CAS CS 561: Data Systems Architectures
Data-intensive Systems and Computing Lab
Department of Computer Science
College of Arts and Sciences, Boston University
http://bu-disc.github.io/CS561/

CS561 Spring 2025 – Project 1

Title: *Analyzing the impact of LSM-buffer implementation on performance*

This programming assignment is for groups of two or three. If there is a strong reason you wish to work on it alone, please reach out to the teaching staff and explain why.

This programming assignment is also recommended to be completed on Azure.

Introduction

A common question in production LSM-engines is how to configure some key tuning knobs in a database system for specific workloads. A good data engineer can provide such answers through experience, benchmarking, and intuition. The goal of this task is to start building these skills, starting with benchmarking.

In this assignment, we focus on one of the most common LSM-based key-value stores, RocksDB. We will compare the performance of RocksDB with different settings of the memory buffer and for specific workloads.

Instruction

Please prepare a 50-minute time slot to go through this instruction manual. We strongly encourage you to use SCC for this project. You can follow the same SCC guide as in project 0 to get started.

1. Download and Compile RocksDB

(1) **If you are using SCC please skip to point (2), otherwise** prepare your Linux environment with:

"`sudo apt-get upgrade && sudo apt-get update`"

"`sudo apt-get install make g++`"

"`sudo apt-get install libsnappy-dev libgflags-dev zlib1g-dev libbz2-dev liblz4-dev libzstd-dev`"

For the following steps, it is highly recommended that you can work in a tmux terminal in case any network issues stop your running programs. Just run "`tmux`" or "`tmux attach –t [session_id]`".

(2) We use one of the latest versions (v 8.9.1) of RocksDB. Navigate to your working directory for project1 (e.g., "`mkdir ~/project1/ && cd ~/project1/`") and run

"`wget https://github.com/facebook/rocksdb/archive/refs/tags/v8.9.1.tar.gz`".

This will download a compressed tarball "v8.9.1.tar.gz". Decompress it with "`tar -xvzf v8.9.1tar.gz`" and you will get a new directory named "rocksdb-8.9.1".

(3) Go into this directory and compile its static library:

"`cd rocksdb-8.9.1 && make –j2 static_lib`".

If you have 2 vCPUs in your virtual machine, compiling the static library may take around 15 minutes, assuming no other resources are consuming vCPUs. If there is only one vCPU in your machine, this may take at least half an hour to compile. (You may have a similar problem when you run the workload generator.)

You will have a new file (≈1.2GB) called "librocksdb.a", which serves as the static library of RocksDB.

Note: **NO NEED TO COMPILE THE DEBUG VERSION UNLESS YOU NEED IT**. If you really want to debug your own program later, you can run "`make –j2 dbg`" to generate a debug version of RocksDB so that you can have full control when debugging things. Compiling the debug version of RocksDB is very ***time-consuming***, you can skip this step if you do not require full control of the codes when debugging.

2. Get Started with Sample Code

(1) Assuming that we just successfully compiled the static library under the directory "rocksdb-8.9.1", now we go into directory "examples/" and compile the "simple_example" file with "`cd examples/ && make simple_example`".

(2) Simply type "`./simple_example`" in your terminal to execute it and you will get nothing printed in the terminal. However, now if you go to directory "/tmp/" and you will find a directory called "rocksdb_simple_example" is created. The newly created directory is actually the database home directory, which was just populated from "simple_example".

You will find many necessary files for RocksDB under this directory (e.g., MANIFEST, LOCK, LOG).

(3) Read the code from "simple_example.cc" and get a high-level understanding of how we create a database with specific path, how we call a basic version of "get()", "put()". (**DON'T** be panic if you cannot understand the code, since you are not required to understand everything. You might be confused about "`IncreaseParallelism(), OptimizeLevelStyleCompaction(), WriteBatch, PinnableSlice`", just skip these when you read the example code).

3. Workload Generator

(1) Go into your working directory for project 1 and run

"`git clone https://github.com/BU-DiSC/K-V-Workload-Generator.git`".

(2) Compile the workload generator with "`cd K-V-Workload-Generator && make`" and you will get an executable file "./load_gen" that can produce much more complicated workload than what we have in project0. For example, run

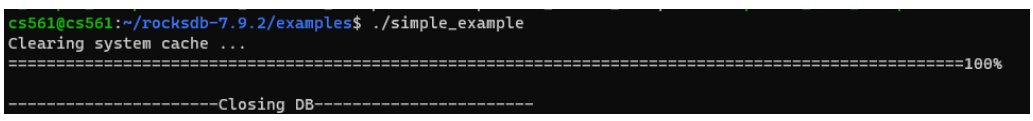
"`./load_gen -I 100000 -Q 3000 -S 1000 -Y 0.3`".

Where "I" specifies the number of inserts, "Q" specifies the number of point queries, "S" specifies the number of range queries with selectivity "Y" (Note that the configuration characters differ from what we have in project 0). After you run the above command, you will get a "workload.txt" mixed with these operations.

The workload file will have 100000+3000+1000=104000 lines. Every line in this workload file starts with a character that specifies the operation type, and then the parameters for the operation. For example, "I 801627617 woQf" means that inserts a key-value pair (801627617, "woQf"), and "S 2992416503 4294786680" means a range query between key 2992416503 and 4294786680.

(3) To simplify your implementation, we provide another example file "simple_example.cc" that can parse the workload file we just produced. (See https://bu-disc.github.io/CS561/projects/simple_example.cc). The workflow is as follows:

a) Rename the origin "simple_example.cc" as "simple_example_old.cc". Move the new one (provided by us) to the same path as the old one.
b) Compile the new "simple_example.cc" by "`make simple_example`" under the "examples/" folder.
c) Move the workload file ("workload.txt", that we just created using "./load_gen") under the same directory with "simple_example.cc" (i.e., "rocksdb-8.9.1/examples/").
d) Run "`simple_example`" under the "examples/" directory and you will get the following messages (omitting the path info since you may have different path):



e) Check if the database directory "/tmp/cs561_project1/" has been created. If the path exists and it has similar files to "/tmp/rocksdb_simple_example/", you are good now.

Experiment

As you will be required to benchmark the system performance under different settings, you will have to know how to change the configuration in the example code. In most cases, you will play with the variable "`Options& op`". For example, if you want to set 32MB as the write buffer size (essentially L0 size in the Log-Structured Merge Tree) in RocksDB, you can write "op.write_buffer_size= 33,554,432" (32*1024*1024 Bytes) where the default value of write buffer size is 64MB and put this line of code before we pass "`op`" to the "`Open(op,...,...)`" function.

In the submission document, you need to present their performance (either the latency for each type of operation or the overall latency to execute the workload).

1. Configure the system with the following specifications (examples are provided):
    a. Set the write buffer size as 2MB.
    b. Try **four** different types of memTable implementation and compare the operation latency (examples are provided). You can check the available implementation from the file "rocksdb-8.9.1/include/convenience.h" (lines 201 - 229). The example of specifying the memTable is in line 790 of "rocksdb-8.9.1/include/advanced_options.h". A more detailed declaration can be found in "rocksdb-8.9.1/include/memtablerep.h". Note that if you choose vector as the memTable factory, you need to add "`op.allow_concurrent_memtable_write = false;`" before you call "`Open`" function. Every time you make some changes in your code to vary the memTable implementation, remember to re-compile "simple_example.cc" by "`make simple_example`" under the "examples/" folder.

2. Generate the following 3 types of workloads (the entry size should be set as 64B) and store the workload files somewhere safe with distinct names. For each workload, copy it under the "examples/" folder (rename the workload file by "workload.txt") and test the system performance with varying the memory buffer implementation as mentioned above (4 experiments per workload, each experiment specifies one type of memTable implementation):

    (a) Insert-only workload: 1M inserts
    (b) Mixed workload (insert and point queries): 900K inserts, 100K point queries
    (c) Mixed sequential workload (insert and point queries): 900K inserts, 100K point queries

Note: To generate sequential workload (inserts and then queries), you make the following changes in "load_gen.cc", re-compile "load_gen" and re-generate the workload using "load_gen":

"#define PQ_THRESHOLD 0.1" → "#define PQ_THRESHOLD 1.0"

3. Summarize the above experiments (with figures) and explain how the implementation of the memory buffer affects the system performance.

Question

1. How do you think a change (increase or decrease) in the memory buffer size would affect the workload execution latency? You may want to run some experiments to quantify this; otherwise, you can provide your intuition to explain your answer.

Resources

RocksDB: official website , github,

Submission instructions

Please submit a single PDF file in Gradescope. Hand-written reports are NOT allowed! **Make sure that you submit as a group! In this file include your full names and BU IDs. There should be only one submission per group.**