモリに取ってこれるデータ量はそれぞれから 4GB ずつということになるので、4GB どうしのデータの結合を並列処

理し、その高速化の度合いを調査した。

使うスレッド数を 1,2,4... と倍にしていけば、理論的には処理時間は 0.5,0.25,... と半減していくはずである。今回はその検証のため、スレッド数による処理時間の変化を実験で調査する。またデータ数の違いによる変化も検討する。

6 評　　価

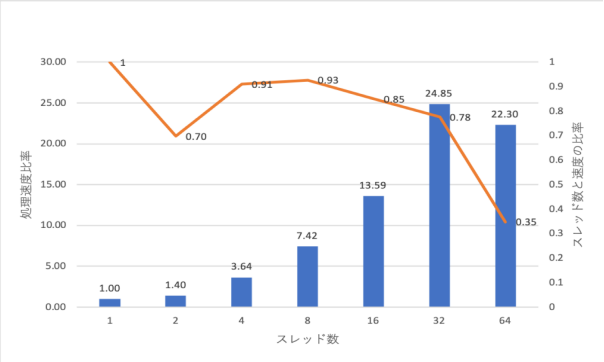

図 8　スレッド数による実行速度の変化

今回行った実験の結果は図 8 に示すような結果となった。青の棒グラフが 1 スレッド時の処理速度を 1 としたときの処理速度の比率を表しており、オレンジの線はその処理速度比率をスレッド数で割った値を表している。これが大きいと効率よく並列化ができているということになる。

この結果から、N スレッドにすれば N 倍早くなるというわけではないものの、ある程度の高速化が見込めることがわかった。中でも 8 スレッドの時の比率が 0.93 であり、効率よく並列化できていることがわかる。

スレッド数 2 の時は OpenMP のシステム使用時間がネックとなって高い性能を出すことができなかったが、結合の実行時間自体は削減することができました。また、64 スレッドの場合は 32 スレッド時より処理時間が遅くなってしまったが、これは実験環境が CPU 物理コアが 32 個しかないためであると考えられる。ハイパースレッディングを利用して 64 スレッドまでの実行が可能だったが、コアの負担が大きくなり性能を出せなくなったと考えられる。1 スレッドあたりの実行時間についても、32 スレッドの場合は 0.41 秒であったが、64 スレッドでは 0.46 秒であり減速した。

7 終 わ り に

データ構造として、リニアハッシュファイルデータベースを用いることで、予め Join を行いやすい状態でデータを格納することができる。またその上で行われる Fluent Join は高速で同じデータを 2 回以上読み書きしないというメリットがある。

本研究では多次元リニアハッシュファイルデータベース上での Fluent Join の並列処理による高速化が可能であるかを調査することが目的であった。

文　　献

[1] 遠山元道, “多次元リニアハッシュファイルデータベースにおける流暢なアルゴリズム,” 情報処理学会第 34 回全国大会, pp. 407–408, 1987.

[2] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” Communications of the ACM (CACM), Vol. 13, No. 6, pp. 377–387, 1970.

[3] Aho, A. V., Ulman, J.D., “Optimal Partial-Match Retrival when Fields are Independently Specified,” ACM Trans on Database Systems, Vol. 4, No. 2, pp. 168–179, 1979.

[4] Litwin, W., “Linear Hashing: A New Tool for file and table addressing,” International Conference on Very Large Databases, pp. 168–179, 1980.

[5] “OpenMP,” https://www.openmp.org

[6] “PostgreSQL 文書,” https://www.postgresql.jp/document