SuperSQLにおける分散処理を用いた大規模データへの対応

田畑　篤智†　　五嶋　研人†　　遠山　元道†

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

あらまし　日々増え続ける各種データを集計し, その結果を効果的に提示することは様々な場面で重要となる. 例えば小売店経営者の意思決定のための日々数千万件のオーダーで増え続ける購買データの集計とその結果の提示などがそうである. 本研究では SuperSQL を用いて, この多量のデータの集計および提示を可能とすることを目的としている. 現在の SuperSQL では内部処理の関係でこのような多量のデータを扱うことができない. そこで SuperSQL クエリにある集約を基にしたクエリの分割機能と種々の処理への Hadoop を用いた分散処理の導入を行なった. その結果 SuperSQL において今までは処理を行うことができなかった量のデータも処理を行うことができるようになり, SuperSQL によって出版できるデータの種類が広がった.

キーワード　SQL, SuperSQL, Hadoop, Bigdata

1. はじめに

日々増え続ける各種データを集計し, その結果を効果的に提示することは様々な場面で重要となる. 例えば小売店経営者の意思決定のためには, 日々約 1000 万件以上増え続ける購買データを集計し見やすく出版する事が必要となる. これを実現するためには, 数千万オーダーのデータも解析が行える DBMS を用いてデータの取得及び集約を行い, そのデータを人の手で整形を行い表などの形で出版する必要がある. この場合まずデータを取得するため SQL に対する理解をしている人が必要になり, その後プレゼンテーションの効果を考えデータを整形する人も必要となる. このようにデータ取得と出版が分離されそれぞれに労力が必要になるという問題が生じる. この問題を一つの言語でデータ取得から出版までをこなせるようにしたのが SuperSQL である.

SuperSQL とは, Target Form Expression(TFE) という属性名を用いて構造指定をする表現形式を用いてデータベースの値を様々な構造で出力することができる, SQL の拡張言語である.SQL ではフラットな一次元の表のみの作成が可能だが, SuperSQL を用いれば多様な構造の表を簡単に作成することができる. 出力先のメディアも HTML, PDF, PHP など様々なメディアに対応しており, 近年では Unity と組み合わせ 3D でのデータ可視化も行われている. また SuperSQL では SQL と異なり記述者のクエリの文脈依存の集約を行うことができる. 第 5 章で詳述するが SQL では一つのクエリにおいて複数の集約を用いたい場合, SELECT 句で列挙した属性全てで集約をしないといけない. それに対して SuperSQL では一つクエリ上で記述者の意図した属性で集約を行うことができそれは複数箇所でも可能となっているのである.

この SuperSQL を用いればデータ取得と整形を同時に行うことができる. しかし上記のような数千万オーダーのデータの解析を行おうとすると前述の文脈依存集約をする為の処理で問題が発生する. 本論文ではこの問題を解決する方法を提案する. まず SuperSQL クエリの集約の位置を元にクエリの分割を行いデータベースに問い合わせを行う時点で集約をできるだけ行い取得データの削減を行えるようにした. この際用いた DBMS は SQL on Hadoop である Hive である. そしてデータ取得後の整形処理でも多量のデータに対して処理を行えるように Hadoop の Mapreduce を用いて分散処理を行った. この結果 SuperSQL で多量のデータを元にした出版ができるようになった.

以下本論文の構成を示す. 第 2 章では関連研究と関連技術について述べ, 第 3 章では SuperSQL についての説明, 第 4 章では SuperSQL と SQL の集約の違いを説明, 第 5 章ではクエリ分割について, 第 6 章では Mapreduce を用いた分散処理の導入について説明を行う. 第 7 章では実験評価を述べ, 第 8 章では結論を述べる.

2. 関連研究・技術

本章では今回用いる分散処理フレームワークである Hadoop の紹介をする. その後, Hadoop 上で動く SQL ライクなクエリを動かす SQL on Hadoop について紹介を行う.

2.1 分散処理フレームワーク

Apache Hadoop [4] は大規模データの分散処理フレームワークである. “HDFS”, “MapReduce”, “YARN” からなる. “HDFS” は “Hadoop Distributed File System” の略称で Hadoop 独自の分散ファイルシステムで, サイズの大きいファイルをブロック毎に分けて書くノードに格納する形式をとっている. “MapReduce” は Google の MapReduce [5] をベースとして Java 実装されている. 分散したい処理を Map 処理と Reduce 処理に分けてユーザが記述することで他の分散処理は自動的に行われる. “YARN” は “Yet Another Resource Negotiator” の略称でリソースの管理とジョブのスケジューリングを行っている.

本研究ではこの Hadoop を用いることで分散処理を導入している.

2.2 SQL on Hadoop

Apache Hive [6] は, Hadoop で扱えるファイルシステムに格納された大規模なデータセットに対してデータの取得及び分析を行う. その際 HiveQL という SQL-like なクエリを用いている. HiveQL は内部の処理で MapReduce 処理に置き換えられ実行がなされている. また前述のように MapReduce を用いているため, バッチ処理に適している. 本研究ではクエリ分割したのちの SQL クエリを Hive を用いて実行することになる.

Cloudera Impala [7] は Hive がバッチ処理に適していたことに対して, リアタイムの処理を可能とするために新しい分散クエリエンジンを用いている. また Hadoop 上で動作が可能となっている.

Apache Hawq [8] は Hadoop 上で動く SQL の実行エンジンである. PostgreSQL をベースとしているため他の SQL on Hadoop に比べて標準 SQL への準拠レベルが高いことが特徴である.

3. SuperSQL とは

SuperSQL は関係データベースの出力結果を構造化し, 多様なレイアウト表現を可能とする SQL の拡張言語であり, 慶應義塾大学遠山研究室で開発されている [1] [2]. そのクエリは SQL の SELECT 句を GENERATE <*media*> <*TFE*> の構文を持つ GENERATE 句で置き換えたものである. ここで <*media*> は出力媒体を示し, HTML, PDF, Mobile_HTML5 [3] などの指定ができる. また <*TFE*> はターゲットリストの拡張である Target Form Expression を表し, 結合子, 反復子などのレイアウト指定演算子を持つ一種の式である.

3.1 SuperSQL アーキテクチャ

SuperSQL のアーキテクチャを図 1 に示す. SuperSQL 処理系は, 構文解析部 (Parser), リス ト構造生成部 (List Constructor), メディア生 成部 (Code Generators) から成る. SuperSQL ク エリが発行されると, まず構文解析部で通常の SQL 文とレイアウト式に分け, SQL 文を DBMS に渡し て検索結果を受け取る. リスト構造生成部では 受け取ったフラットな結果に対し, レイアウト 式に従って階層的な構造を持たせる. 最後にメ ディア生成部がクエリで指定したメディアファ イルを結果として出力する.

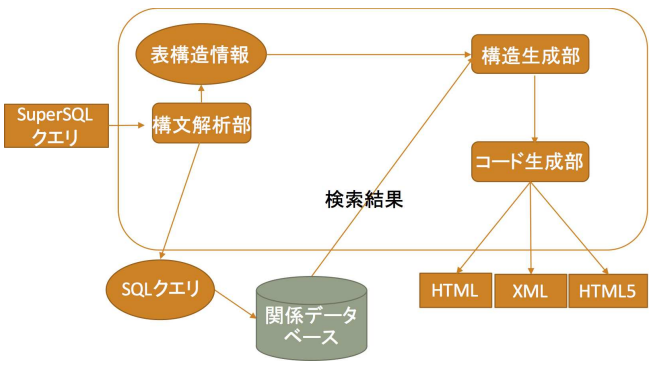

図 1 SuperSQL アーキテクチャ

3.2 結 合 子

結合子はデータベースから得られたデータをどの方向 (次元) に結合するかを指定する演算子であり, 以下の 3 種類がある. 括弧内はクエリ中の演算子を示している.

- 水平結合子 (,)

データを横に結合して出力する.

例 : name, place

name	place

- 垂直結合子 (!)

データを縦に結合して出力する.

例 : name! place

name
place

- 深度結合子 (%)

データを 3 次元方向へ結合する. 出力が HTML ならばリンクとなる.

例 : name % place

name	→	place

3.3 反 復 子

反復子は指定する方向に, データベースの値があるだけ繰り返して表示する. また反復子はただ構造を指定するだけではなく, そのネストの関係によって属性間の関連を指定できる. 例えば

[部署名]! , [雇用者名]! , [給料]!

とした場合には各属性間に関係はなく, 単に各々の一覧が表示されるだけである. 一方, ネストを利用して

[部署名! [雇用者名, 給料]!]!

とした場合には, その部署毎に雇用者名と給料の一覧が表示されるといったように, 属性間の関連が指定される. 以下, その種類について述べる.

- 水平反復子 ([ ],)

データインスタンスがある限り, その属性のデータを横に繰り返し表示する.

例 : [Name],

name1	name2	…	name10

- 垂直反復子 ([ ]!)

データインスタンスがある限り, その属性のデータを縦に繰り返し表示する.

例 : [Name]!

name1
name2
…
name10

3.4 装 飾 子

SuperSQL では関係データベースより抽出された情報に, 文字サイズ, 文字スタイル, 横幅, 文字色, 背景, 高さ, 位置などの情報を付加できる. これらは装飾演算子 (@) によって指定する.

< 属性名 >@{< 装飾指定 >}

装飾指定は “装飾子の名称 = その内容” として指定する. 複数指定するときは各々を “,” で区切る.

[name@{width=100, color=red}]!

3.5 関　　数

3.5.1 image 関数

image 関数を用いると画像を表示することが可能となる. 引数には属性名, 画像ファイルの存在するディレクトリにパスを指定する.

image(pict, “./pic”)

3.5.2 link 関数 (出力メディアが HTML の場合のみ)

link 関数は, FOREACH 句と同時に用いる. これらを用いることで深度結合子と同様にリンクを生成することができる.

link(name, “./menu.ssql”, place)

4. SQL と SuperSQL の集約の違い

この章では, SQL の集約と SuperSQL の文脈的集約について説明を行う.

4.1 SQL の集約

SQL の集約関数では属性の和を取る SUM, タプル数を数える COUNT などが存在する. 例えばとある学校で “数学” の授業の年齢及び性別ごとの生徒の点数の合計点を求めるクエリは以下のようになる.

SQL の集約例 1

```
SELECT s.age, s.gender, SUM(p.score)
FROM student s, class c, performance p
WHERE s.id = p.s_id AND c.name = ‘数学’ AND c.id
= p.c_id
GROUP BY s.age, s.gender
```

このクエリでは部署の名前 “s.age” と “s.gender” 毎に集約が行われている. このように集約関数以外の属性で集約され計算結果を返すのが SQL の集約機能である. ただ SQL の集約機能では一つのクエリ上で複数の集約を行った場合ユーザーの意図が適切に反映されないことがある. 例えば先ほどの例で年齢ごとの合計点も同時に出したいと思い以下のように一つのクエリを書いたとする.

SQL の集約例 2

```
SELECT s.age, SUM(p.score), s.gender, SUM(p.score)
FROM student s, class c, performance p
WHERE s.id = p.s_id AND c.name = ‘数学’ AND c.id
= p.c_id
GROUP BY s.age, s.gender
```

このクエリを実行すると, 集約関数はどちらも “s.age” と “s.gender” で集約されてしまい, ユーザーの意図と異なる検索結果が返ってきてしまう. つまりこの場合はユーザーのクエリの文脈的意味を反映した集約を行えていないと言える.

4.2 SuperSQL の集約

第 3 章で説明したが SuperSQL の TFE にはグルーピングという考えが反映されている. そのため属性の書いた順を元に集約が行われるので, SQL での集約のように集約しようと思った地点以降の属性によって集約されることはない. つまり先ほどの学校の生徒の成績で年齢別に点を表示し, 年齢別かつ性別毎に点数を表示したい場合は

SuperSQL の集約例 1

```
[s.age, SUM[p.score],
        [s.gender, SUM[p.score]]!
]!
FROM student s, class c, performance p
WHERE s.id = p.s_id AND c.name = ‘数学’ AND c.id
= p.c_id
```

という形で記述をすれば年齢の横に年齢別の合計点が, 性別の横に年齢別で性別毎の合計点が表示される.

他にも性別毎の点と年齢毎の点を同時に出力したい場合は

SuperSQL の集約例 1

```
[s.age, SUM[p.score]]!, [s.gender, SUM[p.score]]!
FROM student s, class c, performance p
WHERE s.id = p.s_id AND c.name = ‘数学’ AND c.id
= p.c_id
```

という様にグルーピングなくしてしまえば良い.

この様に SuperSQL ではどこで集約したいかを記述者の直感に従ってグルーピングを用いて記述すればその通りに集約が行われることになる.

5. クエリ分割

従来の SuperSQL では第 4 章で説明した文脈的集約を行うためクエリ上に現れた属性を並置し一括してデータベースに問い合わせ, その結果を SuperSQL の内部処理を用いて集約などの処理を行っている. しかしこの方法では検索結果が数千万件に上る場合処理しきれない, という問題が生じることがある. この問題を解決するため, 本章ではクエリを分割する方法を説明する. まず SuperSQL で用いる表現形式である TFE 特有の属性間の構造などを示した木構造の説明を行い, その後分割のアルゴリズムを説明する. また, このクエリ分割を行った際の取得データの減少の具体例も説明する.

尚, 以降本論文内でサンプルクエリを用いる際の元となるスキーマは以下のようになっている.

- items(id, name, price, genre)
- genres(id, name)
- customers(id, name, age, gender)
- boughts(id, c_id, i_id, num. day)

このデータベースはとある小売店の商品とそのジャンル, 顧客に関する情報と購買履歴を保持している. 顧客登録数は約 7000

万件, ジャンルは 40 種, 商品数は約 3500 点, 購買件数は 1 日約 1000 万件, を前提としている.

5.1 TFE の木構造

SuperSQL では第 3 章で説明したように反復子 (Connector) や結合子 (Grouper) を用いて属性同士の位置関係や集約関係を記述している. そして図 1 にあるように, 記述した SuperSQL クエリは構文解析部で処理され SQL クエリと表の構造情報に分かれる. この時, 構造情報は木構造で保持されている. 以下に SuperSQL クエリの一例 (TFE のみ) とそれを構文解析し木構造にした際の木構造を示す.

SuperSQL クエリの TFE 部分

```
[c.gender!
        [c.age, count[c.id],
                [g.name,
                        [i.name, sum[b.num]]!
                ]!
        ]!
]!
```

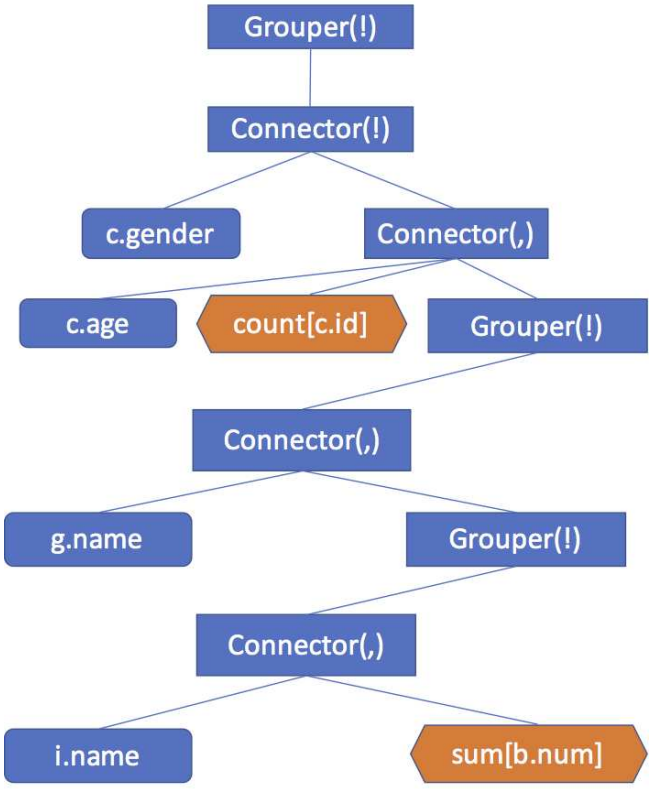

図 2　木構造の例

図 2 の様に結合子, 反復子, 属性等で木は構成されている. 次節では図 2 の木を元にクエリ分解のアルゴリズムを説明する.

5.2 クエリ分割アルゴリズム

以下にクエリ分解のためのアルゴリズムを示す. このアルゴリズムは各集約部分とその集約に関係する属性をペアで取得する事を目的としている. このアルゴリズムの説明を行う. まず 2, 4 行目で属性を保管する attributes_list と結果を保管する

Algorithm 1 クエリ分解アルゴリズム

Require: TFE 解析後の属性間の関係を示す木構造 (List 型)
Ensure: 各集約部分を Key, その集約に関係する属性を Value とした HashMap

```
1: //探索中に発見した属性を追加するリストを生成
2: attributes_list ⇐ φ
3: //結果を保管する Hashmap を作成
4: result ⇐ φ
5: 関数 getDependencies(TFE_tree)
6: if 現在参照している TFE_tree のノードが属性なら then
7:     //attributes_list の一番下の階層に属性を追加
8:     attributes_list ⇐ 属性
9: end if
10: if 現在参照している TFE_tree のノードが集約なら then
11:     //result にノードの属性と現在の attributes_list を保管
12:     result ⇐ { 属性, attributes_list}
13: end if
14: if 現在参照している TFE_tree のノードがグルーパーなら then
15:     tmp_list ⇐ attributes_list
16:     //attributes_list に空のリストを追加
17:     attributes_list ⇐ []
18:     getDependencies(ノードの小要素)
19:     //attributes_list からさっき追加したリストを削除
20:     attributes_list ⇐ tmp_list
21: end if
22: if 現在参照している TFE_tree のノードがコネクターなら then
23:     for all 現在のノードの小要素それぞれについて do
24:         getDependencies(小要素)
25:     end for
26: end if
```

result を宣言している．その後 getDependencies 関数に TFE を解析した木構造を渡している．その一番上のノードの種類によって 6, 10, 14, 22 行目のそれぞれの分岐が発生する．

14 行目のグルーパーの場合一時的に attributes_list を tmp_list という変数に保管し，末尾に空のリストを加えている．そして小要素を引数として getDependencies 関数を呼び出している．20 行目で追加したリストを削除しているのは集約関数は自分よりグルーパーを挟んで下にある属性によって集約が行われないからである．

22 行目のようにコネクターだったら，その子要素分 getDependencies 関数を呼び出す．

6 行目のようにノードが属性なら attributes_list の一番奥のリストに追加し，10 行目のように集約だったら集約属性と現在の attributes_list をペアにして result に保管する．

このようにして作成した集約属性とそれに関係する属性の依存関係を元にクエリの分割を行うことで取得データ量を減らすことができる．

5.3 具 体 例

上記の “SuperSQL クエリの TFE 部分” を元にアルゴリズム適用の具体例とデータ削減の説明を行う．

5.3.1 アルゴリズム適用の具体例

まずルートノードである一番上の “Grouper(!)” から処理が始まる．attributes_list に空のリストが追加され小要素を getDependencies 関数に与える．“Connector(,)” なのでそのさらに小要素について getDependencies 関数を呼び出す．一つ目の小要素は属性 “c.gender” なので attributes_list に追加される．二つ目の小要素は “Connector(,)” なのでその小要素について getDependencies 関数を呼び出す．一つ目は属性 “c.age” なので attributes_list に追加する．二つ目は集約 “count[c.id]” なので現在の attributes_list とともに result に格納される．尚，この時の attributes_list は同階層に “c.gender” と “c.age” を持つ．次は “Grouper(,)” なので attributes_list に空のリストが追加されその子要素を getDependencies 関数の引数として呼びだす．すると “Connector(,)” なので小要素それぞれについて getDependencies 関数を呼びす．一つ目は属性 “g.name” なので attributes_list に追加し，二つ目は “Grouper(!)” なので先ほどど同じように小要素を引数として getDependencies 関数を呼び出す．同じく “Connector(,)” なので小要素それぞれについて呼び出す．一つ目は属性 “i.name” なので attributes_list に追加，二つ目は集約 “sum[b.num]” なので現在の attributes_list とペアで result に保管される．

以上の処理から result は以下のようになる．

処理結果

result = {count[c.id] = [[c.gender, c.age]], sum[b.num] = [[c.gender, c.age, [g.name, [i.name]]]]}

5.3.2 データ減少の具体例

従来の SuperSQL で “SuperSQL クエリの TFE 部分’ を実行した場合，以下のような SQL クエリ (SELECT 句のみ) が発行される．

SQL クエリの SELECT 句

```
SELECT c.gender, c.age, c.id, g.name, i.name, b.num
```

その結果図 3 ようなタプル群が取得される．取得タプル数は最

gender	age	id	genre_name	item_name	bought_num
male	20	134923	お惣菜	筑前煮	5
male	20	134923	お弁当	牛丼	3
male	35	2432156	お惣菜	唐揚げ	1
male	35	2432156	お惣菜	サラダ	1
⋮	⋮	⋮	⋮	⋮	⋮
female	20	55292431	デザート	ロールケーキ	2
female	20	55292431	お弁当	幕の内弁当	1
⋮	⋮	⋮	⋮	⋮	⋮

図 3 取得結果例 1

も多い boughts のタプル数と同じになり参照する部分によっても異なるが仮に 3 日分を参照したとすると約 3000 万タプルとなる．この量を後の構造生成部で処理しようとするとメモリに乗り切らず処理が行えない，という問題が発生する．

それに対して上記のクエリ分割を適用すると以下の二つのクエリに分割できる．

クエリ分割によってできる SQL クエリの SELECT 句 1

```
SELECT c.gender, c.age, count(c.id)
```

クエリ分割によってできる SQL クエリの SELECT 句 2

```
SELECT c.gender, c.age, g.name, i.name, sum(b.num)
```

SQL クエリ 1 では結果が図 4 のようになり，顧客の性別と年

gender	age	count(id)
male	20	100023
male	21	110914
⋮	⋮	⋮
female	20	97132
⋮	⋮	⋮

図 4 取得結果例 2

gender	age	genre_name	item_name	sum(bought_num)
male	20	お惣菜	筑前煮	2891
male	20	お惣菜	切り干し大根	3024
male	20	お惣菜	唐揚げ	34102
⋮	⋮	⋮	⋮	⋮
female	20	お惣菜	豚トロ焼き	1802
⋮	⋮	⋮	⋮	⋮

図 5 取得結果例 3

齢で集約しているので高々200 行程度となる．SQL クエリ 2 では結果が図 5 のようになり，顧客の性別，年齢，ジャンル名，商品名で集約しているので取得結果の行数は購買数と年齢と性別

の件数の席で高々70 万行となる．このようにクエリ分割を行いあらかじめ集約を行ってしまうことで取得するデータ量は減らすことができる．

6. 分散処理の導入

従来の SuperSQL では一つのクエリしか発行せず集約も内部処理で行っていたため返ってくる検索結果の属性値のセットは属性で見ると 1 種類しかなかった．つまり前述の “SuperSQL クエリの TFE 部分の例” で言うと返ってくるタプルは “c.gender”, “c.age”, “c.id”, “g.name”, “i.name”, “b.num” の検索結果のみだった．本研究では集約を先に行う代わりにクエリを分割したため属性で見ても様々な種類のタプルが返ってくることになった．前述の例でいうと “c.gender”, “c.age”, “count(c.id)” のセットと “c.gender”, “c.age”, “g.name”, “i.name”, “sum(b.num)” のセットの検索結果が返ってくるのである．この様々な種類の結果を 1 種類にまとめる必要がある．本章では第 5 章で生成された SQL クエリにより返ってきた検索結果についてこのまとめる処理を行った．その処理は第 5 章の処理で集約をしてもタプル数が減らせていないことを考え Hadoop の Mapreduce を用いて実装した．

6.1 アルゴリズム

Map 処理では検索結果を Key に, 属性値それぞれが SuperSQL クエリの何番目に登場する属性に対応するかの番号をリストにしたものを Value とするキーバリュー型にする．Reduce 処理では Value の配列を元に二つの属性値を比較しどちらかがどちらかに完全に一致するなら同一のものとみなし合体させる処理を行う．以下にそのアルゴリズムを示す．

Algorithm 2 Map 処理

Require: 検索結果と SuperSQL クエリの情報
Ensure: 検索結果を Key, 対応する番号のリストを Value とした kv セット

```
1: 関数 Map(検索結果のタプル群, クエリ識別子と番号のリスト)
2: //結果保持用の haspmap 作成
3: result ⇐ φ
4: for all 検索結果タプルそれぞれについて do
5:     //各検索結果タプルとそれに該当する番号のリストをセットにして格納
6:     result ⇐ { タプル = 番号のリスト }
7: end for
```

6.2 具 体 例

上記の “SuperSQL クエリの TFE 部分の例” を元にアルゴリズム適用の具体例の説明を行う．
まず分割後のクエリ 1 で得られる検索結果は図 4 を元に考えると以下のようなリストになる．

検索結果 1

[[‘male’, 20, 100023], [‘male’, 21, 110914], ⋯, [‘female’, 20, 97132], ⋯]

となる．同様に分割後のクエリ 2 で得られる検索結果は

Algorithm 3 Reduce 処理

Require: Map 処理の結果
Ensure: 該当検索結果をまとめたタプル群

```
1: 関数 Reduce()
2: //結果保持用のリスト作成
3: result ⇐ φ
4: for all Map 処理の結果それぞれについて do
5:     for all Map 処理の結果それぞれについて do
6:         if 2 つの処理結果の番号リストの内容が完全に異なっていたら then
7:             //番号リストの番号順に合体する
8:             marge(kv セット 1, kv セット 2)
9:         else if 2 つの処理結果の番号リストが一部一致していたら then
10:            //一致している部分の属性値を比較し全て同じなら合体
11:            for all 一致している番号リストの番号それぞれについて do
12:                if 該当する属性値が一致していなかったら then
13:                    break
14:                end if
15:                marge(kv セット 1, kv セット 2)
16:            end for
17:        end if
18:    end for
19: end for
20: //二つの kv セット (番号リスト含む) の検索結果を合体する関数
21: 関数 marge(kv セット 1, kv セット 2)
22: count ⇐ 0
23: result_tmp ⇐ φ
24: for i = 0  SuperSQL クエリの属性数-1 do
25:     if どちらかの kv セットの番号リストの番号 = count then
26:         result_tmp ⇐ 番号に該当する属性値
27:     end if
28:     count++
29: end for
```

検索結果 2

[[‘male’, 20, ‘お惣菜’, ‘筑前煮’, 2891], [‘male’, 20, ‘お惣菜’, ‘切り干し大根’, 3024], …, [‘female’, 20, ‘お惣菜’, ‘豚トロ焼き’, 1802], …]

と, なる.

map 処理では検索結果と属性値の該当する属性の SuperSQL クエリ上で記述されている順番をセットにするので検索結果 1 と 2 に対して行うと以下のようになる.

検索結果 1 の Map 処理後

{[‘male’, 20, 100023] = [1, 2, 3], [‘male’, 21, 110914] = [1, 2, 3], …, [‘female’, 20, 97132] = [1, 2, 3], …}

検索結果 2 の Map 処理後

{[‘male’, 20, ‘お惣菜’, ‘筑前煮’, 2891] = [1, 2, 4, 5, 6], [‘male’, 20, ‘お惣菜’, ‘切り干し大根’, 3024] = [1, 2, 4, 5, 6], …, [‘female’, 20, ‘お惣菜’, ‘筑前煮’, 2891] = [1, 2, 4, 5, 6], …}

この結果を Reduce 処理に渡し合成すべきと判断できたものを合体させる. その結果最終的に以下のような検索結果にまとまる.

最終的な検索結果

[[‘male’, 20, 100023, ‘お惣菜’, ‘筑前煮’, 2891], [‘male’, 20, 100023,‘お惣菜’, ‘切り干し大根’, 3024], …, [‘female’, 20, 97132, ‘お惣菜’, ‘豚トロ焼き’, 1802], …]

7. 評　　価

本研究の評価として性能評価を行う. そのためにクエリ分割機能とデータ取得後の処理に対する分散処理を用いた場合と用いなかった場合の各データ量ごとの実行時間を計測した. 評価で使うデータとしては第 5 章で挙げたスキーマを元としたデータベースを用いており, 購買件数は 1 日分が 1000 万件としているいて評価の際に参照する対象のデータ量はそれぞれ半日分 (500 万件), 1 日分 (1000 万件), 一週間分 (7000 万件), 1ヵ月分 (3 億件) となっている. 使用するクエリは第 5.1 節の “SuperSQL クエリの TFE 部分” とする.

実行環境は以下のスペックのブレードサーバを 11 台用意し 1 台をマスターノード, 残りの 10 台をスレーブノードとした.

- CPU: 2.80GHz 4 Cores
- メモリ: 96GB
- HDD: 270GB
- OS: CentOS 7.3.1611
- Hadoop 2.8.2
- Hive 2.2.0

その結果が図 6 である. この結果を見るとまず本研究のクエリ分割機能と分散処理を導入しなかった場合のグラフがなくなっている. これは 500 万件でも実行できていないためである.

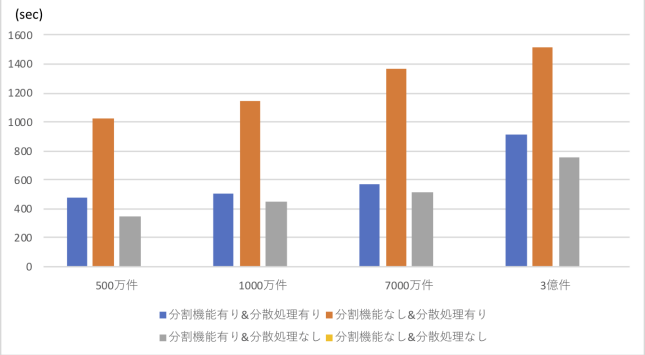

図 6　実 行 時 間

この原因としてはデータの取得に時間がかかり尚且つその後の処理でデータ量が大きすぎてエラーを起こしてしまう, ということがある. このように本研究を全く導入しないと 500 万件のデータも処理が行えない. 実際に SuperSQL を使用していく中ではこれでも問題ない場合は多々あるが, 本研究では多量の購買データの解析のための処理を行えるようにすることを目的としているため従来の SuperSQL の処理では不十分であったことが再確認できた.

クエリ分割機能のみを導入すると, 3 億件のデータ量の場合でも処理できていることがわかる. それは今回用いたクエリは購買データを商品名などで集約しているので, クエリ分割する場合の検索結果のデータ量が購買件数に関係ないためであることが考えられる.

分散処理のみを導入した場合は処理できているもののクエリ分割機能のみを導入した場合と比べて時間がかかってしまっていることがわかる. ここで時間がかかっている原因としては Map 処理や Reduce 処理に時間がかかっていることが考えられる. つまり当然のことではあるが検索結果のデータ量が多い場合は分散処理を導入しないとそもそもデータ取得後の処理を行うことができないので分散処理の導入は必要だが, 検索結果のデータ量が従来の SuperSQL で処理できる量であれば分散処理を用いないほうが早いことになる.

クエリ分割機能と分散処理をどちらも導入した場合はクエリ分割機能のみを導入した場合に比べて遅くなっていることがわかる. その原因は前述の分散処理のみを導入した場合と同じで分散処理そのものに時間がかかっていることだと考えられる. 使用したクエリではクエリ分割をして取得データ量を減らしてしまえば従来の SuperSQL を用いて処理ができるので, クエリ分割のみを導入した場合の方が早くなるのである.

このように本研究のクエリ分割機能はどのようなデータベースを対象としていても取得データ量を削減できるため有用であると言える. しかし分散処理については従来の SuperSQL の処理では行えない場合のみ有効にする, というように使う状況を考えないといけないことがわかった.

8. 結　　論

SuperSQL では小売店の購買データなど大規模なデータを処理することができなかった。そこで本研究では SuperSQL 内部

で集約を基としたクエリの分割を行い取得データを削減し, その後の処理に Hadoop を用いた分散処理を導入した。このことにより SuperSQL によって大規模データの処理が可能となり, SuperSQL を用いたデータ出版を行えるデータの種類が増えた. しかし第 7 章にあるように分散処理に関しては有効な場合と逆に処理を遅くしてしまう場合があるので適用するかしないかを判断する機能が必要になる.

今後の研究としては, HDFS からの取得データ量が少なければ Apache Spark でも十分処理できるのでその導入を検討を行う. また, NoSQL データベースに保管されたデータについても SuperSQL を用いて必要なデータだけを取得し集計し出力することも考えている.

また BI ツールとして活用できるように, 表形式のみならず様々なデータ出版の形式を SuperSQL で使用できるようにすることも今後の研究として考えている.

文　　献

[1] SuperSQL: http://SuperSQL.db.ics.keio.ac.jp

[2] M. Toyama: “SuperSQL: An Extended SQL for Database Publishing and Presentation”, Proceedings of ACM SIGMOD ’98 International Conference on Management of Data, pp. 584-586, 1998

[3] 五嶋 研人, 遠山 元道. “SuperSQL によるモバイル Web アプリケーション生成機構の実装”, 慶應義塾大学 修士論文, 2013.

[4] Apache Hadoop: http://hadoop.apache.org/

[5] Jeffrey Dean, Sanjay Ghemawat: “MapReduce: simplified data processing on large clusters”, Communications of the ACM - 50th anniversary issue, Volume 51 Issue 1, January 2008, pp 107–113

[6] Apache Hive: https://hive.apache.org/

[7] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, Michael Yoder: “Impala: A Modern, Open-Source SQL Engine for Hadoop”, 7th Biennial Conference on Innovative Data Systems Research

[8] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, Milind Bhandarkar: “HAWQ: a massively parallel processing SQL engine in hadoop ”, Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1223–1234, 2014