Performance Analysis
RockDB 분석

컴퓨터소프트웨어학부
김현정 박정호

CONTENTS

Perf

사용한 옵션

분석 결과

사용한 옵션 : sudo perf report -g graph -no-children
sudo perf report -g graph --children

```
11.63%  read_bottleneck  libc-2.27.so                  [.] __random_r
 9.71%  read_bottleneck  libc-2.27.so                  [.] __random
 6.20%  read_bottleneck  read_bottleneck_breakdown_test [.] main
 2.27%  read_bottleneck  libc-2.27.so                  [.] rand
 2.02%  rocksdb:low0     [kernel.kallsyms]             [k] copy_user_enhanced_fast_string
 1.94%  read_bottleneck  read_bottleneck_breakdown_test [.] rand@plt
 1.43%  read_bottleneck  [kernel.kallsyms]             [k] copy_user_enhanced_fast_string
 1.20%  rocksdb:low0     libc-2.27.so                  [.] __memmove_avx_unaligned_erms
 1.16%  read_bottleneck  libc-2.27.so                  [.] cfree@GLIBC_2.2.5
 1.06%  read_bottleneck  [kernel.kallsyms]             [k] do_syscall_64
 1.02%  read_bottleneck  libc-2.27.so                  [.] __memmove_avx_unaligned_erms
 0.99%  rocksdb:low0     libc-2.27.so                  [.] __memset_avx2_erms
 0.86%  read_bottleneck  [vdso]                        [.] __vdso_clock_gettime
 0.82%  rocksdb:low0     libsnappy.so.1.1.7            [.] snappy::internal::CompressFragment
 0.81%  read_bottleneck  read_bottleneck_breakdown_test [.] rocksdb::MemTable::KeyComparator::operator()
 0.67%  read_bottleneck  read_bottleneck_breakdown_test [.] rocksdb::IndexBlockIter::SeekImpl
 0.65%  read_bottleneck  libc-2.27.so                  [.] malloc
```

분석 결과

- 제외하면 좋은 경우들
  1. malloc, memset
  2. string을 read하거나 copy하는 경우

  3. key값을 compare하는 operation
  4. lock, unlock
  5. 라이브러리 함수들

-> 메모리 alloc과 관련된 경우
-> string read와 copy는 필수불가결한 부분이고, 이를 줄이기 위해 string의 포인터를 받는 slice등이 쓰이고 있기 때문.
-> 하드웨어 연산이 필요한 경우.
-> 성능을 위해 잘 설계되어 있을 것이라 가정.
-> snappy와 같은 라이브러리 함수들

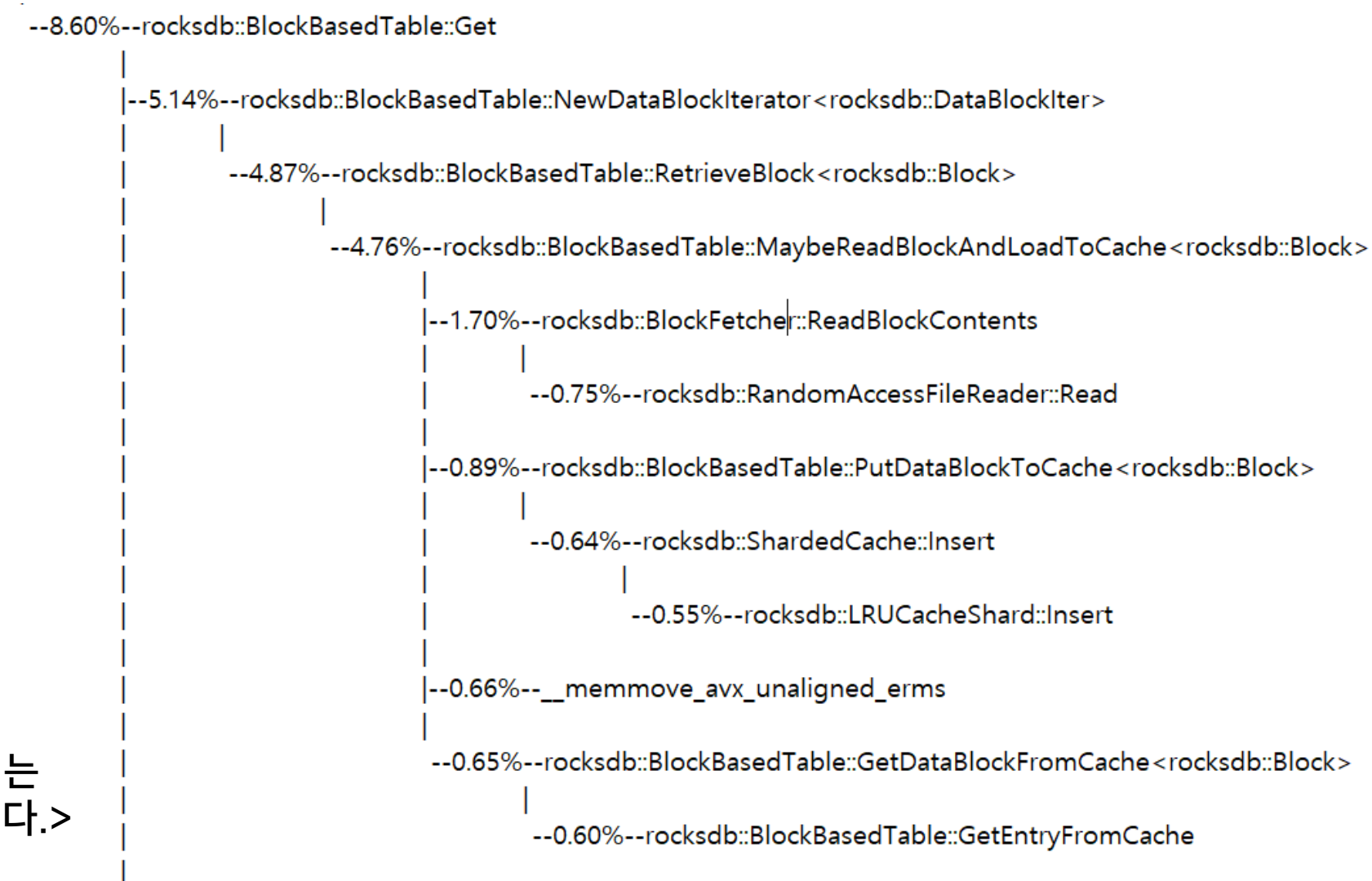
- 제외가 불확실한 경우들
  1. iterator
  2. key값을 compare하는데 쓰이는 구조나 로직

-> iterating을 수행하는 함수나 compar하는 함수들이 상위 순위에 종종 눈에 띔. Data structure자체에 관한 문제일 수 있으므로 보류.

- 병목이 의심되어 확실하게 볼 필요가 있는 부분
  1. block base table이라는 namespace에서의 operation

분석 결과

1. SST file과 Cache에서 block을 읽어오는 과정



<이 페이지와 다음페이지의 이미지는 -children 옵션으로 분석한 내용이다.>

분석 결과

- Index block에서 data block을 찾아오는 과정

```
|
|--1.72%--rocksdb::BlockIter<rocksdb::IndexValue>::Seek
|          |
|           --1.61%--rocksdb::IndexBlockIter::SeekImpl
|                     |
|                      --0.56%--rocksdb::BlockIter<rocksdb::IndexValue>::CompareCurrentKey
|
 --0.51%--rocksdb::DataBlockIter::SeekImpl
```

--children옵션으로 분석한 결과를 토대로 다시 -no-children옵션으로 분석했을 때, 기타 시스템적인 함수나 라이브러리 함수를 제외하고 rocksdb에서 동작하는 함수 중에서 비교를 한 결과는 아래와 같다.
2위(0.67%)는 IndexBlockIter::SeekImpl 이다.
6위는(0.40%)는 BlockIter<rocksdb::IndexValue>::CompareCurrentKey 이다.
→ index block이 binary search 구조를 가지고 prefix seek를 사용하고 있다는 점에서 개선할 여지를 생각해 볼 수 있을 것 같다.

분석 결과

- Block base table은 sst file의 기본적인 구조이긴 하지만 get을 빠르게 수행하기 위해서 prefix seek와 binary search data structure를 가지고 있다. 이 과정에서 많은 cpu가 소요되는 것으로 보인다.
- 분석의 근거를 찾기 위해 facebook에서 작성한 wiki를 찾아보았는데, 이와 관련해서 해결점을 찾기 위해 시도한 방법이 보였다.
  https://github.com/facebook/rocksdb/wiki/Data-Block-Hash-Index
  hash index를 통해 binary search의 range lookup이외에도 point lookup을 지원할 수 있지만, support되는 datatype의 한계가 있고 index의 datatype size로 인한 interval의 한계점도 존재한다.
- 사용자가 옵션으로 hash index를 사용하더라도 위의 한계점에 부딪히면 기존의 binary search로 수행이 된다.

Call Graph

실험 방식

Callgrind 결과

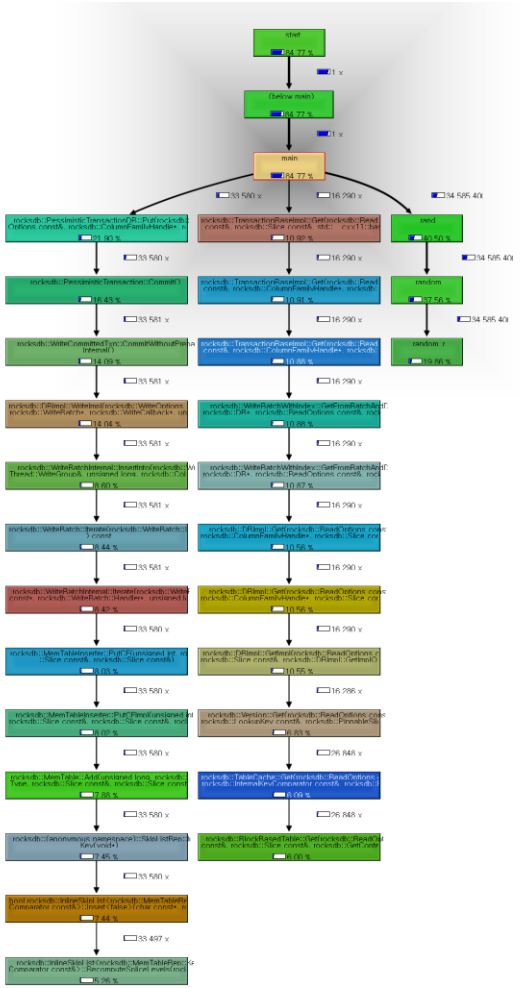
실험 방식

Call Graph를 구하기 위해 valrind에 내장된 툴인 callgrind를 사용했다.

callgrind의 결과는 단순한 텍스트이기 때문에 이를 시각화 하기 위해서 Kcachegrind를 사용했다.

사용한 결과는 오른쪽과 같다. 왼쪽이 PUT, 가운데가 GET, 오른쪽이 rand (무작위의 key-value 를 삽입/탐색하기 위해 사용)이다.

일정 이상 branch 가 나눠질 경우 축약된 상태로 그려지며, 각 블록을 클릭하면 해당 블록에서부터 시작되는 call graph 를 다시 그려준다. 이를 이용해서 좀더 자세한 call graph를 확인할 수 있다.

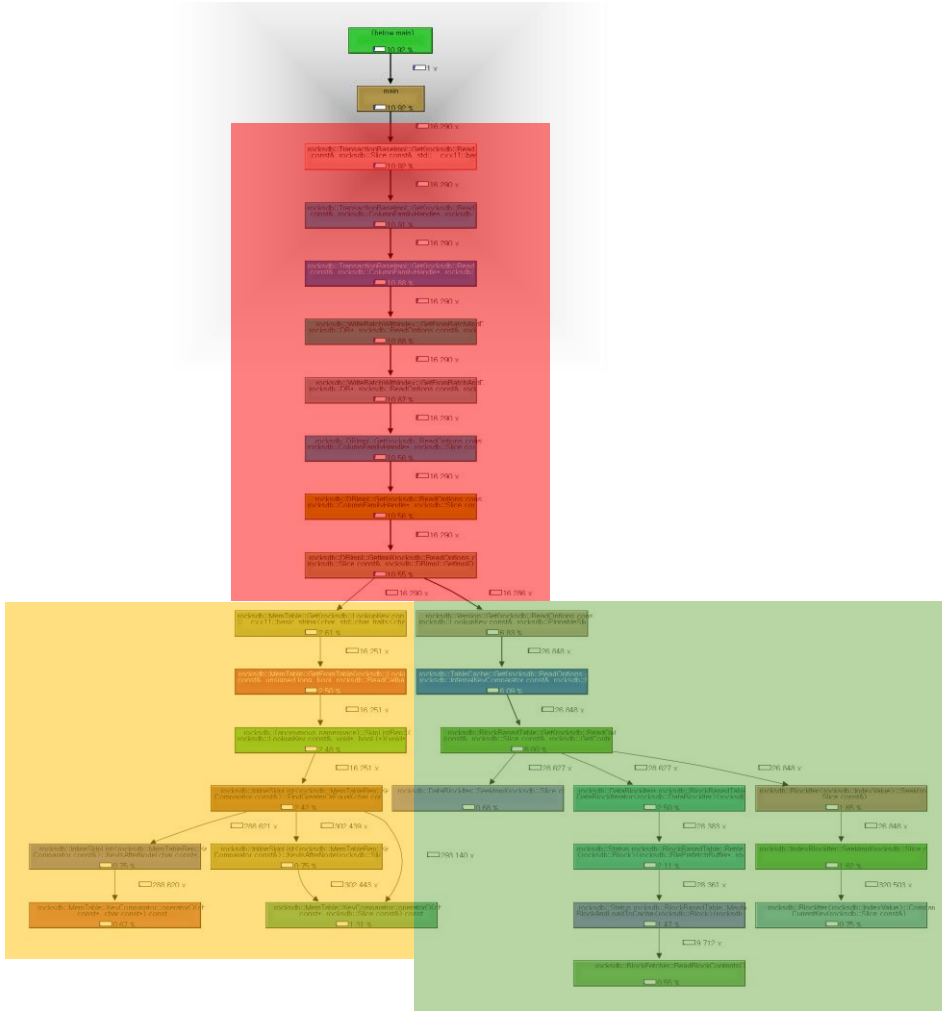


Callgrind 결과 (1)

Get의 전체적인 Call Graph 는 다음과 같다. Kcachegrind 가 아래쪽 일부 branch를 생략하고 보여준 상태이다.

크게 나눠서 보자면, 붉은 영역은 wrapper 함수들이 포함된 영역이다. 이 과정에서 WriteBatch 에서의 탐색이 수행된다.

그 아래의 노란 영역은 memtable 에서의 탐색이 수행되는 영역이고, 초록색 영역은 version 에서의 탐색, 즉 SST File 에서의 탐색이 수행되는 영역이다. 이 두 영역에는 생략된 path가 있으므로, 다음 페이지에서 더 자세한 call path graph를 보도록 하겠다.

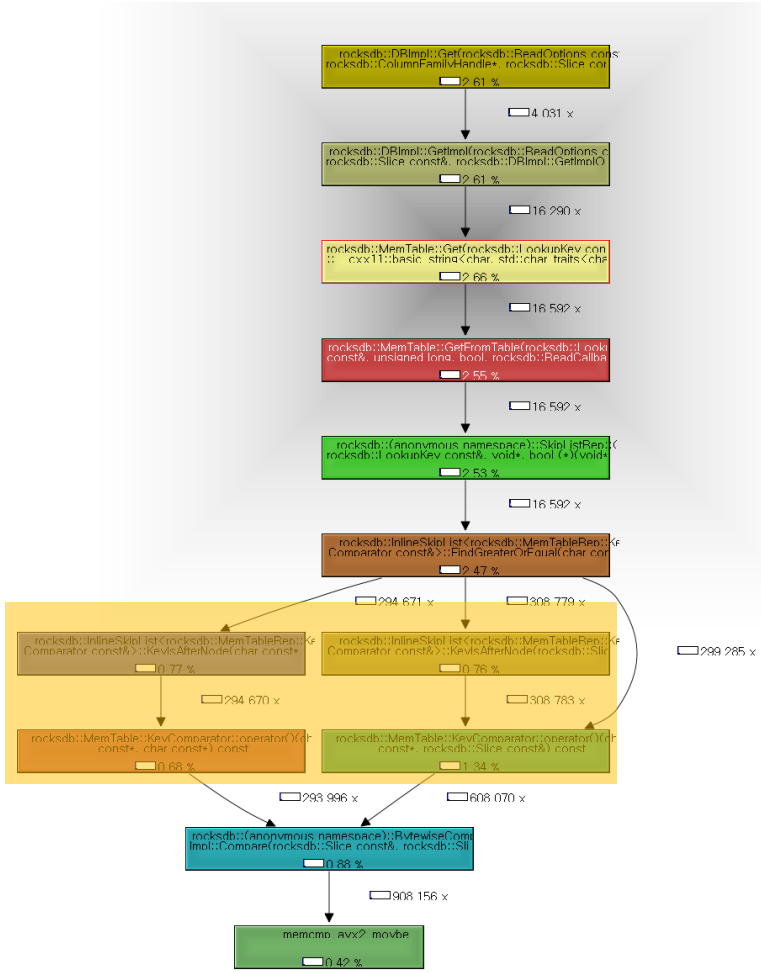


Callgrind 결과 (2)

memtable 에서의 탐색 path 의 call graph이다.

Skip list를 순회하면서 찾고자 하는 key 이상의 key를 갖는 노드를 찾고, 찾아낸 key와 target을 비교하는 과정을 갖고 있다.

노란색 영역에 해당하는 부분은 시스템의 무결성을 확인하는 assert 문에 사용된 것으로, 리스트가 정렬된 상태이고, 이미 target을 지나치지 않았는지 확인하는 과정이었다.

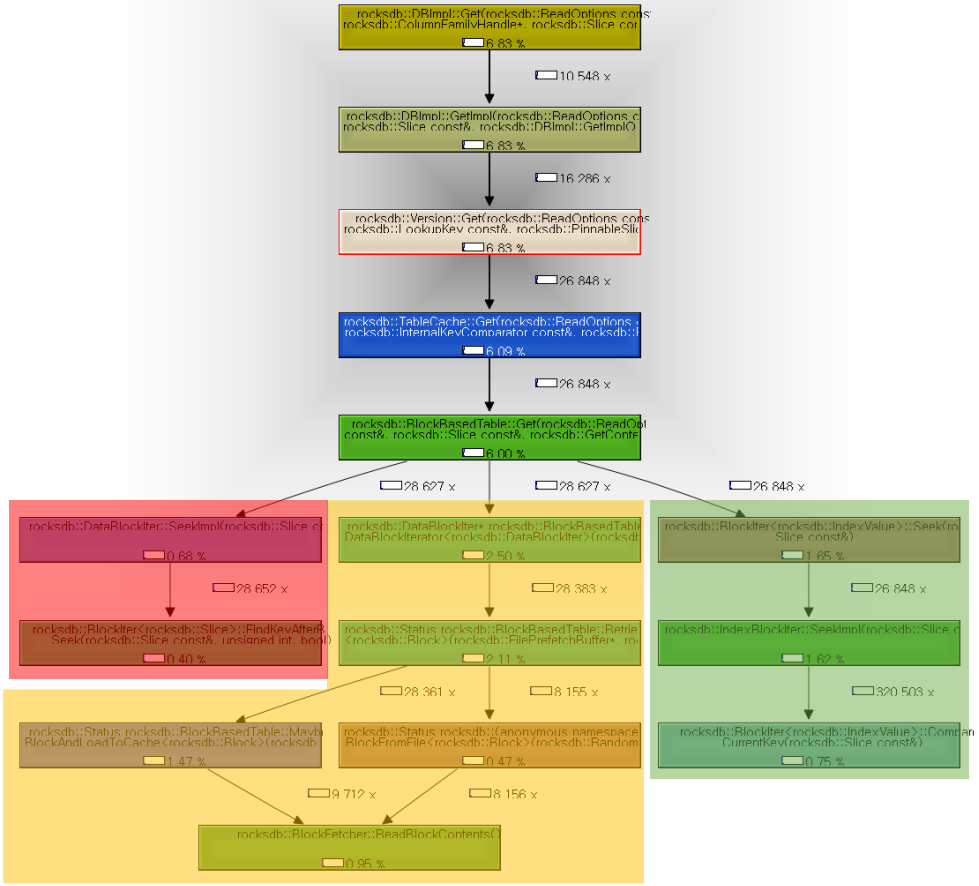


Callgrind 결과 (3)

SST File 에서의 탐색 path 의 call graph이다.

노란색 영역은 SST File 혹은 cache 에서 block 을 읽어오는 과정, 초록색은 인덱스 블록에서 데이터 블록을 찾아오는 과정, 붉은색 영역은 이렇게 찾아온 데이터 블록에서 원하는 key-value 를 탐색하는 과정이었다. (callgrind의 그래프가 순차적인 형태를 띄지는 않는 것 같다.)

기본적으로 bloom filter 를 사용해서 의미없는 탐색을 줄이고, 인덱스 블록을 사용해서 데이터 블록을 읽어와서 탐색을 진행하는 방식을 사용하고 있었다.



To Do

해결해야 할 문제

질문

해결해야 할 문제 및 질문

1. perf 를 사용해서 병목이 되는 부분을 찾으려 하는데, 현재 사용하려는 방식이 맞는 것인지 궁금하다.

2. Perf사용해서 보니 cpu cycle이 소요되고 있는 부분 중 제외해도 될 것 같은 요소들이 존재하는데 맞게 판단을 한 것인지 잘 모르겠다.
3. 현재 분석에서 병목이 된다고 예상이 되는 부분을 발견하고 wiki를 찾아본 결과 facebook에서 개선하려고 시도한 방법을 발견했는데 default가 아니라 옵션으로 제공되는 부분이어서 아직 perf로 해당 해결책이 적용된 성능을 체크해보지 못했다.
4. Wiki상으로는 3번의 해결책이 완전하지 않기 때문에 dafault로 제공되지 않는 것처럼 보였다. 아직 해당 옵션을 적용한 perf 분석을 하지 못했지만, default 옵션이 아니라는 이유로 이를 병목현상의 원인으로 볼 수 있는지가 의문이다.

THANK YOU!

감사합니다!