Vol.2021-OS-152 No.9
2021/5/27

DBMSのメモリオブジェクトに基づいたCPUキャッシュ分割手法

清水 彰文[1,a)]　山田 浩史[1,b)]

概要：データベース管理システム (DBMS) は Web システムにおいて必要不可欠なコンポーネントとなっている．近年 DBMS では多くのデータアイテムをメインメモリ上で管理している．これらの DBMS ではでは実行順序によってクエリの性能が大きく変わることがある．クエリの実行順序によって CPU キャッシュからデータアイテムが排出されてしまうとその後のクエリ実行に大きな影響を与える．本研究では DBMS のクエリ内でアクセスされるデータを CPU キャッシュに留める手法を提案する．事前にクエリを実行してから，CPU キャッシュのパーティショニングを行いそのデータを CPU キャッシュ内に留め保護する．提案手法を SQLite3.29.0 上に実装し，JOIN を含む単純なクエリでは実行時間をおよそ 25%程度，TPC-H を用いた実践的なクエリでは実行時間をおよそ 25%程度高速化できることを確認した．

1. はじめに

データベース管理システム (DBMS) は様々なシステムを構成する上で必要不可欠なコンポーネントとなっている．関係データベースシステム (RDBMS) はウェブシステムを構成する際に用いられる．RDBMS の例としては MySQL[1] や PostgreSQL[2]， SQLite[3] などが知られている．キーバリューストア (KVS) はネットワークシステムのキャッシュとして利用されることが多くなっている．KVS の例としては Redis[4] や Memcached[5] などが知られている．これらの DBMS はディスクアクセスを極力減らすようにメモリ上に多くのメモリオブジェクトを配置する．また，Memcached といったインメモリ KVS はメモリのみにデータアイテムを配置する．

これらの DBMS では，実行順序によってクエリの性能が大きく変わることがある．データアイテム全体にアクセスするようなメモリインテンシブなクエリは実行時間は安定するものの，インデックスのみやデータアイテムの一部にしかアクセスしない CPU キャッシュインテンシブなクエリは実行前の CPU キャッシュの内容によってその実行時間が大きく異なる．CPU キャッシュインテンシブなクエリは，そのクエリが使うメモリオブジェクトを CPU キャッシュに残すクエリを事前に実行すると，キャッシュ内容をそのまま利用できるため，実行時間が短くなる．一方で，CPU キャッシュインテンシブなクエリが利用しないメモリオブジェクトを使うクエリを事前に実行すると，クエリの実行時間が長くなってしまう．

本研究では，上述した CPU キャッシュインテンシブなクエリが存在すること，またその実行時間がクエリの実行順序によって変わることを示す．加えて，CPU キャッシュインテンシブなクエリの実行時間を安定して短くする手法を提案する．提案手法では，メモリインテンシブなクエリは CPU キャッシュがほぼ効かない点に着目し，CPU キャッシュのパーティショニング機能を応用することで CPU キャッシュインテンシブなクエリが使うメモリオブジェクトを CPU キャッシュに保持する．本研究ではケーススタディとして，Intel RDT を用いることで Index のメモリオブジェクトを CPU キャッシュ上に維持する．

本研究の貢献は以下の通りである．

- Real-world のアプリケーションを用いて，クエリの実行順序によって実行時間が変化することを示した．CPU キャッシュインテンシブなクエリとメモリインテンシブなクエリを交互に実行する場合と連続的に実行する場合とで実行時間が変化することを確認した．
- 特定のメモリオブジェクトを CPU キャッシュ内に留める手法を示した．Intel RDT の機能を用いて，Index のメモリオブジェクトを CPU キャッシュから追い出されないようにする．
- SQLite3.29.0 に実装を行い，実行時間とキャッシュミス率を計測した．その結果，マイクロベンチマークにおいて 25%程度，TPC-H[6] のクエリにおいて 25%程

[1] 東京農工大学 工学府 情報工学専攻

[a)] s204565w@st.go.tuat.ac.jp
[b)] hiroshiy@cc.tuat.ac.jp

item size	16B
column	320,000
table size	7.4MB

表 1 tableA の概要

item size	16B
column	640,000
table size	15MB

表 2 tableB の概要

度クエリの処理時間を高速化できることが確認された．

2. 背景

2.1 CPU キャッシュの重要性

近年データアクセスにおけるボトルネックが移り変わり，CPU キャッシュがより重要な役割を果たすようになっている．メインメモリの容量が大きくなったとにより，インメモリデータベースのような，インメモリコンピューティングが台頭してきたためである．メインメモリを活用することにより，ディスクへのアクセスレイテンシを隠すことができるようになり，ディスクアクセスのボトルネックを取り除くことができるようになった．しかし，メモリアクセスと CPU の処理との間には処理速度の差が存在し，メモリアクセスのレイテンシがデータアクセスの際のボトルネックとなる．このメモリへのアクセスレイテンシを吸収する役割を持つのが CPU キャッシュであり，これらの要因からその役割が以前よりも重要なものとなっている．

一方，CPU キャッシュの大きさはあまり進歩していないという問題がある．DB の大きさはディスクやメインメモリの容量が大きくなるにつれて大きくなってきた．しかし，CPU キャッシュの容量は，CPU のダイのスペースが限られていることからあまり成長していない．実際，近年 100GB を超えるような DB が見られるようになったが，CPU キャッシュの大きさは Xeon Platinum 9282[7] でも 77MB しかない．このため，DBMS では限られた CPU キャッシュの領域を有効活用していく必要が生じる．

2.2 予備実験: クエリの実行順序

2.2.1 目的

DBMS ではクエリを実行する順序によって，その実行速度に差が生じるという問題がある．クエリの実行順序によって CPU キャッシュの利用のされ方が異なるためである．これを確認するために予備実験を行う．

2.2.2 方法

実験で用いるテーブルである tableA の概要を表 1 に示し，tableB の概要を表 2 に示す．

それぞれのテーブルは val というカラムを 1 つだけ持ち，1 アイテムのサイズは 16B となっている．それぞれ 32 万行と 64 万行を持ち，テーブルの持つメタデータを含むデータのサイズはそれぞれ 7.4MB と 15MB となっている．

表 3 実験マシンのスペック

機器名	PowerEdge T630
CPU	Intel(R) Xeon(R) CPU E5-2623 v4 2.6GHz
ソケット数	2
1 ソケットあたりのコア数	4
L1d キャッシュサイズ (コアごと)	32KB
L1i キャッシュサイズ (コアごと)	32KB
L2 キャッシュサイズ (コアごと)	256KB
L3 キャッシュサイズ (ソケットごと)	10240KB
RAM	126GB

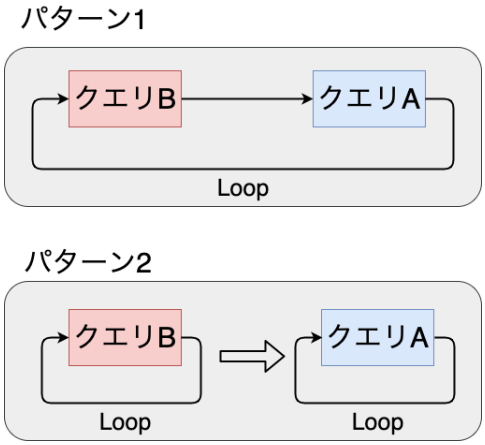

図 1 各パターンでのクエリの実行順序

	実行時間	キャッシュミス率
パターン 1	41.3ms	98.3%
パターン 2	33.5ms	1.0%

表 4 クエリ A の実行時間とキャッシュミス率

実験に用いるマシンのスペックを表 3 に示す．また，実験は SQLite3.29.0 を用いて行う．

表 1,2 に示したテーブルを用いてクエリ実行順序がクエリ実行のパフォーマンスに影響を与えることを示す．実行するクエリは 2 種類用意する．クエリ A では *SELECT val FROM tableA* を実行する．クエリ B では *SELECT val FROM tableB* を実行する．クエリ A とクエリ B の実行順序を変更した際のパフォーマンスの変化についてみていく．クエリ実行の順序のパターンは 2 種類用意する．それぞれのパターンでの実行順序を図 1 に示す．パターン 1 ではクエリ B の実行とクエリ A の実行を交互に行う．パターン 2 ではクエリ B とクエリ A をそれぞれ連続に実行する．

それぞれのパターンではクエリ A とクエリ B を 100 回ずつ実行し，それぞれのクエリの実行時間とキャッシュミス率の平均値を取得する．

2.2.3 結果

それぞれの実行パターンで実行した際のクエリ A, B の実行時間とキャッシュミス率を計測した結果をそれぞれ表 4, 5 に示す．

	実行時間	キャッシュミス率
パターン 1	83.2ms	99%
パータン 2	81.0ms	85%

表 5 クエリ B の実行時間とキャッシュミス率

表 4 より，パターン 1 ではパターン 2 と比較してクエリ A の実行速度がおよそ 23%程度低速になり，大量のキャッシュミスが引き起こされていることがわかる．これはクエリ B を実行した際に，tableA のデータが CPU キャッシュから追い出されてしまうためであると考えられる．表 5 より，パターン 1 でもパターン 2 でもクエリ B の実行速度はほとんど変化せず，キャッシュミス率の変化も小さいことがわかる．これは tableB のテーブルサイズが 15MB であり，CPU キャッシュの容量である 10MB よりも大きいことが原因であると考えられる．以上より，クエリ B では CPU キャッシュを有効利用できないのにも関わらず，CPU キャシュに大量のデータを載せてしまうことがわかる．これにより，クエリの実行順序によって CPU キャッシュを有効利用することのできるクエリの実行速度が影響を受けてしまうことがわかる．

3. 関連研究

CPU キャッシュを活用することでシステム全体やアプリケーションのパフォーマンスを向上させる研究が行われている．

dCAT[8] という研究では，VM の CPU 時間や物理メモリの配分の研究は存在するが，CPU キャッシュの配分に関する研究が少ないことから，CPU キャッシュを VM に配分する方法についての手法を提案していた．dCAT では状態遷移を用いて，アプリケーションの動作フェーズを意識した CPU キャッシュの配分を行えるようにしていた．これにより，各 VM が均等に CPU キャッシュを持っている時に比べた性能の劣化を抑え，全体の性能を向上させることに成功してた．

Sun らの手法 [9] では，ハードウェアプリフェッチによって CPU キャッシュが汚染されることに着目して，ハードウェアプリフェッチを考慮した CPU キャッシュの配分についての手法を提案していた．この手法では，各アプリケーションがプリフェッチを有効活用できるかどうかを判断し，それによって CPU キャッシュの配分を決定していた．プリフェッチを有効活用することのできるアプリケーションは CPU キャッシュをあまり利用しないため，その他の CPU キャッシュを有効利用できるアプリケーションが多くの CPU キャッシュを利用できるようにしていた．これにより，各アプリケーションの公平性とパフォーマンスを向上させることに成功していた．

Noll らの手法 [10] では，In-Memory DBMS において並列実行されるクエリ間での CPU キャッシュの競合が生じること着目して，クエリ間での CPU キャッシュの配分を決定する手法を提案していた．この手法では，CPU キャッシュにセンシティブなクエリかそうでないのかを事前に計測していた．計測結果に基づいて，CPU キャッシュにセンシティブなクエリには多くの CPU キャッシュを割り当て，CPU キャッシュにセンシティブでないクエリには少ない CPU キャッシュを割り当てていた．これにより，CPU キャッシュにセンシティブでないアプリの性能はあまり低下させることなく，CPU キャッシュにセンシティブなクエリの性能を向上させることに成功していた．

また，SLB[11] では，Index の探索の際に CPU キャッシュが汚染されることに着目して，Index の探索結果をキャッシュしておく手法を提案していた．Index の探索の際にはデータアイテムとは直接関係のない Index のデータが大量にアクセスされることになる．そこでこの手法では，データアイテムへのポインタをキャッシュしておく機構である SLB を導入することで，Index へのアクセスを最小限にできるようにしていた．これにより，B^+-Tree やハッシュテーブルの探索を高速化することに成功し，Facebook のトレースでは 73%の性能向上を実現していた．

dCAT と Sun らの手法では複数アプリケーション間の性能を向上させることはできるが，単一アプリケーションのみで動作している環境には適用できないという問題がある．単一アプリケーション内での CPU キャッシュの競合については考慮しておらずこれらの手法では，単一アプリケーショのみが動作する環境ではパフォーマンスを向上させることができない．

Noll らの手法では，事前にクエリに関する詳細なプロファイリングを行う必要があり，プロファイリングを行っていないクエリには適用することができないという問題がある．SQL では複雑なクエリを作成することができるため，SQL で生成されるすべてのクエリに対してプロファイリングを行うことは困難である．

SLB では，SLB 自体が CPU キャッシュに残ることを保証していないという問題がある．CPU キャッシュを汚染するような大きなデータアイテムへのアクセスが生じると SLB 自体が CPU キャッシュから追い出され，アクセスに時間がかかってしまう．

4. 提案

本研究では，クエリ実行時のデータを CPU キャッシュに挿入し，そのデータを保持する手法を提案する．既存研究の問題点を解決するために本研究では以下のデザインゴールを満たすように設計する．

- 単一の DBMS が動くマシン上でのクエリの高速化を実現する．
- DBMS で実行されるクエリ実行時にアクセスされるデータを CPU キャッシュに保持しクエリの高速化を

行う．

大きいデータイテムにアクセスするクエリが実行されると，CPU キャッシュを活用できるはずのクエリのデータアイテムが CPU キャッシュから追い出されてしまう．CPU キャッシュを活用できるクエリのデータアイテムが CPU キャッシュから追い出されてしまうと，メモリアクセスが発生してクエリ実行のパフォーマンスが大きく低下する．本研究では，クエリ内のデータを CPU キャッシュに留めることで，クエリの実行時間を高速化する．

以上を実現するために解決しなければならないデザインチャレンジは以下の通りである．

- CPU キャッシュに特定のデータを挿入しなければならない
  CPU キャッシュは透過的にデータを挿入するポリシーをとっているため，特定のデータを挿入するインターフェースを用意していない．よって，本研究では意図したデータを CPU キャッシュに挿入する必要がある．
- CPU キャッシュの特定のデータを保護しなければならいない
  CPU キャッシュのデータの置換ポリシーは LRU などの形式をとっているため，特定のデータを保護することは難しい．よって，本研究では特定のデータを CPU キャッシュに留めることを考えなければならない．

5. 設計

5.1 データアイテムの挿入

本研究ではクエリ内のデータを CPU キャッシュに留めることを考える．CPU キャッシュは透過的なポリシーを採用しており，挿入されるデータは実際にアクセスされたデータとなる．そこで，本研究では CPU キャッシュに特定のデータを留めるために，クエリを事前に実行してデータアイテムにアクセスすることによってデータアイテムを CPU キャッシュに挿入することを考える．

また，クエリ内のデータへのアクセスはクエリを直接実行することの他に，特定のメモリオブジェクトにアクセスすることも考えられる．そこで，本研究では頻繁にアクセスされるメモリオブジェクトとして Index へのアクセスの関数も用意する．Index へアクセスする関数では Index の各アイテムを網羅的にアクセスできるようになっている．事前に Index へのアクセスを行う関数を呼び出して CPU キャッシュに Index のデータを挿入できるようにしておく．

5.2 CPU キャッシュのパーティショニング

本研究では CPU キャッシュパーティショニングを行うために Intel CAT(Cache Allocation Technology)[12] を用いる．Intel CAT は Intel RDT(Resource Detection Technology)[13] の中に含まれる技術の 1 つで，CPU キャッシュのパーティショニングを行うのに利用される．Intel RDT には他にも，メモリ帯域の監視やパーティショニングを行うための Intel MBM(Memory Bandwidth Monitiroing)[14] や Intel MBA(Memory Bandwidth Allocation)[15] や CPU キャッシュの利用を監視するための Intel CMT(Cache Monitoring Technology) が存在する．

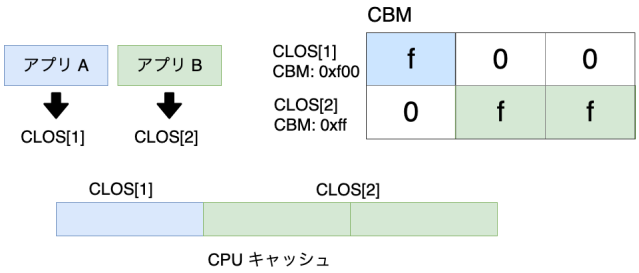

図 2 Intel CAT を用いて LLC のパーティショニングを行う様子

Intel CAT では LLC(Last Level Cache) の配分を行う機能を提供している．Intel CAT では CLOS(Class Of Service) と CBM(Capacity Bit Mask) を定義することにより LLC の配分を指定することができる．CLOS はアプリやコアを CAT で扱う 1 単位として紐付けることができる．また，CBM では各 CLOS に配分する LLC の量を指定することができる．これらの要素を用いることで CPU キャッシュのパーティションを区切ることができる．Intel CAT を用いたパーティショニングの例を図 2 に示す．図 2 ではアプリ A に CLOS1 を割り当て，アプリ B に CLOS2 に割り当てている．この際，それぞれの CLOS1，CLOS2 には CBM を 0xf00, 0x0ff として LLC を割り当てている．これにより，アプリ A は LLC の $\frac{1}{3}$，アプリ B は LLC の $\frac{2}{3}$ をそれぞれ排他的に利用できるようになる．CPU キャッシュのパーティショニングを行っている際には，排他的な CPU キャッシュの利用を行うため CLOS1 から CLOS2 の領域への書き込みはできないが，CLOS1 から CLOS2 の領域内にあるデータの読み込みは可能となっている．

本研究では Intel CAT を利用して CPU キャッシュパーティショニングを行い，クエリ内で用いるデータを保護する．クエリ内で利用されるデータに事前にアクセスしてデータを CPU キャッシュに載せた後，CPU キャッシュのパーティショニングを行う．これにより，クエリ内のデータが CPU キャッシュから追い出されるのを防ぎ，データを保護することを可能にする．

6. 実装

6.1 SQLite への実装

SQLite には Index Btree と Table Btree の 2 種類の Index が存在する．Index Btree は特定のカラムの検索を高速化するために，主にユーザによって作成される Index となっている．Table Btree は row id に対して作成される Index で，すべてのテーブルに存在し，自動で作成される．

Index Btree は B^+-Tree のような構造を持ち，特定のカ

ラムのデータの検索を高速化する．Index Btree を作成する際には *CREATE INDEX idx ON tbl(col1)* のような SQL 文を利用して作成する．これにより，テーブル tbl のカラム col1 に対して idx という Index が作成される．この Index を利用することで，col1 のデータを検索する *SELECT col1 FROM tbl WHERE col1>10* のようなクエリを高速化することができる．Index Btree は B^+-Tree であるため，ソート済みの構造を持っていることから，このクエリでは Index Btree を順に辿ることで結果を取得することができる．一方，Index Btree には tbl の他のカラムである col2 に対する検索に対しては利用することができないという性質がある．

Table Btree は B-Tree のような構造を持ち，row id に対する検索を行うために存在する．ほとんどのクエリはこのテーブルを用いてアクセスされる．例えば *SELECT col2 FROM tbl* のクエリは Index Btree が存在しないため，Table Btree を辿ることでアクセスされる．また，SELECT col1, col2 FROM tbl WHERE col1>10 というクエリでは，col1 に Index が存在する際には col2 のデータを取得するために Table Btree の探索が必要となる．これは Index Btree は特定のカラム以外の情報を持たない性質によるものである．

本研究では Index Btree, Table Btree のそれぞれに対して網羅的にアクセスするような関数を作成する．Table Btree と Table Btree にアクセスする関数では Btree をたどり，すべてのノードにアクセスする．その際，リーフノードや中間ノードに存在するデータアイテムにはアクセスしないこととする．

6.2 Intel CAT を用いたパーティショニング

本研究では CPU キャッシュをデータ保護用のパーティションとそれ以外の処理用のパーティションに分割する．CPU キャッシュのパーティショニングには前述の Intel CAT を用いる．データ保護用のパーティションとして CLOS1 を用意し，それ以外の処理用のパーティションとして CLOS2 を利用する．CLOS1 では CPU キャッシュ全体を指定し，CLOS2 の CBM は最小を割り当てることとする．これにより，クエリ内のデータがそれ以外の処理によって CPU キャッシュから追い出されてしまうことを防げるようになる．

前述の通り，Intel CAT によるパーティショニングにおいてはデータの読み込みに対しては制約が生じない．このため，CLOS2 に割り当てられている際にも CPU キャッシュに留めておいたデータへのアクセスを行うことができる．これにより特定のデータが CPU キャッシュから追い出されることを防ぎつつ，任意のコードからそのデータを呼び出すことが可能となる．

item size	8B
column	400,000
index size	5MB

表 6 tblA の概要

item size	8B
column	1,000

表 7 tblB の概要

item size	16B
column	640,000
table size	15MB

表 8 tblC の概要

7. 実験

7.1 実験環境

本研究で用いるマシンのスペックは予備実験で使用したものと同様であるため，表 3 に示したものと同様である．

また，すべての実験は Ubuntu 18.04 上で行い，すべての実験はハイパースレッティング機能とターボブースト機能をオフにした状態で行う．

実験の際の実行時間の計測は rdtsc を用いる．さらに，CPU キャッシュのアクセス数とキャッシュミスの測定の際には likwid を用いる．

7.2 目的

提案手法の評価のために，2 つの実験を行う．マイクロベンチマークでは，Index を CPU キャッシュに留めることで，JOIN 句を含む単純なクエリを高速化できることを示す．クエリ内で頻繁にアクセスされる代表例として Index を CPU キャッシュ内に留めることを考える．また，TPC-H を用いた実験では，あるクエリのデータを CPU キャッシュ内に留めることで，その他のクエリの動作が高速化できることを示す．TPC-H のような複雑で実践的なクエリでも CPU キャッシュにデータを留める手法が効果的であることを示す．

7.3 実験 1: マイクロベンチマーク

7.3.1 目的

単純なクエリでもクエリ内で用いるデータを CPU キャッシュに残すことで高速化を行えることを示す．その際には頻繁にアクセスされるメモリオブジェクトである Index を CPU キャッシュ内に留めることを考える．

7.3.2 実験方法

JOIN 句を含むクエリと大量のデータにアクセスするクエリを交互に実行し，その際の JOIN 句を含むクエリの実行時間を計測する．本実験で利用するテーブルである tblA, tblB, tblC の概要をそれぞれ表 6, 7, 8 に示す． JOIN 句を

表 9　JOIN を含むクエリの実行時間とキャッシュミス率

	実行時間	キャッシュミス率
パーティショニング無し	1.16ms	77%
パーティショニング有り	0.84ms	25%

含むクエリは
SELECT COUNT(key) FORM tblA INNER JOIN tblB ON tblA.key == tblB.key
を実行する．また，大量のデータにアクセスするクエリでは *SELECT * FROM tblC* を実行する．それぞれのクエリを交互に実行する際には，CPU キャッシュのパーティショニングを行った場合と行っていない場合を比較する．

7.3.3　実験結果

実験を行った結果を表 9 に示す．表 9 より CPU キャッシュのパーティショニングを行うことによって JOIN 句を含むクエリを高速化できることがわかる．CPU キャッシュのパーティショニングが無い場合に比べて，パーティショニングがある場合はおよそ 25%程度高速化を行うことができた．これより，Index を CPU キャッシュに留めることで，クエリの処理を高速化できることが確認できた．

7.4　実験 2: TPC-H のクエリ

7.4.1　目的

複雑かつ実践的なクエリでもクエリ内のデータを残すことでクエリの実行を高速化できることを示す．この際．1 種類のクエリが高速化できることだけでなく，複数のクエリ高速化が行えることを確認する．

7.4.2　実験方法

TPC-H のクエリ 18 とクエリ 14 を用いて，CPU キャッシュ内にデータを残した際のクエリの実行時間を計測する．まずはクエリ 14 を実行し，その後 CPU キャッシュのパーティショニングを行い，データを CPU キャッシュ内に留める．その後，likwid の memsweep_domain 関数を用いて CPU キャッシュ内のデータをクリアする処理を実行してから，クエリ 14 を実行した際の実行時間とクエリ 18 を実行した際の実行時間とキャッシュミス率を計測する．

さらに，CPU キャッシュのパーティショニングをせずに, likwid の memsweep_domain 関数を用いて CPU キャッシュ内のデータをクリアしてからクエリ 14 とクエリ 18 を実行した際の実行時間とキャッシュミス率についても計測する．

TPC-H の scale factor は 0.01 とし, DB のサイズは 12MB とする．

7.4.3　実験結果

クエリ 14 の実行時間とキャッシュミス率を表 10 に示す．表 10 より，CPU キャッシュにクエリ内のデータを留めることで，TPC-H のクエリを高速化できることが示せた．CPU キャッシュのパーティショニングが無い場合に比べて，パーティショニングがある場合はおよそ 30%程度高速化を行うことができた．また，CPU キャシュのミス率を大幅に低下させられることを示せた．これより，クエリのデータを CPU キャッシュ内に留めておくことで，高速化が行えることが示せた．

表 10　クエリ 14 の実行時間とキャッシュミス率

	実行時間	キャッシュミス率
パーティショニング無し	19.4ms	94%
パーティショニング有り	13.5ms	18%

表 11　クエリ 18 の実行時間とキャッシュミス率

	実行時間	キャッシュミス率
パーティショニング無し	17.5ms	98%
パーティショニング有り	13.1ms	33%

クエリ 18 の実行時間とキャッシュミス率を表 11 に示す．表 11 より，他のクエリのデータを CPU キャッシュ内に留めることでもクエリの実行速度を高速化できることを示した．CPU キャッシュのパーティショニングを行った場合では，行っていない場合に比べておよそ 25%程度高速化をすることができた．また，CPU キャッシュのミス率を大幅に低下させられることを示せた．これより，クエリ内の共通で利用されるデータを CPU キャッシュに留めておくことで高速化を行えることが示せた．クエリの実行順序によって CPU キャッシュ内のデータがクリアされてしまうようなことが起きるとパフォーマンスに大きな影響が生じることがわかる．

8.　おわりに

8.1　まとめ

本研究ではクエリの実行順序によって，クエリ内のデータがフラッシュされてしまう現象に注目し，CPU キャッシュ内にデータを留めることでクエリの高速化を行えることを示した．CPU キャシュ内にデータを留めることは Intel RDT を用いた CPU キャッシュパーティショニングを用いて実現した．この手法により，マイクロベンチマークと TPC-H を用いた実践的なクエリのそれぞれでクエリの実行速度を向上させられることを示した．

8.2　今後の課題

8.2.1　その他の DBMS への応用

本研究では SQLite を題材に実験を行ったが, 他の DBMS でも同様にクエリ内でアクセスされるデータを CPU キャッシュに留めることでクエリを高速化することができるのではないだろうか．SQLite は B − Tree を Index のデータ構造として利用する DBMS であった．一方，Berkeley DB のようなハッシュテーブルを Index として持つような DBMS でもデータアイテムを CPU キャッシュに留めることでクエリを高速化できることが考えられる．

8.2.2 動的な割当

本研究では静的な割当を行っていたが，クエリの特性を考慮して動的に CPU キャッシュの割当を変更することが考えられる．静的な割当を行うとメモリインテンシブでないクエリの処理が低速になる恐れがある．これにより，CPU キャッシュインテンシブかどうかを判断し，動的に CPU キャッシュの配分を決められるような機構を導入することでより全体のパフォーマンスを向上させられることが考えられる．

参考文献

[1] *MySql*. URL: https://www.mysql.com/jp/ (visited on 05/03/2021).

[2] *PostgreSQL*. URL: https://www.postgresql.org/ (visited on 01/28/2020).

[3] *SQLite3*. URL: https://www.sqlite.org/index.html (visited on 05/03/2021).

[4] *Redis*. URL: https://redis.io/ (visited on 05/03/2021).

[5] *memcached - a distributed memory object caching system*. URL: https://memcached.org/ (visited on 05/03/2021).

[6] *TPC-H Homepage*. URL: http://tpc.org/tpch/ (visited on 05/03/2021).

[7] *Intel® Xeon® Platinum 9282 Processor (77M Cache, 2.60 GHz) Product Specifications*. URL: https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html (visited on 05/03/2021).

[8] Cong Xu et al. “dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-Service”. In: *Proceedings of the 13th EuroSys Conference 2018*. Vol. 2018-Janua. 2018, p. 13. ISBN: 9781450355841. DOI: 10.1145/3190508.3190555.

[9] Gongjin Sun, Junjie Shen, and Alexander V. Veidenbaum. “Combining prefetch control and cache partitioning to improve multicore performance”. In: Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 953–962. ISBN: 9781728112466. DOI: 10.1109/IPDPS.2019.00103.

[10] Stefan Noll et al. “Accelerating Concurrent Workloads with CPU Cache Partitioning”. In: 2018.

[11] Xingbo Wu, Fan Ni, and Song Jiang. “Search lookaside buffer: Efficient caching for index data structures”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 27–39. ISBN: 9781450350280. DOI: 10.1145/3127479.3127483.

[12] *Introduction to Cache Allocation Technology in the Intel® Xeon®*. URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html (visited on 05/03/2021).

[13] *Intel® Resource Director Technology (Intel® RDT) Reference Manual*. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-resource-director-technology-rdt-reference-manual.html (visited on 05/03/2021).

[14] *Introduction to Memory Bandwidth Monitoring in the Intel® Xeon®*. URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-monitoring.html (visited on 05/03/2021).

[15] *Introduction to Memory Bandwidth Allocation*. URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html (visited on 05/03/2021).

[16] *pqos man page*. URL: https://www.mankier.com/8/pqos (visited on 05/03/2021).

[17] *LIKWID Performance Tools*. URL: https://hpc.fau.de/research/tools/likwid/.

[18] *Intel cmt cat wiki*. URL: https://github.com/intel/intel-cmt-cat/wiki (visited on 05/03/2021).

[19] *Intel ® 64 and IA-32 Architectures Software Developer ’ s Manual*. URL: http://www.intel.com/design/itanium/manuals/iiasdmanual.htm (visited on 05/03/2021).

[20] Sally A. McKee. “Reflections on the memory wall”. In: *In Proceedings ofthe 1st Conference on Computing Frontiers* (2004), pp. 162–167. DOI: 10.1145/977091.977115.

[21] *SQLite B-Tree Module*. URL: https://www.sqlite.org/btreemodule.html.

[22] Onur Kocberber et al. “Meet the walkers: Accelerating index traversals for in-memory databases”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 2013, pp. 468–479. ISBN: 9781450326384. DOI: 10.1145/2540708.2540748.

[23] Huanchen Zhang et al. “Reducing the storage overhead of main-memory OLTP databases with hybrid indexes”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Vol. 26-June-20. 2016, pp. 1567–1581. ISBN: 9781450335317. DOI: 10.1145/2882903.2915222.

[24] Alireza Farshin et al. “Make the most out of last level cache in Intel processors”. In: *Proceedings of the 14th EuroSys Conference 2019*. 2019, p. 17. ISBN: 9781450362818. DOI: 10.1145/3302424.3303977.

[25] Jinsu Park, Seongbeom Park, and Woongki Baek. “CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers”. In: *Proceedings of the 14th EuroSys Conference 2019*. 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303963.

[27] Shuang Chen, Christina Delimitrou, and Jose F. Martinez. “PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services”. In: Association for Computing Machinery, Apr. 2019, pp. 107–120. ISBN: 9781450362405. DOI: 10.1145/3297858.3304005.

[28] Jinsu Park et al. “HyPart: A hybrid technique for practical memory bandwidth partitioning on commodity servers”. In: Institute of Electrical and Electronics Engineers Inc., Nov. 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243211.

[29] Kefei Wang, Jian Liu, and Feng Chen. “Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores”. In: vol. 13. 2020, pp. 1540–1554. DOI: 10.14778/3397230.3397247.

[30] Jongwook Chung et al. “Enforcing Last-Level Cache Partitioning through Memory Virtual Channels”. In: vol. 2019-September. Institute of Electrical and Electronics Engineers Inc., Sept. 2019, pp. 97–109. ISBN: 9781728136134. DOI: 10.1109/PACT.2019.00016.

[31] Fan Ni et al. “SDC: A software defined cache for efficient data indexing”. In: Association for Computing Machinery, June 2019, pp. 82–93. ISBN: 9781450360791. DOI: 10.1145/3330345.3330353.