국문제목

rocksdb에서 read optimization

영문제목

Read optimization in rocksdb

요 약

Rocksdb에서는 read optimization을 위해 bloom filter라는 자료구조를 사용한다. 이 bloom filter는 필요 없는 I/O를 줄이는 장점이 있지만, CPU overhead가 많이 발생한다. 이를 해결하기 위해 많은 연구와 시도가 있었으며, 이 부분을 공부해 보았다.

1. 서 론

key - value store databases는 비 관계형 데이터베이스로, key와 value를 쌍을 이루어 저장한다. key를 고유한 식별자로 사용하고, key를 사용하여 값을 검색한다. 관계형 데이터베이스와 비교하면, 비 관계형 데이터베이스는 상대적으로 상당한 유연성을 가진다. rocksdb는 페이스북에서 시작된 오픈소스 데이터베이스 개발 프로젝트로, key - value 저장 방식이다. c++로 작성됐으며, 빠른 저장장치와 서버 워크로드에 적합하다. rocksdb에서는 읽기 최적화를 위해 bloom filter라는 자료구조를 사용한다. bloom filter는 불필요한 I/O를 많이 줄일 수 있지만, 높은 CPU overhead를 발생시킨다. 이 CPU overhead를 줄이기 위해 많은 연구가 있었다.

2. bloom filter

bloom filter는 membership test를 위한 확률적 자료구조로 Burton Howard Bloom 에 의해 1970년도에 고안되었다. bloom filter는 어떤 원소가 집합에 속한다고 판단되는 경우 실제로 원소가 집합에 속하지 않는 긍정 오류(false positive)가 발생하는 것은 가능하지만 반대로 원소가 집합에 속하지 않는 것으로 판단되었는데 실제로는 원소가 집합에 속하는 부정 오류(false negative)는 절대로 발생하지 않는다는 특징이 있다. 즉 이것을 보면 bloom filter는 ‘데이터가 확실히 있다’는 것에 초점이 맞추어진 자료구조가 아니라 ‘데이터가 확실히 없다’에 초점이 맞추어진 자료구조이다.

false positive가 발생하게 되면 불필요한 I/O가 발생된다. 또한 bloom filter에서 hash 함수를 많이 사용할 수록 높은 CPU overhead가 발생한다. 그래서 false positive의 비율과 cpu overhead 이 두 가지가 bloom filter의 성능에 중요한 영향을 끼친다.

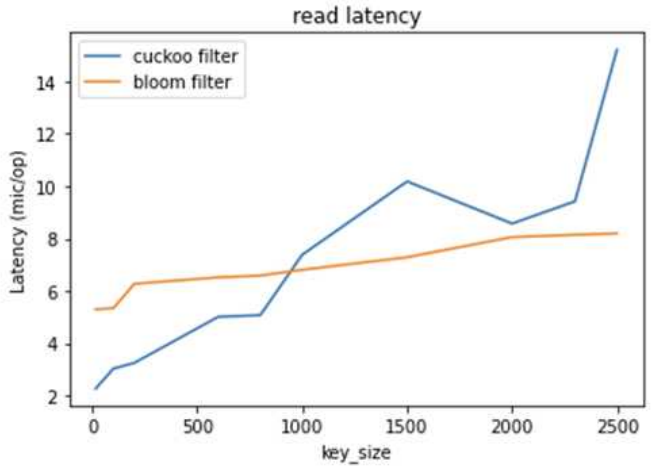
2-1 bloom filter 원리

기본적인 bloom filter는 0으로 초기화된 이진배열이다. 원소를 추가할 때는 추가하려는 원소에 대해 k개의 해시 값을 계산한 다음 각 해시 값에 대응하는 비트를 1로 맵핑한다. 원소를 검사하는 경우, 해당 원소에 대해 k가지의 해시 값을 계산한 다음, 각 해시 값에 대응하는 비트 값을 읽는다. 만약 모든 비트가 1인 경우 positive 아니라면 negative 값을 판정한다. 만약 원소가 storage에 없음에도 해시값에 대응하는 비트가 기존에 추가된 원소로 인하여 전부 1인 경우 false positive가 발생한다. db query 중 false positive가 발생하게 될 경우 필요 없는 storage scan이 필요하게 되며, 만약 second storage가 느린 경우에 false positive는 치명적일 수 있다. 그러므로 false positive rate는 되도록 낮아야 한다. bloom filter에 false positive확률은 $(1-e^{(kn/m)})^k$이다(k는 hash function의 수, n은 삽입된 원소의 수, m은 filter의 크기이다). 즉 false positive의 확률을 줄이기 위해선 filter에서 key당 bit(k)가 커야 한다. 그러나 k가 증가하면 할수록 계산해야 하는 hash function의 수가 증가한다.

3. cuckoo filter

cuckoo filter역시 bloom filter와 같은 확률적 자료구조이고 조회 기능에 더 최적화된 자료구조이다. bloom filter

보다 공간 효율성이 높고 전체적으로 bloom filter보다



(그림.1)

false positive 비율이 적으며, bloom filter에서 지원하지 않는 delete 기능을 지원한다. cuckoo filter는 이름에서 보이는 것처럼 뻐꾸기처럼 삽입하려는 index에 다른 값이 들어가 있으면 그 값을 다른 index로 보내고 자리를 차지한다. 그리고 이 과정에서 cuckoo filter는 2개의 해시 함수를 사용하게 된다. 하지만 cuckoo filter의 array에 모든 값들이 들어가게 되면 삽입이 실패할 수 있다. bloom filter가 삽입 실패가 절대 일어나지 않는 것에 비해 대조적이다.

4. bloom filter와 cuckoo filter의 비교

bloom filter와 비교해서 cuckoo filter는 거의 모든 성능이 우수하다. 조회 기능에 최적화되어있고 공간 효율성도 높으며 bloom filter보다 false positive rate도 낮으며 delete기능을 지원하고 해시함수도 적게 사용한다.

그래서 모든 면에서 cuckoo filter가 bloom filter보다 성능이 좋아보지만 사실은 그렇지 않다.

cuckoo filter는 L3같은 높은 sst file 레벨 또는 DRAM에서 filter가 사용된다면 cache miss가 더 많이 발생하기 때문에 조회 비용이 더 크다. 그래서 cuckoo filter는 높은 조회 비용과 낮은 fp 비율을 가지고 있고, bloom filter는 낮은 조회 비용과 높은 fp비율을 가지고 있다. 이는 저장 하는 장치의 속도와 저장해야 하는 데이터의 크기에 따라 선호하는 것이 다르다.

속도가 빠르고 저장해야 하는 데이터 크기가 큰 상황에서는 높은 fp비용보다 낮은 조회비용이 더 중요하고, 반대로 속도가 느리고 저장해야 하는 데이터 크기가 작은 상황에서는 높은 조회비용보다 낮은 fp 비율이 더 중요하다. 그래서 많은 데이터 사이즈를 저장해야 하는 상황에서는 bloom filter가 더 우세하고 적은 데이터 사 이즈를 저장해야하는 상황에서는 cuckoo filter가 더 우세하다.

(그림.1)에서 확인할 수 있는 거처럼 초반에는 더 우수한 성능을 보여주던 cuckoo filter는 key size가 늘어나자 성능이 떨어지는 모습을 보였다. 그에 비해 bloom filter는 초반에 cuckoo filter 보다 낮은 성능을 보였지만 데이터 양이 증가하여도 성능이 별로 떨어지지 않으면서 마지막에는 cuckoo filter보다 좋은 성능을 보였다.

5. bloom filter의 cpu overhead

Bloom filter는 BF query 즉 메모리에 대한 해싱 및 접근과 HDD,SSD 같은 보조 기억장치에 대한 접근의 속도의 차이가 클 때 더 유용하다. bloom filter가 느린 보조 기억장치의 접근을 최대한 줄이기 때문에 효율적인 성능을 보인다. 보조 기억장치의 속도가 느릴수록 bloom filter의 효율성이 증가하게 되는 것이다. 하지만 점점 스토리지의 속도가 SSD와 같은 NVM 등이 발달하면서 스토리지 속도가 빨라지고 이는 bloom filter가 갖는 이점이 약화되었다.

블룸 필터는 데이터 엑세스를 할 때 해싱 계산이 필요하다. hash 계산이 물론 좋은 기법이긴 하지만 상황에 따라 해시 계산을 많이 할 때도 있고 점점 스토리지의 속도가 빨라지면서 불필요한 I/O를 줄여준다는 장점이 약화되었다. 그리고 hash 계산 비용은 LSM트리의 레벨이 높아질수록 즉 key size가 커질수록 증가하게 된다. 즉 bloom filter의 cpu overhead는 hash 계산에 의해서 발생하게 된다.

6. elastic bf

실제 환경에서 database에 data 접근은 매우 치우치게 발생된다. 어떤 KV pairs는 많이 접근되는 것에 비해 어떤 KV pairs는 거의 접근이 되지 않는다. 이러한 특성을 이용하여, 자주 사용되는 SST file에 더 많은 filter를 할당할 경우 false positive rate를 줄일 수 있다. 이 방법에는 문제가 있는데, 첫 번째로 다른 레벨에 있는 SST table사이에 접근 불균형이 매우 크지 않다는 것이다. low level에 있을수록 접근이 많이 되는 것이 맞지만, high level에 있는 data도 접근이 많이 된다. 두 번째로는 같은 SST table 내에서도 접근 불균형이 발생한다. 세 번째로는 자주 접근되는 KV pairs가 수시로 변하는 것이다. 위 문제점을 해결하기 위한 방법으로 elastic bf가 있다. 이 방법은 매우 작게 나눠진 bloom filter을 사용하는 것이다. KV pairs들의 접근도를 meta data에 기록하여, 접근도에 따라 bloom filter를 할당하는 것이다.

7. hash sharing

cpu overhead를 줄이기 위해 가장 중요한 것은 hash 계

산을 최대한 줄이는 것이다. 이를 위해 전에 사용했던 해시 계산을 다시 사용하는 hash sharing을 사용한다.
hash sharing의 장점으로는
첫 번째 SST table 즉 LSM 트리의 모든 레벨의 hash비용을 일정하게 유지할 수 있어서 트리가 높아질수록 overhead가 더 걸리는 것을 방지할 수 있다.
두 번째 hash sharing을 사용하면 False positive의 비율을 줄일 수 있습니다. FP비율이 줄게 되면 불필요한 데이터 접근이 줄게 되어서 결과적으로 불필요한 hash 계산이 줄게 되어 overhead가 줄어든다.
마지막으로 double hashing을 사용하면 더 적은 fp비율과 더 좋은 성능을 얻을 수 있는데 일반 hashing보다 hash 계산이 배로 들기 때문에 key size가 큰 자료를 다룰 때 불리하다. 하지만 hash sharing을 사용하면 overhead를 줄이고 double hashing의 장점이 극대화할 수 있다.

7-1. hash sharing의 방법

앞서 bloom filter의 작동방식에서 hash 함수를 통해 계산되어서 나온 hash digest를 사용해서 membership query를 한다는 것을 알았다. 여기에서 hash digest만 모든 sst file에 똑같이 재사용을 해서 hash sharing을 할 수 있다. 이렇게 사용한다면 몇 번을 반복해서 해시 계산을 해야 하는 것을 한번의 계산으로 줄일 수 있다.
또 다른 방법은 bit rotation이다. 해시 함수를 통해 나온 hash digest가 변경시킨 bit array를 index의 위치만 변경시켜서 다른 bit array를 만다는 것이다 이런 작업을 bit rotatino이라고 한다. 그리고 이렇게 변경된 bit array를 가지고 작업을 실행한다. 이런 bit rotation을 하게 된다면 원래는 해시 계산을 여러번 해야 하는 것을 1번으로 줄일 수 있고 bit rotation의 계산은 cpu over head가 거의 없는 수준이기에 빠르게 계산을 할 수 있다.