1. 比赛攻略 (Rapids团队 - C++实现)

1.1. 赛题介绍

评测程序分为2个阶段：

1）Recover正确性评测： 此阶段评测程序会并发写入特定数据（key 8B、value 4KB） 同时进行任意次kill -9来模拟进程意外退出（参赛引擎需要保证进程意外退出时数据持久化不丢失），接着重新打开DB，调用Read、Range接口来进行正确性校验。

2）性能评测

- 随机写入：64个线程并发随机写入，每个线程使用Write各写100万次随机数据（key 8B、value 4KB）
- 随机读取：64个线程并发随机读取，每个线程各使用Read读取100万次随机数据
- 顺序读取：64个线程并发顺序读取，每个线程各使用Range有序（增序）遍历全量数据2次

注： 2.2阶段会对所有读取的kv校验是否匹配，如不通过则终止，评测失败； 2.3阶段除了对迭代出来每条的kv校验是否匹配外，还会额外校验是否严格递增，如不通过则终止，评测失败。

1.2. 最终线上效果

- 进程耗时

子阶段	进程最佳占用时间	阶段IO吞吐量波动范围
随机写入	114.1 seconds左右	`2176.27-2195.34` MB/s
随机读取	105.9 seconds左右 (包括0.2 seconds 索引构建)	`2287.23-2291.56` MB/s
顺序读取	192.1 seconds左右 (包括0.2 seconds 索引构建)	`2596.04-2601.45` MB/s

顺序读取中，在内存并发读取visit总时间大概在`105`-`110`秒，吞吐量: `284.1`-`296.2` GB/s 。

- 进程启动间隔(波动不大，取其中一次作为样本)

阶段之间间隔	耗时
随机写入启动的间隔	`0.1` seconds 左右
随机写入到读取的间隔	`0.35` seconds 左右
随机读取到顺序读取的间隔	`0.45` seconds 左右

- 最优成绩

最优成绩	总时间
理论历史最佳成绩	`114.1+105.9+192.1+0.1+0.35+0.45=413.00 seconds`
历史最佳成绩	`413.69 seconds`

- 历史最佳成绩离理论历史最佳成绩累加差了`0.69 seconds`； 这应该是与机器读写性能波动有关，其中写性能波动最大。

2. 赛题背景分析及理解

2.1. 赛题注意点

- Recover正确性评测要求: 每次Write调用都要将数据至少写入到page-cache。
- 每个阶段开始，DB Engine都会被重新打开； 每个阶段进行中， 只有读取或者写入中的一种情况发生； 每一阶段结束，page cache会被清空(清空page cache的时间占用总时间)。
- 对于复赛的顺序读取，recovery阶段使用单线程，并且只遍历全量数据1次，选手需要做相应处理。
- 评测中， Key-Value大小都是固定的， Key可用64bit无符号整数表示，并且分布均匀。
- 评测中，允许使用的最大内存`2GB` (不计评测程序内存占用)；各子阶段会有固定的64线程随机写入或随机读取或顺序读取。

2.2. 赛题分析

- 傲腾存储特征
    - 为达到磁盘的峰值吞吐量， 需要保证`iostat`中的两个关键参数`avgrq-sz`(扇区个数，一般每个扇区`512B`)和`avgqu-sz`(IO队列长度) 处于合适大小。 并且每个request有最大大小，不能设置太大，具体可以查看`blockdev --getmaxsect /dev/sda`(`/dev/sda`为查看的设备)。
    - 随机读写只要有合适大小块(`avgrq-sz`)，保证足够IO队列长度(`avgqu-sz`)就可以达到接近顺序读写的效果。
    - 顺序读取通过`128KB`大小请求,`QD=8`可以达到`2595-2605 MB/s`左右。
    - 随机读取`4KB`大小请求, `QD=20+`可以达到`2290 MB/s`左右，波动很小。
    - 随机写入`16KB`大小请求, `QD=20+`可以达到`2180-2200 MB/s`。
- 注意点
    - 随机写入：保证`iostat`中的合适大小的`avgrq-sz`(通过实验测得 mmap buffer 16KB比较优)， 保证`avgqu-sz`(handle tail threads)，至少需要`QD=8`(实验中测得8,16,32都差不多)，通过写文件时候系统的同步(锁)。
    - 随机读取：保证`iostat`中的合适大小的`avgqu-sz`(handle tail threads)。
    - 顺序读取：保证`iostat`中的合适`avgrq-sz`和`avgqu-sz`。做好充分的overlap-io和内存访问(100-110秒左右)。
    - 三阶段是独立进行的(无混合的读写操作)，因此评测不要求索引支持`O(logn)`或更低复杂度的动态插入。

3. 核心思路

每一阶段结束，page cache会被清空。 因此，整体设计中除了内存映射文件(meta-file, key/val buffer files), 其他文件操作都通过DirectIO。

3.1. 存储和索引设计

- 文件设计
    - Key和Value都使用write-ahead-log方式，顺序append到相应位置，并通过一个meta文件记录相应记录个数。
    - Value大小**固定**为`4KB`，因此在设计索引时候可以不存Value长度， 并且可以将偏移量直接除以`4KB`来减少空间占用。

    - Key大小**固定**为`8B`， 根据`PolarString`定义, Key可以被转化为64bit-Big-Endian无符号整数表示，Key分布比较随机，因此可按Key分Bucket来设计存储结构, 来支持并发恢复索引。
    - 可以将Key-Value在写入时候一一对应，这样就不用写偏移量，而通过顺序遍历write-ahead-log恢复出来。

- 索引设计

    - 评测中不要求索引支持`O(logn)`或更低复杂度的动态插入，因此采用 bucket + sorted array 的方案；按照Key的Big-Endian无符号整数高位分bucket，每个Bucekt内采用根据Bucket内偏移比较的sorted array； 通过计算bucket id 和 branchless-lower-bound加跳过重复支持value in-bucket offset查询。
    - 并行索引构建(0.2秒，`throughput = 488.28125MB/0.2s = 2441.4 MB/s`)，每个线程分配一些buckets， 因为每个bucket的大小均匀所以workload balanced，多bucket使得sort时间可以忽略，此外sort还和io overlap在一起。

- 写入相关的文件设计

    - 写入buffer必须使用文件作为backend的(通过mmap)，来保证正确性。
    - 考虑到range时候顺序读盘更快，写入时候同一bucket写在同一区域。

3.2. 文件设计

- key/value write-ahead logs 各32个(一个文件里面分bucket, bucket id相邻的在文件中相邻)。
- meta-file, mmap key buffer file, mmap value buffer file各1个, slice成`BUCKET_NUM`的views (e.g. 1024)。

文件整体设计分为三部分: (1) K-V-Log文件, (2) Meta-Count文件, (3) Key/Value Buffer文件。

3.2.1. K-V Log文件

- key-value的对应: 逻辑上, key-value被写到一个的相同bucket, 对应到相同的in-bucket offset, 通过write-ahead追加到对应的log文件。 我们把`8-byte-key`通过big-endian转化出`uint64_t`类型的整数`key`。

对应从`key`到`bucket_id`的计算如下代码所示:

```
inline uint32_t get_par_bucket_id(uint64_t key) {
    return static_cast<uint32_t >((key >> (64 - BUCKET_DIGITS)) & 0xffffffu);
}
```

- 逻辑上的bucket到实际中的文件, 通过下面的函数算出, 在设计中, 我们让相邻的value buckets被group到同一个value log file, 来为range查询顺序读服务。

```
inline pair<uint32_t, uint64_t> get_key_fid_foff(uint32_t bucket_id, uint32_t
bucket_off) {
    constexpr uint32_t BUCKET_NUM_PER_FILE = (BUCKET_NUM / KEY_FILE_NUM);
    uint32_t fid = bucket_id / BUCKET_NUM_PER_FILE;
    uint64_t foff = MAX_KEY_BUCKET_SIZE * (bucket_id % BUCKET_NUM_PER_FILE) +
bucket_off;
    return make_pair(fid, foff * sizeof(uint64_t));
}

inline pair<uint32_t, uint64_t> get_value_fid_foff(uint32_t bucket_id,
uint32_t bucket_off) {
    // Buckets 0,1,2,3... grouped together.
```

```
    constexpr uint32_t BUCKET_NUM_PER_FILE = (BUCKET_NUM / VAL_FILE_NUM);
    uint32_t fid = bucket_id / BUCKET_NUM_PER_FILE;
    uint64_t foff = MAX_VAL_BUCKET_SIZE * (bucket_id % BUCKET_NUM_PER_FILE) +
bucket_off;
    return make_pair(fid, foff * VALUE_SIZE);
}
```

- 最终设计中, 我们采用了32个value文件和32个key文件。 这是因为多线程写入同一文件时候可以有一定的同步作用(有写入锁的存在), 来缓解最后剩下tail threads打不满IO的情况。 此外，文件过多容易触发Linux操作系统的bug，从DirectIO进入BufferIO, 即使已经标志flag设置了`O_DIRECT`.

3.2.2. Meta-Count 文件

- meta-count文件用来记录每个bucket中现在write-ahead进行到第几个in-bucket位置了, 该文件通过内存映射的方式, 来通过操作对应数组`mmap_meta_cnt_`， 记录每个bucket写入write-ahead-entry个数。

```
// Meta.
meta_cnt_file_dp_ = open(meta_file_path.c_str(), O_RDWR | O_CREAT,
FILE_PRIVILEGE);
ftruncate(meta_cnt_file_dp_, sizeof(uint32_t) * BUCKET_NUM);
mmap_meta_cnt_ = (uint32_t *) mmap(nullptr, sizeof(uint32_t) * (BUCKET_NUM),
                                                PROT_READ | PROT_WRITE,
MAP_SHARED, meta_cnt_file_dp_, 0);
memset(mmap_meta_cnt_, 0, sizeof(uint32_t) * (BUCKET_NUM));
```

3.2.3. Key/Value Buffer 文件

buffer files用来在写入时候进行对每个bucket-entries的buffer (通过内存映射文件得到aligned buffer, 来具备`kill-9`容错功能). 一个bucket对应相应的key-buffer和value-buffer; 所有的key-buffers从一个key-buffer文件内存映射出来; 同理所有的val-buffers从一个val-buffer文件映射出来。

我们给出一个Value Buffer文件的示例, Key Buffer文件相关的设计与之类似。

```
// Value Buffers. (To be sliced into BUCKET_NUM slices)
value_buffer_file_dp_ = open(value_buffers_file_path.c_str(), O_RDWR |
O_CREAT, FILE_PRIVILEGE);
if (value_buffer_file_dp_ < 0) {
   log_info("valbuf err info: %s", strerror(errno));
   exit(-1);
}
ftruncate(value_buffer_file_dp_, tmp_buffer_value_file_size);
mmap_value_aligned_buffer_ = (char *) mmap(nullptr,
tmp_buffer_value_file_size, \
            PROT_READ | PROT_WRITE, MAP_SHARED, value_buffer_file_dp_, 0);
for (int i = 0; i < BUCKET_NUM; i++) {
    mmap_value_aligned_buffer_view_[i] =
            mmap_value_aligned_buffer_ + VALUE_SIZE * TMP_VALUE_BUFFER_SIZE *
i;
}
```

- 在设计中, 我们使用了`16KB` value buffer 和 `4KB` key buffer, 分别整除`VALUE_SIZE`和`sizeof(uint64_t)`。我们选择较小的buffer是为了让IO尽可能快地均衡地被打出去(不要有很少的线程最后还在打IO以致于打不满), value buffer不选择更小是为了防止sys-cpu过高影响性能，并且我们对磁盘的benchmark显示`16KB`是一个比较优的值。

3.3. 写入阶段思路

- 清空page cache占用总时间，傲腾存储iops和throughput都高， 因此使用DirectIO绕过page cache手动管理缓冲和写盘。
- 写入时候先对bucket加锁，将Key/Value一一对应，分别写入这个bucket对应的key-mmap-buffer和value-mmap-buffer， 在buffer满的时候写入log文件。

3.4. 随机和顺序读取阶段设计思路

- 随机读取
    - 做最小粒度的同步。奇数和偶数线程同步，来保证64线程间较小的读取进度差别和稳定的queue-depth（24左右）。
    - 尝试了读`8KB`，并进行缓存置换，命中率不高：2%；这说明通过读更大block-size来优化随机读取不可行。
- 顺序读取
    - 流水线设计：io部分(io协调者， io线程)，通过set-affinity减少numa结点间的切换；内存读取部分(64threads)同步读一个bucket与下一块bucket的io overlap在一起。
    - 每块buffer为一个读取单位，io协调者通过提交任务给io线程打到合适`avgrq-sz`和`avgqu-sz`，从而打满IO throughput，详见io协调者。
    - 流水线使用2块buffer滚动。
    - 在多次全量遍历中，偶数次次遍历重用奇数次的前几块buffer(块数通过程序中的`KEEP_REUSE_BUFFER_NUM`指定，作为cache)。

3.5. 容错(K/V Buffer Files Flush)的思路

- 大体思路: 我们通过`ParallelFlushTmp`并行flush key, value buffers; 该函数在写入阶段的析构函数调用(如果进行到对应代码), 否则在读取阶段构建index前会调用。
- 优化: 我们通过`ftruncate`对应文件长度为0表示所有buckets对应的需要flush的buffers已经Flush出去了, 避免重复的Flush. 相应逻辑在`FlushTmpFiles`函数中。

4. 关键代码

4.1. 随机写入

4.1.1. 实现逻辑

通过锁一个bucket使得key-value在bucket中**一一对应**， 并且使得bucket的meta-count被正确地更改; 写入之前先写bucket对应buffer, buffer满了之后进行阻塞的`pwrite`系统调用。

大体逻辑如下代码所示:

```
```

```
{
    unique_lock<mutex> lock(bucket_mtx_[bucket_id]);
    // Write value to the value file, with a tmp file as value_buffer.
    uint32_t val_buffer_offset = (mmap_meta_cnt_[bucket_id] %
TMP_VALUE_BUFFER_SIZE) * VALUE_SIZE;
    char *value_buffer = mmap_value_aligned_buffer_view_[bucket_id];
    memcpy(value_buffer + val_buffer_offset, value.data(), VALUE_SIZE);

    // Write value to the value file.
    if ((mmap_meta_cnt_[bucket_id] + 1) % TMP_VALUE_BUFFER_SIZE == 0) {
        uint32_t in_bucket_id = mmap_meta_cnt_[bucket_id] -
(TMP_VALUE_BUFFER_SIZE - 1);
        uint32_t fid;
        uint64_t foff;
        tie(fid, foff) = get_value_fid_foff(bucket_id, in_bucket_id);
        pwrite(value_file_dp_[fid], value_buffer, VALUE_SIZE *
TMP_VALUE_BUFFER_SIZE, foff);
    }

    // Write key to the key file.
    uint32_t key_buffer_offset = (mmap_meta_cnt_[bucket_id] %
TMP_KEY_BUFFER_SIZE);
    uint64_t *key_buffer = mmap_key_aligned_buffer_view_[bucket_id];
    key_buffer[key_buffer_offset] = key_int_big_endian;
    if (((mmap_meta_cnt_[bucket_id] + 1) % TMP_KEY_BUFFER_SIZE) == 0) {
        uint32_t in_bucket_id = (mmap_meta_cnt_[bucket_id] -
(TMP_KEY_BUFFER_SIZE - 1));

        uint32_t fid;
        uint64_t foff;
        tie(fid, foff) = get_key_fid_foff(bucket_id, in_bucket_id);
        pwrite(key_file_dp_[fid], key_buffer, sizeof(uint64_t) *
TMP_KEY_BUFFER_SIZE, foff);
    }

    // Update the meta data.
    mmap_meta_cnt_[bucket_id]++;
}
```

4.1.2. 优化: 调整文件个数为32

调整文件个数为32后， 可以利用不同线程写同一文件时候的阻塞，取得一定程度上的同步效果。 使得QueueDepth在基本所有时刻(包括最终快结束时刻)还处于大于8的水平， 来应对tail threads queue-depth打不高的挑战。

4.2. 并行索引构建设计

- 思路: 对每个Bucket构建SortedArray作为Index。
- 回顾: 文件设计中统一bucket的key-value对应起来了, 那么在构建中key的in-bucket offset和value的in-bucket offset是一样的。

每个worker处理对应的buckets, 逻辑上的buckets可以通过之前讲的K-V Log文件设计对应过去。 在整个数组被填充好了之后可以根据下面这个comparator函数对象进行排序.

```
```

```
[](KeyEntry l, KeyEntry r) {
if (l.key_ == r.key_) {
    return l.value_offset_ > r.value_offset_;
} else {
    return l.key_ < r.key_;
}
```

- 具体逻辑： 每个线程分配到的一个任务分成两部分：1. 读取填充in-bucket-offset, 2. 排序。 1024 buckets被均匀地分到64个线程(key大致均匀地分布到每个bucket)， 构建过程中排序和磁盘IO是overlap在一起的。
- 这个阶段主要时间开销在于读key-logs文件(sort开销可以忽略不计), 总开销大概 `0.2 seconds`左右。
- 详细代码如下：

```
vector <thread> workers(NUM_READ_KEY_THREADS);
for (uint32_t tid = 0; tid < NUM_READ_KEY_THREADS; ++tid) {
    workers[tid] = thread([tid, local_buffers_g, this]() {
        uint64_t *local_buffer = local_buffers_g[tid];
        uint32_t avg = BUCKET_NUM / NUM_READ_KEY_THREADS;
        for (uint32_t bucket_id = tid * avg; bucket_id < (tid + 1) * avg;
bucket_id++) {
            uint32_t entry_count = mmap_meta_cnt_[bucket_id];
            if (entry_count > 0) {
                uint32_t passes = entry_count / KEY_READ_BLOCK_COUNT;
                uint32_t remain_entries_count = entry_count - passes *
KEY_READ_BLOCK_COUNT;
                uint32_t file_offset = 0;

                auto fid_foff = get_key_fid_foff(bucket_id, 0);
                uint32_t key_fid = fid_foff.first;
                size_t read_offset = fid_foff.second;
                for (uint32_t j = 0; j < passes; ++j) {
                    auto ret = pread(key_file_dp_[key_fid], local_buffer,
                                     KEY_READ_BLOCK_COUNT * sizeof(uint64_t),
read_offset);
                    if (ret != KEY_READ_BLOCK_COUNT * sizeof(uint64_t)) {
                        log_info("ret: %d, err: %s", ret, strerror(errno));
                    }
                    for (uint32_t k = 0; k < KEY_READ_BLOCK_COUNT; k++) {
                        index_[bucket_id][file_offset].key_ = local_buffer[k];
                        index_[bucket_id][file_offset].value_offset_ =
file_offset;
                        file_offset++;
                    }
                    read_offset += KEY_READ_BLOCK_COUNT * sizeof(uint64_t);
                }

                if (remain_entries_count != 0) {
                    size_t num_bytes = (remain_entries_count *
sizeof(uint64_t) + FILESYSTEM_BLOCK_SIZE - 1) /
                                       FILESYSTEM_BLOCK_SIZE *
FILESYSTEM_BLOCK_SIZE;
                    auto ret = pread(key_file_dp_[key_fid], local_buffer,
num_bytes, read_offset);
```

```
                    if (ret < static_cast<ssize_t>(remain_entries_count *
sizeof(uint64_t))) {
                        log_info("ret: %d, err: %s, fid:%zu off: %zu", ret,
strerror(errno), key_fid,
                                 read_offset);
                    }
                    for (uint32_t k = 0; k < remain_entries_count; k++) {
                        index_[bucket_id][file_offset].key_ = local_buffer[k];
                        index_[bucket_id][file_offset].value_offset_ =
file_offset;
                        file_offset++;
                    }
                }

                sort(index_[bucket_id], index_[bucket_id] + entry_count, []
(KeyEntry l, KeyEntry r) {
                    if (l.key_ == r.key_) {
                        return l.value_offset_ > r.value_offset_;
                    } else {
                        return l.key_ < r.key_;
                    }
                });
            }
        }
    });
}
```

4.3. 随机读取

4.3.1. 实现逻辑

- 随机读取基本逻辑就是：查询index, 如果是key-not-found就返回; 否则读文件。
- 查询index代码如下，其中主要用了带prefetch优化的二分查找:

```
uint64_t big_endian_key_uint = bswap_64(TO_UINT64(key.data()));

KeyEntry tmp{};
tmp.key_ = big_endian_key_uint;
auto bucket_id = get_par_bucket_id(big_endian_key_uint);

auto it = index_[bucket_id] + branchfree_search(index_[bucket_id],
mmap_meta_cnt_[bucket_id], tmp);
```

- 剩余的key-not-found判断和读value逻辑如下所示:

```
if (it == index_[bucket_id] + mmap_meta_cnt_[bucket_id] || it->key_ !=
big_endian_key_uint) {
    NotifyRandomReader(local_block_offset, tid);
    return kNotFound;
}
```

```
uint32_t fid;
uint64_t foff;
std::tie(fid, foff) = get_value_fid_foff(bucket_id, it->value_offset_);

pread(value_file_dp_[fid], value_buffer, VALUE_SIZE, foff);
NotifyRandomReader(local_block_offset, tid);

value->assign(value_buffer, VALUE_SIZE);
```

4.3.2. 优化：细粒度同步

思路: 我们设计了同步策略来保证足够的queue-depth (25-30之间)的同时， 又使得不同线程可以尽量同时退出, 尽量避免少queue-depth打IO情况的出现。

- 实现细节: 我们引入了4个blocking queues `notify_queues_`来作为 偶数和奇数线程的 当前和下一轮读取的同步通信工具 (`tid%2==0`与`tid%2==1`线程互相通知)。
- 实现细节1 (初始化逻辑): 初始化时候放入偶数线程的blocking queue来让他们启动起来。

```
if (local_block_offset == 0) {
    if (tid == 0) {
        notify_queues_.resize(4);
        for (auto i = 0; i < 4; i++) {
            // Even-0,1  Odd-2,3
                notify_queues_[i] = new
moodycamel::BlockingConcurrentQueue<int32_t>(NUM_THREADS);
        }
        for (uint32_t i = 0; i < NUM_THREADS / 2; i++) {
                notify_queues_[0]->enqueue(1);
            }
    }
    read_barrier_.Wait();
}
```

- 实现细节2 (等待逻辑): 每一round的开始的时候会有一个等待。

```
uint32_t current_round = local_block_offset - 1;
if ((current_round % SHRINK_SYNC_FACTOR) == 0) {
    uint32_t notify_big_round_idx = get_notify_big_round(current_round);
    if (tid % 2 == 0) {
        notify_queues_[notify_big_round_idx % 2]->wait_dequeue(tmp_val);
    } else {
        notify_queues_[notify_big_round_idx % 2 + 2]->wait_dequeue(tmp_val);
    }
}
```

- 实现细节3 (通知逻辑): 偶数线程通知奇数线程当前round, 奇数线程通知偶数线程下一round。

```
void EngineRace::NotifyRandomReader(uint32_t local_block_offset, int64_t tid)
{
```

```
    uint32_t current_round = local_block_offset - 1;
    if ((current_round % SHRINK_SYNC_FACTOR) == SHRINK_SYNC_FACTOR - 1) {
        uint32_t notify_big_round_idx = get_notify_big_round(current_round);
        if (tid % 2 == 0) {
            notify_queues_[(notify_big_round_idx) % 2 + 2]->enqueue(1);   //
Notify This Round
        } else {
            notify_queues_[(notify_big_round_idx + 1) % 2]->enqueue(1);   //
Notify Next Round
        }
    }
}
```

4.4. 顺序读取和并发全量遍历

4.4.1. 内存访问和磁盘IO流水线设计

- 主体逻辑: 单个IO协调线程一直发任务让IO线程打IO, 其他线程消费内存, 每进行一个bucket进行一次barrier来防止visit内存占用太多资源。
- 实现细节1 (IO协调线程通知memory visit 线程 value buffer结果ready 的同步): 通过使用promise和future进行(`promises_`, `futures_`). 每个bucket会对应一个promise, 来表示一个未来的获取到的返回结果(也就是读取完的buffer), 这个promise对应了一个`shared_future`, 使得所有visitors可以等待该返回结果。
- 实现细节2 (通知IO协调线程free buffers已经有了): 通过一个blocking queue `free_buffers_`来记录free buffers, visitor线程push buffer进入 `free_buffers_`, IO协调者从中pop buffer。

4.4.2. 具体实现

具体实现分为三个部分：IO协调者，IO线程，以及内存vistor线程。

- 对应的IO协调thread逻辑如下 (其中最重要的`ReadBucketToBuffer` 通过保证`request-size = 128KB`，和`queue-depth=8`来打满IO):

```
single_range_io_worker_ = new thread([this]() {
    // Odd Round.
    log_info("In Range IO");
    for (uint32_t next_bucket_idx = 0; next_bucket_idx < BUCKET_NUM;
next_bucket_idx++) {
        // 1st: Pop Buffer.
        auto range_clock_beg = high_resolution_clock::now();
        char *buffer = free_buffers_->pop(total_io_sleep_time_);
        auto range_clock_end = high_resolution_clock::now();
        double elapsed_time =
                duration_cast<nanoseconds>(range_clock_end -
range_clock_beg).count() /
                    static_cast<double>(1000000000);
        total_blocking_queue_time_ += elapsed_time;

        // 2nd: Read
        ReadBucketToBuffer(next_bucket_idx, buffer);
        promises_[next_bucket_idx].set_value(buffer);
    }
    log_info("In Range IO, Finish Odd Round");
```

```
    // Even Round.
    for (uint32_t next_bucket_idx = 0; next_bucket_idx < BUCKET_NUM;
next_bucket_idx++) {
        uint32_t future_id = next_bucket_idx + BUCKET_NUM;
        char *buffer;
        if (next_bucket_idx >= KEEP_REUSE_BUFFER_NUM) {
            // 1st: Pop Buffer.
            auto range_clock_beg = high_resolution_clock::now();
            buffer = free_buffers_->pop(total_io_sleep_time_);
            auto range_clock_end = high_resolution_clock::now();
            double elapsed_time =
                    duration_cast<nanoseconds>(range_clock_end -
range_clock_beg).count() /
                         static_cast<double>(1000000000);
            total_blocking_queue_time_ += elapsed_time;

            // 2nd: Read
            ReadBucketToBuffer(next_bucket_idx, buffer);
        } else {
            buffer = cached_front_buffers_[next_bucket_idx];
        }
        promises_[future_id].set_value(buffer);
    }
    log_info("In Range IO, Finish Even Round");
});
```

- 其中IO协调thread具体的submit读单个bucket任务的ReadBucketToBufferh函数， 通过保证`request-size = 128KB`，和`queue-depth=8`来打满IO, 详细逻辑如下:

```
void EngineRace::ReadBucketToBuffer(uint32_t bucket_id, char *value_buffer) {
    auto range_clock_beg = high_resolution_clock::now();

    if (value_buffer == nullptr) {
        return;
    }

    // Get fid, and off.
    uint32_t fid;
    uint64_t foff;
    std::tie(fid, foff) = get_value_fid_foff(bucket_id, 0);

    uint32_t value_num = mmap_meta_cnt_[bucket_id];
    uint32_t remain_value_num = value_num % VAL_AGG_NUM;
    uint32_t total_block_num = (remain_value_num == 0 ? (value_num /
VAL_AGG_NUM) :
                                (value_num / VAL_AGG_NUM + 1));
    uint32_t completed_block_num = 0;
    uint32_t last_block_size = (remain_value_num == 0 ? (VALUE_SIZE *
VAL_AGG_NUM) :
                                (remain_value_num * VALUE_SIZE));
    uint32_t submitted_block_num = 0;

    // Submit to Maintain Queue Depth.
    while (completed_block_num < total_block_num) {
```

```
        for (uint32_t io_id = 0; io_id < RANGE_QUEUE_DEPTH; io_id++) {
            // Peek Completions If Possible.
            if (range_worker_status_tls_[io_id] == WORKER_COMPLETED) {
                completed_block_num++;
                range_worker_status_tls_[io_id] = WORKER_IDLE;
            }

            // Submit If Possible.
            if (submitted_block_num < total_block_num &&
range_worker_status_tls_[io_id] == WORKER_IDLE) {
                size_t offset = submitted_block_num * (size_t) VAL_AGG_NUM *
VALUE_SIZE;
                uint32_t size = (submitted_block_num == (total_block_num - 1)
?
                                 last_block_size : (VAL_AGG_NUM * VALUE_SIZE));
                range_worker_status_tls_[io_id] = WORKER_SUBMITTED;
                range_worker_task_tls_[io_id]->enqueue(
                UserIOCB(value_buffer + offset, value_file_dp_[fid], size,
offset + foff));
                submitted_block_num++;
            }
        }
    }

    auto range_clock_end = high_resolution_clock::now();
    double elapsed_time = duration_cast<nanoseconds>(range_clock_end -
range_clock_beg).count() /
                          static_cast<double>(1000000000);
    total_time_ += elapsed_time;
}
```

- 对应的IO线程逻辑如下:

```
void EngineRace::InitPoolingContext() {
    io_threads_ = vector<thread>(RANGE_QUEUE_DEPTH);
    range_worker_task_tls_.resize(RANGE_QUEUE_DEPTH);
    range_worker_status_tls_ = new atomic_int[RANGE_QUEUE_DEPTH];
    for (uint32_t io_id = 0; io_id < RANGE_QUEUE_DEPTH; io_id++) {
        range_worker_task_tls_[io_id] = new
moodycamel::BlockingConcurrentQueue<UserIOCB>();
        range_worker_status_tls_[io_id] = WORKER_IDLE;
        io_threads_[io_id] = thread([this, io_id]() {
            UserIOCB user_iocb;
#ifdef IO_AFFINITY_EXP
            setThreadSelfAffinity(io_id);
#endif
            double wait_time = 0;
            for (;;) {
                range_worker_task_tls_[io_id]->wait_dequeue(user_iocb);

                if (user_iocb.fd_ == FD_FINISHED) {
                    log_info("yes! notified, %d", io_id);
                    break;
                } else {
                    pread(user_iocb.fd_, user_iocb.buffer_, user_iocb.size_,
```

```
user_iocb.offset_);
                    range_worker_status_tls_[io_id] = WORKER_COMPLETED;
                }
            }
        });
    }
}
```

- 对应的内存visitor线程的逻辑如下: 其中每个bucket开始有个barrier过程, 在每次结束的时候会更新 `free_buffers_`。 更新buffers的逻辑就是最后一个线程将使用完的buffer放入blocking queue。

```
 // End of inner loop, Submit IO Jobs.
int32_t my_order = ++bucket_consumed_num_[future_id];
if (my_order == total_range_num_threads_) {
    if ((future_id % (2 * BUCKET_NUM)) < KEEP_REUSE_BUFFER_NUM) {
        cached_front_buffers_[future_id] = shared_buffer;
    } else {
        free_buffers_->push(shared_buffer);
    }
}
```

```
static thread_local uint32_t bucket_future_id_beg = 0;

uint32_t lower_key_par_id = 0;
uint32_t upper_key_par_id = BUCKET_NUM - 1;
for (uint32_t bucket_id = lower_key_par_id; bucket_id < upper_key_par_id + 1;
bucket_id++) {
    range_barrier_ptr_->Wait();

    uint32_t future_id = bucket_id + bucket_future_id_beg;
    char *shared_buffer;
    uint32_t relative_id = future_id % (2 * BUCKET_NUM);
    if (relative_id >= BUCKET_NUM && relative_id < BUCKET_NUM +
KEEP_REUSE_BUFFER_NUM) {
        shared_buffer = cached_front_buffers_[relative_id - BUCKET_NUM];
    } else {
        if (tid == 0) {
            auto wait_start_clock = high_resolutIOn_clock::now();
            shared_buffer = futures_[future_id].get();
            auto wait_end_clock = high_resolutIOn_clock::now();
            double elapsed_time = duratIOn_cast<nanoseconds>(wait_end_clock -
wait_start_clock).count() /
                                            static_cast<double>(1000000000);
            wait_get_time_ += elapsed_time;
        } else {
            shared_buffer = futures_[future_id].get();
        }
    }

    uint32_t in_par_id_beg = 0;
    uint32_t in_par_id_end = mmap_meta_cnt_[bucket_id];
    uint64_t prev_key = 0;
```

```
    for (uint32_t in_par_id = in_par_id_beg; in_par_id < in_par_id_end;
in_par_id++) {
        // Skip the equalities.
        uint64_t big_endian_key = index_[bucket_id][in_par_id].key_;
        if (in_par_id != in_par_id_beg) {
            if (big_endian_key == prev_key) {
                continue;
            }
        }
        prev_key = big_endian_key;

        // Key (to little endian first).
        (*(uint64_t *) polar_key_ptr_->data()) = bswap_64(big_endian_key);

        // Value.
        uint64_t val_id = index_[bucket_id][in_par_id].value_offset_;
        polar_val_ptr_ = PolarString(shared_buffer + val_id * VALUE_SIZE,
VALUE_SIZE);

        // Visit Key/Value.
        visitor.Visit(*polar_key_ptr_, polar_val_ptr_);
    }

    // End of inner loop, Submit IO Jobs.
    int32_t my_order = ++bucket_consumed_num_[future_id];
    if (my_order == total_range_num_threads_) {
        if ((future_id % (2 * BUCKET_NUM)) < KEEP_REUSE_BUFFER_NUM) {
            cached_front_buffers_[future_id] = shared_buffer;
        } else {
            free_buffers_->push(shared_buffer);
        }
    }
}

bucket_future_id_beg += BUCKET_NUM;
```

4.4.3. 优化1： 增大IO和内存访问的overlap区域

- 每次处理两轮的任务, 来最小化没有overlapped的IO和内存访问的时间。 详细可见IO线程的逻辑(Odd Round, Even Round)。

4.4.4. 优化2： 减少IO数量 (充分利用剩余的内存)

利用cache一些buffers减少IO数量: cache前几块buffer来进行第二次range的优化, 减少IO数量。

我们设计了`free_buffers_`的逻辑来精确地控制IO buffers和cache的使用。 详细实现可见内存visitor线程收尾阶段，核心代码如下：

```
// End of inner loop, Submit IO Jobs.
int32_t my_order = ++bucket_consumed_num_[future_id];
if (my_order == total_range_num_threads_) {
    if ((future_id % (2 * BUCKET_NUM)) < KEEP_REUSE_BUFFER_NUM) {
        cached_front_buffers_[future_id] = shared_buffer;
    } else {
```

```
            free_buffers_->push(shared_buffer);
        }
    }
```

4.4.5. 优化3： 第一次IO前Populate value-buffer内存

- 通过耗时0.06seconds 左右的内存populate，来使得buffer在第一次使用时候也能达到磁盘读取峰值性能(减少0.1秒)。

```
if (is_first && tid < MAX_TOTAL_BUFFER_NUM) {
    // Really populate the physical memory.
    log_info("Tid: %d, Load Physical Mem %d", tid, tid);
    for (uint32_t off = 0; off < val_buffer_max_size_; off +=
FILESYSTEM_BLOCK_SIZE) {
        value_shared_buffers_[tid][off] = -1;
    }
    is_first = false;
}
```

4.4.6. 优化4: 设置affinity

- 通过set-affinity减少numa之间的切换(io线程绑定到 core-id 0-7)，(减少0.2秒)。

可参考文档：https://blogs.igalia.com/dpino/2015/10/15/multicore-architectures-and-cpu-affinity/ 。

> If a process is running on a core which heavily interacts with an I/O device belonging to different NUMA node, performance degradation issues may appear. NUMA considerably benefits from the data locality principle, so devices and processes operating on the same data should run within the same NUMA node.

5. 比赛经验总结和感想

在Key-Value存储引擎比赛中，我们学习了POLARDB数据库存储引擎相关技术， 设计了相应的文件结构和并发查询算法，来充分榨干傲腾存储的IO能力。 阿里巴巴一向是一个注重开放，使用和创新技术的公司，感谢贵司举办的活动， 让我们了解最新的存储硬件，业界需求以及和其他高手互相切磋，共同进步

6. 附录

6.1. 代码中的一些常量

6.1.1. 存储和索引设计相关

```
// Buffers.
#define TMP_KEY_BUFFER_SIZE (512)
#define TMP_VALUE_BUFFER_SIZE (4)
// Key/Value Files.
#define VALUE_SIZE (4096)

// Buckets.
#define BUCKET_DIGITS (10)      // k-v-buckets must be the same for the range
```

```
query
#define BUCKET_NUM (1 << BUCKET_DIGITS)

// Max Bucket Size * BUCKET_NUM.
#define MAX_TOTAL_SIZE (68 * 1024 * 1024)

#define KEY_FILE_DIGITS (5)     // must make sure same bucket in the same file
#define KEY_FILE_NUM (1 << KEY_FILE_DIGITS)
#define MAX_KEY_BUCKET_SIZE (MAX_TOTAL_SIZE / BUCKET_NUM /
FILESYSTEM_BLOCK_SIZE * FILESYSTEM_BLOCK_SIZE)

#define VAL_FILE_DIGITS (5)
#define VAL_FILE_NUM (1 << VAL_FILE_DIGITS)  // must make sure same bucket in
the same file
#define MAX_VAL_BUCKET_SIZE (MAX_TOTAL_SIZE / BUCKET_NUM /
FILESYSTEM_BLOCK_SIZE * FILESYSTEM_BLOCK_SIZE)
```

6.1.2. 三个阶段逻辑相关

```
// Write.
#define WRITE_BARRIER_NUM (16)
// Read.
#define NUM_READ_KEY_THREADS (NUM_THREADS)
#define NUM_FLUSH_TMP_THREADS (32u)
#define KEY_READ_BLOCK_COUNT (8192u)
// Range.
#define RECYCLE_BUFFER_NUM (2u)
#define KEEP_REUSE_BUFFER_NUM (3u)
#define MAX_TOTAL_BUFFER_NUM (RECYCLE_BUFFER_NUM + KEEP_REUSE_BUFFER_NUM)

#define SHRINK_SYNC_FACTOR (2)      // should be divided

// Range Thread Pool.
#define RANGE_QUEUE_DEPTH (8u)
#define VAL_AGG_NUM (32)

#define WORKER_IDLE (0)
#define WORKER_SUBMITTED (1)
#define WORKER_COMPLETED (2)
#define FD_FINISHED (-2)
```