DEIM Forum 2013 F10-4

RDBMSへアドオン可能な高CPU負荷演算の分散並列処理手法

柴田　秀哉†　　田村　孝之†

† 三菱電機株式会社 情報技術総合研究所　〒 247-8501 神奈川県鎌倉市大船 5-1-1
E-mail: †{Shibata.Hideya@cb, Tamura.Takayuki@eb}.MitsubishiElectric.co.jp

あらまし　本稿では，関係データベース管理システム（RDBMS）の機能拡張機構であるユーザ定義関数を利用して，分散並列処理による高 CPU 負荷演算のアクセラレータを，既存 RDBMS へのアドオンとして実現する手法を提案する．提案手法では，表値型のユーザ定義関数を用い，複数レコードを並列に処理することで高速化を実現する．提案手法により，RDBMS 利用者は SQL 言語を介したアクセラレータの利用が可能となる．例として，代表的な RDBMS の 1 つである PostgreSQL への実装方式を説明する．また，高 CPU 負荷演算を要するアプリケーションとして，検索可能暗号による暗号化データの検索処理を対象とした評価結果を報告する．

キーワード　分散並列処理，アドオン，ユーザ定義関数，高 CPU 負荷演算，関係データベース

A Distributed Parallel Processing Method as an RDBMS Add-on for High CPU Cost Operations

Hideya SHIBATA† and Takayuki TAMURA†

† Information Technology R&D Center, Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura, Kanagawa, 247-8501 Japan
E-mail: †{Shibata.Hideya@cb, Tamura.Takayuki@eb}.MitsubishiElectric.co.jp

Abstract　In this paper, we propose an acceleration method for database operations with high CPU cost, which is available as an add-on to existing relational database management systems (RDBMS). Our method is based on distributed parallel processing, and uses user-defined function that is expansion feature of RDBMS. In the proposed method, the acceleration results from overlap processing of multiple records with a table-valued function. Our method allows RDBMS users to access to the accelerator through SQL interface. As an example, this paper refers to an implementation for PostgreSQL that is one of the most popular RDBMSs. We also evaluate the performance of encrypted data search with searchable encryption, which is one of applications requiring high CPU cost operations.

Key words　distributed parallel processing, add-on, user-defined function, high CPU cost operation, relational database

1. はじめに

近年，セキュリティ要求の高度化やデータ形式の多様化に伴い，データ処理は複雑化する傾向にある．例えば，セキュリティの観点からは，クラウドとセキュリティを両立させる暗号技術として，データを暗号化したまま検索することが可能な検索可能暗号が注目されつつある．検索可能暗号では，暗号化データを検索するために特殊な照合を行うが，このような照合演算は多くの既存データベースが備えていない機能であるため，ユーザ定義の演算を追加するといった機能拡張が必要となる．また，データ形式の多様化という観点では，画像や音声などを扱う巨大なバイナリデータに対して特殊な処理を施したいという要求に対し，やはり機能拡張の必要性が生じる．

これらの拡張機能に伴う演算は，通常のデータベース演算と比較して，CPU 負荷が高い傾向にある．そのため，ユーザ定義の演算を追加するに当たっては，処理の高速化を併せて考慮しなければならない場合が多い．

データ処理を高速化するための最も有効な手段の 1 つとして，分散並列処理がある．近年では，Hadoop (注1) [1] に代表されるように，多数の計算機を用いた分散並列処理により，大規模データの集計・分析処理速度を向上させるためのフレームワークが提供されている．このようなスケールアウトアプローチは，大規模データを対象とするバッチ処理において効果を発揮する一方で，データ一貫性の維持が困難となるため，対話処理に向

(注1)：Hadoop は，Apache Software Foundation の登録商標である．

かないという欠点がある．

そのため，データ更新が頻繁に発生するようなシステムにおいては，豊富なトランザクション機能を備えた関係データベース管理システム（RDBMS）の重要性は依然として高い．RDBMS を採用したシステムにおいて高 CPU 負荷演算を含む処理を実行する場合には，RDB に格納されたデータを一度 Hadoop へ移行し，分散並列処理を実行する，といった運用が多く為されている．スケールアウトアプローチの利用性向上という観点からは，Hadapt [2] や SQL-MapReduce (注2) [3] のように，RDBMS 連携，及び SQL インタフェース利用を可能とするための技術，ソリューションが多く提案されている．しかしながら，データ移行コストや，データ一貫性保持といった観点からは，対話処理に適した RDBMS の枠組みで分散並列処理を行いたいという要求がある．

このような背景から，本稿では，コスト要因となるデータの分散管理を行わずに CPU 演算処理のみを並列化する，という方針の下，高 CPU 負荷演算のアクセラレータを RDBMS にアドオンする手法を提案する．提案手法では，表値型のユーザ定義関数を用い，複数レコードを並列に処理することで高速化を実現する．ユーザ定義関数は RDBMS が備える機能拡張機構であり，RDBMS 利用者はユーザ定義関数を追加するだけで，提案するアクセラレータを SQL 言語インタフェースを介して利用可能となる．

本稿の構成は以下のとおりである．まず，2 節で関連技術について述べる．3 節では，ユーザ定義関数を用いて，分散並列処理による高 CPU 負荷演算のアクセラレータを既存 RDBMS へのアドオンとして実現するための手法を提案する．4 節では，オープンソースソフトウェア（OSS）の RDBMS である PostgreSQL [4] に対する提案手法の実装について言及する．5 節では，提案手法に対する評価結果を報告する．評価においては，高 CPU 負荷演算を要するアプリケーションとして，検索可能暗号による暗号化データの検索処理を対象とした処理性能を評価する．最後に，6 節で本稿のまとめを行う．

2. 関 連 技 術

RDBMS において問合せ処理を分散並列処理させる手法として，並列問合せ（parallel query）がある．これは，複数ノードにデータを分割して配置し，それぞれのノードで並列に検索要求を実行し，処理時間を短縮する方法である．

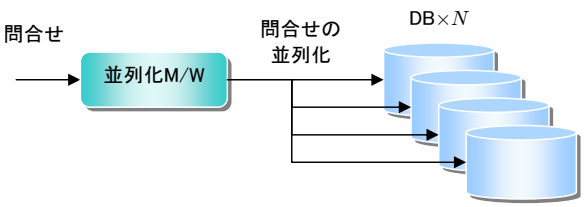


図 1　並列問合せ

並列問合せの概略を図 1 に示す．例えば，PostgreSQL では専用ミドルウェアである OSS の pgpool-II [5] や Postgres-XC [6] を利用することで，複数のデータベース間での並列問合せが実現可能となる．並列問合せの仕組みを備えていない RDBMS において，並列問合せのアプローチを採る場合には，データベースサーバとクライアント間で問合せを仲介するために，pgpool-II のような並列化ミドルウェアを用意する必要がある．

並列問合せによる分散並列化のアプローチでは，複数データベース間でのデータ整合性を保証するために，分散トランザクション処理が必要となり，データ更新処理の管理に手間が掛かるといった欠点がある．実際 pgpool-II においては，並列問合せを利用する際のトランザクション機能が大幅に制限されている [5]．また，並列問合せのアプローチでは，データを配置するノードのリソースを占有するというリソースコスト面での欠点もある．

3. 演算アクセラレータのアドオン化手法

本節では，コスト要因となるデータの分散管理を行わずに CPU 演算処理のみを並列化する，という方針の下，高 CPU 負荷演算のアクセラレータを既存 RDBMS へのアドオンとして実現する手法を提案する．

3.1 演算アクセラレータアプローチ

まず，提案手法の基本方針となる演算アクセラレータアプローチについて説明する．図 2 に概略を示す．本アプローチでは，対象とする演算を処理するためのプロセッサを動的に確保し，演算に必要なデータのみをワーカノードに配布する形式で分散並列処理を実施する．分散並列化対象とする演算は，高い CPU 負荷を伴う一部の演算のみであり，その他の処理は全てマスタノードで実行するという想定である．全てのデータをワーカノードに分散配置するわけではなく，演算対象データのみを必要に応じて配布する形式であるため，動的にリソースを割り当てることができ，待機リソースコストが削減されるという利点がある．

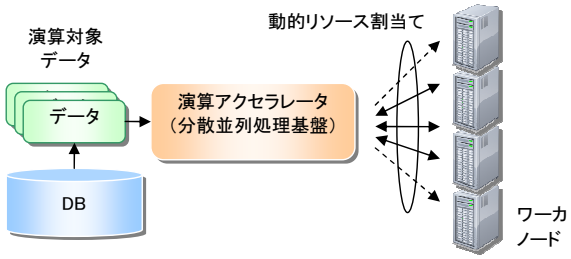


図 2　演算アクセラレータアプローチ

また本アプローチを採ることで，RDBMS の機能拡張機構であるユーザ定義関数により，演算アクセラレータを実現することが可能となる．特に，データ更新に関わるトランザクション処理を，既存 RDBMS の枠組みで全て実現できることは大きな利点である．ユーザ定義関数による演算アクセラレータの実現方法については，3.3 項で説明する．

(注2)：SQL-MapReduce は，Teradata US, Inc. における登録商標である．

3.2 演算アクセラレータ

演算アクセラレータの構成を図 3 に示す．演算アクセラレータは，マスタ・ワーカ型の分散並列処理基盤として実現される．処理対象データを保有する単一のマスタノードが，演算処理を受け持つ複数ワーカノードへのデータ分配，及び演算結果収集をレコード単位で並行して行う．これにより，高 CPU 負荷演算の高速化を実現する．

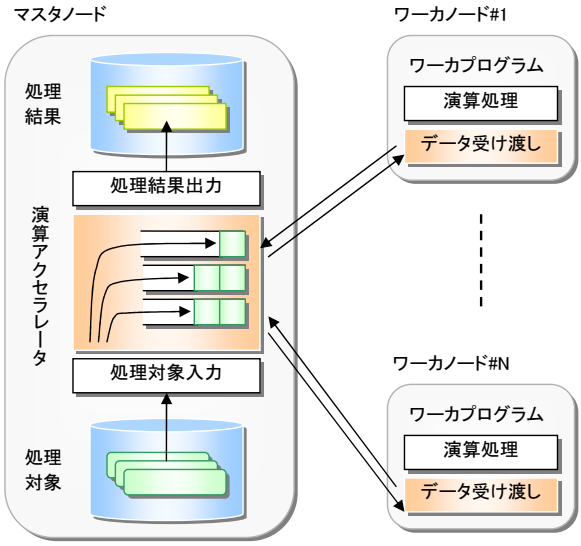


図 3　演算アクセラレータの構成

マスタ・ワーカ型の構成を採用することで，処理能力に余裕のあるワーカノードに動的に負荷を割り当てることができ，システム全体の利用率を向上させられる．また，ワーカプログラムの異常終了を検出した場合，処理結果を受信していないデータを別のワーカプログラムに送出することで，ワーカ障害耐性が得られる．

なお，このようなマスタ・ワーカ型構成のアクセラレータ利用例として，文献 [7] では，MATLAB (注3) の並列化について報告されている．本提案では，並列化対象が RDBMS 処理であり，実行インタフェースを SQL 言語で実現するという点に新規性がある．

3.3 ユーザ定義関数による演算アクセラレータの実現

本項では，前項で述べた演算アクセラレータを，RDBMS のユーザ定義関数として実現する方法について説明する．演算アクセラレータは，データベースに格納された複数レコードを対象として，ワーカノードへの分配処理を実施することで高速化を実現する．そのため，処理対象となるレコード集合を一括で演算アクセラレータに入力する必要がある．従って，演算アクセラレータの入出力は，単一レコードではなく複数レコードから成る集合となる．

以上のことから，演算アクセラレータの実体となるユーザ定義関数としては，一般的なスカラ値型の関数ではなく，レコード集合を出力することが可能な表値型の関数を採用する（図 4）．

(注3)：MATLAB は The MathWorks, Inc. の登録商標である．

本稿では，便宜上この抽象的な関数を表関数 `proc_by_set()` と表記する．

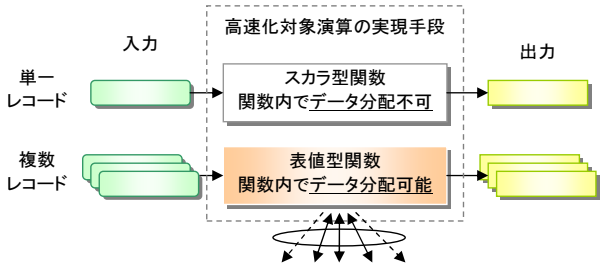


図 4　スカラ値/表値型関数それぞれによる演算の実現

表関数 `proc_by_set()` は，処理対象となるレコード集合を入力として受け取る必要があるが，この入力集合は一般に，あるSELECT 文を実行した結果集合と言い換えることできる．そこで，入力集合の受け渡し方法として，表関数 `proc_by_set()` の引数に SELECT 文 q を文字列として渡し，`proc_by_set()` の内部で SELECT 文 q を再帰的に実行する方法を採用する．

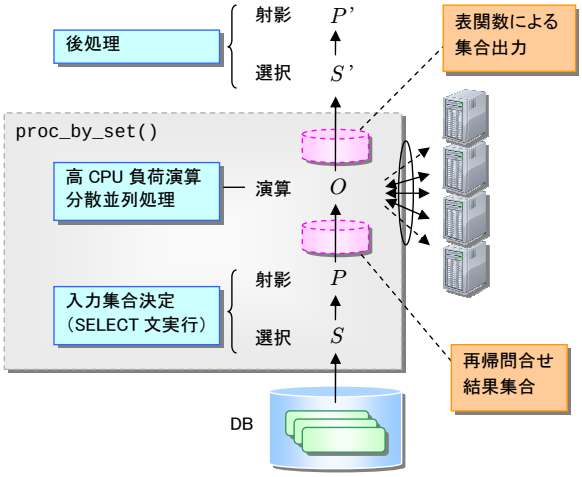


図 5　表関数 `proc_by_set()` の動作

表関数 `proc_by_set()` 及びその周辺の動作を図 5 に示す．表関数 `proc_by_set()` は，まず，引数として指定された SELECT 文 q を実行し，分散並列処理対象とするレコード集合を決定する．この際，分散並列処理対象を絞り込み，処理効率を向上させるための条件を，SELECT 文 q に付加することが可能である．次いで，SELECT 文 q の実行結果を対象とし，高 CPU 負荷演算の処理を実施する．この際，レコード単位に演算処理を並列化することで処理の高速化を実現する．なお，演算処理に必要な定数パラメータ等は，表関数 `proc_by_set()` への引数として渡される．演算処理の結果は，表関数 `proc_by_set()` の出力表における適当な列として返却される．最後に，表関数 `proc_by_set()` の実行結果表に対して，選択や射影等の後処理を実行する．これらの後処理は，表関数の外に記述される SELECT 句や WHERE 句の処理に相当する．当然ながら，後処理で使用される列は表関数 `proc_by_set()` の結果表に含ま

れなければならない．これを実現するためには，SELECT 文 q で指定する選択列に，後処理で使用する列を含めておき，演算処理結果と対応付けて結果表を生成するようにすれば良い．

演算アクセラレータの利用イメージを PostgreSQL の流儀に倣った擬似 SQL により説明する．2 項演算 op を真偽値を返す高 CPU 負荷な比較演算とし，演算 op のアクセラレータが表関数 `proc_by_set()` により実現されているとする．ここで，ある定数値 l に対して，

```
SELECT c1 FROM t WHERE c2 op l                    (1)
```

という問合せを考える．問合せ (1) は `proc_by_set()` を用いることで

```
SELECT c1 FROM proc_by_set(
  'SELECT c2, c1 FROM t', l
  ) AS r (c1 typeof(c1), c2^(r) boolean)
  WHERE c2^(r) = true                             (2)
```

のように書き換えることができる．ここで，AS 句は表関数が返す結果表のレコード型に対するキャストを意味し，“typeof” は対象列のデータ型を表現している．この例では，表関数 `proc_by_set()` は SELECT 文と 1 つの定数パラメータを引数に取り，結果表として，列 c_1 及び演算結果列である $c_2{}^{(r)}$ を返す．

3.4 RDBMS への要件

提案手法を実現するためには，対象とする RDBMS にいくつかの要件が課される．本項では，それらの要件を整理しておく．

まず，表関数 `proc_by_set()` がユーザ定義関数として適当の言語で作成できる必要がある．記述言語としては，分散並列処理に必要な通信等の機能を全て呼び出すことが可能なものでなければならない．

次いで，ユーザ定義関数の内部で SELECT 文の再帰実行が可能なインタフェースが，RDBMS 側に用意されている必要がある．更に，再帰実行された SELECT 文の結果集合に対して，演算処理結果を格納するための列を追加する等，結果集合の加工が可能であることが要件となる．

最後に，再帰実行される SELECT 文に指定された選択リストに合わせて，表関数の結果表におけるレコード型を動的に定義可能である必要がある．これは，ユーザ定義関数のコンパイル時には結果表のレコード型定義が未定であることが許可されており，SQL の実行時に，結果表のレコード型を指定可能である，という意味である．この要件が満たされない場合，再帰実行される SELECT 文毎にユーザ定義関数を用意する必要があり，現実的な手法とは言えなくなる．

4. PostgreSQL への実装

本節では，OSS RDBMS である PostgreSQL に対する提案手法の実装について言及する．PostgreSQL の詳細に関しては [4] を参照されたい．

PostgreSQL では，提案手法において必要となる表値型のユーザ定義関数を C 言語で実装することが可能である．ユーザ定義関数内部からの SELECT 文再帰実行に関しては，サーバプログラミングインタフェース（server programming interface, SPI）と呼ばれるインタフェースが用意されており，SELECT 文の再帰実行，及び結果表の受け取りを比較的容易に実装可能である．また，SPI により渡された結果表の加工も自由に行うことができるため，分散並列処理された演算結果を結果表に追加することも可能である．

表関数の結果表におけるレコード型については，問合せ (2) で例示したように，表関数を指定する FROM 句の直後に AS 句を記述することで，動的な定義が可能である．AS 句で指定した情報は，ユーザ定義関数の内部から参照することが可能であり，指定されたデータ型等の情報に併せて，結果表を生成すれば良い．関数定義の際は，CREATE FUNCTION 文において，RETURNS 句に `setof record` を指定すれば良い．

5. 評　　価

本節では，検索可能暗号に関する検索処理に提案手法を適用し，検索性能評価を実施した結果を報告する．

5.1 評 価 方 針

評価実験は，検索可能暗号に関する検索性能を対象に実施する．検索可能暗号とは，データ及び検索キーワードを共に暗号化した状態のまま，両者が平文情報として一致するか否かのみを判定可能とする，比較的新しい分野の暗号技術である．判定の際は，データとキーワードが一致するか否かのみが開示され，その他の情報が一切漏れないため，安全性の高いデータ検索が実現される．検索可能暗号は文献 [8] に端を発し，クラウドとセキュリティを両立させる暗号技術であるとして，今日までに多くの研究報告が為されている．

検索可能暗号では乱数を用いた確率的暗号を用いる．そのため，検索処理においては，単純なバイナリデータ比較による一致判定は行えず，暗号化データ復号に相当する複雑な暗号処理が要求される．確率的暗号の利用は，安全性向上には不可欠な要素であるが，一方で処理速度を低下させるという課題を併せ持つ．

そこで，本評価実験では，検索可能暗号によるデータ一致判定処理を高 CPU 負荷演算と見做し，提案手法を適用する．特に，データ件数や並列度に応じた検索性能特性に重点を置き，評価を実施する．

5.2 評 価 条 件

評価実験の実施条件を以下に記す．

- **RDBMS**

  PostgreSQL 9.1（Windows x86-64 版）を使用する．

- **表**

評価で使用する表の定義文を図 6 に示す．なお，検索可能暗号で一致判定に用いられるデータは暗号化タグと呼ばれ，元の平文に復号することはできない．従って，平文に復号可能な暗号化データを別途用意する必要がある．評価では，検索では使用されない参照専用の暗号化データ（data）と，検索可能暗号による検索の際に利用する暗号化タグ（tag）の 2 種類を用意

する．

```
CREATE TABLE tb (
  id    varchar(10)  NOT NULL,   /* 平文属性 */
  data  bytea        NOT NULL,   /* 暗号化データ */
  tag   bytea        NOT NULL    /* 暗号化タグ */
);
```

図 6　評価用の表定義文

- **問合せ文**

評価で使用する問合せ文を図 7, 8 に示す．図はそれぞれ演算アクセラレータの適用前，適用後に対応する．ここで，@keyword@は利用者が入力する暗号化された検索キーワードである．また，2 項演算 *op* は暗号化タグと検索キーワードとの一致判定処理に相当する高 CPU 負荷演算であり，表関数 proc_by_set() がそのアクセラレータである．

```
SELECT id FROM tb WHERE tag op @keyword@;
```

図 7　評価用の問合せ文（アクセラレータ適用前）

```
SELECT id FROM proc_by_set (
  'SELECT tag, id FROM tb', @keyword@
) AS result (
  id varchar(10), tag_res boolean
) WHERE tag_res = TRUE;
```

図 8　評価用の問合せ文（アクセラレータ適用後）

- **暗号化データサイズ**

評価で使用する各種暗号化データのサイズを表 1 に示す．

表 1　暗号化データサイズ

暗号化データ（data）	440 byte
暗号化タグ（tag）	652 byte
暗号化検索キーワード	1,028 byte

- **評価環境**

評価で使用するマスタノード，ワーカノードの詳細をそれぞれ表 2, 3 に示す．マスタノードは PostgreSQL サーバでもある．また，ワーカノードは最大で 9 台使用するが，いずれも同一機種のマシンである．

5.3　評 価 結 果

レコード件数を 10 万件，及び 1.6 万件とした場合のそれぞれの検索時間を図 9 に示す．横軸は分散並列処理における並列度数，すなわち利用コア数を表している．また，主要な並列度数に対する検索時間を表 4 に示す．但し，本評価では，1 並列時の実測はしていないため，4 並列時の実測値を単純に 4 倍した値を 1 並列時の値としている．

図 9 より，並列度数の増加に伴い検索時間が短縮されていることが分かる．表 4 からは，レコード 10 万件の検索を 100 並列時に 14.2 秒で完了し，アクセラレータ適用前（1 並列相当）と比較し約 80（= 1133/14.2）倍の高速化が為されている．一方，レコード 1.6 万件に対しては，アクセラレータ適用前と比較し，約 38（= 186/4.9）倍の高速化に留まる．また図 9 より，並列度数がある程度増加すると，検索時間短縮の度合いが小さくなる．これらは，データ件数が少ないとき，或いは並列度数が大きい場合に，並列化のためのオーバヘッドが顕在化し，性能が飽和していることを表している．

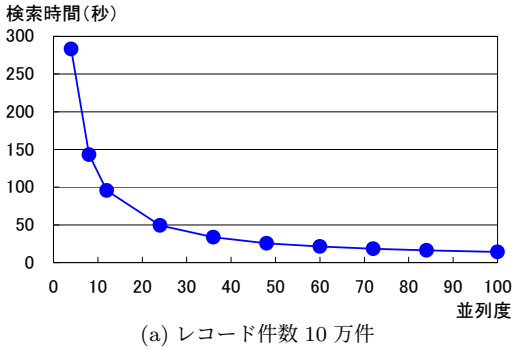
このことは，図 10 から容易に確かめることができる．図 10 は，並列度数に対する検索処理速度を表しており，凡例「理想値」で表される点線は，4 並列時の 10 万件データに対する検索処理速度を外挿した比例直線である．図 10 より，並列度数の増加に伴い処理速度が「理想値」から乖離していくことが分かる．更に，レコード件数が 10 万件の場合と比較し，1.6 万件の

表 2　マスタノード（PostgreSQL サーバ）

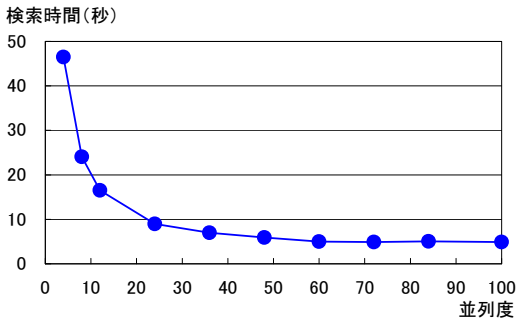
OS	Windows Server 2008R2（x64）
CPU	Xeon X5650 2.66GHz（6 コア）×2, Hyper-Threading 無効
メモリ	4GB
HDD	2TB 7.2krpm 3.5inch SATA

表 3　ワーカノード（最大 9 台）

OS	Windows Server 2008R2（x64）
CPU	Xeon X5650 2.66GHz（6 コア）×2, Hyper-Threading 無効
メモリ	4GB
HDD	1TB 7.2krpm 3.5inch SATA



(a) レコード件数 10 万件



(b) レコード件数 1.6 万件

図 9　並列度数に対する検索時間

表 4　検 索 時 間

並列度	レコード 10 万件	レコード 1.6 万件
100	14.2 秒	4.9 秒
4	283.2 秒	46.5 秒
1	（1133 秒）	（186 秒）

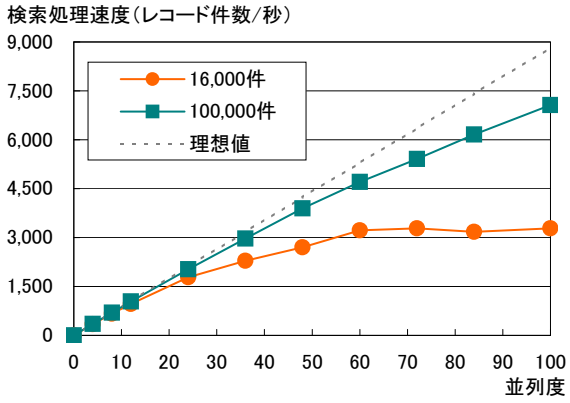


図 10　並列度数に対する検索処理速度

場合の方が乖離の度合いが大きい．

5.4　オーバヘッド解析

検索性能が並列度数に応じて線型に向上しない要因として，並列化に伴う各種オーバヘッドの顕在化，及び並列化できないデータベース処理の顕在化が挙げられる．並列化に伴うオーバヘッドには，大きく通信オーバヘッドと表関数利用に伴うオーバヘッドの 2 種類がある．また，並列化できないデータベース処理には，PostgreSQL による前処理と後処理の 2 種類がある．表 5 にそれぞれの詳細を示す．

表 5　性能飽和要因

並列化に伴うオーバヘッド	
通信オーバヘッド	ワーカプロセス起動，ネットワーク転送
表関数のオーバヘッド	結果表の生成・出力

並列化できないデータベース処理	
PostgreSQL 前処理	SELECT 文の再帰実行
PostgreSQL 後処理	表関数結果表に対する選択・射影処理

表 6 にそれぞれの処理の内訳を示す．ここでは，100 並列時の実測値を記載しているが，通信オーバヘッド以外の処理時間は並列度数には依存しない．なお，本表には並列化できないデータベース処理として「PostgreSQL 後処理」が本来含まれるべきであるが，実際には,「表関数のオーバヘッド」との切り分けが困難であり，実測は事実上不可能であるため，ここでは「表関数のオーバヘッド」に「PostgreSQL 後処理」の時間も含んでいる．

表 6　性能飽和要因の処理時間内訳（100 並列時）

処理	10 万件の処理時間	1.6 万件の処理時間
通信オーバヘッド	2.1 秒	2.9 秒
表関数のオーバヘッド	0.25 秒	0.09 秒
PostgreSQL 前処理	0.19 秒	0.03 秒
オーバヘッド合計	2.6 秒	3.0 秒
検索時間合計	14.2 秒	4.9 秒

表 6 より，並列化に伴う通信オーバヘッドが主な飽和要因となっていることが分かる．検索時間に占める割合は，データ 10 万件の場合に 15%（= 2.1/14.2），データ 1.6 万件の場合には 59%（= 2.9/4.9）にも上り，無視できない時間となっている．通信オーバヘッド発生の要因としては，プロセス起動処理やネットワーク転送処理など不可避なものが含まれるため，今後，改善の余地があるかを含めて検討する必要がある．

6.　おわりに

本稿では，RDBMS の機能拡張機構であるユーザ定義関数を利用して，分散並列処理による高 CPU 負荷演算のアクセラレータを，既存 RDBMS へのアドオンとして実現する手法を提案した．提案手法を用いることで，ユーザ定義関数を追加するだけで，RDBMS 利用者は SQL インタフェースにより，容易にアクセラレータ利用が可能となる．また，提案手法を PostgreSQL への，及び検索可能暗号による暗号化データの検索性能を対象とした評価実験により，提案手法が有効に機能することを実証した．

しかしながら，提案手法にはまだ課題が残されている．今回の実装では，並列度が高い場合やデータ件数が少ない場合に，オーバヘッドが顕著であった．今後，更なるオーバヘッド解析を進め，性能の改善に取り組んでいく予定である．

また，提案手法を PostgreSQL 以外の RDBMS へ実装することも今後の課題である．

文　献

[1] Apache Hadoop, http://hadoop.apache.org/.
[2] Hadapt, http://hadapt.com/.
[3] Teradata Aster, http://www.asterdata.com/.
[4] PostgreSQL, http://www.postgresql.org/.
[5] pgpool-II, http://pgpool.projects.pgfoundry.org/pgpool-II/doc/pgpool-ja.html.
[6] Postgres-XC, http://postgres-xc.sourceforge.net/.
[7] 牟田英正, 阪本正治, 井上忠宣, 佐藤喬, 多田好克, “MatalabParallelizer：MATALAB の Master-Worker 並列化”, 電子情報通信学会論文誌 D, Vol.93-D, No.10, pp.2186-2196, 2010.
[8] D. X. Song, D. Wagner and A. Perrig, “Practical Techniques for Searches on Encrypted Data”, Proc. 2000 IEEE Symposium on Research in Security and Privacy, pp.44-55, 2000.