creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

메인 메모리 데이터베이스의 데이터 접근빈도를 이용한 안티-캐싱 기법

Frequency Based Anti-Caching for In-Memory DBMS

2017 년 2 월

서울대학교 대학원
컴퓨터공학과
유 진 선

초 록

메인 메모리 데이터베이스중 하나인 H-Store는 데이터의 크기가 사용 가능한 메모리를 넘어서는 상황에서 안티-캐싱을 사용해 주 저장장치인 메모리에 있는 데이터를 부 저장장치인 디스크로 내려 보낸다. 안티-캐싱은 데이터의 핫-콜드를 분석하여 콜드 데이터를 디스크로 이동시켜 부 저장장치에 접근함으로써 발생하는 오랜 시간이 소요되는 트랜잭션의 수를 줄인다. 기존의 핫-콜드 분류 기법으로는 LRU, Timestamp 등이 사용되며 주로 데이터가 얼마나 최근에 접근되었는지의 정보를 이용하여 데이터의 핫-콜드를 판단한다. 이러한 방식은 과거의 정보를 기록하지 않은 채 가장 최근 정보만을 사용한다.

본 논문에서는 적은 오버헤드로 과거의 정보를 고려하는 접근빈도 방식의 안티-캐싱 기법을 제안한다. 이는 튜플마다 접근빈도를 기록, 관리하며 튜플에 접근할 경우 빈도를 증가시키는 방향으로 설계되었다. TPC-C 벤치마크를 이용해 성능을 평가하였으며 기존 안티-캐싱 기법보다 정확한 핫-콜드 분류를 통해 디스크 접근을 감소시켜 트랜잭션 처리량이 좋아짐을 확인하였다.

키워드: 메인 메모리 데이터베이스, 안티-캐싱, 접근빈도

목 차

1. 서론

1.1 연구 배경

전통적으로 데이터베이스는 디스크 기반의 시스템으로, 페이징을 통해 디스크에서 메모리로 데이터를 복사하여 데이터를 관리하거나 캐싱을 통해 자주 사용되는 데이터를 캐시에 미리 복사하여 디스크에 접근하는 것 보다 빠르게 데이터에 접근할 수 있도록 설계되었다. 그러나 최근 수십 년 동안 메모리의 가격이 계속해서 하락하고 그 성능이 증가함에 따라 보다 큰 데이터를 메모리에 저장하는 것이 가능하게 되었고, 전체 데이터를 메인 메모리에 저장하는 메인 메모리 데이터베이스 시스템 (IMDB)이 현실성 있는 시스템이 되었다. IMDB는 기존의 디스크 기반 데이터베이스 시스템과는 다르게 디스크를 거치지 않고 메모리에서 바로 데이터에 접근할 수 있기 때문에 트랜잭션 처리량과 실행 시간에 있어 훨씬 좋은 성능을 보인다.

그렇지만, 메모리의 용량이 커졌음에도 불구하고 디스크의 용량과 비교하기에는 상당히 작아 매우 큰 데이터를 관리해야 할 경우 전체 데이터를 메인 메모리에 저장하기 어려워 IMDB가 디스크 기반의 데이터베이스 시스템보다 더 좋다고 말하기는 어렵다. 데이터가 전체 메모리의 크기를 넘어서게 되면 운영체제가 가상의 메모리를 통해 데이터를 관리하게 되고 페이지 폴트 (page fault)가 발생하게 된다. 페이지 폴트는 운영체제에서 수행을 하기 때문에 사용자나 데이터베이스 시스템은 그 과정을 알기 어려우며, 트랜잭션을 수행할 때 예상치 못한 디스크 접근이 발생할 수 있으며 이는 성능 하락으로 이어진다.

이러한 문제를 해결하기 위해 안티-캐싱 (anti-caching) 이라는 새로운 메인 메모리 데이터베이스 시스템 구조가 제안되었으며 이는 자주 접근되는 "Hot" 데이터는 메모리에 저장하고, "Cold" 데이터는 메모리에서 내쫓아 디스크에 저장해 메모리에 여유 공간을 제공하는 방식이다. 전통적인 데이터베이스 시스템의 캐시는 디스크에서 시작해 핫 데이터를 메모리에 복사하는 방식인 반면, 안티-캐싱은 메모리를 주 저장장치로 하여 자주 사용되지 않는 콜드 데이터를 디스크로 내쫓기 때문에 데이터는 메모리 혹은 디스크 둘 중 하나에만 존재하게 된다. [2]

메모리 접근 속도가 디스크 접근 속도보다 빠르기 때문에 성능을 향상시키기 위해 디스크 접근을 줄이는 것이 요구되며, 데이터의 핫-콜드를 정확하게 판단하여 디스크 접근을 최소화하는 것이 중요하다. [10] 그러나 핫-콜드를 정확히 판단하기 위해 많은 계산이 필요하여 전체적인 성능 저하를 일으킬 수 있기 때문에 이 부분을 효과적으로 고려하여야 한다.

이 논문에서는 데이터의 접근 빈도 (frequency)를 사용하여 적은 오버헤드로 데이터의 핫-콜드를 판단하는 방법을 제안한다. TPC-C 벤치마크를 통해 기존의 안티-캐싱 방식인 LRU를 이용한 방식과 타임 스탬프 (Timestamp)를 이용한 방식과의 비교를 통해 기존의 방식보다 효과적임을 보인다.

1.2 논문의 구성

본 논문의 구성은 다음과 같다. 2장에서는 메인 메모리 데이터베이스와 안티-캐싱에 관한 기존 연구에 대해 설명한다. 3장에서는 안티-캐싱에 관한 기존의 핫-콜드 분류 기법과 함께 접근빈도 기반의 새로운 안티-캐싱 기법에 대해 기술하며 4장에서는 TPC-C 벤치마크를 이용한 실험 결과를 설명한다. 마지막으로 5장에서는 연구 내용을 요약하고 앞으로의 연구 과제와 함께 결론을 맺는다.

2. 관련 연구

1장에서 메인 메모리 데이터베이스 시스템과 기존의 디스크 기반 데이터베이스 시스템을 간략히 비교하였으며, 안티-캐싱의 기본 개념에 대해 설명하였다. 2장에서는 관련 연구를 통해 보다 자세히 설명하고자 한다.

2.1 메인 메모리 데이터베이스 시스템 (IMDB)

메인 메모리 데이터베이스 시스템에 대한 연구는 메인 메모리와 디스크의 접근 속도 차이를 통해 보다 효율적인 시스템을 만드는 것을 목적으로 하며 현재 H-Store, VoltDB, Hekaton 등 다양한 시스템이 사용 되고 있다. [5], [6], [7]. 그렇기 때문에 메인 메모리와 디스크의 차이점을 이해하는 것이 무엇보다 중요하다.

메인 메모리는 디스크와는 완전히 다른 특징을 가지며 그렇기 때문에 데이터베이스의 설계나 그 성능에 큰 영향을 미친다. 두 시스템의 차이를 소개하기에 앞서 메인 메모리와 디스크의 차이를 간략히 정리하면 다음과 같다. [1]

- 메인 메모리에 대한 접근시간은 디스크 저장장치에 비해 상당히 작다.
- 메인 메모리는 일반적으로 휘발성이 있으며, 디스크 저장장치는 영구적으로 저장하는 것이 가능하다.
- 일반적으로 디스크 저장장치는 블록 단위로 데이터를 저장, 관리하며, 메인 메모리는 이보다 작은 바이트 단위로 관리한다.
- 디스크 저장장치에서는 임의의 데이터에 접근하는 것 (random access) 보다 순차적인 접근 (sequential access)이 훨씬 빠르기

때문에 데이터의 배치가 상당히 중요한 반면, 메인 메모리에서는 크게 중요하지 않다.

위의 특징들은 데이터베이스 시스템을 설계하고 관리하는데 큰 영향을 미치며 크게 2가지로 나누어 설명하고자 한다.

- **큰 캐시를 가지고 있는 디스크 기반의 데이터베이스와의 차이점 [1]**

디스크 기반의 데이터베이스가 큰 캐시를 가지고 있어 전체 데이터를 카피를 캐시에 유지할 수 있는 경우에도 데이터베이스의 설계상 메모리의 장점을 다 살리기는 어렵다. 디스크 접근을 위해 B-tree와 같은 인덱스 구조를 사용하고 있는데 전체 데이터가 메모리에 있을 경우에 기존의 인덱스는 적합하지 않다. 또한 시스템은 기본적으로 데이터가 디스크에 있음을 가정하고 있기 때문에, 어떤 튜플에 접근하기 위해 디스크 주소를 계산하고 버퍼 매니저 (buffer manager)를 통해 해당하는 튜플이 메모리에 있는지 확인한 후 메모리에 있다면 해당하는 튜플을 시스템에게 돌려주게 된다. 데이터가 항상 메모리에 있을 경우 이러한 과정을 거치지 않고 메모리 주소로 바로 접근하는 것이 훨씬 효과적이다.

- **휘발성이 있는 메모리에서 데이터를 보존하는 방법**

IMDB는 휘발성이 있는 메모리를 주 저장장치로 사용하기 때문에 시스템에 문제가 발생하게 되어 시스템이 종료될 경우 기존의 데이터를 잃어버리게 된다. 이러한 문제를 해결하기 위하여 오픈소스 IMDB중 하나인 H-Store에서는 스냅샷 (snapshot, checkpointing)과 로깅 (logging)을 수행한다. [3]

스냅샷의 경우 주기적으로 커밋된 데이터베이스의 상태를 영구적으로 저장할 수 있는 디스크에 쓰는 과정이다. 데이터베이스의 성능을 유지하기 위해 비동기적 스냅샷 (Asynchronous Snapshot)을 사용하는데, 이는 스냅샷을 시작하기 전에 트랜잭션이 copy-on-write (COW)로 동작하도록 한다. 이 과정을 통해서 스냅샷이 시작되어 데이터베이스를 읽을 때 새로운 데이터를 쓰는 경우 그 데이터는 스냅샷이 끝나기 전까지 기존의 데이터를 지우는 것이 아닌 다른 공간에 저장되게 되고 스냅샷이 끝난 다음에 반영되어, 전체 시스템을 멈추지 않은 상태로 논리적으로 한 순간의 데이터베이스의 상태를 저장할 수 있다.

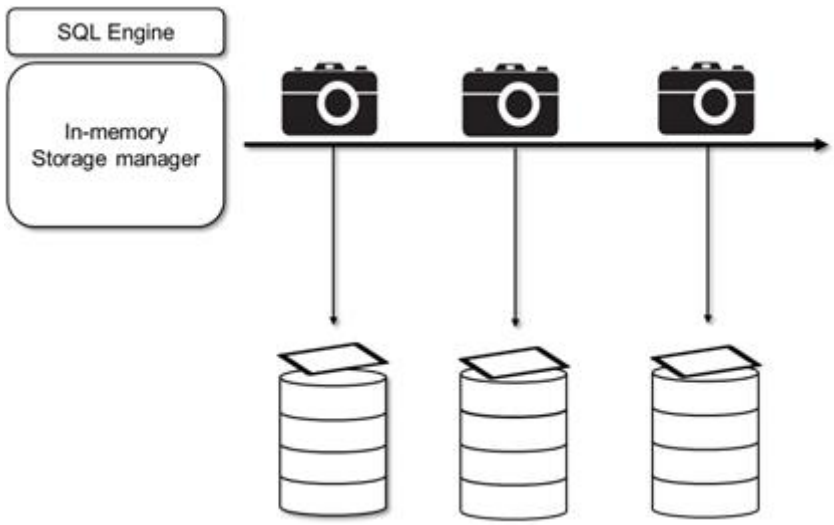

그림 1 Snapshot

로깅의 경우 IMDB의 성능을 유지하기 위하여 간단한 정보만을 담아서 기록한다. 이를 커맨드 로깅 (command logging)이라 부르고 단순히 데이터베이스에 어떠한 커맨드가 입력되었는지를 실행하기 전에 로그 저장장치에 기록한다. 이는 단순한 정보만을 기록하기 때

문에 로그를 쓰는 단계에서는 오버헤드가 거의 없으나 복구하는 단계에서 로그를 하나씩 실행하며 데이터베이스를 새로 만들어야 한다는 단점이 있다. 그러나 전통적인 로깅을 사용할 경우 오버헤드가 커 IMDB의 높은 성능 (Throughput)을 활용할 수 없어 커맨드 로깅을 사용한다. 커맨드 로깅은 데이터를 복구하는 데에 사용되며 데이터 복구는 자주 발생하는 현상이 아니므로 일반적인 경우에 높은 성능을 보이는 효율적인 방법이다.

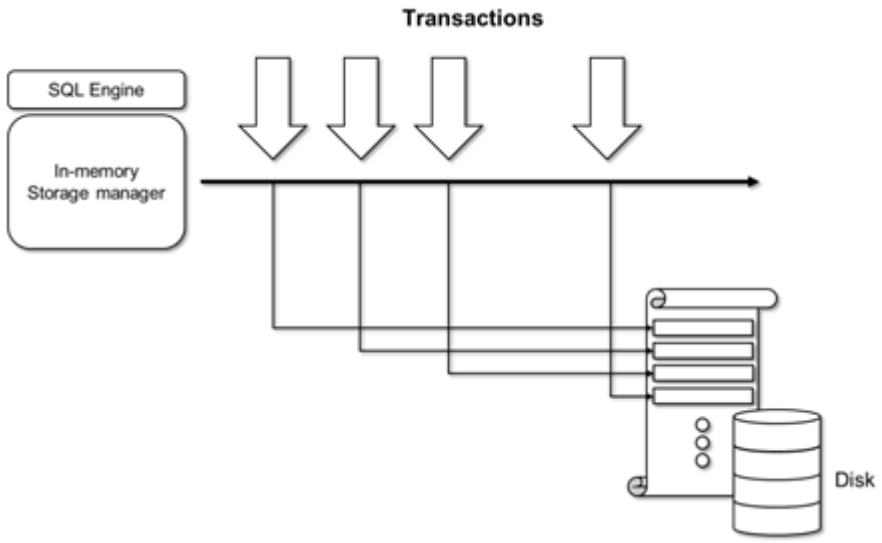

그림 2 Command Logging

위와 다르게 저장장치의 도움을 받아 이 문제를 해결하는 경우도 있다. NVM (non-volatile memory)는 메모리처럼 디스크보다 빠른 접근시간을 제공하지만 메모리와 다르게 휘발성이 없어 데이터를 영구적으로 저장할 수 있다. 현재 NVM은 메인 메모리 (DRAM) 만큼의 성능을 내지는 못하나 이에 관한 연구가 계속 진행되고 있으며 디스크 기반의 데이터베이스보다 좋은 성능을 보인다. [4]

2.2 H-Store

H-Store [5] 는 메인 메모리 데이터베이스 시스템 중 하나이며 오픈소스로 제공되는 시스템이다. 그림 3은 H-Store의 구조를 나타낸다. 하나의 H-Store 인스턴스 (Instance)는 하나 이상의 노드가 모인 클러스터의 형태로 구성되어 있으며, 각각의 노드는 물리적으로 하나의 컴퓨터를 의미하고 하나 이상의 사이트 (site) 로 구성된다. 사이트는 연산을 수행하는 기본 단위로 하나의 쓰레드를 사용하는 데몬 (single-threaded daemon) 으로 설계되어 있다. 하나의 사이트는 여러 파티션 (partition) 으로 분리 될 수 있으며 각각의 파티션은 실행 엔진 (execution engine)의 역할을 하며 하나의 쓰레드를 사용하여 트랜잭션을 수행한다. 각각의 사이트는 서로 독립되어 있으며 같은 노드에 있어도 데이터와 메모리를 공유하지 않는다. [8]

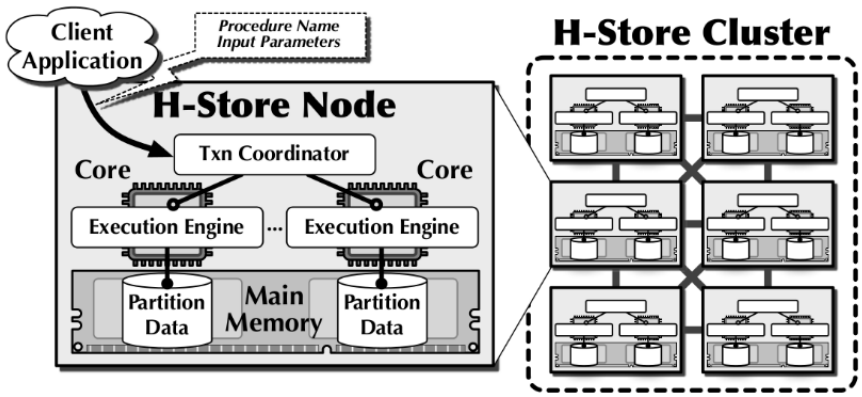

그림 3 H-Store Architecture [12]

2.3 안티-캐싱 (Anti-caching)

안티-캐싱은 H-Store에서 사용되는 메모리 관리 방식이다. 앞서 언급한 바와 같이 디스크 기반의 데이터베이스는 핫 데이터를 캐싱하는 방식으로 설계 되어 있지만, H-Store의 안티-캐싱은 콜드 데이터를 내쫓는 방식으로 설계되어 반대로 동작한다. 그림 4에서 두 시스템 모두 메모리에 핫 데이터를 가지고 있고 디스크에 콜드 데이터를 가지고 있다는 점에서 저장장치의 특징을 살리고 있지만, 설계 원리가 반대로 되어 있어 성능상의 차이를 보인다.

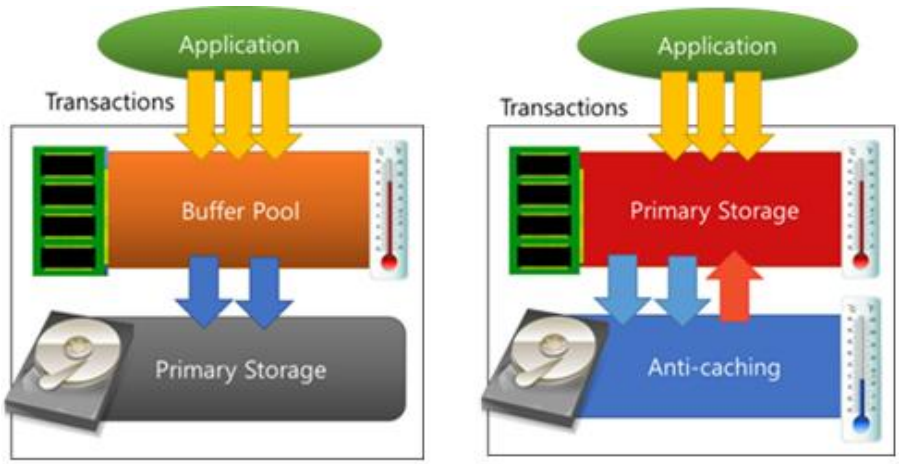

그림 4 디스크 기반의 데이터베이스 vs 메인 메모리 데이터베이스

안티-캐싱은 운영체제의 가상 메모리 스와핑 (virtual memory swapping) 과 유사하다. 가상 메모리 스와핑은 데이터의 양이 사용 가능한 메모리의 크기를 넘어서면 콜드 페이지를 디스크에 쓰게 되며, 페이지의 핫-콜드는 일반적으로 LRU (least recently used) 순서로 관리한다. 쫓겨난 페이지에 접근하게 되면 그 페이지를 메모리

로 가져오고 메모리의 크기를 넘어서는 상태이면 다른 페이지를 내쫓게 된다. 이와 비슷하게 안티-캐싱은 운영체제가 하던 가상 메모리 스와핑과 유사한 작업을 데이터베이스 시스템이 수행하도록 하는 것이다. 다만 쫓아낼 데이터와 디스크에 쫓겨나 있는 데이터에 관한 정보를 알 수 있기 때문에 크게 두 가지의 측면에서 효과적이다. [2]

- 튜플 단위로 데이터의 핫-콜드를 관리하여 쫓아낼 데이터를 튜플 단위로 선택하여 콜드 데이터만을 모아서 디스크로 내쫓는다. 반면에 페이지 단위로 내쫓는 가상 메모리 스와핑의 경우, 튜플보다 훨씬 큰 페이지 단위로 쫓아낸다. 하나의 페이지는 여러 튜플을 포함하게 되고, 다른 모든 튜플이 콜드한 경우에도 하나의 핫한 튜플이 전체 페이지를 내쫓지 못하게 할 수 있어 효과적이지 못하다.

- 페이징을 통해 쫓아낸 데이터를 읽는 경우 트랜잭션을 멈춘 채 데이터를 읽을 때 까지 기다려야 하므로 전체 시스템이 멈춘 상태로 기다리는 반면, 안티-캐싱은 해당 트랜잭션을 중지 (abort) 한 후 다른 트랜잭션을 수행하며 디스크로부터 데이터를 읽은 후 중지 시켰던 트랜잭션을 다시 수행하여 전체 시스템이 멈추지 않는다.

안티-캐싱은 크게 세 가지의 요소로 구성되어 있는데, 디스크에 여러 튜플이 모인 블록의 형태로 쫓겨난 데이터를 담고 있는 *블록 테이블 (Block Table)*과 메모리에서 쫓겨난 튜플의 위치를 블록과 매핑해주는 *Evicted Table*, 마지막으로 데이터의 핫-콜드를 판별하는 정책이 필요하다.

블록 테이블 (Block Table)

블록 테이블은 해시 테이블 (hash table)로써 각각의 블록을 식별할 수 있는 고유의 키 값을 가진다. 바이트 단위로 접근할 수 있는 메모리와는 다르기 때문에 디스크에 맞게 여러 튜플을 블록의 형태로 모아 디스크에서 관리하는데 사용한다.

Evicted Table

Evicted Table은 디스크로 쫓겨난 튜플을 관리하는데 사용한다. 튜플이 쫓겨나게 되면 튜플을 임시의 블록에 담고 메모리에서 지우며 블록이 다 차면 블록 테이블로 내쫓는다. 이때 블록의 아이디와 튜플의 아이디를 기록하여 Evicted Table에 저장하고, 접근하려는 튜플이 디스크에 있을 경우 이 정보를 이용해 디스크에서 필요한 튜플을 가져온다.

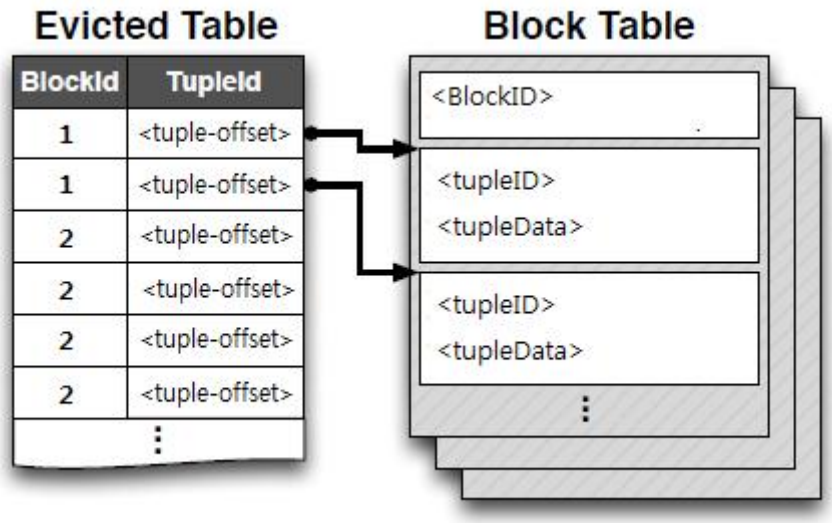

그림 5 Evicted Table과 Block Table

핫-콜드 판단 기법

디스크 접근에 오랜 시간이 걸리기 때문에 전체 시스템의 성능을 향상시키기 위해서 데이터의 핫-콜드를 적은 오버헤드로 정확히 판별하는 것이 중요하고, 이를 위해 핫-콜드 분류 정책이 많이 연구되었다.

2.3.1 LRU (least recently used)

LRU는 최근에 사용되지 않은 페이지를 찾는 버퍼 관리 정책이다 [9]. 이를 H-Store에서는 양쪽방향을 참조할 수 있는 연결 리스트(double-linked list)로 구현하였으며 튜플 단위로 이전 튜플과 다음 튜플을 가리키는 포인터를 넣어 LRU Chain의 형태로 표현한다. 어떤 튜플에 접근하거나 수정되는 경우에는 LRU Chain에서 해당하는 튜플을 찾아서 제거한 후 Chain의 가장 뒤 (tail)에 새로 삽입하게 된다. 새로운 튜플이 삽입되는 경우에도 마찬가지로 LRU Chain의 가장 뒤에 새로운 튜플을 삽입하게 된다. 튜플을 내쫓아야 할 경우 LRU Chain의 가장 앞 (head)에서부터 일정량의 튜플을 선택하여 블록에 넣고 해당하는 튜플을 Chain에서 제거한다. [2]

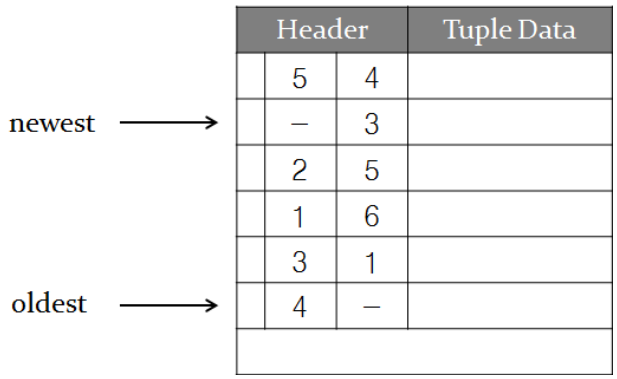

그림 6 LRU Chain 예시

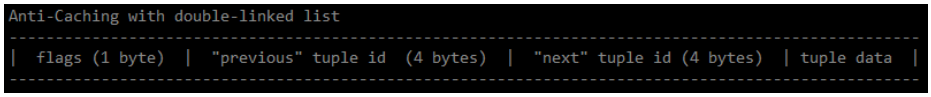

그림 7 H-Store의 LRU Chain

매번 튜플에 접근할 때 마다 LRU Chain을 갱신하는 경우 CPU 오버헤드가 상당히 크다. 그렇기 때문에 몇 개의 샘플을 선택하여 LRU Chain을 갱신한다. 핫 데이터는 접근될 확률이 높기 때문에 샘플을 선택할 때 포함될 확률이 높다는 것을 전제로 하는 approximate LRU (aLRU) 방식을 제안한다. 기존 연구에서는 0.01의 sample rate를 통해 LRU Chain에 튜플을 삽입하고, 해당되는 튜플만을 관리하여 핫-콜드 분류의 정확성이 하락하나 Chain을 갱신하는 속도를 높여 전체적인 성능 향상을 이끌어냈다.

2.3.2 타임 스탬프 (Timestamp)

기본적인 핫-콜드 판단 방식인 LRU Chain은 연결 리스트로 설계되어 리스트를 갱신하는 오버헤드가 크기 때문에 오버헤드를 줄이기 위해 새로운 방식인 타임스탬프 방식의 핫-콜드 분류 방식이 제안되었다. 이 방식은 튜플에 접근하거나 새로운 튜플이 만들어질 때 그 튜플에 현재의 타임스탬프 값을 부여하여 최근에 접근된 튜플일수록 높은 타임스탬프 값을 지니도록 하여 핫-콜드를 판단한다.

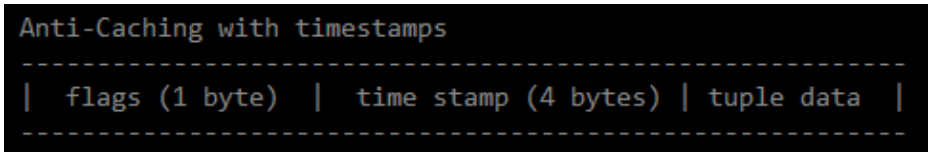

그림 8 Anti-caching with Timestamp

메모리가 넘쳐 튜플을 내쫓아야 할 경우 LRU Chain과는 다르게 따로 내쫓을 튜플을 관리하고 있지 않아 내쫓을 양의 3~4배의 튜플을 읽어 그 중 상대적으로 오래된 튜플을 선택하여 내쫓는다. 내쫓아야 하는 튜플이 많을수록 데이터를 더 많이 읽어야 하며 이는 성능이 저하의 원인이 된다.

3. 접근 빈도 기반의 안티-캐싱 기법

본 논문에서는 안티-캐싱을 위한 튜플 접근 빈도 기반의 핫-콜드 분류 기법을 제안한다. 기존 핫-콜드 분류 정책들과 마찬가지로 적은 오버헤드로 핫-콜드를 판단할 수 있으며 기존의 정책은 얼마나 최근에 접근되었는지를 우선적으로 고려하는 반면, 본 기법은 접근 빈도를 통해 데이터의 핫-콜드를 보다 정확히 판단하고자 한다.

안티-캐싱은 데이터의 핫-콜드를 구분할 수 있는 상황에서 사용되며 데이터의 핫-콜드를 판단할 수 없이 고르게 접근되는 경우에는 디스크로 쫓겨난 데이터에 계속해서 접근하게 되어 성능이 하락하여 큰 효과를 보기 어렵다. 또한, 데이터의 대부분이 메모리에 있는 경우 상당히 적은 양의 데이터를 내쫓게 되고 핫-콜드 분류에 관계없이 디스크 접근이 적어 핫-콜드 분류의 중요성이 떨어진다. 따라서 본 논문에서는 데이터의 핫-콜드를 구분할 수 있으며 데이터의 양이 사용 가능한 메모리를 넘어 계속 증가하는 상황을 기본 전제로 하여 본 안티-캐싱 기법을 설계하였다.

튜플 구조

접근 빈도 기반의 안티-캐싱을 위한 데이터는 그림 9와 같다. 타임스탬프 방식과 유사하게 4 바이트의 추가 메모리를 사용하며 튜플 단위로 안티-캐싱을 수행하기 때문에 튜플마다 접근 빈도를 부여한다.

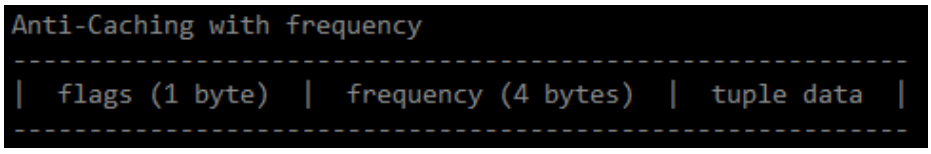

그림 9 Anti-Caching with frequency

튜플에 접근될 때 접근 빈도 (frequency)를 1씩 증가시키며 데이터의 핫-콜드를 관리한다. 기존의 LRU나 타임스탬프 기반의 핫-콜드 분류 기법의 경우 데이터를 관리하는 방법은 다르지만 콜드 데이터에 접근하는 경우 그 데이터를 가장 핫한 데이터로 인식되게 되고 이전의 핫-콜드 분류 정보는 버려지게 되어, 가장 최신의 정보만을 기록하는 방식이다. 그러나 접근 빈도를 이용하면 과거의 정보를 별다른 오버헤드 없이 반영할 수 있으며 콜드 데이터가 한번 접근되어도 그대로 콜드 상태를 유지한다. 즉, 핫-콜드 정보를 누적해서 관리하는 역할을 하고, 순간적인 핫-콜드가 아닌 지금까지의 정보를 모아 핫-콜드를 판단하여 내쫓을 데이터를 선택하므로 핫 데이터가 쫓겨나갈 확률이 작다.

접근 빈도 기반의 안티-캐싱 기법

이미 기술한 것처럼 안티-캐싱은 온라인으로 트랜잭션을 처리하면서 새로 데이터가 삽입되거나 기존의 데이터를 읽고 수정하는 과정에서 데이터의 핫-콜드를 판단하는 방식이며, 핫-콜드 분류 정책은 아래 두 가지를 고려하여 설계되었다.

- 안티-캐싱을 위해 추가로 사용하는 메모리를 최소화 한다.
- 트랜잭션 처리 속도의 하락 없이 핫-콜드를 분류해야 한다.

첫째로, IMDB는 메인 메모리를 주 저장장치로 사용하며 안티-캐싱은 메모리에 공간이 부족할 때 디스크 접근을 최소화 하기위해 자주 사용되지 않는 콜드 데이터를 디스크로 내려 보내는 방식이다. 즉, 안티-캐싱을 위해 메모리를 추가로 사용하면 사용 가능한 메모리의 양이 줄어들게 되어 더 많은 양의 데이터를 디스크로 내쫓아야 하며, 이는 더 많은 디스크 접근으로 이어지고 성능 하락으로 나

타난다. 따라서 안티-캐싱을 위해서 추가로 사용하는 메모리를 최소화 하는 것은 당연하다.

둘째로, 디스크 기반의 데이터베이스보다 빠른 트랜잭션 처리를 위해 IMDB를 설계하였기 때문에 트랜잭션 처리 성능을 저하시키면서 안티-캐싱을 수행하는 것은 큰 의미가 없다. 또한, 안티-캐싱을 위해 데이터를 관리하는 오버헤드가 안티-캐싱을 사용하지 않는 상태에 비해 무시할 수 없는 정도의 성능 저하를 일으킨다면 성능 향상을 위해 설계된 안티-캐싱이 오히려 전체적인 시스템의 성능 저하를 일으킬 수 있기 때문에 트랜잭션 처리 속도의 하락 없이 핫-콜드를 분류하는 것이 중요하다.

LRU 정책의 경우 LRU Chain을 갱신하는 오버헤드가 전체적인 성능저하를 일으켜 이를 줄이기 위해 sampling rate를 두어 일정 확률로 Chain을 갱신하는 approximate LRU로 개선하였으며,

타임스탬프 정책의 경우 적은 양의 메모리를 사용하기 위해 튜플마다 4 바이트의 메모리 만을 추가로 할당하며 튜플에 접근될 때 타임스탬프를 갱신하여 가장 최근에 접근된 시간을 기록한다. 또한 핫-콜드 분류를 위해 데이터베이스를 풀 스캔 (full scan) 하지 않고 일부분만을 읽어 성능 하락을 막는 방향으로 설계하였다.

접근 빈도 기반의 안티-캐싱 역시 위의 설계 원리를 따라 설계하였다. 접근 빈도를 저장할 4 바이트의 추가 공간을 튜플마다 할당하였으며, 타임스탬프 방식과 유사하게 핫-콜드 분류를 위해 데이터베이스의 일부분만을 읽는 방식을 택하였다. 그러나 이전의 정책은 가장 최근의 정보만을 저장하는 반면, 접근 빈도 방식은 접근 빈도에 현재까지 튜플에 접근되었던 정보를 저장하여 과거 데이터 (historical data)를 반영한 핫-콜드 분류를 가능하게 한다.

과거 데이터를 반영함으로써 새로 삽입되는 튜플은 상대적으로 낮은 접근 빈도를 가지게 되어 심지어 핫 데이터인 경우에도 콜드 데이터로 인식되어 디스크로 쫓겨날 수 있다. 이러한 문제를 막기 위해 새로 삽입되는 데이터에 최대 빈도 값 (maximum frequency)의 α (0 ≤ α ≤ 1) 배 만큼의 값을 주어 새로 삽입된 튜플이 바로 쫓겨나는 것을 방지하였다.

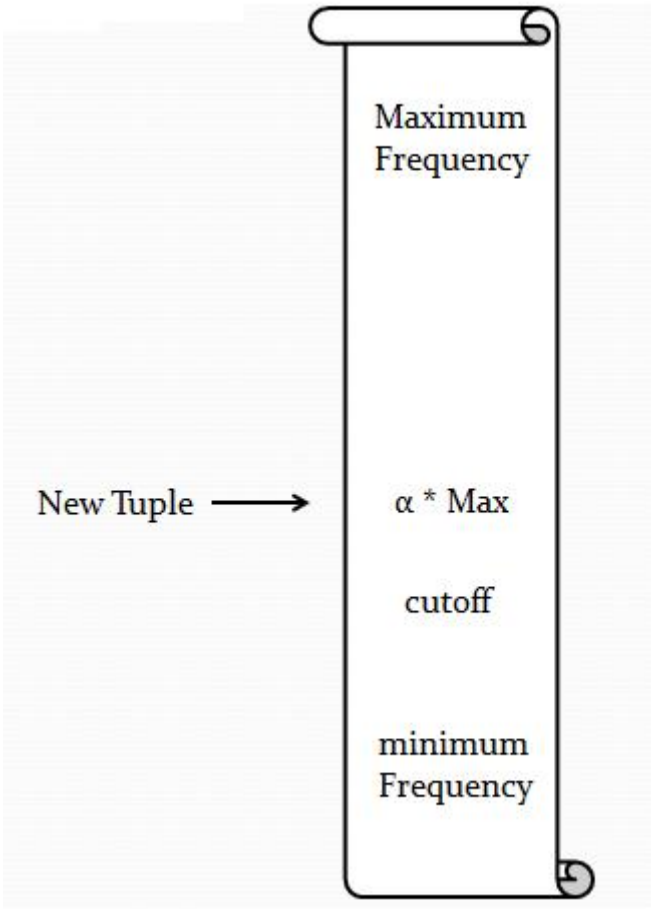

그림 10 접근 빈도 방식의 새로운 튜플 삽입

새로 삽입되는 튜플에 cutoff frequency 이상의 빈도 값을 할당하여 새로 삽입된 튜플이 바로 쫓겨나는 것을 방지하였다. 핫 데이터의 경우 자주 접근되어 빈도 값이 증가하게 되고, cutoff frequency에서 멀어져 쫓겨날 확률이 낮아지지만 콜드 데이터의 경우 트랜잭션이 수행됨에 따라 cutoff frequency의 증가로 빈도 값이 cutoff frequency보다 낮아지게 되어 쫓겨나게 된다.

4. 실험 및 결과

실험에서는 접근 빈도 기반의 안티-캐싱의 성능을 평가하기 위해 기존의 안티-캐싱 기법과 추가로 기본 (Baseline) 정책을 하나 선택해 성능을 비교하고자 하며 각각의 정책은 다음과 같다.

- sLRU
  - single-linked list를 사용하는 LRU Chain 방식
- rLRU
  - double-linked list를 사용하는 LRU Chain 방식
- aLRU
  - double-linked list를 사용하며 sampling rate를 통해 LRU Chain에서 관리할 튜플을 선택하는 방식
- Timestamp
  - 타임스탬프를 이용하여 튜플을 관리하는 방식
- Baseline
  - 핫-콜드 구별 없이 임의로 튜플을 선택하여 내쫓는 방식
- Frequency
  - 접근 빈도를 이용하여 튜플을 관리하는 방식

접근 빈도 기반의 안티-캐싱을 오픈소스 IMDB인 H-Store에 구현하였으며 TPC-C [11] 벤치마크를 이용해 위의 여섯 가지 핫-콜드 분류 기법의 초당 트랜잭션 처리량과 내쫓은 블록에 접근하는 횟수를 분석하여 접근 빈도 기반의 안티-캐싱의 성능을 평가하고자 한다.

4.0 실험 환경

실험에서 사용한 데이터베이스와 하드웨어는 다음과 같으며, 사용 가능한 메모리의 크기 (Threshold), 쫓아내는 데이터의 크기 (Evict size), 쫓아내는 데이터의 주기 (Interval), 전체 실험시간 (Duration) 등을 바꾸어가며 실험하였으며, 정밀한 실험을 위해 메모리가 큰 머신에서 추가로 실험을 진행하였다.

[Machine 1]
- Database : H-Store (Final Release 2016. 6)
- Partition : 4 partitions (use 1 thread per partition)
- H-Store Site Memory : 2,048 MB
- H-Store Client Memory : 512 MB
- Memory : 24 GB
- CPU : Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz 4 cores
- Cold Storage : samsung SSD 850 evo

[Machine 2]
- H-Store Site Memory : 4,096 MB
- H-Store Client Memory : 1,024 MB
- Memory : 64 GB
- CPU : Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz 6 cores
- Cold Storage : samsung SSD 850 pro

대부분의 실험은 1번 환경으로 진행되었으며, 보다 정확히 성능을 평가하기 위해 2번 환경에서 추가로 접근 빈도 방식과 타임스탬프 방식의 성능을 비교하였다.

4.1 TPC-C Benchmark

H-Store의 성능을 평가하기 위해 널리 사용되는 OLTP (Online Transaction Processing) 벤치마크중 하나인 TPC-C 벤치마크를 선택하여 실험을 진행하였다. TPC-C는 새로운 주문을 입력하고 주문 상품을 배달하고 계산하며 주문 상태를 확인하고 창고의 재고를 확인하는 총 5개의 트랜잭션으로 이루어져 있다. [11] TPC-C의 스키마는 그림 11에 표현되어 있으며 각각의 화살표는 그 도착점에 있는 테이블의 크기가 출발점에 있는 테이블의 크기의 N배 혹은 그 이상 (+)이 됨을 나타낸다. TPC-C 벤치마크는 이미 H-Store에 구현되어 있으며 [5], 그 내용을 표 1에 간략히 정리하였다.

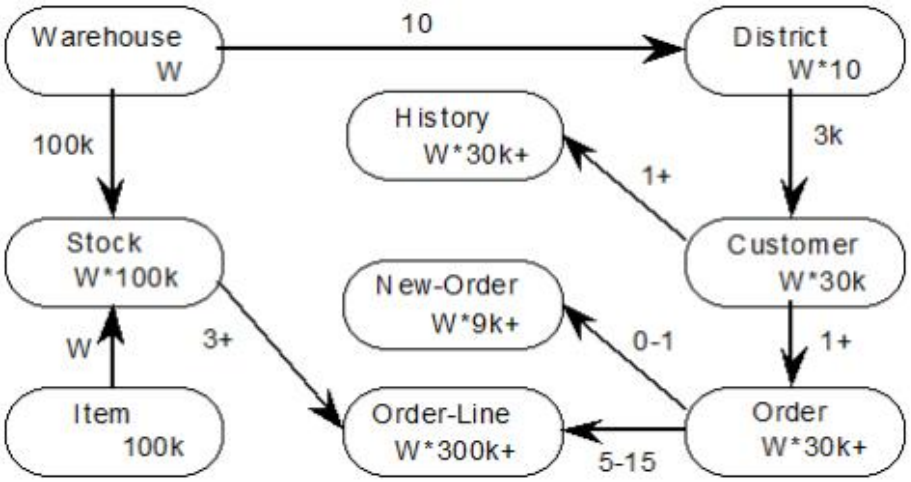

그림 11 TPC-C 스키마

트랜잭션	작업 (Workload)
New-Order	고객의 주문을 처리하는 트랜잭션으로 90%의 주문이 고객의 창고 (warehouse)에서 이루어진다. 읽기/쓰기 (Read/Write)를 모두 수행한다.
Order-Status	고객의 마지막 주문에 대한 쿼리이다. 읽기만을 수행하는 트랜잭션이다. (Read-only)
Payment	고객의 잔고 (balance)와 창고 (warehouse)를 갱신하며 읽기/쓰기 (Read/Write)를 모두 수행한다.
Delivery	10개 정도의 새로운 주문을 만들어 고객의 잔고와 New-Order 테이블을 갱신한다.
Stock-Level	물품을 선택하며 많은 읽기 작업을 수행한다. (Read-heavy)

표 1 TPC-C Transactions

전체 9개의 테이블 중 디스크로 Evict할 테이블을 4개 지정하여 실험을 수행하였다. Orders, Order-Line, History, New-Order가 그 대상이며, 이 테이블들은 H-Store의 안티-캐싱에서 기본 값으로 설정되어 있으며 변경할 수 있다.

실험에 사용한 TPC-C Workload는 일반적인 TPC-C Workload와 같으며 다음과 같다.

트랜잭션	작업 (Workload)
New-Order	45%
Order-Status	4%
Payment	43%
Delivery	4%
Stock-Level	4%

표 2 TPC-C Workload

4.2 실험 및 결과 분석

4.1절에 언급한 TPC-C Workload로 실험을 수행하였으며 앞의 6가지 정책의 성능을 비교한다. 본 실험은 각각의 정책마다 Machine 1에서 10분씩 TPC-C 벤치마크를 수행한 결과이며 최소 3회 이상의 실험을 통해 나온 평균치로 결과를 나타낸다.

실험 1) LRU 정책의 성능비교

본 실험을 진행하기에 앞서 여러 가지 LRU 정책 (sLRU, rLRU, aLRU)의 성능을 평가하여 가장 효과적인 LRU를 선택하여 본 실험을 진행하는데 사용하였다.

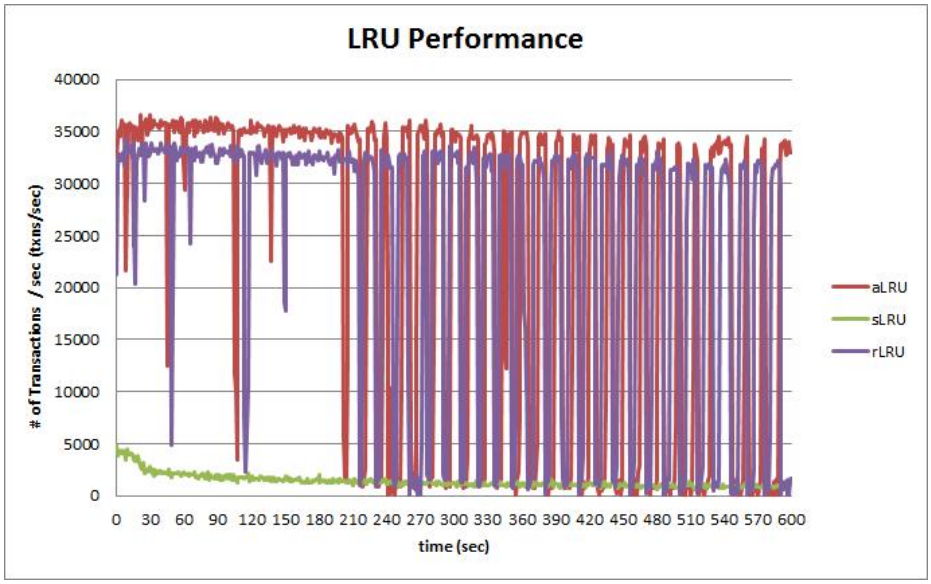

그림 12 LRU 초당 트랜잭션 처리량 (TPS)

최대 메모리 사용량 (Threshold)는 3,000MB로 제한하였으며 Evict가 발생할 때 약 160개의 블록을 내쫓도록 하였고, 블록 하나의 크기는 1MB이다.

그림 12는 LRU 정책들의 초당 트랜잭션 처리량을 나타낸 것이다. sLRU는 다른 LRU에 비해 트랜잭션 처리량이 현저히 나쁜데 이는 다른 방식과 다르게 하나의 포인터만을 사용해 LRU Chain을 관리하기 때문에 트랜잭션이 튜플에 접근하여 LRU Chain을 갱신해야 할 경우 LRU Chain의 가장 뒤 (tail)로 옮겨야 하는데 sLRU의 구현상 LRU Chain을 전부 읽어야 해서 성능이 하락하게 된다. 실제로 모든 데이터가 메모리에 있어 디스크에 접근하지 않음에도 불구하고 낮은 트랜잭션 처리량을 보인다.

aLRU는 rLRU보다 평상시의 TPS (초당 트랜잭션 처리량, txns/sec) 가 6% 정도 높으며 평균 TPS도 rLRU에 비해 전체적으로 4% 정도 높다. 이는 aLRU가 rLRU에 비해 LRU Chain을 관리하는데 드는 오버헤드가 적어 더 많은 트랜잭션을 수행할 수 있기 때문이다. 표 3은 LRU 기법들의 최대 TPS와 평균 TPS를 나타낸다.

LRU Policy	maximum TPS	avg TPS (txns/sec)
sLRU	4,909 txns/sec	1,400 txns/sec
rLRU	34,312 txns/sec	21,614 txns/sec
aLRU	36,568 txns/sec	22,412 txns/sec

표 3 LRU 기법 트랜잭션 처리량

또한, LRU 정책의 디스크 접근을 비교하기 위해 내쫓은 블록(Evicted Block)과 다시 메모리로 가지고 온 블록(Fetched Block)의 수를 기록하였다. 각각의 기법마다 TPS가 다르고 더 적은 양의 트랜잭션을 수행했을 경우 일반적으로 내쫓은 블록의 수가 적으며, 메

모리에 콜드 데이터가 많아 콜드 데이터를 내쫓을 확률이 높아 다시 메모리로 가지고 올 확률이 낮다. 공평한 실험을 진행하기 위하여 특정 시점에서의 값을 측정하는 것이 요구되며 본 실험에서는 쫓아낸 데이터의 양에 따라 다시 디스크로 가져온 블록의 양을 측정하였다. 이는 표 4에 나타나 있으며 안티-캐싱의 주기인 15초 마다 해당 값을 받아오고 쫓아낸 데이터의 양에 따라 정리하였다. 각각은 블록의 수를 의미하며, 하나의 블록은 여러 튜플로 이루어져 있으며 크기는 1MB이다.

rLRU			aLRU		
Evicted	Fetched	ratio	Evicted	Fetched	ratio
495	264	0.35	467	293	0.39
1043	480	0.32	955	566	0.37
1523	608	0.29	1503	788	0.34
2034	709	0.26	1979	926	0.32

표 4 LRU 기법 디스크 접근

전체 내쫓았던 데이터 (Evicted + Fetched)에 비해 다시 메모리로 가지고 온 데이터 (Fetched)의 양을 측정한 결과 모든 경우에서 rLRU가 aLRU에 비해 성능이 좋으며 평균적으로 rLRU는 0.3의 값을 가져 0.35의 값을 가지는 aLRU보다 디스크 접근이 적어 성능 향상으로 나타난다. 최대 TPS가 6%의 차이를 보이는 반면 평균 TPS는 그보다 낮은 4%의 차이를 보이는 것으로 보아 디스크 접근 횟수의 감소가 성능 향상으로 이어짐을 알 수 있다. 그러나 평균 TPS의 측면에서 aLRU가 더 좋은 성능을 보이므로 안티-캐싱 실험에서 사용할 LRU 기법은 aLRU로 선택하였다.

실험 2) 안티-캐싱 기법 성능비교

본 실험에서는 LRU (aLRU), Baseline, Timestamp, Frequency 총 4개의 안티-캐싱 기법에 대해 성능을 평가하고자 한다. Baseline 기법은 Timestamp와 Frequency의 기본이 되는 정책으로 두 방식 모두 전체 데이터베이스를 다 읽어야 하는 오버헤드를 피하기 위하여 쫓아낼 데이터의 3~4배의 튜플을 읽고 그중에서 상대적으로 콜드 튜플을 내쫓는 방식으로 구현되어 있는데, 이때 핫-콜드의 구분 없이 임의로 쫓아낼 데이터만큼의 튜플을 읽어 내쫓는 방식이다.

실험 2-1)

[Machine 1]
Threshold : 2,000MB
Evict Size : 200MB
Interval : 15 sec
Duration : 600 sec

위의 조건으로 실험하였으며 그 결과는 그림 13, 14, 15과 같다.

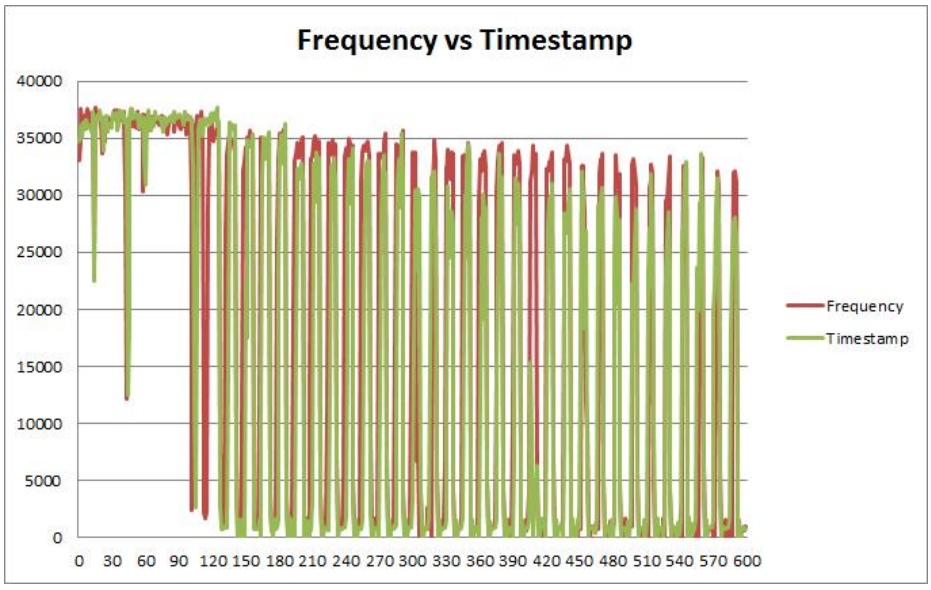

그림 13 Frequency vs Timestamp TPS

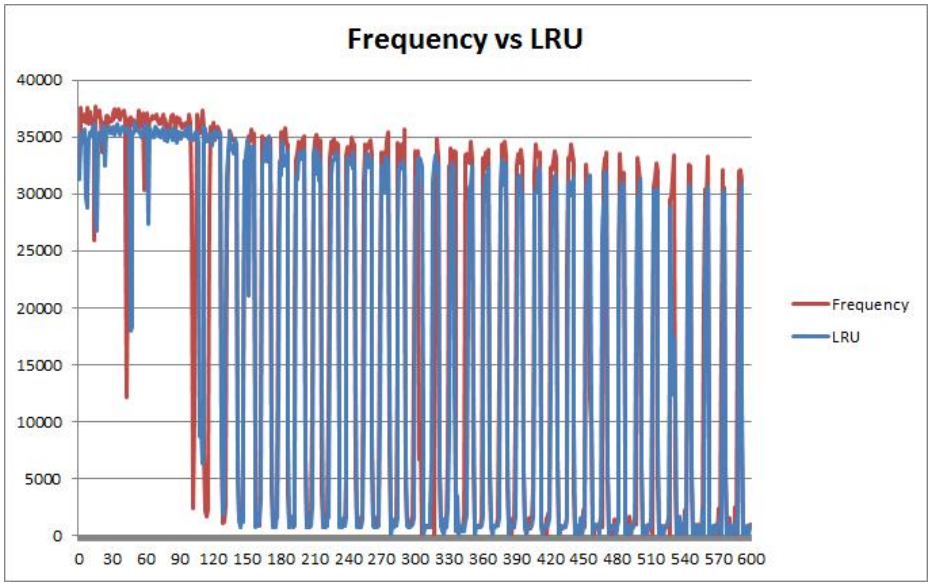

그림 14 Frequency vs LRU TPS

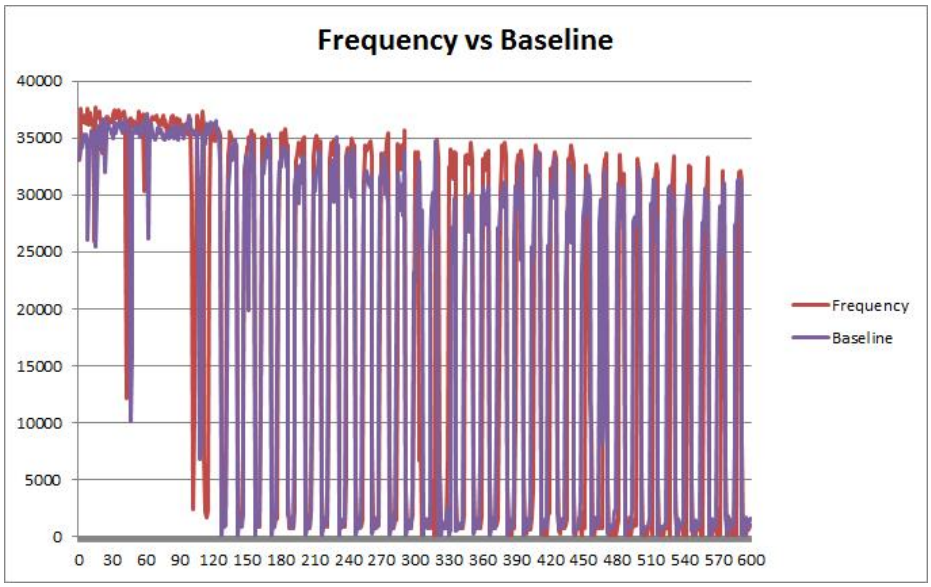

그림 15 Frequency vs Baseline TPS

실험 결과를 쉽게 확인할 수 있도록 하기위해 3개의 그래프로 나누어 표현하였다. TPS에서 큰 차이를 보이지는 않지만 접근 빈도 방식이 다른 방식에 비해 대부분 좋은 성능을 가짐을 표 5에서 확인할 수 있다. Timestamp 방식에 비해 성능이 좋은 것으로 보아 이는 보다 나은 핫-콜드 분류 방법의 차이에 의한 결과라 할 수 있다.

표 6의 디스크 접근 분석에서 접근 빈도 기반의 안티-캐싱은 시간이 지날수록 핫-콜드 분류가 정확해 지는 반면 Timestamp와 Baseline의 경우 점차 성능이 하락함을 확인할 수 있다. Timestamp 기반의 안티-캐싱의 경우 과거의 정보를 고려하지 않고 가장 최근의 정보만을 저장하기 때문에 사용 가능한 메모리의 양이 작아짐에 따라 핫-콜드 분류가 부정확해짐을 확인할 수 있었다.

Anti-Caching	maximum TPS	avg TPS (txns/sec)
Frequency	37,739 txns/sec	18,636 txns/sec
Timestamp	37,852 txns/sec	17,978 txns/sec
Baseline	38,002 txns/sec	19,838 txns/sec
LRU	36,296 txns/sec	17,256 txns/sec

표 5 안티-캐싱 기법 트랜잭션 처리량

Frequency			Timestamp			Baseline		
Evict	Fetch	ratio	Evict	Fetch	ratio	Evict	Fetch	ratio
262	699	0.73	458	504	0.52	89	832	0.90
496	1236	0.71	655	1076	0.62	128	1538	0.92
824	1679	0.67	868	1630	0.65	175	2242	0.93
1316	2351	0.64	1203	2445	0.67	253	3292	0.93
1814	3011	0.62	1484	3323	0.69	332	4346	0.93
2088	3510	0.63	1660	3919	0.70	388	5239	0.93

표 6 디스크 접근 분석

Baseline 기법이 핫-콜드 분류를 하지 않고 튜플을 내쫓음에도 불구하고 가장 좋은 성능을 보이는데, 이는 Baseline 기법이 튜플을 내쫓기 위해 데이터를 모으는 과정에서의 오버헤드를 감소시키기 때문이다. 뿐만 아니라 가장 큰 원인은 내쫓은 데이터의 거의 대부분을 다시 메모리로 가지고 오기 때문에 메모리 제한을(Threshold) 만족하지 않기 때문이다. 본 실험에서 메모리 제한을 2,000MB로 설정하였으나 4,000MB의 데이터를 메모리에 지니는 상태가 되어 대부분의 데이터가 메모리에 존재하게 되고 이는 성능 향상으로 이어져 가장 좋은 성능을 보이게 된다.

반면에 디스크로 가장 많은 블록을 내보내 메모리에 가장 적은 양의 데이터를 가지고 있는 접근 빈도 기반의 안티-캐싱이 Baseline을 제외하고 가장 좋은 성능을 낸 것으로 보아 핫-콜드 분류가 중요함을 확인할 수 있었다.

실험 2-2)

[Machine 2]
Threshold : 2,000MB
Evict Size : 200MB
Interval : 15 sec
Duration : 900 sec

실험 2-1에서 시간이 지남에 따라 데이터베이스의 크기가 커져 디스크로 쫓겨난 데이터의 양이 많아지고, 그로인해 디스크 접근이 많아지게 되어 TPS가 하락하는 것을 확인할 수 있었다. 더 많은 코어와 메모리를 가지는 보다 나은 환경의 머신에서 추가 실험을 진행하였다. 900초 동안 실험을 진행하였으며 그 결과는 그림 16과 표 7에 나타나 있다.

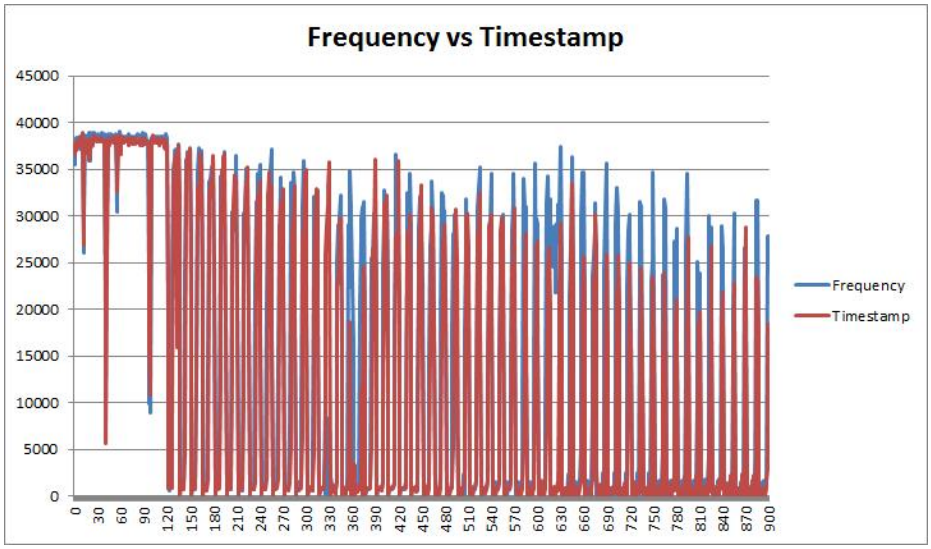

그림 16 Frequency vs Timestamp TPS

Anti-Caching	maximum TPS	avg TPS (txns/sec)
Frequency	39,019 txns/sec	14,873 txns/sec
Timestamp	39,012 txns/sec	13,789 txns/sec

표 7 Frequency vs Timestamp TPS

전체적인 경향은 실험 2-1과 유사하며 시간이 지남에 따라 두 정책 모두 성능이 하락하였다. 이는 앞서 언급한 바와 같이 디스크에 쫓겨난 데이터의 양이 많아져 디스크 접근이 증가하기 때문임을 알 수 있다.

표 8에서 디스크 접근에 대해 분석하였으며 시간이 지남에 따라 두 방법 모두 ratio가 증가하는 경향을 보임을 확인할 수 있었으나 접근-빈도 방식이 타임스탬프 방식보다 더 낮은 ratio를 가지는 것을 확인할 수 있었다. 이는 핫-콜드 분류를 통해 디스크 접근을 감소시켰다는 것을 의미한다.

데이터의 크기가 2,000MB보다 작은 처음 120초 동안 두 안티-캐싱 기법은 데이터를 디스크로 내쫓지 않고 메모리상에서 핫-콜드를 분류하는 과정을 진행한다. 이 과정에서 접근-빈도 방식은 과거 데이터를 반영하는 반면, 타임스탬프 방식은 가장 최신의 정보만을 저장하기 때문에 준비과정이 성능에 미치는 영향을 보기 위하여 추가 실험을 진행하였다.

Frequency			Timestamp		
Evicted	Fetched	ratio	Evicted	Fetched	ratio
296	282	0.49	225	351	0.60
472	815	0.63	365	985	0.73
722	1590	0.69	608	1710	0.74
956	2327	0.71	866	2422	0.74
1206	3052	0.72	1106	3159	0.74
1591	3835	0.71	1402	4037	0.74
1758	4834	0.73	1671	4941	0.75
2164	5403	0.71	1933	5654	0.75
2363	5787	0.71	2047	6120	0.75
2607	6710	0.72	2211	7114	0.76

표 8 Frequency vs Timestamp 디스크 접근 분석

실험 2-3)

[Machine 2]
Threshold : 3,000MB
Evict Size : 200MB
Interval : 15 sec
Duration : 900 sec

실험 2-2에서 언급한 바와 같이 데이터를 내쫓기 위해 준비하는 과정이 전체 안티-캐싱에 미치는 영향을 보기 위하여 메모리 제한을 3,000MB로 늘려 실험을 진행하였다. 그러나 핫-콜드 분류의 영향뿐만 아니라, 사용가능한 메모리의 양이 증가함에 따라 상대적으로 적은 양의 데이터를 내쫓게 되고, 디스크 접근이 줄어들게 되어 전체적으로 성능이 향상될 것으로 생각된다. 자세한 실험 결과는 그림 17과 표 9, 표 10에 나타나 있다.

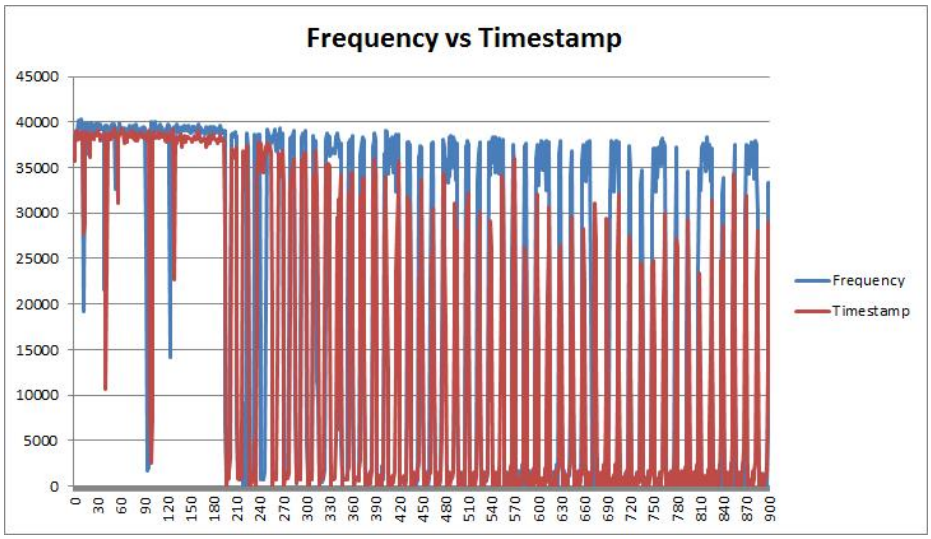

그림 17 Frequency vs Timestamp TPS

Anti-Caching	maximum TPS	avg TPS (txns/sec)
Frequency	40,339 txns/sec	20,551 txns/sec
Timestamp	39,407 txns/sec	18,772 txns/sec

표 9 Frequency vs Timestamp TPS

본 실험에서는 약 200초 정도의 순수한 핫-콜드 분류기간을 가지며 이는 과거 정보를 반영하는 접근 빈도 방식에게 더 나은 핫-콜드 분류를 가능하게 한다. 실제로 그림 16의 그래프와 비교하였을 때 성능 하락 폭이 감소하여 전체적으로 성능이 향상된 것을 확인할 수 있으며, 특히 접근-빈도 방식의 경우 디스크로 내쫓는 순간을 제외하고는 대부분 35,000 txns/sec 이상을 유지하여 성능이 거의 하락하지 않았다.

Frequency			Timestamp		
Evicted	Fetched	ratio	Evicted	Fetched	ratio
458	117	0.20	431	145	0.25
895	254	0.22	770	381	0.33
1433	486	0.25	1163	752	0.39
1802	696	0.28	1508	989	0.40
2157	917	0.30	1821	1256	0.41
2569	1081	0.30	2079	1578	0.43
3005	1412	0.32	2432	1996	0.45
3464	1722	0.33	2746	2454	0.47
3910	2052	0.34	3091	2877	0.48
4311	2421	0.36	3499	3242	0.48

표 10 Frequency vs Timestamp 디스크 접근 분석

표 10에서 두 안티-캐싱 기법의 디스크 접근에 대해 분석하였다. 두 방법 모두 시간이 지남에 따라 디스크 접근 횟수가 증가하게 되고 쫓아낸 데이터를 메모리로 다시 가져오는 경우 (ratio)가 증가하였다. 그러나 실험 2-2 에서의 결과인 표 8과 비교하였을 때 둘 모두 상당히 작은 값을 가짐을 확인할 수 있었으며 이는 사용가능한

메모리의 양의 증가가 전체적으로 나은 핫-콜드 분류를 가능하게 하였기 때문이다.

안티-캐싱의 성능을 보기 위해 디스크로 내쫓는 순간부터 그래프를 세 부분으로 나누어 각 부분의 트랜잭션 처리량을 측정하였으며 이는 그림 18, 19, 20에 나타나 있다.

첫 번째 구간의 경우 그림 18에 나타나 있으며 평균 TPS는 접근 빈도 방식이 20,740로 16,455를 기록한 타임스탬프 방식에 비해 26% 정도의 성능 향상이 있었으며 디스크로 쫓겨난 데이터는 400MB정도의 차이를 보였다. 이는 디스크에서 다시 메모리로 가져온 데이터의 양이 더 작다는 것을 의미하며 트랜잭션 처리량이 더 많음에도 불구하고 위와 같은 차이를 보였다.

두 번째 구간의 경우 그림 19에 나타나 있으며 평균 TPS는 각각 16,432와 7,741로 약 110% 정도의 성능 향상이 있었으며 디스크로 쫓겨난 데이터는 약 600MB의 차이를 보였다. 이는 접근-빈도 방식이 과거로부터 정보를 누적하여 저장하기 때문에 시간이 지날수록 보다 정확한 핫-콜드 분류가 가능할 것이라 예상한 것과 일치하는 결과이다.

세 번째 구간 역시 평균 TPS는 각각 13,559와 6,160으로 약 120% 정도의 성능 향상이 있었으며 쫓겨난 데이터 역시 800MB 정도의 차이를 보여 접근 빈도 방식이 보다 정확한 핫-콜드 분류를 통해 전체적인 성능을 향상시킬 수 있다는 것을 확인하였다.

데이터 크기 (MB)	1/3	2/3	3/3
Frequency	1,943	3,228	4,311
Timestamp	1,585	2,646	3,499

표 11 디스크로 쫓겨난 데이터베이스의 크기 (MB)

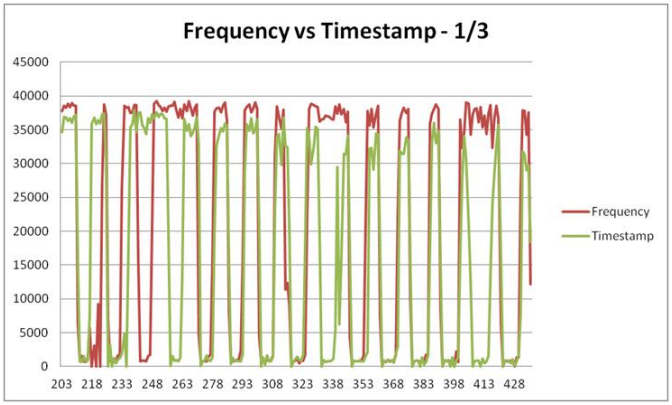

그림 18 Frequency vs Timestamp (1)

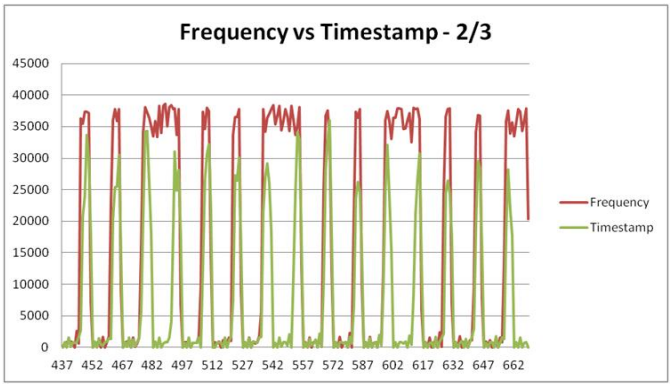

그림 19 Frequency vs Timestamp (2)

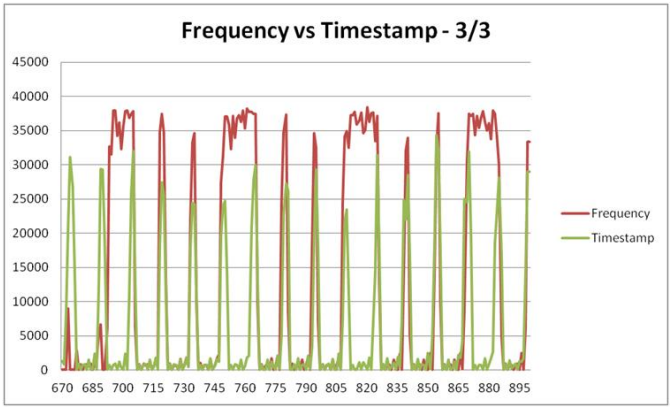

그림 20 Frequency vs Timestamp (3)

5. 결론 및 고찰

본 논문에서는 안티-캐싱의 효율적인 핫-콜드 분류를 위하여 접근 빈도 기반의 안티-캐싱을 제안하였다. 이는 데이터를 적은 오버헤드로 관리한다는 점에서 기존의 방법과 유사하지만, 과거의 데이터를 누적하여 핫-콜드 분류에 사용하기 때문에 시간이 지날수록 보다 정확히 데이터의 핫-콜드를 구분할 수 있다는 점에서 기존의 방식과 다르다. 또한 새로 삽입되는 데이터에 초기 값을 부여하여 새로 삽입된 데이터가 콜드 데이터로 분류되어 쫓겨나는 것을 막는 방식으로 설계하였다. 접근 빈도 기반의 안티-캐싱의 성능을 평가한 결과 기존의 방식에 비해 디스크 접근이 감소하고 트랜잭션 처리량이 증가한 것으로 보아 안티-캐싱에서 데이터의 핫-콜드 분류가 중요하다는 것을 알 수 있다.

향후 과제로는 다음과 같은 내용의 연구가 필요하다. 본 연구에서는 접근빈도 기반의 안티-캐싱 기법을 제안하였는데 이는 데이터에 접근할 경우 접근 빈도를 1씩 증가시킨다. 그렇기 때문에 한번 콜드로 판단된 데이터가 자주 접근되는 경우 이를 핫 데이터로 인식하기는 어렵다. Workload가 변경되어 오랫동안 콜드로 판단된 데이터가 최근에 자주 접근되는 경우 타임스탬프나 LRU 방식은 쉽게 핫 데이터로 인식할 수 있지만, 접근빈도 기반의 안티-캐싱은 계속해서 콜드 데이터로 인식하여 성능하락의 원인이 된다. 따라서 이러한 변화에 적응할 수 있도록 접근 빈도 값을 상황에 맞게 (adaptive) 갱신하는 방법에 대한 연구가 필요하다.

참고문헌

[1] Hector Garcia-Molina, and Kenneth Salem. Main memory database systems: an overview. In IEEE, vol 4, no. 6, Dec. 1992.

[2] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. In VLDB, 6(14):1942-1953, Sep. 2013.

[3] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In ICDE, 2014.

[4] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. R. Dulloor. A prolegomenon on oltp database systems for non-volatile memory. In ADMS, 2014.

[5] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, The end of an Architectural Era: (It's Time for a Complete Rewrite), In VLDB 1150-1160, 2007.

[6] M. Stonebraker, and A. Weisberg. The VoltDB Main Memory DBMS, In IEEE. 2013.

[7] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. Proceedings of the VLDB Endowment, 7(11):931-942, 2014.

[8] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, H-Store: a high-Performance, distributed main memory transaction processing system, Proc. VLDB Endow., vol. 1, iss. 2, 1496-1499, 2008

[9] T. Johnson, and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In VLDB. 1994.

[10] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, Reducing the storage overhead of main-memory OLTP databases with hybrid indexes, In Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 2016.

[11] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[12] H-Store. http://hstore.cs.brown.edu.

Abstract

H-Store, one of the main memory database management system, uses anti-caching to evict data from primary storage (main memory) to secondary storage (disk) where the size of the data exceeds memory threshold. Anti-caching uses hot-cold separate policy to manage data, and moves cold data to disk in order to reduce number of long-latency transactions caused by accessing to secondary storage. Current methods (e.g. LRU, Timestamp) use "recency" (how recently data is accessed) information to separate data. These methods do not keep past information.

In this paper, I present low overhead, frequency-based anti-caching method which keeps the past information. This method adds frequency value to each tuple, and increases the value when corresponding tuple is accessed. Performance was evaluated using the TPC-C benchmark. Through accurate hot-cold separation, overall transaction throughput is increased by reducing secondary storage access.

Keywords: Main Memory Database System, IMDB, Anti-caching, Frequency