【VLDB2011勉強会】

Session 9: New Hardware Architecture

担当：山室健（NTT）

概要

- このセクションの特徴
  - 性能向上のための最新Hardware（CPU/GPU/FPGA･･･）適用を目的とした研究に関する
  - 今回は4本ともCPU最適化（in-memory前提）のみ！

- 紹介する論文リスト
  - 1. HYRISE – A Main Memory Hybrid Storage Engine
  - 2. Fast Set Intersection in Memory
  - 3. Efficiently Compiling Efficient Query Plans for Modern Hardware
  - 4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

HYRISE - A Main Memory Hybrid Storage Engine

Martin Grund (Hasso-Plattner-Institute), Jens Krüger (Hasso-Plattner-Institute), Hasso Plattner (Hasso-Plattner Institute), Alexander Zeier (Hasso-Plattner Institute), Philippe Cudre-Mauroux (MIT), Samuel Madden (MIT)

1. HYRISE—A Main Memory Hybrid Storage Engine

- A overview
  - **r/wのlocalityが高く**なるように，workload（OLAP/OLTP）に対して**動的にvertical partitioningのcolumn数を調整**するstorageの提案，従来のnaïveなrow-/columnar-storageと比較して**20-400%の高速化**を実現
- Contribution
  - **data-layout分析**， multi-cores CPU環境で、L1/L2 cachesが性能に与える影響の詳細なprofilingの実施
  - **cost-modelの構築**，workloadとdata-layoutからoperation（projection，selection，etc）性能を推定する評価式を定義
  - **design-toolの実装**，schema，workload，cost-modelを入力に最適なdata-layoutを決定するtoolの実装

1. HYRISE―A Main Memory Hybrid Storage Engine

- A basic idea – data-layoutの選択
  - workloadに対してlocalityが高くなるようにdataを配置
    - OLAP-workloadであればcolumnar志向の狭いpartitioning
    - OLTP-workloadであればrow志向の広いpartitioning

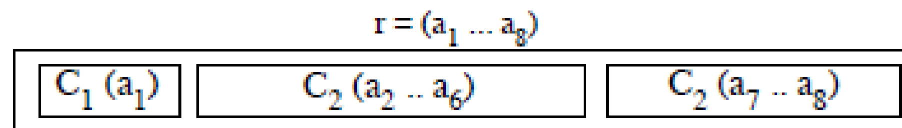


Figure 2: Partitioning example

論文内のFigure.2から引用

- 各operation（selection/projection･･･）におけるcache-miss回数を推定する評価式を定義
  - 実APを調査したところcolumn数（up to 300)が非常に多く，探索空間が非常に大きい問題→Scalableな解探索の方法も提案

1. HYRISE—A Main Memory Hybrid Storage Engine

- 実際のAPから作成した独自のworkloadで評価
  - 理由: TPC-X系のbenchmarksはcolumn数の観点で現実から乖離
  - workloadの詳細はAppendix.C

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Total
Row- R	567.7	240.3	270.5	152.5	967.8	0.9	13890.3	52301.5	5431.5	32297.6	29687.8	117471.1	4899.2	258179.2
Column- C	90.5	35.1	112.2	119.3	309.4	2.6	2154.8	9416.0	795.5	6032.4	6744.1	45468.6	2939.8	74221.0
Hybrid- H	92.9	29.1	40.1	128.1	116.6	1.0	1795.3	7114.8	723.2	6243.5	6852.6	45751.1	2517.7	71406.3

Figure 5: Benchmark Results; graphs at the top show normalized (slowest system=1.0) CPU cycles (left) and normalized L2 cache misses (right); table shows absolute CPU cycles / 100k

論文内のFigure.5から引用

1. HYRISE—A Main Memory Hybrid Storage Engine

- 実際のAPから生成した独自のworkloadで評価
  - 理由: TPC−X系のbenchmarksはcolumn数の観点で現実から乖離
  - workloadの詳細はAppendix.C

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11			Total
Row- R	567.7	240.3	270.5	152.5	967.8	0.9	13890.3	52301.5	5431.5	32297.6	29687.8	117471.1	4899.2	258179.2
Column- C	90.5	35.1	112.2	119.3	309.4	2.6	2154.8	9416.0	795.5	6032.4	6744.1	45468.6	2939.8	74221.0
Hybrid- H	92.9	29.1	40.1	128.1	116.6	1.0	1795.3	7114.8	723.2	6243.5	6852.6	45751.1	2517.7	71406.3

Figure 5: Benchmark Results; graphs at the top show normalized (slowest system=1.0) CPU cycles (left) and n
s / 100k

~400%の改善

~20%の改善

クエリQによってR>C/R<Cになるような Hybridなworkload

論文内のFigure.5から引用

1. HYRISE—A Main Memory Hybrid Storage Engine

- 実際のAPから生成した独自のworkloadで評価
  - 理由: TPC-X系のbenchmarksはcolumn数の観点で現実から乖離
  - workloadの詳細はAppendix.C

~400%の改善

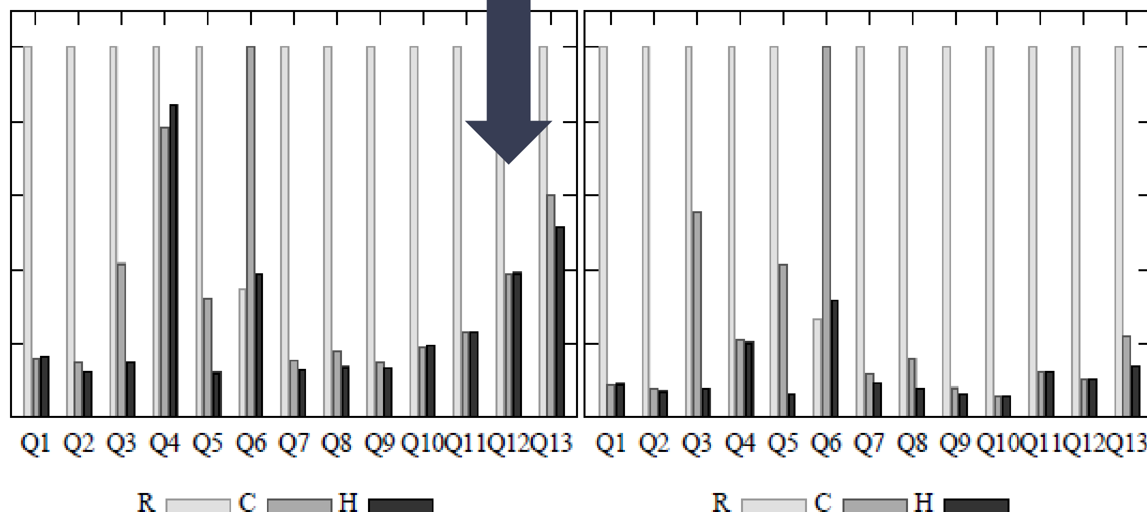
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	[illegible]	[illegible]	Total
Row- R	567.7	240.3	270.5	152.5	967.8	0.9	13890.3	52301.5	5431.5	32297.6	29687.8	117471.1	4899.2	258179.2
Column- C	90.5	35.1	112.2	119.3	309.4	2.6	2154.8	9416.0	795.5	6032.4	6744.1	45468.6	2939.8	74221.0
Hybrid- H	92.9	29.1	40.1	12[illegible]	116.6	1.0	1795.3	7114.8	723.2	6243.5	6852.6	45751.1	2517.7	71406.3

ns at the top show normalized (slowest system=1.0) CPU cycles (left) and n
PU cycles / 100k

~20%の改善

最悪性能を基準に正規化した
CPU cyclesの棒グラフ

Normalized CPU Cycles　　Normalized Cache Misses(L2)



論文内のFigure.5から引用

左の棒グラフから得られる結論
- 概ね全てのクエリQに対して提案手法Hの性能が良い
- L2 Cache Missesの優劣が最終的な性能(CPU Cycles)が決定

Fast Set Intersection in Memory

Bolin Ding (UIUC), Arnd Christian König (Microsoft Research)

2. Fast Set Intersection in Memory

- A overview
  - 2個以上の集合から共通項(Intersection)を探索する処理を，**得られる共通項が小さいことを前提**に，**オンメモリ上での処理を高速化**する手法を提案し，従来手法と比較した**計算量Oの改善とPracticalな環境での高速化を実現**
- Contribution
  - **計算量 $O(n/\sqrt{w}+kr)$ に改善**
    - n: 要素数，k: 前処理で分割する集合数，r: 共通項数，w: word長
    - 最善の従来手法の計算量[6]: $O((n(\log_2 w)^2)/w+kr)$

      ※ $1/\sqrt{w} < (\log_2 w)^2/w \quad (w<2^{16})$
  - **計算量を悪化させたSimple&Efficientなalgorithmの提案**
    - 計算量は $O(n/\sqrt{w}+kr)$ から $O(n/\alpha^m + mn/\sqrt{w} + kr\sqrt{w})$ に

      α : wから計算される値，m: 任意のパラメータ
    - Practicalな環境では高速

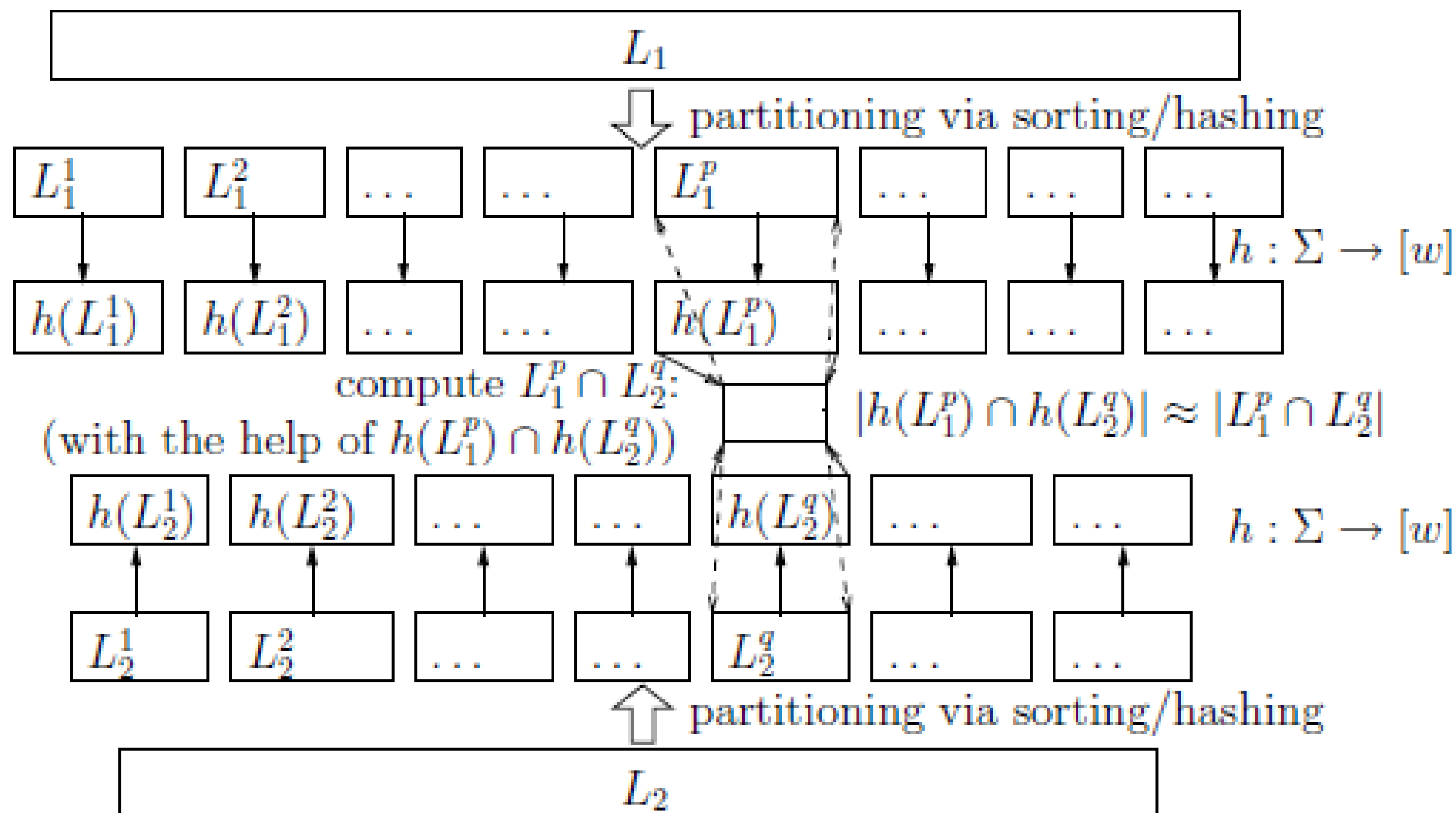
2. Fast Set Intersection in Memory

- A basic idea – 2つの着眼点(前提)
  - 1. word長(32-bit/64-bit)での共通項取得は高速
  - 2. 経験的に結果集合が小さい

- 処理の概要(集合が2つの場合)
  - 集合L_1/L_2を小さいブロックL_1^k/L_2^kに分割
  - 作成した小ブロックをword長の領域にhash(・)を用いてmapping
  - 小ブロック$h(L_1^k)$/$h(L_2^k)$をAND演算を用いて高速に共通項を探索

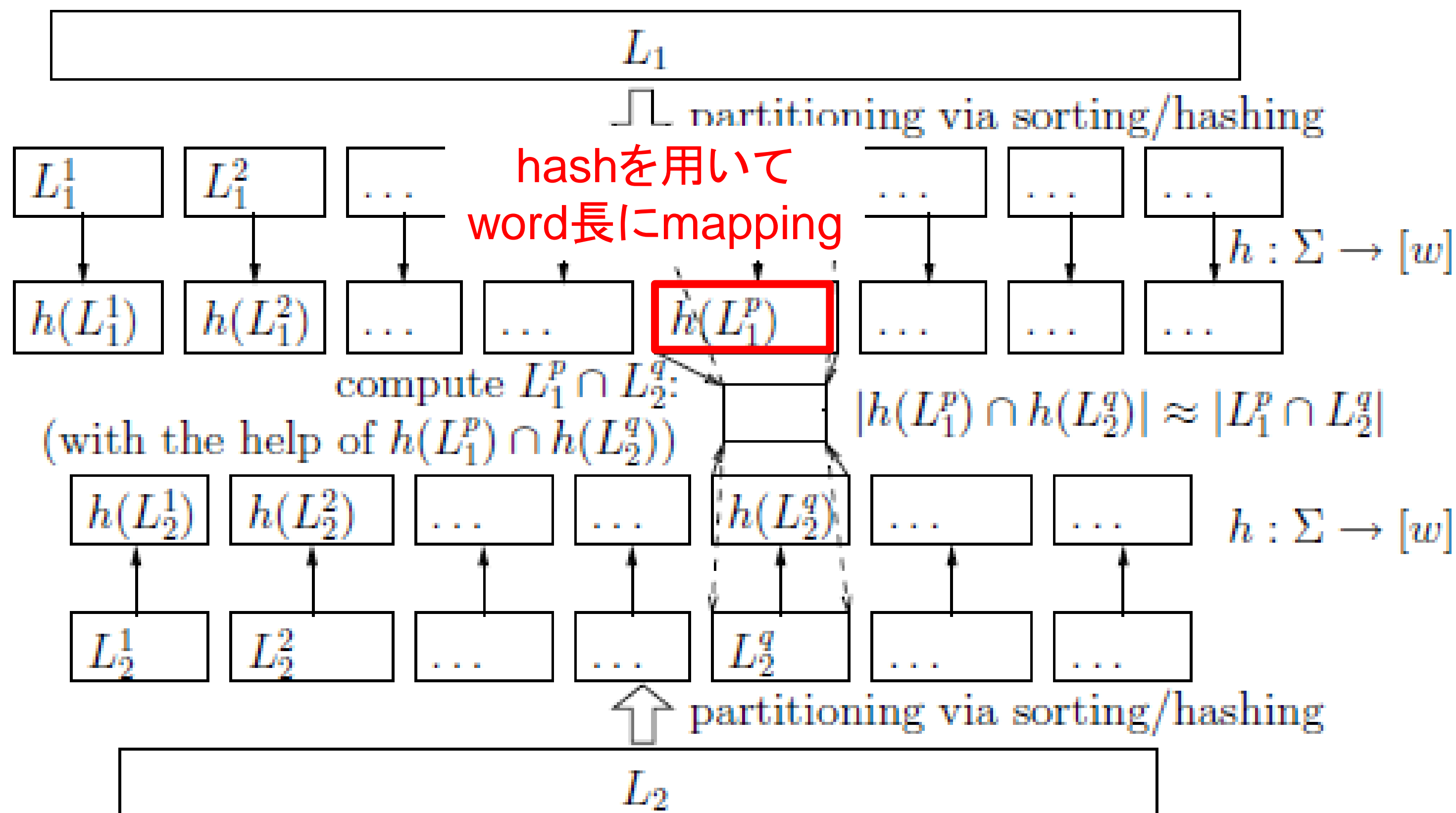
    →前提2. により大半がAND演算の結果が0(共通項無し)という結果になりSkip可能

2. Fast Set Intersection in Memory



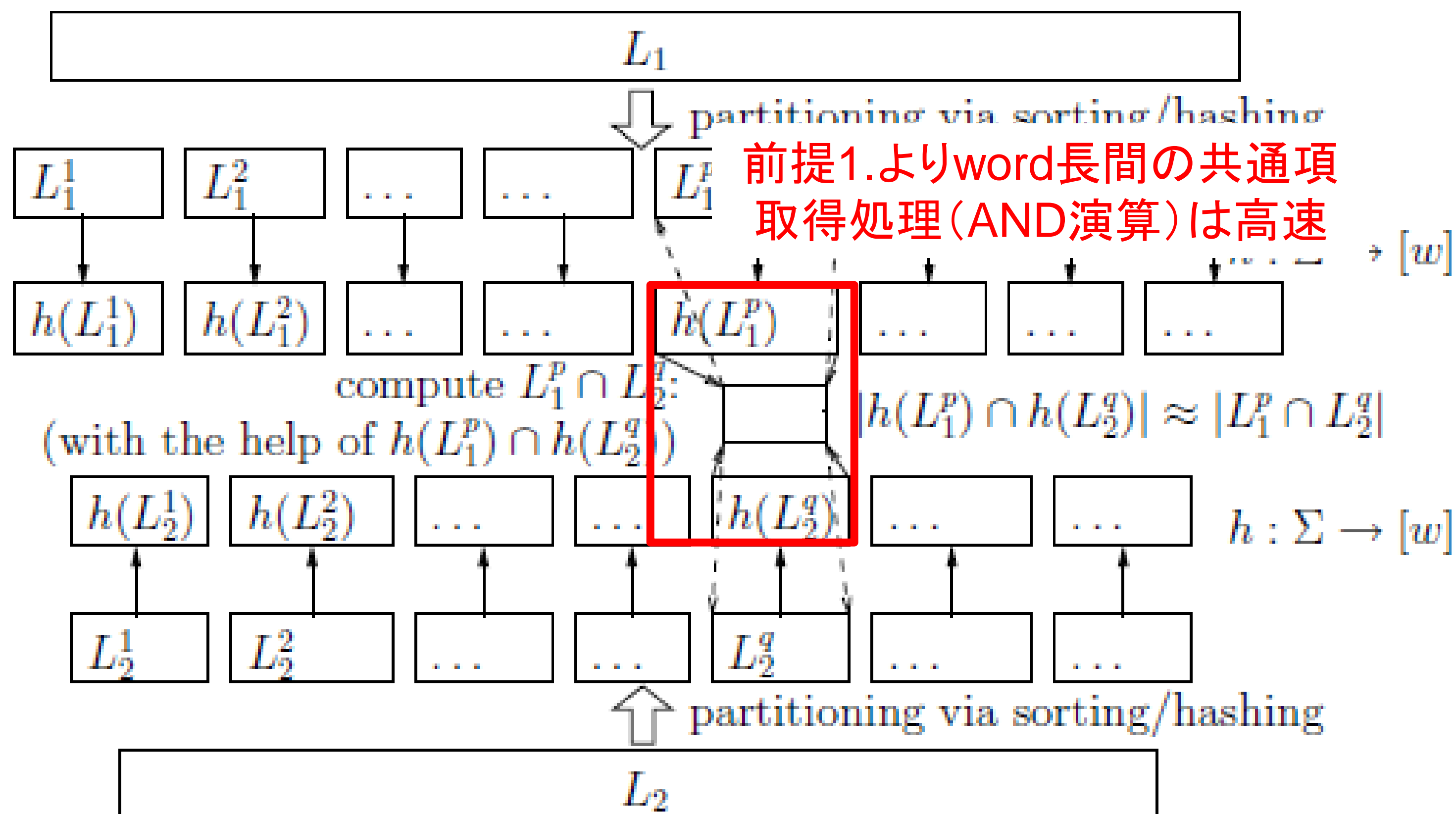
論文内のFigure.1から引用

2. Fast Set Intersection in Memory



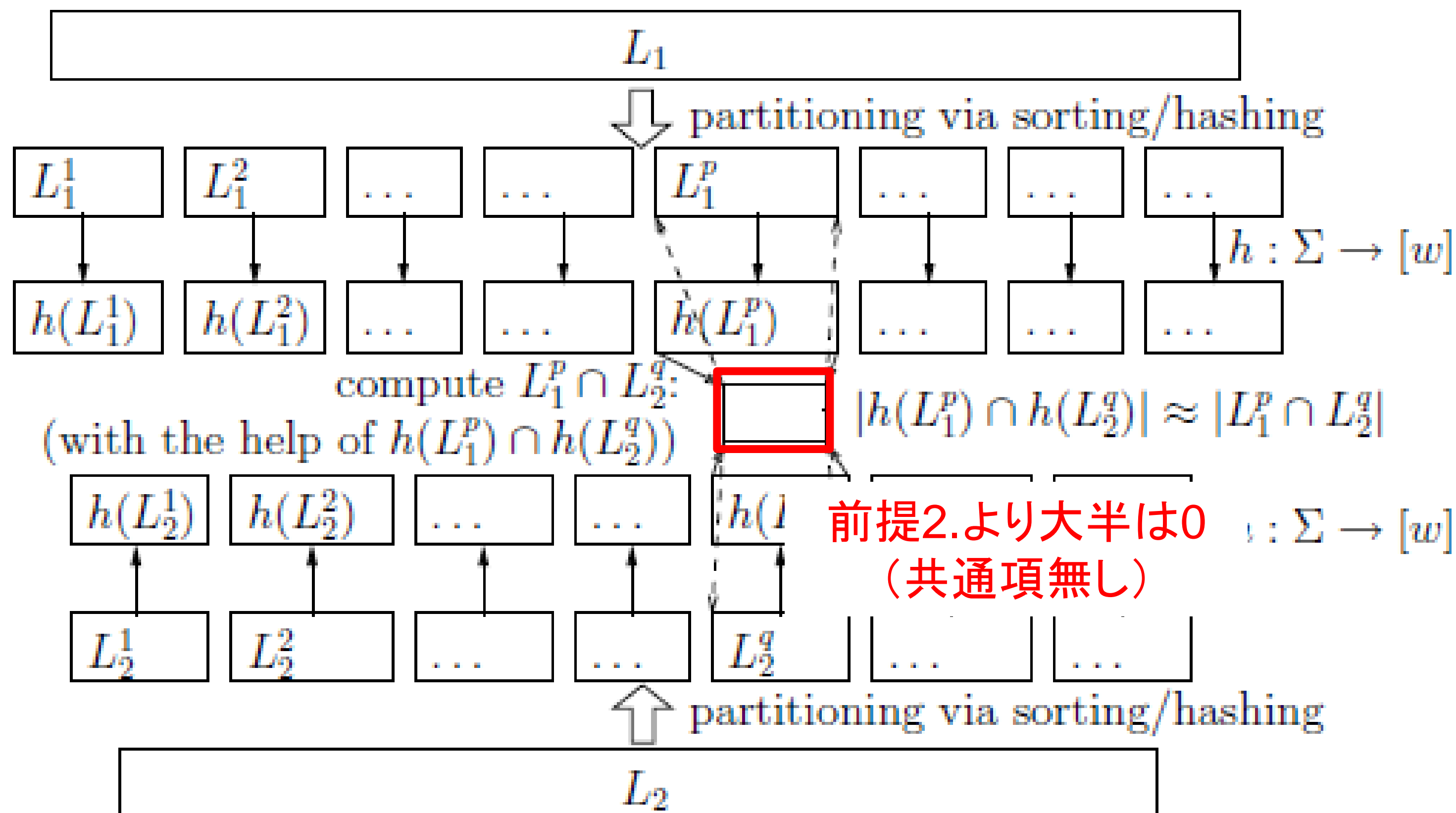
論文内のFigure.1から引用

2. Fast Set Intersection in Memory



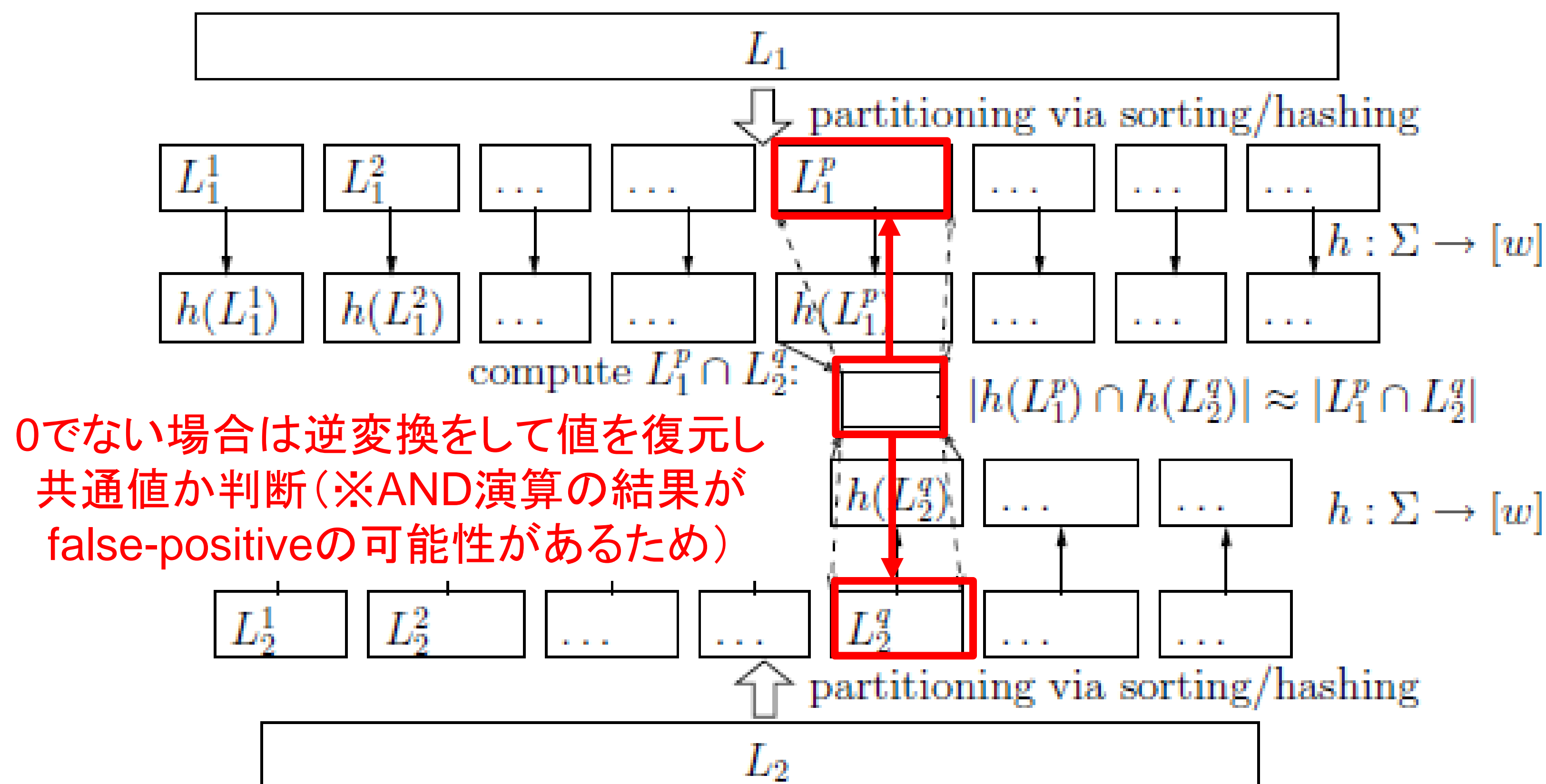
論文内のFigure.1から引用

2. Fast Set Intersection in Memory



論文内のFigure.1から引用

2. Fast Set Intersection in Memory



論文内のFigure.1から引用

2. Fast Set Intersection in Memory

- 評価方法
  - 同等サイズの2つのリストの共通項の抽出
  - 結果サイズは1%に固定
- 従来手法と提案手法の比較
  - 従来手法: Merge/SkipList/Hash/BPP/Adaptive/Lookup
  - 提案手法: IntGroup(理論計算量重視)/RanGroupScan(Practical重視)

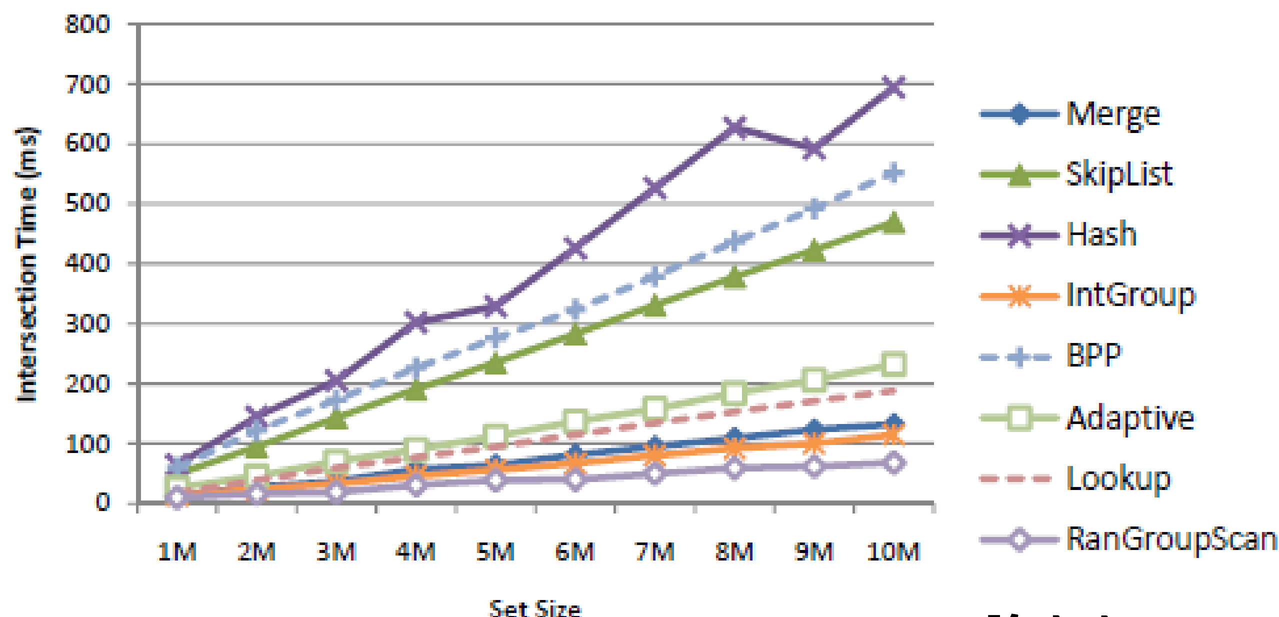


Figure 4: Varying the Set Size

論文内のFigure.4から引用

2. Fast Set Intersection in Memory

- 評価方法
  - 同等サイズの2つのリストの共通項の抽出
  - 結果サイズは1%に固定
- 従来手法と提案手法の比較
  - 従来手法: Merge/SkipList/Hash/BPP/Adaptive/Lookup
  - 提案手法: IntGroup(理論計算量重視)/RanGroupScan(Practical重視)

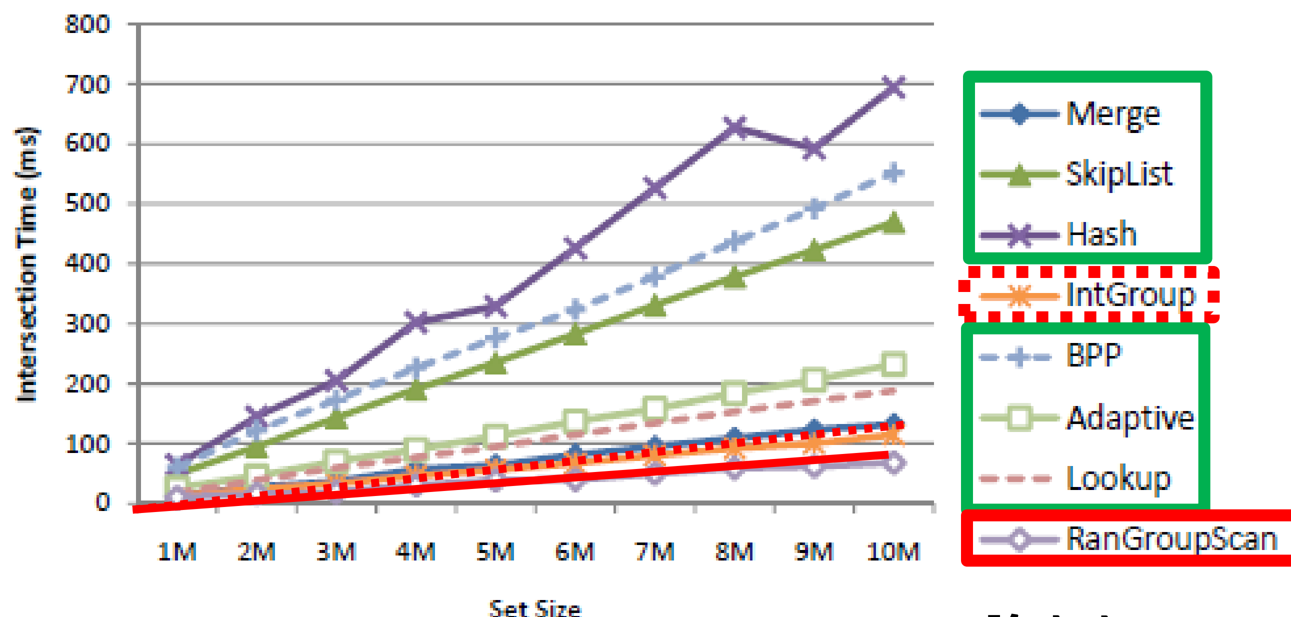


Figure 4: Varying the Set Size

論文内のFigure.4から引用

Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann (Technische Universität München)

3. Efficiently Compiling Efficient Query Plans for Modern Hardware

- A overview
  - **従来DBのquery-compilerはCPUに対するbranch-penaltiesやlocalityを考慮しない**ため，手続き型言語で記述された同等の処理（Hand-Coded C++）と比較して遅い，そこで**CPU最適化されたquery-compilerを提案**し，OLAP向けクエリに対して**cache-miss/branch-miss数を減らし，実行時間の改善**
- Contribution
  - **data-centricな処理flow**，極力CPUレジスタにdataを残す
  - **query処理の高locality（dataをPUSH）**，従来のiterator-model（Volcano-style）はdataをPULLするが，この処理方式はlocalityの観点で×[1][11][2][16]

3. Efficiently Compiling Efficient Query Plans for Modern Hardware

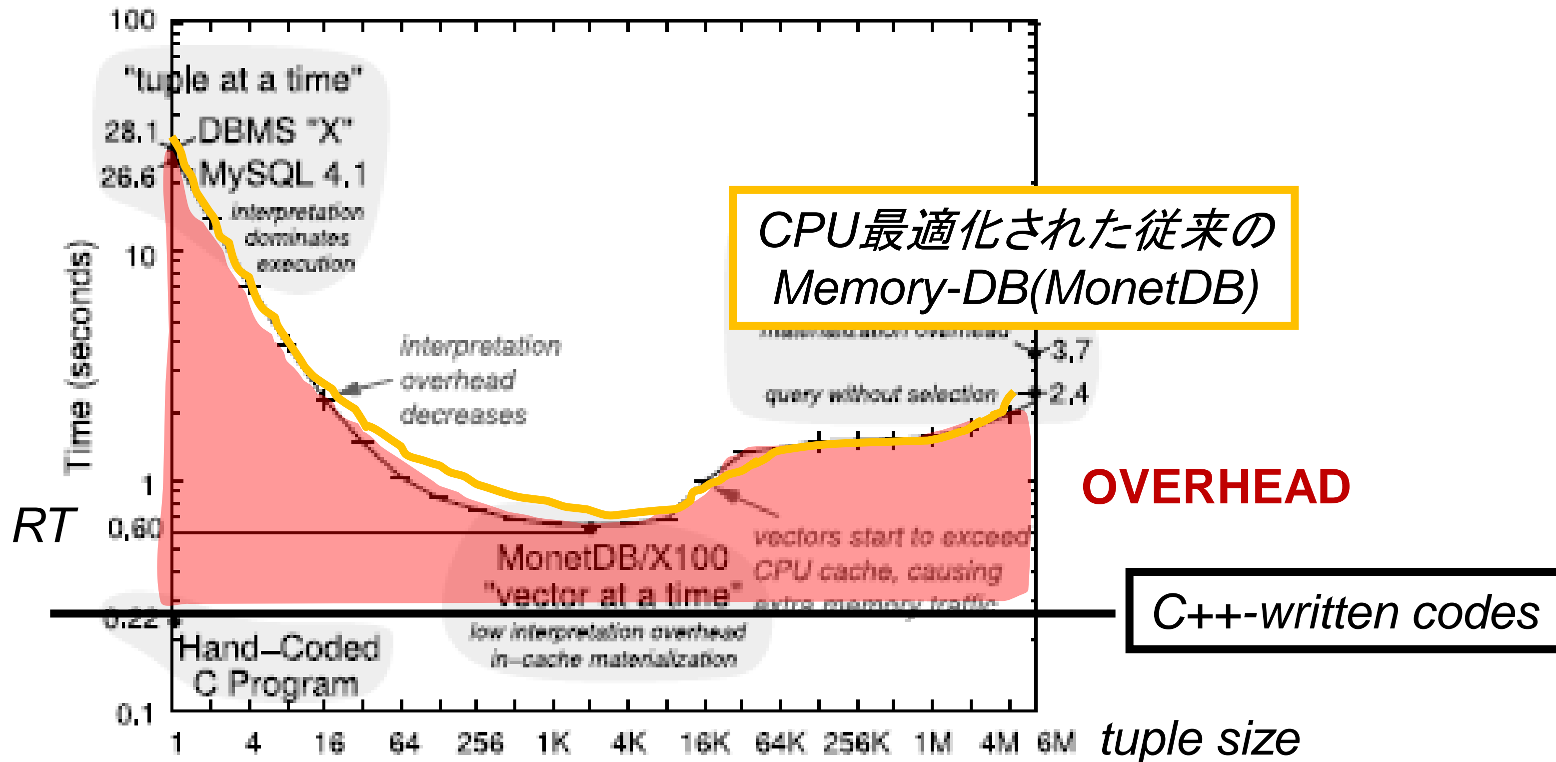


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])

論文内のFigure.1から引用

3. Efficiently Compiling Efficient Query Plans for Modern Hardware

- A basic idea – data-centricな実行planの生成
  - <u>メモリアクセスを最小化（registerの利用を最大化）</u>可能な実行flowを選択

一度registerにloadしたdataを極力unloadしないように実行flowを再構築

- next()を利用したiterator-modelからconsume()/produce()を用いた新しい実行モデルの提案

```
select   *
from     R1,R3,
         (select   R2.z,count(*)
         from      R2
         where     R2.y=3
         group by R2.z) R2
where    R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

Figure 2: Example Query

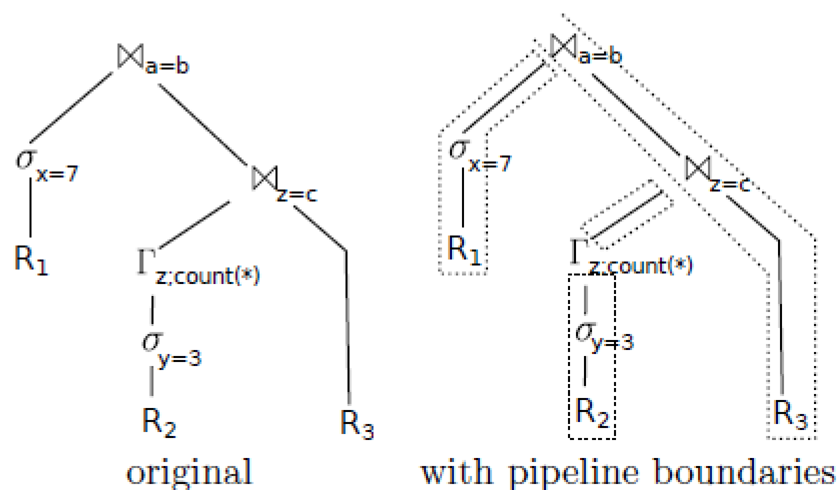


Figure 3: Example Execution Plan for Figure 2

論文内のFigure.3から引用

3. Efficiently Compiling Efficient Query Plans for Modern Hardware

- A basic idea – data-centricな実行planの生成
  - <u>**メモリアクセスを最小化（registerの利用を最大化）**</u>可能な実行flowを選択

一度registerにloadしたdataを極力unload しないように実行flowを再構築

- next()を利用したiterator-modelからconsume()/produce()を用いた新しい実行モデルの提案

```
select      *
from        R1,R3,
            (select     R2.z,count(*)
            from        R2
            where       R2.y=3
            group by R2.z) R2
where       R1.x=7 and R1.a=R3.b and R2.z
```

Figure 2: Example Query

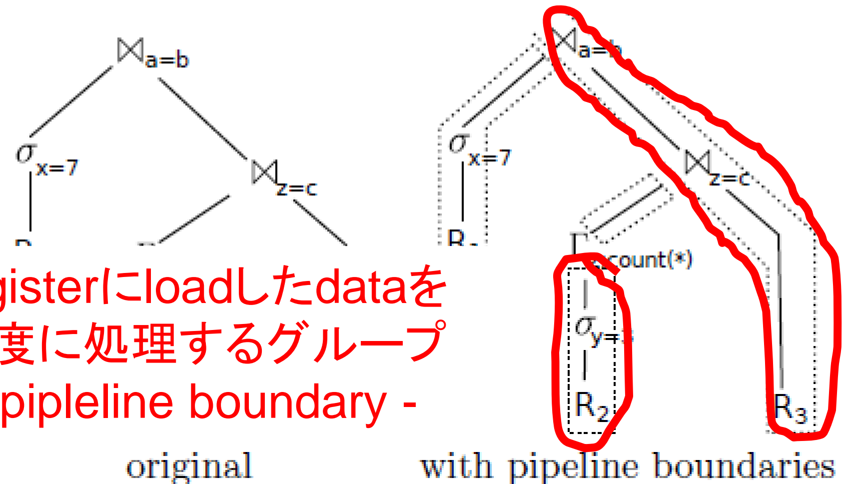


Figure 3: Example Execution Plan for Figure 2

論文内のFigure.3から引用

3. Efficiently Compiling Efficient Query Plans for Modern Hardware

- 評価はOLTP/OLAPのHybridなworkloadを利用[5]
  - workloadの詳細はAppendix.D
- *valgrindでmicro-benchmarkを実施
  - cache-miss/branch-miss/実行命令数(I refs)を測定

*仮想マシンを用いたCPUプロファイラ

		Q1		Q2		Q3	
		LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB
命令に関する	branches	19,765,048	144,557,672	37,409,113	114,584,910	14,362,660	127,944,656
	mispredicts	188,260	456,078	6,581,223	3,891,827	696,839	1,884,185
	I1 misses	2,793	187,471	1,778	146,305	791	386,561
データに関する	D1 misses	1,764,937	7,545,432	10,068,857	6,610,366	2,341,531	7,557,629
	L2d misses	1,689,163	7,341,140	7,539,400	4,012,969	1,420,628	5,947,845
	I refs	132 mil	1,184 mil	313 mil	760 mil	208 mil	944 mil

論文内のTable.3から引用

- 性能(throughputs/応答性能)も従来手法に対して優位
  - 論文内とTable.1とTable.2参照

3. Efficiently Compiling Efficient Query Plans for Modern Hardware

- 評価はOLTP/OLAPのHybridなworkloadを利用[5]
  - workloadの詳細はAppendix.D
- *valgrindでmicro-benchmarkを実施
  - cache-miss/branch-miss/実行命令数(I refs)を測定

*仮想マシンを用いたCPUプロファイラ

		Q1		Q2		Q3	
		LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB
命令に関する	branches	19,7[illegible]	[illegible]7,672	37,4[illegible]	[illegible]4,910	14,3[illegible]	[illegible]4,656
	mispredicts	1[illegible]	[illegible]6,078	6,5[illegible]	[illegible]1,827	6[illegible]	[illegible]4,185
	I1 misses	2,7[illegible]	187,471	1,7[illegible]	146,305	[illegible]	386,561
データに関する	D1 misses	1,764,9[illegible]	7,545,432	10,068,8[illegible]	6,610,366	2,341,5[illegible]1	7,557,629
	L2d misses	1,689,163	7,341,140	7,539,400	4,012,969	1,420,628	5,947,845
	I refs	132 mil	1,184 mil	313 mil	760 mil	208 mil	944 mil

大幅改善 大幅改善 大幅改善

論文内のTable.3から引用

- 性能(throughputs/応答性能)も従来手法に対して優位
  - 論文内とTable.1とTable.2参照

PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

Jason Sewall (Intel Corporation), Jatin Chhugani (Intel Corporation), Changkyu Kim (Intel Corporation), Nadathur Satish (Intel Corporation), Pradeep Dubey (Intel Corporation)

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

- A overview
  - 近年の並列性の増加傾向を背景に，**latch処理の非scalable性**が問題視されている，そこでin-memory環境を前提としたquery（retrieve/update/detele）の**batch処理向けlatch-free B+Treeを提案**し，従来手法と比較して**2.3X-19Xの性能改善**
- Contribution
  - **latch-free構造**，同期制御はBSP（Bulk Synchronous Parallel）によるLocal Computation/Communication/Barrierで構成
  - **並列環境（multi-/many-cores）志向な構造**，cache-awareなデータ配置とSIMDによる並列処理
  - **最新の並列環境による評価**，Many-Integrated Core（MIC）architectureであるIntel Knights Ferryに提案手法を適用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

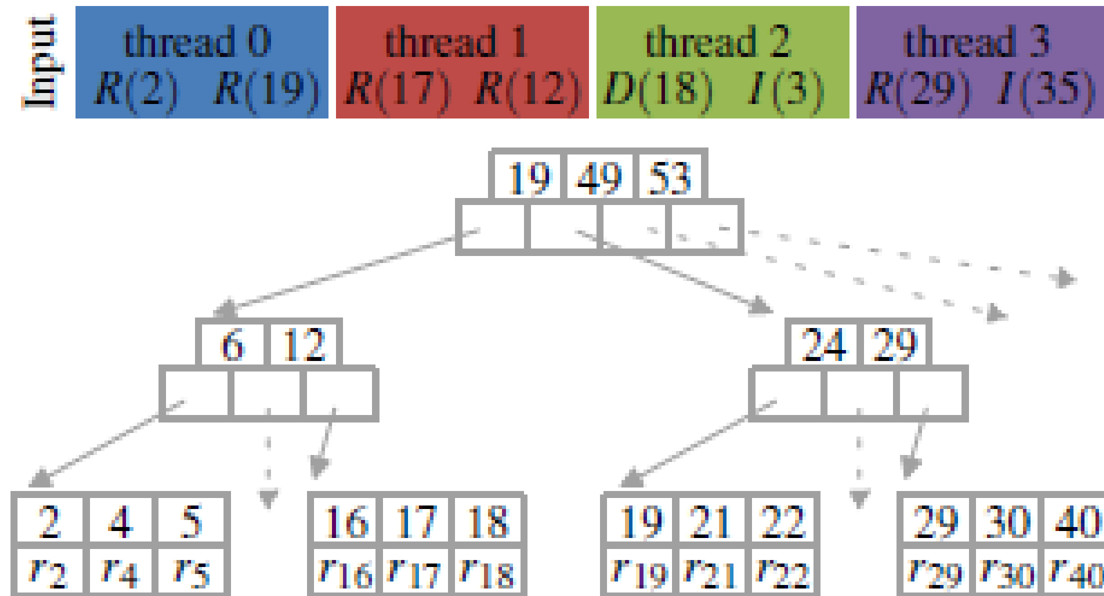
- A basic idea
  - batchで入力されるk個の順序を持つ$O=o_0, o_1, o_2, \ldots, o_{k-1}$（$o \in$ {retrieve, update, delete}）に対して，**全てのretrieve処理を初めに完了**させ，update/deleteで**conflictしている結果を後で修正**することでserializabilityを確保

理由: 前半の処理をretrieve処理のみにすることで，同期のpenaltyを最小化し，従来手法のCPU最適化処理（cache-aware/SIMD）で性能を最大化するアプローチ

※conflictが少ないことが前提

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

R: Retrieve
I: Insert
D: Delete

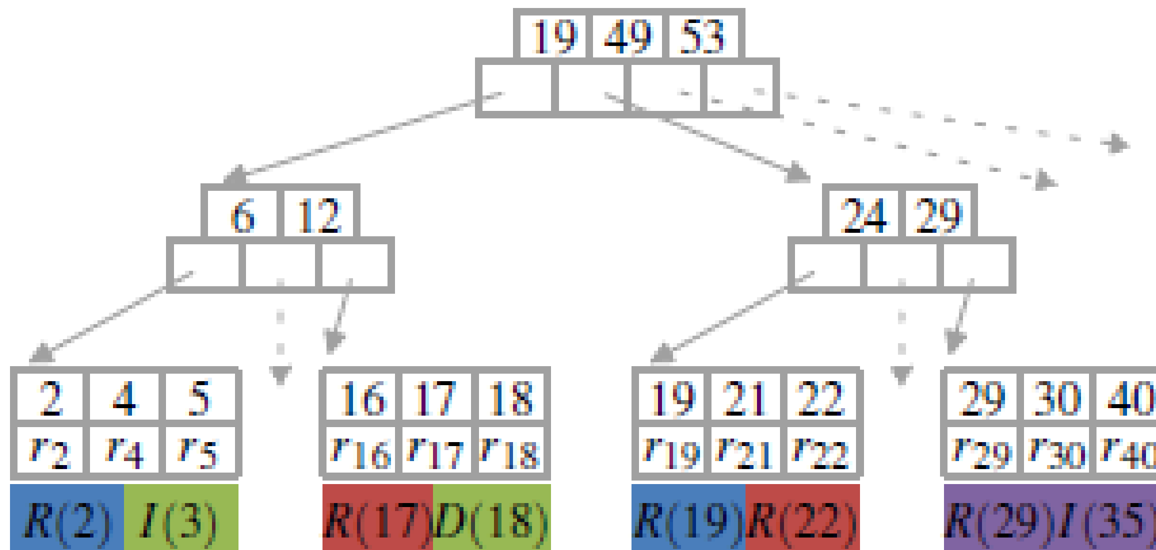


(a) Divide queries among threads

論文内のFigure.1から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

R: Retrieve
I: Insert
D: Delete



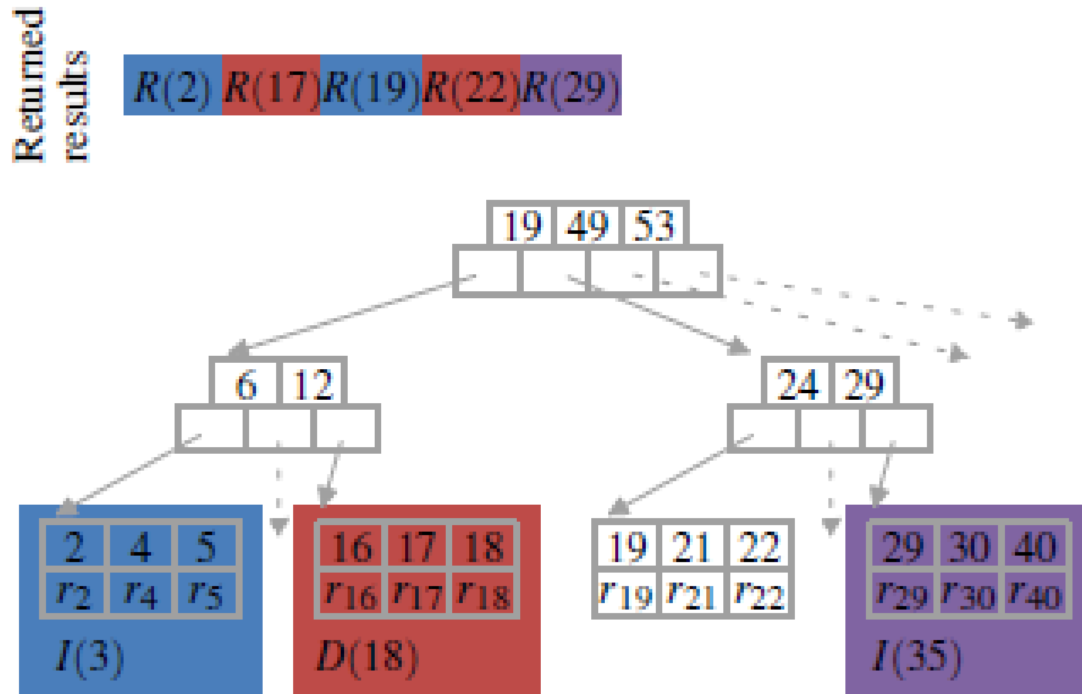
(b) Search leaves; retrieve results

先にすべてのretrieve処理を完了

論文内のFigure.1から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

R: Retrieve
I: Insert
D: Delete



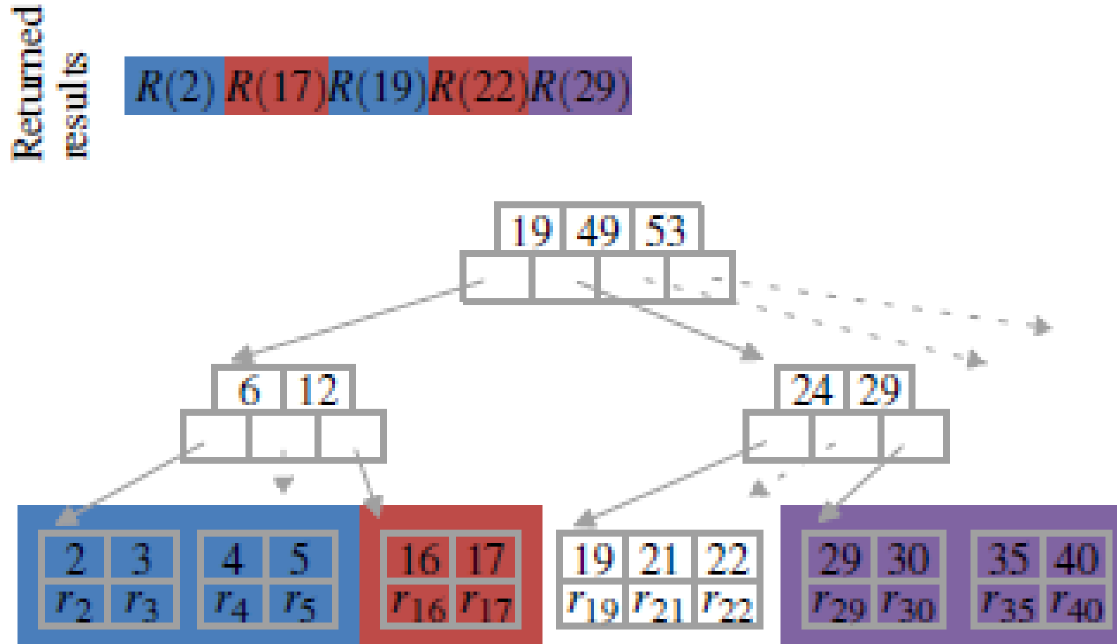
(c) Redistribute and ensure order of ops

Serializablityの確保

論文内のFigure.1から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

R: Retrieve
I: Insert
D: Delete

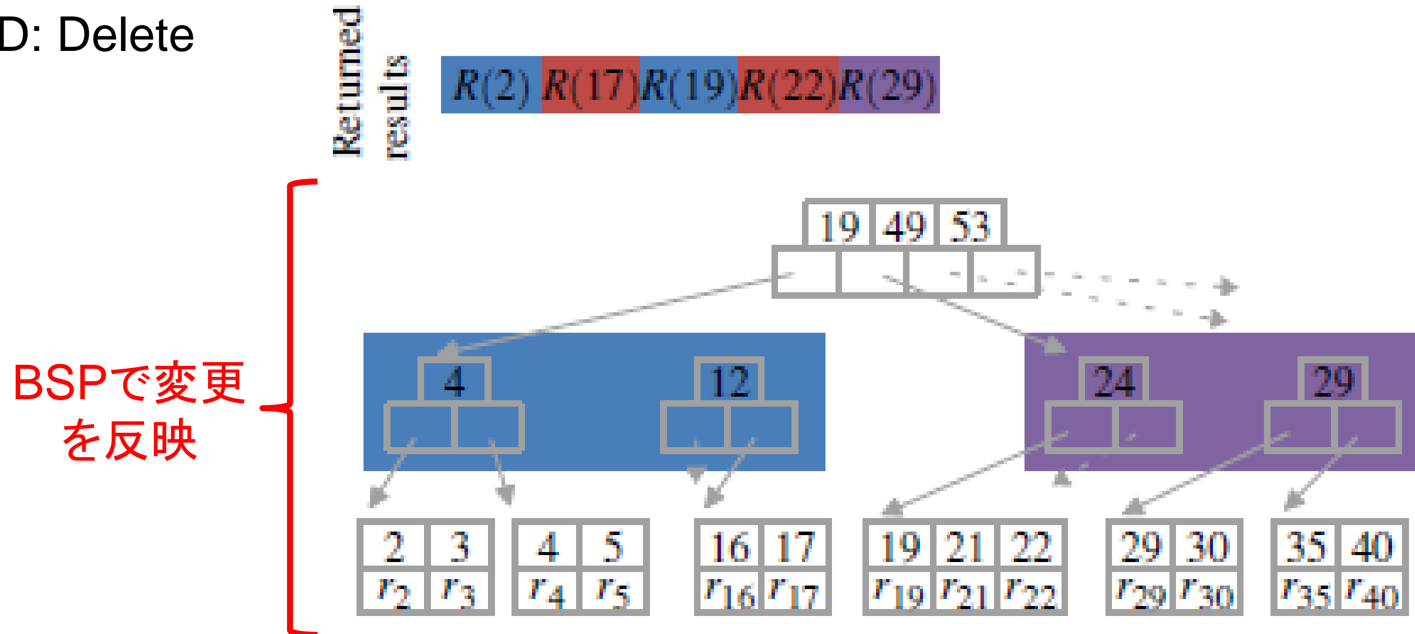


BSPで変更を反映

論文内のFigure.1から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

R: Retrieve
I: Insert
D: Delete

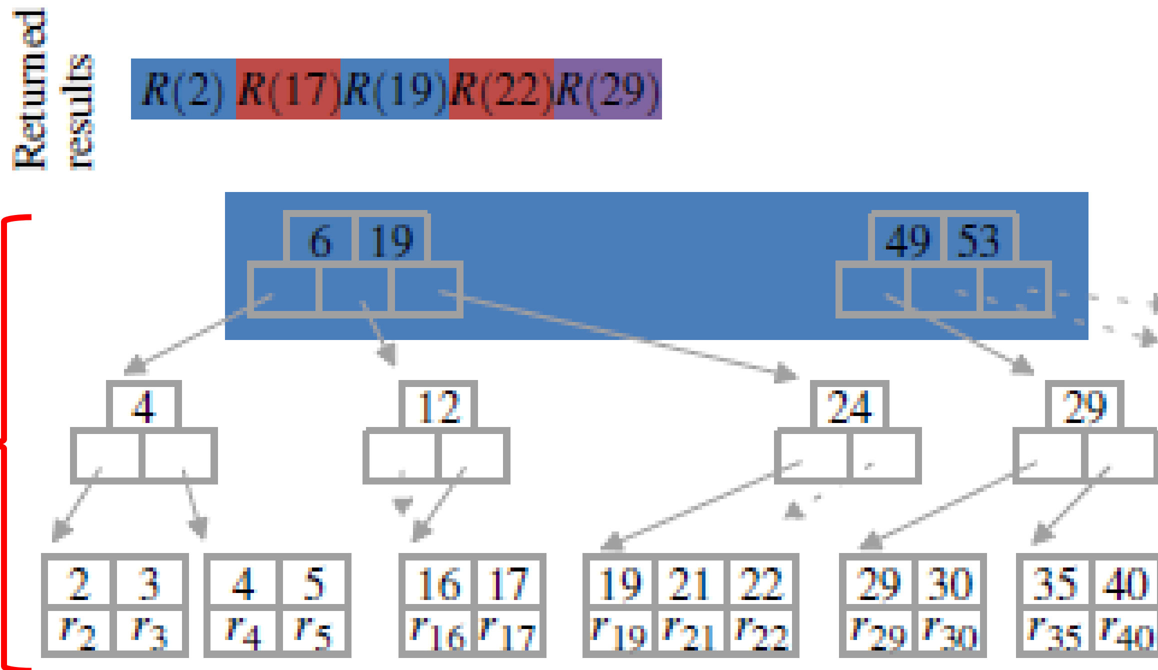


(e) Modify internal nodes; redistribute

論文内のFigure.1から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

R: Retrieve
I: Insert
D: Delete



(f) Modify the root

論文内のFigure.1から引用

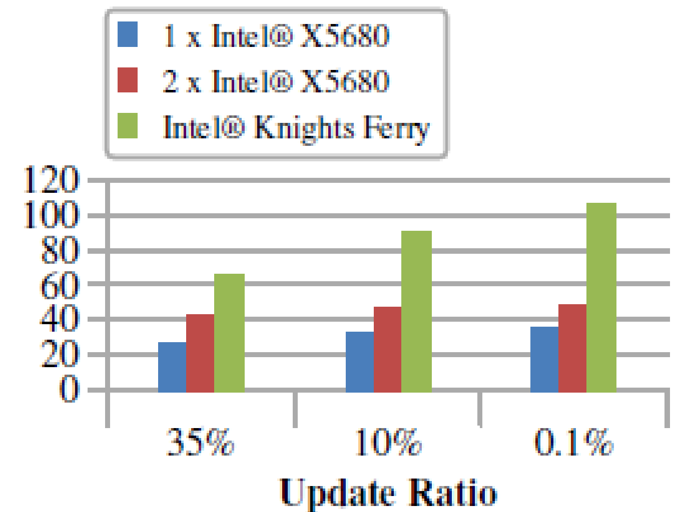
4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

- batch処理を前提としているため，8192個のqueryを1単位として入力
- 2つの環境を用いて評価を実施
  - CPU: Intel Xeon 5680

    2 sockets, total **12-cores**, **128bit-SIMD**, and 96GiB RAM
  - MIC: Intel Knights Ferry

    x86-based **32-cores**, and **512bit-SIMD**

論文内のFigure.2から引用



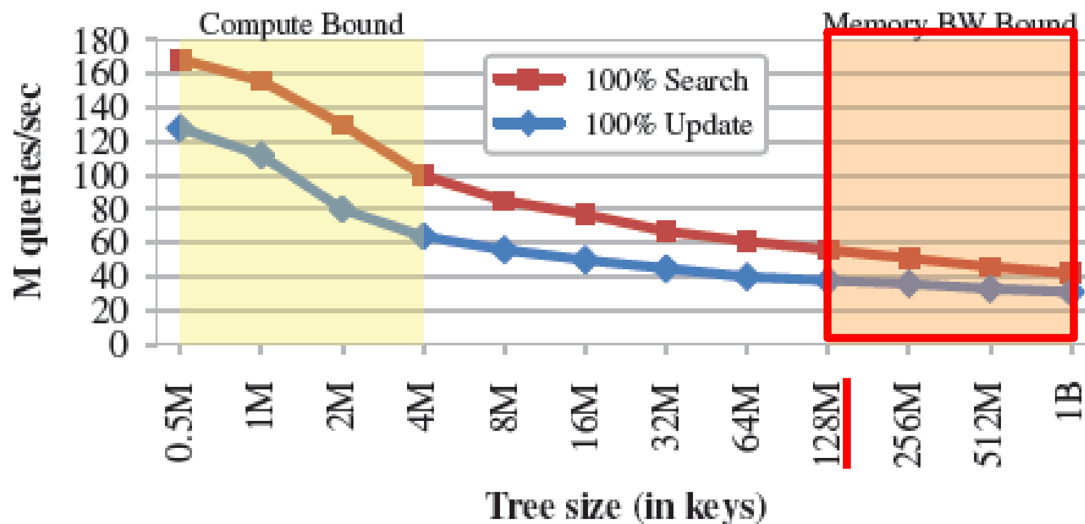
論文内のFigure.5から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

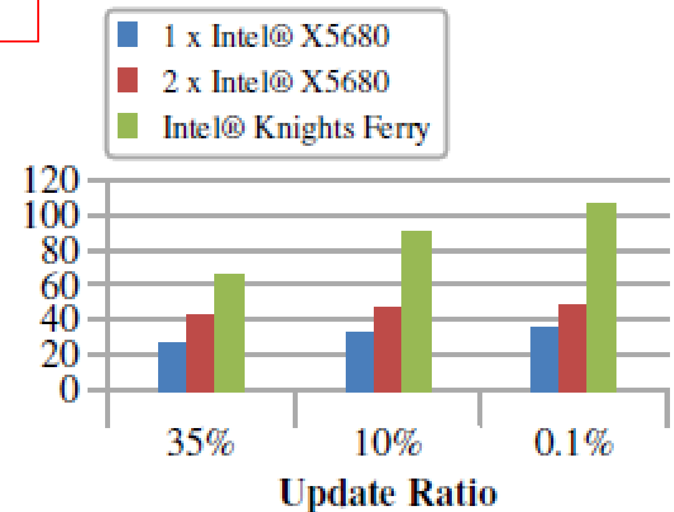
- batch処理を前提としているため，8192個のqueryを1単位として入力
- 2つの環境を用いて評価を実施
  - CPU: Intel Xeon 5680

    2 sockets, total **12-cores, 128bit-SIMD**, and 96GiB RAM

NOTICE1: sizeが128Mでbottleneck-shiftが発生
→コア数の多い環境ではlocalityを考慮しないと
コア数増加に対して性能スケールしない



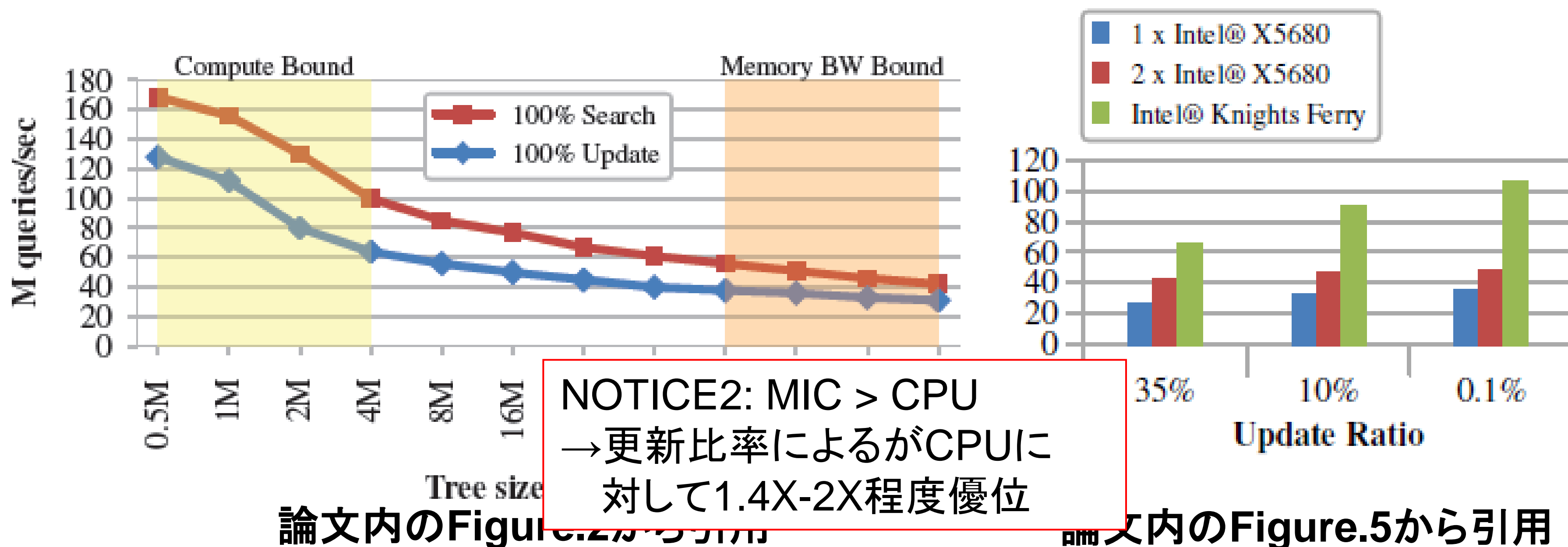
論文内のFigure.2から引用



論文内のFigure.5から引用

4. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

- batch処理を前提としているため，8192個のqueryを1単位として入力
- 2つの環境を用いて評価を実施
  - CPU: Intel Xeon 5680
    2 sockets, total **12-cores**, **128bit-SIMD**, and 96GiB RAM
  - MIC: Intel Knights Ferry
    x86-based **32-cores**, and **512bit-SIMD**



論文内のFigure.2から引用　　論文内のFigure.5から引用

まとめと，所感

- この分野は研究/産業の両側面で非常に重要視
  - 研究的: DB界隈としては比較的に課題認識を共有できているので reviewerに背景を納得されやすい
  - 産業的: 2012で最も注目すべき10個の技術に”in-memory” tech.
    - Gartner Inc.による報告

      http://www.techrepublic.com/blog/hiner/look-out-the-10-rising-tech-trends-of-2012/9470?tag=content;roto-fd-feature
- 一方で･･･
  - 詳細なHardwareの知識が必要
    - 論文内には結構マニアックなチューニングの説明も･･･
  - 正確な評価が難しい
    - 測定したい値に対して、ツールも様々（oprofile/perf/VTUNE/valgrind･･･）
  - micro-benchmark結果は良くても、全体としての性能が出にくい
    - in-memoryの処理なので数倍高速化してもimpactが少ない