하드디스크와 플래시SSD상에서 열-지향 저장 모델 고찰1)2)

박지영, 강운학, 이상원
성균관대학교 전자전기컴퓨터공학과
e-mail:bjy8027@skku.edu

A Study of Column-oriented Storage Method on Harddisks and Flash SSDs

Ji-Young Park, Woon-Hak Kang, Sang-Won Lee
Department of electrical and Computer Engineering, SungKyunKwan University

요 약

열-지향 데이터베이스 시스템인 C-Store는 많은 상용 데이터베이스 시스템과는 달리 데이터를 행(row) 위주가 아닌 열(column) 위주로 저장을 하여, 데이터 웨어하우스와 같이 주로 읽기 IO를 유발하는 환경에서 데이터의 전송량을 줄임으로써, 높은 성능을 보였다. 본 논문에서는 대표적인 열 지향 저장 DBMS인 C-Store와 행 위주의 저장구조를 사용하는 기존 DBMS와의 차이점을 알아보고, C-Store의 저장장치로 하드디스크와 차세대 저장장치로 주목받고 있는 플래시 SSD(Solid State Disk)를 사용하였을 때, 발생할 수 있는 장단점에 대해 분석하였다.

1. 서론

대부분의 상용 DBMS들은 행-지향(row-oriented) 데이터베이스 시스템이다. 이러한 행-지향 DBMS들은 행 저장소(row store)를 사용하는데, 이 행 저장소는 레코드들의 속성(attribute)들을 연속해서 저장한다. 따라서 레코드 단위로 데이터 업데이트(update)가 발생하는 OLTP(OnLine Transaction Processing)환경에서 관계형 DBMS의 저장소로 주로 사용되어 왔다.

정보기기가 다양해지고, 데이터가 크게 증가하면서 데이터를 통해 정보를 얻고자하는 OLAP(OnLine Analytical Processing)환경에서의 데이터베이스 시스템을 사용하는 빈도가 잦아졌다. OLAP환경에서는 데이터를 분석하고자, 읽기 위주의 비정형적인 쿼리(ad-hoc query)가 주로 발생한다. 주로 대용량 데이터를 다루는 OLAP환경에서는 데이터의 전송속도가 전체 성능을 좌우한다. 일부 열만을 접근하여 결과를 얻을 수도 있는데 행-지향 DBMS에서는 전체 레코드를 메모리로 전송하게 되면서 불필요한 데이터까지 함께 전송하여 디스크IO 병목현상(bottleneck)이 발생하였다. 이러한 문제를 해결하고자, MonetDB[4]와 C-Store[1] 열-지향 DBMS들이 제안되어 왔다.

MonetDB는 데이터를 엔트리(entry) 순서대로 열 저장소(column store)를 사용하여 저장하며, 모든 열은 정렬키를 사용하여 정렬된 순서로 저장된다. 그런데 이러한 열 저장소에서 업데이트가 발생하면 각 열을 모두 갱신해야 하므로, 많은 IO를 발생하게 된다. 이러한 문제를 해결하기 위해, C-sotre가 제안되었다.

C-store는 쓰기 가능한 열 저장소(Writeable Store)와 읽기 최적화된 열 저장소(Read-optimized Store)로 분리되어 데이터가 저장된다. 쓰기 가능한 열 저장소에는 데이터의 삽입과 갱신이 용이하도록 원본 데이터를 변환 없이 저장하여 높은 쓰기 속도를 보장하고, 읽기 최적화된 열 저장소는 많은 양의 정보를 저장할 수 있도록 데이터 변환과정을 거쳐 저장된다.

플래시 SSD(Solid State Drive)는 하드 디스크에 비해 기계적인 구성 요소가 없기 때문에 데이터 접근 속도가 매우 빠른 장점을 가지고 있다. SSD는 연속적인 데이터의 읽기/쓰기와 임의 읽기(random read) 성능이 뛰어나지만, 랜덤 쓰기(random write)가 느리기 때문에 OLAP환경과 같이 쓰기보다는 읽기가 주를 이루는 환경에 매우 적합하다. 본 논문에서는 C-store가 향후 저장장치를 SSD로 변경하였을 때, 가질 수 있는 장단점을 하드디스크와 비교하여 분석하였다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 열-지향 저장소와 행-지향 저장소를 비교하고, 3장과 4장은 C-Store의 저장장치로 각각 하드 디스크와 SSD를 사용하

1) 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No. 2011-0026492)
2) 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2011-0027316)

는 경우 어떠한 장단점을 가질 수 있는지 분석하였다. 마지막으로 5장에서는 결론으로 논문을 맺는다.

2. 열-지향 저장소와 행-지향 저장소

기존의 행-지향 DBMS에 대해서는 많이 소개되었기에 본 논문에서는 언급을 자제하고, 열-지향 DBMS인 C-Store에 대한 구조를 알아본다.

2.1 C-Store 데이터 모델(Data Model)

C-Store는 논리적으로 관계형 데이터 모델을 지원 하지만, 실제 저장된 데이터들은 논리적 데이터 모델을 사용하지 않는다. 행 저장소를 사용하는 대부분의 DBMS들은 데이터를 직접적으로 테이블로 나타내고 다양한 인덱스들을 사용하여 성능을 높이는 반면에, C-Store는 오직 프로젝션(projection)들로만 구성된다.

Name	Age	Dept	Salary
Mike	29	Accounting	330
Jon	31	Marketing	320
Cindy	27	Administration	300

그림 1 EMP 예제 테이블

프로젝션은 테이블에 있는 열을 하나 또는 그 이상을 포함하고 있고, 한 열이 여러 프로젝션에 나타날 수 있다. 논리적인 테이블을 그대로 물리적으로 저장하지 않고, 여러 프로젝션들로 저장을 한다. 그림 2와 같이 각 프로젝션은 정렬키(sort key)를 사용하여, 정렬된 순서로 저장되어 있다.

```
EMP1 (name, age | age)
EMP2 (dept, salary, DEPT.location | DEPT.location)
EMP3 (name, salary | salary)
DEPT1(dname, location | location)
```

그림 2 정렬키를 포함한 프로젝션

모든 프로젝션들은 하나 또는 그 이상의 세그먼트로 수평 분할(horizontally partition)되는데, 이때 각 세그먼트에는 세그먼트를 식별할 수 있는 세그먼트 아이디(SID)가 주어진다. 이렇듯 C-Store는 하나의 논리적인 레코드가 분리되어 저장되어 있으므로 하나의 레코드로 만들기 위해서는 여러 프로젝션을 조인(join)하여 나타내야 한다. 조인을 하기 위해서 조인 인덱스(join index)와 스토리지 키(storage key)를 두어 나타낸다.

스토리지 키는 다른 프로젝션 세그먼트에 저장된 같은 논리적인 레코드의 위치를 나타내는 키이다. 이 키는 읽기 최적화된 열 저장소에는 저장되지 않고, 쓰기 가능한 열 저장소에만 저장이 된다.

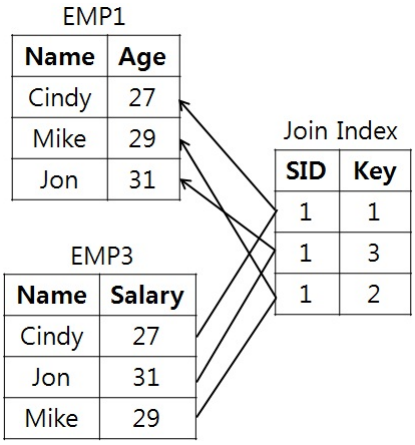

그림 3 조인 인덱스 예

그림 3과 같이 조인 인덱스는 같은 테이블 프로젝션들의 조인 인덱스로 1 대 1 맵핑된다. 조인 인덱스는 세그먼트 아이디와, 스토리지 키의 쌍으로 표현이 되며, 이것을 통해 하나의 레코드로 만들 수 있다. 저장된 조인 인덱스는 출발하는 프로젝션과 같은 방법으로 수평 분할을 하고, 관련된 세그먼트 끼리 연속해서 할당한다.

2.2 읽기 최적화된 저장소(RS)

읽기 최적화된 저장소에서는 데이터를 압축하여 나타낸다. 압축을 통하여 열을 중복 저장해도 효과적 저장할 수 있다. 4가지 압축 방법이 있는데 프로젝션의 정렬키와 값의 분포에 따라 선택하여 압축을 한다.

2.2.1 압축 기법

1) 정렬키 열이 자신의 테이블이며(주키-primary key), 값의 분포도가 작은 경우 (값, 열안의 위치, 반복 횟수)로 나타낸다. 예를 들어, 값이 4이고 12번째 위치에서 18번째까지 값이 나타난다면 (4, 12, 7)로 표현한다.

2) 정렬키 열이 다른 테이블에 위치하며(외래키-foreign key), 값의 분포도가 작은 경우 값과 비트맵으로 표현을 한다. 예로 들어, 열의 값이 0,0,1,1,2,1,0,2,1 이라면, (0, 110000100), (1, 001101001), (2, 000010010)로 표현한다.

3) 정렬키 열이 자신의 테이블이며, 값의 분포도가 큰 경우 처음 값은 그대로 표현을 하고, 다음 값부터는 이전 값으로부터의 델타값으로 표현을 한다. 예로 들어 값이 1,4,7,7,8,12 로 나타난다면 1,3,3,0,1,4로 압축하여 나타낸다.

4) 정렬키 열이 다른 테이블에 위치하며, 값의 분포도가 큰 경우 데이터를 압축하지 않고 그대로 표현을 한다.

2.3 쓰기 가능한 열 저장소(WS)

쓰기 가능한 열 저장소 역시 읽기 최적화된 열 저장소와 같은 형태를 가지고 있지만, 효과적으로 데이터를 갱신하여야 하므로 다르게 표현된다. 데이터를 표현하는데 있어서 압축을 하지 않고 나타내고, 스토리지 키를 물리적으로 저장하여 나타낸다. 쓰기 가능한 열 저장소에서는 값과 스토리지 키의 쌍으로 나타낸다.

2.4 스토리지 관리(Storage Management)

그리드 시스템(grid system)에서 노드에게 세그먼트를 할당할 때 스토리지 관리가 발생한다. C-Store에서는 스토리지 할당자(storage allocator)를 사용하여 자동적으로 할당한다. 이것은 프로젝션의 세그먼트 안에 있는 모든 열들을 연속해서 저장을 한다. 또한 초기에 할당한 후 균형이 깨지게 되면 재할당 한다.

2.5 삽입과 삭제

삽입을 실시하면 쓰기 가능한 열 저장소의 한 프로젝션에 하나의 열 값과 스토리지 키가 저장된다. 삭제를 실시하면 저장소 데이터의 잠금(lock)이 필요한데, 일반적으로 잠금을 사용하면 잠금 경쟁(lock contention)이 발생하여 성능저하가 발생할 수 있다. C-Store는 이러한 문제를 스냅샷 분리(snapshot isolation)를 통하여 데이터의 읽기만 유발하는 트랜잭션(only-read transaction)은 빠른 성능을 제공해준다.

분산 시스템 안에서 스냅샷 분리를 지원하기 위해, 하이 워터 마크(High Water Mark)와 로우 워터 마크(Low Water Mark)를 이용한다. 하이 워터 마크는 여러 노드에서 동시에 트랜잭션이 동작하므로, 하이 워터 마크 이전까지 완료(commit) 된 트랜잭션이라는 것을 보장해주기 위해 사용되며, 로우 워터 마크는 튜플 무버(tuple mover)가 쓰기 저장소에서 읽기 저장소로 이동해야하는 데이터들을 알려주기 위해 사용된다. 이러한 스냅샷 분리를 지원함으로써, 갱신 작업은 삽입 후 삭제하는 작업으로 대체된다.

스냅샷 분리를 지원하기 위해, 각 레코드들은 시간표(timestamp)를 관리한다. 쓰기 저장소에는 삽입이 일어난 레코드들의 시간을 가지고 있는 삽입 벡터(Insertion Vector)를 관리하며, 각 프로젝션에는 삭제 레코드 벡터(Deleted Record Vector)를 저장한다. 삭제 레코드 벡터 각 항목의 초기 값은 0이고, 이 레코드가 삭제가 되면 해당 시점의 시간표를 저장하게 된다.

2.6 튜플 무버(Tuple Mover)

쓰기 가능한 열 저장소에 저장된 데이터들을 읽기 최적화된 열 저장소로 옮기는 작업이 필요하게 되는데, 이 작업을 튜풀 무버(tuple mover)가 수행한다. 옮기는 작업 이외에도 조인 인덱스를 갱신하고, 백그라운드에서는 머지-아웃 프로세스(Merge-Out Process)라는 작업을 수행한다. 머지-아웃 프로세스는 쓰기 가능한 열 저장소 세그먼트에 저장된 데이터들 중에서 로우 워터 마크 이전 또는 그 때 삽입된 모든 레코드들을 찾아, 로우 워터 마크 이전 또는 그 때 삭제된 레코드는 읽기 최적화된 열 저장소로 내려 보내지 않고 버리고, 삭제되지 않은 레코드와 로우 워터 마크 이후에 삭제된 레코드는 읽기 최적화된 열 저장소로 내려 보내는 작업을 한다.

이렇게 분리를 하고나면 머지-아웃 프로세스는 읽기 최적화된 열 저장소에 새로운 세그먼트를 생성하고, 이전에 저장되어 있던 데이터 중 삭제된 레코드 벡터에 저장된 값이 로우 워터 마크보다 작거나 같은 데이터들은 삭제한다. 그 후 쓰기 가능한 저장소에서 내려온 데이터와 기존에 있었던 데이터를 프로젝션별 정렬키 순으로 병합한다. 병합된 데이터들은 머지-아웃 프로세스를 통해 새로 생성된 세그먼트에 쓰여 진다.

3. 하드 디스크에서의 C-Store 장단점

2장에서 언급하였듯이, C-Store는 열 저장소를 사용한다. 이렇듯 특정 열의 값을 읽고자 하는 쿼리가 주어졌을 때, 메모리로 무관한 열의 값이 올라오는 것을 피할 수 있어 디스크I/O를 줄일 수 있다. 이러한 가장 큰 특징으로 데이터웨어하우스와 같은 쓰기보다 읽기가 주인 시스템에서는 효과적인 성능을 보이는 것을 볼 수 있다.

디스크 대역폭이 증가고 있는 것보다 CPU는 훨씬 빠르다. 이렇듯 하드디스크에서 많은 데이터를 한 번에 읽어 들이는 경우 병목현상(bottleneck)이 발생하게 된다. 이러한 현상을 줄이고자 C-Store에서는 CPU를 사용해서 디스크 대역폭을 절약하는 두 가지 방법을 사용한다.

첫 번째, 데이터를 압축하여 표현을 한다. 데이터를 압축을 함으로서 많은 데이터들을 작은 용량으로 저장할 수 있으며, 중복 저장을 하더라도 용량이 크게 증가하지 않는다. 또한 많은 데이터를 읽어 올 때, 압축하지 않은 곳에서의 디스크I/O 비용에 비해, 압축하여 많은 데이터를 한 번에 읽어 들일 수 있어 디스크I/O 비용을 상당히 줄일 수 있는 장점을 가진다. 압축 외에도 현재 관계형 DBMS에서는 데이터들을 열별로 구별하기 위하여 바이트(byte) 또는 워드(word) 경계선을 두고 데이터를 직접적으로 표현을 한다. 하지만 C-Store에서는 동일한 열의 값들을 연속해서 저장하기 때문에 경계선을 둘 필요가 없게 된다. 그러므로 행 저장소에 비해 더 많은 데이터를 저장할 수 있게 된다.

두 번째로는 값을 밀집 배치하여 저장을 한다. 이것은 관련 있는 데이터들을 스토리지 할당자(storage allocator)가 서로 연속해서 저장함으로서, 관련 있는 데이터들까지 읽어 들이는데 빠른 성능을 보일 수 있다. 이렇듯 읽기에 빠른 성능을 주기 위해 밀집 배치를 하게 되는데, 어느 순간 밀집 배치할 수 없는 상황에 이르게 되는 경우에는 밀집 배치되지 않은 데이터에 대해서는 빠른 성능을 보장할 수 없게 되는 단점이 있다. 또한 이 밀집 배치의 균형이 깨지는 경우 재할당 하여야하는 작업이 추가적으로 필요하게 된다.

튜플 무버가 쓰기 가능한 저장소에서 읽기 최적화된 저장소로 데이터를 내려 보내는데 있어서, 새로 할당된 세그먼트로 이전 읽기 최적화된 저장소의 내용을 다시 저장하는 작업을 수행하는데, 이것은 읽기 최적화된 저장소에 새로운 데이터를 기존 데이터와 병합하여 정렬키 순으로 순차적으로 읽기 위한 작업 때문에 수행한다. 하지만 C-Store같은 경우에는 중복 저장을 수행하므로 이러한 오

버헤드가 많이 발생하게 된다.

4. 플래시 SSD에서의 C-Store 장단점

SSD는 플래시 메모리(Flash Memory)를 통해 전자적인 방식으로 사용하기 때문에 접근속도 측면에서 많은 이점이 있다. 쓰기보다는 읽기가 훨씬 빠르며 순차접근 처리뿐만 아니라, 임의 읽기 성능 또한 하드 디스크 보다 좋다.

이러한 특징에 더불어 C-Store는 데이터를 압축하여 표현하고 열 저장소를 사용하기 때문에 하드 디스크를 사용할 때 보다 좋은 성능을 낼 것이다.

플래시 SSD에서 랜덤 쓰기(random write)가 아닌 이상 쓰기 역시 좋은 성능을 나타낸다. C-Store는 많은 데이터를 순차적으로 저장하기 때문에 쓰기 성능 또한 하드 디스크보다 더 빠를 것이다. 하지만 C-Store는 데이터를 밀집 배치하고, 그 균형이 깨지는 경우에는 다시 재배치하는 작업을 수행하는 특징이 있어, 데이터를 자주 옮기게 되므로, 플래시 SSD의 성능과 수명에 좋지 않다. 왜냐하면, 하드 디스크에서는 읽기와 쓰기 속도가 동일하기 때문에 데이터를 자주 이동 시키는 것이 큰 문제가 되지 않지만, 플래시 SSD에서는 읽기와 쓰기의 명령의 접근 속도가 비대칭적이기 때문에, 성능저하가 발생한다. 또, 플래시메모리 SSD는 제한적인 수명을 지녔기 때문에, 계속적으로 데이터를 옮겨 쓰는 것은 수명저하를 유발하게 된다. 따라서 읽기 저장소로 SSD를 사용하는 경우, 재배치하기 보다는 임의 읽기가 발생하더라고 재배치를 하지 않는 것이 성능과 수명측면에서 더 효율적일 것이다.

5. 결론

본 논문에서는 대표적인 열-지향 데이터베이스 시스템으로 C-Store를 살펴보고, 하드 디스크와 플래시 SSD에서의 장단점을 살펴보았다.

C-Store는 빠른 읽기 성능을 지원하며, 쓰기에도 효과적으로 지원하는 것을 알 수 있었다. C-Store는 하드 디스크를 고려하고 구현되어, 플래시 SSD를 사용하지 않아도 좋은 성능을 보일 것으로 예상된다. 하지만 C-Store의 특징인 밀집배치를 하지 않는다면, 플래시 SSD에서는 밀집배치에 대한 오버헤드 없이도 충분히 읽기에 빠른 성능을 보일 것이다.

밀집 배치에 대한 오버헤드를 두 저장장치에서 모두 나타나는 것을 알 수 있었다. 하지만 하드디스크의 경우에는 임의 읽기가 연속한 데이터를 읽는 속도보다 현저하게 떨어지므로 밀집배치를 수행하는 것이 데이터를 읽을 때 성능상 유리하다.

그러나 플래시 SSD에서는 임의 읽기가 순차 읽기에 비해 크게 느리지 않고, 밀집배치로 인해 발생하는 SSD 성능하락과 수명저하를 막기 위해서는 밀집배치를 사용하지 않고 임의 읽기를 수행하는 편이 좋을 것으로 예상한다.

참고문헌

[1] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik, "C-Store: A Column-oriented DBMS" Proc. VLDB, 2005

[2] Daniel J. Abadi and Peter A. Boncz and Stavros Harizopoulos, "Column-oriented Database Systems", Proc. VLDB, 2009

[3] Alan Halverson, Jennifer L. Beckmann, Jeffrey F. Naughton, David J. DeWitt, "A Comparison of C-Store and Row-Store in a Common Framework", Proc. VLDB, 2006

[4] Peter Boncz, Marcin Zukowski, Niels Nes, "MonetDB/X100: Hyper-Pipelining Query Execution", Proc. CIDR, 2005

[5] 배영현, "고성능 플래시 메모리 SSD(Solid State Disk) 설계 기술", Proc. 한국정보과학회, 2007