阿里云 intel

TIANCHI天池

POLARDB 数据库性能大赛总决赛

“Rapids”战队答辩

团队成员介绍

- 车煜林
  - Ph.D. candidate in HKUST (year 4)
  - Research Interests：Parallel Computing, Graph Algorithms
  - Github: https://github.com/CheYulin

- 孙世轩
  - Ph.D. candidate in HKUST (year 4)
  - Research Interests：Parallel Computing, Graph Algorithms
  - Github: https://github.com/shixuansun

- 王立鹏
  - Ph.D. candidate in HKUST (year 4)
  - Research Interests：Parallel Computing, Heterogeneous Computing and Graph Algorithms
  - Github: https://github.com/WANG-lp

提纲

- 赛题背景和分析
- 存储和索引设计
- 随机写入设计
- 随机读取设计
- 顺序全量增序遍历设计
- 总结

赛题背景和分析： Key/Value (K/V) 数据特点，评测流程

- Key固定大小8B，Value固定大小4KB
- Key分布均匀
- 评测含三个分别的子阶段，随机写入，随机读取，顺序读取每个子阶段内不含同时读写
- 评测需支持kill -9后recovery的正确性，需要保证至少写入page cache
- 每个子阶段都有固定64个线程进行Write, Read, Range调用（除了recovery阶段的Range调用）

傲腾存储特征

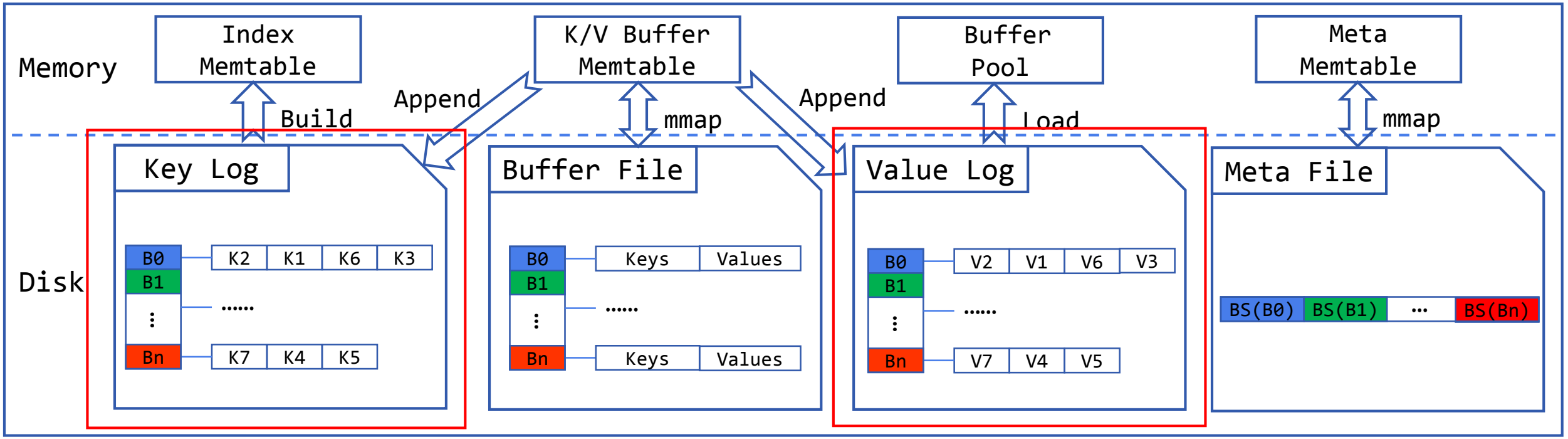
- Request Size (RS) 和 IO Queue Depth (QD)决定latency和throughput
- IO请求过大会被操作系统内核拆分
  - RS_{max}限制可通过 cat /sys/block/sda/queue/max_sectors_kb 进行查看
  - Linux默认设置下， >512KB请求会被操作系统内核拆开
- throughput与RS和QD关系
  - RS=128KB 随机跳读写与顺序读写性能相同
  - RS=16KB 随机写入， QD>8就可以近乎达到峰值throughput
  - RS= 4KB 随机读，QD至少要≥16才能打满iops
  - RS=128KB 顺序读，QD=8就可以达到峰值throughput

随机写，随机读，顺序读分析

- 随机写入，需要使用buffer来达到接近顺序写的性能，并且通过page-cache (mmap-buffer)来保证kill -9时候恢复的正确性，buffer不能太大，太大latency较高，过大会被拆分请求；随机写最后收尾阶段可能只有很小的QD
- 随机读取，无法通过读更大块缓存置换方式提高吞吐量，因此只考虑4KB时候加上同步的优化，来尽量保证QD大于16
- 顺序读取，IO部分选择合适的请求块大小 (RS=128KB)，保持稳定的QD；64线程全量内存遍历，需要充分overlap IO和内存遍历

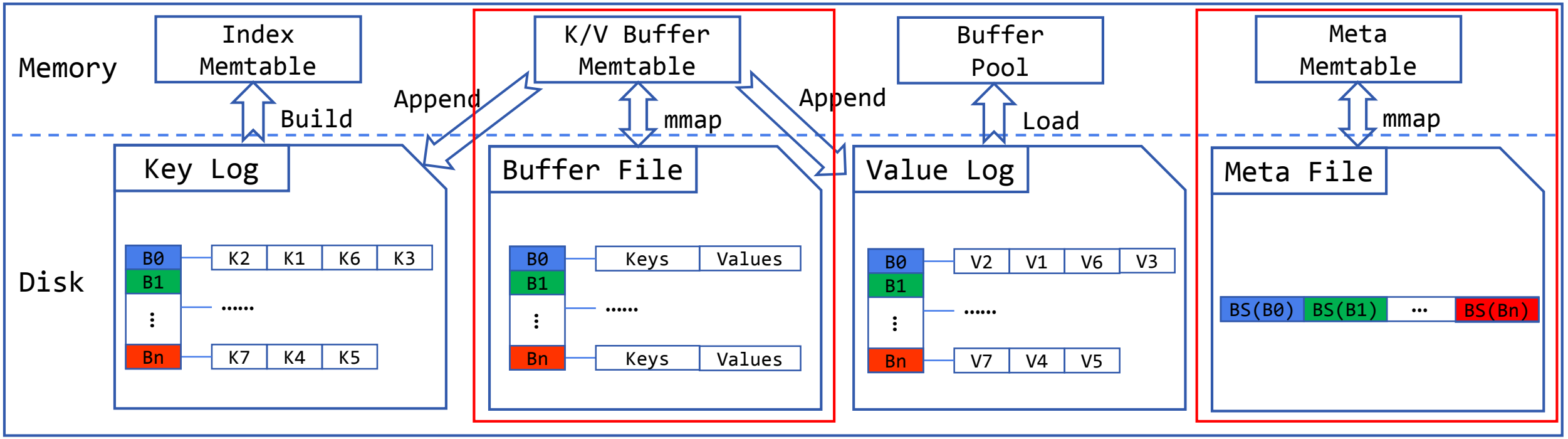
存储及索引设计

- **Key/Value Logs:**
  - (Key, Value)分开存储；
  - (Key, Value)根据Key分别存储于不同桶中；
  - (Key, Value)的桶编号和偏移值相同；
  - Log具有Append-only属性。



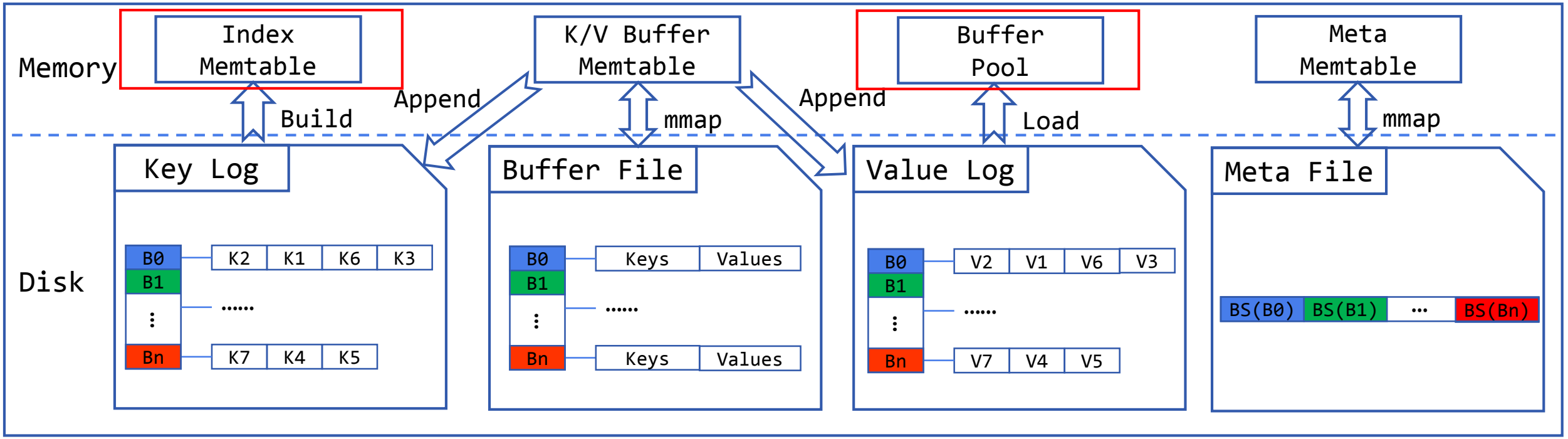
存储及索引设计

- **Key/Value Logs:**
  - (Key, Value)分开存储；
  - (Key, Value)根据Key分别存储于不同桶中；
  - (Key, Value)的桶编号和偏移值相同；
  - Log具有Append-only属性。
- **Buffer/Meta Files (Memtable):**
  - 文件mmap到相应内存中用于保证程序意外退出后数据正确性；
  - Meta File记录桶中Key/Value个数；
  - K/V Buffer用于写入时缓存，当被填满后append到相应log文件。



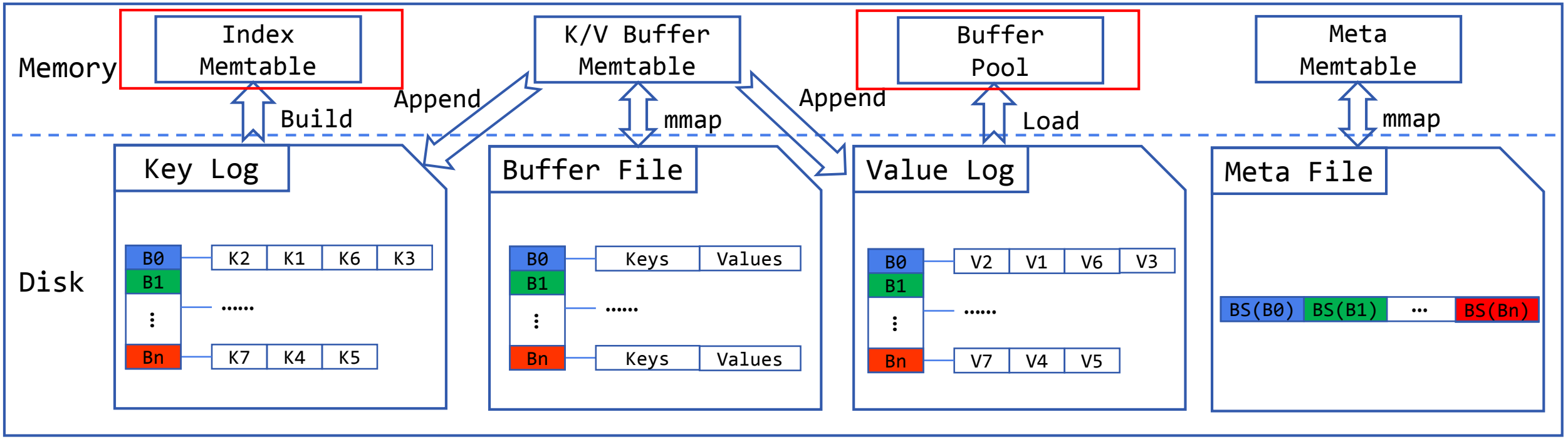
存储及索引设计

- **Key/Value Logs:**
  - (Key, Value)分开存储；
  - (Key, Value)根据Key分别存储于不同桶中；
  - (Key, Value)的桶编号和偏移值相同；
  - Log具有Append-only属性。
- **Buffer/Meta Files (Memtable):**
  - 文件mmap到相应内存中用于保证程序意外退出后数据正确性；
  - Meta File记录桶中Key/Value个数；
  - K/V Buffer用于写入时缓存，当被填满后append到相应log文件。
- **Index Memtable:**
  - 数据库打开时根据Key Log构建;
  - 数据结构: partition + sorted array;
- **Buffer Pool:**
  - Range查询时缓存Value数据。



存储及索引设计

- 除了mmap-buffer和meta-file使用mmap，其他文件操作全使用DirectIO (DIO)，来尽可能消除page-cache使用，减少进程间启动间隔



随机写入阶段思路

- 把8Byte的Key转为Big-Endian的整数，按照高位(10 bits)计算出Bucket ID
- 通过锁一个Bucket使得K/V一一对应起来
- 在整个Bucket锁保护的区域中
  - 先把K/V写入对应Buffer
  - 如果Buffer满了之后就通过pwrite写入到文件中对应位置 (K/V Buffer size = 4KB/16KB)
  - 在K/V均成功写入后更新meta-count

随机写入阶段优化1： 保持Queue Depth (QD)

- 建立Bucket到文件映射，减少K/V文件个数到32
  - 逻辑上的Bucket和文件有对应关系，K/V bucket-id/bucket-off可以对应到K/V WAL文件对应的 fid/foff，因为K均匀所以预先可以为每个Bucket预留足够的空间
  - 增加64线程写同一文件的可能性，而写同一文件可以造成的内核管理的同步，使得线程间收尾时间差别不大，从而维持QD始终大于8

随机写入阶段优化2： 并行Flush buffers ，防止重复Flush

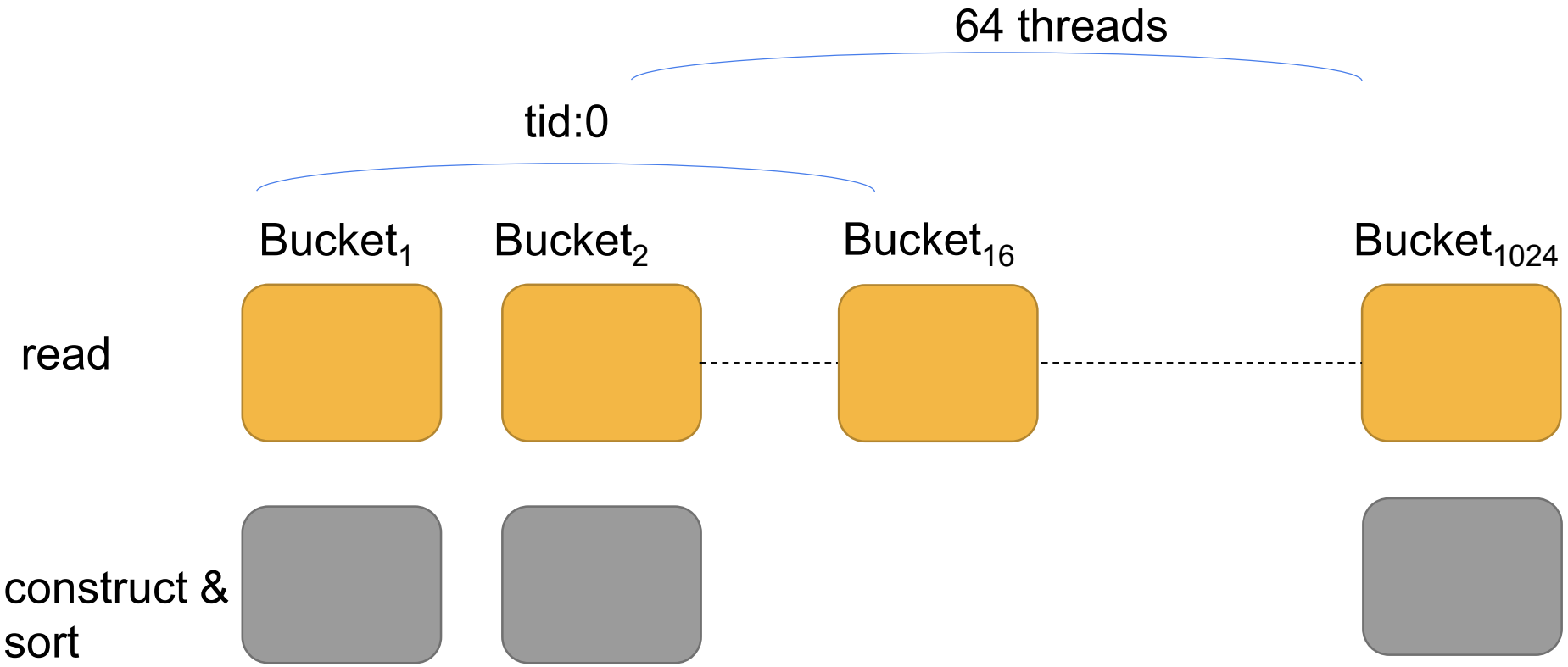
- 如果未被kill -9，写入进行到析构函数时候，尝试并行把mmap-buffers刷出去，并且ftruncate对应文件为0，来减少page cache对进程间启动的影响和防止重复的Flush
- 读取阶段只有在查看对应文件大小不为0，才做并行Flush mmap-buffers和ftruncate对应文件

随机写入阶段效果

- 在有了32个文件的操作系统隐式同步后，稳定性得到了一定的提升
- Throughput $_{max}$ = 1000000*64*(4096./1024./1024+ 8/1024./1024.) / 114.1= 2195.3 MB/s
- Throughput $_{min}$ = 1000000*64*(4096./1024./1024+ 8/1024./1024.) / 115.1 = 2176.3 MB/s

并行索引构建阶段思路

- 读取阶段开始的时候进行并行索引构建
- 每个Bucket对应一个任务，每个任务分为文件读取和构造桶内Index并排序两部分
- IO (read) 和计算部分 (construct & sort) 可以Overlap在一起

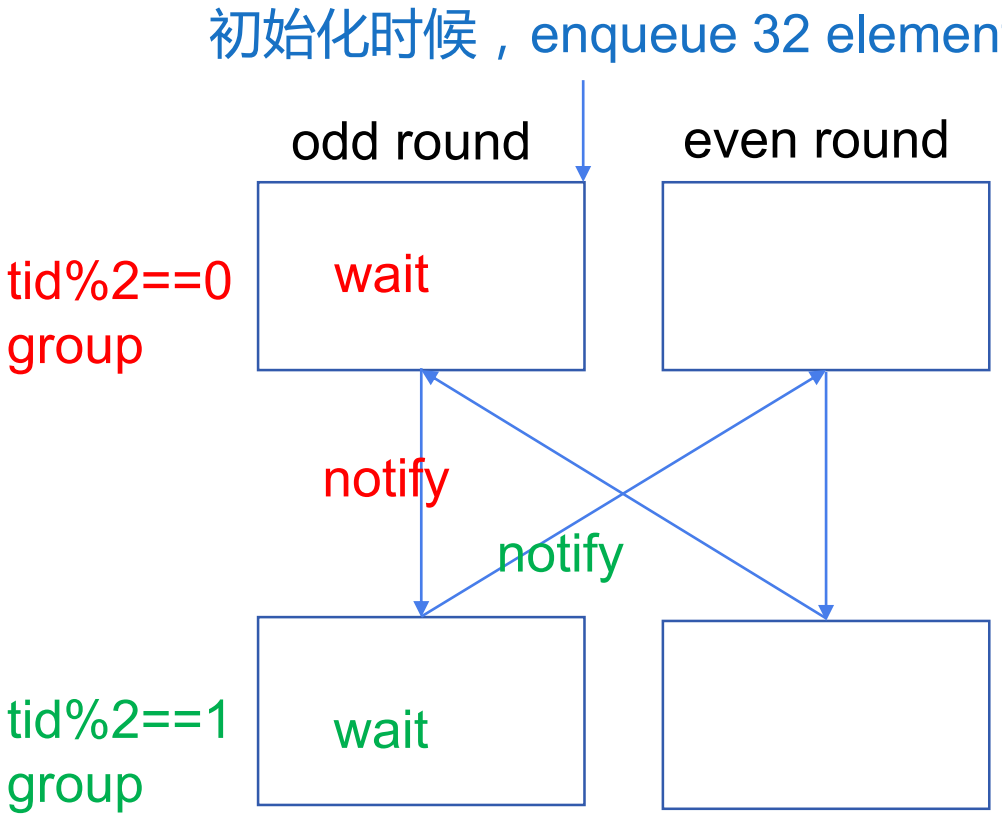


随机读取阶段思路

- 先通过Key计算Bucket，用带prefetch优化的二分查找查询K/V对应bucket中的offset
- 找到的话再看有无更多相等的Key，若有则前进到最后一个相等的Key值对应偏移
- 如果二分查找找到的话直接进行pread 4KB的Value；之前尝试过8KB加缓存的方式，但是缓存占整体命中率低（只命中了4-5GB）没有实质性效果

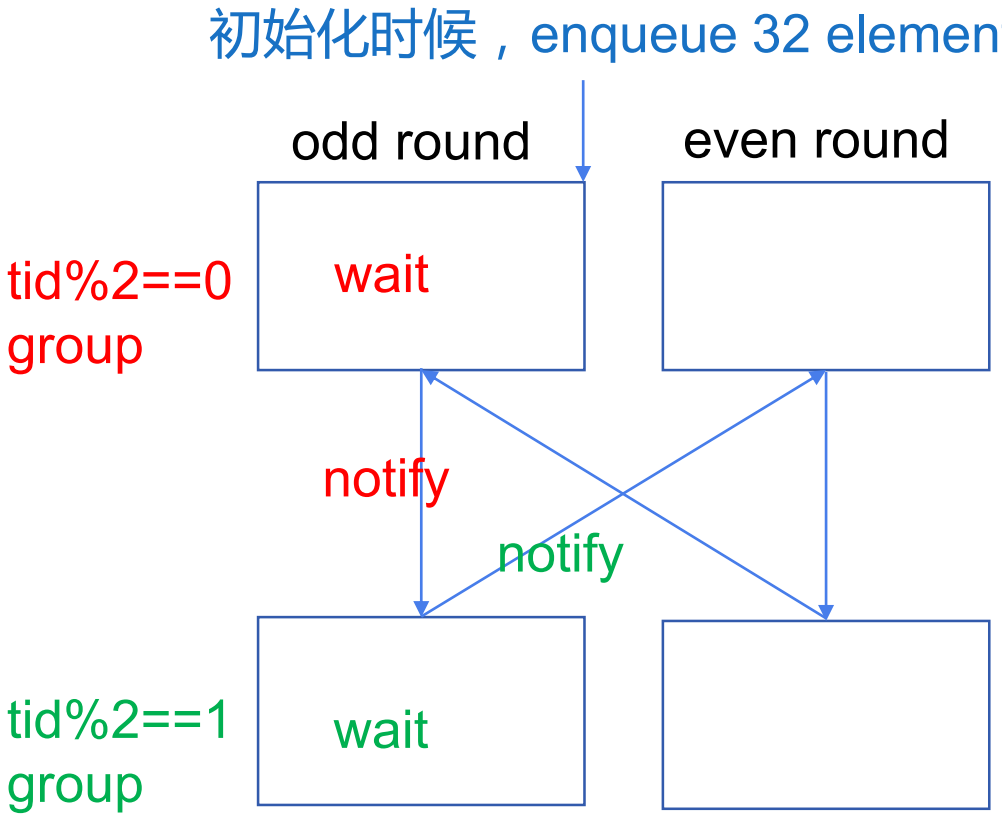
随机读取阶段优化：细粒度同步

- 组成2个group，2个group线程间保持总进度大致一致
- 分4个blocking queue，每个group的奇数和偶数轮分别使用对应queue



随机读取阶段优化：细粒度同步

- 每个group分2个queue是为了为了防止有饥饿的情况，也就是说group中线程进入下一round需要有另一group的通知才可以，从而保证QD稳定20+，并利用同步尽量消除收尾阶段QD<16的情况



随机读取阶段效果

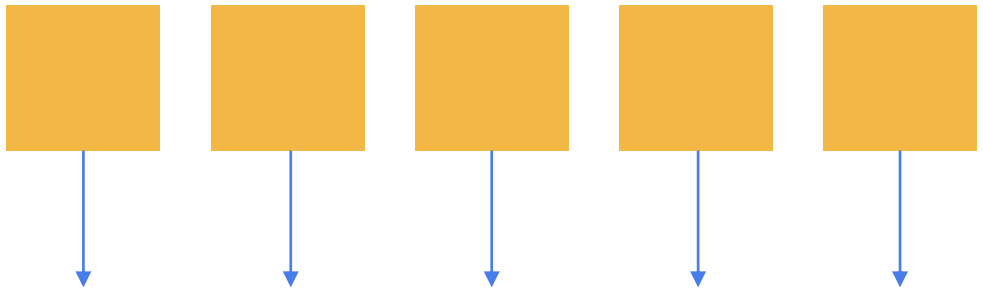
- 在这种细粒度的同步下，所有随机读线程结束仅仅差别0.2s以内，随机读进程时间稳定在105.9-106.1秒，除去0.2秒的索引构建，可以计算出相应的吞吐量
- Throughput $_{max}$ = 1000000*64*(4096./1024./1024.*62./64) / 105.7 = 2291.3 MB/s
- Throughput $_{min}$ = 1000000*64*(4096./1024./1024.*62./64) / 105.9 = 2287.0 MB/s

顺序读取并发访问阶段思路

- 单个IO协调线程
  - 等待free-buffer，已有free-buffer(存一个bucket数据)就一直发大小为128KB IO任务给IO Worker保持QD=8把bucket数据读入free-buffer
  - 读完后通知内存访问线程
- 64个内存访问线程
  - 消费bucket对应的buffer，每进行一次value bucket buffer消费进行一次barrier同步
  - 消费完归还到free-buffer池中，通知IO协调线程

初始化Free Buffer Blocking Queue (BQ)

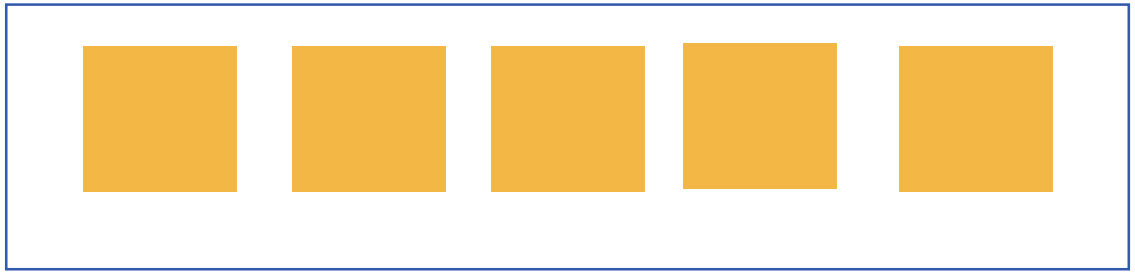
- 构造5个aligned value buffers (size: max bucket disk consumption e.g. 250MB given 1024 buckets)



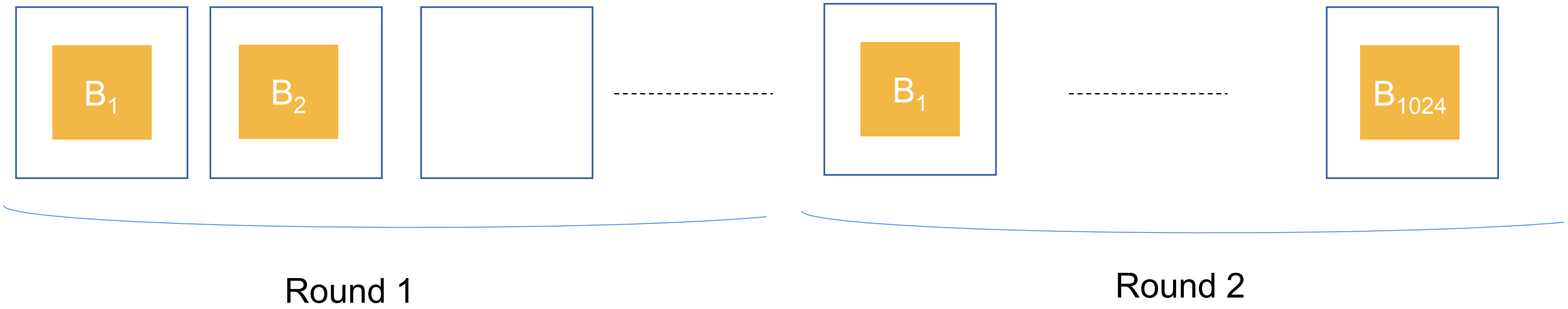
- Enqueue到Free Buffer BQ

初始化Free-Buffer BQ和Promises

- 初始化free-buffer BQ

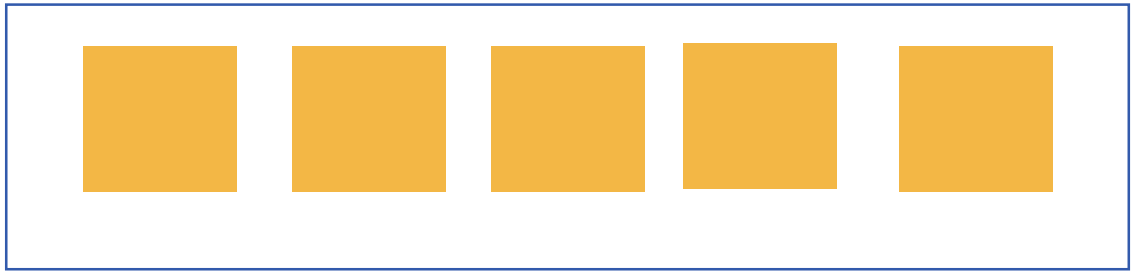


- 初始化promises (promise a bucket value buffer to be used in the future)



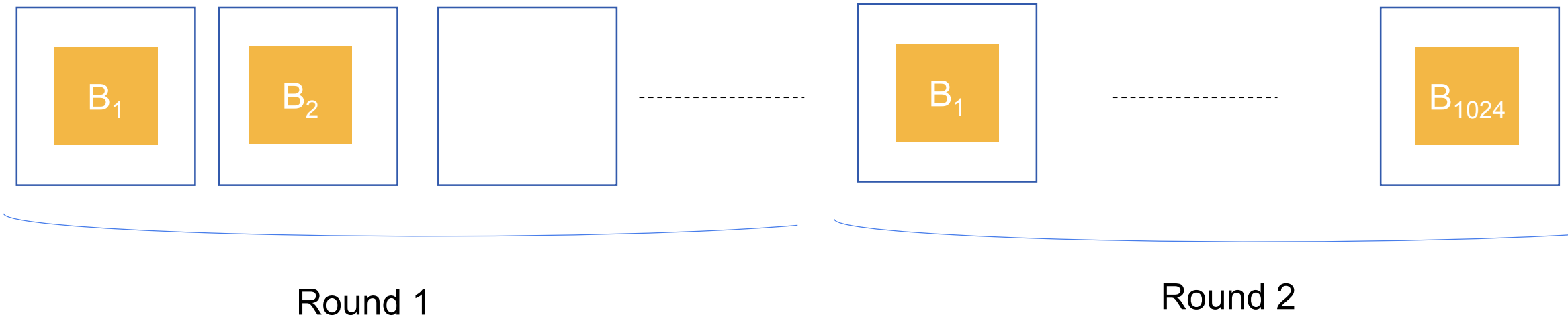
初始化用来减少第二轮全量遍历IO的Cached Buffers

- 初始化free-buffer BQ



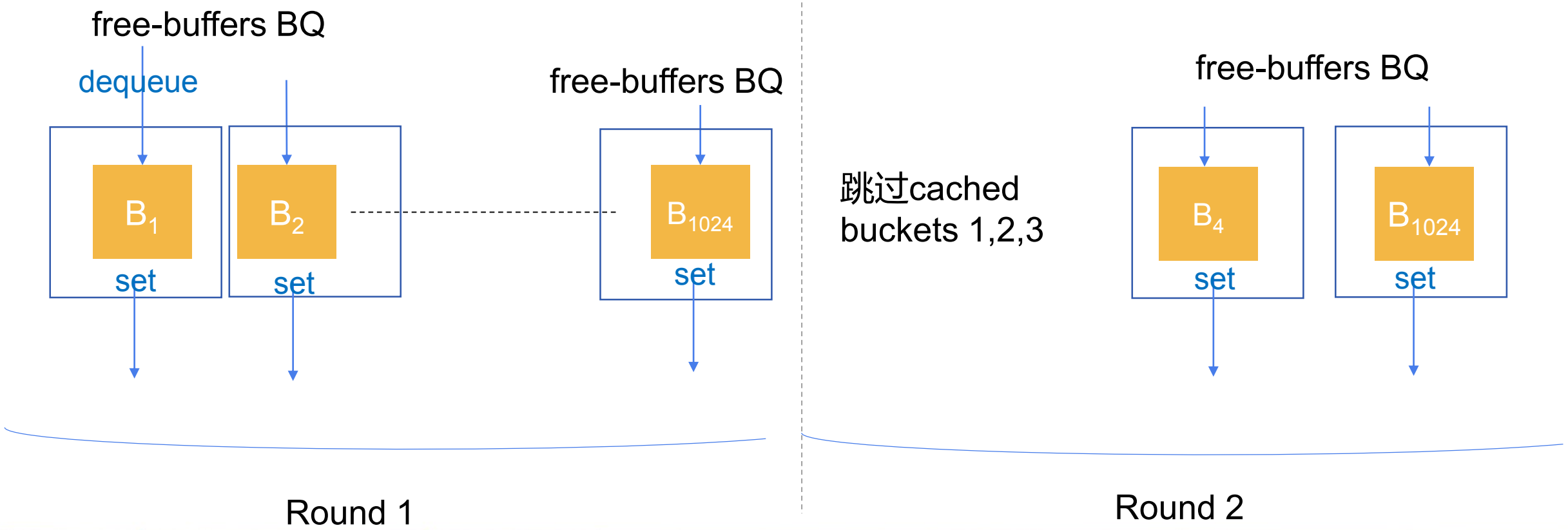
- 初始化cached front buffers

- 初始化promises (promise a bucket value buffer to be used in the future)

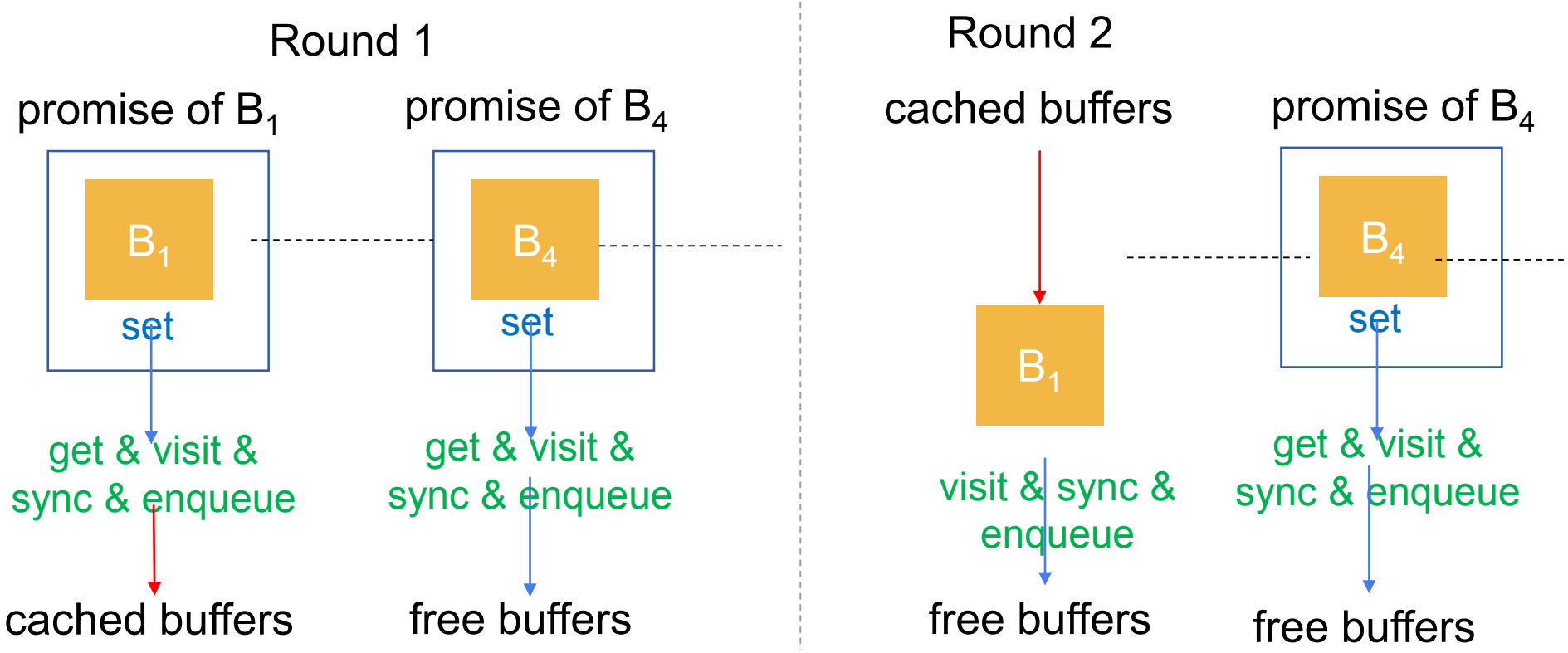


IO协调线程(单线程)行为

- 等待free-buffer，已有free-buffer(存一个bucket数据)就一直发大小为128KB IO任务给IO Worker保持QD=8把bucket数据读入free-buffer
- 当数据读完的时候，set promise的状态为ready

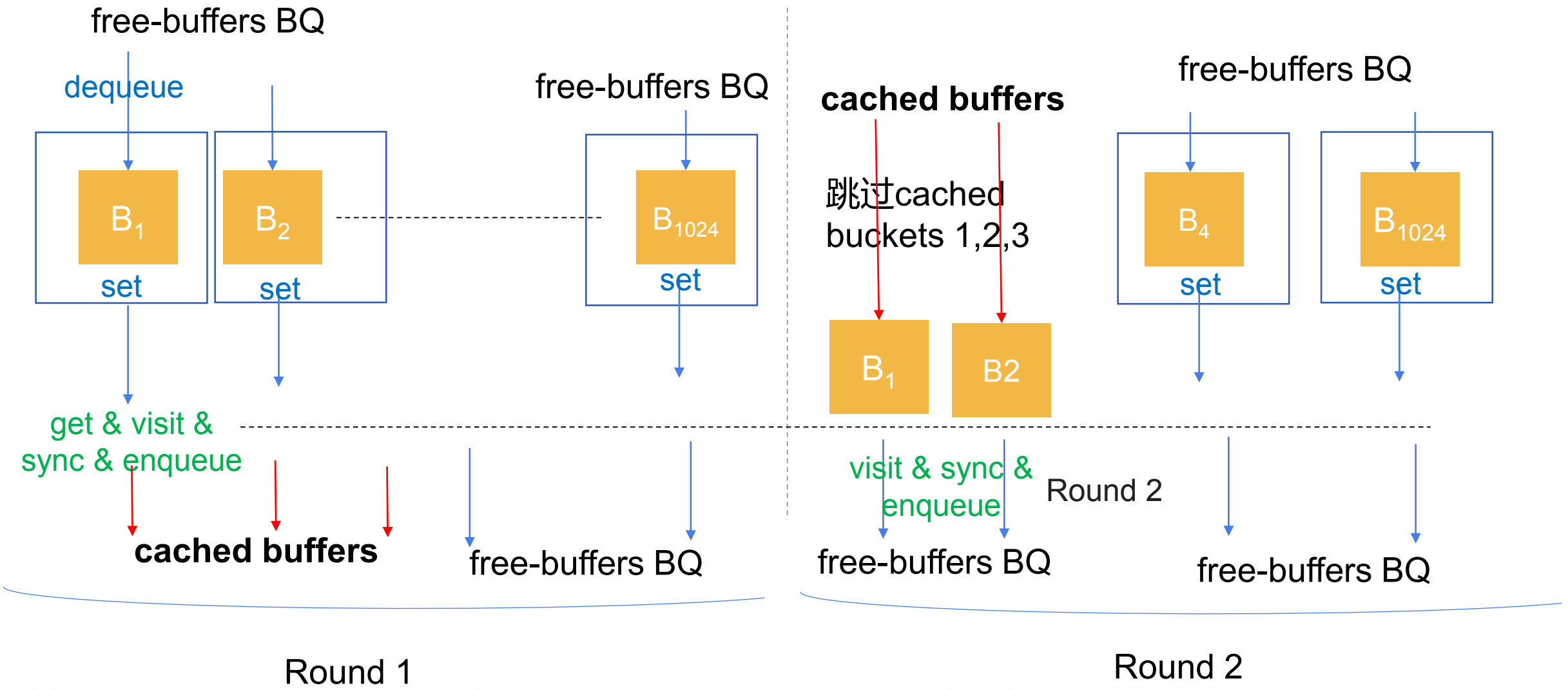


Visitor线程行为



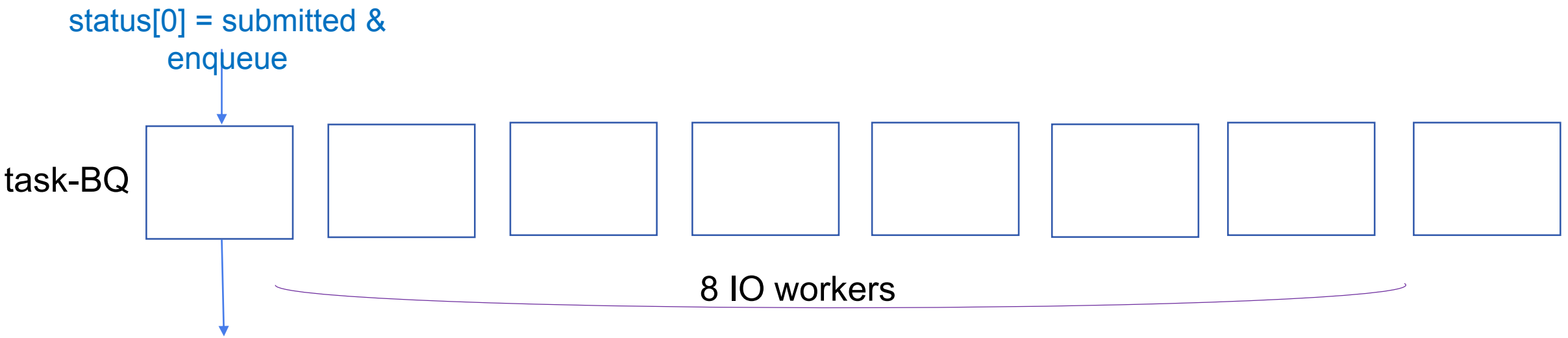
- Get: 通过get IO线程promise的shared future等待bucket buffer
- Visit & Sync: 每人独自visit Bucket，最后进行barrier sync
- Enqueue: 最后一个Visitor把buffer放入free-buffer或者放入cached buffers
- 内存访问大概100秒，大幅小于IO时间192秒，这样通过2块free-buffer就可以充分overlap内存访问与IO，其余buffer用于cached buffers

IO协调线程和Visitor线程行为总结



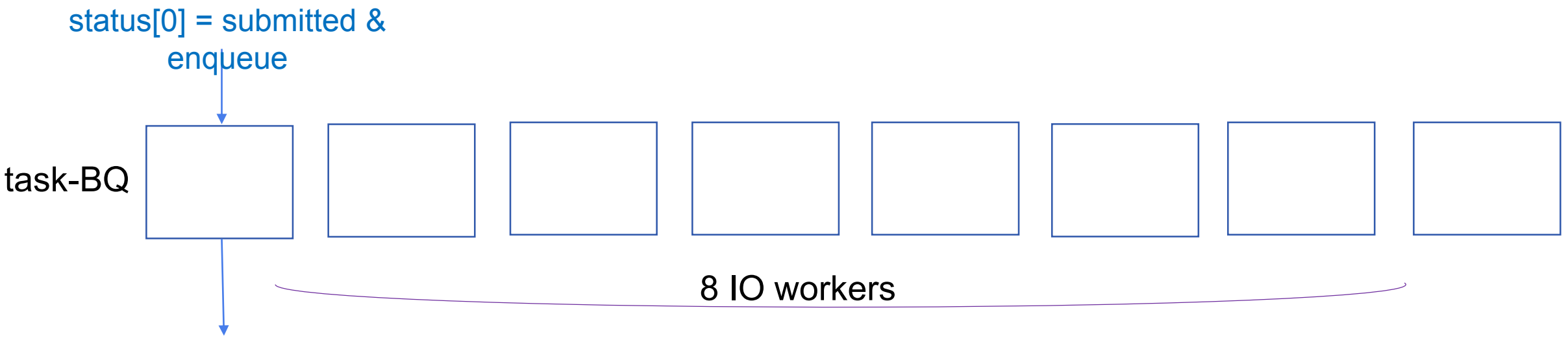
读取单个Bucket B_i

- 单个IO协调线程等到free-buffer后就一直发大小为128KB IO任务给IO Worker保持QD=8把bucket数据读入free-buffer
- 协调者：轮询IO Workers状态直到所有IO任务完成，对当前查看的IO Worker i执行两步操作
  - 查询完成状态，若为completed则将状态(status[i])设置为idle
  - 若有未提交任务并且worker状态为idle，则将状态设为submitted，并enqueue任务到其队列



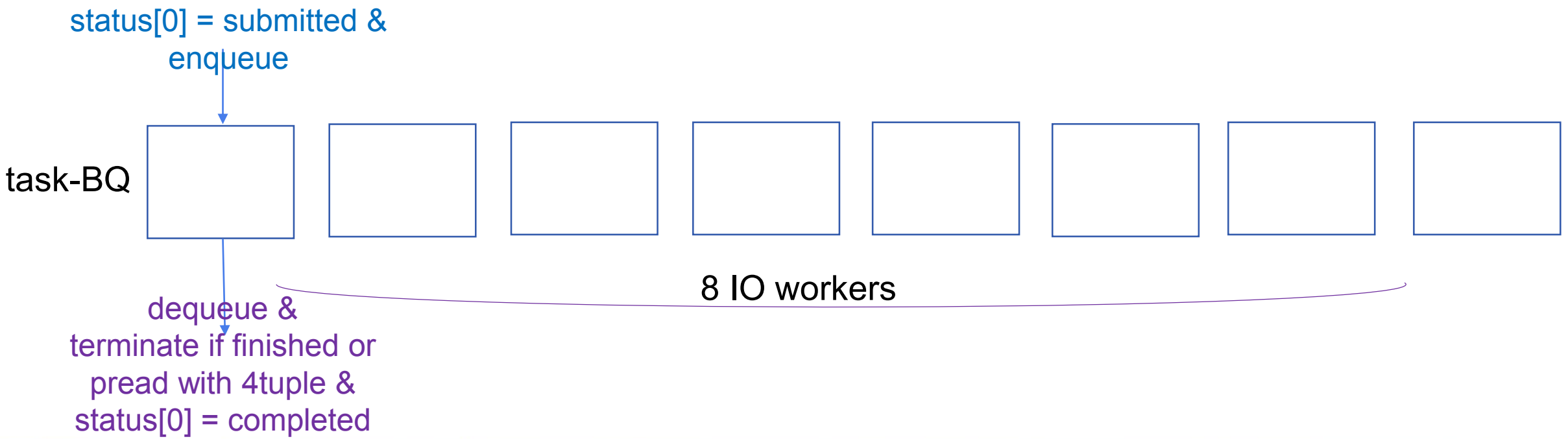
读取单个Bucket B_i

- 单个IO协调线程等到free-buffer后就一直发大小为128KB IO任务给IO Worker保持QD=8把bucket数据读入free-buffer
- IO任务
  - UserIOCB(int fd, char* buffer, size_t size, size_t off)
  - fd=-2 表示特殊的终止提示任务



读取单个Bucket B_i

- 单个IO协调线程等到free-buffer后就一直发大小为128KB IO任务给IO Worker保持QD=8把bucket数据读入free-buffer
- IO Worker
  - 等待在task-BQ上，若出现终止提示任务，则退出线程；若取到任务则根据任务里面的四元组进行pread的操作
  - pread结束后，设置自己状态为completed



顺序读取阶段优化1：增大IO和内存访问Overlap

- 每次处理两轮的任务，最小化没有overlap的IO和内存访问时间；使得第一轮任务的尾部和第二轮任务的开始的IO和内存访问可以overlap
- 可以减少1个bucket内存访问的时间，1024buckets时候大概节省 0.05秒

顺序读取阶段优化2：充分利用内存，减少两次遍历IO数量

- 利用剩余内存cache前几块value-buffer来减少第二次全量遍历时候对这几个bucket的文件读取
- 其中内存访问者的每bucket最后访问者将cached_buffer对应指针存下来
- 在第二轮访问时候前几个bucket使用cached_buffer
- 利用剩余内存，cache 3个250MB buckets时候可以减少 0.25到0.28秒 左右的 IO的时间

顺序读取阶段优化3：第一次使用value-buffer，populate内存

- memalign后物理内存并没有真的申请出来，这会降低第一次利用这些value-buffer进行IO的性能
- 因此花费大概 0.07秒 时间通过写每个page的最后一个byte把内存populate出来，可带来IO上大概 0.2秒 的时间减少， 带来 0.13秒 左右提升。

顺序读取阶段优化4 ：设置线程Affinity，减小NUMA影响

- IO Workers 绑定到CPU Core ID 0到7，对应到同一个NUMA Node
- 大概整体带来 0.2秒 提升

顺序读取阶段效果

- 尝试切换1024/2048/4096 buckets，随机读性能没有太大差异，2048 buckets相对更快
  - 1024 时候：cached buffers总内存占用少250MB
  - 4096 时候：每个bucket较小，不满128KB的最后Request占比增加
- Throughput $_{max}$ = (1000000*64*(2 * 4096./1024./1024. + 8/1024./1024.) - 3 * 250) / 192.1 = 2601.4 MB/s
- Throughput $_{min}$ =(1000000*64*(2 * 4096./1024./1024. + 8/1024./1024.) - 3 * 250) / 192.5 = 2596.0 MB/s

总结

- 存储和索引
  - K/V各32个文件，1个meta文件，K/V buffers各1个文件；索引使用bucket + sorted array；文件实现中，K/V一一对应减少K-WAL大小（不存偏移，动态恢复）
- 写入设计
  - 锁bucket统一K/V，先写16KB mmap-buffer，满了再写对应的文件
  - 并行Flush mmap-buffer，并且通过ftruncate文件为0，保证只flush一次
- 索引构建中根据WAL动态恢复K/V对应bucket中的offset
- 随机读取设计
  - 带prefetch的lower-bound进行基于二分的索引查询
  - 细粒度的同步提高稳定性
- 顺序读取设计
  - 设计IO协调者，内存访问者流水线; IO协调者利用IO Workers保证固定的RS和QD
  - 充分overlap第一轮和第二轮，利用两块free-buffers流畅滚动
  - 利用三块cached-buffers，减少3个bucket的读取
  - 首次使用Value Buffer先Populate物理内存，来打满IO
  - 设置IO Worker affinity，消除NUMA影响

总体效果

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`

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

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

比赛经验总结和感想

- 充分了解了傲腾存储，数据库产品调优时候RS和QD两个参数非常重要
  - Request Size (RS)和IO Queue Depth(QD)决定latency和throughput
  - IO请求过大会被操作系统内核拆分， 过小则距离128KB读写性能有差异；IO请求越大，latency越大；128KB左右为单位的随机读写可认为是顺序读写
  - RS=4KB随机读取时候QD至少要16；更大RS时候可以用更小的QD达到峰值性能

- 展望：绕过操作系统内核，开发用户态直接操作管理IO Request Submission和Completion Queue的组件，用于支持不同场景的需求 (包含latency/throughput两个维度)

Thanks！

Q&A

后备：评测机上的初步性能评测，顺序 vs 随机位置写，变RS

顺序

4KB

8KB

16KB

32KB

64KB

随机位置

```
Step one File exists, 508,500, 116.089
1
Step one File exists, 506,536, 114.852
2
Step one File exists, 508,584, 114.419
3
Step one File exists, 510,688, 114.930
4
Step one File exists, 510,824, 115.395
Close file end..
0
Step one File exists, 513,432, 116.023
1
Step one File exists, 513,432, 114.868
2
Step one File exists, 513,432, 114.491
3
Step one File exists, 513,432, 114.862
4
Step one File exists, 513,432, 115.382
```

16KB最优

16KB最优

顺序与随机写
几乎没有差别

后备：评测机上的初步性能评测，是否fallocate的影响

```
Step one File exists, 508,592, 238.938
Step one File exists, 508,592, 238.992 end.
Step two File exists, 508,592, 109.900
1
rm -r test_directory/*
Step one File exists, 506,752, 115.441
Step one File exists, 506,752, 115.443 end.
Step two File exists, 506,936, 109.898
2
rm -r test_directory/*
Step one File exists, 506,936, 114.264
Step one File exists, 506,936, 114.273 end.
Step two File exists, 506,936, 105.959
3
rm -r test_directory/*
Step one File exists, 506,980, 114.714
Step one File exists, 506,980, 114.724 end.
Step two File exists, 507,132, 100.984
4
rm -r test_directory/*
Step one File exists, 507,344, 115.299
Step one File exists, 507,344, 115.305 end.
Step two File exists, 508,928, 98.516
```

4KB时候 fallocate 影响很大

16KB时候 fallocate 影响不大

```
Release File exists, 508,420, 0.263
Step one File exists, 508,576, 130.198
Step one File exists, 508,576, 130.241 end.
1
rm -r test_directory/*
Release File exists, 506,844, 0.183
Step one File exists, 507,132, 114.513
Step one File exists, 507,132, 114.551 end.
2
rm -r test_directory/*
Release File exists, 507,148, 0.247
Step one File exists, 507,260, 113.964
Step one File exists, 507,260, 113.971 end.
3
rm -r test_directory/*
Release File exists, 507,420, 0.209
Step one File exists, 507,668, 114.389
Step one File exists, 507,668, 114.393 end.
4
rm -r test_directory/*
Release File exists, 507,864, 0.268
Step one File exists, 508,156, 114.736
Step one File exists, 508,156, 114.744 end.
```

后备：评测机上的初步性能评测，顺序 vs 随机 （读写）

顺序

```
Step one File exists, 508,516, 115.656
Step one File exists, 508,516, 115.657 end.
Step two File exists, 508,580, 109.911
1
Step one File exists, 514,656, 114.454
Step one File exists, 514,656, 114.455 end.
Step two File exists, 515,016, 109.804
2
Step one File exists, 515,016, 114.241
Step one File exists, 515,016, 114.242 end.
Step two File exists, 515,016, 106.102
3
Step one File exists, 515,036, 114.576
Step one File exists, 515,036, 114.577 end.
Step two File exists, 515,192, 101.067
4
Step one File exists, 515,308, 115.066
Step one File exists, 515,308, 115.066 end.
Step two File exists, 517,124, 98.652
```

顺序与随机几乎没有差别

随机

```
Step one File exists, 518,108, 115.630
Step one File exists, 518,108, 115.630 end.
Step two File exists, 518,108, 109.734
1
Step one File exists, 518,108, 114.688
Step one File exists, 518,108, 114.688 end.
Step two File exists, 518,108, 110.020
2
Step one File exists, 518,108, 114.349
Step one File exists, 518,108, 114.349 end.
Step two File exists, 518,108, 105.950
3
Step one File exists, 518,108, 114.743
Step one File exists, 518,108, 114.743 end.
Step two File exists, 518,108, 101.057
4
Step one File exists, 518,108, 115.197
Step one File exists, 518,108, 115.197 end.
Step two File exists, 518,108, 98.599
```

后备：参考IO峰值性能

- https://www.storagereview.com/intel_optane_ssd_dc_p4800x_review
- 64KB seq-read peak: `2.53*1024 = 2590.72 MB/s`, latency `394 µs`
- 64KB seq-write peak: `2.17*1024 = 2222.08 MB/s`, latency `380 us`
- 4KB rand-read peak: `585754* 4096./(1024.**2) = 2288.10 MB/s` latency `231 us`

后备：IO请求过大会被拆分

- https://www.pcper.com/reviews/Storage/Intel-Optane-SSD-DC-P4800X-750GB-Review-Flesh/Enterprise-SSD-Testing-and-Jargon
- Transfer Size：If you have ever combed through the various reviews of a given enterprise SSD, you will first note how ‘generic’ the data is. You won’t see specific applications used very often - instead you will see only a hand full of small workloads applied. These workloads are common to the specifications seen across the industry, and typically consist of 4KB and 8KB transfer sizes for random operations and 128KB sizes for sequential operations. 4KB and 8KB cover the vast majority of OLTP (on-line transaction processing) and Database (typically 8K) usage scenarios. 128KB stemmed as the default maximum transfer size as it meshes neatly with the maximum IO size that many OS kernels will issue to a storage device. Little known fact: Windows Operating System kernels will not issue transfer sizes larger than 128KB to a storage device. If an application makes a single 1MB request (QD=1) through the Windows API, that request is broken up by the kernel into 8 128KB sequential requests that are issued to the storage device simultaneously (QD=8, or up to the Queue Depth limit for that device). I’m sorry to break it to you, but that means any benchmark apps you might have seen reporting results at block sizes >128KB were actually causing the kernel to issue 128KB requests at inflated queue depths.

其他：存储设计

- K/V 都是Write-Ahead-Log(WAL)，顺序append，通过meta-count文件记录相应个数
- V大小固定，索引中不需要存V的长度，在文件中偏移可以用偏移量除以4KB来间接代表，减小内存占用
- K大小固定8B，分布随机，可以将K转为Big-Endian整数，再按高位分Bucket来为顺序读取全量访问准备
- 写K/V时候可以将同一Bucket的K/V对一一对应，节省偏移量的序列化空间

其他：索引设计

- 索引不需要支持快速的动态插入，因此可采用先分Bucket，再在Bucket内Sort K/Off对的方式构建，索引的查询通过branchless的lower_bound，找第一个>=pivot的offest进行
- 并行索引构建，每个bucke对应一个任务，这样overlap sort和读K WAL的时间

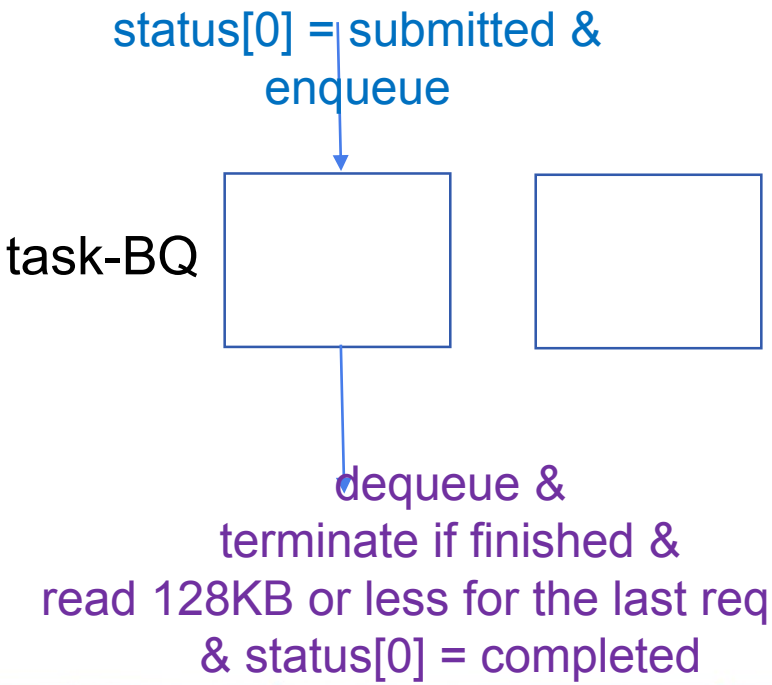
其他：文件细节设计

- K/V文件各32个(每个文件里面再拆开成多个bucket, bucket id相邻的再文件中xianglin)
- meta-count, K-mmap-buffer, V-mmap-buffer各一个，每个文件slice成BUCKET_NUM个views，比赛中我们使用了1024个buckets
- 逻辑上的Bucket和文件有对应关系，K/V bucket-id/bucket-off可以对应到K/V WAL文件对应的fid/foff，因为K均匀所以预先可以为每个Bucket预留足够的空间
- 32个文件设计是为了使得写入时候带有一定同步作用，并且避免过多文件触发操作系统bug从DirectIO进入BufferIO
- meta-count文件，记录每个bucket当前大小
- K/V-mmap-buffer作为缓存, 每个value-buffer/key-buffer 16KB/4KB分别整除4KB和8

其他：读取单个Bucket（维持QD & RS）

- 单个IO协调线程等到free-buffer后就一直发大小为128KB IO任务给IO Worker保持QD=8把bucket数据读入free-buffer

loop for completions and submissions to fill B_i




```
// 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++;
}
```