Paper Review
MyRocks

컴퓨터소프트웨어학부
김현정 박정호

논문 – Yoshinori Matsunobu, Siying Dong, Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph.

CONTENTS

Background

InnoDB의 문제

RocksDB의 구조

RocksDB의 이점

Migration 중 해결해야 할 문제들

InnoDB의 문제

기존에 HDD를 사용하던 UDB를 SSD로 바꾸면서, read throughput은 증가했으나, 상대적으로 비싼 SSD로 인해 공간 절약이 필요해졌다. 하지만, B Tree 구조를 사용하는 InnoDB는 특성상 index fragmentation 현상이 쉽게 발생했고, disk block의 약 20~25%가 낭비되었다.

게다가 데이터 압축을 하게 되더라도, block의 크기가 1, 2, 4, 8KB로 제한되어 있어서 역시 디스크 공간을 낭비하게 되었다. 가령 압축된 데이터가 5KB가 되어도, 8KB 블록을 써야 하기 때문에 3KB가 낭비되는 것이다.

또한 MySQL이 caching이 되지 않는 하드웨어에서 동작하고 있기 때문에 쓰기 증폭 현상이 발생한다. 가령 8KB 페이지의 row 중 하나만 변경이 되어도, write는 전체 페이지를 해야 한다는 말이다. 게다가 InnoDB가 double-write 구조를 사용하고 있기 때문에 이 쓰기 증폭 현상이 더 심화되었다.

RocksDB의 구조

Write 연산은 우선 메모리 상의 write buffer 인 MemTable (LSM Tree 에서 C_0에 해당한다.)로 들어간다. MemTable 의 크기가 임의의 크기를 넘어섰을 경우, 그 내용물을 SST(Sorted Strings Table) File 로 flush 한다.

SST File(LSM Tree에서 C_1~C_K의 multi-page block에 해당한다.)은 데이터를 블록으로 나누어 저장하며 그 순서는 정렬되어 있다. 또한 각 SST File 에는 탐색을 위한 Index Block 이 포함되어 있다. SST File 들은 exponential 하게 크기가 증가하는 Level(LSM Tree에서 C_1~C_K에 해당한다.)로 정리되어 있다. 가장 마지막 Level은 Lmax라고 한다.

각 Level의 크기를 관리하기 위해, Level의 크기가 일정 수준 이상이 되면 SST File 일부를 선택하여 다음 레벨의 SST File들과 병합한다. 이를 Compaction 과정(LSM Tree에서 Rolling Merge에 해당한다.)이라고 한다.

탐색 과정 또한 LSM Tree와 같다. MemTable부터 시작하여 Lmax까지 순서대로 탐색을 진행하며, 해당하는 row 를 찾는다면 다음 Level로 진행하지 않고 정지한다.

RocksDB의 이점

기본적으로 LSM Tree 는 in-place update 를 사용하지 않는다. 모든 update 를 MemTable에 일단 모아두고, 어느 정도 모였을 때 한번에 flush하기 때문에 쓰기 증폭 현상이 줄어든다.

또한 LSM Tree는 그 특성 상 B Tree 구조가 가지는 index fragmentation의 문제를 갖지 않는다 (각 블록의 크기가 특정 값으로 제한되지 않았기 때문에). 물론 Tombstone 이 쌓일 수는 있으나, 이는 후술할 구현으로 어느 정도 완화할 수 있다.

InnoDB는 TXN을 처리하는 데에도 상당한 공간을 사용하는데, 그에 반해 RocksDB는 매 compaction마다 sequence number를 0으로 바꾸어서 데이터 압축의 효율을 올린다.

Migration 중 해결해야 할 문제들

1. 늘어난 CPU, 메모리, I/O 부담
   LSM Tree 구조는 DB의 크기를 절반 정도로 압축하는데, 이로 인해 CPU와 메모리, I/O에 대한 부담이 커진다. (같은 I/O 양에 비해 2배 많은 데이터량, 압축 및 해제에 따른 연산량 등)
   B Tree에 비해 Key 비교 횟수가 많다.(다수의 Level을 관리하기 때문에)
   성능 상승을 위해 Bloom Filter를 사용하기 때문에 메모리 부담이 증가한다.
   Compaction 연산으로 인해 write가 증가한다.

2. Scan 방향에 따른 overhead 차이
   LSM Tree의 delta encoding 방식으로 인해 backward scan이 forward scan에 비해 느려짐

3. Range Query의 성능

4. Tombstone 의 관리
   LSM Tree의 특성 상 DELETE 연산이 많을 경우 Tombstone이 많이 쌓이고, 이는 성능 저하의 원인이다.

구현

기본 구현

Migration 중 구현

기본 구현 – CPU 부담 개선

1. Mem-Comparable Key
   LSM Tree 구조는 여러 개의 Level로 구성되어 있기에, 어쩔 수 없이 탐색에 더 많은 Key Comparison을 요한다. 따라서 비교 연산의 성능을 올리기 위해 MySQL 데이터를 bitwise comparison이 가능한 형식으로 encoding한다.

2. Reverse Key Comparator
   RocksDB 는 여러가지 이유로 (byte encoding, 단방향 Skip List 구조인 MemTable 등) backward scan에 맞지 않는다. 다만, UDB 쿼리가 대부분 비슷한 형태를 가지기에, data의 구조를 이에 맞게 조정해 놓을 수 있다.
   또한 backward scan의 성능을 높이기 위해 keys를 reverse order로 저장한 comparator를 두어, 실제로는 forward scan으로 원하는 key를 찾을 수 있게 한다.

3. Faster Approximate…?

기본 구현 – Latency 개선

1. Prefix Bloom Filter
   LSM Tree에서의 탐색은 기본적으로 모든 레벨의 페이지 하나를 읽어야 하기 때문에, Seek Latency가 많이 발생한다. 따라서 각 key의 접두사 몇 바이트를 가지고 Bloom Filter를 만들어서, 해당하는 접두사가 없는 경우, 그 레벨의 탐색을 생략할 수 있다.
   또한 이는 range scan에도 쓰이는데, 접두사의 비교를 통해 탐색의 범위를 찾을 수 있기 때문이다.

2. Tombstone 줄이기
   기본적으로 Tombstone은 compaction 중 대응되는 record 를 발견할 경우 이를 지운다. 하지만 Tombstone 자체는 지워지지 않는데, 이는 다음 compaction 단계에서 다시 대응되는 record를 만날 수 있기 때문이다. 이렇게 누적된 Tombstone은 range scan 의 성능 저하를 야기한다.

   따라서 SingleDelete라는 DELETE 연산을 만들어서 대응되는 record를 만나면 바로 Tombstone 또한 사라지도록 한다. 물론 이를 위해 같은 key에 대해서는 연속해서 PUT이 발생할 수 없도록 제한해야 한다.

   이와 함께, Delete Triggered Compaction 을 두어서 어떤 SST File이 일정 비율 이상의 Tombstone을 포함할 경우 이 SST File을 바로 Compaction 하도록 한다.

기본 구현 – 공간 사용량 개선

1. DRAM 사용량 줄이기
   각 레벨 별로 Bloom Filter를 두는 것은 메모리에 상당한 부하를 준다. 따라서, Lmax에 대해서는 Bloom Filter를 두지 않을 수 있게 한다. Lmax가 전체 데이터의 약 90%를 차지하기 때문에 이는 메모리 부하를 상당히 줄일 수 있다. 물론 이로 인해 특정 operation의 비용이 커질 수 있다.

2. Compaction 으로 인한 SSD 성능 저하
   Compaction 은 종종 많은 양의 파일 삭제를 야기하고, 이로 인해 성능이 주기적으로 튀는 부분이 생긴다. 이를 방지하기 위해 파일 삭제하는 속도를 제한한다.

3. Stale Data에 대한 삭제
   각 데이터의 age를 보고 일정 수준 이상으로 오래되었다면, Lmax까지 compaction하도록 한다.

4. Bulk Loading
   몇몇 Burst Write가 발생하는 곳에서는 Bulk Loading을 통해 바로 Lmax로 compaction되도록 한다. (다른 레벨을 거치지 않는다.) 물론 Bulk Loading하는 데이터의 범위가 기존 데이터와 겹쳐서는 안된다.

향후 계획

읽어볼 논문

다음 단계

읽어볼 논문

1. HYU-SCSLAB의 rolling merge(compaction) 관련 논문
   “LSM-tree 기반 Key-value 데이터베이스의 재귀적 컴팩션 기법”
   기존 LSM Tree의 성능 문제의 해결책 중 하나, 이를 따라가지는 않아야 하지만, 참고할 가치가 있음

2. 성균관대의 LSM Tree 관련 논문
   “BoLT: Barrier-optimized LSM-Tree”
   LSM Tree 의 Compaction에 대한 논문인 듯

3. Thomas Lively의 LSM Tree 관련 논문
   “Splaying Log-Structured Merge-Trees”
   LSM Tree 기반 시스템의 읽기 성능에 대한 논문인 듯

다음 단계

RocksDB 동작 분석
GDB, cscope를 통해 실제 function flow 확인하기

perf 등을 통해 성능 병목 확인하기

참고할 만한 슬라이드
https://www.slideshare.net/meeeejin/rocksdb-detail

https://www.slideshare.net/meeeejin/rocksdb-compaction

https://www.slideshare.net/HiveData/tech-talk-rocksdb-slides-by-dhruba-borthakur-haobo-xu-of-facebook

THANK YOU!

감사합니다!