

LevelDB-Study

Team_Cache Analysis

Made by Subin Hong, Seungwon Ha

E-Mail: zed6740@dankook.ac.kr, 12gktmddnjs@naver.com

Contents

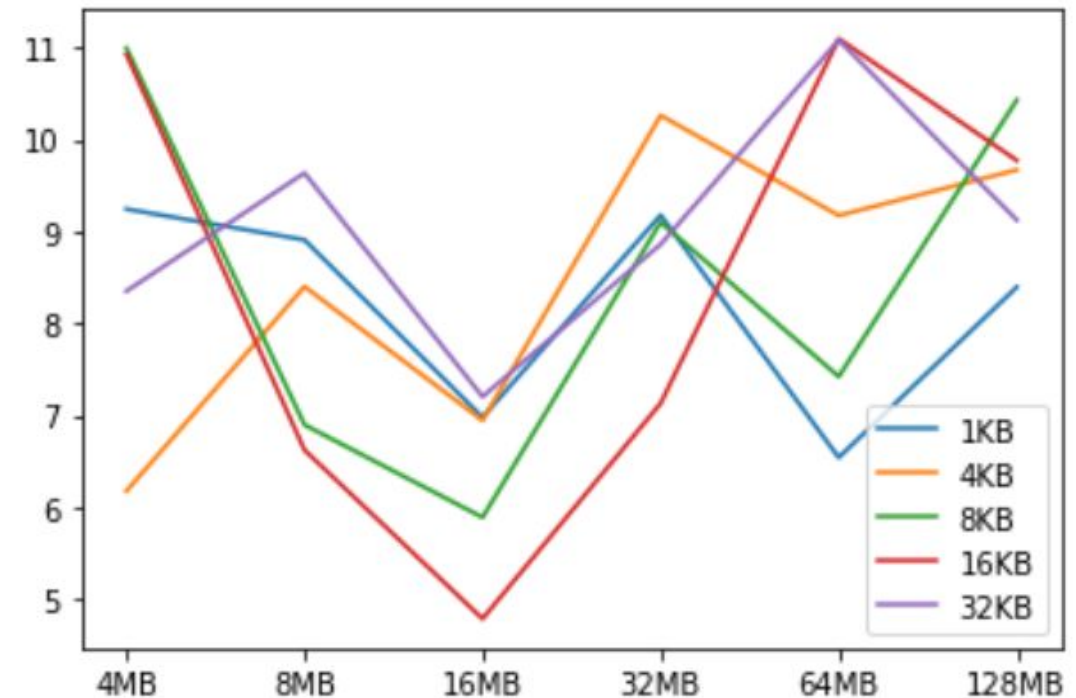
1. Benchmarks experiment
2. Source code analysis

- Benchmarks experiment
- Source code analysis

Benchmarks Experiment

1. Fillrandom[Load] 후 각 블록 사이즈에서의 캐시 사이즈 변화의 readrandom latency 측정

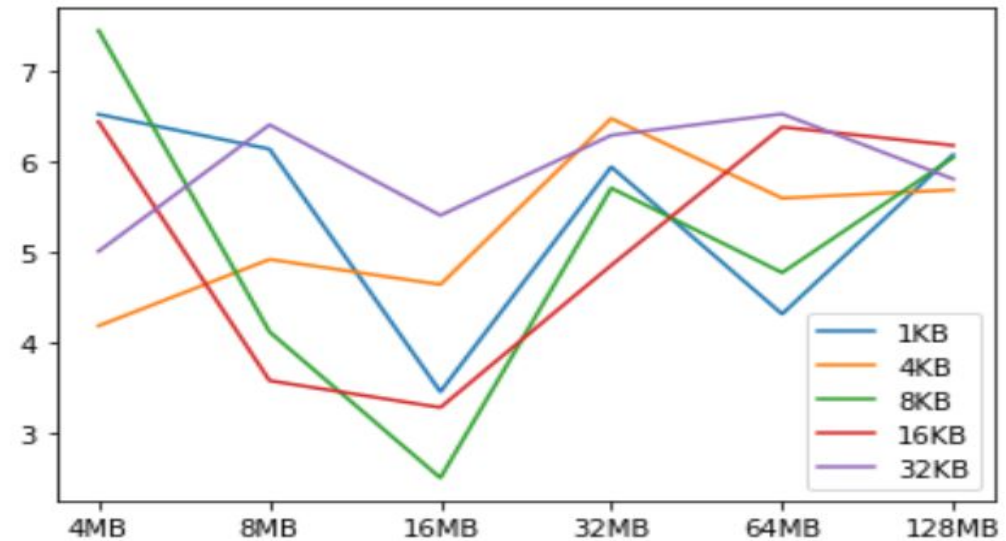
	1KB	4KB	8KB	16KB	32KB
4MB	9.243	6.177	10.994	10.926	8.351
8MB	8.908	8.400	6.897	6.623	9.635
16MB	6.979	6.945	5.891	4.788	7.203
32MB	9.177	10.262	9.100	7.131	8.860
64MB	6.541	9.173	7.421	11.095	11.081
128MB	8.396	9.669	10.429	9.772	9.124



Benchmarks Experiment

1. Fillrandom[Load] 후 각 블록 사이즈에서의 캐시 사이즈 변화의 readhot latency 측정

	1KB	4KB	8KB	16KB	32KB
4MB	6.512	4.176	7.437	6.429	4.999
8MB	6.129	4.910	4.109	3.570	6.397
16MB	3.447	4.632	2.496	3.274	5.396
32MB	5.932	6.466	5.698	4.845	6.281
64MB	4.306	5.586	4.763	6.372	6.517
128MB	6.066	5.678	6.034	6.169	5.798



Benchmarks Experiment

conclusion

readrandom, readhot 외에도 seekrandom, readseq로 실험을 수행했을 경우에도 각 `cache_size`가 올라감에 따라 성능이 좋아지는 모습은 보기 힘들었음

이에 따른 해결 방안

1. `cache`의 크기에 따른 성능 변화가 보이지 않으므로 `cache`크기를 큰 폭으로 증가시켜 성능 향상을 관찰

=> 캐시의 크기를 늘려 더 많은 항목을 캐시하여 성능 향상을 체크하기 위해

[4MB,8MB,16MB,32MB..] -> [1MB,10MB,100MB,1GB,10GB]

2. DB의 크기를 10GB로 늘림

=> 큰 폭의 변화를 관찰하기 위함, 이전까지는 DB크기가 작아 `latency`가 작게나와 변화의 차이가 미미했음

Benchmarks Experiment

+ Add knowledge

LSM 기반 키 값 저장소에는 두 유형의 읽기 작업이 있음 두 유형의 읽기 작업은 다른 캐싱 요구 사항을 나타냄

1.point-lookup(Get) (ex) readrandom ,readhot

=> 공간 효율성을 위해 개별 키-값 쌍을 캐싱하는 것을 선호

2.범위 쿼리(Scan = Seek, Next) (ex) readseq

=> 범위 쿼리를 지원하기 위해 캐싱 블록에 의존

다만 해시 기반 Key-Value Store는 범위 쿼리를 하기에는 공간적으로 이점이 없음

LevelDB는 LRU캐시 구조이며 해시 테이블을 기반으로 두 double linked list로 이루어져 있음

이 때문에 readrandom과 readhot을 option으로 확인해야

cache_size변화에 관한 read latency 변화를 체크가 가능할 것으로 예상

Benchmarks Experiment

```
./db_bench --benchmarks=readrandom,stats --use_existing_db=1 --num=100000000 --c  
ache_size=1000000000000 --block_size=4096 --compression_ratio=1
```

```
BlockSize:      4096  
CacheSize:      1215752192  
WriteBufferSize: 4194304
```

LevelDB 자체에서 Cache_size는 1.21GB를 넘지 못하게 되어있음을 확인
만약 넘을 경우 자체적으로 1.21GB를 설정해 줌

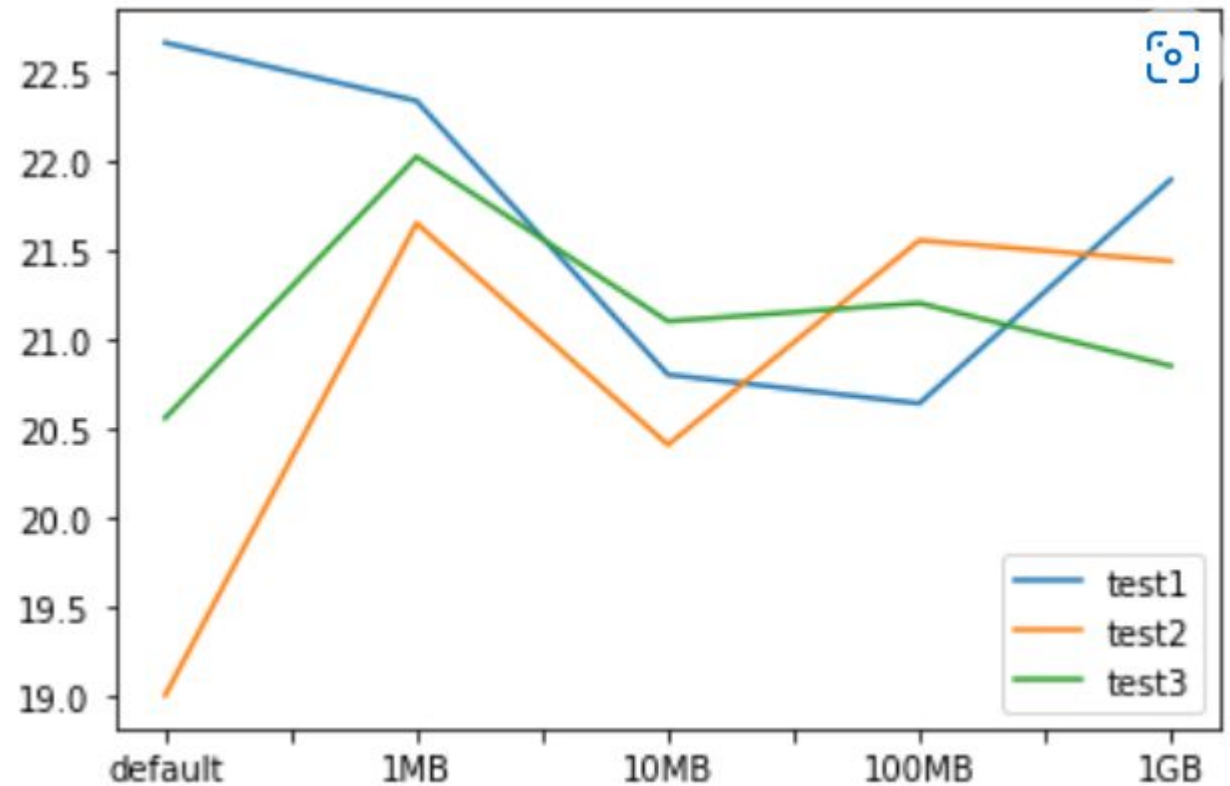
Benchmarks Experiment

[Load] fillrandom으로 고정

(1)readrandom

	test1	test2	test3
default	22.671	19.002	20.561
1MB	22.344	21.656	22.030
10MB	20.805	20.411	21.105
100MB	20.643	21.560	21.207
1GB	21.901	21.443	20.854

```
// Cache size. Default 4 MB  
static int FLAGS_cache_size = 4194304;
```

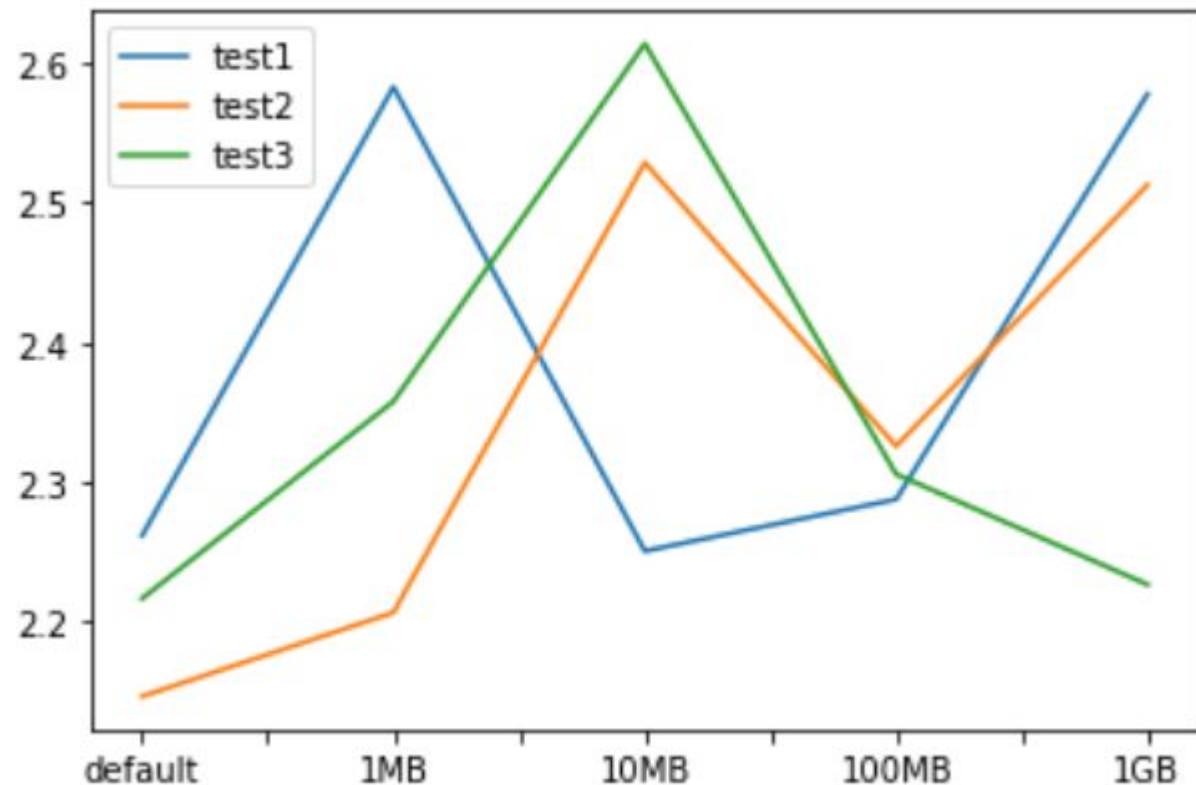


Benchmarks Experiment

[Load] fillrandom으로 고정

(2)readhot

	test1	test2	test3
default	2.262	2.147	2.217
1MB	2.583	2.207	2.358
10MB	2.251	2.529	2.614
100MB	2.288	2.326	2.306
1GB	2.578	2.513	2.227



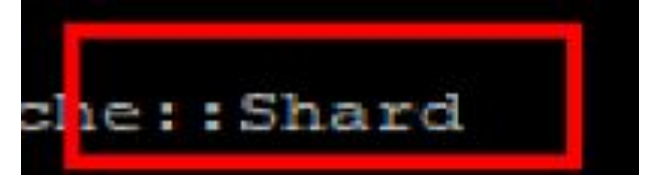
Source code analysis

LevelDB's Cache Type : BlockCache / TableCache

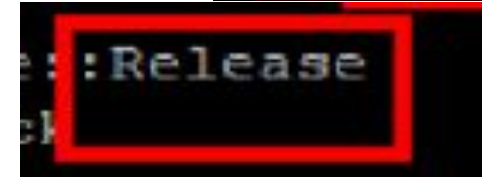
1. BlockCache : 캐시된 블록 데이터(cache.cc)



```
::Insert
```



```
che::Shard
```



```
::Release
```

2. **Tablecache** : 파일 시스템의 inode캐시와 유사한 **sstable**의 인덱스 데이터(table_cache.cc)

(tablecache의 **cacpacity**는 최대 **sstable** 파일의 인덱스 정보를 캐시할 수 있는 양을 나타냄)

=> 메모리에 있는 **SSTable**의 데이터 구조를 캐시하면 테이블을 사용하기 전에 테이블이 필요함
(table_cache)

Source code analysis

```
135.410 ms : ~T~B ~T~B ~T~B ~T~\ ~T~@(1) leveldb::DB::Open
           : ~T~B ~T~B ~T~B ~T~B ~T~\ ~T~@(3) operator new
           : ~T~B ~T~B ~T~B ~T~B ~T~B
253.242 us : ~T~B ~T~B ~T~B ~T~B ~T~\ ~T~@(1) leveldb::DBImpl::DBImpl
           : ~T~B ~T~B ~T~B ~T~B ~T~\ ~T~@(1) leveldb::DB::DB
```



```
0.086 us : ~T~B ~T~B ~T~B ~T~B ~T~B ~T~\ ~T~@(1) leveldb::TableCacheSize
          : ~T~B ~T~B ~T~B ~T~B ~T~B ~T~B
0.484 us : ~T~B ~T~B ~T~B ~T~B ~T~B ~T~\ ~T~@(3) operator new
          : ~T~B ~T~B ~T~B ~T~B ~T~B ~T~B
38.963 us : ~T~B ~T~B ~T~B ~T~B ~T~B ~T~\ ~T~@(1) leveldb::TableCache::TableCache
```

Source code analysis

```
static int TableCacheSize(const Options& sanitized_options) {  
    // Reserve ten files or so for other uses and give the rest to TableCache.  
    return sanitized_options.max_open_files - kNumNonTableCacheFiles;  
}
```

```
dbname_(dbname_),  
table_cache_(new TableCache(dbname_, options_, TableCacheSize(options_))),
```

dbimpl.cc의 DBImpl:DBImpl에서 tablecache의 크기를 설정 -> tablecache 생성

Source code analysis

<table_cache.h>

```
Iterator* NewIterator(const ReadOptions& options, uint64_t file_number,  
                    uint64_t file_size, Table** tableptr = nullptr);
```

Source code analysis

<table_cache.h>

```
Status Get(const ReadOptions& options, uint64_t file_number,  
           uint64_t file_size, const Slice& k, void* arg,  
           void (*handle_result)(void*, const Slice&, const Slice&));
```

Source code analysis

<table_cache.h>

```
void Evict(uint64_t file_number);
```


Source code analysis

<table_cache.h>

NewIterator -> FindTable / Get -> FindTable

```
Status FindTable(uint64_t file_number, uint64_t file_size, Cache::Handle**);
```

file_number: sstable 파일의 이름

file_size: sstable 파일 크기

handle: 반환할 sstable에 해당하는 cache 엔터티

캐시 조회로 이동

```
*handle = cache_->Lookup(key);
```

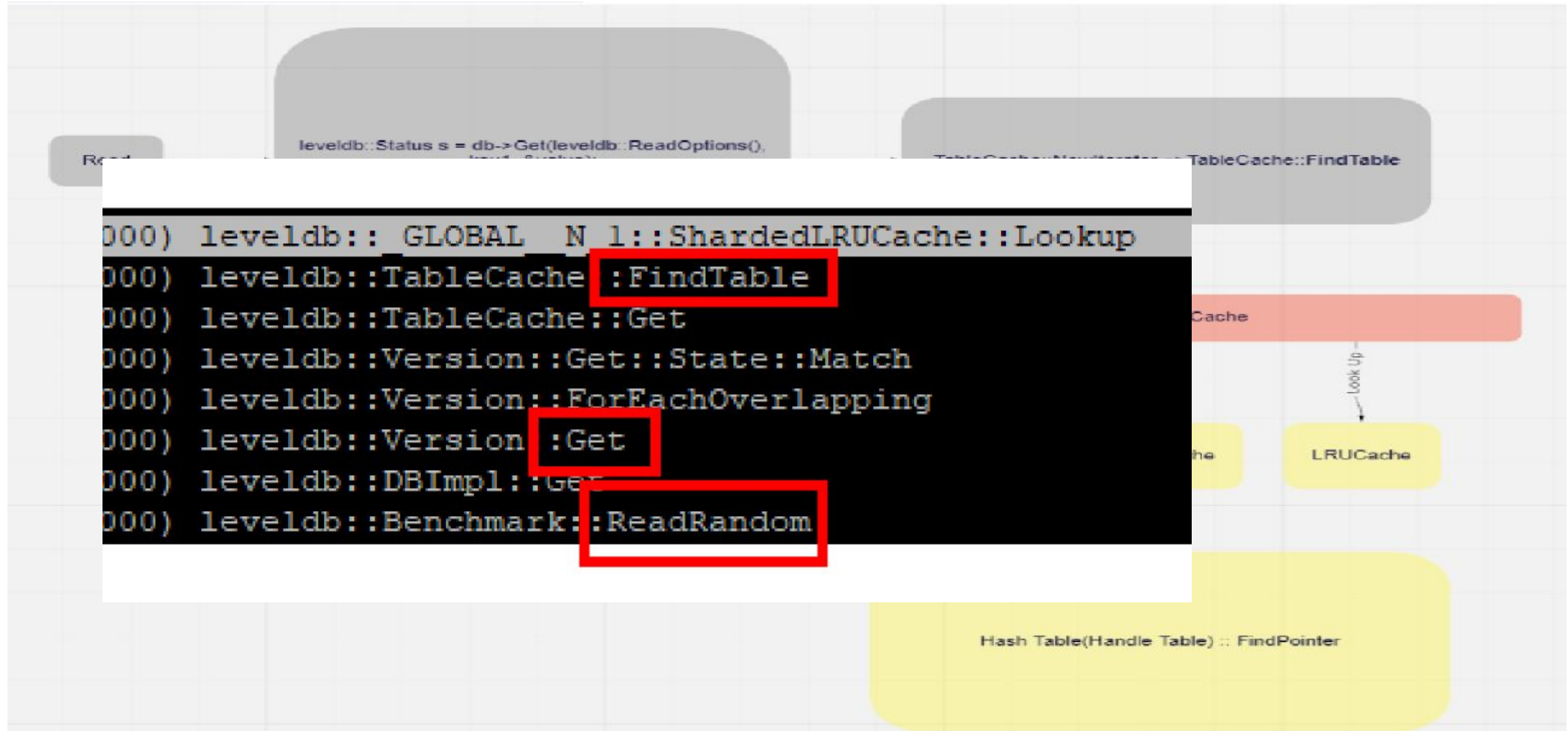
Source code analysis

캐시 조회를 하는 세가지 flow

1. 읽기 데이터 처리
2. sstable 작업
3. BlockReader

Source code analysis

1.읽기 데이터 처리



Source code analysis

1.읽기 데이터 처리

DMImpl::Get(db_impl.cc)

Memtable, immutable memtable, sstable을 찾음

백그라운드 압축 작업을 시작하는지에 대한 여부를 결정함

Memtable, immutable memtable에 없을 경우 => 디스크 파일 검색(sstable)에 중점을 둠

=>Version::Get(version_set.cc)

먼저 키가 존재할 수 있는 sstable을 찾은 다음

table_cache에서 키에 해당하는 값을 찾음

=>TableCache::Get(tabel_cache.cc)

table_cache에서 table구조를 가져옴

그렇지 않은 경우 새 table구조를 만들고 table_cache에 join한 다음

Table::Get(table.cc)을 호출하여 특정 sstable을 찾음

Source code analysis

2.sstable 작업(빌드)

```
~T~\ ~T~@(1) leveldb::_GLOBAL__N_1::ShardedLRUCache::Lookup
~T~B (1) leveldb::TableCache::FindTable
~T~B (1) leveldb::TableCache::NewIterator
~T~B (1) leveldb::BuildTable ←
~T~B (1) leveldb::DBImpl::WriteLevel0Table
~T~B (1) leveldb::DBImpl::RecoverLogFile
~T~B (1) leveldb::DBImpl::Recover
~T~B (1) leveldb::DB::Open
~T~B (1) leveldb::Benchmark::Open
~T~B (1) leveldb::Benchmark::Run
~T~B (1) main
```

Source code analysis

2.sstable 작업(빌드)

Status BuildTable(builder.cc)

```
// verify that the table is usable  
Iterator* it = table_cache->NewIterator(ReadOptions(), meta->number,  
                                         meta->file_size);
```

iter를 사용하여 TableBuilder에 키/값 쌍을 추가한 다음 파일을 쓰고 동기화

나중에 사용할 수 있도록 BuildTable을 통해 새로 생성된 테이블 구조를 table_cache에 추가함
(이를 위해 table_cache.cc의 **NewIterator**사용)



Source code analysis

```
~T~T ~T~@(1000) leveldb::_GLOBAL__N_1::ShardedLRUCache::Lookup
(1000) leveldb::Table::BlockReader ←
(1000) leveldb::Table::InternalGet
(1000) leveldb::TableCache::Get
(1000) leveldb::Version::Get::State::Match
(1000) leveldb::Version::ForEachOverlapping
(1000) leveldb::Version::Get
(1000) leveldb::DBImpl::Get
(1000) leveldb::Benchmark::ReadRandom
(1000) leveldb::Benchmark::ThreadBody
```


Reference

[\[LevelDB\] 스토리지 6: 적자 생존 - 캐시 - 지식 \(zhihu.com\)](#)

<https://karatos.com/art?id=aa18f628-0000-4dee-9f4a-2ea8ea5be71b>

[AC 키: LSM 기반 키-값 저장소 |를 위한 적응형 캐싱 유세닉스 \(usenix.org\)](#)