大規模データベースにおける処理データサイズのリアルタイム計測手法の検討

齋藤　和広†　渡辺　泰之†　村松　茂樹†　小林　亜令†

†株式会社 KDDI 研究所 〒102-460 東京都千代田区飯田橋 3-10-10 ガーデンエアタワー

E-mail: †{ku-saitou, yasuyuki, mura, arei}@kddilabs.jp

あらまし　アドホックなクエリにおける実行時間予測や，複数データベースへのクエリ振り分け等，クエリ実行前にクエリ内の処理毎のデータサイズが必要となる場合がある．このような場面ではユーザクエリ実行時に処理データサイズを計測する必要があるが，対象となるテーブルが大規模な場合，このような計測によってユーザクエリに大幅な遅延が発生する．そこで，ユーザクエリ実行時に，処理対象テーブルからランダム抽出したサンプリングデータに対して対象クエリを実行することで選択率（Selectivity Factor）を計算し，リアルタイムに処理データサイズを計算する手法を提案し，ランダムデータ取得方法，ユーザクエリへの影響，処理データサイズの正確性を検証する．

キーワード　大規模データベース，マルチデータベースシステム，データ仮想化，処理データサイズ推定

1. はじめに

近年の IT 技術の発展により,様々な情報がデータ化され,多種多様で莫大な量のデータが生成されている．これに伴い，様々なデータベースシステムや分散システムが開発され，多種多様で莫大な量のデータを容易に扱うことが可能となってきた．このようなシステムは必ずしも万能ではなく，不得意とするデータの処理や規模などがあり，そのようなシステム固有の特性を考慮した上で利用する必要がある．しかし，ユーザのスキル度合いによっては，実行する処理がその特性と一致しているかどうか，判断できない可能性がある．例えば，ユーザが投稿するクエリの入力データと出力データが小規模だったとしても，処理の途中で生成されるデータの規模が，システムの許容範囲を超えることがある．

上記の課題を解決するためには，複数データベースシステムを仮想的なテーブルにマッピング可能なマルチデータベースシステム[1]や，筆者らが提案してきたデータ共有型マルチデータベースシステム[2]のように，一つ上のレイヤーでデータベースシステム等を使い分けることが考えられる．ここで対象となるデータベースシステム等の使い分け条件が処理データサイズであった場合，処理毎の入出力データサイズを計算する必要がある．特に，ユーザがアドホックに様々なクエリを投稿する環境では，特定のクエリに依存することなく，大規模なデータベースにおいてもユーザクエリの実行時間に大きな影響を与えることなく高速にデータサイズを推定する必要がある．また，多種多様なシステムを対象にデータサイズを予測するためには，そのようなシステムの実装の変化に都度対応するのではなく，共通の方法でデータサイズを推定できることが求められる．

本稿では，特定のデータベースに依存せずに，大規模データベースにおいても少ないユーザ影響で処理データサイズを推定する手法を提案する．具体的には，SQL クエリを用いて処理対象のテーブルからレコードをランダム抽出し，このサンプリングデータに対してユーザクエリと同様の処理を行うことでクエリ実行直前に処理データサイズを推定する．そして，実際のシステムを用いて評価実験を行い，提案手法の有効性と実現性について検証する．

2. 関連研究

データベースに対するクエリ処理のデータサイズやレコード数は，クエリの最適化を行う上で非常に重要な要素であり，古くから様々な推定手法が提案されている．ここでは，モデリングベース，機械学習ベース，サンプリングベースの三つに分類し，それぞれの特徴と課題を言及する．

モデリングベースの予測では，クエリにおける処理毎のデータサイズ計算式（モデル）を予め定義しておき，事前に取得したテーブル情報等と実行時のクエリの内容から処理データサイズを予測する．データサイズ計算式では，対象のクエリの入力データに対する処理後の出力データのサイズ比を表す選択率（Selectivity Factor，以下 SF）を利用する．この SF は，一様分布している列に対する一致検索や範囲検索において定式化が可能である[1]が，一様に分布していない列に対しては，列毎に含まれる値（ドメイン）のヒストグラムを利用してクエリ結果の全てのドメインの統計値を計算する必要があり，この計測にかかるコストが大きい[3]．特に同値の少ない列に関しては，実質的に元のテーブ

ルにクエリを実行することと同様となる．

機械学習ベースの予測では，クエリ処理に関する様々なログ情報を利用して，処理データサイズに関するモデルを自動生成し，このモデルをベースにクエリ毎の処理データサイズを推定する．代表的な手法の例として，回帰分析を用いてドメインの分散を求める手法[4]や，相関分析を利用して近似する複数のクエリ結果から推定する手法[5]がある．教師データとなるクエリ実行の数や種類を十分に用意することで，より精度の高い予測が可能となるが，一方でシステム稼働直後などの十分な教師データがない状況では予測の精度が低くなる．更に，大規模なデータベースでは運用上テーブルの利用頻度に偏りがでることが往々にして有り，あまり使われず十分に学習できないテーブルが発生する．このようなテーブルが利用された場合においても，十分な精度の予測ができない可能性がある．

サンプリングベースの予測は，母集団となる処理対象テーブルからランダムに抽出したサンプリングデータを利用して統計的に処理データサイズを推定する手法である．データサイズを推定するための様々な計算方法やサンプリング手法が提案されている[6]．これらはデータベースが保持するデータファイルに直接アクセスし，1 レコードずつランダムに標本データを取得することで処理データサイズを推定する．そのため，SQL インタフェースを前提として実現するにはサンプリングの数だけのクエリ実行が必要となり，そのコストが無視できない．また，一様でないデータ群を持つ列を利用する JOIN 処理など，一部の処理のデータサイズ推定では，精度を向上させるために母集団のドメインのヒストグラムを利用する必要があり，モデリングベース同様にコストが大きい．

3. ランダム抽出を用いた処理データサイズのリアルタイム計測手法

本稿では，特定のデータベースシステムに依存しない SQL インタフェースを利用して，処理対象のテーブルからサンプリングデータをランダムに抽出し，このサンプリングデータに対してユーザクエリと同じ SQL 処理を行うことでクエリ実行直前に処理データサイズを計測する手法を提案する．

3.1. クエリ実行フロー

本手法の概要図を図 1 に示す．本手法では，ユーザクエリが投稿されると，通常のデータベースシステムと同様にクエリの解釈が行われ，クエリプランが生成される．クエリプランが生成された時点で，クエリプランにおける処理単位の出力データサイズを計算する．SF がない場合，処理毎に対象となるテーブルからランダム抽出して対象の処理を行う SQL クエリを生成し，SF を計測する．この SF を管理データに保存し，利用することで処理毎の出力データサイズを計算する．なお，一致検索や Union 処理，Aggregate 処理等の定式化が可能な処理ではランダム抽出を利用せず，管理データ及び計算式を利用して容量を計算する．

図 2 は単項演算におけるデータサイズ推定の流れを示しており，対象のテーブルからランダム抽出したサンプリングデータに対して計測対象の処理を行い，この結果から計測したレコード数をサンプリング数で割ることで SF を得ることができる．なお SF(X)は，計測対象の処理 X の SF を表す．そしてユーザクエリの当該処理における入力データサイズと，この SF の積をとることで，出力データサイズを算出する．この出力データサイズ計算をクエリプラン上の実行順序に沿って行うことで，ユーザクエリの全ての処理データサイズを計算することができる．

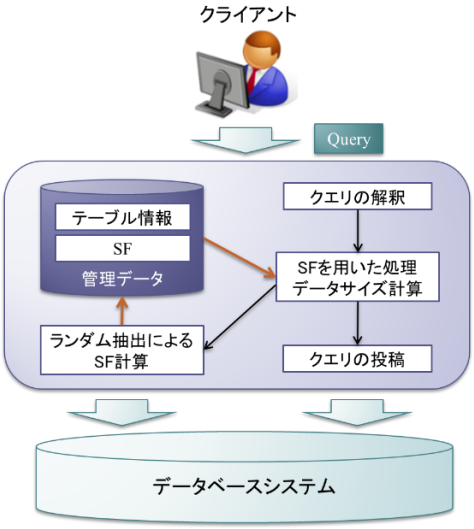

図 1　提案手法の概略図

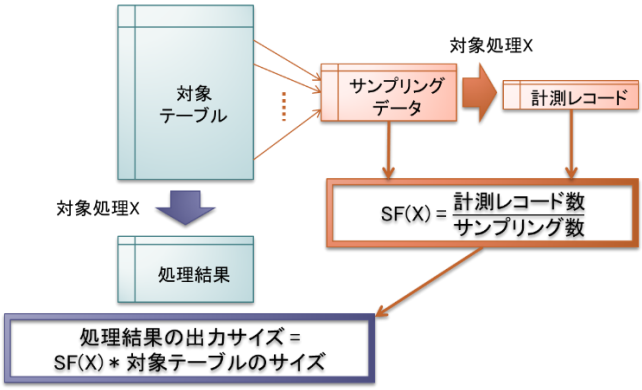

図 2　データサイズ推定の例

3.2. SQL を用いたランダム抽出方法

大規模なデータベースから SQL を用いてレコードをランダム抽出するための要件は，ユーザクエリに大きな影響を与えないことと，完全にランダムであることである．また，様々なデータベースに対応可能とするために,データベース実装に依存しない必要がある．理想的な方法は，データベース実装固有の機能を利用せず，対象のテーブルに対する全件読み込みを行わずに，偏りなくランダムに選択したレコードのみを読み込むことである．本章では，この理想形に近い手段を検討するために，実際のデータベースにおいて共通のインタフェースである SQL で実行可能な方法を以下に列挙する．

ランダムなレコードの出力に利用される最も単純な方法である SQL クエリは，図 3 の方法である．対象のテーブルの全レコードに一時的なランダム値を付与し，そのランダム値でソートする手法である．その結果，ランダムな順序となったレコード群から，上位 n 件を取得することでランダム抽出を可能とする．この手法は，全件の読み出しとソート処理が行われることから，大規模なテーブルでは明らかに高コストであると言える．

```
SELECT * FROM table ORDER BY RANDOM() LIMIT n;
```

図 3　ソートを利用したランダム抽出

全件に対するソート処理を排除する方法として考えられるのが図 4 と図 5 のような，ある特定の列の値を利用したランダム抽出である．これらの方法により，一度の全件読み込みだけでランダム抽出が可能となる．しかし，数値型がないテーブルが存在すると適用できない．また MOD 計算を利用したランダム抽出の場合，ランダムの度合いを調整するために除数と，抽出する値の範囲 X を指定する必要がある．

```
SELECT * FROM table
    WHERE column >= RANDOM() * [最大値] limit n;
```

図 4　ある列の最大値以下のランダム値を利用したランダム抽出

```
SELECT * FROM table
        WHERE MOD(column, [除数]) < X limit n;
```

図 5　ある列に対する MOD 計算を利用したランダム抽出

特定の列に依存しない方法として，行番号を利用する方法がある．行番号の利用は，必ずしも全てのデータベースで実装されているとは限らないが，対応したデータベースの場合，ソートなく，かつ特定の属性に縛られることなくランダム抽出することが可能である．この手法の実際の SQL 文を図 6 に示す．この図ではカラム番号がレコードに紐付いていない場合を想定していて，一度明示的に読みだして列を追加している．このテーブルと，最大値をレコード数 N としたランダム値を持つテーブルで JOIN を行うことで，ランダム抽出が可能となる．問題として列番号にはインデックスがないために, JOIN 処理によるコストがテーブルのサイズに比例して大きくなる点が挙げられる．

```
SELECT * FROM
 ( SELECT row_number() over() AS id, *
   from table ) AS row_id
 JOIN
 ( SELECT GENERATE_SERIES(1,n),
   FLOOR(RANDOM()*N)+1 AS id ) AS rand_table
ON rand_table.id = row_id.id
```

図 6　行番号を利用したランダム抽出

データベースの実装によっては，ランダムに特定数のレコードを読み込む UDF（ユーザ定義関数）が提供されていることがある．例えば図 7 で示すような関数を利用することで，指定数のレコードをランダムに抽出することができる．このような関数は Apache Hive や多くのデータベース製品では実装されているが，OSS である PostgreSQL[7]や MySQL[8]では実装されていない．そのため，前述した行番号を利用した方式も含め，マルチデータベース等の，対象となるデータベースを特定しない環境では別の方法と組み合わせて使わざるを得ない．またデータベースは，テーブルデータをある特定のブロック単位で保持している．このような関数の実装上，ある特定のブロックのみから対象レコードをランダム選択することが起こり得るため，完全なランダムとならない可能性がある．

ランダム抽出手法をコスト，ランダム性，環境依存（データベースの実装と，抽出対象テーブルの属性）の点からまとめると，表 1 のようになる．

```
SELECT * FROM table TABLESAMPLE (n ROWS);
```

図 7　UDF によるランダム抽出（Hive におけるクエリ例）

表 1　ランダム抽出のまとめ

	コスト	ランダム	環境依存
ランダムソート	×	○	○
ランダム値	○	△	実数のみ
Mod 計算	○	△	実数のみ
行番号	△	○	DB 実装
UDF	◎	△	DB 実装

3.3. SQL によるデータサイズ推定方法

ランダム抽出を用いた推定は，前述の通り汎用性を高めるために SQL を前提とした手法を提案する. このことから対象は, 関係モデルにおける演算のうち, SQL で定義されている処理となる．また，ドメインのヒストグラムを利用するなどの，レコードの中身を参照する処理はコストが高いことから行わず，テーブルのレコード数と，属性ごとのドメインの個数と平均サイズのみを事前に集計し利用する．このような環境において, Projection 処理は属性の平均サイズをベースに計算し, Aggregation 処理は属性のドメインの個数から算出する．また Union 処理（Union all ではなく，Distinct 処理が必要となる場合）では，互いに一致しているレコードを Join 処理及び Selection 処理で算出し，レコード数の和から一致しているレコード数を引くことで算出する. すなわち, サンプリングが必要となる処理は, Selection 処理及び Join 処理であり, これらの推定方法をここで述べる.

Selection 処理においては，対象のテーブルに対して rand レコードをランダム抽出し，抽出したデータに対して対象の処理を実行した結果 R からSF_sを算出してデータサイズを推定する.

$$SF_s = \frac{R}{rand}$$

Join 処理においては，二つのテーブルのどちらかに対してのみ，rand レコードをランダム抽出し，この抽出したデータともう一方のオリジナルのままのテーブル(レコード数 b)との Join 処理の結果 R からSF_jを算出し，テーブルサイズを推定する.

$$SF_j = \frac{R}{rand * b}$$

4. 検証実験

4.1. 実験の目的

本実験では，実際のデータベースシステムを利用し，SQL を用いたランダム抽出によるデータサイズ推定の有効性の検証を行う．ここでは特に，ランダム抽出の SQL クエリと，サンプリングデータに対する処理データサイズ計測処理をそれぞれ分けて検証を行う.

初めに，3.2 節で述べた SQL によるランダム抽出手法を実際に実行し，コストと実用性を検証する.

次に，ランダム抽出の手法において，3.3 節で列挙した SQL による抽出を実行し，その実行時間を評価し，ランダム抽出したデータに対して，提案した Selection 及び Join 処理のデータサイズ計測におけるコストと誤差を評価する．Selection 処理においては，一つのテーブルを利用し，条件式の変化によってどのような誤差が生じるかを検証する．そのときに誤差が最大となる条件式で，part と orders で誤差の目標値となるサンプリング数を計測し実行時間を評価する．本実験では，この目標値を 2%と仮定して実施した．この目標値に対する最小のサンプリング数を求め，ユーザクエリに対する本手法の影響度を評価することで，本手法の有効性を検証する．Join 処理においては，結合対象の属性が一様な場合とそうでない場合におけるランダム抽出の効果を評価する.

4.2. 実験環境

対象とするデータベースシステムは，分散処理プラットフォーム Apache Hadoop[9]上のデータを SQL インタフェースで操作できる Apache Hive[10]と，同じく Apache Hadoop 上のデータを SQL インタフェースでオンメモリ処理が可能な Cloudera Impala[11]を利用する. Impala は SF 計測クエリを実行するために利用するが，ランダム取得ができないことから，大規模処理を得意とする Hive をランダム抽出処理に利用する.

システム構成は図 8 に示す環境で，三台のスレーブノード上で処理を行う．表 2 にソフトウェア構成を，表 3 にハードウェア構成を示す.

実験用データセットには TPC-H[12]のうち表 4 に示す三つのテーブルを利用し，データセットの規模を表す Scale は 100 に設定してデータを生成した.

表 2　ソフトウェア構成

OS	CentOS release 6.3 (Final)
Java	OpenJDK 1.6.0_24 (rhel-1.57.1.11.9.el6_4-x86_64)
Hadoop	CDH 4.4.0 (MapReduce1)
Hive	0.10.0 (CDH4.4.0)
Impala	1.1.1
PostgreSQL	8.4.13

表 3　ハードウェア構成

	スレーブノード	マスタノード	クライアント
CPU	Xeon 5160 3.0GHz Dual-Core x 2	Dual-Core AMD Opteron 1222 3.0GHz	Xeon E5410 2.33GHz Quad-Core
メモリ	4GB x 8	2GB x 4	4GB
HDD	1.5TB	1TB x 2	1TB x 4

表 4　データセット

Table	part	orders	customer
Record	20 M	150 M	15 M
Size	2.28 GB	16.57 GB	2.29 GB
Size/Record	123 B	119 B	164 B

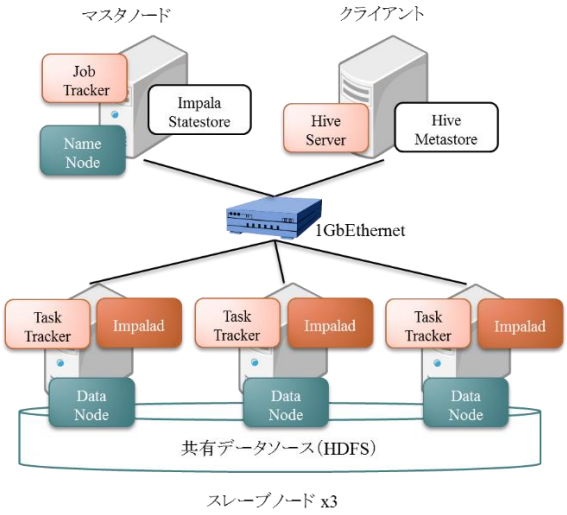

図 8　システム構成図

4.3. SQL によるランダムサンプリングの評価

4.3.1. 実験結果

Hive を利用して，3.2 節で述べた SQL によるランダム抽出を実施し，実行時間を計測した．なお，「ランダムソート」及び「ランダム値」で利用した関数は RAND()，「UDF」で利用した関数は TABLESAMPLE(n PERCENT)を利用した．実験に利用したテーブルは part である．ランダム抽出方法それぞれの実行結果を表 5 に示す．行番号を利用した方式は，Hive において行番号を取得する関数が提供されていないため，実行することができなかった．

表 5　ランダム抽出実行結果

	実行時間(s)
ランダムソート	156.783
ランダム値	24.431
Mod 計算	25.591
行番号	
UDF	22.773

4.3.2. 考察

ランダム抽出の実行結果を見ると，「ランダムソート」が最も時間がかかっており，またそれ以外の手法に関しても比較的実行時間が長い．シンプルに全件のカウント処理を実行したところ，72 秒の実行時間であったことから，ランダム抽出のコストとしては 25%を占め，ユーザクエリに対する影響度は大きいと言える．

また，「ランダム値」を利用した手法，「Mod 計算」を利用した手法は，全件に対するランダムテーブルを作成する点に優れているが，出力件数を指定すると結果がテーブルの前半に寄る傾向があり，ソートされているテーブル等ではランダム性に欠けていた．「UDF」に関しては，Hive の実装上，ブロック単位（64MB）でのランダム選択となるため，ブロック上の先頭から指定のレコード数が出力されることとなり，ランダム性に欠けていた．

環境依存の問題に関しては，データベース実装や対象テーブル毎にランダム抽出方法を変えることで回避可能ではあるが，選択できる別の方法が必ずしもコストとランダム性の両方で優れているとは限らない．

以上のことから，現状で実現可能な SQL を用いたランダム抽出では実行時に低コストでランダムに抽出することが厳しいと言える．以降の実験では，事前に各テーブルからランダム抽出して挿入した専用のテーブルを利用することで，ランダム抽出を除いた性能を評価する．

4.4. データサイズ推定コストの評価

4.4.1. Selection の実験結果

part テーブルを利用して，条件を変えることによる誤差の変化を調査した．実験に利用した SQL クエリは図 9 の Selection クエリで，[table]に part を指定した．[attribute]には part テーブルのキー属性である p_partkey を利用し，[param]の値を変化させて実行することで，SF の変化に対する結果の容量を分析する．まず part テーブルに対してランダム抽出クエリを発行し，ランダム抽出用テーブルに格納する．ここでは，10000 件のサンプルをランダム抽出した．これにより得られたデータに対する Selection クエリの結果の件数を，COUNT 関数を利用してカウントする．この件数とサンプリング数から SF を算出し，元のテーブル part のサイズとの積から処理データサイズを推定する．この処理データサイズと，正解のデータサイズとの比を誤差として評価する．

以上の実験を行った結果が**図 10** である．横軸が[param]を変化させることで得られる SF であり，縦軸が推定した処理データサイズの誤差である．折れ線グラフが 5 回実行した時の平均値であり，その上下に広がる面グラフは平均値に対する 95%信頼区間である．

次に，誤差のピークとなった SF で，サンプリング数，元テーブルサイズを変化させて実行し，誤差 2%未満となるサンプリング数と実行時間を評価する．

表 6 は SF が 0.5 となるように[param]の値を指定して実行したときのテーブル情報であり，二つのテーブ

ル part と orders に対する SQL クエリで設定する[attribute]と，その SQL クエリを実行した場合の実行時間及び結果のレコード数とデータサイズを示している．

図 11 と図 12 は， part 及び orders テーブルのレコード数に対する割合でサンプリング数を変化させた時の推定したデータサイズの誤差及び測定処理の実行時間を示している．また**図 10** と同様に 5 回実行したときの誤差の平均と 95%信頼区間をグラフにしている．

```
SELECT * FROM [table]
                WHERE [attribute] > [param];
```

図 9　Selection クエリ

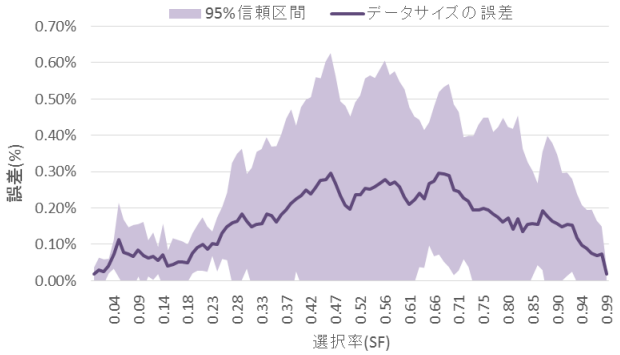

図 10　SF 変化によるデータサイズの誤差

表 6　SF=0.5 におけるテーブル情報

Table	part	orders
Attribute	p_partkey	o_orderkey
Record	10 M	75 M
Size	1.20 GB	8.69 GB
SF	0.5	0.5
Time	42.05	137.42

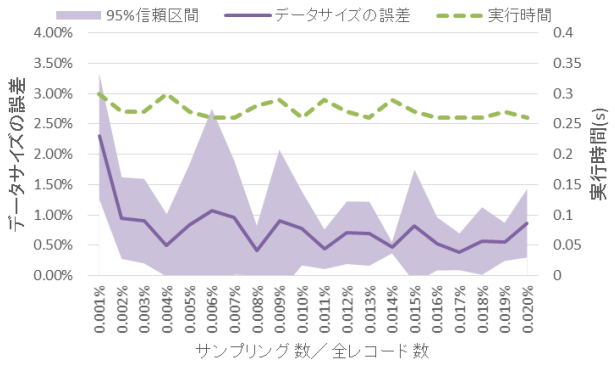

図 11　part におけるデータサイズの誤差

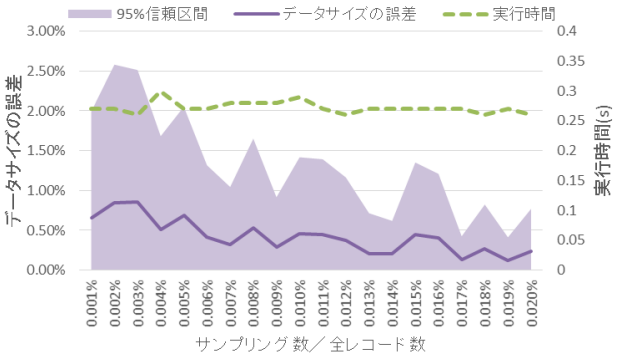

図 12　orders におけるデータサイズの誤差

4.4.2. Join の実験結果

本実験では，orders テーブルと customer テーブルの Join 処理におけるサンプリング数の変化の誤差と実行時間を評価した．表 7 はこの Join 処理をランダム抽出せずに実行した時の出力レコード数，サイズ，SF，実行時間を示している．

orders テーブルに対してランダム抽出を行ったデータを利用して SF の推定値を算出した場合は，まったく誤差なく SF を算出した．これはサンプリングレコード数を 0.001%から 0.02%まで 0.001%ずつ変化させて実行しても変わらなかった．これらの平均実行時間は 2.43 秒であり，ユーザクエリの実行時間の 1%程度であった．

一方，customer テーブルに対してランダム抽出を行った場合は，サンプリング数に影響を受ける形で誤差に変化が生じた．この結果が図 13 であり，サンプリング数を変化させた場合の誤差とその 95%信頼区間，及び実行時間を表している．

4.4.3. 考察

図 10 は，SF が 0.4 から 0.6 となる条件値を選択することでデータサイズの誤差が最大となることを示している．この実験で得られた山型のグラフは，統計的に得られた Selection クエリにおける特徴であり，テーブルサイズやサンプリング数が変化してもこの山型になる傾向は変わらないといえる．そのためサンプリング数別のデータサイズ誤差の上限を知りたい場合には，SF が 0.5 前後となる条件値を選択することで見積もることが可能であることが分かった．

サンプリング数を変化させた場合のデータサイズの誤差は，サンプリング数が増えることで小さくなる．このことから，先の実験で得られた SF=0.5 において，サンプリング数を変化させてデータサイズの誤差を調査することで，目標とする誤差の範囲内に収まるサンプリング数を見積もることが可能と言える．これによって得られた結果によると，図 11 及び図 12 の結果から，信頼区間を考慮すると part テーブルではサンプリ

表 7　Join 処理のテーブル情報

Table	orders join customer
Attribute	o_orderkey = c_orderkey
Record	150 M
Size	39.51 GB
SF	6.67E-08
Time	180.74

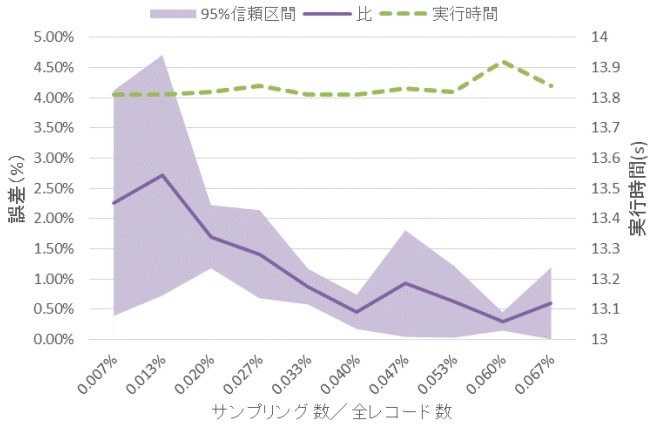

図 13　customer のランダム抽出における Join 処理の誤差と実行時間

ング数 0.009%以上（1800 レコード以上）で，orders テーブルではサンプリング数は 0.007%以上（10500 レコード以上）で安定的に 2%未満の誤差となることがわかった．

サンプリングデータに対する計測クエリの実行時間は，part 及び orders 共に約 0.25 秒であった．今回利用した実験環境の Hadoop（Hive 及び Impala）は，データを全てブロック単位で保存している．そのため，ブロックサイズ未満のデータであっても 1 ブロックとして保存される．本環境ではこれを 64MB と設定していて，part 及び orders のサンプリングデータが本実験の範囲内では全て一つのブロックに収まることにより，サンプリング数が変わっても実行時間が変わらない結果となった．この結果は，part に適用しても実際のユーザクエリの実行時間の 0.6%程度であり，規模の大きいテーブルにおけるユーザクエリへの影響は非常に小さいと言える．

Join 処理では，orders テーブルをサンプリングした場合に，低コストかつ誤差なく SF を推定できたが，customer テーブルをランダム抽出した場合は Selection 処理と同様に誤差が生じた．本実験における Join 処理の条件となる属性は，orders 上ではキー属性であり，結合先は一意に決まる．このことから，orders テーブルのランダム抽出によるデータ分布の変化が発生しなかったと言える．一方 customer 上では 1 レコードに付き複数のレコードが結合され，かつその結合するレコードの数が一様でない．そのため，ランダム抽出によって得られたレコードの結合レコード数の分布に誤差が生じ，図 13 のように推定サイズの誤差が生じたことがわかった．つまり，Join 処理において本手法を適用する場合に，レコード毎の結合数の一意性を事前に知ることができれば，正確かつ低コストに出力容量を予測することができる．

一様な結合数とならない場合，誤差を考慮した容量推定のために，サンプリング数の調整が必要となる．ここでも Selection 処理と同様に，誤差 2%未満となる場合のサンプリング数及びコストを評価する．図 13 から，この Join 処理におけるサンプリング数が 0.033%以上（5000 レコード以上）であれば，2%未満の誤差となることがわかり，その場合における実行時間は 13.8 秒となり，ユーザクエリの実行時間の 7%程度であった．実行時間がサンプリング数によらず安定している点は Selection 処理と同様に Hadoop の仕組み上の影響であり，ブロックサイズである 64MB 以上（customer では約 42 万レコード以上）のサンプリング数としない限りは実行時間に大きな差は発生しないと言える．

以上のようにしてサンプリング数を調整することで，サンプリング処理が有効と考えられた Selection 処理及び Join 処理において，あるデータサイズの上限を超えないためにデータサイズを見積もる場合においては，テーブル毎のサンプリング数で得られた推定データサイズに対して 2%のデータサイズを加える事で，確実に上限を超えているかどうかを判定することができる．更には，例えば 1%未満などのより精度を高める場合においても，同様にサンプリング数を調整することで対応でき，比較的低コストでユーザクエリ実行時に処理データサイズを推定することが可能である．

5. おわりに

本稿では，ランダム抽出を用いることで，大規模データベースに対する SQL クエリの処理データサイズを実行時にリアルタイムに計測する手法を提案し，実現手法の比較と検証実験により有効性を示した．今後は，処理の種類やテーブルのサイズに応じた，テーブル毎のサンプリングサイズの計算方法の効率化を検討したい．また，事前にランダム抽出することで余計に必要となるテーブルを排除するために，大規模データでも効率的にランダム抽出可能な SQL の実装を検討したい．

参 考 文 献

[1] M. T. Özusu and P. Valduriez, “Principles of Distributed Database Systems”, Third Edition, Springer, 2011

[2] 齋藤和広，渡辺泰之，小林亜令,“データ共有型マルチデータベースシステムにおけるクエリ効率化手法”, 情報処理学会研究報告，Vol.2013-DBS-158, No.6, 2013

[3] M. V. Mannino, P. Chu and T. Sager, “Statistical Profile Estimation in Database Systems”, ACM Computing Surveys, Vol. 20, No.3, pp. 191-221, 1988

[4] C. M. Chen and N. Roussopoulos, “Adaptive Selectivity Estimation Using Query Feedback”, Proc. 1994 ACM SIGMOD Intl. Conf. on Management of Data, pp. 161-172, 1994

[5] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan and D. Patterson, “Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning”, ICDE '09 Proc. of the 2009 IEEE Intl. Conf. on Data Engineering, pp. 592-603, 2009

[6] F. Olken, “Random Sampling from Databases”, PhD thesis, University of California, Berkeley, 1993

[7] PostgreSQL, http://www.postgresql.org/

[8] MySQL, http://www.mysql.com/

[9] Apache Hadoop, http://hadoop.apache.org/

[10] Apache Hive, http://hive.apache.org/

[11] Cloudera Impala, http://www.cloudera.com/content/cloudera/en/products/cdh/impala.html

[12] TPC-H, http://www.tpc.org/tpch/