存储技术基础 2020 Spring

Key-Value 存储引擎

作业描述

本作业要求在给定 C++ 代码框架下，实现高效的并发 Key-Value 存储引擎，支持 KV 基本的 Read (Get)、Write (Put) 和 Range (Scan)（可选）操作。

作业要求

1. **一人一组**完成代码实现工作。
2. 基于课程提供的代码框架进行程序开发（修改 engine_race.{cc|h} 文件），实现 Read、Write 接口。
3. 要求实现**多线程**并发正确的版本，保证线性一致性（Linearizability)：即并发读写时，写请求一旦返回，其更改需要体现在接下来所有的读请求中。
4. 要求 Key-Value 引擎保证崩溃一致性（**Crash Consistency)：**Write 成功返回之后，保证该操作插入的键值对被持久化，即使机器崩溃重启后也不会丢失。
5. 通过课程提供的正确性和性能测试程序（bench 目录下）。
6. 要求提交**实验代码**和**实验报告。**报告内容包括 KV 设计和实现细节，性能结果及分析，思考题。
7. 提交截止时间第 7 周周日晚 24：00 之前。

附加题（可选）：

实现范围查找接口 Range.
RetCode Range(const PolarString& lower, const PolarString& upper, Visitor &visitor)

思考题（任选一）：

1. 如何保证和验证 Key Value 存储引擎的 Crash Consistency？考虑如下 Crash 情况：（1）KV 崩溃（进程崩溃）;（2）操作系统崩溃；（3）机器掉电。
2. 如何在实现高性能 KV 的同时，提高有效 IO 利用率，降低内存使用率？目前 KV 对外存读写数据的方式有以下几种：（1）系统调用 read、write、mmap、fsync；（2）异步 IO 框架 libaio, io_uring 和 SPDK。以上方式对 KV 的整体吞吐，IO 利用率和内存使用率有何差异？

作业详细说明

1. 需实现的接口有:

// name 为存储引擎的数据路径, 初始化存储引擎， 返回指针到*eptr
RetCode EngineRace::Open(const std::string& name, Engine** eptr);

// 将<key, value>插入存储引擎， 如果 key 已经存在， 则该操作为更新
RetCode EngineRace::Write(const PolarString& key, const PolarString& value)

//根据 key 在存储引擎中索引数据， 返回数据到*value 变量 RetCode EngineRace::Read(const PolarString& key, std::string* value)

其中 PolarString 是一个封装的字符串类，详见 include/polar_string.h。接口定义在 include/engine.h 和 engine_race/engine_race.{cc|h}。更详细的接口语义可阅读 test/single_thread_test.cc。

2. KV 框架在压缩代码包 engine.zip 中，可从网络学堂下载。编译运行环境为 Linux，大家自行搭建虚拟机、docker 或 Windows Subsystem for Linux 环境。
3. engine.zip 解压后有如下文件: 其中 engine_example 里是一个参考版

本的 KV 存储引擎。engine_race 中包含 engine_race.{cc|h}, 这是本次作业中唯一需要修改的两个文件。test 中包含正确性测试代码，bench 中包含性能测试代码。

4. 编译时，执行 make 命令（如果需要编译 engine_example 中的参考代码，执行 make TARGET_ENGINE=engine_example)。编译完成后在 lib 目录下生成静态链接库 libengine.a。
5. 测试正确性时， 进入 test 目录， 执行./build.sh 来编译测试程序， 执行./run_test.sh 来 运行测试程序。现提供的三个测试程序比较简单，single_thread_test 测试单线程正确性； multi_thread_test 测试多线程情况下的正确性；crash_test 测试进程被 kill 后的系统正确 性。
6. bench 目录中提供了性能测试程序，执行./build.sh 来编译，执行./bench 来运行测试程序。./bench 程序有三个参数, thread_num 是并发执行的线程个数，read_ratio 是 Read 操作的比例， isSkew 代表 key 的分布 (0 时为均匀分布，1 时为 zipfan 分布)。bench 程序中 key 和 value 的大小分别固定为 8bytes 和 4096bytes。

参考文献

[1] https://github.com/google/leveldb. (基于 LSM Tree 的 KV 存储引擎)

[2] https://github.com/facebook/rocksdb. (基于 LSM Tree 的 KV 存储引擎)

[3] https://fallabs.com/kyotocabinet/. (基于 B+Tree 或 Hashtable 的 KV 存储引擎)

[4] Lu, Lanyue, et al. “WiscKey: separating keys from values in SSD-conscious storage.” Proceedings of the 14th Usenix Conference on File and Storage Technologies. USENIX Association, 2016.

[5] Raju, Pandian, et al. “Pebblesdb: Building key-value stores using fragmented log-structured merge trees.” Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017.

[6] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP ’ 19).

[7] https://spdk.io/ (SPDK)
[8] http://man7.org/linux/man-pages/man2/io_submit.2.html (Linux libaio)
[9] https://lwn.net/Articles/776703/ (Linux io_uring)