修　　士　　論　　文

大規模データベースシステムにおけるSSDを使用した問い合わせ処理高速化に関する研究

A Study on Acceleration Techniques for Query Processing of Large Database Systems on SSDs

指導教員　　**喜連川　優　教授**

東京大学　情報理工学系研究科
電子情報学専攻

氏　名　　**48-126423 鈴木　恵介**

提　出　日　　**平成26年2月6日**

概 要

大規模データ処理システムにおいては、さらなる処理能力の向上が望まれている。大規模な解析的データ処理を行う関係データベースシステムでは、I/O が性能のボトルネックとなるため、HDD と比較して高い転送レートを持つ SSD が、高速化に適したストレージデバイスとして注目されている。ところが、従来のデータベースの問い合わせ処理方式では、SSD を用いた利点を十分に活かしていない。

本論文では、SSD を使用する大規模関係データベースシステムの処理を高速化する問い合わせ処理方式として、関係データベース演算処理のデータアクセス局所性を利用し、データベースバッファをキャッシュレベルで管理する手法を提案した。この手法では、SSD を用いたハッシュ結合演算ではデータベースバッファ使用量を縮小し、処理においてランダム I/O が増加しても、I/O 処理コストが増加しないことを利用し、HDD 使用時より細かい粒度でデータベースバッファを管理することで、キャッシュミス数を低減し、CPU コストを減少させる。また、この手法では、複数の演算でキャッシュを分割して使用することを可能にするため、複数問い合わせの同時処理によって高いスケーラビリティで処理性能を向上できる。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や、ハッシュ結合演算とテーブルスキャンの同時処理の性能計測を行い、バッファサイズや同時処理数を適切に設定することで、処理性能が向上することを実証した。

謝辞

はじめに、指導教官の喜連川優教授に深く感謝いたします。喜連川教授には本研究について貴重な助言を賜り、ご指導して頂きました。本研究室における二年間では、研究分野に関して深い知識と鋭い洞察力を有しておられる方々からの質の高いご指導を賜り、また豊富な実験設備を与えて頂いて思うままに勉学、研究に励むことができました。こうした環境は、喜連川教授を初めとする研究室の先輩方によって築きあげられた成果の結晶であり、そのような場所で日々刺激を受け、意義深い時間を過ごすことができて幸せでした。

次に、研究室スタッフの皆様には、本研究に関する議論や論文の執筆指導など、研究生活に関わる様々な面で熱心にご指導して頂いたことに感謝いたします。中野美由紀教授、横山大作助教には、研究の指針に関する相談や研究発表の練習に度々お付き合い頂き、数多くの助言を頂きました。合田和生特任准教授、博士過程三年の早水悠登さんには、計算機における測定技法に関して初歩の初歩から系統立てて丁寧にご指導して頂き、技術面で研究の土台となる知識を与えて頂きました。また、豊田正史准教授をはじめとする、週次の進捗報告会の参加者の皆様には、様々な視点から意見を頂き、大変勉強になりました。併せて、研究を快適を行えるよう研究室環境を支えてくださった秘書の皆様に感謝いたします。

最後に、常に私のことを気にかけ、故郷の静岡から私を支えてくださった家族へ感謝の意を捧げます。

2014 年 2 月 6 日

目 次

謝辞 **i**

第 1 章　はじめに **1**

1.1　大規模データ処理高速化の必要性 1

1.2　本研究の目的と貢献 3

1.3　本論文の構成 5

第 2 章　SSD とその I/O 性能 **7**

2.1　フラッシュメモリの基本特性 7

2.2　フラッシュメモリをベースにした SSD 8

2.3　SSD の I/O 基本性能 11

2.3.1　実験環境 11

2.3.2　シーケンシャルアクセス・ランダムアクセスのスループット . 11

2.3.3　並列 I/O 処理のスループット 12

2.4　SSD の I/O 特性 17

第 3 章　関係データベースシステムと結合演算 **18**

3.1　大規模関係データベースシステム 18

3.1.1　大規模データ処理アプリケーション 18

3.1.2　TPC ベンチマーク 21

3.1.3　SSD を用いた関係データベースシステム 22

3.2　ハッシュ結合演算 23

3.2.1　Grace ハッシュ結合 [16] 24

3.2.2 ハイブリッドハッシュ結合 [27] 25
3.2.3 ハッシュ結合演算の処理コスト 25
3.2.4 複数ハッシュ結合演算の処理 26
3.2.5 SSD を用いたハッシュ結合演算 27

第 4 章 関連研究 **29**
4.1 SSD を用いた関係データベースシステムの研究動向 29
4.2 バッファプール拡張 30
4.3 ハイブリッドストレージ 31
4.4 インデックス 32
4.5 ページレイアウト 32

第 5 章 SSD を用いた結合演算処理とその性能評価 **34**
5.1 SSD を用いた関係データベースシステムの実験環境 34
5.2 単一のハッシュ結合演算を含む問い合わせの処理性能 35
5.2.1 ハッシュ結合演算におけるデータアクセス局所性 39
5.3 複数のハッシュ結合演算を含む問い合わせの処理性能 40
5.3.1 複数ハッシュ結合処理時のデータアクセス局所性 48
5.4 キャッシュサイズを考慮したハッシュ結合演算処理方式 48

第 6 章 SSD を用いた複数問い合わせの同時処理 **52**
6.1 大規模データを扱う問い合わせの同時処理 52
6.2 ハッシュ結合を行う問い合わせの複数同時処理性能の計測 53
6.2.1 CPU コストの増加傾向の変化 54
6.2.2 I/O スループットと全体の処理性能 57
6.2.3 ハッシュ結合同時実行数の制御 58
6.3 ハッシュ結合とスキャンの同時処理 59
6.4 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 61

第 7 章　おわりに **64**

参考文献 **67**

発表文献 **72**

図 目 次

2.1 SSD の内部アーキテクチャ([25] より引用) 9
2.2 SSD と HDD それぞれにおけるシーケンシャルアクセスとランダムアクセスの I/O スループット 12
2.3 SSD の並列シーケンシャルアクセスの I/O スループット 13
2.4 SSD の並列ランダムアクセスの I/O スループット 14
2.5 逐次の混合ワークロードにおける SSD と HDD の I/O スループット . 15
2.6 SSD における読み書き混合ワークロードの並列 I/O スループット . . 16

3.1 OLTP システムと OLAP システムの関係 19
3.2 TPC-H ベンチマーク Q9: 地域毎の売上利益の集計を行う問い合わせ 20
3.3 TPC-H ベンチマークのテーブルとレコード数の関係 22
3.4 ハッシュ結合演算によるタプルのマッチング 23
3.5 Grace ハッシュ結合の処理フロー 25

5.1 lineitem 表と part 表の結合演算を行う問い合わせ 35
5.2 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 36
5.3 `work_mem` = 128 kB のときの図 5.1 の問い合わせ実行時の I/O スループットのタイムライン 37
5.4 図 5.1 の問い合わせ処理時の各 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 上の実行) 38
5.5 二つの結合演算を行う問い合わせ 40
5.6 図 5.5 の問い合わせの実行プラン 40

5.7 二つ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 41
5.8 図 5.5 の問い合わせ処理時の各 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 上での実行) 42
5.9 work_mem = 128 kB のときの図 5.5 の問い合わせ実行時の I/O スループットのタイムライン 42
5.10 TPC-H query 8 (一部の計算部分削除) 44
5.11 TPC-H query 8 実行プラン 45
5.12 TPC-H query 8 の各 work_mem の値における実行時間 45
5.13 図 5.10 の問い合わせ処理時の各 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 上での実行) 46
5.14 work_mem = 128 kB のときの TPC-H query 8 実行時の I/O スループットのタイムライン 46
5.15 HDD の特性に基づいて設計された従来の関係データベースシステムの処理方式 49
5.16 提案手法:キャッシュレベルでデータベースバッファを管理する処理方式 50

6.1 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 . . 54
6.2 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) 55
6.3 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 = 256 kB) 56
6.4 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 (work_mem = 256 kB) 58
6.5 ハッシュ結合演算を行う問い合わせの実行時間 (左:ハッシュ結合演算単体、右:スキャン同時処理時) 60
6.6 ハッシュ結合演算とスキャン同時処理時の各 work_mem の値における L3 キャッシュ参照・ミス数及びミス率 61

第1章　はじめに

1.1　大規模データ処理高速化の必要性

情報機器の普及に伴い、社会の情報化が進み、人間が生成するデジタルデータは増加の一途をたどっている。多くの企業や組織が、データを収集・分析することで、市場や世情の調査を行い、企業経営等の意思決定に利用している。米国最大手小売業 WalMart 社では、2013 年時点で 100 万人毎時を越える顧客情報を取り扱っており、データベースに蓄積されるデータ量は、2.5 PB 毎時以上である [5]。また、インターネット上のソーシャル・ネットワーキング・サービスサイト Facebook では、毎日 2 億 5000 万の写真のアップロードや、8 億のユーザー間インタラクションがみられる。さらに、[26] の調査によると、1PB を越えるデータを保有する企業は 2013 年時点で 10%であり、3 年後には 23%まで上昇する見込みであるという。こうしたペタバイト級、さらにはエクサバイト級のデータを解析し、日々の経営判断に利用するためにはより多くのデータを高速に処理する計算基盤が必要不可欠である。

蓄積された大規模なデータを解析するには、日々更新されるデータを全てアクセスしなければならない。つまり、大規模意思決定支援システムなどでは、ストレージ上のデータの大部分を読み込む必要がある。大規模データ解析処理システムの処理性能は、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、処理性能向上が難しい。大量のデータへのアクセスを高速化するためのアプローチとして、大容量のメモリを搭載した計算機を用いる方法があるが、一般にそのようなエンタープライズ用途の高機能システムは非常に高価であるため、より手軽に導入できる代替案が望まれている。

近年では、NAND フラッシュや Phase Change Memory、FeRAM などの Storage Class Memory (SCM) と総称される、大容量・不揮発・HDD よりも高速、という特徴をもつストレージデバイスの研究、開発が盛んになっており、HDD に置き換わる存在として注目を集めている。中でも、NAND フラッシュをベースにした Solid State Drive (SSD) は、NAND フラッシュチップ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いる。Google[11] や Facebook[22] などでの使用例もみられ、[5] によると、2013 年時点で、SSD をデータストレージに使用している企業は 53%であるという。

現在の意思決定支援システムは関係データベース上に構築されていることが多い。関係データベースシステムは、大量の情報処理や管理の用途で広く使用されており、データセンターにおいて中心的な役割を担っている。関係データベースシステムにおいて SSD を使用することで、I/O スループットの増加によって、全体の処理性能が向上することが期待される。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 の使用を前提として設計されており、SSD を用いる利点を十分に活かしていない。SSD のアクセス特性を活用するため、関係データベースシステムの問い合わせ処理方式を見直す必要がある。

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オーバーヘッドがなく、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。

関係データベースシステムの処理においてはこれらの特性を考慮すると、ランダムアクセススループットが高いため、小さな I/O サイズでランダムアクセスが多い

ワークロードでもI/Oスループットが飽和しにくくなり、HDD使用時よりランダムI/Oを許容したデータ処理方式を選択できる。また、複数の処理を同時に行い、並列I/O処理能力を活用することで、システムの処理を高速化することが可能である。

このように、SSDを用いたシステムでは、高いI/Oスループットによって、処理性能を向上することができる。HDDを使用したシステムでは、全体の処理性能において、I/Oコストが支配的であったが、SSDによってI/Oボトルネックが解消されると期待される。これにより、CPU演算やメモリアクセスなど、新たなボトルネックが顕在化することが考えられる。SSDを使用したシステムの性能を最大限に引き出すためには、I/Oスループットの向上だけでなく、新たに現れるボトルネックにも着目し、改善する必要がある。

1.2 本研究の目的と貢献

本論文では、大規模関係データベースシステムにおいて、SSDのアクセス特性を十分に活かした、解析的問い合わせの処理方式を検討し、SSDを用いた関係データベースシステムの性能向上を図る。従来のHDDを用いた関係データベースシステムでは、入出力コストが支配的であるため、CPU処理コスト等についてはHDDの性能に応じた検討しか行われていない。そこで、従来の関係データベースシステムの実装方式でSSDを用いてもその性能が十分活かされているか、改めて性能評価を行い、詳細に解析する。ここで得られた知見を基に、SSDを用いた関係データベースシステムに適した問い合わせ処理方式を設計し、実際の関係データベースシステムを用いて評価を行う。

関係データベースシステムでの大規模データ処理において多用され、解析的問い合わせ処理において大きな割合を占めることが多い、ハッシュ結合演算を中心として、SSDを用いた高速化について議論する。

まず、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、以下の事実を明らかにした。

- 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 を使用した関係データベースシステムでは、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、バッファ使用量を縮小することが可能である。

- バッファ使用量を小さく設定することで、ハッシュテーブルサイズが小さくなり、キャッシュミス数が小さくなるため処理性能が向上する。このように、SSD を使用した関係データベースシステムでは、ハッシュ結合演算のデータアクセスの局所性を考慮する重要性が高い。

- バッファ使用量を小さく設定することで、キャッシュやメモリ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。

この結果のように、SSD を用いたハッシュ結合演算においては、I/O の処理コストが低減され、全体の処理コストに対して、CPU の処理コストの割合が増加する。そのため、キャッシュミスのペナルティによる、メモリアクセスが頻発することで、処理性能が低下してしまう。HDD を用いたハッシュ結合演算の処理方式では、ハッシュテーブルのデータアクセスの局所性をメモリレベルでしか考慮していなかったため、キャッシュミス数の増加が生じる。そこで、SSD を用いた場合に、ハッシュテーブルのデータアクセス局所性を高め、データベースバッファをキャッシュレベルで管理する処理方式を提案する。

実際の関係データベースシステムでは、複数の問い合わせを処理する必要がある。SSD では、並列 I/O 処理が可能であるため、同時処理により処理性能を向上することができる。複数問い合わせの同時処理では、各問い合わせ処理においてキャッシュを共有するため、キャッシュミス数が増加しやすくなる。HDD を用いたハッシュ結合演算の処理方式では、大量のデータベースバッファを使用する必要があったため、複数演算を同時処理するとキャッシュ参照の競合が発生してしまい、その分逐次処理時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る。一方、データベースバッファをキャッシュレベルで管理する処理方式では、キャッシュを分割して使用することで、キャッシュ参照の競合を抑えることができる。同時処理を行っても、CPU コスト増加のオーバーヘッドが発生しないため、データベースバッファ

をキャッシュレベルで管理する処理方式と比較すると、相対的な速度差は更に広がる。このことを基に、SSD を使用した関係データベースシステムにおける、複数問い合わせの同時処理について検討及び検証を行った。

- 複数問い合わせ同時処理時において、ハッシュ結合演算のバッファ使用量を適切なサイズに設定することで、ハッシュテーブルのデータアクセスの局所性を高め、キャッシュミス数を低減し、CPU コストの増加を抑えることができる。
- SSD の並列 I/O 処理能力によって、複数問い合わせ同時処理における I/O スループットが増加する。ただし、過度に I/O の並列処理数を増加させると、SSD においても I/O のフラグメンテーションにより、スループットが低下してしまう。
- 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、ハッシュ結合演算の同時実行や、ハッシュ結合演算とテーブルスキャンの同時実行について計測を行い、ハッシュ結合演算のバッファ使用量や同時実行数を適切に設定することで、全体の処理を高速化できることを実証した。

1.3 本論文の構成

本論文の構成は次のとおりである。

第 2 章 SSD の一般的な特性について述べたのち、実際の SSD を使用して I/O 性能の計測及び分析を行う。また、その結果を基に、SSD を使用するシステムにおいて、考慮すべき I/O 特性について説明する。

第 3 章 本論文で対象とする、大規模関係データデータベースシステムのワークロードの特性について説明する。そして、本論文のアプローチとその位置付けについて説明する。

第 4 章　関係データベースシステムにおける SSD の使用に関する、これまでの研究について述べる。

第 5 章　ハッシュ結合演算を対象とし、TPC-H のデータベースを用いた計測を行い、その性能を詳細に解析する。

第 6 章　SSD を使用した関係データベースシステムにおける、複数問い合わせの同時処理性能の計測を行い、高速化の効果を実証する。また、計測の結果を基に、複数問い合わせ処理のスケジューリング方法について検討する。

第 7 章　全体のまとめと今後の課題について述べる。

第2章　SSDとそのI/O性能

本章では、フラッシュメモリをベースにしたSSDの性能特性について述べる。なお、フラッシュメモリにはNAND型とNOR型が存在するが、一般に計算機でデータストレージとして用いられるのは集積度や書き込み性能で勝るNAND型であり、本論文でもフラッシュメモリと言えばNAND型フラッシュメモリを指すものとする。

2.1　フラッシュメモリの基本特性

フラッシュメモリは、EEPROM (Electrically Erasable Programmable Read-Only Memory) の一種である。HDDと比較してデータアクセスが高速であり、特にランダムアクセスについては、HDDの100 - 1000倍の転送レートを持つため、HDDに代わるストレージデバイスとして注目されている。

フラッシュメモリは、各セル当たりに格納するデータ量によって、SLC (Single Level Cell) とMLC (Multi Level Cell) の2種類に分類される。SLCは、アクセス速度や耐久性に優れるが、容量単価が高いためハイエンドな製品で使用され、MLCは、安価で大容量化しやすいため、容量が重視される製品で使用される。フラッシュメモリの一般的な特性として、以下が挙げられる。

HDDと比較して高速なランダム読み込みI/O :

フラッシュメモリは電子回路のみで構成されており、HDDのような物理的な機械部分が存在しない。シーク時間が存在しないため、シーケンシャルアクセスやランダムアクセスといったアクセスパターンに関わらず、転送レートが高くかつ一様である。

インプレースでない書き込み I/O :

フラッシュメモリのデータ更新では、既に存在しているデータを同じアドレスに対して直接上書きすることができず、古いデータを消去し、新たに書き込むといった手順が必要である。データの読み込みや書き込みは、ページ (2 kB や 4 kB など) 単位で処理され、データ消去はブロック (64 - 128 程度のページの集合) 単位で処理される。現在のフラッシュメモリにおける、それぞれの処理のレイテンシを表 2.1 に示す。ブロック消去処理は、読み込みや書き込み処理と比較して低速である。ランダム書き込み I/O では、ブロック消去が頻発するため速度が低下しやすくなる。

書き込み I/O の限度回数 :

フラッシュメモリのセルは、トランジスタに、電荷を蓄積するためのフローティングゲート層が加えられたものである。ページ書き込みやブロック消去の処理ではセルに蓄積された電荷を変化させるため、大きな電流を流す。これによりトランジスタの酸化被膜が劣化し、フローティングゲート層を分離できなくなると電荷の保持ができなくなり、データを扱えなくなる。現在では、ページ書き込み・ブロック消去処理の限度回数は、MLC で 10^3-10^4 程度、SLC で 10^4-10^5 程度となっている。

低消費電力・耐衝撃性 :

フラッシュメモリは電子回路のみで構成されており、HDD のような物理的な機械部分が存在せず、電力消費が小さい。また、スタートアップに必要な時間も HDD と比較して短くなっている。機械部分がないことは、衝撃耐性の面でも有利である。

2.2 フラッシュメモリをベースにした SSD

フラッシュメモリ単体では、帯域が限られている (32 - 40 MB/s 程度 [6])、ランダム書き込み I/O が低速、各セルの書き込み限度回数があるなどといった問題点が存

表 2.1: フラッシュメモリの各処理のレイテンシ [28]

ページ読み込み	80 μs
ページ書き込み	200 μs
ブロック消去	1500 μs

在する。これらの問題点を改善するものとして、近年では、複数のフラッシュチップを内蔵したフラッシュSSD が用いられている。

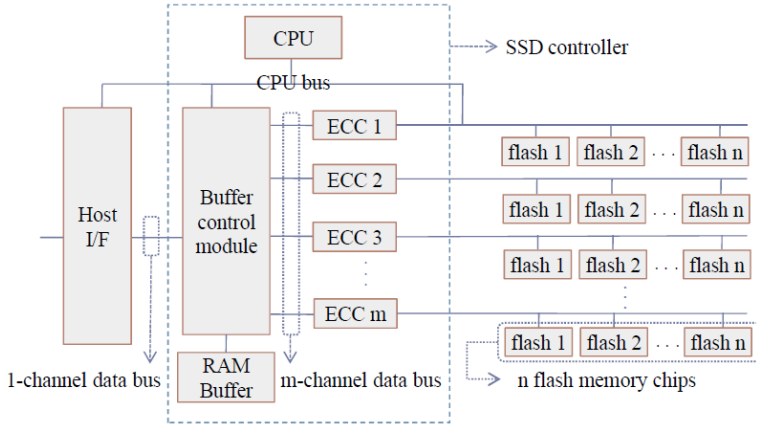


図 2.1: SSD の内部アーキテクチャ([25] より引用)

フラッシュメモリをベースにした SSD の一般的な内部アーキテクチャを図 2.1 に示す。SSD の構成要素の中でも、I/O 処理の高速化を実現するうえで重要となるのは、複数のフラッシュチップ、FTL (Flash Translation Layer)、RAM バッファの 3 つであり、それぞれ以下のような役割を担う。

複数フラッシュチップの内蔵 :

SSD では、I/O の高帯域化を実現するため、複数のフラッシュチップを内蔵している。多くの SSD では、図 2.1 に見られるように、内部のコントローラーに対し、複数本のチャネルが接続しており、それぞれのチャネルに複数枚のフラッシュチップが接続するという構造をとっている [10]。このため、複数チャ

ネルによる並列性と、各チャネル上に接続された複数フラッシュチップによる並列性の 2 種類を持つ。コントローラに接続された各チャネルは、独立かつ同時に処理を行うことができるので、コントローラが I/O リクエストを複数チャネルに分配することで、並列 I/O 処理を行い、スループットを増加させることが可能である。各チャネルに複数フラッシュチップを接続しているのは、チャネルの帯域使用率を向上するためである。チャネル上のそれぞれのフラッシュチップは独立に処理を行うことが可能であり、連続した論理ブロックアドレスに同一バス上の複数チップの物理アドレスをインターリーブして配置することでストラインピングアクセスを可能にし、スループットを増加させることが可能である [6, 13]。

FTL (Flash Translation Layer) :

FTL は、SSD において HDD と同様のブロックインターフェースを提供する機能の他に、特定のセルへの書き込みの集中を防ぎ、フラッシュチップの劣化を緩和するウェアレベリングや、書き込み処理の最適化を行う論理アドレスマッピングや、ガーベジコレクションの機能を提供する。フラッシュメモリのブロック消去処理は非常に低速であるので、多くの SSD では、データの更新処理の際にブロック消去が発生するのを防ぐため、未使用ページのプールを利用している。書き込み処理において、書き込み対象の論理アドレスをページプール中のページの物理アドレスにマッピングして使用し、古いページを無効化するのみといった手順をとることで、ブロック消去の発生を抑制できる。無効化されたページは、ガーベジコレクションによって後に回収され、消去されてページプールに加えられる。

RAM バッファ :

最近の SSD では、スループット増加のため大容量の RAM バッファを内蔵しているものが見られる。シーケンシャル読み込み I/O における、先読み結果のバッファリングや、書き込み I/O リクエストのバッファリングによる、書き込み処理の集約化によってスループットが増加する [9]。

表 2.2: Experimental platform setup

CPU	Xeon X7560 (L3 Cache: 24 MB) @ 2.27 GHz x 4
DRAM	64 GB
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

2.3 SSD の I/O 基本性能

本節では、現在流通している SSD を用いて、I/O の基本性能を計測・分析する。SSD において、HDD との I/O 特性の違いが大きい、ランダム I/O や並列 I/O 処理に焦点をあて、マイクロベンチマークプログラムの実行結果を示す。

2.3.1 実験環境

表 2.2 に、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。

2.3.2 シーケンシャルアクセス・ランダムアクセスのスループット

SSD が、HDD と比較して高い読み込み I/O スループットを持つことを示すため、シーケンシャルアクセスとランダムアクセスそれぞれについてマイクロベンチマークを実行した。ベンチマークプログラムは、単一のファイルに対して、指定した 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。I/O サイズを 4 kB - 2 MB の範囲で変化させ、I/O スループットの計測を行った。

図 2.2 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。図では、横軸が I/O サイズ、縦軸が MB 毎秒の I/O

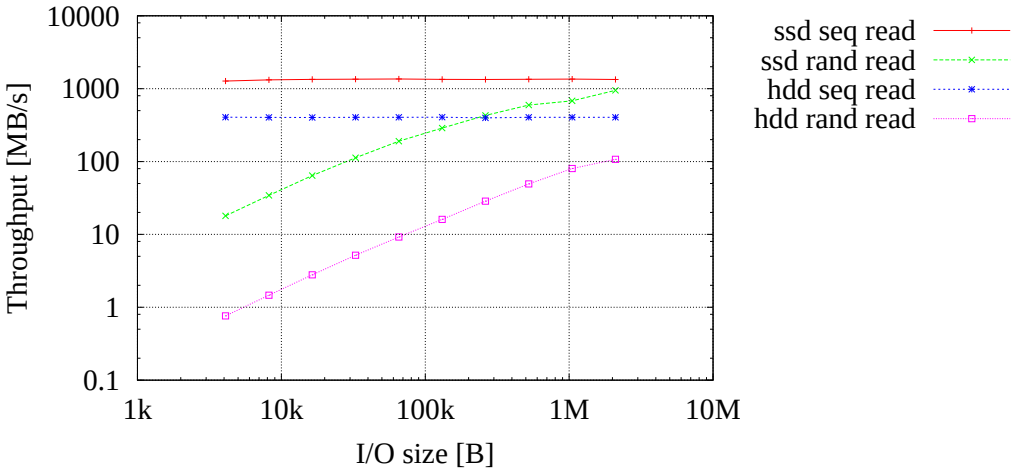


図 2.2: SSD と HDD それぞれにおけるシーケンシャルアクセスとランダムアクセスの I/O スループット

スループットを示している。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。

2.3.3 並列 I/O 処理のスループット

2.2 節で説明した通り、SSD の内部は複数のフラッシュ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という特性がある。本節では、SSD の並列 I/O 処理時の I/O スループットを計測し、並列 I/O 処理によってスループットが増加することを示す。

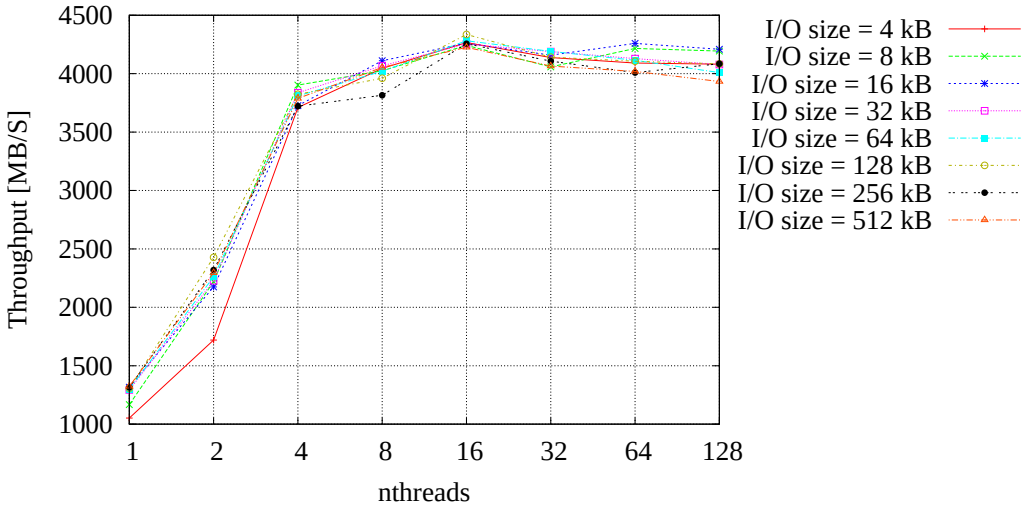


図 2.3: SSD の並列シーケンシャルアクセスの I/O スループット

2.3.3.1 複数スレッドでのシーケンシャルアクセス・ランダムアクセスのスループット

SSD 上で並列 I/O 処理を行うマイクロベンチマークを実行し、I/O スループットを計測する。ベンチマークプログラムは、マルチスレッドで動作し、各スレッドはそれぞれ 1 つのファイルに対して、指定した 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。ストレージ容量の都合により、全ファイルの合計サイズが 256 GB となるように、スレッド数分のファイルを用意し計測を行っている。主記憶サイズの 64 GB より十分に大きい範囲にアクセスすることで、ランダムアクセスにおいて、ファイルキャッシュがヒットしないようにしている。I/O サイズを 4 kB - 512 kB、スレッド数を 1 - 64 で変化させ、I/O スループットの計測を行った。

図 2.3、2.4 は、並列シーケンシャル I/O・ランダム I/O のスループットを示す。図では、横軸がスレッド数、縦軸が MB 毎秒の I/O スループットを示している。

図 2.3 の並列シーケンシャル I/O の計測結果では、8 スレッドまでスループットが増加しており、16 スレッド以上では飽和し変化が無い。スループットは、1 スレッ

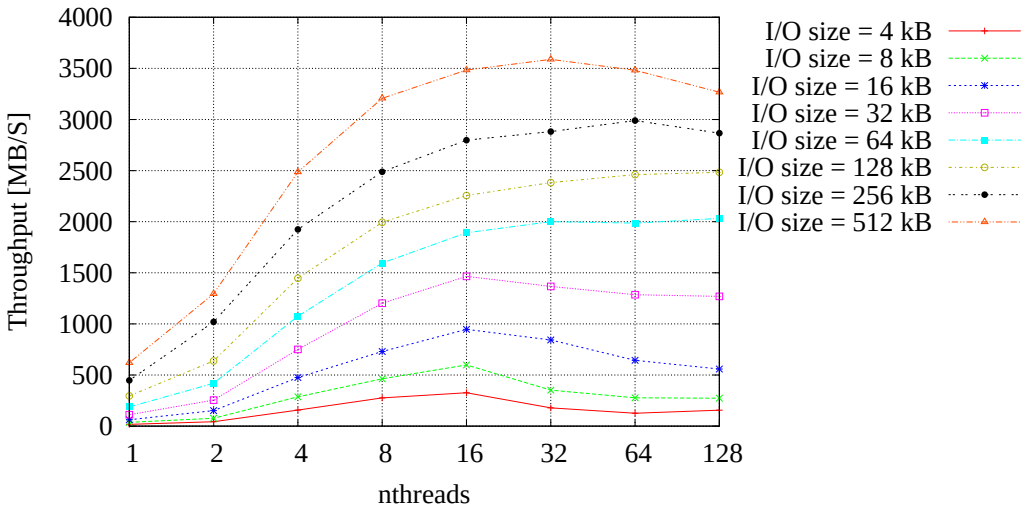


図 2.4: SSD の並列ランダムアクセスの I/O スループット

ドのときで 1300 MB/s 程度、8 スレッドのときで 4300 MB/s 程度であり、並列アクセスにより 3.3 倍増加している。

図 2.4 の並列ランダム I/O の計測結果では、I/O サイズに応じてスループットの最大値は異なっているが、各々16 - 32 スレッドでピークを迎え、その後はスループットが低下している。これは、フラグメンテーションにより、先読みなどによる SSD 内の RAM バッファの効果が低下するためであると考えられる。

2.3.3.2 読み込み I/O と書き込み I/O の混在するワークロードのスループット

実アプリケーションにおける I/O ワークロードは、読み込みと書き込みの混在するものである場合が多い。そこで、読み込みと書き込み両方が同時に実行されるワークロードにおける、I/O スループットを計測・分析する。

ベンチマークプログラムでは、後の第 5 章で計測の対象としている、図 5.1 の問い合わせ処理中の I/O ワークロードを模し、読み込み 75%、書き込み 25%となる負荷を与える。これは、計測対象のワークロードに即して I/O 帯域をより正確に計測するためである。ベンチマークプログラムはマルチスレッドで動作する。各スレッ

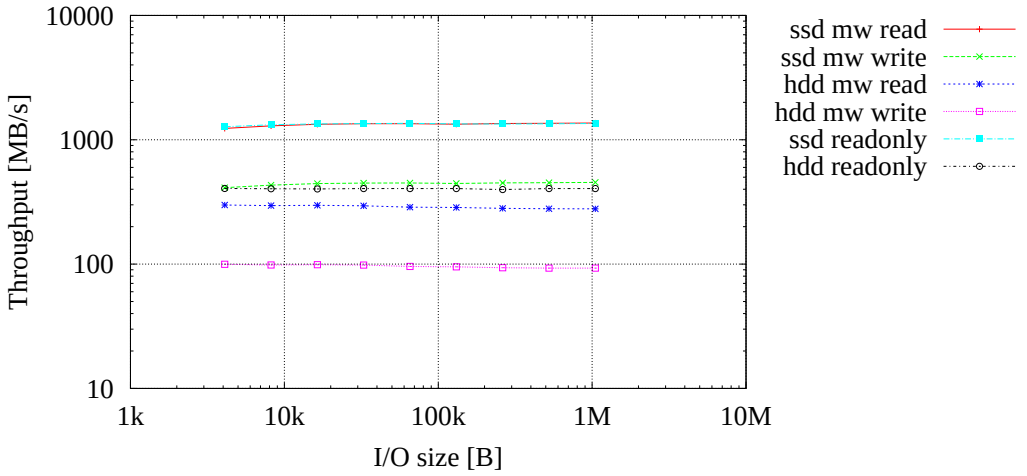


図 2.5: 逐次の混合ワークロードにおける SSD と HDD の I/O スループット

ドは、読み込み用と書き込み用 2 つのファイルをオープンし、指定した I/O サイズで、384 個の読み込み I/O を処理し、次に 128 個の書き込み I/O を処理、これを繰り返す。読み込み I/O は読み込み用ファイルをシーケンシャルスキャンするもので、書き込み I/O は書き込み用ファイルにデータを追記していくものである。

まず、SSD と HDD における混合ワークロードのスループットについて、比較するため、ベンチマークプログラムを 1 スレッドで実行する。図 2.5 に、計測結果 (図中の mw) を示す。比較のため、2.3.2 節の読み込みのみのワークロードの結果を並べてある。SSD のスループットは、混合ワークロードの読み込み、書き込み共に HDD の 4.2 倍である。混合ワークロードにおける SSD と HDD の読み込みスループットの性能差は、読み込みのみのワークロードにおける性能差よりも大きくなっている。HDD においては、混合ワークロードの読み込みスループットは、読み込みのみの場合と比較して、0.74 倍に低下している。一方、SSD においては、混合ワークロードの読み込みスループットは、読み込みのみの場合から低下しない。書き込み I/O は、メモリ上にバッファリングされ、集約化してストレージに書き込み I/O を発行するが、このとき、ベンチマークによって発行されている読み込み I/O が同時に I/O キューに入る。HDD では、これらは逐次処理されるため、I/O 帯域は読み込みと書

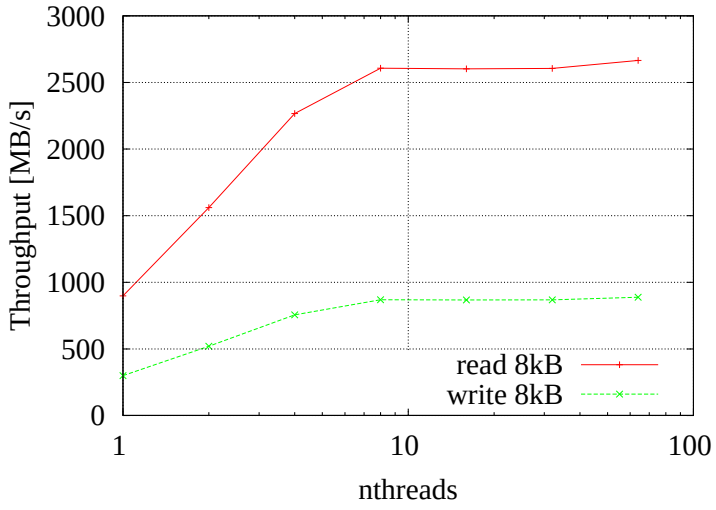


図 2.6: SSD における読み書き混合ワークロードの並列 I/O スループット

き込みによって分割される。一方、SSD では並列 I/O 処理が可能であるので、読み込み I/O と書き込み I/O を同時に処理できるため、読み込みスループットを低下することがない。このように、SSD は並列 I/O 処理によって混合ワークロードの I/O スループットを増加する。

SSD は、読み込みにおける先読みの機構や、書き込みにおけるバッファを利用したバックグラウンドプロセスでの処理機構などによってスループットを向上するが、読み書きが入り混じったワークロードでは、それらが互いに干渉しあい全体のスループットが低下することがある [9]。

図 2.6 に、ベンチマークの計測結果を示す。スレッド数 1 - 8 では、スレッド数が 2 倍になる毎に、I/O スループットも読み書き共に 1.1 - 1.7 倍増加している。スレッド数 16 以降は、I/O スループットはほぼ変化しておらず、読み込みは 2600MB/s、書き込みは 900MB/s 程度で飽和しており、このワークロードにおける I/O 帯域を示している。

2.4 SSD の I/O 特性

SSD では、FTL によって HDD と同様のブロックインターフェースが提供されているため、HDD と SSD は透過的にアクセスすることが可能である。そのため、HDD を使用したシステムにおいて、HDD を SSD に置き換えることは容易である。しかし、SSD は I/O 処理については、HDD とは全く異なる I/O 特性を示す部分も多い。

HDD を使用したシステムでは、ランダム I/O はシーケンシャル I/O と比較して非常に低速であるため、ランダム I/O を避けるデータ処理方法が重視される。一方、SSD ではランダム I/O でもシーケンシャル I/O と遜色ない速度差を得ることができるため、HDD 使用時よりランダム I/O を許容したデータ処理方法を選択できる。

また、SSD では HDD と異なり、並列 I/O 処理によってスループットが増加する。SSD の I/O 帯域の使用効率を高めるためには、非同期 I/O や、マルチプロセス・マルチスレッド処理によって並列 I/O 処理能力を活用することが必要である。

このように、SSD を使用することでシステムの HDD による I/O ボトルネックが改善され、全体の処理スループットが向上することが期待できる。一方で SSD を使用するシステムでは、I/O ボトルネックが解消され、CPU による演算やメモリアクセス、ネットワーク帯域など他の要素が新たなボトルネックになる可能性がある。SSD を使用するシステムの性能を最大限に引き出すためには、HDD と比較したときの I/O スループットの向上のみでなく、新たなボトルネックが現れることにも着目し、改善することが必要である。

第3章　関係データベースシステムと結合演算

3.1　大規模関係データベースシステム

関係データベースシステムは、大量の情報処理や管理の用途で広く使用されており、データセンターにおいて中心的な役割を担っている。関係データベースシステムの用途は多岐に渡り、その使用法によって処理の負荷も異なるため、高速化を実現するためには処理の特徴を考慮することが重要である。本節では、関係データベースシステムを用いた典型的なアプリケーションについて説明する。

3.1.1　大規模データ処理アプリケーション

関係データベースシステムの利用用途の違いから、それぞれのシステムの処理内容は大きく異なる。関係データベースシステムを用いる典型的なアプリケーションとして、オンライントランザクション処理 (OLTP) システムと、オンライン分析処理 (OLAP) システムの 2 つが挙げられる。OLTP と OLAP それぞれの処理を特徴付けるのは、個々のトランザクションが扱うデータ量、データアクセスの規則性、想定されるトランザクションの同時実行数などである。

OLAP と OLTP の使用用途、およびそれらの関係を表したのが図 3.1 である。業務における取引等の処理が OLTP システムにより行われ、OLAP システムは業務で蓄積された情報を集約・分析することにより経営者などに有用な情報を提供する。

OLTP とは、小規模なトランザクションを同時に多数扱うようなデータベース処理のことを指す。電子商取引・金融取引など、個々のリクエストへの即時応答性や、

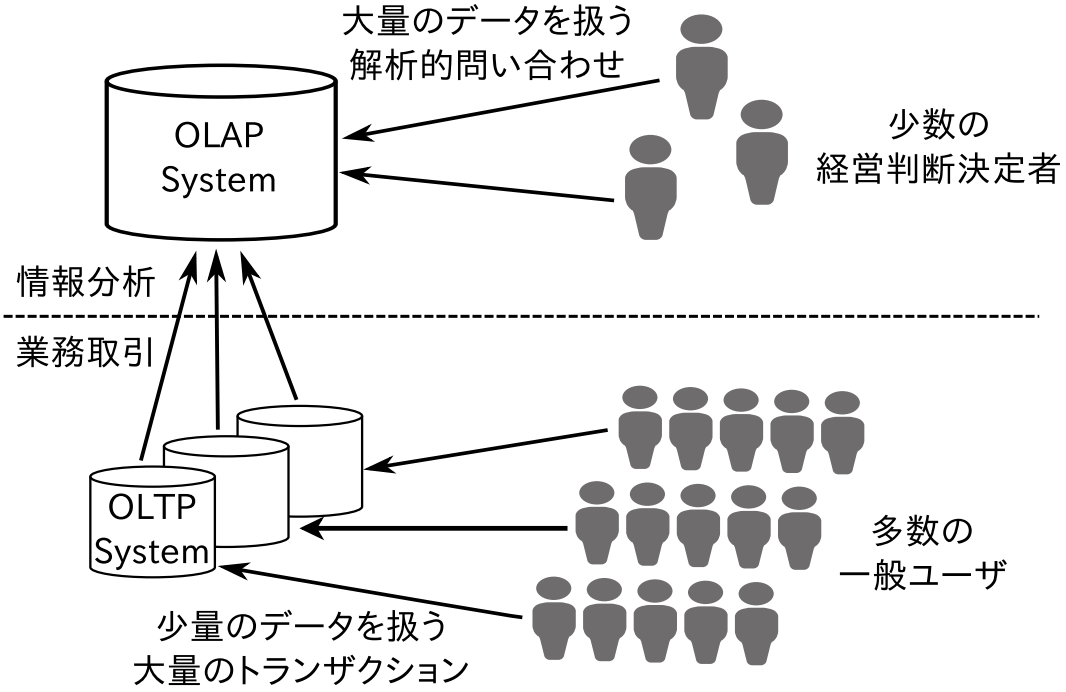


図 3.1: OLTP システムと OLAP システムの関係

大量のユーザへの同時対応が求められるアプリケーションの基盤となる技術である。例として、銀行口座への振り込みは振込元、振込先の口座残高を示すレコードの更新を行うトランザクションによって表現される。この例のように、OLTP では、個々のトランザクションが扱うデータ量は比較的小さく、同時に多数のユーザからトランザクション要求が送信される。OLTP では、一般的に特定のデータに対してアクセスが集中しやすい、つまり、データアクセスの局所性が高いため、データアクセスが高速なキャッシュやメモリのデータ管理が、処理を高速化するうえで重要な要素となる。

OLAP とは、蓄積された大量のデータを分析し、情報を抽出するデータベース処理を指す。日々蓄積されるデータを用いて、市場分析や経営判断を行う意思決定支援システム (Decision Support System、DSS) の用途で用いられることが多い。OLAP の典型的な利用例としては、1 日に蓄積された業務データを業務終了時間から翌日の業務開始までに整理・分析する夜間バッチ処理などがある。

```
1  select
2          nation, o_year, sum(amount) as sum_profit
3  from (
4          select
5              n_name as nation,
6              extract(year from o_orderdate) as o_year,
7              l_extendedprice * (1 - l_discount)
8              - ps_supplycost * l_quantity as amount
9          from
10             part,
11             supplier,
12             lineitem,
13             partsupp,
14             orders,
15             nation
16         where
17             s_suppkey = l_suppkey
18             and ps_suppkey = l_suppkey
19             and ps_partkey = l_partkey
20             and p_partkey = l_partkey
21             and o_orderkey = l_orderkey
22             and s_nationkey = n_nationkey
23             and p_name like '%green%'
24     ) as profit
25 group by
26         nation, o_year
27 order by
28         nation, o_year desc;
```

図 3.2: TPC-H ベンチマーク Q9: 地域毎の売上利益の集計を行う問い合わせ

図 3.2 に OLAP における問い合わせの例を示す。これは OLAP の業界標準ベンチマーク TPC-H における問い合わせのひとつである。OLAP では、大量のデータを

まとめ、集約化した情報を出力する処理を行う。処理の過程において、テーブルスキャン、結合演算、集約演算、ソート等のデータベース処理が何段階にも組み合わせられて実行される。またもう 1 つの特徴として、OLAP は蓄積されたデータの分析を主な目的としているため、データアクセスのほとんどが読み込みに偏っていることが挙げられる。

3.1.2 TPC ベンチマーク

Transaction Processing Council[4] はデータベース関連企業により運営される非営利団体であり、データベースシステムの評価を行うためのベンチマークを提供している。これらは、データベース分野における標準ベンチマークとして位置付けられている。TPC の提供するベンチマークは、実際に使われているシステムに近い負荷を生成することにより、データベースシステムの実践的な評価を行うことを目的とする。現在は、OLTP システム向けの TPC-C、TPC-E、そして OLAP 向けの TPC-H が提供されている。それぞれの TPC ベンチマークに関しては、実際に測定が行われたシステムのランキングが公開されており、システム構成やソフトウェアの実装が全て公開されている。

TPC-H は OLAP システムの評価を目的としたもので、大型基幹業務向け意思決定支援システムにおけるデータベース構成や、問い合わせをシミュレートして性能を評価するベンチマークである。図 3.3 に、TPC-H のデータベースのスキーマを示す。図中では、各枠内一段目がテーブル名、二段目がタプル数である。スケールファクタ (SF) は、データベースサイズを決定するパラメタである。テーブル間の矢印による接続は、外部キー制約によるテーブル同士の参照関係を表している。図 3.3 のデータベースに対して、複雑なデータ解析処理を行うアドホックな問い合わせが複数定義されており、これらを用いて OLAP システムを評価する。

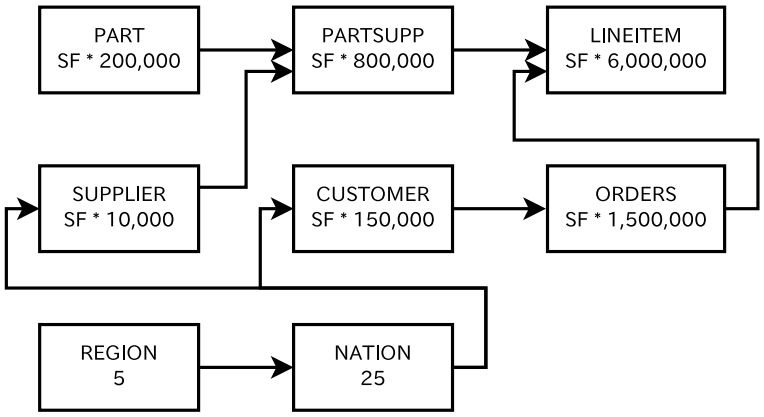


図 3.3: TPC-H ベンチマークのテーブルとレコード数の関係

3.1.3 SSD を用いた関係データベースシステム

OLAP システムの問い合わせ処理では、蓄積されたデータの集計及び分析という処理の性質上、データベースの大部分を読み込む必要がある。大規模データベースにおいては、全てのデータがメモリには収まらず、ストレージアクセスが頻発する。HDD を使用した関係データベースシステムでは、ストレージアクセスが非常に低速なので、I/O がボトルネックとなる場合が多い。そのため、SSD の使用によって、ストレージアクセスが高速化することで、OLAP システムの問い合わせの処理性能を向上することが期待される。

ただし、SSD は、HDD と比較するとランダム I/O スループットが高く、並列 I/O 処理が可能といった I/O 特性があるため、関係データベースシステムにおいて単純に HDD を置き換えるだけでは、SSD を用いる利点を十分に活かすことができない。また、DRAM と比較しても、過度なフラグメンテーションによりスループットが低下するという性質があるため、SSD によって単純に主記憶を拡張するという使用法も不十分である。

本論文では、SSD をストレージとして使用した関係データベースシステムにおいて、I/O スループットの向上のみでなく、I/O ボトルネックが解消されることによって、新たに生じる CPU 演算やメモリアクセスのボトルネックにも着目し、システ

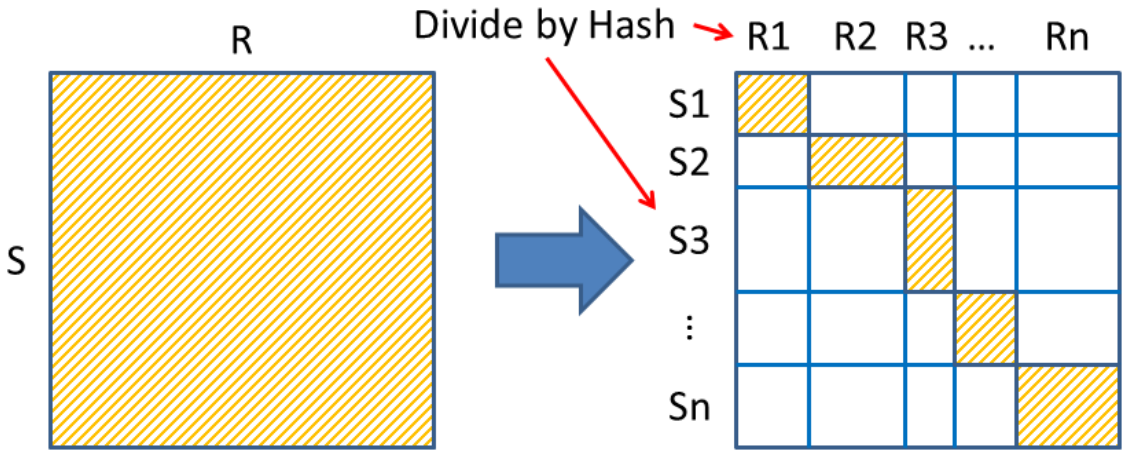


図 3.4: ハッシュ結合演算によるタプルのマッチング

ム全体の処理速度を高速化する。

3.2 ハッシュ結合演算

結合演算は、関係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り、関係データベースシステムの全体の処理の中で大きな割合を占める。そのため、関係データベースシステムの処理を高速化するにあたって、結合演算の高速化は根幹となる問題である。

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関係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。

ハッシュ結合演算の基本的な考え方は、結合対象のリレーションをハッシュ関数によって分割し、ハッシュ値がマッチするタプル同士のみについて実際にマッチングを行うことで、マッチングの回数を低減するというものである。

図 3.4 は、ハッシュ結合演算による、タプル同士のマッチングの回数の削減効果を示している。図中の R や S はリレーションであり、図の左側は入れ子ループによる結合演算処理時のマッチング回数、右側はハッシュ結合演算処理時のマッチング回

数を、それぞれ黄色の斜線部で表している。$N(X)$ をリレーションやハッシュ分割後のリレーションの小片のタプル数とすると、結合演算処理時のマッチングの回数は、入れ子ループ結合演算では $N(R) \times N(S)$、ハッシュ結合演算では $\sum N(R_i) \times N(S_i)$ となる。

結合対象のリレーションが大規模な場合、片方のリレーションをキャッシュやメモリに載せておき、高速に繰り返し参照することができない、つまりデータアクセスの局所性が低いため、マッチング処理のコストが高くなる。そのため、あらかじめハッシュを用いて分割しておくことにより、入れ子ループ結合演算と比較して、無駄なマッチング処理を削減し、さらにデータアクセスの局所性を高めることができるため、処理が高速化される。これが大規模データを扱う問い合わせで、ハッシュ結合演算が多用される理由である。

以下では、ハッシュ結合演算アルゴリズムとその処理コストについて説明する。

3.2.1 Grace ハッシュ結合 [16]

説明の便宜のため、リレーション R と S の結合演算で、R 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 R 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。

図 3.5 は、Grace ハッシュ結合の処理フローを示している。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、R_i についてハッシュテーブルを作成する。続けて S_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。

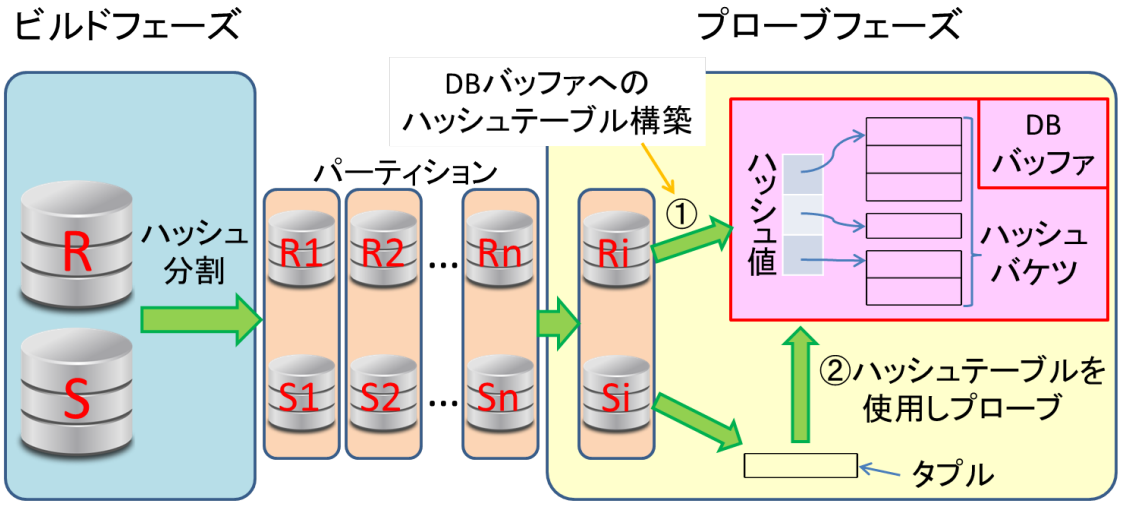


図 3.5: Grace ハッシュ結合の処理フロー

3.2.2 ハイブリッドハッシュ結合 [27]

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 R の 1 つ目のパーティション R_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 R_1 のパーティションと、それと同じハッシュ値を持つ S_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。

3.2.3 ハッシュ結合演算の処理コスト

HDD の使用を前提とした関係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(ワーキングメモリ) の使用可能量によって決まる。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、R の全てのパーティションのハッシュテーブルがワーキングメモリに収まるように決定されるためである。そのため、ワーキングメモリが小さいときは、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、大きなワーキングメモリが必要とされる。

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[1]が生じていることを意味する。ハッシュテーブルサイズはワーキングメモリサイズを越えないように決定されるので、ワーキングメモリサイズによって、ハッシュテーブルのデータアクセスの局所性が規定されることになる。HDD を使用する場合、パーティションファイルの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抑えるため、ワーキングメモリをある程度大きく設定する必要があったため、このハッシュテーブルのデータアクセス局所性に関してはメモリレベルで考慮されることが多い。

3.2.4 複数ハッシュ結合演算の処理

複数演算を組み合わせた問い合わせでは、個々の演算を逐次に実行する場合、蓄積される中間生成データをストレージに書き戻す必要がある。ソートやハッシュテーブルの作成といったブロッキング演算を除けば、後続の演算を実行するために全ての結果データを揃える必要はなく、演算の結果データを後続の演算に渡し、即座に処理することで、中間生成データをストレージに書き戻す処理を省略することができる。これを演算のパイプライン処理と呼ぶ。パイプライン処理により、低速な I/O

[1] データアクセス局所性とは、データアクセスが全体のデータのうちの特定の一部分に集中する性質のことを指す。以下では、メモリ空間の範囲でデータデータアクセス局所性があることをメモリレベルのデータアクセス局所性などと呼ぶことにする。

を削減することが可能である。

複数のハッシュ結合演算がアウターループ側に連結している場合、パイプライン処理が可能であるが、このとき、それぞれの演算で使用するハッシュテーブルを保持するため、複数のハッシュテーブルが同時に存在することになる。それぞれのハッシュ結合演算は、前の演算から結果を受け取る度にタプルのマッチングを行うため、それぞれのハッシュテーブルは、交互にアクセスされる。そのため、全てのハッシュテーブルに対してデータアクセスの局所性を保つ必要がある。これは、全てのハッシュテーブルの合計サイズでデータアクセスの局所性を管理する必要があるということである。

このように、問い合わせ処理においては、含まれるハッシュ結合演算の数に応じて、ワーキングメモリサイズを調節し、ハッシュテーブルへのデータアクセスの局所性を保たなければならない。

3.2.5 SSD を用いたハッシュ結合演算

ハッシュ結合演算では、パーティションファイルの読み込みや書き込みのために、ストレージアクセスが多発する。そのため、SSD の使用により、I/O スループットが増加し、処理性能が向上することが期待される。

I/O スループットが増加することで、I/O ボトルネックが解消され、メモリアクセスコストなどの CPU コストが新たなボトルネックとして顕在化することが考えられる。このような場合、SSD を用いたハッシュ結合演算では、キャッシュミス数を低減し、CPU コストについても改善することが可能である。

HDD を使用したハッシュ結合演算では、ワーキングメモリサイズを大きくする必要があり、メモリレベルでしかハッシュテーブルのデータアクセスの局所性を扱うことができなかった。一方、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く、HDD 使用時よりワーキングメモリサイズを縮小することができる。これにより、ハッシュテーブルサイズが縮小し、より細かい粒度でハッシュテーブ

ルのデータアクセスの局所性を管理することが可能になる。ハッシュテーブルサイズをキャッシュサイズより小さく設定することで、ハッシュテーブル参照時のハッシュテーブルへのアクセスでキャッシュミスが発生せず、CPU コストが低減される。

また、ハッシュテーブルのデータアクセスの局所性がメモリレベルである場合、複数の問い合わせを同時処理すると、キャッシュレベルでのデータアクセスの局所性が低下してしまい、個々の処理速度が低下してしまうため、逐次処理せざるを得なかった。SSD を用いたハッシュ結合演算では、ハッシュテーブルのデータアクセスの局所性を高め、キャッシュレベルでデータベースバッファの管理ができるため、複数問い合わせの同時処理においても、キャッシュを分割して使用することで、データアクセスの局所性を高く保つことが可能である。これを利用することで、SSD を使用した関係データベースシステムにおける複数問い合わせ処理において、同時実行数やワーキングメモリサイズを適切に設定することで、全体の処理性能を向上することが期待される。

第4章　関連研究

4.1　SSDを用いた関係データベースシステムの研究動向

Flash ストレージを使用した関係データベースシステムの研究で、初期に見られたのは組み込みシステムを対象としたものが多い。FlashDB[23] は、読み込み中心のワークロードでは Disk モード、書き込み中心のワークロードでは Log モードと使い分ける手法を提案している。B-FTL[31] は、FTL で B+-tree についての最適化を行っている。この手法では、更新データをバッファリングすることにより、更新処理を集約し、低速な書き込み処理を低減している。

近年では、フラッシュの集積技術の向上に伴い、大容量化・低価格化が進み、計算機の二次記憶として使用することを目的とした SSD の普及が進んでいる。これに伴い、SSD を対象とした研究も多く見られるようになった。関係データベースシステムにおいて、SSD を使用によって問い合わせ処理性能の大きな向上が期待できる要素・手法としては、以下のものが考えられる。

- バッファプール拡張
- HDD と SSD のハイブリッドストレージ運用
- インデックス
- ページレイアウト

これらに関する先攻研究について以降の節で述べる。

4.2 バッファプール拡張

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関係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 [7, 8] では、データのアクセスパターンをモニタリングし、アクセス頻度に応じて設定される temperature が高いデータブロックが SSD に保持される。ライトスルー方式を採用しているため、読み込みに対してはキャッシュの効果が出るが、書き込みに対してディスクアクセスを削減する効果は無い。Do らによる Lazy Cleaning (LC) 法 [14] ではライトバック方式を採用しており、書き込み時もディスクアクセス削減の効果がある。キャッシュ追い出しは LRU-2[24] による。LRU-2 では、追い出し時にランダム書き込みが多発してしまう点が問題となる。Kang らの提案する FaCE システム [15] 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。また、キャッシュ追い出しの集約や Second Chance によってもスループットを向上させている。Liu らの研究 [19] では、SSD と HDD のアクセスコストが異なることを考慮したキャッシュ置換アルゴリズム GD2L を提案している。GD2L は Dual Greedy[21] をベースにしており、アクセスコストが小さい SSD に格納されているデータを追い出しやすくすることで処理全体の I/O コストを抑えている。

バッファプール拡張は、既にいくつかの商用関係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[30]、IBM XIV Storage System[1] などが挙げられる。

SSD は不揮発性のストレージデバイスであるので、SSD にキャッシュされたデータは永続化される。この効果によって、バッファプール拡張では HDD のみを使用する場合と比較して、永続化のオーバーヘッドの削減やデータベース障害時の復旧時間の短縮が見込める。キャッシュ置換アルゴリズムの他に、障害回復手法についても、[7, 15, 12, 19] など多くの研究が見られる。

[14] で述べられているように、バッファプール拡張で用いられている、モニタリングをベースとして、データアクセスの局所性を推定する手法は主に 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。

4.3 ハイブリッドストレージ

現在の SSD は、容量や価格の点では HDD と比較すると劣る。データウェアハウスなど非常に大容量のデータを扱う必要のあるシステムでは、SSD と HDD の双方を使用し、用途によって使い分けるというのが現実的である。SSD と HDD が共存するハイブリッドなストレージ環境では、アクセス頻度が高い、もしくは高い I/O スループットが要求されるデータを選別し、SSD 内に置くといったデータ管理が必要である。

単純には、あらかじめワークロードを想定して、静的に SSD に配置するデータを決めてしまうといった方法が考えられる。ただし、静的な割り当てでは、小さな粒度でデータを分割し、より効率的なデータの配置を行うことはデータベース管理者の負担が大きく現実的には難しい。また、同じデータに対してでも異なるアクセスパターンが生じることもあるが、静的な割り当てではこれを同様に扱わざるを得ない。動的なワークロードに対応できないのも問題である。

長時間稼働するシステムであれば、I/O のモニタリングにより、アクセスパターンを解析して動的にローカリティの高いデータを選別する手法が効果的であり、Teradata Virtual Storage System [32] などの商用データウェアハウスにも組み込まれている。Koltsidas らの研究 [17] では、モニタリングにより、ページ毎のワークロードを特定し、読み込みが多いページを SSD に、書き込みが多いページを HDD に配置する方法を提案している。CAC[19] では、キャッシュの使用効率も考慮に入れたデータ配置を行う。

モニタリングによる手法は、安定した解析結果が得られるまで一定の時間が必要であること、モニタリングのオーバーヘッドが生じることなどが問題点となる。hStorage-DB[20] では、問い合わせに関するセマンティック情報を利用したデータ管

理を行っている。関係データベースシステムは、問い合わせ処理最適化機構や問い合わせ実行エンジンなど様々な要素から成り、問い合わせ中の演算の実行順序やデータアクセスパターン、処理中のデータのライフサイクルなどのワークロードに関する情報を内包する。これらの情報を問い合わせの実行に前もって取得することで、SSD と HDD に対して効率的なデータ配置を実現している。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といった長所がある。アドホックな問い合わせ処理が行われる、OLAP システムでは、ワークロードについて、モニタリングによる解析ができないので、このように関係データベースシステムの持つ情報を利用することが不可欠である。

4.4 インデックス

DBMS で扱う典型的なデータ構造の 1 つとして、B+-tree インデックスが挙げられる。B+-tree インデックスのアクセスではランダムアクセスが発生しやすいので、SSD の使用により大きくアクセス速度を向上できると考えられる。

FD-tree[18] は、木構造のルート周辺のノードを主記憶上に保持し、挿入されたデータをその部分にバッファリングすることで、インデックスの更新処理を集約化し、ランダム書き込みを削減している。FD-tree の問題点としては、B+-tree と比較して木が高くなってしまうため、検索性能が低下することが挙げられる。PIO B-tree[15] では、SSD の internal parallelism に着目した最適化を行っている。レンジサーチの並列化や挿入データのバッファリングによって、並列アクセスを実現し、スループットを向上させるものである。

4.5 ページレイアウト

Tsirogiannis らの研究 [29] では、SSD の高速なランダム I/O 性能に着目し、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。

第5章　SSDを用いた結合演算処理とその性能評価

本章では、実際のSSDを用いた関係データベースシステムを使用し、単一ハッシュ結合演算や複数ハッシュ結合演算の処理性能について詳細に解析する。SSDの高いI/Oスループットによって、I/Oコストが低減することを確認し、HDDの使用を前提として設計された関係データベースシステムにおけるハッシュ結合演算の処理方式が、SSDを使用する場合でも適しているか検討する。

5.1　SSDを用いた関係データベースシステムの実験環境

計測には、表2.2の計算機環境を用いる。関係データベースシステムとしてPostgreSQL[3]を使用する。データベースの構築には、TPC-Hベンチマークのデータを Scale Factor = 100で用い、SSDと、比較のためHDD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112 GBである。

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 プロファイラ perf[2] をそれぞれ使用して取得した。

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。ワーキングメモリが小さいとき、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、ワーキングメモリサイズを大きくとることが多い。ワーキングメモリ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

```
1  SELECT
2      COUNT(*)
3  FROM
4      lineitem, part
5  WHERE
6      l_partkey = p_partkey
```

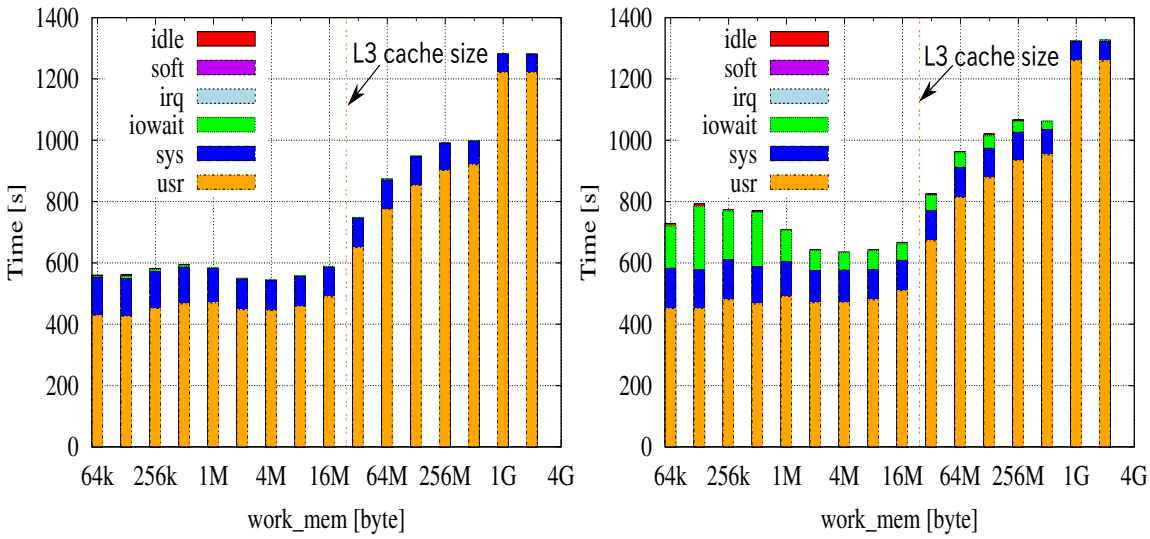
図 5.1: lineitem 表と part 表の結合演算を行う問い合わせ

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 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インメモリバッファサイズを示す。これは、ハッシュ結合では、ワーキングメモリサイズに相当する。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。

5.2 単一のハッシュ結合演算を含む問い合わせの処理性能

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。問い合わせは、図 5.1 を用いる。この問い合わせでは、lineitem 表のそれぞれのタプルについて、part 表の partkey 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 20 GB と 80 GB 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 800 MB であった。work_mem を、64 kB - 2 GB 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 5.2 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



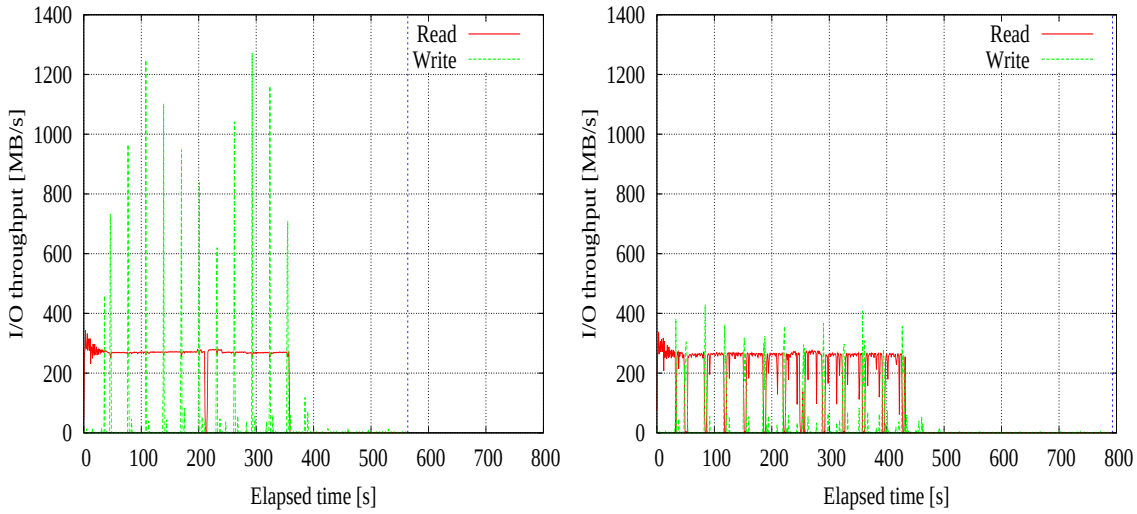
(a) SSD　　(b) HDD

図 5.2: 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

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。

図 5.3 は、work_mem = 128 kB の設定における、問い合わせ実行時の I/O スループットのタイムラインを SSD と HDD それぞれについて示したものである。各図では、青破線までが結合演算の実行時間を示している。図 5.3a の 0 - 360 秒区間、図 5.3b の 0 - 430 秒区間は、ハッシュ結合演算のビルドフェーズ部分 (1 つ目のパーティションのプローブ処理も含む) を示しており、読み込み I/O が始めの 40 秒程度は part 表、その後は lineitem 表の読み込み処理、書き込み I/O がパーティションの書き込み処理に相当する。図 5.3b を参照すると、HDD 上での実行では、読み込み I/O ス



(a) SSD　　(b) HDD

図 5.3: work_mem = 128 kB のときの図 5.1 の問い合わせ実行時の I/O スループットのタイムライン

ループットが時折、パーティションの書き込み I/O の発生によって低下している。SSD では、並列 I/O 処理が可能なので、読み込み I/O と書き込み I/O の処理を同時に行うことで、読み込み I/O スループットの低下を防ぐことができ、HDD 上の場合と比較して 70 秒ビルドフェーズの処理時間が短かくなっている。[1]ビルドフェーズ終了後から処理終了までの、プローブフェーズの処理時間に関しては、HDD 上での実行では、パーティションファイルのフラグメンテーションの影響で、I/O スループットが低くなっており、SSD 上での実行が 160 秒高速である。SSD 用いたことによる、並列 I/O 処理とランダム I/O スループットの向上によって、ハッシュ結合演算の処理が高速化している。

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、

[1]図 5.3a の 210 秒部分で、一瞬スループットが大きく低下しているが、これは OS のスワップ領域管理プロセスによるものである。読み取りづらいが、図 5.3b の図でも同様の現象は見られる。

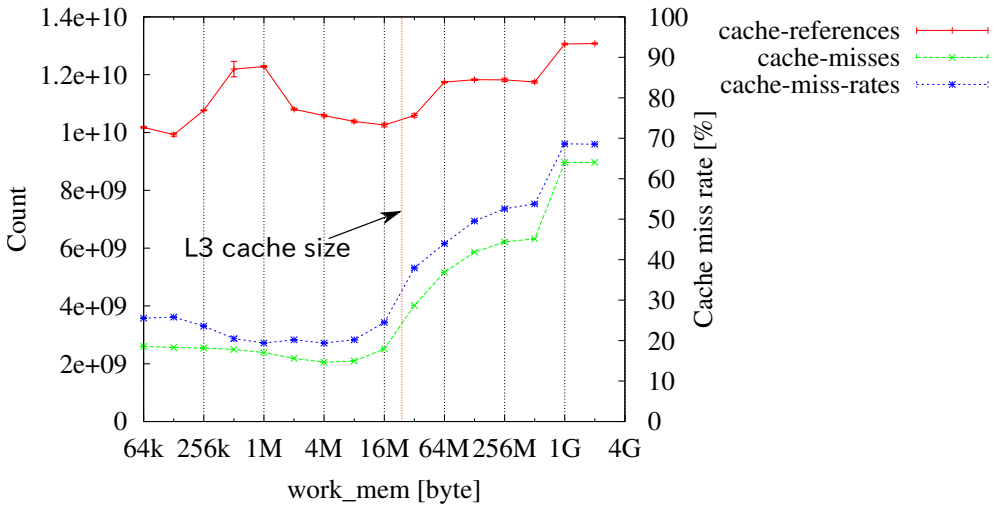


図 5.4: 図 5.1 の問い合わせ処理時の各 work_mem の値における L3 キャッシュ参照・ミス数及びミス率 (SSD 上の実行)

キャッシュミス数が増加しているためである。

図 5.4 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` > 32MB 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` = 4 MB と 1 GB で比較する。キャッシュミス数は `work_mem` = 1 GB 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` = 1 GB の方が $7 \times 10^9 \times 100(\mathrm{ns}) = 700(\mathrm{s})$ 程度 I/O コストが大きくなる。これは図 5.2a 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 800 MB) 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 512 MB から 1 GB 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` = 1 GB 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 64 kB のとき 2.2、128 kB のとき 3.8、256 kB - 512 MB のとき 5.7、1 GB 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、1 GB のときのハッシュバケツのタプルスキャン数は 512 MB のときより $(10.5 - 5.7) \times 6 \times 10^8 \approx 3 \times 10^9$ 多くなり、図 5.4 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。

5.2.1 ハッシュ結合演算におけるデータアクセス局所性

ワーキングメモリサイズが小さいとき、HDD 上での実行では、図 5.2b の結果の通り、プローブフェーズにおいて、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 スループットが小さくなり、I/O コストが増加してしまう。一方、SSD 上での実行では、図 5.2a の結果の通り、I/O スループットが HDD 使用時より高いため、I/O コストが小さい。また、ワーキングメモリサイズによらず I/O コストはほぼ一定の大きさである。

このように、SSD は高いランダム I/O スループットを持つため、ハッシュ結合演算の処理で、ワーキングメモリサイズが小さい設定において、I/O コストが低減する。そのため、CPU コストの最適化のみを考慮すればよい。

ハッシュ結合演算の処理において、CPU コストを増加させる要因となるのは、図 5.4 の結果にみられる通り、主にプローブフェーズにおけるハッシュテーブルアクセスで生じる、キャッシュミスペナルティによるメモリアクセスコストである。SSD を用いたハッシュ結合演算では、ハッシュテーブルがキャッシュに収まるサイズになるようにワーキングメモリサイズを調節することで、キャッシュミス数を低減することが可能である。

SSD を用いたハッシュ結合演算の処理性能を最適化するためには、ハッシュテーブルのデータアクセスの局所性をキャッシュサイズレベルで管理することが必要である。

```
1  SELECT
2      COUNT(*)
3  FROM
4      lineitem, part, customer
5  WHERE
6      l_partkey = p_partkey
7      AND l_suppkey = s_suppkey
```

図 5.5: 二つの結合演算を行う問い合わせ

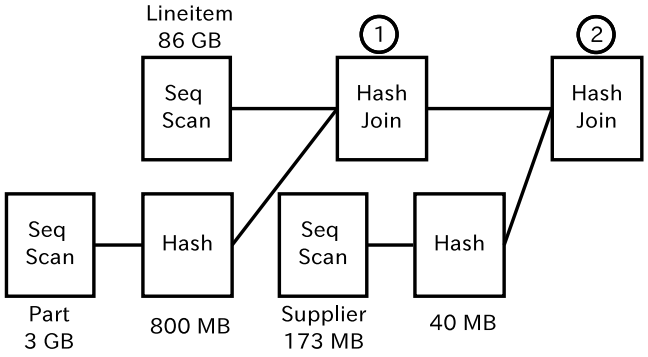
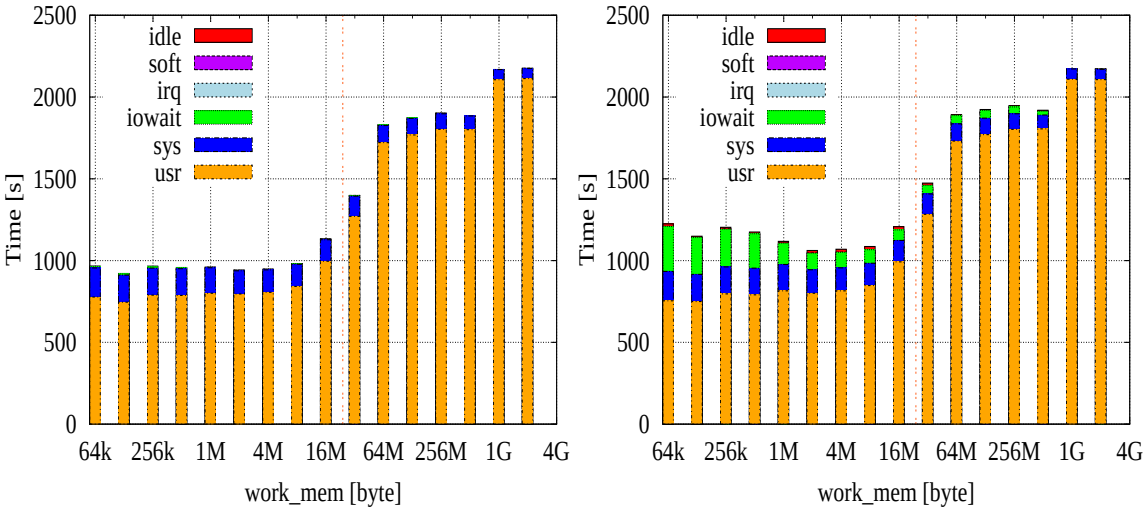


図 5.6: 図 5.5 の問い合わせの実行プラン

5.3 複数のハッシュ結合演算を含む問い合わせの処理性能

複数のハッシュ結合演算を含む問い合わせとして、二つのハッシュ結合演算を含む問い合わせと、TPC-H query 8 について計測を行う。計算機やデータベースの環境は前節と同様のものを用いる。

二つのハッシュ結合を含む問い合わせ　複数ハッシュ結合演算のパイプライン処理によって、複数のハッシュテーブルが同時に存在する場合の処理性能について分析するため、まず、二つのハッシュ結合を含む問い合わせについて計測する。問い合わせは、図 5.5 を用い、実行プランは図 5.6 の通りである。



(a) SSD　　(b) HDD

図 5.7: 二つ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

図 5.6 のプローブフェーズの処理の流れを追うと、まず、supplier 表、part 表、lineitem 表の順番でパーティション分割が行われる。その後、lineitem 表のタプルが一つ取り出され、①のハッシュ結合演算で、part 表に対し構築されたハッシュテーブルを使用してマッチングを行う。そこでマッチするタプルの組があれば、それを結合し、結果を次の②のハッシュ結合演算にそのまま渡す。②のハッシュ結合演算では、supplier 表に対し構築されたハッシュテーブルを使用し、マッチングを行い結果を出力する。

このように、①と②のハッシュ結合演算が交互に処理されるため、同時に二つのハッシュテーブルが存在することになる。

図 5.7 は、SSD と HDD それぞれにおける計測結果を示しており、図 5.8 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。図 5.7 では、`work_mem` = 16 MB から CPU コストが増加しており、図 5.8 を参照すると、L3 キャッシュミス数の増加が原因となっている。L3 キャッシュサイズは 24 MB であるので、全てのハッシュテーブルの合計サイズが L3 キャッシュサ

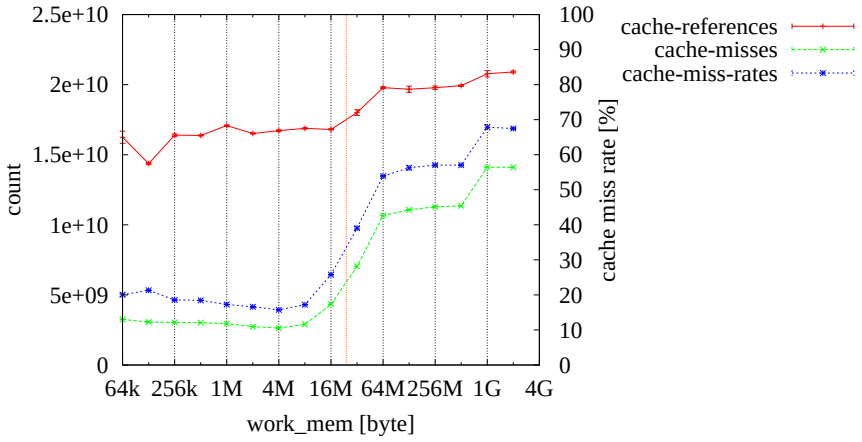
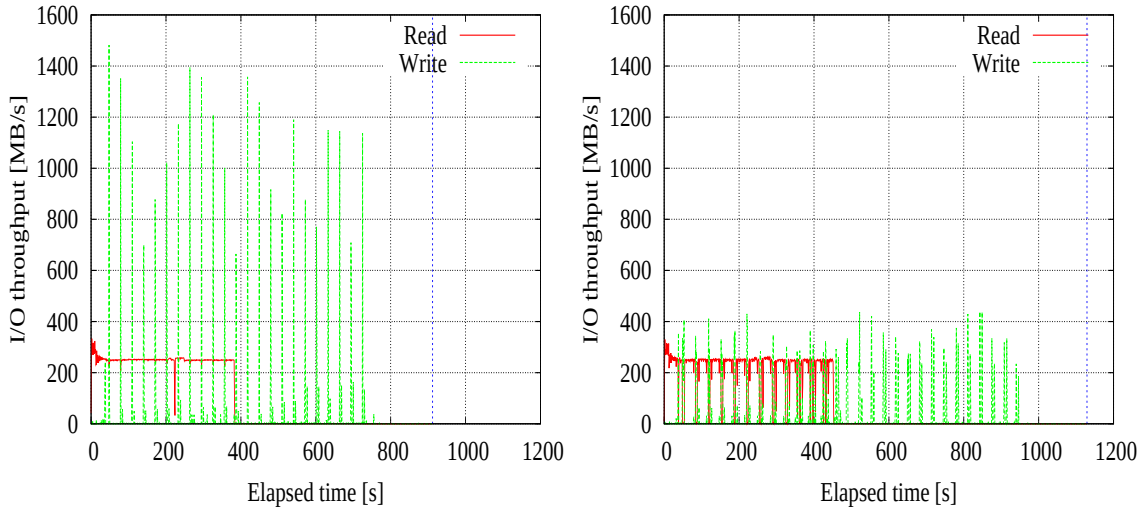


図 5.8: 図 5.5 の問い合わせ処理時の各 work_mem の値における L3 キャッシュ参照・ミス数及びミス率 (SSD 上での実行)



(a) SSD　　(b) HDD

図 5.9: work_mem = 128 kB のときの図 5.5 の問い合わせ実行時の I/O スループットのタイムライン

イズを越える 24/2 = 12MB を越えた点から、L3 キャッシュミス数が増加している。

I/O コストに関しては、図 5.7 では、図 5.2 の単一ハッシュ結合演算実行時と比較

して同程度である。図 5.9 は、work_mem = 128 kB の設定における、図 5.5 の問い合わせ実行時の I/O スループットのタイムラインを SSD と HDD それぞれについて示したものである。

図 5.9a の 40 - 390 秒区間、図 5.9b の 40 - 450 秒区間は、図 5.6 の①のハッシュ結合のビルドフェーズの lineitem 表のスキャン処理の部分である。SSD 上での実行時では、HDD 上の場合と比較して 60 秒ビルドフェーズの処理時間が高速である。また、それぞれ lineitem 表のスキャン後は、part 表、supplier 表それぞれのハッシュテーブルを用いてタプルのマッチングが行われる。HDD 上での実行では、パーティションファイルのフラグメンテーションの影響で、I/O スループットが低くなっており、SSD 上での実行が 180 秒高速である。

TPC-H query 8　複数ハッシュ結合を含む問い合わせで、より複雑な処理を行うものの例として、TPC-H query 8 について計測する。TPC-H query 8 は、8 個のテーブルの結合演算を含む問い合わせである。I/O 性能の計測が主眼であるので、元の問い合わせから計算部分を除去したものを扱う。問い合わせは図 5.1 の通りであり、実行プランは図 5.11 の通りである。図 5.11 中のそれぞれのハッシュ結合では、下側に置かれたノードのリレーションに対してハッシュテーブルが生成される。問い合わせ中で最も負荷が高いのは、図 5.11 中の赤線枠内であり、ハッシュテーブルの合計サイズは約 400 MB であった。

図 5.12 は、SSD と HDD それぞれにおける計測結果を示しており、図 5.13 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。図 5.12 では、work_mem = 8 MB から CPU コストが増加しており、図 5.13 を参照すると、キャッシュミス数の増加が原因となっている。図 5.11 の実行プランでは、3 つのハッシュ結合がパイプライン処理される部分 (青線枠内) が存在するため、メモリには 3 つのハッシュテーブルが共存する。全てのハッシュテーブルを L3 キャッシュに収めるためには、work_mem $< 24/3 = 8$MB でなければならず、それ以上ではキャッシュミス数が増加する。

I/O コストに関しては、図 5.12 では、図 5.2 の単一ハッシュ結合演算実行時と比

```
1  SELECT
2      EXTRACT(year FROM o_orderdate) AS o_year,
3      l_extendedprice * (1 - l_discount) AS volume,
4      n2.n_name AS nation
5  FROM
6      part,
7      supplier,
8      lineitem,
9      orders,
10     customer,
11     nation n1,
12     nation n2,
13     region
14 WHERE
15     p_partkey = l_partkey
16     AND s_suppkey = l_suppkey
17     AND l_orderkey = o_orderkey
18     AND o_custkey = c_custkey
19     AND c_nationkey = n1.n_nationkey
20     AND n1.n_regionkey = r_regionkey
21     AND r_name = 'AMERICA'
22     AND s_nationkey = n2.n_nationkey
23     AND o_orderdate BETWEEN
24         DATE '1995-01-01' AND DATE '1996-12-31'
25     AND p_type = 'ECONOMY ANODIZED STEEL'
```

図 5.10: TPC-H query 8 (一部の計算部分削除)

較すると、特に `work_mem` が小さいときに、SSD と HDD の I/O コスト差が顕著に現れている。図 5.14 は、`work_mem` = 128 kB の設定における、TPC-H query 8 実行時の I/O スループットのタイムラインを SSD と HDD それぞれについて示したものである。

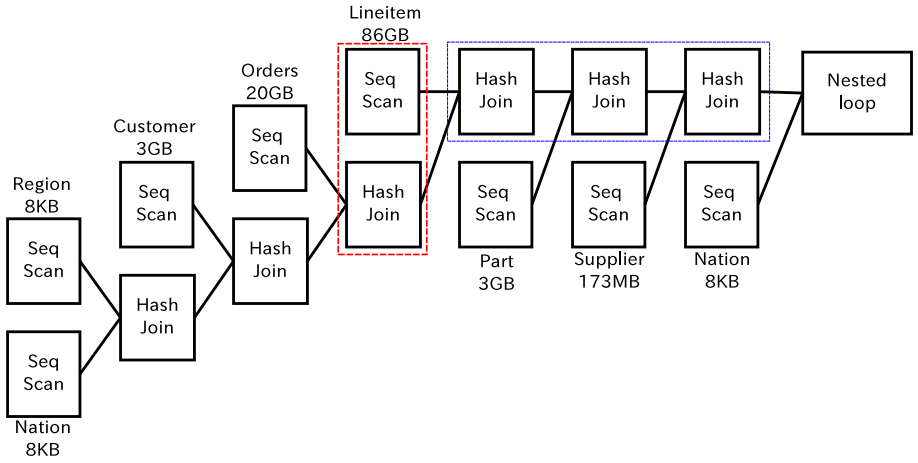
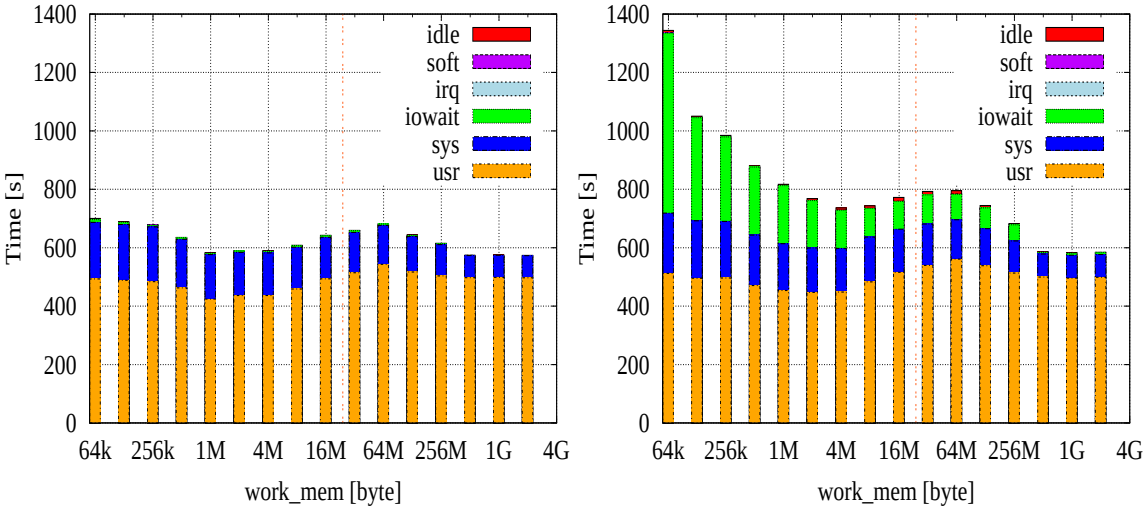


図 5.11: TPC-H query 8 実行プラン



(a) SSD (b) HDD

図 5.12: TPC-H query 8 の各 work_mem の値における実行時間

図 5.14a の 90 - 550 秒区間、図 5.14b の 100 - 700 秒区間は、図 5.11 の赤線枠内のハッシュ結合のビルドフェーズの lineitem 表のスキャン処理の部分である。SSD 上での実行時では、HDD 上の場合と比較して 150 秒ビルドフェーズの処理時間が高速である。また、図 5.14a の 550 - 680 秒区間、図 5.14b の 700 - 1050 秒区間は、図

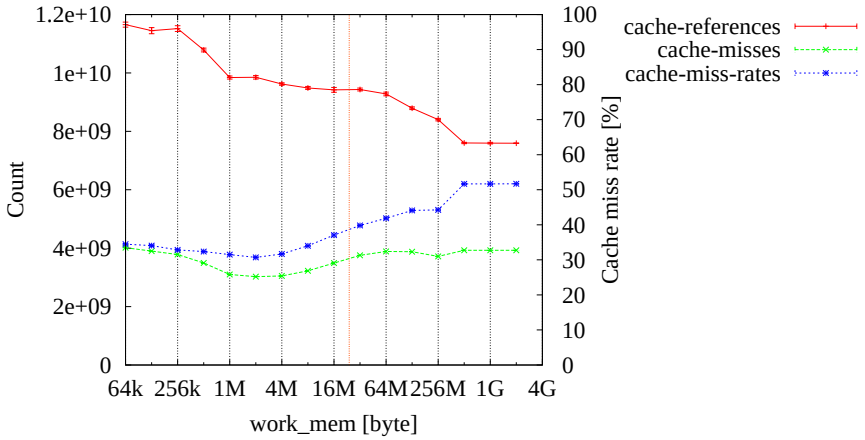
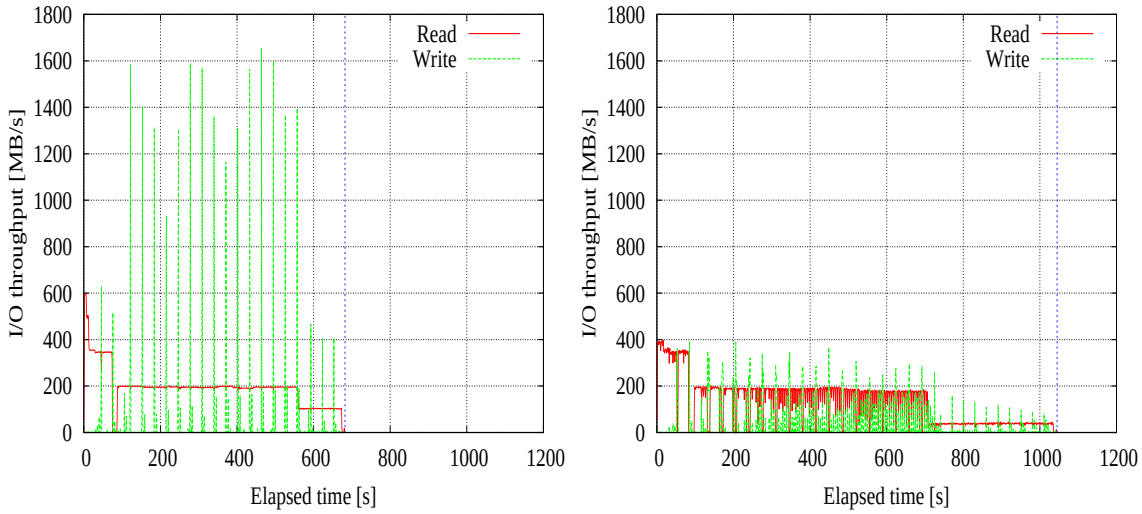


図 5.13: 図 5.10 の問い合わせ処理時の各 work_mem の値における L3 キャッシュ参照・ミス数及びミス率 (SSD 上での実行)



(a) SSD (b) HDD

図 5.14: `work_mem` = 128 kB のときの TPC-H query 8 実行時の I/O スループットのタイムライン

5.11 の赤線枠内のハッシュ結合のプローブフェーズ部分である。HDD 上での実行では、パーティションファイルのフラグメンテーションの影響で、I/O スループット

が低くなっており、SSD 上での実行が 330 秒高速である。

リレーションのデータ局所性　図 5.2 や図 5.7 のハッシュ結合の計測で得られた結果では、ワーキングメモリサイズによって全てのハッシュテーブルの合計サイズが L3 キャッシュサイズより大きいときは、キャッシュミス数の増加により CPU コストが増加しており、3.2.3 節や 5.3.1 節で説明した挙動に一致している。しかしながら、図 5.13 の TPC-H query 8 のキャッシュミス計測の結果では、全てのハッシュテーブルが L3 キャッシュに収まらなくなる、`work_mem` = 8 MB から L3 キャッシュミス数が増加し始めているものの、図 5.4 ではキャッシュミス数が 69% まで増加しているのに対し、53% までしか増加しておらず、そのため CPU コストも大きく増加していない。これは、結合演算で扱われるリレーション間で高いデータ局所性が生じていることにに起因する現象であった。

図 5.10 の問い合わせで最も処理が重い、図 5.11 の赤線枠内のハッシュ結合の結合条件は、`orders.orderkey` = `lineitem.orderkey` である。実行プランでアウターループ側にある、lineitem 表では結合キーとなる `orderkey` について、同じ値を持つタプルが平均で 4 個存在している。lineitem 表ではタプルは `orderkey` に対して昇順で格納されていたため、同じ `orderkey` を持つタプルは 1 箇所に集中している。そのため、ハッシュ結合のプローブフェーズにおいても、アウターループで同じ `orderkey` を持つタプルが連続して出現し、ハッシュテーブルの同じバケツが連続して参照され、キャッシュがヒットする。ハッシュテーブル全体がキャッシュに収まっていなくても、キャッシュミスが生じるのは、各 `orderkey` の値について最初の 1 タプルのときだけであるので、プローブフェーズのキャッシュミス率は $100/4 = 25\%$ を越えることはない。このようなリレーション間のデータ局所性のために、図 5.10 の問い合わせでは、CPU コストの変化が小さくなっている。CPU コストが支配的な状況では、データ自体が持つ局所性も全体の処理性能に大きな影響を及ぼすということである。

5.3.1 複数ハッシュ結合処理時のデータアクセス局所性

複数のハッシュ結合演算を含む問い合わせでは、図 5.7 や図 5.12 の結果の通り、図 5.2 の単一ハッシュ結合演算を含む問い合わせより、小さいワーキングメモリサイズから L3 キャッシュミス数が増加し、CPU コストが増加している。これは、パイプライン処理によって複数のハッシュテーブルが同時に存在するため、全てのハッシュテーブルが L3 キャッシュに収まっていないと、互いのハッシュテーブルへのアクセスによって、L3 キャッシュ上に存在する他のハッシュテーブルを追い出してしまうからである。このように、ハッシュ結合演算では、ハッシュテーブルのデータアクセスの局所性は、節で説明した通り、問い合わせに含まれる全てのハッシュ結合演算のハッシュテーブルの合計サイズで管理しなければならない。

SSD を用いたハッシュ結合演算では、図 5.7、5.12、5.2 の結果の通り、全てのハッシュテーブルの合計サイズが L3 キャッシュサイズの 24 MB より小さくても I/O コストが増加していない。そのため、ハッシュテーブル数に応じて、同時に存在する全てのハッシュテーブルが L3 キャッシュに収まるような、ワーキングメモリサイズの選択が可能である。

5.4 キャッシュサイズを考慮したハッシュ結合演算処理方式

従来の HDD を用いたハッシュ結合演算処理では、ワーキングメモリサイズを小さくすると、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 スループットが小さくなり、I/O コストが増加してしまう。そのため、ワーキングメモリサイズをメモリ領域を目一杯使うような大きさに設定していた。つまり、図 5.15 のように、メモリレベルのデータベースバッファ管理を行っていたということである。

一方、SSD を用いたハッシュ結合演算では計測の結果、I/O 処理コストが減少し、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。このような条件では、図 5.15 のような HDD の特性に基づいた処理方式では、SSD を

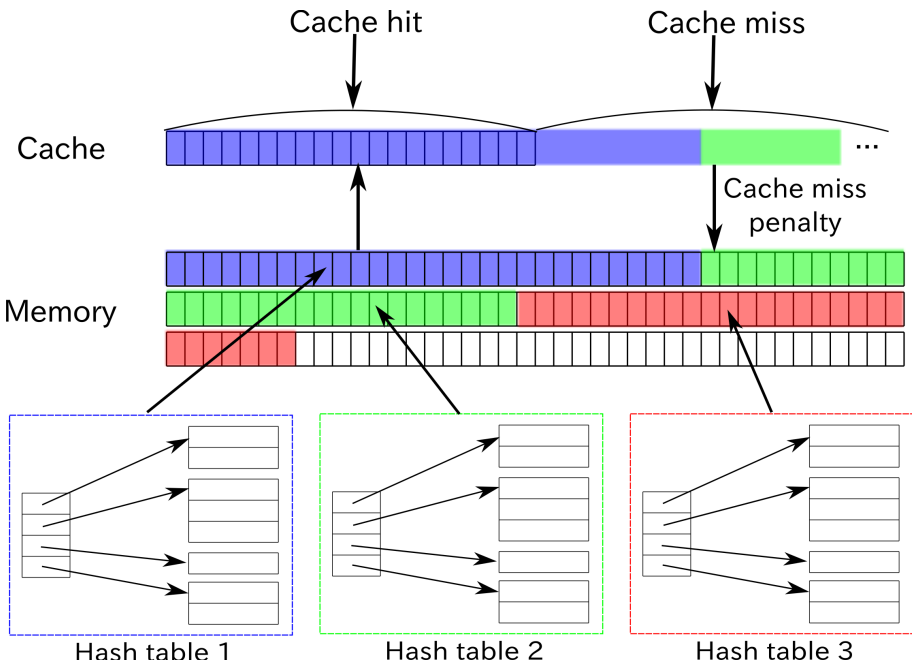


図 5.15: HDD の特性に基づいて設計された従来の関係データベースシステムの処理方式

用いる利点を十分に活かすことができない。

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るため、ハッシュテーブルの局所性をキャッシュレベルに高めることで、キャッシュミス数を低減す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、ワーキングメモリサイズを小さくしても I/O コストが小さい。そのため、より細かい粒度で、ハッシュテーブルのデータアクセスの局所性を調節することが可能である。これらの特性を利用し、SSD を用いた関係データベースシステムに適した問い合わせ処理方式として、データベースバッファをキャッシュレベルで管理する手法を提案する。

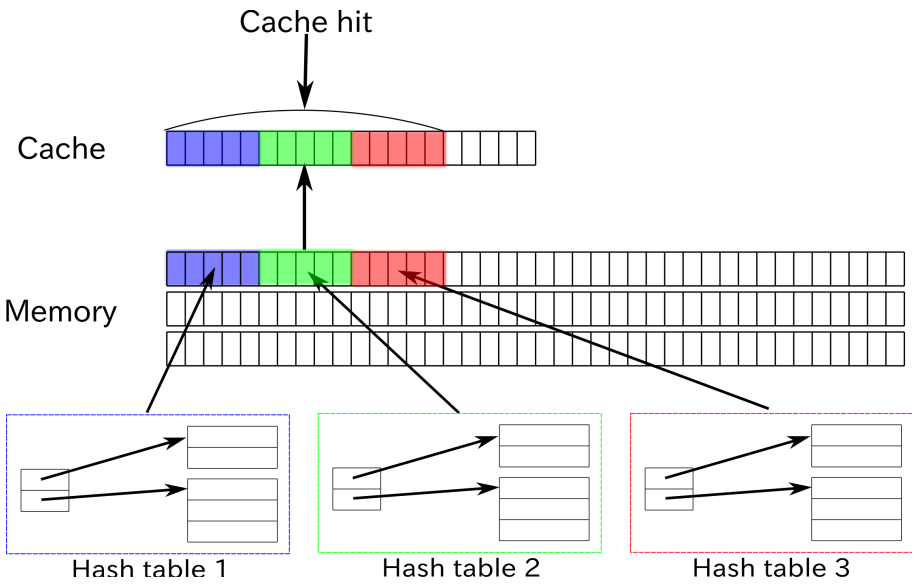


図 5.16: 提案手法:キャッシュレベルでデータベースバッファを管理する処理方式

メモリレベルでデータベースバッファを管理する場合、システムの問い合わせの処理時において、図 5.15 のように、同時に存在する全てのハッシュテーブルの合計サイズがメモリサイズより小さくなるように、個々のハッシュテーブルサイズを調節する。キャッシュレベルでデータベースバッファを管理する場合は、図 5.16 のように、全てのハッシュテーブルの合計サイズがキャッシュサイズより小さくなるように、個々のハッシュテーブルサイズを調節する。

問い合わせ処理時に同時に存在するハッシュテーブル数は、図 5.6 や図 5.11 にみられるような問い合わせの実行プランによって、事前に計算できる。同時に存在するハッシュテーブル数でキャッシュサイズを割り、その値以下にワーキングメモリサイズを設定する。これによって、問い合わせ処理で使用される全てのハッシュテーブルはキャッシュに収まるため、ハッシュ結合演算のプローブフェーズにおけるタプルのマッチングの際のハッシュテーブルアクセスで、キャッシュミスが発生しなくなり、処理全体の CPU コストが低減される。

ただし、ハッシュテーブルサイズは際限なく縮小できるわけではない。SSD においても、図 2.2 や図 2.4 の結果の通り、I/O のフラグメンテーションにより、スルー

プットが低下するため、I/O ボトルネックを引き起さないように、可能な限りデータベースバッファサイズを大きくするのががよい。

キャッシュレベルでデータベースバッファを管理する処理方式は、単一の問い合わせにおいてキャッシュミス数を低減する効果以外にも利点がある。メモリレベルでデータベースバッファを管理する処理方式では、単一もしくは少数のハッシュ結合演算によって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、キャッシュが追い出されてしまい、キャッシュミスが増加してしまう。そのため、ハッシュ結合演算に関しては、逐次処理せざるを得なかった。一方、キャッシュレベルでデータベースバッファを管理する処理方式では、問い合わせで使用されるハッシュテーブルは全てキャッシュに収まっている。そのため、キャッシュに未使用の領域があれば、他の処理で使用することができ、複数処理を同時に実行することが可能である。複数処理の同時実行によって、キャッシュ以外にも、CPU コアやメモリ、ストレージ帯域などの計算資源の使用効率を向上し、処理性能を向上することが期待される。

第6章　SSDを用いた複数問い合わせの同時処理

6.1　大規模データを扱う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。SSD は並列 I/O 処理が可能であるため、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において考慮すべき点として、ストレージ、メモリ、CPU 毎の共有キャッシュ等の計算資源が共有される。同時処理する問い合わせのワークロードの組み合わせによっては、互いの処理によってキャッシュ中のデータを追い出し合い、キャッシュミス数が増加してしまったり、同時に I/O が発行されることで、I/O のランダムさが増加し、I/O スループットが低下してしまうなど、計算資源の使用について競合が生じ、反対に処理性能が低下してしまう場合もある。

第 5 章の分析の結果から、SSD を用いたハッシュ結合演算において、処理性能を向上する処理方式として、キャッシュレベルでデータベースバッファを管理する手法を提案した。この手法を利用し、複数の問い合わせに渡ってハッシュテーブルのテータアクセス局所性を考慮し、ワーキングメモリサイズを調節することで、キャッシュミス数を増加させることなく、複数問い合わせの処理が可能になる。

複数問い合わせの同時処理では、データアクセスの局所性と同時に、同時処理数の考慮も必要となる。これは、SSD においても過度な並列数で I/O を発行すると、I/O のフラグメンテーションの度合いが高まり、I/O スループットが低下してしまうためである。

複数問い合わせの同時処理において、ワーキングメモリサイズと同時処理数について適切な設定を行うことで、SSD の並列 I/O 処理能力や CPU の複数のコアを活用し、逐次の問い合わせ処理と同じ処理時間で、より多くの問い合わせ処理を実行できることが期待される。

6.2 ハッシュ結合を行う問い合わせの複数同時処理性能の計測

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。表 2.2 の環境において、図 5.1 の問い合わせを対象とし、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 13 GB、part 表、lineitem 表のサイズはそれぞれ 320 MB、8.8 GB、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 80 MB 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 64 kB - 512 MB の範囲で変化させて計測する。

図 6.1 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。

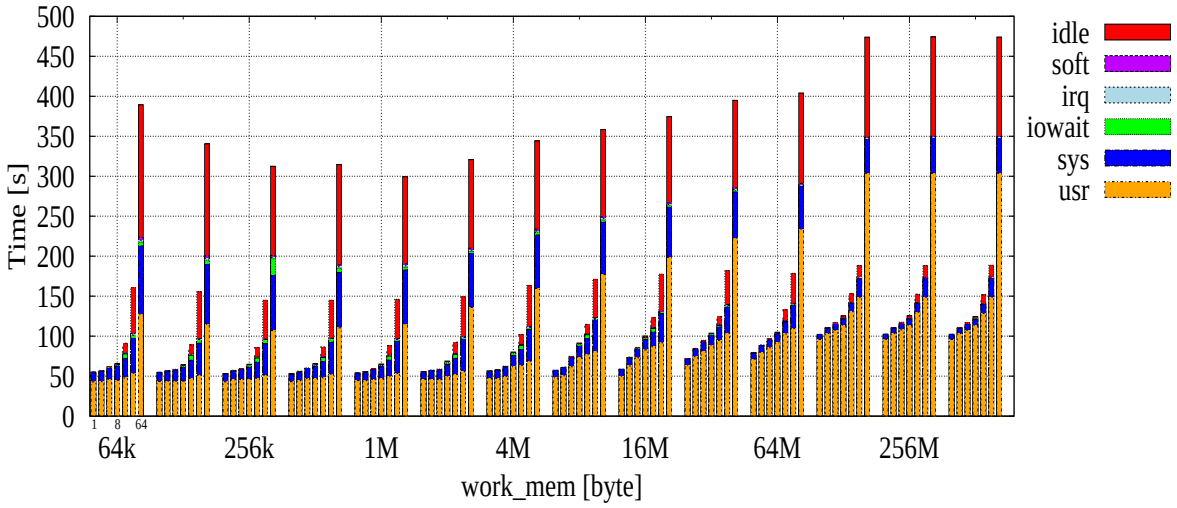


図 6.1: 複数問い合わせ同時処理時の各 work_mem の値における実行時間

6.2.1 CPU コストの増加傾向の変化

図 5.4 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 32 MB から CPU コストが増加しているが、図 6.1 においては、同時処理数 2 では 16 MB、4 では 8 MB、8 では 4 MB 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 6.2 は、データベースバッファをキャッシュレベルで管理する処理方式 (work_mem = 256 kB) と、メモリレベルで管理する処理方式 (work_mem = 256 MB) の 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

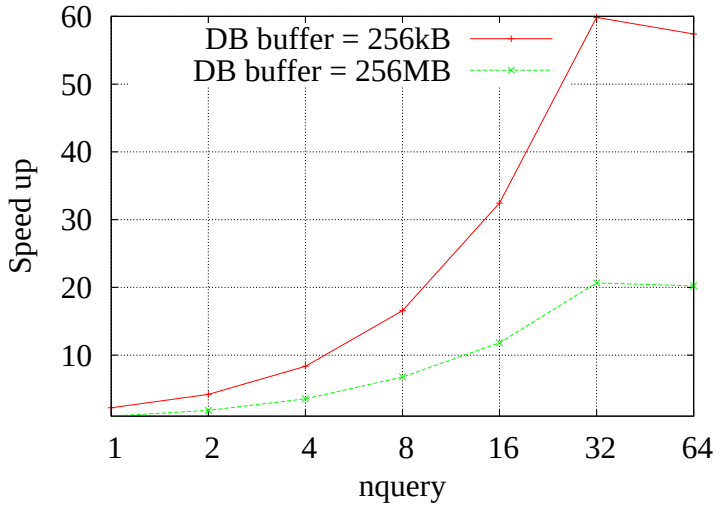
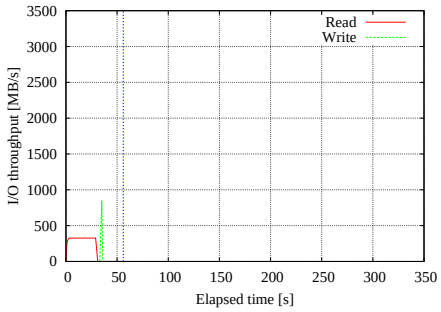


図 6.2: 各問い合わせ同時処理数における CPU 処理の高速化 (`work_mem` = 256 kB, 256 MB)

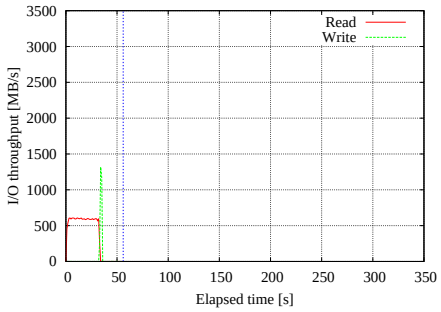
CPU 処理速度を 1 としたときの速度向上を示している。

逐次処理時の CPU 処理速度は、キャッシュレベルでの管理の場合では、メモリレベルでの管理の場合の 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、キャッシュレベルの管理の場合で 59.8、メモリレベルの管理の場合で 20.6 であった。

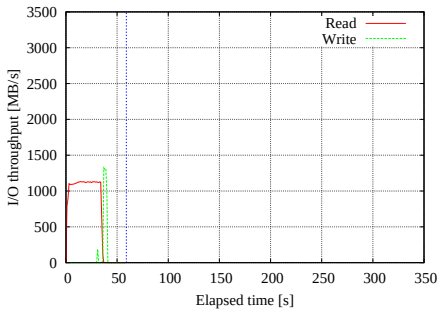
キャッシュレベルの管理の場合は、キャッシュを分割して使用し、キャッシュ参照の競合を抑えることができるため、同時処理数を増加してもキャッシュミス数が増加せず、CPU 処理速度増加の台数効果が大きい。一方、メモリレベルの管理の場合は、同時処理の効果によって CPU 処理速度が増加はしているが、同時処理数の増加に伴いキャッシュ参照の競合の度合いが高まるため、逐次処理の場合と比較するとキャッシュミス数が増加するオーバーヘッドが生じ、キャッシュレベルの管理の場合と比較すると、台数効果が抑えられている。



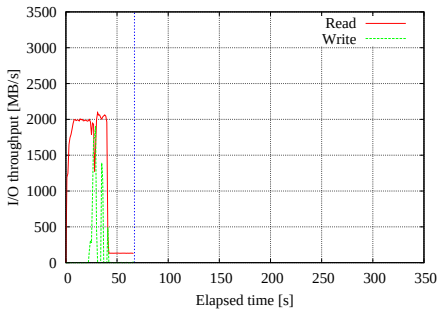
(a) 同時処理数 = 1



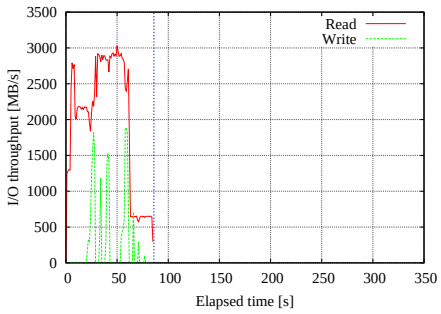
(b) 同時処理数 = 2



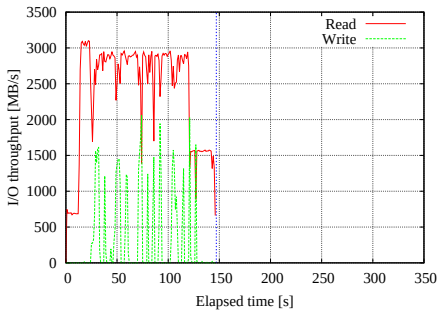
(c) 同時処理数 = 4



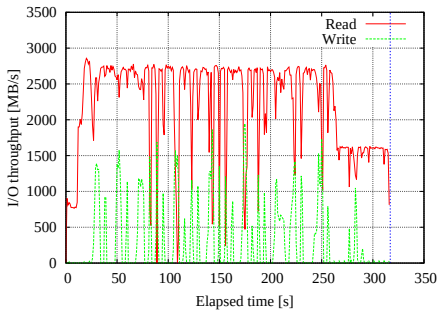
(d) 同時処理数 = 8



(e) 同時処理数 = 16



(f) 同時処理数 = 32



(g) 同時処理数 = 64

図 6.3: 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` = 256 kB)

6.2.2 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 6.3 は、work_mem = 256 kB 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 6.3a では 0 - 30 秒区間、図 6.3g では 0 - 260 秒区間など) 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 6.3 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 6.4 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 スループットは図 2.6 で示した読み込みと書き込みが混合するワークロードにおける、帯域の値 (2600 MB/s) 付近まで達し、飽和している。また、32 から 64 では 200 MB/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 6.1 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

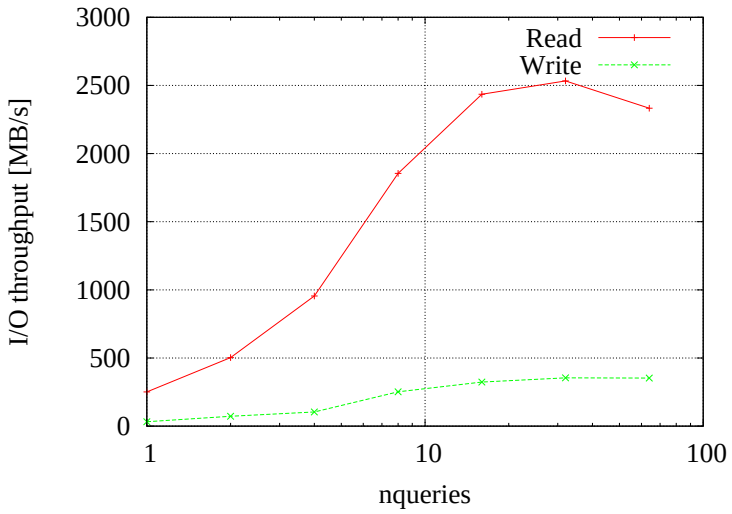


図 6.4: 各問い合わせ同時処理数におけるビルドフェーズの I/O スループット (`work_mem` = 256 kB)

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ということも考えられる。

6.2.3 ハッシュ結合同時実行数の制御

図 6.1 において、同時処理数 1 - 16 の範囲では、I/O スループットが向上し、小さいワーキングメモリサイズを設定することで、キャッシュミス数も低く保たれている。システム全体の処理性能としては、同時処理数 16 のときで逐次処理 (同時処理数 1) のときに対して、13 倍以上の性能向上がみられた。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、更なる処理性能向上が期待できる。

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、単純に同時処理数を増加させることが、計算リソー

スの使用効率が高め、性能向上に結び付くというわけではない。同時処理数を増加し、異なるI/Oワークロードを同時に処理することは、I/Oの複雑性を高め、スループットを低下する要因ともなり得る。SSDにおいても、フラグメンテーションや読み書きのI/Oの混在は、ファイルオープンのオーバーヘッドの増加、デバイス内部での、データの先読みや書き込みバッファの使用効率の低下などにより、I/Oスループット低下の原因である[9]。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。

問い合わせ毎のワークロードを分析し、ストレージ帯域やSSDの内部アーキテクチャなどにあわせ、適切な同時処理数を選択する必要がある。

6.3 ハッシュ結合とスキャンの同時処理

複数問い合わせ同時処理のもう一つの例として、ハッシュ結合演算とテーブルスキャンの組み合わせの場合について計測を行った。TPC-H Scale factor=100のデータベースでlineitem表とpart表をハッシュ結合する問い合わせ(図5.1)を実行し、この問い合わせが終了するまで同時に他のデータベースでテーブルの順次スキャンを繰り返す。

ハッシュ結合演算とテーブルスキャンの同時処理時の、各`work_mem`における実行時間の計測結果を図6.5に示す。図中では、各`work_mem`において結合演算を行う問い合わせのみを実行した場合の結果を併載している。

まず、CPUコストに関しては、結合演算単体処理時は`work_mem` = 32 MBから増加が始まっているのに対し、テーブルスキャン同時処理時は`work_mem` = 8 MBと、より小さい値から増加が始まっている。図6.6は、ハッシュ結合演算それぞれの`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。図6.6の結果の通り、L3キャッシュミスの増加により、CPUコスト増加していることになる。これは、同時に処理されているテーブルスキャンにおいて読み込まれたデータがL3キャッシュに入ることによって、ハッシュテーブルの一部が追い出されることによるものである。この効果は、ハッシュテーブルサイズがL3キャッシュサイズの大き

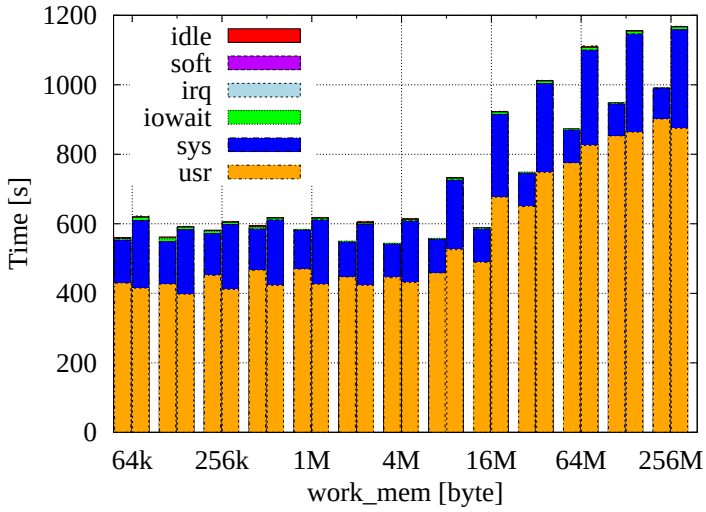


図 6.5: ハッシュ結合演算を行う問い合わせの実行時間 (左:ハッシュ結合演算単体、右:スキャン同時処理時)

さに対して、比較的大きいときに、ハッシュテーブル中の個々のハッシュバケツで参照頻度が低いものが生じ、キャッシュ置き換えアルゴリズムによって選択されるため生じるものである。そのため、ハッシュテーブルサイズを十分小さくすることで、ハッシュテーブルの個々のハッシュバケツへの参照頻度を高めれば、キャッシュからの追い出しを抑えることができる。

全体の処理性能の向上に関して見ると、ハッシュ結合演算とテーブルスキャンで L3 キャッシュを分割して使用できている、`work_mem` $\leq$ 4MB 以下の値では実行時間の増加が小さく、問い合わせ同時処理による効果が大きい。例として `work_mem` $=$ 2MB の場合を挙げると、まず、テーブルスキャンによって読み込んだデータ量は 369 GB であった。順次スキャン単体実行時のスループットは 750 MB/s であったので、ハッシュ結合とスキャンを逐次実行すると $578(\mathrm{s}) + (369(\mathrm{GB})/750(\mathrm{MB/s})) = 1070(\mathrm{s})$ となり、同時実行により 1.7 倍処理性能が向上している。

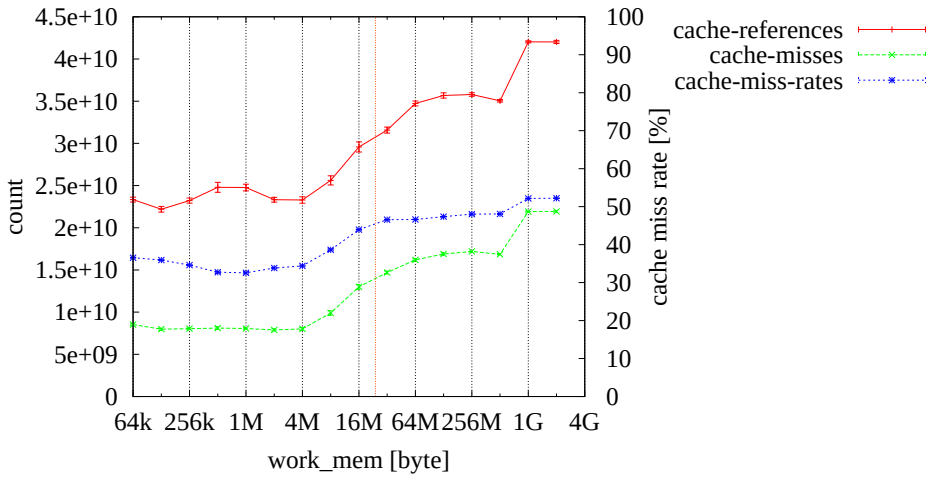


図 6.6: ハッシュ結合演算とスキャン同時処理時の各 work_mem の値における L3 キャッシュ参照・ミス数及びミス率

6.4 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

前節までの実験によって、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。同時処理のスケジューリングを適切に組み立てることで、問い合わせを逐次実行する場合と比較して大きく処理性能を向上することが可能である。

SSD を用いたハッシュ結合演算の複数同時処理では、全ての問い合わせに含まれるハッシュ結合演算の合計数から、全てのハッシュテーブルがキャッシュに収まるよう、個々のハッシュテーブルサイズを決定することで、キャッシュミスの増加が抑制される。問い合わせの複数同時処理を行っても、問い合わせ当たりの処理コストが増加しないため、スケーラビリティを保ったまま同時処理数を増加することができ、処理性能が向上する。

このようなハッシュテーブルのデータアクセス局所性をキャッシュレベルで考慮する処理方式は、他のハッシュテーブルを用いるデータ処理アルゴリズムでも有効で

ある。例えば、大規模データを扱う問い合わせで多用される関係データベース演算の一つである集約演算では、入力のリレーションのサイズが大きいとき、ハッシュテーブルを使用するアルゴリズムが用いられる。ハッシュ集約演算についても、ハッシュ結合演算と同様の方法でハッシュテーブルサイズを決定することで、キャッシュレベルでデータアクセスの局所性を保ち、他の演算との同時実行が可能になる。また、関係データベースシステムに限らず、例として、Key-Value ストアデータベースシステムなどでも、ハッシュテーブルを用いたデータ処理は使用されている。大規模な並列処理を行う Key-Value ストアデータベースシステムにおいても、データアクセス局所性をキャッシュレベルで考慮する処理方式は、処理性能を大きく向上すると考えられる。

ハッシュ結合演算とテーブルスキャンの組み合わせにおいても、テーブルスキャンによって一定量のデータがキャッシュから追い出されることを考慮し、ハッシュテーブルサイズを十分小さく設定することで、データアクセス局所性を保つことができるので、同時処理が有効である。これは他の演算における同時処理の場合でも同様で、各演算のデータアクセスの局所性をキャッシュレベルで考慮し、キャッシュを分割して使用することで、問い合わせ同時処理が可能になる。つまり、SSD を使用する関係データベースシステムでは、関係データベース演算処理のデータアクセスの局所性をキャッシュレベルで考慮し、データベースバッファを管理する処理方式を用いることで、全体の処理性能を向上できるということである。

データベースバッファをキャッシュレベルで管理する処理方式によって、複数問い合わせの同時処理が可能になったことに伴い、問い合わせのプランナや問い合わせ実行エンジンといった、関係データベースシステムの問い合わせ処理最適化機構に、SSD の使用に適合したコストモデルを導入し、同時処理のスケジューリングを行う必要がある。それは、SSD においても大量のファイルへのアクセスや過度な並列数での I/O の発行によって、I/O のフラグメンテーションの度合いが極端に高まることで、I/O スループットが低下してしまうためである。そのため、問い合わせ同時処理のスケジューリングでは、各関係データベース演算の I/O ワークロードとキャッシュレベルでのデータアクセスの局所性に基づいて、同時処理する演算の組み

合わせを選択しなければならない。これを実現するための SSD を使用した関係データベースシステムにおけるコストモデルは、キャッシュミス数や並列 I/O 処理、I/O のフラグメンテーションの度合いなどについて考慮したものである必要がある。こうしたコストモデルの構築やそれを関係データベースシステムの問い合わせ処理最適化機構に組み込み、評価を行うことは、今後の課題としたい。

第7章　おわりに

現在、関係データベースシステムの問い合わせ最適化機構は、HDD のアクセス特性に基づいて設計されており、新しい高速ストレージデバイスである SSD のアクセス特性が十分に反映されていない。そこで、本論文では従来の HDD に基いた問い合わせ処理方式を SSD に適応した場合の問題点を明らかにし、SSD を用いた関係データベースシステムに適した問い合わせ処理方式の提案及びその評価を行った。

まず、SSD を用いた場合の演算処理性能について分析するため、ハッシュ結合演算を対象とし計測を行った。計測結果では、SSD を用いたハッシュ結合演算では、高いランダム I/O スループットにより、パーティションファイルのフラグメンテーションの影響が小さく、データベースバッファ使用量によらず I/O コストが小さくなっていた。I/O コストの低下に伴い、全体の処理コストに占める CPU コストの影響が相対的に大きくなる。ハッシュ結合演算では、タプルのマッチング処理の際にハッシュテーブルを繰り返し参照し、このときに生じるキャッシュミスペナルティのメモリアクセスが、CPU コストを増加させる要因となっていた。単一ハッシュ結合演算や複数ハッシュ結合演算を含む問い合わせでの計測結果によると、問い合わせ中のハッシュ結合演算数に応じて、データベースバッファ使用量を適切に設定し、ハッシュテーブルがキャッシュに収まるように調節することで、ハッシュテーブル参照時のキャッシュミス発生を抑え CPU コストを改善することが可能である。

このように、SSD を用いたハッシュ結合演算処理では、データアクセスの局所性を考慮する重要性が高く、HDD を用いる場合と異なり、ハッシュテーブルの局所性をキャッシュレベルで考慮するべきであることが明かになった。この性質に基づき、SSD を用いた大規模関係データベースシステムに適した問い合わせ処理方式として、関係データベース演算処理のデータアクセスの局所性を利用して、データベースバッ

ファをキャッシュレベルで管理する手法を提案した。

データベースバッファのキャッシュレベルでの管理によって、単一問い合わせ処理における CPU コスト低減効果が確認された。SSD では、並列 I/O 処理が可能であるため、問い合わせ同時処理により処理性能を向上することができる。同時処理においても、データベースバッファをキャッシュレベルで管理する処理方式は、メモリレベルで管理する処理方式より優位であった。HDD 使用時はデータベースバッファをメモリレベルで管理しており、単一問い合わせによってキャッシュ領域の大半を使用する。そのため、複数問い合わせを同時処理すると各問い合わせでキャッシュの使用に関して競合が発生してしまい、キャッシュミス数が増加するオーバーヘッドが生じてしまう。一方、データベースバッファをキャッシュレベルで管理することによって、複数演算においてキャッシュを分割して使用することができる。これにより、キャッシュミス数を増加させることなく複数問い合わせの同時処理を行うことができる。そのため、複数問い合わせの同時処理では、キャッシュレベルでの管理の場合の方が、メモリレベルでの管理の場合と比較して、CPU 処理の速度向上が大きい。

複数問い合わせの同時処理による性能向上について、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検証を行った。ハッシュ結合演算を行う問い合わせの複数同時処理では、同時処理数 16 まで処理性能が増加し、逐次処理の際の 13 倍のスループットを示した。ただし、同時処理数を過度に増加させ、I/O フラグメンテーションの度合いが高まると、I/O スループットが低下し、全体の処理性能が低下してしまう。ハッシュ結合演算を行う問い合わせとテーブルスキャンを行う問い合わせの同時処理についても、適切なデータベースバッファ管理を行うことで、処理性能が向上することを確認した。

計測の結果で見られた通り、SSD においても極度の I/O フラグメンテーションによって処理性能が低下してしまう。このような現象が生じるため、SSD による問い合わせ同時処理の効果を高めるためには、同時処理のスケジューリングが必要である。スケジューラの設計の指針として、適切な同時処理数の設定とキャッシュレベルでのデータベースバッファの管理の双方について考慮するべきであることを示し

た。複数問い合わせ同時処理によるキャッシュ参照の競合を抑え、CPU コストの増加のオーバーヘッドを避けるためには、キャッシュレベルのデータベースバッファ管理が必要である。また、同時処理数を増やせば、フラグメンテーションの度合いも増すため、一定の同時処理数を境に I/O スループットは増加から減少に転じる。SSD の I/O スループットを最大化するためには、適切な同時処理数を選択しなければならない。

今後の課題としては、複数問い合わせにおいてキャッシュを共有して使用する際の分割のアプローチや、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法の検討が挙げられる。また、それらを基に新たに SSD に適合したコストモデルを設計し、関係データベースシステムの問い合わせ処理最適化機構に組み込み、評価を行うことが必要である。

参考文献

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

[2] Perf. https://perf.wiki.kernel.org/.

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

[4] Transaction Processing Performance Council. http://www.tpc.org/.

[5] Big Data Meets Big Data Analytics - Three Key Technologies for Extracting Real-Time Business Value from the Big Data That Threatens to Overwhelm Traditional Computing Architectures, September 2013. http://www.sas.com/resources/whitepaper/wp_46345.pdf.

[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[7] B. Bhattacharjee, K. A. Ross, C. Lang, G. A. Mihaila, and M. Banikazemi. Enhancing recovery using an ssd buffer pool extension. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 10–16, New York, NY, USA, 2011. ACM.

[8] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, Sept. 2010.

[9] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. *SIGMETRICS Perform. Eval. Rev.*, 37(1):181–192, June 2009.

[10] F. Chen, R. Lee, and X. Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 266–277, Washington, DC, USA, 2011. IEEE Computer Society.

[11] T. 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?

[12] D. J. DeWitt, J. Do, J. M. Patel, and D. Zhang. Fast Peak-to-peak Behavior with SSD Buffer Pool. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1129–1140, Washington, DC, USA, 2013. IEEE Computer Society.

[13] C. Dirik and B. Jacob. The Performance of PC Solid-state Disks (SSDs) As a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.

[14] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011*

ACM SIGMOD International Conference on Management of data, SIGMOD '11, pages 1113–1124, New York, NY, USA, 2011. ACM.

[15] W.-H. Kang, S.-W. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, July 2012.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Relational Algebra Machine GRACE. In *Proceedings of RIMS Symposium on Software Science and Engineering*, pages 191–214, London, UK, UK, 1983. Springer-Verlag.

[17] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, Aug. 2008.

[18] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, Sept. 2010.

[19] X. Liu and K. Salem. Hybrid Storage Management for Database Systems. *Proc. VLDB Endow.*, 6(8):541–552, June 2013.

[20] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang. hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, 5(10):1076–1087, June 2012.

[21] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive Algorithms for Server Problems. *J. Algorithms*, 11(2):208–230, May 1990.

[22] D. 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.

[23] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 410–419, New York, NY, USA, 2007. ACM.

[24] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. ACM.

[25] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.*, 5(4):286–297, Dec. 2011.

[26] P. Russom. TDWI Best Practice Report: MANAGING BIG DATA. Technical report, TDWI, 2013. http://tdwi.org/research/2013/10/tdwi-best-practices-report-managing-big-data/asset.aspx?tc=assetpg.

[27] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 110–121, New York, NY, USA, 1989. ACM.

[28] R. Stoica and A. Ailamaki. Improving Flash Write Performance by Using Update Frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[29] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 59–72, New York, NY, USA, 2009. ACM.

[30] R. 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.

[31] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[32] R. Yellin. Conquer Complexity with Teradata Virtual Storage. White paper, Teradata, 2011. http://www.teradata.com/white-papers/Conquer-Complexity-with-Teradata-Virtual-Storage/?type=WP.

発表文献

1. 鈴木恵介, 早水悠登，横山大作，中野美由紀，喜連川優．SSD を利用したリレーショナルデータベースにおける大規模意思決定支援クエリ処理性能の特性．電子情報通信学会データ工学研究会，電子情報通信学会技術報告, Vol. 113, No. 150, DE2013-24, pp.117-122 (2010.12).

2. 鈴木恵介, 早水悠登，横山大作, 中野美由紀，喜連川優．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．電子情報通信学会第 6 回データ工学と情報マネジメントに関するフォーラム (第 12 回日本データベース学会年次大会 (DEIM2014)) (2014.03). (to appear)

3. 鈴木恵介, 早水悠登, 横山大作, 中野美由紀, 喜連川優．複数問い合わせ処理のワークロードに着目した SSD を用いたデータベースの最適化．情報処理学会 76 回全国大会, 3N-1 (2014.03)．(to appear)