

Week#7 RocksDB Compaction

Sangeun Chae
2018314760

1. INTRODUCTION

RocksDB 는 Facebook 에서 시작된 오픈소스 데이터베이스 개발 프로젝트로, 서버 워크로드와 같은 대용량 데이터처리에 적합하고 빠른 저장장치, 특히 플래시 저장장치(SSD)에서 높은 성능을 내도록 최적화되어 있다. RocksDB 에는 크게 3 가지 핵심 요소가 있다. Memtable, SST files, WAL. 이번 랩에서는 SST files 간에 일어나는 compaction 과정을 다뤄보고자 한다. Memtable 은 Memory 내부에 전채하며 disk 의 SST file 로 Flush 되기 전에 데이터를 저장하는 데이터 구조이다. 쓰기 요청이 들어오면 반드시 memtable 에 우선적으로 쓰이며, 읽기 또한 RocksDB 의 특성상 가장 최신의 데이터부터 찾기 시작하므로 memtable 부터 확인한다. 만약 memtable 이 특정 threshold 크기 이상이 되면, 데이터는 변경이 불가능한 읽기 전용 (Immutable memtable)로 flush 됩니다. 만약 읽기 전용 memtable 도 가득 차면, disk 로 flush 가 된다. Memtable 이 flush 되게 되면, 가장 먼저 level0 의 SST file 로 가게 된다. SST file 은 Key 의 순서에 따라 정렬이 되어 있는데, Level 0 는 각 SST file 마다 Key 값이 겹칠 수 있다. 만약 Level 0 의 크기가 특정 threshold 를 넘어가게 되면, 2 가지 compaction policy 를 선택하여 compaction 을 진행할 수 있다. 이번 랩에서는, 두가지 compaction policy 에 대해서 다뤄볼 예정이다.

2. METHODS

RocksDB DB_bench 를 실행시킬 때, compaction style 을 leveled compaction(compaction style:0), universal compaction(compaction style: 1) 로 parameter 를 변경하면서 번갈아 실행한다. 각각의 DB_bench 를 실행 후, LOG 파일을 상위 디렉토리에 복사를 하고(복사를 하지 않으면, log 가 append 되기 때문에 기존의 log 파일이 삭제된다), 2 가지의 bench 가 끝나면 두개의 log 파일을 비교한다

3. Performance Evaluation

3.1 Experimental Setup

Type	Specification
OS	Ubuntu 20.04.3 LTS
CPU	AMD Ryzen 7 5800X 8-Core Processor (VMware support 4 Core)
Memory	4GB
Kernel	Linux ubuntu 5.11.0.34 -generic
Disk	VMware Virtual 80GB

Table 1: System setup

Type	Configuration
Bench Type	"readrandomwriterandom"
Direct flush_compaction	True
Direct read	True
Duration	7200s

Table 2: Benchmark setup

3.2 Experimental Results

우선, compaction policy 를 leveled compaction policy 를 사용했을 때의 결과이다.

** Compaction Stats [default] **				
Level	Files	Size	Score	Read(GB)
L0	3/0	2.73 MB	0.8	0.0
L1	1/0	13.11 MB	0.8	1.2
L2	1/0	61.23 MB	0.4	1.1
Sum	5/0	77.07 MB	0.0	2.3
Int	0/0	0.00 KB	0.0	0.0

Figure 1: Leveled Compaction [1]

Rn(GB)	Rnp1(GB)	Write(GB)	Wnew(GB)
0.0	0.0	0.4	0.4
0.4	0.8	1.2	0.3
0.3	0.8	0.9	0.0
0.7	1.7	2.4	0.7
0.0	0.0	0.0	0.0

Figure 2: Levelled Compaction [2]

Moved(GB)	W-Amp	Rd(MB/s)	Wr(MB/s)	Comp(sec)
0.0	1.0	0.0	13.5	28.77
0.0	3.1	42.5	40.6	29.37
0.0	2.9	51.9	40.2	21.75
0.0	6.3	29.7	30.7	79.89
0.0	1.0	0.0	64.3	0.01

Figure 3: Levelled Compaction [3]

CompMergeCPU(sec)	Comp(cnt)	Avg(sec)	KeyIn	KeyDrop	Rblob(GB)	Wblob(GB)
4.01	427	0.067	0	0	0.0	0.0
16.73	106	0.277	19M	844K	0.0	0.0
12.60	16	1.360	18M	3856K	0.0	0.0
33.34	549	0.146	37M	4701K	0.0	0.0
0.01	1	0.014	0	0	0.0	0.0

Figure 4: Levelled Compaction [4]

Levelled compaction 은 L0 의 파일 수가 특정 threshold 에 도달했을 때 발생하고, L0 의 파일들은 L1 에 병합(Merge)된다. L0 의 파일들은 파일별로 키가 overlapping 되기 때문에, 모든 L0 에 존재하는 파일들은 compaction 에 참여하게 된다. 따라서 L0 의 파일들을 L1 과 merge 하고 나면, L1 level 에서도 threshold 를 초과하게 되는 상황이 올 수 있다. 이러한 경우에는, L0 level 과 다르게 특정 파일만 L2 level 에 compaction 되는 과정을 거치게 된다. 따라서 level 이 증가할수록, 오래된 데이터가 축적되는 형태가 만들어 지게 된다.

다음은, compaction policy 를 universal compaction policy (tiered compaction policy)를 사용했을 때의 결과이다.

** Compaction Stats [default] **				
Level	Files	Size	Score	Read(GB)
L0	0/0	0.00 KB	0.0	0.0
L3	1/0	5.26 MB	0.0	1.3
L4	1/0	13.12 MB	0.0	0.2
L5	1/0	38.43 MB	0.0	0.3
L6	1/0	59.51 MB	0.0	0.3
Sum	4/0	116.32 MB	0.0	2.1
Int	0/0	0.00 KB	0.0	0.0

Figure 5: Universal Compactions [1]

Rn(GB)	Rnp1(GB)	Write(GB)	Wnew(GB)
0.0	0.0	0.3	0.3
0.2	1.1	1.3	0.2
0.1	0.1	0.2	0.1
0.2	0.1	0.2	0.1
0.2	0.1	0.2	0.1
0.7	1.4	2.2	0.8
0.0	0.0	0.0	0.0

Figure 6: Universal Compactions [2]

Moved(GB)	W-Amp	Rd(MB/s)	Wr(MB/s)	Comp(sec)
0.0	1.0	0.0	61.8	4.88
0.0	6.1	81.7	80.7	15.87
0.0	1.7	64.7	62.3	3.81
0.0	1.5	73.1	63.7	3.80
0.0	0.9	117.0	68.2	2.56
0.0	7.4	68.6	72.3	30.92
0.0	6.3	77.1	90.5	0.13

Figure 7: Universal Compactions [3]

CompMergeCPU(sec)	Comp(cnt)	Avg(sec)	KeyIn	KeyDrop	Rblob(GB)	Wblob(GB)
3.39	332	0.015	0	0	0.0	0.0
13.79	201	0.079	19M	237K	0.0	0.0
2.65	49	0.078	3775K	140K	0.0	0.0
2.90	20	0.190	4254K	554K	0.0	0.0
2.52	6	0.426	4685K	1849K	0.0	0.0
25.24	608	0.051	32M	2783K	0.0	0.0
0.12	4	0.032	149K	1728	0.0	0.0

Figure 8: Universal Compactions [4]

Universal compaction 은 Levelled compaction 과 다르게, 파일 수가 threshold 에 도달했을 때 (space threshold 가 도달했을 때) compaction 되는 것은 동일하지만, overlapping 되는 키가 각각의 level 별로 존재한다. 즉, 시간을 기점으로 compaction 을 하기 때문에, sorted run 에는 overlapping 되는 키가 존재하지 않지만, level 별로는 존재하게 된다.

4. Conclusion

Levelled compaction 을 수행하게 되면, 사용되는 space 의 크기는 overlapping 되는 key 가 없기 때문에, 많은 storage 를 사용하지 않게 된다. 또한 read 하는 과정에 있어서, overlapping 되는 키가 없기 때문에, 각각의 level 에서 특정 key 가 존재하는 파일만 search 하면 되기 때문에, read 에 소요되는 overhead 가 줄어든다. 하지만, compaction 과정에 있어서 같은 key 가 여러 번 write 될 수 있기 때문에, write amplification 이 발생하게 되고, 그로 인해 write stall 이 발생하여 write performance 의 저하를 가져온다. 반면, universal compaction 은 level 별로 overlapping 되는 key 가 존재하기 때문에, 전반적인 space 는 늘어나게 된다. 이 과정에서 space amplification 이 발생하게 된다. 또한 read query 를 수행하는 것에 있어서도, overlapping 된 key 를 모두

search 해야하기 때문에, read performance 의 저하를 가져온다. 하지만 write 에 있어서는, write amplification 이 leveled compaction 에 비해 작기 때문에, write performance 는 좋아짐을 알 수 있다. 따라서 heavy write workload 환경에서는 leveled compaction 보다 universal compaction 을 수행하는 것이 performance 과점에서는 유리하다.

5. REFERENCES

- [1] <https://meeeejin.gitbooks.io/rocksdb-wiki-kr/content/leveled-compaction.html>
- [2] <https://github.com/facebook/rocksdb/wiki/Universal-Compaction>