DBSJ Japanese Journal, Vol.13-J, No.2
February 2015

SSDを用いた大規模データベースにおける複数問い合わせ処理高速化手法とその評価

Optimization for Multiple Large Data Queries Processing on SSDs and Evaluation

鈴木 恵介♡ 早水 悠登♢ 横山 大作♠
中野 美由紀♣ 喜連川 優♢♢

Keisuke SUZUKI Yuto HAYAMIZU
Daisaku YOKOYAMA Miyuki NAKANO
Masaru KITSUREGAWA

SSDは、高い転送レートを持つストレージデバイスとして注目されており、大規模データ処理を行うデータセンターで導入が進んでいる。**SSD**のアクセス特性としては、**HDD**の**100**倍から**1000**倍高速なランダム**I/O**や、内部のフラッシュチップの並列構造による**I/O**の並列処理能力などが挙げられる。これらを生かす方法として、本稿では、まずデータベースシステムのハッシュ結合演算において、データベースバッファ使用量を小さくしランダム**I/O**が増加しても、**I/O**の処理コストが増加しないことを実測によって確認する。これにより、さらにキャッシュ使用効率が高まり、全体の処理性能が向上することを示す。また、メモリやキャッシュの使用量を抑えることで、複数の演算でこれらを分割して使用することが可能になる。これを利用し、並列問い合わせ処理の制御を行うことで処理性能向上を実現する方法を探求する。

Solid-state drives (SSDs) are widely used in large data processing applications due to their higher random access throughput than HDDs and capability of parallel I/O processing. In this study, we analyzed the behavior of SSD-based databases and optimized the performace of large data queries. We found that cache misses of the hash join operation on SSD-based databases can be decreased without increasing the I/O costs by reducing the hash table size to fit into the cache. Moreover, by reducing the hash table size, the cache can be utilized on multiple processes. We clarify that we can decrease the cache misses of processing multiple queries by considering the appropriate hash table size to fit into the cache and the total performance of queries are improved.

♡ 正会員 富士通株式会社 keisuke@tkl.iis.u-tokyo.ac.jp
♢ 学生会員 東京大学生産技術研究所 hayamiz@tkl.iis.u-tokyo.ac.jp
♠ 正会員 東京大学生産技術研究所 yokoyama@tkl.iis.u-tokyo.ac.jp
♣ 正会員 芝浦工業大学 miyuki@shibaura-it.ac.jp
♢♢ 正会員 東京大学生産技術研究所、国立情報学研究所 kitsure@tkl.iis.u-tokyo.ac.jp

1. はじめに

現在の計算機では、大規模データ処理性能はストレージデバイスの転送レートに律速される場合が多い。従来の磁気ディスク(HDD)のスループットは、CPUの高速化と比較すると進展が少なく、より高速、広帯域なストレージデバイスが求められている。NAND flash-based Solid State Drive (SSD)は、NAND flash chipの集積技術の進歩により、大容量化、低価格化が進み、データセンターなど大規模データ処理が必要な現場において導入が増加しており、実例としてGoogle[7]やFacebook [14]などでの使用も見られる。

本稿では、大規模データ処理アプリケーションの典型例である関係データベースシステムにおけるSSDの使用について論じる。SSDは、HDDと同様のインターフェースで扱うことが可能なので、HDDを置き換えることで使用されることが多い。しかし、SSDの内部アーキテクチャは、HDDのそれとは大きく異なるため、アクセス特性の違いが大きい。現在の多くの関係データベースシステムは、HDDの特性に基づいて設計されており、SSDを用いる利点を十分に活かしていない。そこで本稿では、従来の関係データベースシステムの処理方式の見直しを行い、SSDのアクセス特性を活用するための処理方式について検討する。

SSDのI/O特性としては、高いランダムアクセススループットや並列I/O処理能力が挙げられる。まずSSDは、HDDのように機械的な動作部品を含まず、電子回路のみで構成されるデバイスであるので、ランダムアクセス時のデータのシークの待ち時間がなく、高速な処理が可能である。HDDと比較すると、100倍から1000倍のランダムアクセススループットを持つ。また、SSDは内部に複数のフラッシュチップを搭載しており、並列I/O処理によってスループットを増加させることができる。問い合わせ処理においてこれらの特性を考慮する必要がある。

本稿ではまず、従来の関係データベースシステムの実装方式でSSDを用いる場合の問題点を明らかにするため、ハッシュ結合演算を対象として、実際のSSDを使用した関係データベースシステムにおいて、演算処理性能を詳細に解析した。その結果以下の事実を解明した。

- ハッシュ結合演算では、データベースバッファ使用量が小さいほど、中間生成データファイルがフラグメント化し、ランダムI/Oが増える。SSDを用いる場合は、HDDを用いる場合と異なり、ランダムI/Oの処理コストが小さいためデータベースバッファ使用量を縮小しても、I/Oコストを小さく保つことができる。
- SSDを用いる場合は、I/Oコストが小さくなり、CPUコスト最適化の重要性が増す。ハッシュ結合演算では、ハッシュテーブルを小さくすることでデータアクセス局所性を高め、キャッシュミス数の増加を抑えることができる。そのため、データベースバッファ使用量を縮小し、ハッシュテーブルを小さくするのがよい。

データベースバッファ使用量を小さく設定することで、キャッシュに空きスペースが生じるため、これを他の処理で利用できる。SSDは並列I/O処理が可能であるので、問い合わせの同時処理によりI/Oスループット向上が可能である。これらの理由から、SSDを用い複数問い合わせ同時処理を行うことで全体の処理速度を向上できる。同時処理による効果を検証するため、実際の関係データベースシステムにおいて、ハッシュ結合演算の同時処理について試験的に処理性能の計測・分析を行い、以下の事実を明らかにした。

- 全ての問い合わせに含まれるハッシュ結合演算のハッシュテーブルがキャッシュに収まるようデータベースバッファ使用量を設定することで、キャッシュミス数の増加を抑えつつ同時処理数を増加できる。そのため、同時処理数増加に伴ってキャッシュミス増加のオーバーヘッドが生じる従来のHDDに基づいた処理方式よりも、同時処理による性能向上が大きい。

- 同時処理数を過度に増加してしまうと、I/O フラグメンテーションにより I/O スループットが低下してしまう。この現象の発生を防ぐため、適切な同時処理数を選択するための問い合わせスケジューリング機構が必要となる。

本稿の構成は以下の通りである。2 節では、本研究の計測で用いるハッシュを用いた結合演算の処理方法とその負荷の性質について述べる。3 節では、本稿の計測で使用した計算機環境と計測の方法について示す。4 節では、まず SSD の基礎性能について触れ、次に SSD 上のハッシュ結合における処理性能計測結果を示す。5 節では、ハッシュ結合を行う複数の問い合わせの同時処理性能を計測し、同時処理の制御について考察する。最後に、6 節で、SSD を用いたデータベースシステムについての関連研究について述べる。

2. ハッシュを用いた結合演算

意思決定支援システムなどに代表される、大規模データ処理を行うデータベースシステムでは、集約や射影、結合といったデータベース演算において、ハッシュテーブルを利用したアルゴリズムが用いられることが多い。本稿では、SSD を使用したデータベースにおいて、ハッシュ結合演算の処理性能を計測対象とする。ハッシュ結合演算を用いるのは、データベース演算の中でも最も処理の負荷の高いものの一つであるためである。SSD の高い I/O スループットによる、計算やメモリアクセスにかかる CPU コストや、I/O コストの変化を確認し、全体の処理性能が向上することを明らかにする。

2.1 Grace ハッシュ結合 [10]

説明の便宜のため、リレーション R と S の結合演算で、S に対してハッシュテーブルが作成されると仮定する。ハッシュテーブルサイズが使用可能メモリサイズを越えるとき、Grace ハッシュ結合では、R、S それぞれのデータを分割し、パーティション毎に結合処理が行われる。分割数は S の各パーティションに対するハッシュテーブルがメモリサイズを越えないように決定される。

Grace ハッシュ結合は、ビルドフェーズとプローブフェーズの 2 つのフェーズで処理される。まず、ビルドフェーズでは R と S それぞれのデータを結合のキーについて同じハッシュ関数で分割し、ストレージに書き込む。次に、プローブフェーズでは、R と S のパーティションで同じハッシュ値を持つもの (R_i、S_i とする) をストレージから読み込み、S_i についてハッシュテーブルを作成する。続けて R_i の各タプルについてハッシュテーブルを利用して結合条件のマッチングを行う。これを全てのパーティションに対して繰り返して処理が完了する。

2.2 ハイブリッドハッシュ結合 [15]

Grace ハッシュ結合のビルドフェーズでは、必要となるメモリはそれぞれのパーティションのストレージへの書き込みバッファの分のみである。ハイブリッドハッシュ結合では、残りのメモリを S の 1 つ目のパーティション S_1 のハッシュテーブルを保持するために利用する。メモリに保持される S_1 のパーティションと、それと同じハッシュ値をもつ R_1 のパーティションは、ストレージに書き込まれることなく処理される。これにより、結合処理の I/O コストを削減できる。

2.3 ハッシュ結合の処理コスト

HDD の使用を前提としたデータベースシステムでは、大規模データを扱う問い合わせ処理において、大量の I/O の発生による性能低下を防ぐため、Grace ハッシュ結合かハイブリッドハッシュ結合を用いることが一般的である。本稿では、以下ハイブリッドハッシュ結合を扱うものとする。

ハッシュ結合の I/O パターンは、ハッシュテーブルを保持するためのデータベースバッファの使用量によって決まる。これは、各リレーションのパーティションの数とサイズが、S の全てのパーティションのハッシュテーブルがデータベースバッファに収まるように決定されるためである。そのため、データベースバッファが小さいときは、多数の小さなパーティションファイルが作られることになり、フラグメンテーションが発生する。フラグメント化したファイル群へのアクセスは、大量のランダム I/O を伴うため、ランダム I/O がシーケンシャル I/O と比較して 100-1000 倍程度低速な HDD では、I/O スループットが大きく減少してしまう。こうした理由から、HDD 上でのハッシュ結合では、大きなデータベースバッファが必要とされる。一方で SSD においては、ランダム I/O はシーケンシャル I/O とほぼ遜色ない速度であるため、フラグメンテーションによる I/O スループットへの影響が小さい。よって SSD 上のハッシュ結合では、データベースバッファサイズを HDD 使用時より小さくできる。

表 1: 計測環境

CPU	Xeon X7560 @ 2.27GHz x 4
L3 Cache	24MB /CPU
DRAM	64GB
Storage (SSD)	ioDrive Duo x4 (8 Logical units, Software RAID0)
Storage (HDD)	SEAGATE ST3146807FC x12 (Software RAID0)
kernel	linux-2.6.32-220
File system	ext4
DBMS	PostgreSQL 9.2.4
Shared buffer	8GB

プローブフェーズにおいて構築されるハッシュテーブルは、リレーション S のタプルとマッチングされる際に、繰り返しアクセスされる。これは、ハッシュテーブルのデータアクセスに関して局所性が生じていることを意味する。つまり、データベースバッファのサイズは、プローブフェーズのキャッシュミス数にも関係しているということである。ハッシュテーブルサイズがキャッシュサイズより小さければ、ハッシュテーブル構築後のハッシュテーブルアクセスではキャッシュミスが発生しない。ハッシュテーブルサイズはデータベースバッファサイズの範囲に制限されているので、データベースバッファサイズをキャッシュサイズ以下に定めることで、キャッシュミス数を低減することができる。SSD を使用したハッシュ結合では、I/O スループットを低下することなくデータベースバッファサイズを縮小できるので、キャッシュミス数減少による性能向上が期待される。

3. 計測の環境と方法

表 1 に本稿で行った計測に使用した計算機環境を示す。ストレージのインターフェイスは、SSD は PCI-express、HDD は fibre channel である。それぞれのストレージの I/O スケジューラには、noop を適用した。またそれぞれのデバイスは、ソフトウェア RAID0(チャンクサイズ = 64kB) を構築し、ext4 ファイルシステムを適用している。

ハッシュ結合の処理性能はパーティション数とハッシュテーブルサイズに依存する。データベースバッファが小さいとき、パーティションファイル数が多くなり、ハッシュテーブルサイズは小さくなる。HDD を用いたデータベースでは、パーティションファイル数が増えフラグメンテーションにより I/O スループットが低下するのを避けるため、データベースバッファサイズを大きくとることが多い。データベースバッファサイズを介してトレードオフが生じているため、これをワークロードを変化させるためのパラメタとして使用し、それぞれの値について処理性能を計測する。`work_mem` は PostgreSQL[2] のパラメタであり、各データベース演算当たりの使用可能データベースバッファサイズを示す。PostgreSQL はハイブリッドハッシュ結合を使用しているため、全体のハッシュテーブルサイズが `work_mem` より小さいとき、パーティションはストレージに書き込まれない。

問い合わせ実行時の、CPU 使用率の内訳については mpstat(1)、I/O スループットは iostat(1)、L3 キャッシュ参照・ミス回数は linux

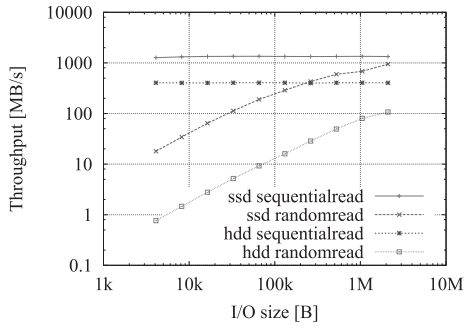


図 1: SSD と HDD におけるシーケンシャル/ランダム I/O スループット

```
1  SELECT count(*) FROM lineitem, part
2  WHERE l_partkey = p_partkey
```

図 2: lineitem 表と part 表の結合演算を行う問い合わせ

プロファイラ perf をそれぞれ使用して取得した。

4. SSD 上でのハッシュ結合の性能

本節では、まず SSD の基礎的な I/O スループットを実測によって確認する。次に、結合演算を行う問い合せの処理時間を計測し、SSD 上でのハッシュ結合では、HDD 上の場合と比較して、I/O コストを小さく保ったまま、データベースバッファを小さくできることを示す。さらに、データベースバッファを小さくすることで、ハッシュテーブルサイズが小さくなり、アクセス時のキャッシュミス数が抑えられ、処理性能を向上することを示す。

4.1 SSD の基礎 I/O 性能

SSD の基礎 I/O 性能を計測するため、マイクロベンチマークを実行する。ベンチマークプログラムは、単一のファイルに対して、指定された I/O サイズで、シーケンシャル I/O またはランダム I/O を繰り返し行うものである。

図 1 は、SSD と HDD における、各 I/O サイズでのシーケンシャル I/O とランダム I/O スループットを示す。SSD のシーケンシャル読み込み I/O スループットは、HDD より 3.1 倍程度大きく、ランダム読み込み I/O スループットは、8.6 - 23.7 倍大きい。I/O サイズが小さいほど、SSD と HDD のランダム I/O スループットの差は大きくなっている。シーケンシャル I/O スループットが I/O サイズによらず一定であるのは、ファイルの先読みが機能しているためである。

4.2 単一の結合演算を行う問い合わせによる計測

SSD と HDD それぞれのデータベースに対して、単一の結合演算を含む問い合わせについて計測を行う。データベースの構築には、TPC-H ベンチマーク [3] のデータを Scale Factor = 100 で用い、SSD と HDD 上に同一のデータベースを用意する。データベースの合計サイズは 112GB である。問い合わせは、図 2 を用いる。lineitem 表のそれぞれのタプルについて、part 表の part_key が一致する 1 タプルが結合される。part 表、lineitem 表それぞれのサイズは 20GB と 80GB であり、ハッシュテーブルは part 表に対して作られ、合計サイズは約 800MB である。work_mem を、64kB - 2GB の範囲で動かし、それぞれの値について、実行時間とその内訳を計測する。

図 3、4 に SSD と HDD それぞれにおける、各 work_mem のハッシュ結合の実行時間とその内訳 (usr, system, iowait, irq, soft irq, idle) を示す。図中では、usr が CPU コストである。sys は主にカーネルでの I/O 発行処理時間を示し、iowait はストレージ

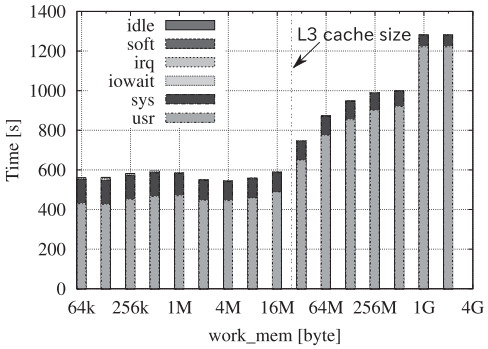


図 3: 単一の結合演算を行う問い合わせの各 work_mem の値における実行時間 (SSD)

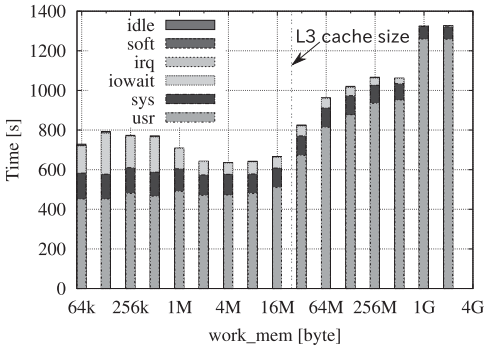


図 4: 単一の結合演算を行う問い合わせの各 work_mem の値における実行時間 (HDD)

デバイスからの I/O の結果待ち時間を示すので、sys と iowait の合計が I/O コストを表す。

work_mem = $64kB - 16MB$ の L3 キャッシュサイズより小さな範囲では、SSD と HDD の結果に異なる傾向が見える。HDD では、work_mem が小さいほど I/O コストが増加している。これは、パーティションファイルのフラグメンテーションにより、I/O スループットが飽和していることが原因である。一方、SSD では I/O コストは総じて HDD より小さく、どの点においてもほぼ変化が無い。これは、フラグメンテーションが生じても I/O 帯域が十分に残されているためである。

work_mem > $32MB$ の L3 キャッシュサイズを越える範囲では、ストレージの種類に関わらず、work_mem の増加に伴い、CPU コストが増加している。これは、work_mem が大きいほど、ハッシュテーブルの L3 キャッシュに収まる部分の割合が小さくなり、キャッシュミス数が増加しているためである。

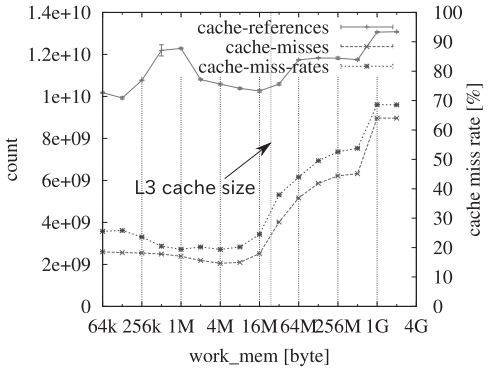


図 5: 各 work_mem の値における L3 キャッシュ参照・ミス数及びミス率 (SSD)

図 5 は、SSD 上の計測でそれぞれの work_mem の値における、L3 キャッシュの参照・ミス数とミス率を示す。work_mem > 32*MB* 以降でキャッシュミス回数が増加している。work_mem の値の違いによる CPU コストの差について、例として work_mem = 4MB と 1GB で比較する。キャッシュミス数は work_mem = 1GB の方が 7×10^9 回多く、本実験環境では DRAM のアクセス遅延が 100ns 程度であるので、work_mem = 1GB の方が $7\times10^9\times100$(ns) = 700(s) 程度 I/O コストが大きくなる。これは図 3 の CPU コストの差とおおよそ同程度の差になっている。

work_mem がハッシュテーブルサイズ (約 800MB) を越える点では、常にパーティションが 1 つに収まるため、実行時間は一定である。なお、work_mem が 512MB から 1GB の部分で急激にキャッシュミス数が増加しているが、これはハッシュテーブル中のバケツに格納されている平均タプル数が、work_mem = 1GB のとき大きく増加しており、プローブフェーズにおけるハッシュバケツのタプルスキャン数が増加したためである。ハッシュバケツ中の平均タプル数は、work_mem が 64kB のとき 2.2、128kB のとき 3.8、256kB - 512MB のとき 5.7、1GB 以上のとき 10.5 であった。lineitem 表のタプル数が 6×10^8 であるので、1GB のときのハッシュバケツのタプルスキャン数は 512MB のときより $(10.5-5.7)\times6\times10^8\approx3\times10^9$ 多くなり、図 5 のキャッシュミス数の増分と一致する。これは、PostgreSQL のハッシュ結合の実装上の問題である。

4.3 SSD に適したハッシュ結合演算処理方式

図 4 の結果では、データベースバッファサイズが小さいときは I/O コストが大きく増加している。これは、節で述べたように、HDD 上ではフラグメンテーションによる I/O スループットの低下を避けるため、比較的大きなデータベースバッファサイズを選ぶ必要があることを示している。

一方、図 3 の計測結果から、SSD を用いたハッシュ結合演算では、I/O コストが減少し、全体の処理コストの中では CPU コストが支配的であることが明らかになった。CPU コストが増加する主要な原因は、プローブフェーズにおけるハッシュテーブルアクセスの際に生じるキャッシュミスペナルティによるメモリアクセスであった。ハッシュ結合演算では、プローブフェーズにおいて、ハッシュテーブルを繰り返し参照するデータアクセス局所性があり、ハッシュテーブルサイズを小さくすることでデータアクセス局所性を高めることが可能である。SSD を用いたハッシュ結合演算処理では、I/O スループットが HDD 使用時より高いため、データベースバッファサイズを小さくしても I/O コストが小さい。このことを利用し、ハッシュテーブルが L3 キャッシュに収まるようにデータベースバッファサイズを設定することで、キャッシュミス数の増加を抑制することができる。このように、SSD を用いたハッシュ結合演算では、CPU コストを最適化する重要性が高いため、キャッシュサイズを意識してデータベースバッファを管理するべきである。

図 3 の結果では、データベースバッファサイズを 64kB まで縮小しても I/O コストは小さいままである。データベースバッファサイズを小さく設定することで、キャッシュに空きスペースが生じ、これを他の処理に用いることができるようになると考えられる。

5. 複数問い合わせの同時処理

関係データベースシステムにおいて、複数問い合わせの同時処理を行うことによって、計算資源の使用効率を高め、処理性能を向上することが可能である。また、SSD の内部は複数の Flash チップを束ねた構造になっているため、並列 I/O 処理によりスループットが増加する [6] という特性がある。そのため、SSD の I/O 帯域を十分に活用するためには、同時処理が不可欠である。複数問い合わせの同時処理では、ストレージ、メインメモリ、CPU 毎の共有キャッシュ等のリソースが共有される。処理性能を向上させるためには、個々の問い合わせで使用するリソースの量をコントロールし、それぞれの干渉を最小限に留める工夫が必要である。

HDD を用いた場合は、I/O のフラグメンテーションを避けるために、比較的大きなデータベースサイズを指定する必要があった。単一の問い合わせ処理でキャッシュがほとんど埋まってしまうため、他の処理を同時に実行すると、キャッシュアクセスの競合が発生してしまい、逐次処理時にはキャッシュヒットしていたものまでミスしてしまうというオーバーヘッドが生じてしまう欠点があった。

一方、SSD 上では、前節の計測結果の通り、ハッシュ結合演算において I/O コストを増加させることなく、データベースバッファサイズを小さく設定することができる。これにより、1 つのハッシュ結合演算で使用するキャッシュ・メモリサイズが小さくなるため、残りの領域を他の処理で使用することが可能になる。そのため、キャッシュミス数を増加させることなく同時処理数を増やすことができ、HDD を用いた場合の処理方式より同時処理による性能向上が大きくなることが期待される。

本節では、複数問い合わせ同時処理のワークロードの例として、ハッシュ結合演算を行う問い合わせの複数同時処理性能を計測し、処理性能の向上について検討する。

5.1 複数問い合わせ同時処理性能の計測

表 1 の環境において、図 2 の問い合わせの複数同時処理性能を計測する。問い合わせの同時処理数を、1 - 64 で変化させ、全ての問い合わせが終了するまでの時間とその間の CPU 使用率の内訳を計測する。データベースは、TPC-H ベンチマークデータの Scale Factor = 10 のものを同時処理数分用意し、個々のデータベースにおいてそれぞれ 1 つの問い合わせを実行する。なお、個々のデータベースの合計サイズは 13GB、part 表、lineitem 表のサイズはそれぞれ 320MB、8.8GB、part 表に対して作られるハッシュテーブルの合計サイズは約 80MB であった。計測を行った環境では、CPU 当たりのコア数が 8 であり、同時処理数 1-8 では 1CPU 内で同時処理数分のコア、16 では 2CPU に対してそれぞれアフィニティを付与して実行し、32、64 では全ての CPU を使用して実行した。また、単一の問い合わせの計測の際と同様に work_mem を 64kB - 512MB の範囲で変化させて計測する。

図 6 は、問い合わせ同時処理数毎の各 work_mem の値における実行時間とその内訳を示している。図では、各 work_mem の値について、それぞれ左から同時処理数 1、2、4、8、16、32、64 のときの結果を表している。図中、同時処理数が大きな場合に idle の割合が大きくなっているが、これはある問い合わせの処理における I/O の発行が、他の処理の I/O によって妨げられている場合であるため、この部分は I/O コストに含まれる。

5.1.1 CPU コストの増加傾向の変化

図 5 の単一の問い合わせの計測では、work_mem の値が L3 キャッシュサイズを越える 32MB から CPU コストが増加しているが、図 6 においては、同時処理数 2 では 16MB、4 では 8MB、8 では 4MB から増加している。これは、同時処理数分ハッシュテーブルが共存し、L3 キャッシュを共有しているためである。全てのハッシュテーブルが L3 キャッシュに収まる場合は、キャッシュミス数が抑えられる。なお、計測を行った環境では、CPU 当たりのコア数が 8 であるため、8 コア毎に L3 キャッシュが共有されている。よって同時処理数 16 以上では、キャッシュミス数の増加の傾向は同時処理数 8 のときと同様である。

また、計測環境の全コア数は 32 であるので、同時処理数 64 のときは、1 コア当たり 2 つの問い合わせを処理する必要がある。そのため、CPU コストは全ての work_mem の値について 32 のときの約 2 倍になっている。

図 7 は、全てのハッシュテーブルがキャッシュに収まるようにデータベースバッファサイズを設定した場合 (work_mem = 256 kB) と、HDD を用いた場合の処理方式として、データベースバッファサイズを大きく設定した場合の (work_mem = 256 MB) の

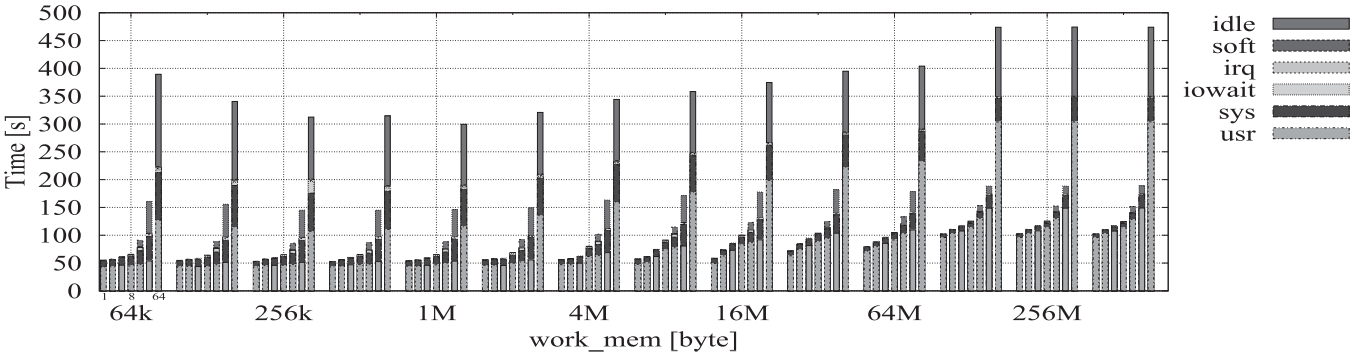


図 6: 複数問い合わせ同時処理時の各 work_mem の値における実行時間

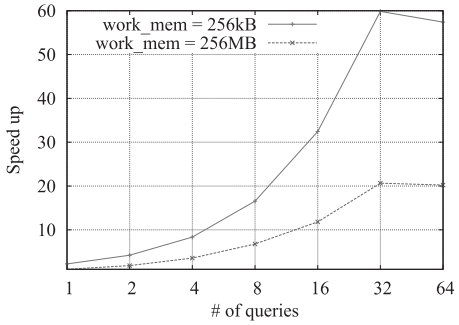


図 7: 各問い合わせ同時処理数における CPU 処理の高速化 (work_mem = 256 kB, 256 MB)

CPU 処理の高速化を表す。図では、work_mem = 256 MB の逐次処理時 (同時処理数 1) の CPU 処理速度を 1 としたときの速度向上を示している。

逐次処理時の CPU 処理速度は、work_mem = 256 kB の場合では、work_mem = 256 MB の場合の 2.2 倍となっている。同時処理による速度向上は、同時処理数 32 のときで、work_mem = 256 kB の場合で 59.8、work_mem = 256 MB の場合で 20.6 であった。

work_mem = 256 kB の場合は、キャッシュを分割して使用し、キャッシュ参照の競合を抑えることができるため、同時処理数を増加してもキャッシュミス数が増加せず、CPU 処理速度の増加幅が大きい一方、work_mem = 256 MB の場合は、同時処理の効果によって CPU 処理速度が増加はしているが、同時処理数の増加に伴いキャッシュ参照の競合の度合いが高まるため、逐次処理の場合と比較するとキャッシュミス数が増加するオーバーヘッドが生じ、work_mem = 256 kB の場合と比較すると、性能向上が小さくなっている。

5.1.2 I/O スループットと全体の処理性能

以下では、全ての問い合わせ処理のハッシュテーブルが L3 キャッシュに収まる、work_mem < $2MB$ の範囲に対象を絞り議論を進める。

まず、同時処理数 1 - 8 では実行時間はほとんど増加していない。これは、SSD が並列 I/O 処理能力をもつため、複数の処理で同時に I/O が発行されても、I/O コストが変化しないためである。

図 8 は、work_mem = 256kB のときについて、問い合わせ同時処理数毎に処理中の I/O スループット (MB/S) のタイムラインを示したものである。各図では、読み込み I/O スループットが高くなっているタイムラインの前半部分 (例えば、図 8a では 0 - 30s 部分、図 8g では 0 - 260s 部分など) は、ビルドフェーズの part 表と lineitem 表のスキャンを示しており、後半部分はプローブフェーズにおけるパーティションファイルのスキャンを示している。書き込み I/O は、ビルドフェーズで生成されたパーティションファイルによるものである。なお、同時処理数 1 - 4 のときは、プローブフェーズにおいてパーティションスキャンの I/O が生じていないが、これは PostgreSQL が OS のファイルキャッシュを利用しており、パーティションのデータが全てファイルキャッシュに収まっているためである。

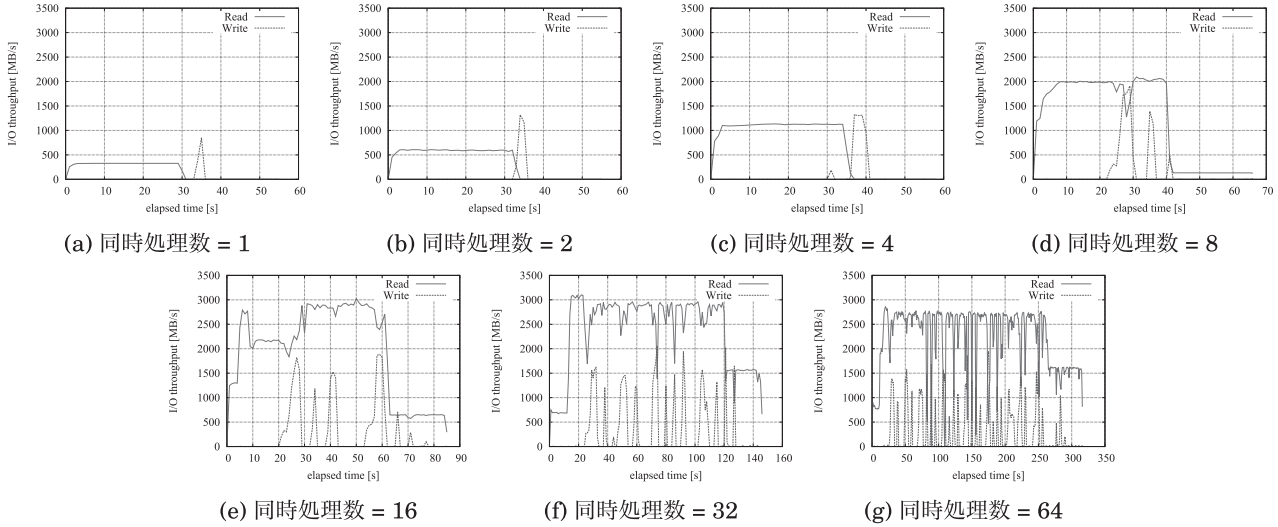
図 8 では、同時処理数を増やしたとき I/O スループットが飽和しているのは、ビルドフェーズの部分である。ビルドフェーズにおける I/O スループットの平均値を図 9 に示す。同時処理数 1 - 8 の範囲では、同時処理数に比例して I/O スループットが増加している。16 のときは、8 のときの 1.4 倍程度に留まっており、読み込み I/O スループットは 2600MB/s 付近まで達し、飽和している。また、32 から 64 では 200MB/s 程度読み込みスループットが低下しているため、I/O コストが 2 倍以上に増加してしまっている。図 6 を参照すると、同時処理数 32 から 64 では、work_mem が小さいほど I/O コストが大きく増加している。原因としては、work_mem が小さく、同時処理数が多いほどパーティションファイル数が増加し、フラグメンテーションが生じるため、ファイルオープンのオーバーヘッドが増加することや、SSD の書き込みバッファの使用効率が低下することが挙げられる。また、同時処理数が多いほど、多くのファイルへの I/O が混在することになるため、テーブルスキャンの読み込み I/O に対する先読みが機能しにくくなる [6] ということも考えられる。

また、I/O スループットは、ハッシュ結合のビルドフェーズにおいてのみ飽和していたため、16 以上の同時処理数においても、I/O が重くないプローブフェーズでは、更なる I/O スループット向上が期待できる。

5.2 SSD を用いた関係データベースシステムにおける問い合わせ処理スケジューリング

図 6 の計測によって、SSD を使用した関係データベースシステムにおいて、複数問い合わせの同時処理によって処理性能が向上することが明らかになった。同時に処理される全てのハッシュ結合演算のハッシュテーブルをキャッシュに収めるよう、キャッシュレベルでデータベースバッファ管理を行うことで、キャッシュミス増加を抑えたまま同時処理数を増加することができ、処理性能向上が大きくなる。キャッシュレベルのデータベースバッファ管理を行った場合の例として、図 6 の work_mem = 256 kB の場合では、同時処理数 1 から 16 に増やしたときに、システム全体の処理性能としては 13 倍の性能向上がみられた。

一方、I/O スループットが帯域の上限まで達した後の、同時処理数 16 から 32 の部分では、I/O コストが 2 倍以上に増加しており、全体の実行時間としても 2 倍以上に増加している。このように、同時処理数を増やしていくことで、並列 I/O 処理によりスループットが増加するが、一定の並列 I/O 処理数を境にスループットは低下に転じる。これは、同時に複数の処理で I/O を発行する



(a) 同時処理数 = 1　(b) 同時処理数 = 2　(c) 同時処理数 = 4　(d) 同時処理数 = 8

(e) 同時処理数 = 16　(f) 同時処理数 = 32　(g) 同時処理数 = 64

図 8: 各問い合わせ同時処理数における処理中の I/O スループットのタイムライン (work_mem = 256kB)

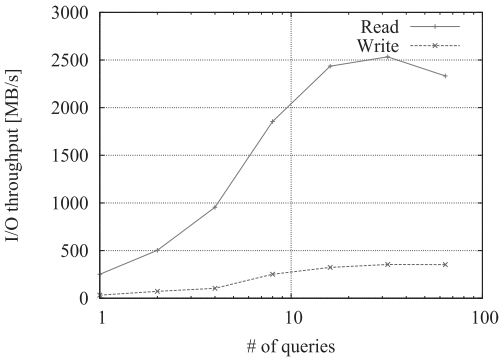


図 9: 各問い合わせ同時処理数におけるビルドフェーズの I/O スループット

ことで、I/O の複雑性が高まりフラグメンテーションが生じてしまうためである。SSD においても過度なフラグメンテーションによって I/O スループットが低下してしまう。

このような弊害を避けるため、同時処理数の制御が不可欠である。SSD を用いた関係データベースシステムにおいて同時処理を有効に利用するためには、新たに問い合わせのスケジューリング機構が必要となる。スケジューラが考慮すべき要素としては、I/O スループットを最大化するための同時処理数の選択と、キャッシュレベルのデータベースバッファ管理が挙げられる。I/O ワークロード調節の手法や、問い合わせ間のキャッシュ分割使用のアプローチについてはこれからの研究課題となる。

6. 関連研究

SSD を利用したデータベースシステムに関しては、既に多くの研究が見られる。問い合わせの実行に関するものでは、バッファプールの拡張や SSD と HDD のハイブリッドストレージ管理、インデクシングなどが中心となっている。

バッファプール拡張は、SSD を DRAM の下のキャッシュ階層として扱いデータベースシステムのバッファプールを拡張するものである。Bhattacharjee らの、temperature-aware caching (TAC) schema [4, 5] では、データのアクセスパターンをモニタリングして統計をとり、アクセス頻度に応じたキャッシュ置換アルゴリズムを提案している。Kang らの提案する FaCE システム [9] では、キャッシュ置換アルゴリズムに multiversion FIFO を利用し、Flash の弱点であるランダム書き込みを削減している。バッファプール拡張は、既にいくつかの商用データベースシステムで実装され組み込まれている。例としては、Oracle Exadata[17]、IBM XIV Storage System[1] などが挙げられる。

SSD は、現在では容量や価格の面では HDD に劣るため、これらの弱点を HDD によって補うために、SSD と HDD のハイブリッドストレージの運用も重要である。Koltsidas らの研究 [11] では、モニタリングにより、データのページ毎のワークロードを特定し、read-intensive なページを SSD に、write-intensive なページを HDD に配置する方法を提案している。これにより、SSD への書き込み回数を低減させ、書き込み回数の限度が小さい問題を緩和している。[8] で述べられているように、バッファプール拡張やハイブリッドストレージで用いられている、モニタリングをベースとした手法は主に OLTP のようなデータの局所性が高いワークロードで効果を発揮する。これに対し本研究では、意思決定支援システムなど、アドホックな大規模データ問い合わせの処理を行うシステムを前提としている。hStorage-DB[13] では、DBMS の持つ、問い合わせに関する semantic information を利用したデータ管理を行っている。この手法では、データアクセスパターンが実行前に解析されるため、モニタリングのためのオーバーヘッドや処理開始直後のデータ配置の問題などが生じない。本稿で述べた、複数問い合わせの同時処理スケジューリングにおいても、こうした問い合わせのもつ I/O ワークロードに関する情報を活用する必要がある。

インデクシングに関する研究では、FD-tree[12] は、データ更新時に書き込まれたデータをバッファリングし、SSD への書き込みをシーケンシャルに行う最適化を実現している。PIO B-tree[9] では、SSD の並列 I/O 処理能力に着目した最適化を行っている。インデックスは、リレーションに対するデータの選択率が比較的小さいときに多用される。一方で、大規模データ処理問い合わせで選択率が高い時はシーケンシャルアクセスや、ハッシュテーブルを用いた処理などが多用される。Tsirogiannis らの研究 [16] では、カラムストアによってデータベースを格納し、必要なカラムの読み込みだけを行いメモリ使用量及び I/O を削減する Flash scan を提案している。また、ハッシュ結合についてもセミジョインを利用し、同様の最適化を行った Flash join を提案している。Flash scan 及び Flash join ではランダムアクセスが頻発するが、SSD ではランダムアクセスが高速なため HDD を使用する場合と比較して I/O コストを抑えることができる。

既存研究の多くは、SSD のアクセス特性に着目し、I/O を最適化するデータ構造や処理のみを目的にしている。本稿では、I/O の

挙動のみでなく、I/O 以外のボトルネックについても分析し、SSD を用いたデータベースにおける問い合わせの処理全体の性能向上について論じている。

7. おわりに

本稿では、SSD を用いた関係データベースシステムにおいて、複数問い合わせ同時処理に着目し、システム全体の処理性能を向上する手法について論じた。従来の HDD に基づいた問い合わせ処理方式を見直し、SSD に適した処理方式の検討を行った。

まず、大規模データ処理で多用されるハッシュ結合演算を対象として、単体での処理時の性能計測を行い、SSD を用いる場合は CPU コスト最適化の重要性が高いことを明らかにした。SSD を使用したデータベースでは、ランダム I/O の処理コストが小さいため I/O コストを小さく保ちながら、データベースバッファサイズを小さくすることが可能である。本稿で行った計測では、データベースバッファサイズを 64kB まで縮小しても、I/O コストは増加しなかった。ハッシュテーブルアクセスにデータアクセス局所性があることを利用し、データベースバッファを小さく設定することで、ハッシュテーブルをキャッシュに収めキャッシュミス数を抑えることで処理性能を向上できることを示した。

また、データベースバッファ使用量を低減することで、複数問い合わせの同時処理においてキャッシュを分割して使用することが可能になり、キャッシュミスの増加を抑えて同時処理を行うことが可能になった。ハッシュ結合演算の複数同時処理において、同時処理中の全てのハッシュ結合演算のハッシュテーブルをキャッシュに収めるようデータベースバッファを管理する処理方式によって、従来の HDD の特性に基づいた処理方式の場合よりも処理性能向上が大きくなることを実証した。

本研究によって、SSD を用いた大規模関係データベースシステムの処理性能向上にむけて、1 つの指針を示すことができたと考える。SSD を用いる場合では HDD の場合と比較して、データ局所性と問い合わせ同時処理の重要性が高い。今後の研究課題として、問い合わせの同時処理のスケジューリング機構を導入することや、ハッシュ結合演算等のデータベース演算のアルゴリズムを SSD に適したものへと見直すことが必要である。

[謝辞]

本研究の一部は JSPS 科費 24300034 の助成を受けたものです。また、本稿を精読し、助言をしていただいた査読者の皆様に感謝致します。

[文献]

[1] IBM XIV Storage System. http://www-03.ibm.com/systems/storage/disk/xiv/index.html.

[2] PostgreSQL. http://www.postgresql.org/.

[3] Transaction Processing Performance Council, an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/.

[4] Bishwaranjan Bhattacharjee, Kenneth A. Ross, Christian Lang, George A. Mihaila, and Mohammad Banikazemi. Enhancing recovery using an ssd buffer pool extension. DaMoN '11, pp. 10–16. ACM, 2011.

[5] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, Vol. 3, No. 1-2, pp. 1435–1446, September 2010.

[6] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. *SIGMETRICS Perform. Eval. Rev.*, Vol. 37, No. 1, pp. 181–192, June 2009.

[7] Thomas Claburn. Google Plans To Use Intel SSD Storage In Servers, 2008. http://www.informationweek.com/infrastructure/storage/google-plans-to-use-intel-ssd-storage-in-servers/d/d-id/1067741?

[8] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging DBMS buffer pool using SSDs. SIGMOD '11, pp. 1113–1124. ACM, 2011.

[9] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, Vol. 5, No. 11, pp. 1615–1626, July 2012.

[10] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Relational Algebra Machine GRACE. In *Proceedings of RIMS Symposium on Software Science and Engineering*, pp. 191–214. Springer-Verlag, 1983.

[11] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, Vol. 1, No. 1, pp. 514–525, August 2008.

[12] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, Vol. 3, No. 1-2, pp. 1195–1206, September 2010.

[13] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, Vol. 5, No. 10, pp. 1076–1087, June 2012.

[14] Domas Mituzas. Flashcache at Facebook: From 2010 to 2013 and beyond, 2013. https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920.

[15] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD '89, pp. 110–121. ACM, 1989.

[16] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query Processing Techniques for Solid State Drives. SIGMOD '09, pp. 59–72. ACM, 2009.

[17] Ronald Weiss. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. White paper, Oracle, 6 2012. http://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf.

鈴木 恵介 **Keisuke SUZUKI**
富士通株式会社
早水 悠登 **Yuto HAYZMIZU**
東京大学生産技術研究所
横山 大作 **Daisaku YOKOYAMA**
東京大学生産技術研究所
中野 美由紀 **Miyuki NAKANO**
芝浦工業大学
喜連川 優 **Masaru KITSUREGAWA**
東京大学生産技術研究所, 国立情報学研究所