ICDE 勉強会 2013
R32 Data storage

引田諭之（東工大）

※資料中の図は論文からの引用です

TBF: A Memory-Efficient Replacement Policy for Flash-based Caches

Cristian Ungureanu, Biplob Debnath, Stephen Rago, Akshat Aranya

NEC Laboratories America

論文概要

- 背景・動機
  - SSD上をキャッシュに利用すると性能向上に有効
  - LRUやCLOCKでは大量なキャッシュデータの全メタデータをRAMに保持するのが困難

Caching Policy	Metadata Overhead	Object Size			
		1 MB	4 KB	1 KB	256 B
LRU	24 bytes	48 MB	12 GB	48 GB	192 GB
CLOCK	8 bytes	16 MB	4 GB	16 GB	64 GB
Our Algorithm	1 byte	2 MB	0.5 GB	2 GB	8 GB

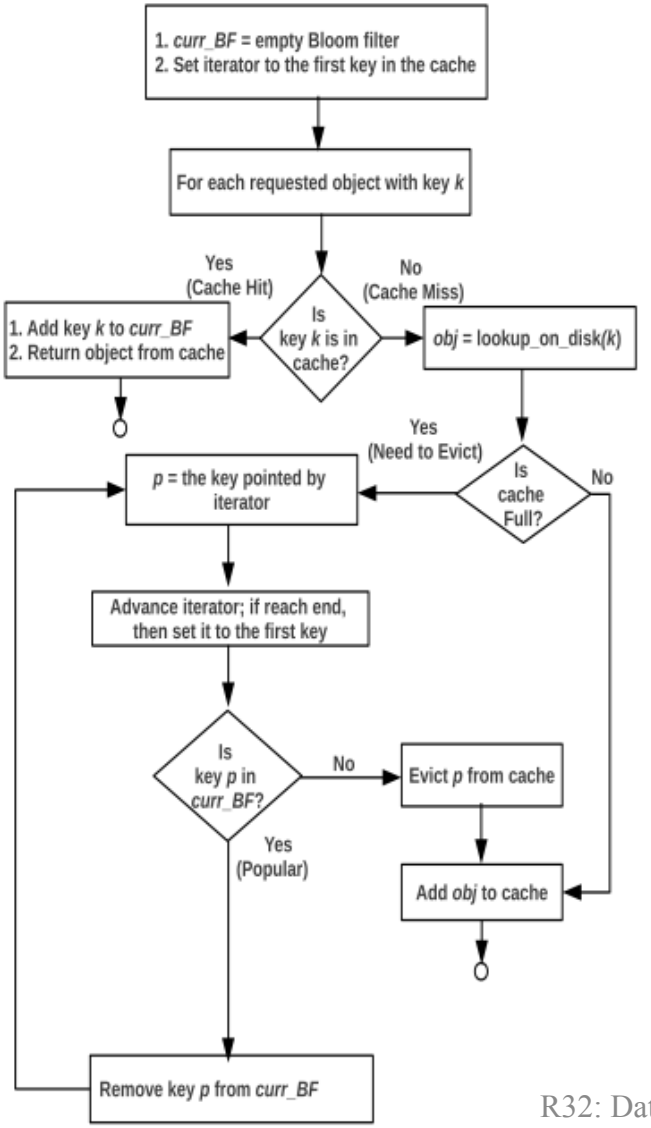
2TB cacheにおけるメタデータ（アクセス情報）のメモリ使用量

- 論文の貢献
  - RAMを節約するキャッシュ置換ポリシーを提案
  - Bloom Filterに削除操作を追加した二つの手法を導入
    - **BFD - Bloom Filter with Deletion**
    - **TBF – Two Bloom sub-Filter**
  - YCSBを用いた性能評価

BFD, TBDの概要

- BFD，TBFはデータへのアクセス情報（最近アクセスされたかどうか）を保持するだけ
- キャッシュデータのインデックス等は採用するKey-Value Storeに依存する
- BFDはBloom Filter(BF)にDeletionを追加
- TBFは二つのBloom Filterを用いる
  - Deletionもサポート
- Deletionによりfalse negativeも発生するが問題なし
  - BFD, TBDはキャッシュではないから
  - もしBFから削除されても影響が出る前にまたBFに追加される確率が高い

BFD : Bloom Filiter with Deletion

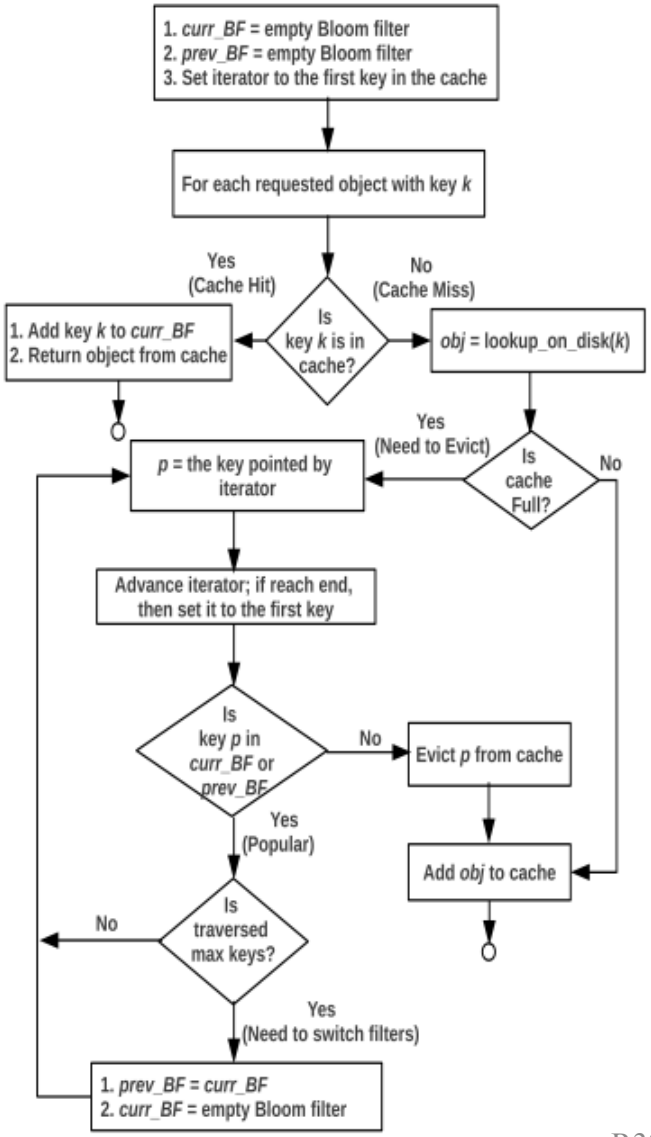


- 検索キーkがCache Hit
  - BloomFilter(BF)に追加
  - cacheからデータを返す
- キーkがCache Miss
  - cacheに空きがあれば追加
  - cacheに空きがない

LOOP

- KVSをトラバースしキーkをBFで検索
  - 見つからなければキーkのオブジェクトを置換
  - 見つかればBFからキーkを削除してLOOPを繰り返す

TBF : Two Bloom sub-Filter



- 検索キーkがCache Hit
  - CurrentBF(curr_BF)に追加
  - cacheからデータを返す
- キーkがCache Miss
  - cacheに空きがあれば追加
  - cacheに空きがない
    - KVSをトラバースしキーkをcurr_BF，prev_BFで検索
      - 見つからなければキーkのオブジェクトを置換
      - 見つかればLOOPを繰り返す※

※もしキャッシュサイズ分のオブジェクトを検索したらcurr_BFとprev_BFを交換し，curr_BFを空にする

性能評価

- TBFによる置換対象エントリ選択が有効に働いている
- 置換対象を検索するkey traversはキャッシュヒット率が高い程多くなる⇒最近アクセスされたデータに多く出会うため

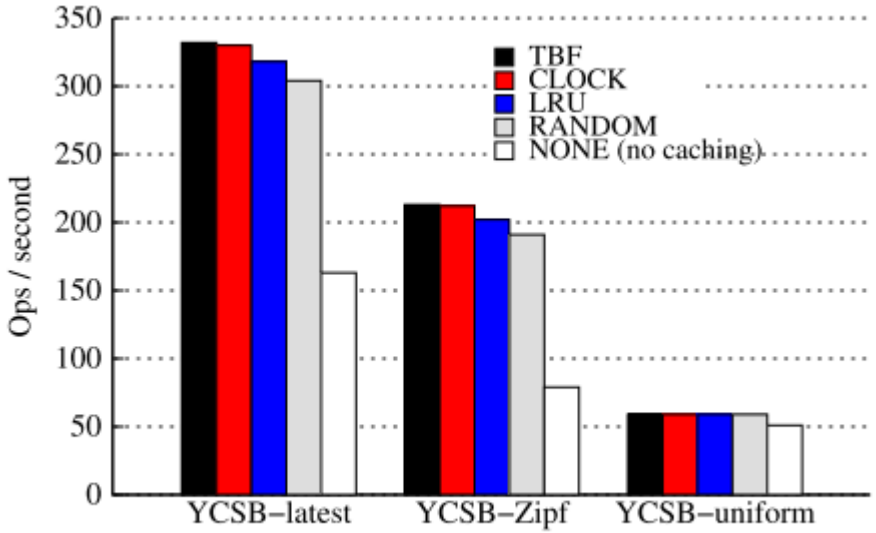


Fig. 7: Performance of various caching policies

TABLE IV: Cache sizes used for simulation

	Cache Size (in number of objects)		
Workload	Small	Medium	Large
All YCSB	1,000,000	2,000,000	4,000,000
Financial	1,000	10,000	100,000
MSR proxy	4,000	20,000	40,000

TABLE V: Memory usage for various caching policies

Cache Policy	Memory Usage (GB)		
	Cache Policy	Java Heap	Total OS
LRU	3.05	7.0	8.0
CLOCK	1.5	5.8	6.8
TBF	0.15	4.1	5.1
RANDOM	0	3.8	4.8
NONE (no caching)	0	3.8	4.8

TABLE VI: Comparison of cache hit rates, and the average number of key traversed during TBF to find good eviction victims

Workload	Cache Hit Rate (%)								Average Number of Keys Traversed by TBF
	TBF		CLOCK		LRU		RANDOM		
	Simulated	Observed	Simulated	Observed	Simulated	Observed	Simulated	Observed	
YCSB-Uniform	10.0	10.0	10.1	10.0	10.0	10.0	10.0	10.0	1.12
YCSB-Zipf	77.3	77.7	77.3	77.6	77.0	77.0	74.9	74.4	1.39
YCSB-Latest	84.4	84.9	84.4	84.8	84.1	84.1	82.2	81.5	1.51

Fast Peak-to-Peak Behavior with SSD Buffer Pool

Jaeyoung Do[#1], Donghui Zhang[#2], Jignesh M. Patel[#1], David J. DeWitt[#3]

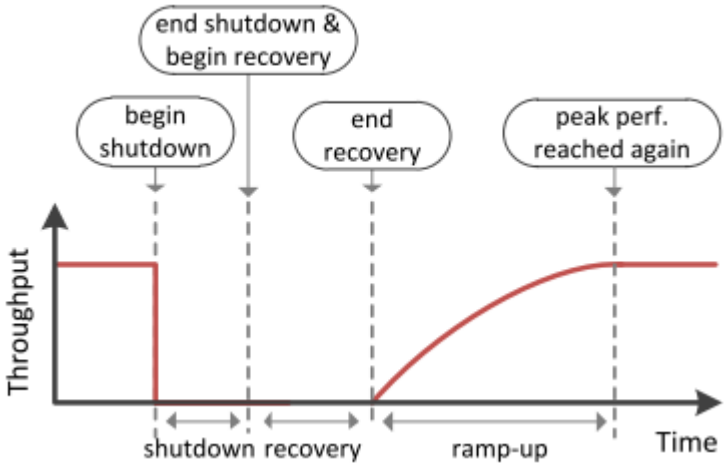
#1 *University of Wisconsin-Madison*

#2 *Paradigm4*

#3 *Microsoft Jim Gray Systems Lab*

論文概要

- 背景・動機
  - メモリ上のBuffer poolをSSDによって拡張する手法はDBMSのピーク性能向上に有効
  - リカバリ時や再起動時におけるピーク性能に達するまでの時間短縮が課題
    - 通常再起動時はSSD上のデータは破棄されるため



- 論文の貢献
  - SSD上のBuffer poolを効率よく再構築する手法を提案
    - LBR: Log-Based Restart
    - LVR: Lazy Verification Restart
  - 著者らの提案するSSDによるBuffer poolの拡張手法[sigmod 2011]に今回の提案手法を組み合わせて性能評価

SSD Buffer Pool Extension

- SIGMOD2011で提案
- メモリのBuffer poolから破棄されたページをSSDでキャッシュする
  - Dual-Write(DW):write-through policy
  - Lazy-Cleaning (LC)write-back policy

※本論文ではこの構成を前提としたリスタートアルゴリズムを提案

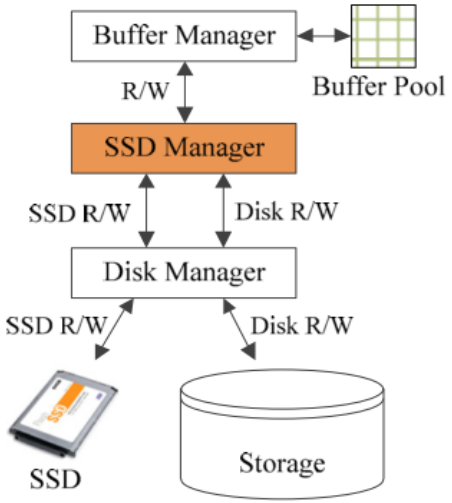
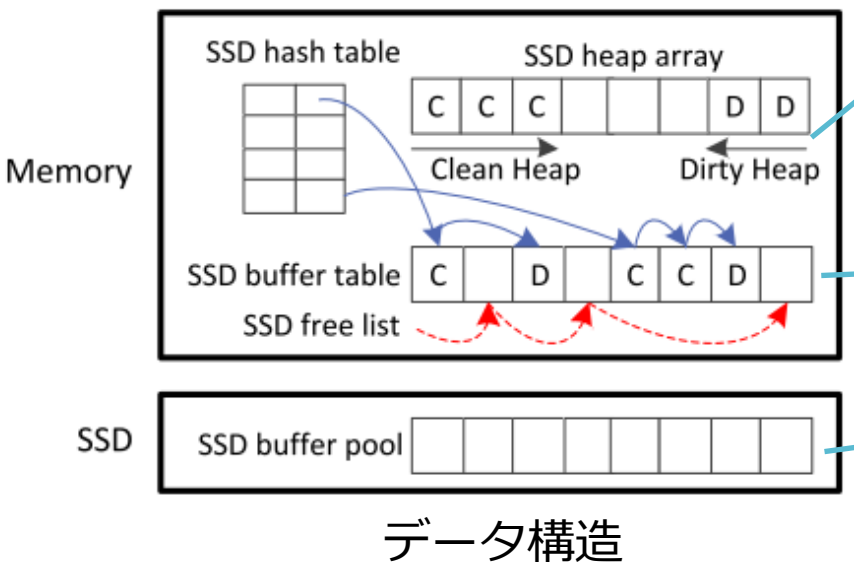


Fig.1 in *Turbocharging DBMS buffer pool using SSDs*



データ構造

- Clean heap: Clean FCへの参照．rootは最小のrecLSNを持つ
- Dirty heap: Dirty FCへの参照．rootは最小のnextToLastUseTimeを持つ

SSD buffer poolの各エントリに対応するレコード(Flash Cache:FCと呼ぶ) の配列

DBのページをキャッシュする配列．各エントリをフレームと呼ぶ

Restart Algorithms

- Memory-Mapped Restart (MMR)
  - SSD buffer tableの一部をMemory-mapped fileとして保持
  - AnalysisフェーズでSSD buffer tableを復元する
  - 通常時における変更内容のフラッシュはコストが高い
- Log-Based Restart (LBR)
  - SSD buffer tableのチェックポイントの作成しSSD buffer tableへの更新もログに記録する
  - ログを用いてAnalysisフェーズでSSD buffer tableを復元する
- Lazy Verification Restart (LVR)
  - FC flusherスレッドを用いて非同期にSSD buffer tableを永続ストレージ(SSD)上のファイルに書き込む
  - リカバリ時はそのファイルをloadしてSSD buffer tableを復元

※ ベースとなるリカバリの仕組みはSQL Server 2012で実装しているARIESベースのアルゴリズム

評価（スループット）

- shutdownからリスタートした場合でのスループット
- TPC-CではSSDBPに対しLBR, LVRは90%以上の性能を維持
- TPC-EではMMRが最もオーバーヘッドが少ない

TPC-C

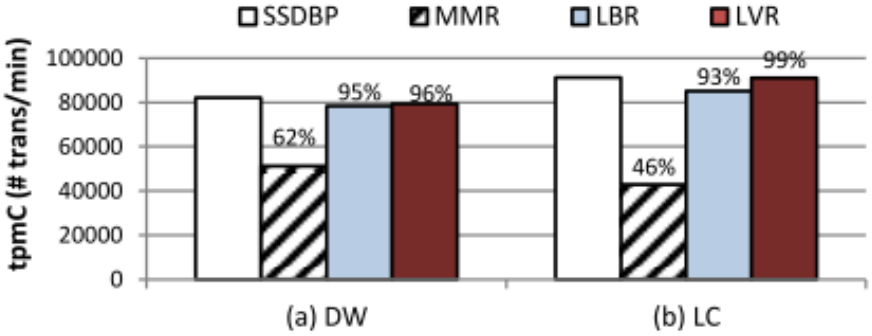


Fig. 10. TPC-C: Sustained throughput after restarting following a shutdown. The behavior when restarting after a crash is similar.

TPC-E

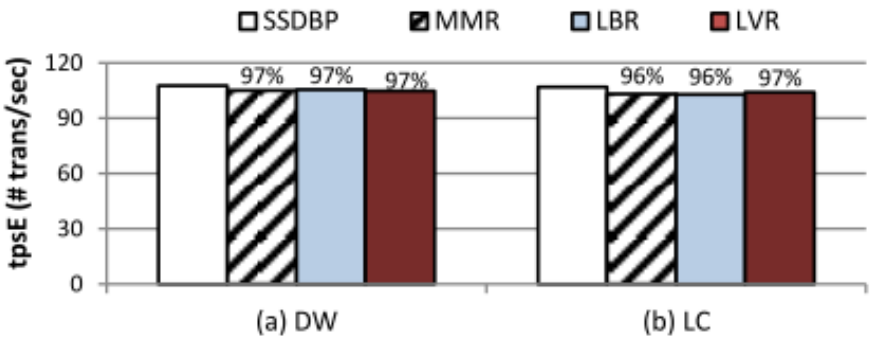
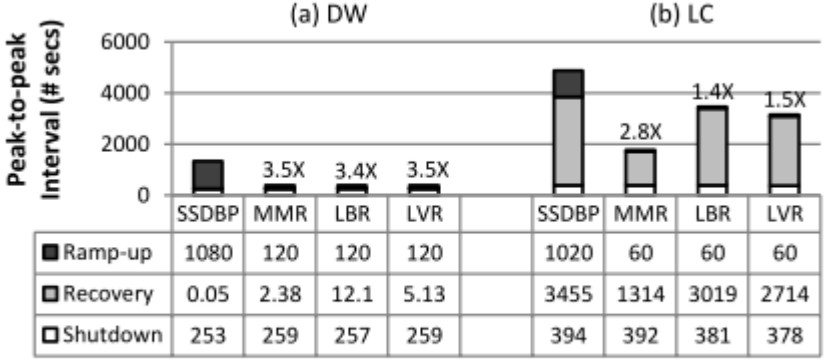


Fig. 13. TPC-E: Sustained throughput after restarting following a shutdown. The behavior when restarting following a crash is similar.

SSDBP：SIGMOD2011の提案手法．restart時はSSD buffer poolは空にする

評価：Peak-to-Peakまでの時間

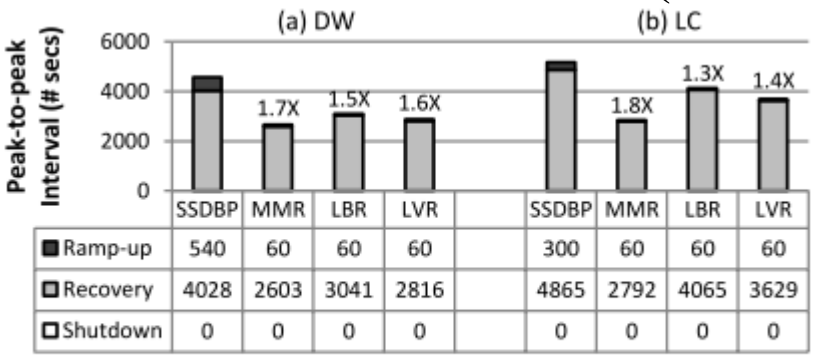
Recover from crash(TPC-C)



	SSDBP	MMR	LBR	LVR	SSDBP	MMR	LBR	LVR
Ramp-up	1080	120	120	120	1020	60	60	60
Recovery	0.05	2.38	12.1	5.13	3455	1314	3019	2714
Shutdown	253	259	257	259	394	392	381	378

Fig. 11. TPC-C: Peak-to-peak interval (in seconds) when restarting after a shutdown.

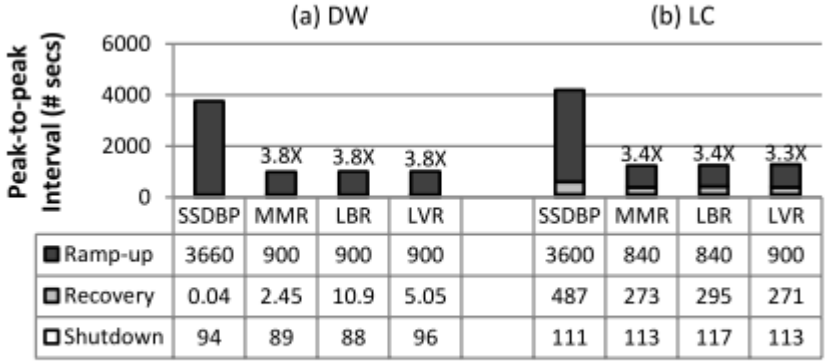
Recover from shutdown(TPC-C)



	SSDBP	MMR	LBR	LVR	SSDBP	MMR	LBR	LVR
Ramp-up	540	60	60	60	300	60	60	60
Recovery	4028	2603	3041	2816	4865	2792	4065	3629
Shutdown	0	0	0	0	0	0	0	0

Fig. 12. TPC-C: Peak-to-peak interval (in seconds) when restarting after a crash.

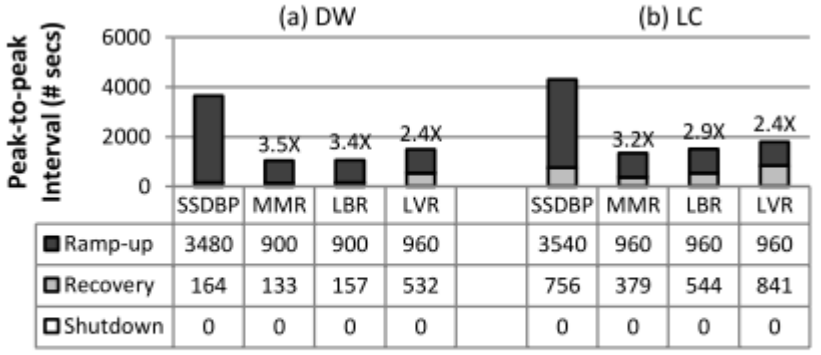
Recover from crash(TPC-E)



	SSDBP	MMR	LBR	LVR	SSDBP	MMR	LBR	LVR
Ramp-up	3660	900	900	900	3600	840	840	900
Recovery	0.04	2.45	10.9	5.05	487	273	295	271
Shutdown	94	89	88	96	111	113	117	113

Fig. 14. TPC-E: Peak-to-peak intervals (in seconds) when restarting after a shutdown.

Recover from shutdown(TPC-E)



	SSDBP	MMR	LBR	LVR	SSDBP	MMR	LBR	LVR
Ramp-up	3480	900	900	960	3540	960	960	960
Recovery	164	133	157	532	756	379	544	841
Shutdown	0	0	0	0	0	0	0	0

Fig. 15. TPC-E: Peak-to-peak interval (in seconds) when restarting after a crash.

SELECT Triggers For Data Auditing

Daniel Fabbri[#1], Ravi Ramamurthy[#2], Raghav Kaushik[#2]

#1 *University of Michigan*

#2 *Microsoft Research*

論文概要

- 背景・動機
  - DB上のsensitive dataへのアクセス監視をしたい
  - 従来のTriggerではsensitive dataの閲覧(SELECT)には適用出来ない
- 論文の貢献
  - SELECT Triggerの提案
  - TPC-Hでのベンチマークにより性能を評価

SELECT Trigger Specification

- SELECT Triggerの仕様
  - Audit Expressionにより監査されるべき“sensitive data”を指定

```
CREATE AUDIT EXPRESSION Audit_Alice AS
 SELECT *
 FROM Patients
 WHERE Name = ‘Alice’
 FOR SENSITIVE TABLE Patients,
 PARTITION BY PatientID
```

Patientsテーブル中で”Alice”という名前のデータがsensitiveであることを指定

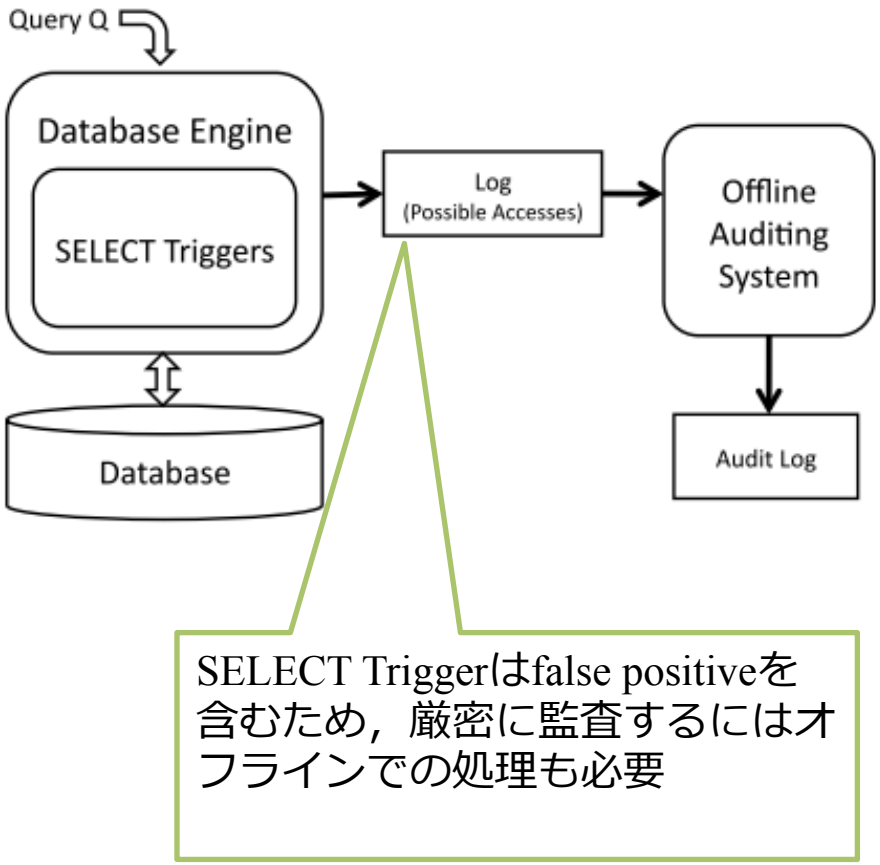
```
CREATE TRIGGER Log_Alice_Accesses
 ON ACCESS TO Audit_Alice AS
 INSERT INTO Log
 SELECT now(), userID(), sql(), PatientID
 FROM ACCESSED
```

そのデータがアクセスされたかどうかを記録しているテーブル

- Audit_Aliceで設定したsensitive dataがアクセスされたら発動するTrigger
- 誰が，何時，どんなSQLでどのデータにアクセスしたかをログに記録する

SELECT Triggerの仕組み

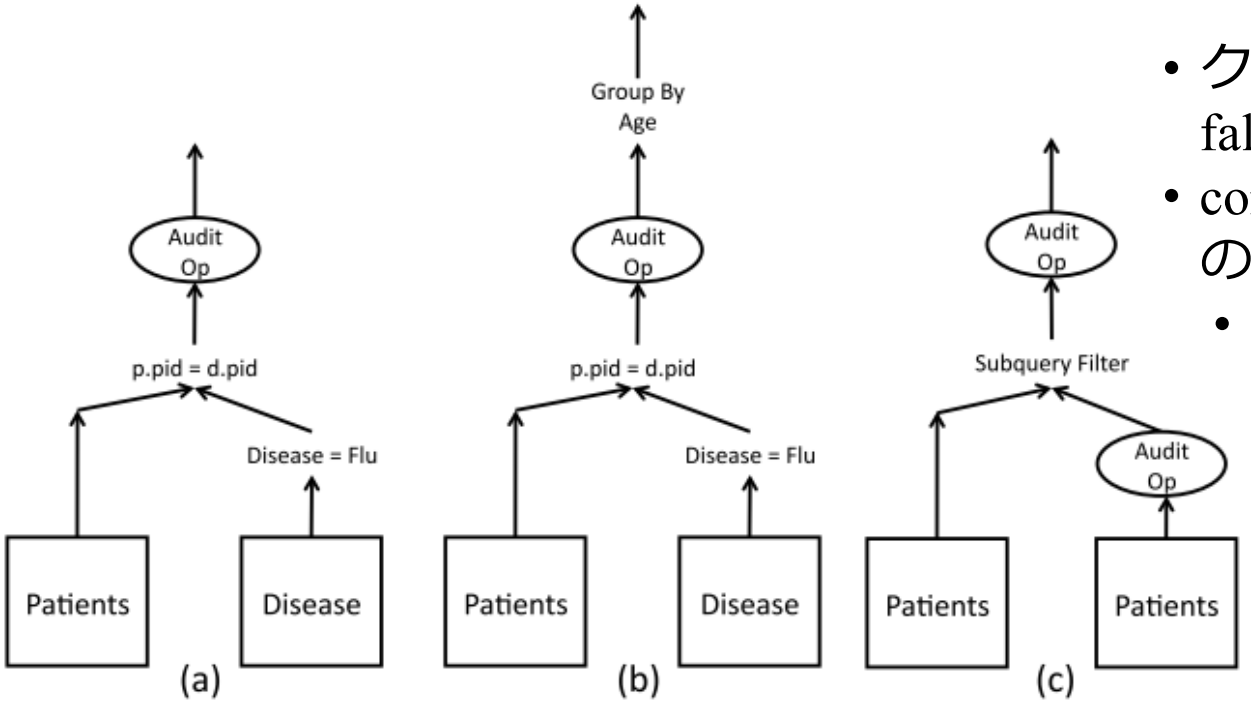
SELECT Triggerのシステム構成



- SELECT Triggerの特徴
  - 軽量なデータ監査処理
    - sensitive dataへのデータアクセス有無を迅速に判断
    - false positiveを含む
  - SQLの複雑性に関係なく正確に機能
  - false negativeは無く全てのアクセスされたsensitive dataを処理する

Audit Operator

- 監査用の演算子でクエリプラン上に配置される
  - この演算子によりsesitive dataへのアクセスがログに記録される
- 目標はfalse negativeが無くfalse positiveを最小にする
  - highest-commutative-nodeというheuristicなアルゴリズムを提案



- クエリ木の根に近い程false positiveが少なくなる
- commutativeでない演算子の上には配置しない
  - false negativeが発生してしまう

性能評価(TPC-H)

- 比較対象
  - Offline：SELECT Triggerを使わず全て静的に解析
  - Leaf-node heuristics：Audit Operatorを常にクエリ木の葉ノードの上に配置する手法

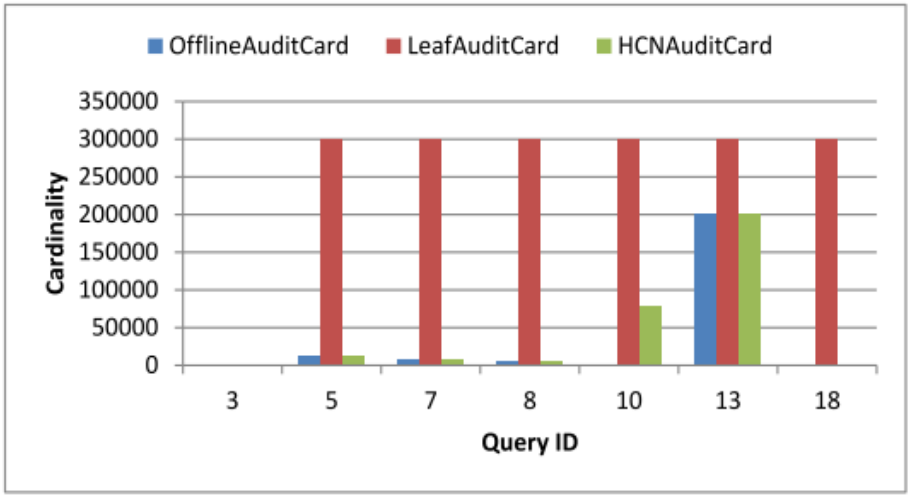


Fig. 9. Evaluating False Positives for Complex Queries

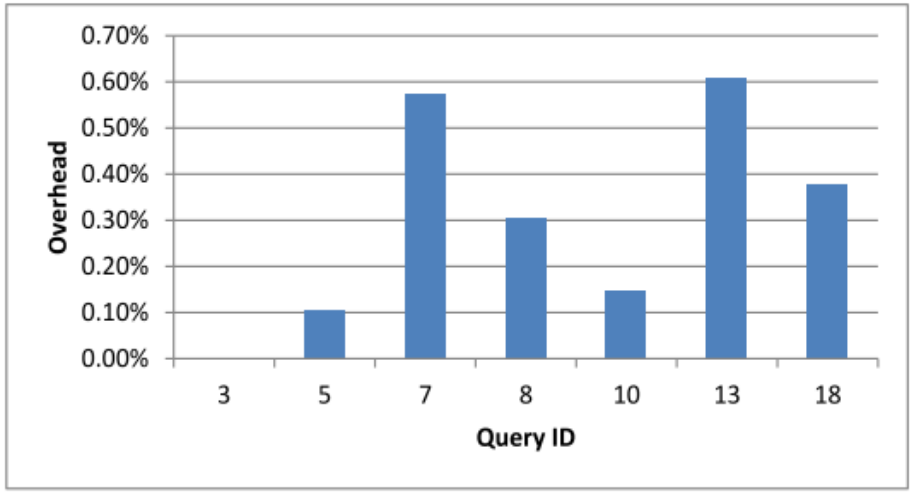


Fig. 10. **HCN** Overheads for Complex Queries

SELECT Triggerを用いない場合に対する実行時間のオーバーヘッド