SSDv2.1 开发与设计可行性研究报告

目　录

1. 引言.....2
1.1 编写目的.....2
1.2 项目背景.....2
1.3 定义、首字母缩写词和缩略语.....2
1.4 参考资料.....2
2. 可行性研究前提.....3
2.1 要求.....3
2.2 目标.....3
2.3 条件、假定和限制.....3
2.4 决定可行性的主要因素.....3
3. 对现有系统的分析（可选）.....4
3.1 功能分析.....4
3.2 性能分析.....4
3.2.1 性能问题.....4
3.2.2 性能分析.....7
3.2.3 性能总结.....11
3.3 可用性分析.....13
3.4 处理流程和数据流程.....13
3.5 局限性.....13
4. 所建议技术和功能流程分析.....14
4.1 业务流程分析.....14
4.2 系统架构分析.....14
4.3 技术可行性分析与对比.....14
4.3.1 锁冲突问题解决之道.....15
4.3.2 索引操作问题解决之道.....19
4.3.3 Read IO 未命中情况下性能较低问题解决之道.....27
4.3.4 版本号更新问题解决之道.....28
4.3.5 写缓存空间不足问题解决之道.....28
4.4 技术可行性测试报告.....29
4.5 对系统运行的影响.....45
4.6 对开发环境的影响.....46
5. 经济可行性分析.....47
6. 其它因素分析（社会因素等）.....48
7. 未来工作.....48
8. 结论.....49

1. 引言

1.1 编写目的

本文档旨在分析当前系统中存在的可以进行性能优化的点，给出相应的优化方案，并给出优化前后的性能对比，以验证性能优化的效果。

本文档主要面向技术委员会评审。

1.2 项目背景

当前系统在升腾测试场景下，性能不尽如人意，甚至在某些情况下测试无法正常进行，经分析讨论，SSD 写缓存具有较大的优化和改进空间，针对 SSD 写缓存在测试中表现出的测试中断、读写性能较低、波动较大、重放较慢和写缓存空间占满之后性能骤降等问题，提出相应的优化方案，以期一定程度上解决上述问题。

1.3 定义、首字母缩写词和缩略语

IOPS： IOs per second；

BW： bandwidth；

Redo：重放，将存储于 SSD 写缓存中的数据转储到后端 HDD 中，以回收写缓存空间；

1.4 参考资料

略。

2. 可行性研究前提

2.1 要求

在不改变现有架构的前提下，尽可能提升系统性能，解决测试中出现的测试中断、读写性能较低、性能波动较大、重放速度较慢和写缓存空间占满之后性能骤降等问题。

2.2 目标

在升腾测试场景下，性能指标全面提升，直至达到升腾测试要求（接近或者超越 XSKY）。

2.3 条件、假定和限制

不改变现有系统架构。

2.4 决定可行性的主要因素

略。

3.对现有系统的分析（可选）

3.1 功能分析

略。

3.2 性能分析

3.2.1 性能问题

当前系统在（升腾）测试过程中发现至少下列问题：

（1）测试性能整体偏低

```
13:53:33.016 Starting RD=SD_format; I/O rate: Uncontrolled MAX; elapsed=(none); For loops: threads=2 iorate=max

六月 06, 2017  interval        i/o   MB/sec   bytes   read     resp     read    write     resp     resp queue  cpu%  cpu%
                             rate  1024**2     i/o    pct     time     resp     resp      max   stddev depth sys+u   sys
19:26:35.522avg_2-19982     304.73    38.09  131072   0.00  246.290    0.000  246.290 27671.003  953.427  75.1   1.8   0.6

19:27:22.003 Starting RD=rd1; I/O rate: Uncontrolled MAX; elapsed=1800; For loops: threads=2 xfersize=4k seekpct=100 rdpct=0

19:57:22.522 avg_2-900      374.73     1.46    4096   0.00  250.916    0.000  250.916 7983.443  767.434  94.0   1.3   0.2   随机写

19:57:29.003 Starting RD=rd1; I/O rate: Uncontrolled MAX; elapsed=1800; For loops: threads=2 xfersize=4k seekpct=100 rdpct=100

20:28:22.522 avg_2-900      420.73     1.64    4096   100  276.022  276.022    0.000 5942.092  731.738  93.9   1.5   0.3   随机读
```

在升腾测试用例下，采用 vdbench 测试，以 128KB 粒度进行置备（对应上图中的 SD_format），性能只有 38 MB/s，以 4KB 粒度进行随机读写测试，性能则分别为 420、374，性能上不忍直视，远远落后于友商的测试结果。

（2）随着数据集的增大，性能迅速下降

```
六月 06, 2017  interval        i/o   MB/sec   bytes   read     resp     read    write     resp     resp queue  cpu%  cpu%
                             rate  1024**2     i/o    pct     time     resp     resp      max   stddev depth sys+u   sys
00:34:33.442       110    1230.00   153.75  131072   0.00  140.622    0.000  140.622 9601.584  612.997 119.7   1.0   0.7
00:34:34.440       111    1165.00   145.63  131072   0.00   87.261    0.000   87.261 1289.949  221.614 120.3   1.7   1.1
00:34:35.441       112    1127.00   140.88  131072   0.00   61.739    0.000   61.739 1342.708  186.443 119.2   1.8   1.0
00:34:36.442       113    1232.00   154.00  131072   0.00   75.428    0.000   75.428 1589.697  242.991 120.1   1.3   0.8
00:34:37.445       114    1377.00   172.13  131072   0.00   54.423    0.000   54.423 1172.126  156.022 119.8   2.0   1.5
00:34:38.441       115    1300.00   162.50  131072   0.00   80.680    0.000   80.680 4797.406  389.232 119.9   1.0   0.7
00:34:39.440       116    1168.00   146.00  131072   0.00  157.510    0.000  157.510 14651.330  910.846 119.6   1.5   1.2
00:34:40.442       117    1403.00   175.38  131072   0.00  155.981    0.000  155.981 8087.538  732.738 119.8   1.4   1.0
00:34:41.441       118    1308.00   163.50  131072   0.00   79.923    0.000   79.923 1310.549  233.589 119.7   1.1   0.8
00:34:42.440       119    1292.00   161.50  131072   0.00   90.321    0.000   90.321 3328.202  327.727 119.8   1.3   1.0
00:34:43.780       120    1301.00   162.63  131072   0.00   86.669    0.000   86.669 2033.163  241.197 120.9   1.8   1.5
```

00:56:53.435	1450	479.00	59.88	131072	0.00	88.723	0.000	88.723	5115.487	462.295	119.8	1.1	0.7
00:56:54.436	1451	563.00	70.38	131072	0.00	321.313	0.000	321.313	4034.492	813.523	119.8	1.7	1.1
00:56:55.437	1452	695.00	86.88	131072	0.00	218.944	0.000	218.944	4174.201	609.452	119.9	1.7	1.1
00:56:56.441	1453	397.00	49.63	131072	0.00	220.680	0.000	220.680	3808.699	604.572	120.0	1.0	0.7
00:56:57.447	1454	489.00	61.13	131072	0.00	205.650	0.000	205.650	3027.773	533.463	119.9	1.0	0.8
00:56:58.439	1455	552.00	69.00	131072	0.00	180.133	0.000	180.133	2595.484	476.638	119.9	0.5	0.4
00:56:59.447	1456	531.00	66.38	131072	0.00	192.441	0.000	192.441	3233.966	527.440	119.9	1.0	0.8
00:57:00.437	1457	497.00	62.13	131072	0.00	198.746	0.000	198.746	2879.915	531.838	119.8	0.9	0.7
00:57:01.437	1458	505.00	63.13	131072	0.00	142.876	0.000	142.876	2636.610	430.130	119.6	1.1	0.8
00:57:02.441	1459	638.00	79.75	131072	0.00	194.740	0.000	194.740	2964.740	505.844	120.3	0.9	0.7
00:57:03.440	1460	655.00	81.88	131072	0.00	188.584	0.000	188.584	9147.902	865.973	119.9	1.0	0.8
01:10:03.436	2240	327.00	40.88	131072	0.00	498.954	0.000	498.954	5553.720	1165.675	120.0	1.4	1.1
01:10:04.443	2241	368.00	46.00	131072	0.00	327.162	0.000	327.162	4691.531	860.158	120.0	1.0	0.8
01:10:05.437	2242	240.00	30.00	131072	0.00	45.015	0.000	45.015	3897.540	358.005	119.9	0.5	0.4
01:10:06.439	2243	166.00	20.75	131072	0.00	768.436	0.000	768.436	6378.723	1405.145	120.1	1.7	1.3
01:10:07.437	2244	17.00	2.13	131072	0.00	793.890	0.000	793.890	2924.143	1271.504	119.9	1.0	0.7
01:10:08.436	2245	496.00	62.00	131072	0.00	521.649	0.000	521.649	6892.600	1372.875	119.8	1.1	0.8
01:10:09.433	2246	317.00	39.63	131072	0.00	327.099	0.000	327.099	5133.639	1059.327	120.0	0.7	0.6
01:10:10.424	2247	260.00	32.50	131072	0.00	341.546	0.000	341.546	3850.822	761.387	119.9	0.7	0.6
01:10:11.435	2248	291.00	36.38	131072	0.00	191.481	0.000	191.481	2696.207	563.124	119.9	0.6	0.5
01:10:12.445	2249	165.00	20.63	131072	0.00	423.769	0.000	423.769	5165.454	1095.040	120.1	0.9	0.7
01:10:13.434	2250	50.00	6.25	131072	0.00	591.156	0.000	591.156	3192.842	1082.804	119.9	0.5	0.4

在第 110-120 个统计周期内，128KB 置备 BW 为 160MB/s 左右，在第 1450-1460 个统计周期内，128KB 置备 BW 则下降到 70MB/s 的水平，在第 2240-2250 个统计周期内，128KB 置备 BW 则进一步下降到 30MB/s 的水平。

（3）IO 波动较大

六月 06, 2017	interval	i/o rate	MB/sec 1024**2	bytes i/o	read pct	resp time
13:54:04.457	31	729.00	91.13	131072	0.00	126.107
13:54:05.435	32	779.00	97.38	131072	0.00	95.883
13:54:06.435	33	933.00	116.63	131072	0.00	77.943
13:54:07.438	34	781.00	97.63	131072	0.00	119.473
13:54:08.436	35	1011.00	126.38	131072	0.00	89.010
13:54:09.436	36	687.00	85.88	131072	0.00	73.266
13:54:10.440	37	651.00	81.38	131072	0.00	116.481
13:54:11.435	38	691.00	86.38	131072	0.00	109.825
13:54:12.436	39	714.00	89.25	131072	0.00	283.427
13:54:13.423	40	1119.00	139.88	131072	0.00	98.951
13:54:14.435	41	677.00	84.63	131072	0.00	88.718
13:54:15.455	42	746.00	93.25	131072	0.00	131.547
13:54:16.433	43	730.00	91.25	131072	0.00	71.467
13:54:17.430	44	713.00	89.13	131072	0.00	119.044
13:54:18.433	45	880.00	110.00	131072	0.00	78.896
13:54:19.437	46	368.00	46.00	131072	0.00	105.992
13:54:20.434	47	772.00	96.50	131072	0.00	109.608
13:54:21.433	48	585.00	73.13	131072	0.00	170.210
13:54:22.434	49	914.00	114.25	131072	0.00	240.926
13:54:23.439	50	713.00	89.13	131072	0.00	162.739
13:54:24.433	51	507.00	63.38	131072	0.00	160.084
13:54:25.435	52	622.00	77.75	131072	0.00	180.807

在第 31-52 个统计周期内，IOPS 在 300-1100 之间大幅震荡，非常不稳定。

（4）测试可能中断

六月 06, 2017	interval	i/o rate	MB/sec 1024**2	bytes i/o	read pct	resp time	read resp	write resp	resp max	resp stddev	queue depth	cpu% sys+u	cpu% sys

01:10:26.430	2263	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.0	1.1	0.7
01:10:27.436	2264	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	119.9	0.2	0.2
01:10:28.434	2265	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	119.9	0.6	0.4
01:10:29.435	2266	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.1	0.2	0.2
01:10:30.436	2267	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.1	0.4	0.3
01:10:31.429	2268	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	119.9	0.4	0.4
01:10:32.432	2269	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.0	0.4	0.4
01:10:33.433	2270	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.1	0.7	0.5
01:10:34.428	2271	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	119.9	0.9	0.7
01:10:35.433	2272	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.1	0.3	0.3
01:10:36.444	2273	0.00	0.00	0	0.00	0.000	0.000	0.000	0.000	0.000	120.0	0.9	0.7

在第 2263-2273 个统计周期内，IO 为 0，vdbench 测试中断。

（5）重放较慢

来自解决方案工程师的反馈：

从最初测试开始至今天暴露出的问题如下：

1.SSD工作机制问题，SSD写满后持续大量数据写入，会造成IO阻塞，虚拟机hung死；

2.对存储进行长时间高负载（3P82V持续写入5小时以上）写入会造成存储间歇性io为零或成块IO为零，测试中断，虚拟机hung死（不可操作，不可强制重启）；

3.8G dom0内存在出现大量数据持续写入是，会出现dom0内存耗尽，物理节点hung住；

3.针对升腾场景测试用例，VM置备和基准测试过程中IO值波动非常大；

4.在用户环境测试，用vdbench持续高负载写入数据时，观察，出现磁盘繁忙程度差异很大，每节点5块磁盘加入存储pool，每节点上3或4块磁盘，处于持续繁忙状态，1-2块持续处于闲置状态；进入各磁盘挂载目录，统计chunk文件数，繁忙程度不同的磁盘，存在较大差异；3000 - 500

5.SONE存储数据用到1.6T，全节点重启或更换OSS程序，挂载存储长时间处于64k，至少半小时以上，SSD进行recover 时间过长；也就是说用户存放一定数据量，如果出现全断电，服务器启动后，将会很长时间业务处于不可用状态（服务器启动时间+服务启动+SSD recover）；

6.SSD redo速度偏慢，从今天凌晨3点左右测试中断，停止测试IO读写，到早上九点查看SSD可用容量仍在30G，6个小时没有测试数据，单块SSD redo释放的容量2G左右（此问题还要进一步验证）

经过 6 个小时左右的时间，Redo 只释放了 2GB 的 SSD 容量！

（6）SSD 写缓存空间写满之后性能骤降

来自解决方案工程师的反馈：

从最初测试开始至今天暴露出的问题如下：

1.SSD工作机制问题，SSD写满后持续大量数据写入，会造成IO阻塞，虚拟机hung死；

2.对存储进行长时间高负载（3P82V持续写入5小时以上）写入会造成存储间歇性io为零或成块IO为零，测试中断，虚拟机hung死（不可操作，不可强制重启）；

3.8G dom0内存在出现大量数据持续写入是，会出现dom0内存耗尽，物理节点hung住；

3.针对升腾场景测试用例，VM置备和基准测试过程中IO值波动非常大；

4.在用户环境测试，用vdbench持续高负载写入数据时，观察，出现磁盘繁忙程度差异很大，每节点5块磁盘加入存储pool，每节点上3或4块磁盘，处于持续繁忙状态，1-2块持续处于闲置状态；进入各磁盘挂载目录，统计chunk文件数，繁忙程度不同的磁盘，存在较大差异；3000 - 500

5.SONE存储数据用到1.6T，全节点重启或更换OSS程序，挂载存储长时间处于64k，至少半小时以上，SSD进行recover 时间过长；也就是说用户存放一定数据量，如果出现全断电，服务器启动后，将会很长时间业务处于不可用状态（服务器启动时间+服务启动+SSD recover）；

6.SSD redo速度偏慢，从今天凌晨3点左右测试中断，停止测试IO读写，到早上九点查看SSD可用容量仍在30G，6个小时没有测试数据，单块SSD redo释放的容量2G左右（此问题还要进一步验证）

SSD 写缓存空间用完之后，后续写 IO 长时间得不到响应！

3.2.2 性能分析

关于上述（升腾）测试中呈现的问题，归纳起来就是：

（1）在 SSD 有可用空间的情况下，写性能问题（包括整体性能较低，性能波动较大，随着数据集增大性能下降厉害，测试可能会中断）；

（2）在 SSD 空间不足情况下，写性能问题（SSD 写满之后性能骤降）；

（3）读性能问题（读性能较低）；

（4）重放（Redo）较慢问题；

关于问题（1）- SSD 空间可用的情况下写性能问题的分析：

根据和 XSKY 测试结果对比，可以排除 IO 压力不足、网络达到瓶颈、SSD 自身性能不足等可能原因，主要从存储引擎内部分析原因所在。经分析，在存储引擎内部，在整条数据通路上，SSD 写缓存所花费的时间在 IO 请求总时间中占据相当大的比例，那么 SSD 写缓存中所花费的时间都去哪儿了呢？SSD 写缓存做的事情很简单，对于写请求来说，就是准备数据（非对齐的情况下）、写入数据和更新索引三个步骤，这些都是必要的步骤，没有精简的余地。既然如此，那么时间到底花费在哪里呢？答案是等待，关于锁的等待！下面我们着重分析关于锁的等待是怎么产生的。

SSD 写缓存中为了防止不同类型的不相关的 IO 请求之间的相互等待实现了读写线程分离，同时也为了防止 redo 对于读写线程的影响，也为 redo 单独提供一个线程，所以对于写缓存来说，它包含三种类型的线程：读线程，写线程，redo 线程，这三种线程各司其职，但是它们之间存在着一个非常重要的共享内存 - 索引，对于线程间的共享内存在访问的时候需要解决并发访问的问题，当前通过采用锁（读写锁）来控制并发访问，如下：

写线程	
写数据	复用 LogUnit
.....	.....
加写锁；	加写锁；
更新索引；	删除所有满足特定条件（率属于当前即将被复用的 LogUnit）的索引；
解除写锁；	解除写锁；
.....	.....

读线程
读取数据

…… 加读锁； 查找索引； 解除读锁； ……

Redo 线程	
Redo 准备阶段	Redo 执行阶段（对于每一个 Redo 的 IO 执行完毕之后）
…… 加读锁； 查找所有满足特定条件（当前即将 redo 的 LogUnit）的索引； 根据查找到的索引建立反向索引； 解除读锁； ……	…… 加写锁； 更新索引，标识该 IO 执行过 Redo； 解除写锁； ……

锁冲突矩阵：

冲突情况		写线程		读线程	Redo 线程	
		写数据	复用 LogUnit	读数据	Redo 准备	Redo 执行
写线程	写数据			×	×	×
	复用 LogUnit			×	×	×
读线程	读数据	×	×			×
Redo 线程	Redo 准备	×	×			
	Redo 执行	×	×	×		

从锁冲突矩阵来看，写线程可能会和 Redo 线程产生锁冲突，当前 Redo 线程实现上一个基本原则是“Redo as much as possible”，在 Redo 准备阶段会较长时间锁住索引，因为它要遍历索引表（上亿量级）查找所有率属于当前即将执行 Redo 的 LogUnit 的索引（上万量级），在 Redo 执行阶段，每执行一个 Redo IO，都会去更新索引表，每次更新索引表过程中持有锁的时间较短，但是会频繁申请和释放锁。

从锁冲突矩阵来看，写线程在混合读写环境下可能会和读线程产生锁冲突，在纯写环境下则不存在和读线程之间的锁冲突。

另外，在写线程复用 LogUnit 的时候，需要从索引表中删除该 LogUnit 相关的索引，这个过程需要遍历索引表，找到所有率属于该 LogUnit 的索引，删除之，并在删除过程中，尝试进行 B*树的合并，根据之前的观察，这个过程往往达到秒级！

综上所述，对于问题（1）中所述的写性能问题，原因总结如下：

写性能问题	原因
整体性能较低	写线程和读线程以及 Redo 线程之间存在频繁的锁冲突；
性能波动较大	写线程和读线程以及 Redo 线程之间存在频繁的锁冲突；
随着数据集增大性能下降厉害	写线程和读线程以及 Redo 线程之间存在频繁的锁冲突； 索引表中元素数目太多，查询/更新/删除耗时随着数据集增大而增大；
测试可能会中断	写线程和 Redo 线程准备阶段存在锁冲突； 写线程复用 LogUnit 时释放索引花费较长时间；

关于问题（2）- SSD 空间不足情况下写性能问题的分析：

在 SSD 写缓存中写数据之前都必须先申请 SSD 缓存空间，随着数据的写入，SSD 缓存空间逐渐变小，持续的写入一定会出现 SSD 缓存空间消耗殆尽的情况，虽然 Redo 线程在不遗余力的 Redo，但是在持续大量 IO 写入的情况下，redo 的速率（写入 HDD）是远远跟不上 SSD 的写入速率的。在当前的设计中，为简单起见，所有的 IO 都是经过 SSD 写缓存的，这样在 SSD 空间不足的情况下，新到达的写 IO 会不断的尝试申请 SSD 写缓存空间，直到申请到缓存空间为止(Redo 线程 Redo 至少一个 LogUnit 之后才可能申请到,该过程通常是秒级别的)，申请到 SSD 写缓存空间之后，该缓存空间并不是立即可用的，必须由写线程释放该缓存空间所关联的所有的索引（根据之前的观察，该过程往往达到秒级别）方可使用。

综上所述，对于问题（2）中所述的 SSD 空间不足情况下写性能问题，原因总结如下：

SSD 空间不足时写性能问题	原因
写性能骤降	写线程等待可用的缓存空间（由 Redo 线程负责回收）； 写线程复用 LogUnit 时删除索引比较耗时；

关于问题（3）- 读性能问题的分析：

在关于问题（1）的分析中，通过锁冲突矩阵可以清晰地看到，读线程会和 Redo 线程执行阶段产生锁冲突，在非纯读环境下也会和写线程产生锁冲突，任何的锁冲突势必对读性能造成影响。另外，读性能与读请求在 SSD 写缓存中命中率也有较大的关系，如果读请求未能在 SSD 写缓存中命中，则该读请求需要去后端 HDD 中读取数据，而 HDD 的访问时延可能高达几毫秒，更为重要的是，读 IO 可能会和 Redo IO 之间争用有限的 IO 资源，对于 HDD 来说，混合读写（Redo IO 是写访问，读 IO 是读访问）性能更差！

综上所述，对于问题（3）中所述的读性能问题，原因总结如下：

读性能问题	原因

读性能较低	读线程会和 Redo 线程执行阶段产生冲突； 在非纯读环境下也会和写线程产生冲突； 未命中 SSD 写缓存的情况下需要从后端 HDD 读取，读 IO 和 Redo IO 之间争用有限的 IO 资源；

关于问题（4）- 重放（Redo）速度较慢问题的分析：

当前关于 Redo 的设计原则是“Redo as mush as possible”，且采用异步 IO 加上预取机制，理论上可以保证源源不断的产生 Redo IO，而不会出现 Redo 线程饿死的情况，但是当我们对 Redo 线程所执行的工作进行分解，就会发现 Redo 线程并非只执行 Redo IO 这么简单，如下：

Redo 线程	
Redo 准备阶段	…… 加读锁； 查找所有满足特定条件（当前即将 redo 的 LogUnit）的索引； 根据查找到的索引建立反向索引； 解除读锁； ……
Redo IO 执行阶段	…… 对于所有查找到的索引，分别执行： 尝试合并相邻的数据； 从 SSD 读取相邻的数据并拼接； 尝试申请 AIO 资源，直到申请到为止； 根据拼接后的数据准备 AIOBlock 并提交给 AIO； …… …… 当收到关于某个提交的 Redo IO 的 AIOCallback 时执行： 加写锁； 更新索引，标识该 IO 执行过 Redo； 解除写锁； 释放 AIO 资源； ……
Redo 扫尾阶段	…… 等待所有关于当前 LogUnit 的 Redo IO 都收到 AIOCallback； 更新当前 LogUnit 关联的所有 Chunk 的版本信息（大量 4K 随机小粒度粒度写）； 确保版本信息成功落盘； ……

从 Redo 线程过程分解来看，在 Redo 准备阶段和扫尾阶段，都是不会执行 Redo IO 的，根据之前的观察，这两个阶段时间消耗可达秒级别，在 Redo IO 执行阶段也有频繁的锁操作。

综上所述，对于问题（4）中所述 Redo 较慢的问题，原因总结如下：

Redo 性能问题	原因
Redo 较慢	Redo 准备阶段读锁会和写线程中写锁产生冲突； Redo 准备阶段遍历索引和建立反向索引操作比较耗时； Redo IO 执行阶段频繁申请写锁，会和写线程的写锁以及读线程的读锁产生冲突； Redo IO 执行阶段 AIO 资源有限，如果申请不到，则需要等待； Redo 扫尾阶段需要等待所有提交的 AIO 的 AIOCallback； Redo 扫尾阶段需要大量更新相关 Chunk 的版本号并确保落盘（大量随机 4K 小粒度 IO）；

3.2.3 性能总结

为方便起见，将 3.2.2 章节中关于性能分析的原因汇总：

写性能问题	原因
整体性能较低	写线程和读线程以及 Redo 线程之间存在频繁的锁冲突；
性能波动较大	写线程和读线程以及 Redo 线程之间存在频繁的锁冲突；
随着数据集增大性能下降厉害	写线程和读线程以及 Redo 线程之间存在频繁的锁冲突； 索引表中元素数目太多，查询/更新/删除耗时随着数据集增大而增大；
测试可能会中断	写线程和 Redo 线程准备阶段存在锁冲突； 写线程复用 LogUnit 时释放索引花费较长时间；

SSD 空间不足时写性能问题	原因
写性能骤降	写线程等待可用的缓存空间（由 Redo 线程负责回收）； 写线程复用 LogUnit 时删除索引比较耗时；

读性能问题	原因
读性能较低	读线程会和 Redo 线程执行阶段产生冲突； 在非纯读环境下也会和写线程产生冲突； 未命中 SSD 写缓存的情况下需要从后端 HDD 读取，读 IO 和 Redo IO 之间争用有限的 IO 资源；

Redo 性能问题	原因

Redo 较慢	Redo 准备阶段读锁会和写线程中写锁产生冲突； Redo 准备阶段遍历索引比较耗时； Redo IO 执行阶段频繁申请写锁，会和写线程的写锁以及读线程的读锁产生冲突； Redo IO 执行阶段 AIO 资源有限，如果申请不到，则需要等待； Redo 扫尾阶段需要等待所有提交的 AIO 的 AIOCallback； Redo 扫尾阶段需要大量更新相关 Chunk 的版本号并确保落盘（大量随机 4K 小粒度 IO）；

上述各问题的原因可以进一步归结如下：

锁冲突问题	索引操作问题	Redo 后版本更新问题	缓存空间不足问题	读 IO 未命中 SSD 问题	其它问题
写线程写锁和读线程读锁冲突	数据集较大情况下，批量查询索引较慢	Redo 扫尾阶段需要更新大量 Chunk 的版本号并确保落盘（大量随机 4K 小粒度 IO）	写线程需要等待可用的缓存空间	读未命中 SSD 写缓存时需要从后端 HDD 读取，和 Redo IO 形成 IO 资源竞争	Redo 执行阶段 AIO 资源有限，可能需要等待 AIO 资源
写线程写锁和 Redo 线程写锁冲突	数据集较大情况下，批量删除索引较慢			在大块顺序读情况下，可以很好的利用预读功能，但是如果存在部分页在 SSD 写缓存中命中，另一部分页在 HDD 命中，则会破坏预读，导致预读失效	Redo 扫尾阶段需要等待所有提交的 AIO 的 AIOCallback
写线程写锁和 Redo 线程读锁冲突					
写线程读锁和 Redo 线程写锁冲突					
读线程读锁和 Redo 线程写锁冲突					

3.3 可用性分析

略。

3.4 处理流程和数据流程

略。

3.5 局限性

略。

4.所建议技术和功能流程分析

4.1 业务流程分析

略。

4.2 系统架构分析

略。

4.3 技术可行性分析与对比

为方便起见，再次列出系统中存在的影响系统性能的因素：

锁冲突问题	索引操作问题	Redo 后版本更新问题	缓存空间不足问题	读 IO 未命中 SSD 问题	其它问题
写线程写锁和读线程读锁冲突	数据集较大情况下，批量查询索引较慢	Redo 扫尾阶段需要更新大量 Chunk 的版本号并确保落盘（大量随机 4K 小粒度 IO）	写线程需要等待可用的缓存空间	读未命中 SSD 写缓存时需要从后端 HDD 读取，和 Redo IO 形成 IO 资源竞争	Redo 执行阶段 AIO 资源有限，可能需要等待 AIO 资源
写线程写锁和 Redo 线程写锁冲突	数据集较大情况下，批量删除索引较慢			在大块顺序读情况下，可以很好的利用预读功能，但是如果存在部分页在 SSD 写缓存中命中，另一部分页在 HDD 命中，则会破坏预读，导致预读失效	Redo 扫尾阶段需要等待所有提交的 AIO 的 AIOCallback
写线程写锁和 Redo 线程读锁冲突					
写线程读锁和 Redo 线程写锁冲突					

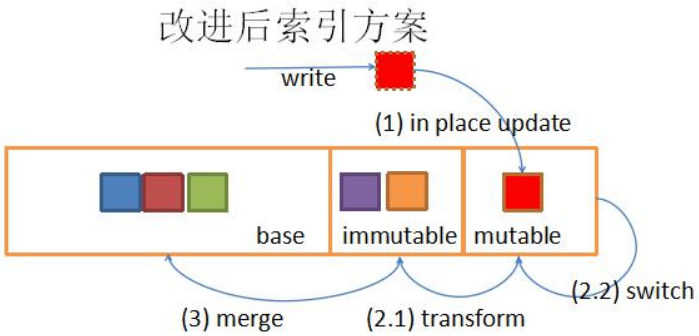
读线程读锁和Redo线程写锁冲突					

从上表中可以明显的看到，锁冲突问题是解决系统性能问题的重中之重，索引操作（尤其是批量索引操作）问题也具有较高的优先级，Redo 后版本更新问题导致 Redo 速率较慢，写缓存空间逐渐被消耗殆尽，最终出现缓存空间不足问题，而读 IO 未命中 SSD 的问题，直接导致读性能大幅下降，其它问题中列举的因素也会一定程度上影响 Redo 线程 Redo 的速率，都需要去寻找相应的解决之道，如下：

问题	锁冲突问题	索引操作问题	版本号更新问题	缓存空间不足问题	读 IO 未命中 SSD	其它问题
解决之道	分类索引	索引分区	利用 LevelDB	ByPass SSD	Redo IO 限流	略

4.3.1 锁冲突问题解决之道

这里不打算描述锁冲突问题解决之道的进化历程，直接描述改进前的索引方案和改进后的索引方案。由于索引更新方式直接决定了索引查询/删除方式，这里直接以索引更新方式来描述索引方案。



改进前的索引方案中，索引更新执行全索引范围就地更新，任何一次索引更新/删除都需要在全索引范围内加写锁，任何一次索引查找都需要在全索引范围内加读锁。

改进后的索引方案则将索引划分为 3 个区：mutable 索引区、immutable 索引区和 base 索引区，这 3 个区的联动关系将通过索引更新、索引查找和索引删除分别讲述。

索引更新分为 3 个步骤：

（1）就地更新 mutable 索引区；

（2）IO 过程中发生 LogUnit 切换且旧的 immutable 索引区的索引已经合并到 base 索引区，则将 mutable 索引区转换为 immutable 索引区；同时重新分配一个 mutable 索引区并切换到该新分配的 mutable 索引区；

（3）Immutable 索引区和 base 索引区合并；

其中步骤（1）在每次写操作过程中都会执行，步骤（2）只在 mutable 索引区中的索引对应的数据量达到一定阈值时才会执行，步骤（1）和（2）都是在写线程内部进行的，步骤（3）只在 mutable 索引区转换为 immutable 索引区之后才会发生，且由 Redo 线程执行。

索引查找则分为 3 个阶段：

（1）查找 mutable 索引区，如果全部找到，则直接返回，否则进入下一步；

（2）如果需要查找的索引中至少有一个在 mutable 索引区中没找到，则查找 immutable 索引区，如果在 immutable 索引区查找的索引都找到了，则直接返回，否则进入下一步；

（3）如果需要查找的索引中至少有一个在 mutable 索引区和 immutable 索引区中都没找到，则查找 base 索引区，如果在 base 索引区查找的索引都找到了，则直接返回，否则在 SSD 写缓存中没有该索引；

索引删除也分为 3 个阶段：

（1）从 mutable 索引区删除给定的索引；

（2）从 immutable 索引区删除给定的索引；

（3）从 base 索引区删除给定的索引；

另外，索引改进前，Redo 线程在执行阶段每当一个 Redo IO 执行完毕都需要去更新该 Redo IO 相关的索引，但是经分析，在 Redo 暂不支持跨 LogUnit 合并的情况下，该操作是不必要的，所以该操作在改进后的索引中直接去掉了，避免了 Redo 线程在 Redo 执行阶段和写线程/读线程发生锁冲突的可能性。

为什么在索引操作中需要锁保护？一方面是为了确保索引操作的原子性，另一方面是为了确保读请求能够读到它之前的最新数据。那么改进后的锁能否满足这两个原则呢？首先，我们将改进前的全索引范围内的锁分裂成改进后的 mutable 索引区锁、immutable 索引区锁和 base 索引区锁，在操作 mutable 索引区的索引时会加上 mutable 索引区锁，在操作 immutable 索引区的索引时会加上 immutable 索引区锁，在操作 base 索引区的索引时会加上 base 索引区锁，可以确保索引操作的原子性。其次，虽然在改进后的索引方案中，存在（关于同一数据页的）多个版本的索引的可能，但是在索引查找过程中，总是按照优先查找 mutable 索引区，再查找 immutable 索引区，最后查找 base 索引区的方式进行，假如某个索引存在多个版本，那么 mutable 索引区中的版本是最新的，immutable 索引区中的版本是次新的，base 索引区中的版本是最旧的，按照前述查找方式，一定找到的是最新版本的索引，假如某个索引只存在单一版本，那么该索引存在且只存在于 mutable 索引区、immutable 索引区和 base 索引区中的一个，按照上述查找方式，也一定能在该索引所在的索引区找到正确的索引，找到正确是索引就可以确保读到正确的最新的数据。

下面我们看一下改进前后，针对系统中各关键加锁操作的加锁范围对比：

改进前方案	
加锁操作	索引范围
写线程写锁	全部索引
写线程读锁	全部索引
读线程读锁	全部索引
Redo 线程准备阶段读锁	全部索引
Redo 线程执行阶段写锁	全部索引

改进后方案	
加锁操作	索引范围
写线程写锁	mutable 索引/immutable 索引 (mutable 转换时发生)
写线程读锁	mutable 索引 + immutable 索引 + base 索引
读线程读锁	mutable 索引 + immutable 索引 + base 索引
Redo 线程准备阶段读锁	base 索引
Redo 线程合并阶段读锁	immutable 索引
Redo 线程合并阶段写锁	base 索引

（*注*：Redo 线程合并阶段对应于改进后索引方案中合并阶段）

相应的，改进后的锁冲突矩阵：

Mutable 索引区锁冲突矩阵						
	写线程写锁	写线程读锁	读线程读锁	Redo 线程准备阶段读锁	Redo 线程合并阶段读锁	Redo 线程合并阶段写锁
写线程写锁			×			
写线程读锁						
读线程读锁	×					
Redo 线程准备阶段读锁						
Redo 线程合并阶段读锁						
Redo 线程合并阶段写锁						

Immutable 索引区锁冲突矩阵						
	写线程写锁	写线程读锁	读线程读锁	Redo 线程准备阶段读锁	Redo 线程合并阶段读锁	Redo 线程合并阶段写锁
写线程写锁			×		×	
写线程读锁						

读线程读锁	×					
Redo 线程准备阶段读锁						
Redo 线程合并阶段读锁	×					
Redo 线程合并阶段写锁						

Base 索引区锁冲突矩阵						
	写线程写锁	写线程读锁	读线程读锁	Redo 线程准备阶段读锁	Redo 线程合并阶段读锁	Redo 线程合并阶段写锁
写线程写锁						
写线程读锁						×
读线程读锁						×
Redo 线程准备阶段读锁						
Redo 线程合并阶段读锁						
Redo 线程合并阶段写锁		×	×			

改进后的索引方案中，除了 mutable 索引区是频繁更新以外（每一次写请求都会更新至少一个索引），immutable 索引区和 base 索引区都是在特定事件发生时进行批量更新。

从 mutable 索引区锁冲突矩阵可以看出，mutable 索引区不存在写写锁冲突，只存在读写锁冲突，且读写锁冲突也只发生在写线程更新和读线程读取并发的时候，但是因为 mutable 索引区中的索引数目相对较小，锁冲突概率大大减小。

从 immutable 索引区锁冲突矩阵可以看出，immutable 索引区不存在写写锁冲突，只存在读写锁冲突，但是 immutable 索引区只在特定事件（相对于 IO 事件，非常少）发生的那一时刻加写锁，锁冲突的概率相对较小，且 immutable 索引区中的索引数目相对较小，即使发生锁冲突，冲突时间也不会太长。

从 base 索引区锁冲突矩阵可以看出，base 索引区也不存在写写锁冲突，只存在读写锁冲突，但是 base 索引区只在特定事件（相对于 IO 事件，非常少）发生的那一时刻加写锁，锁冲突的概率也相对较小。

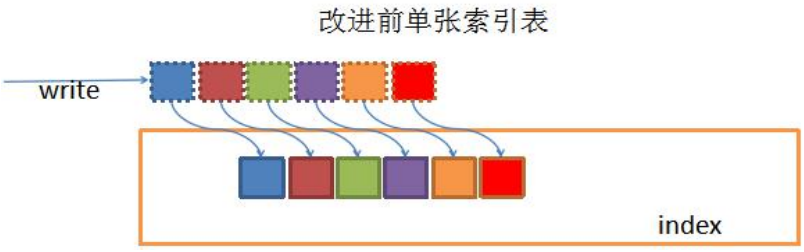
综上所述，通过改进后的索引方案和锁方案，极大减少了发生锁冲突的时机，降低锁冲突概率，减少因为锁导致的 IO 等待。

关于锁，还有另外一项改进。前面讲到，在当前系统实现中，写线程复用 LogUnit 的时候，由自己负责删除关于该即将被复用的 LogUnit 的索引，该过程比较耗时，考虑再三，将该删除操作转嫁给 Redo 线程来执行，写线程在复用 LogUnit 的时候直接复用即可，使得写线程真正专注于执行 IO。但是 Redo 线程在删除所有关于某个 LogUnit 的索引的过程中需要加写锁，这期间无论是读线程还是写线程查询索引的操作都将被较长时间地阻塞，导致读写请求响应时延较大，读写性能降低，为了确保在 Redo 线程删除关于某个 LogUnit 的索引的过程中到达的读写请求的响应时延，采用分时间片删除的方案，在该时间片内部 Redo 线

程持续拥有写锁，时间片结束的时候 Redo 线程释放写锁，以便读写线程被调度到的时候读写请求有机会被执行，然后在 Redo 线程被再次调度到的时候，新的时间片开始计时。

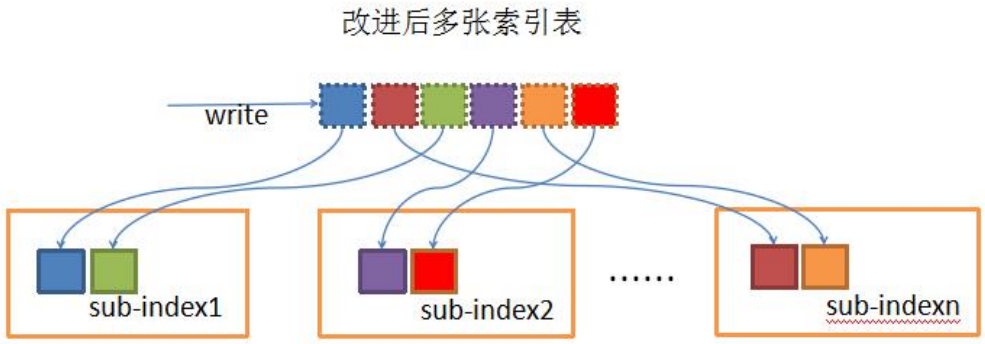
4.3.2 索引操作问题解决之道

索引操作的问题在于数据集较大的情况下，索引中管理了非常多的索引（亿量级），索引操作效率较低。既然数据集大小不是我们可控的，那么是否可以将全索引拆分为多个子索引区呢，这样每个索引区管理的是一个相对较小的子数据集，答案是肯定的，这就是咱们即将谈到的索引分表方案。



改进前所有的索引都更新到单张索引表中，随着数据集的增大，索引表中元素的个数也会增大，更新/查找/删除过程中需要比较的元素数目变大，插入/删除过程中需要移动的元素的个数也变多。

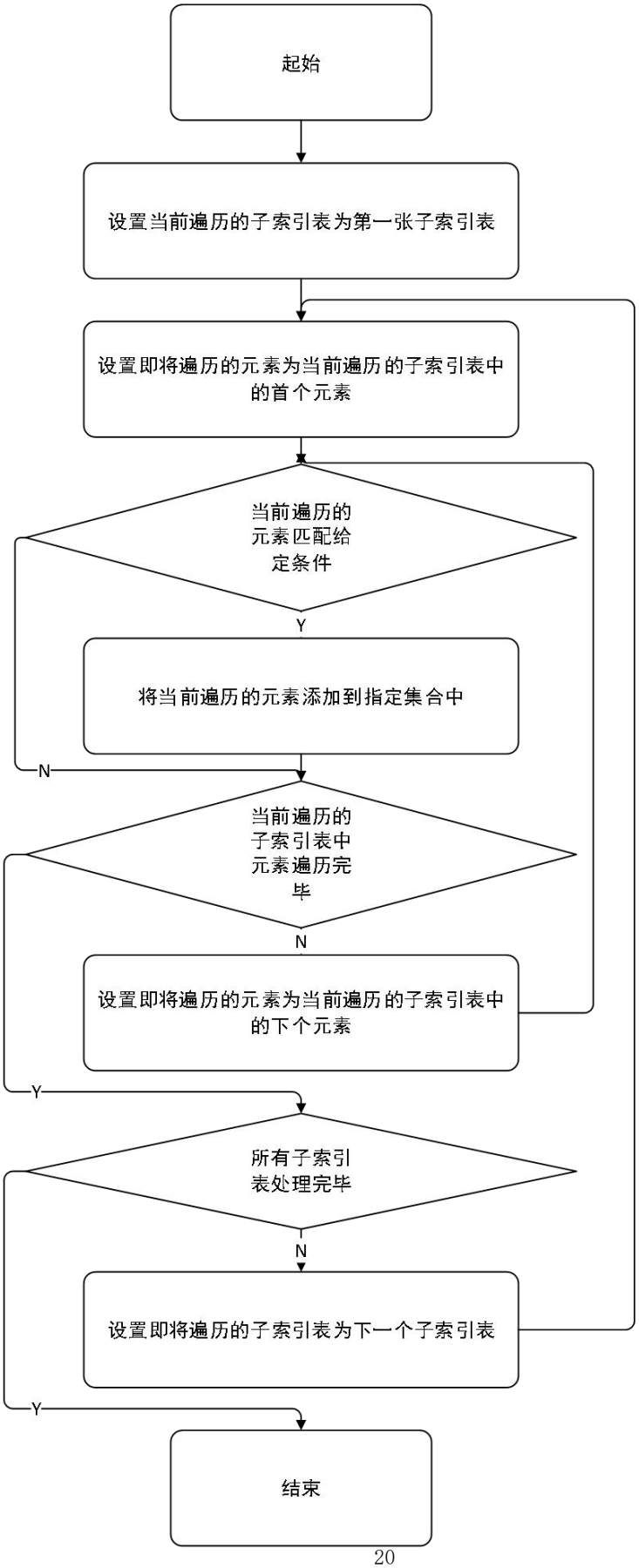
改进后的索引根据每一个 Chunk 的 inodeid 和 chunkindex 计算 Hash 值进行分表，根据计算得到的 Hash 值决定该 Chunk 相关的索引信息插入到哪张子索引表中，在查找和删除过程中也会根据 Chunk 的 inodeid 和 chunkindex 计算 Hash，进入相应的子索引表中进行查找和删除，这样以来，每张子索引表操作的都是全索引的一个互不相交的子集，从统计角度来讲，插入/查找/删除过程中需要比较的元素数目会变少，插入/删除过程中需要移动的元素的个数也会变少。



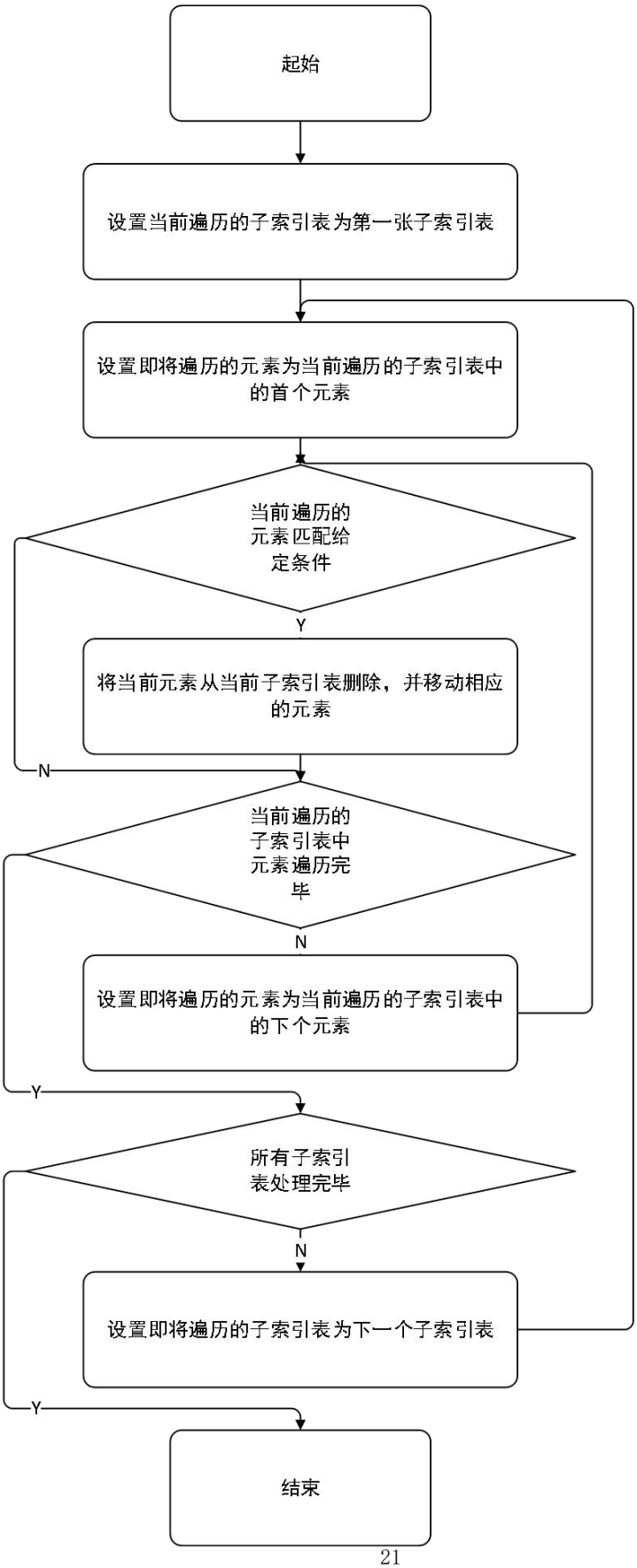
索引分表方案对于解决数据集较大情况下插入/查找/删除效率问题有一定的帮助，但是对于系统中存在的批量查找和批量删除花费较长时间却无能为力。

批量查找发生在 Redo 线程的准备阶段，从所有子索引表中找到所有率属于当前 LogUnit 的索引，这些索引可能杂乱无章的分布在不同的子索引表中，批量删除则发生在复用 LogUnit 的时候，从所有子索引表中删除所有率属于该即将复用的 LogUnit 的索引，同样这些索引也很可能杂乱无章的分布在不同的子索引表中，在改进之前批量查找和批量删除按如下方式进行：

改进前批量查询


起始
设置当前遍历的子索引表为第一张子索引表
设置即将遍历的元素为当前遍历的子索引表中的首个元素
当前遍历的元素匹配给定条件
Y
将当前遍历的元素添加到指定集合中
N
当前遍历的子索引表中元素遍历完毕
N
设置即将遍历的元素为当前遍历的子索引表中的下个元素
Y
所有子索引表处理完毕
N
设置即将遍历的子索引表为下一个子索引表
Y
结束

改进前批量删除


起始
设置当前遍历的子索引表为第一张子索引表
设置即将遍历的元素为当前遍历的子索引表中的首个元素
当前遍历的元素匹配给定条件
Y
将当前元素从当前子索引表删除，并移动相应的元素
N
当前遍历的子索引表中元素遍历完毕
N
设置即将遍历的元素为当前遍历的子索引表中的下个元素
Y
所有子索引表处理完毕
N
设置即将遍历的子索引表为下一个子索引表
Y
结束

从改进前批量查询和批量删除的流程可以看到，批量查询和批量删除中有很多无效（不匹配）的遍历，如果可以采取措施减少遍历这些不匹配的元素（更有针对性的遍历）则遍历会有效很多。基于此，设计了如下改进方案：

（1）在更新关于某个 LogUnit 相关的索引信息的时候，记录下这些索引都更新了哪些子索引表，对于那些在更新过程中未被更新过的子索引表，在遍历过程中就没必要去遍历了，姑且称之为 IndexFilter；

（2）在更新关于某个 LogUnit 相关的索引信息的时候，记录下与该 LogUnit 相关的索引的范围，在遍历过程中只需要遍历该范围内的索引即可，姑且称之为 RangeFilter；

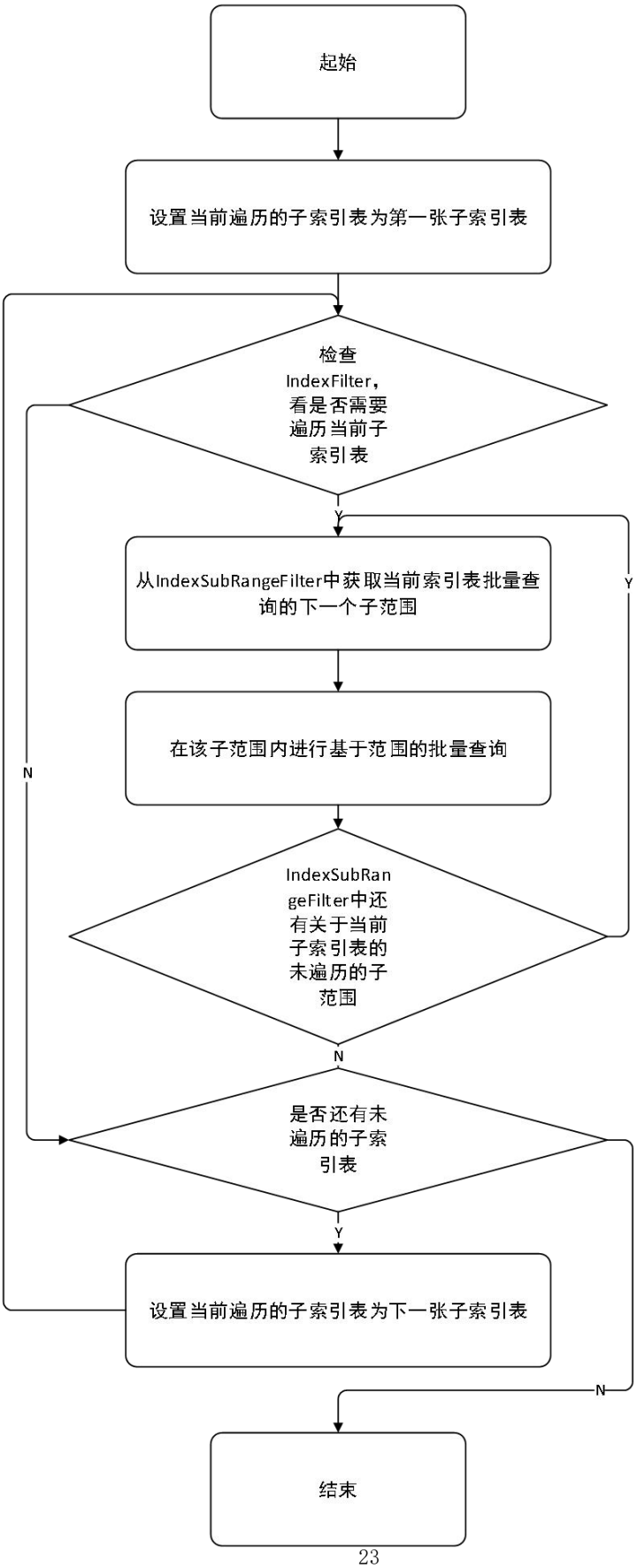
（3）更进一步，在更新每个子索引表的索引信息的时候，分别记录这些子索引表中与该 LogUnit 相关的子索引范围，在遍历该子索引表查找所有匹配该 LogUnit 的索引的时候只需要遍历该子索引表在该子索引范围内的索引即可，姑且称之为 IndexRangeFilter；

（4）更进一步，在 Redo 线程进行索引合并（immutable 索引表合并到 base 索引表）的时候，进一步将子索引表内的子索引进行细化，划分为更多子范围，姑且称之为 IndexSubRangeFilter；

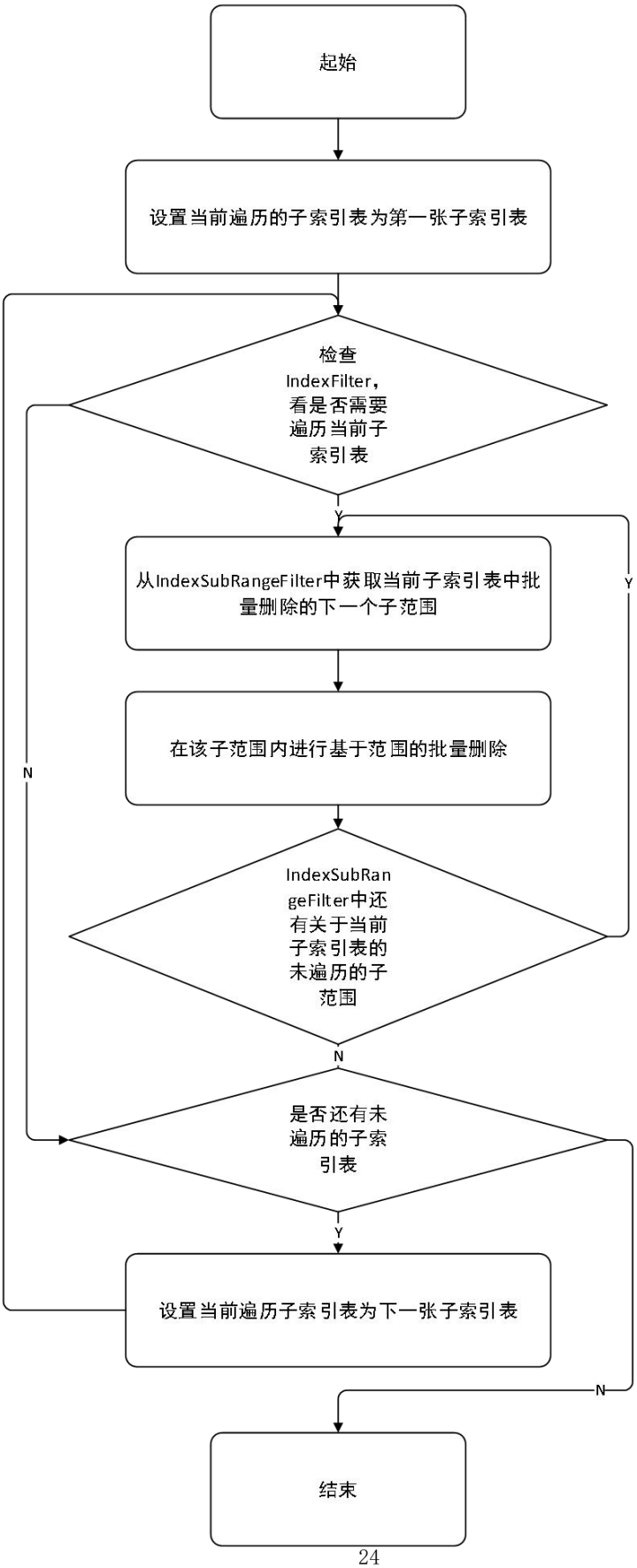
（5）提供基于范围的批量查询和批量删除接口；

改进后的批量查询和批量删除方案流程如下：

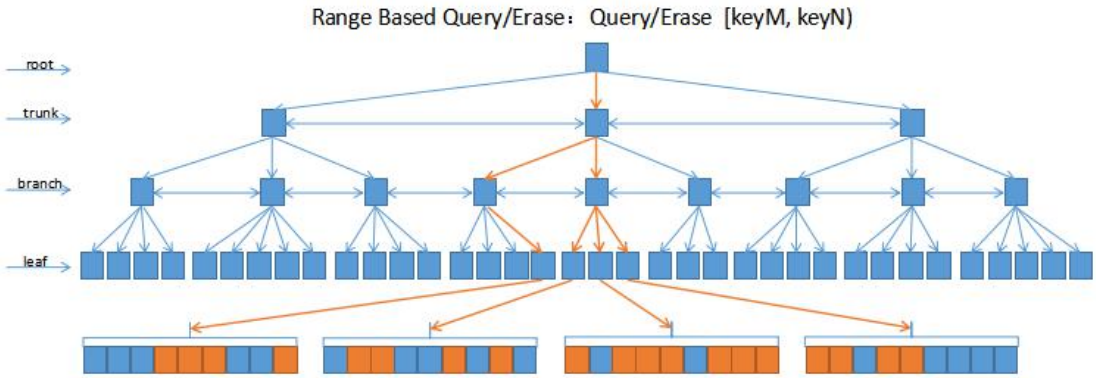
改进后批量查询


起始
设置当前遍历的子索引表为第一张子索引表
检查IndexFilter，看是否需要遍历当前子索引表
Y
从IndexSubRangeFilter中获取当前索引表批量查询的下一个子范围
在该子范围内进行基于范围的批量查询
IndexSubRangeFilter中还有关于当前子索引表的未遍历的子范围
Y
N
是否还有未遍历的子索引表
Y
设置当前遍历的子索引表为下一张子索引表
N
N
结束

改进后批量删除


起始
设置当前遍历的子索引表为第一张子索引表
检查IndexFilter，看是否需要遍历当前子索引表
Y
从IndexSubRangeFilter中获取当前子索引表中批量删除的下一个子范围
在该子范围内进行基于范围的批量删除
IndexSubRangeFilter中还有关于当前子索引表的未遍历的子范围
Y
N
是否还有未遍历的子索引表
Y
设置当前遍历子索引表为下一张子索引表
N
N
结束

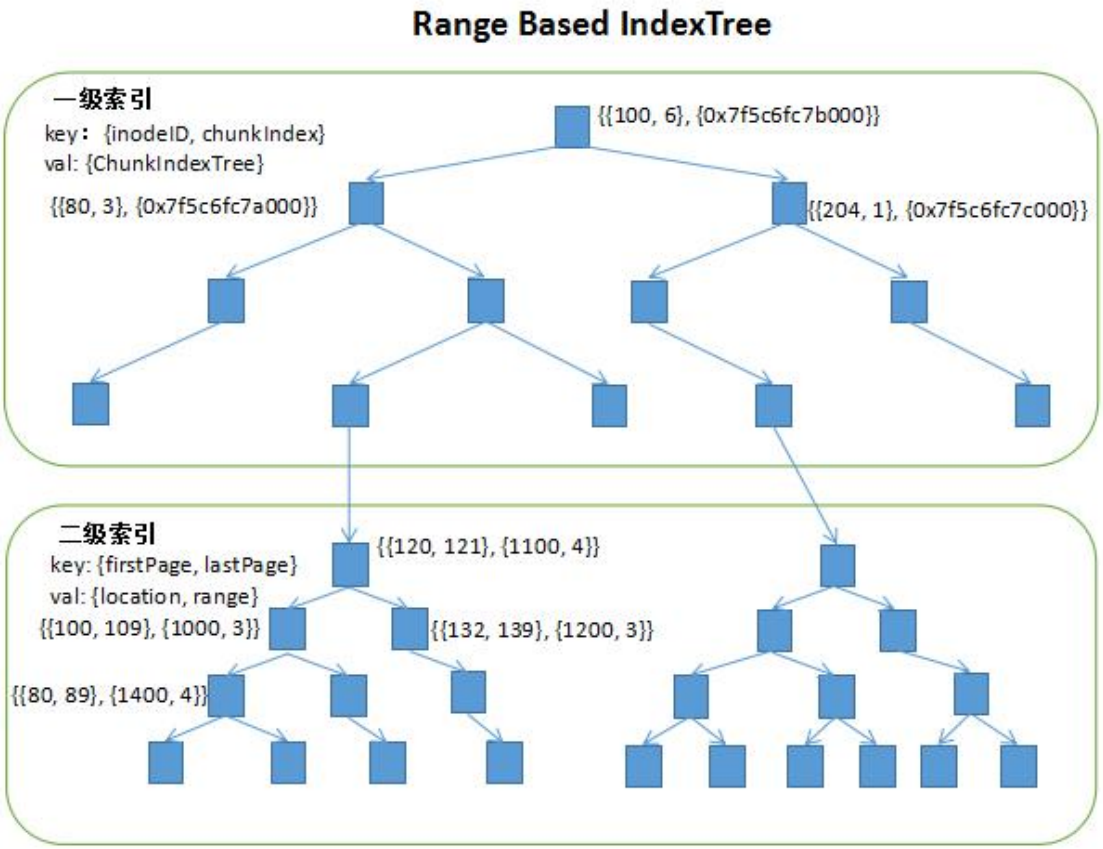
基于范围的批量查询/删除接口优化：



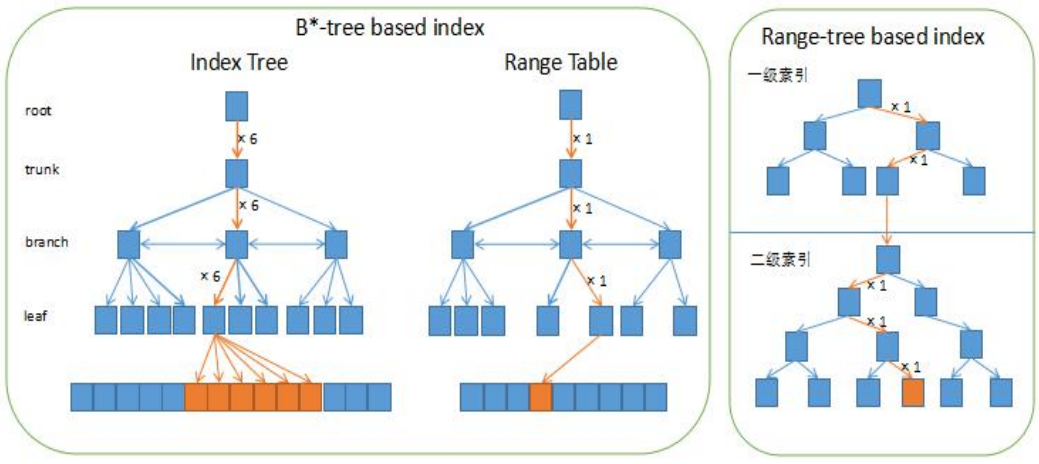
关于索引优化还有一些其他的小优化，主要包括：Redo 线程替代写线程去回收 LogUnit；采用区间树去管理 Mutable 索引区和 Immutable 索引区。

写线程在复用 LogUnit 的时候需要自己去删除索引表中关于该 LogUnit 的所有索引，纵然经过上述优化，该操作仍然比较耗时，且在该操作执行过程中写线程是不能执行 IO 的，显然这是不可接受的，于是我们将该操作转嫁给 Redo 线程，由 Redo 线程进行 LogUnit 的回收，确保写线程在需要写缓存空间且 SSD 中的确有可用的写缓存空间时，直接使用之，这样使得写线程专注于 IO，间接提升写性能。但是相比于写线程在“不得已”情况下执行按需复用/按需删除即将复用的 LogUnit 的索引，Redo 线程需要未雨绸缪，事先删除关于某个 LogUnit 的索引，这样以来，对于读请求在 SSD 写缓存中的命中率是有一些负面影响的。在未来。这是一个可以持续优化的方向。

当前 Mutable 索引区、Immutable 索引区和 Base 索引区都采用 B*树去管理，索引区中的每一个索引都只能表示唯一的一个 page 的索引，对于同一个写请求中包含连续多个页的情况，需要在索引区中逐一更新这些页，如果能批量更新这些连续的页，无论从内存消耗上还是 CPU 使用上都会更有效率，而区间树可以较好的满足这一要求，事实上，区间树不仅在批量更新上更有效率，在批量查询和批量删除上也都比较高效（但是区间树会占用更多内存）。当前只针对 Mutable 索引区和 Immutable 索引区进行了改进，修改为基于区间树的索引。具体来说，基于区间树的索引是一个两级索引，第一级索引是关于{inodeID，chunkIndex}到{ChunkIndexTree}的映射，第二级索引则是关于{firstPage，lastPage}到{location，range}的映射，如下：



假如关于{inode: 10, chunkIndex: 5}的写请求所在的页范围是[1000, 1005]，数据存放的起始位置是 SSD 上第 10000 个页，压缩后数据占用 3 个页，那么，在基于 B*树的索引中，采用两棵树去记录该索引信息，一棵树（Index Tree）中记录[1000, 1005]区间内每个页对应的索引信息，另一棵树（Range Table）中记录这些连续的数据页在 SSD 上实际占据的页的个数，而在基于区间树的索引中，采用两级索引去记录该索引信息，第一级索引记录的是关于{inode: 10, chunkIndex: 5}的索引树的位置，第二级索引记录的是所有率属于{inode: 10, chunkIndex: 5}这一 Chunk 的页的索引信息，其中包含一条记录{key: {1000，1005}, val: {10000, 3}}，更新索引示意图如下：



在基于 B*树的索引中，更新 Index Tree 时需要执行 6 次从 root 到 leaf 的查找，每次在

找到的 leaf 中插入一条记录，最后更新 Range Table 时也需要执行 1 次从 root 到 leaf 的查找，并在找到的 leaf 中插入一条记录，或者更新一条记录。

在基于区间的索引中，首先在一级索引中找到相应的 Chunk 对应的二级索引，然后在二级索引中插入这些页的索引信息，该过程中相同的路径只需查找 1 次，插入过程只需执行一次（但是插入之前可能存在删除操作或者分裂操作，但是删除或者分裂操作无需再次查找，可以充分利用当前迭代器找到需要删除或者分裂的元素）。

索引优化总结：

对比项	改进前	改进后	改进后影响
表数目	不分表，单张索引表	分表，多张索引表	正面影响：每张表操作的索引数目变少，查询/更新/删除过程中需要比较的元素变少，更新/删除过程中需要移动的元素变少
批量查询/删除	批量操作需要遍历整张表	批量操作支持基于范围的过滤器，支持基于范围的查询/删除	正面影响：减少批量查询/删除过程中遍历的元素个数，缩短批量查询/删除过程所消耗的时间
复用 LogUnit 时批量删除索引	由写线程执行	由 Redo 线程执行	正面影响：写线程无需消耗额外时间执行批量删除，专注于 IO 负面影响：read 命中率可能会降低
Mutable/Immutable 索引管理	基于 B*树	基于区间树	正面影响：提高索引插入和查询效率，尤其是批量插入和批量查询的效率

4.3.3 Read IO 未命中情况下性能较低问题解决之道

Read IO 在未命中 SSD 的情况下，需要去后端读取 HDD，读性能就受限于 HDD 读性能，但是在测试中发现，在 Read IO 未命中 SSD 的情况下，从 HDD 读取的性能比理论上 HDD 的读性能要差，考虑到 Redo 线程也在不停地写 HDD，在 Read IO 未命中的情况下，HDD 上的 IO 是混合读写，Read IO 和 Redo IO 会相互影响，但是 Read IO 是应用 IO，Redo IO 是系统内部 IO，可以一定程度上牺牲 Redo 性能来保障 Read IO，于是采用了 Redo IO 限流机制。

系统在运行过程中会根据 Read 线程访问 HDD 的情况判断是否需要在 Redo 线程中限流，那么哪些情况下需要限流？又该如何限流呢？下面逐一讲解。

系统在运行过程中，每隔一个时间周期统计一次在过去这段时间内有多少 Read IO 由 HDD 提供，如果由 HDD 提供的 Read IO 的数目少于预先设定的阈值，则认为 Read 线程对 HDD 的访问需求不高，记录当前无需为 Read 线程限流，否则记录当前需要为 Read 线程限流，但是因为每一个统计周期都比较短，在每一个统计周期内都可能改变限流状态，从而导致 Redo 线程在需要限流和无需限流之间来回抖动，所以对无需为 Read 线程限流的情形进行了约束，修改为如果由 HDD 提供的 Read IO 的数目少于预先设定的阈值且自从上次需要为 Read 线程限流的记录以来经历了一定数目个统计周期，则记录当前无需为 Read 线程限流，这样保证从无需限流状态可以很快切换到需要限流状态，从需要限流状态切换到无需限流状态则至少

需要经历一定数目个统计周期。

关于 Redo IO 限流，采用的是闭源限流，即在 Redo 线程中负责产生 Redo IO 的入口处进行限流，具体来讲就是在处理 Redo 事件的入口和 Redo IO AIOCallback 触发新的 Redo IO 之前分别检查 Redo 线程是否需要为 Read 线程限流，如果需要限流就不产生 Redo IO。

4.3.4 版本号更新问题解决之道

OSS 中每一次数据写操作都会更新相应 Chunk 的版本号，即 WriteVersion，在 Redo 的时候需要将该 Chunk 的 WriteVersion 更新到 HDD 中，且要确保 WriteVersion 持久化到 HDD 上，当前采用的是批量更新（在关于当前被 Redo 的 LogUnit 的所有的数据都 Redo 之后，将所有这些 Redo 的数据所涉及到的 Chunk 的 WriteVersion 更新到 HDD）加上同步文件系统的方式。关于每一个 Chunk 的 WriteVersion 的更新都涉及两次随机 IO：从 HDD 中读取 WriteVersion 所在的数据页，以及更新 WriteVersion 信息到该数据页并写入 HDD。众所周知，HDD 随机访问 IOPS 非常低（100 - 300 不等），而需要更新 WriteVersion 的 Chunk 数目最多可达 8192 个（当前 LogUnit 大小为 32M 的情况下），更新 WriteVersion 过程可能会花费数秒甚至数十秒！鉴于此，迫切需要寻找版本号更新问题的其他方案。

首先想到的是采用更高性能的存储介质（比如 SSD）去存储 WriteVersion，但是一方面考虑到成本因素，另一方面考虑到当前 OSS 实现中 Chunk 的 WriteVersion 是带内存放的（WriteVersion 和数据逻辑上连续存放），如果将 WriteVersion 单独存放于高性能介质中，需要修改现有的 WriteVersion 相关的逻辑。

其次想到的是借助支持高性能随机读写的第三方 KV 存储来专门维护 WriteVersion 信息，因为更新 Chunk 的 WriteVersion 信息是随机写密集型 IO，根据之前对 LevelDB 的了解，这恰恰是 LevelDB 所擅长的，所以选择 LevelDB，后经测试验证，WriteVersion 更新效率的确有所改观。

4.3.5 写缓存空间不足问题解决之道

当前系统在写缓存空间不足的情况下需要等待可用的 LogUnit 方可写入，而可用的 LogUnit 只能通过 Redo 线程来回收，回收过程需要确保数据落盘，同时需要确保 WriteVersion 落盘，然后在真正复用该 LogUnit 之前还需要删除该 LogUnit 相关的所有索引，整个等待可用 LogUnit 的过程可能会多达数秒，在该期间写线程是不能执行 IO 的，对于许多业务来说，这都是不可接受的！

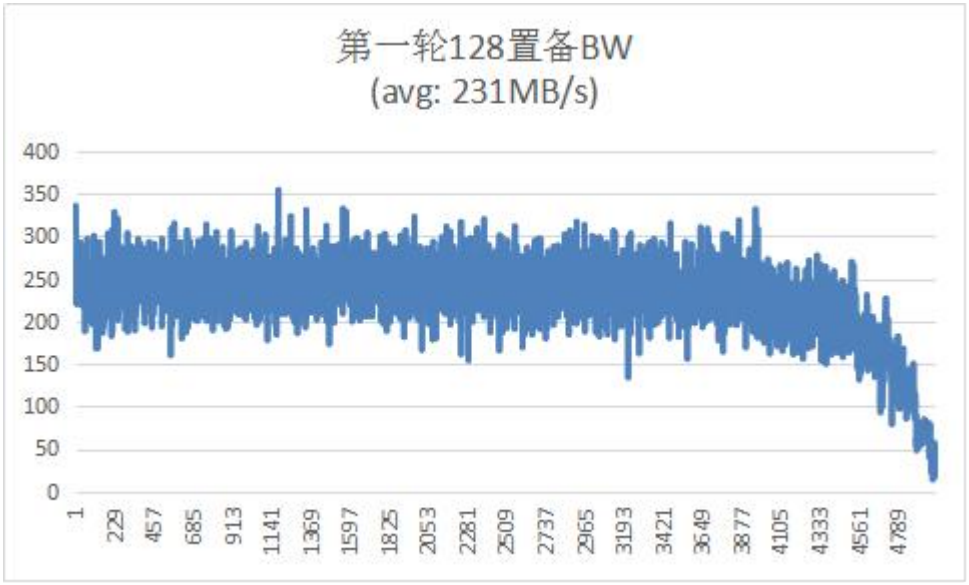
SSD 作为写缓存盘，是用于 IO 加速的，既然 SSD 暂时没空间了，在后端的 HDD 还有可用空间的情况下，可以不做 IO 加速，但是绝不可以影响正常 IO。所以采取的方案是在 SSD 空间不足情况下绕过 SSD 写缓存（ByPass SSD），直接将数据写到 HDD 上，等到 SSD 上逐渐释放出一定数目的可用的 LogUnit 之后，再恢复到先写 SSD 后 Redo 到 HDD 模式。

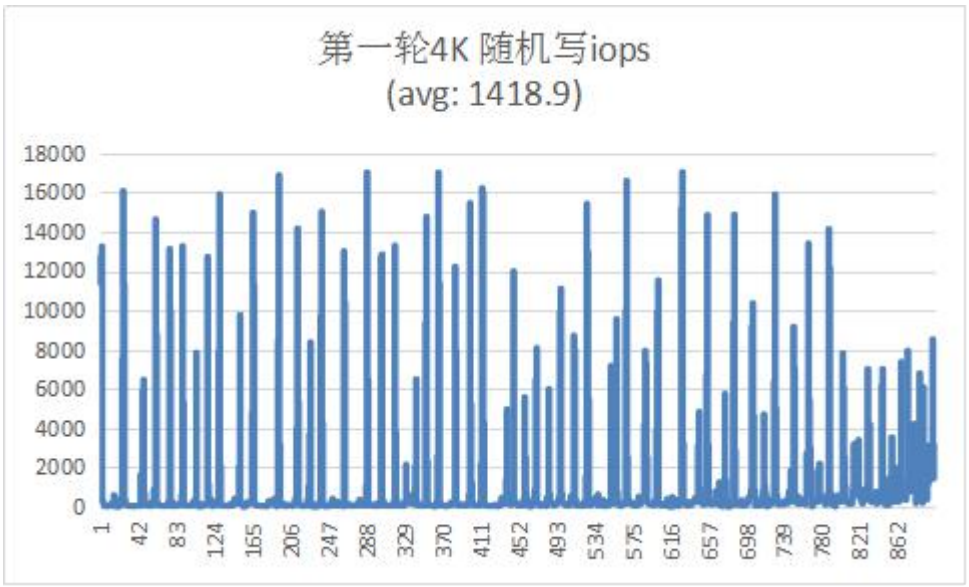
另外，在 ByPass SSD 期间，Redo 线程可能也在执行 Redo IO，和应用 IO 之间形成 IO 资源竞争，从而对应用 IO 造成影响，为了尽可能减小 Redo IO 对应用 IO 的影响，也会对 Redo 线程执行限流。

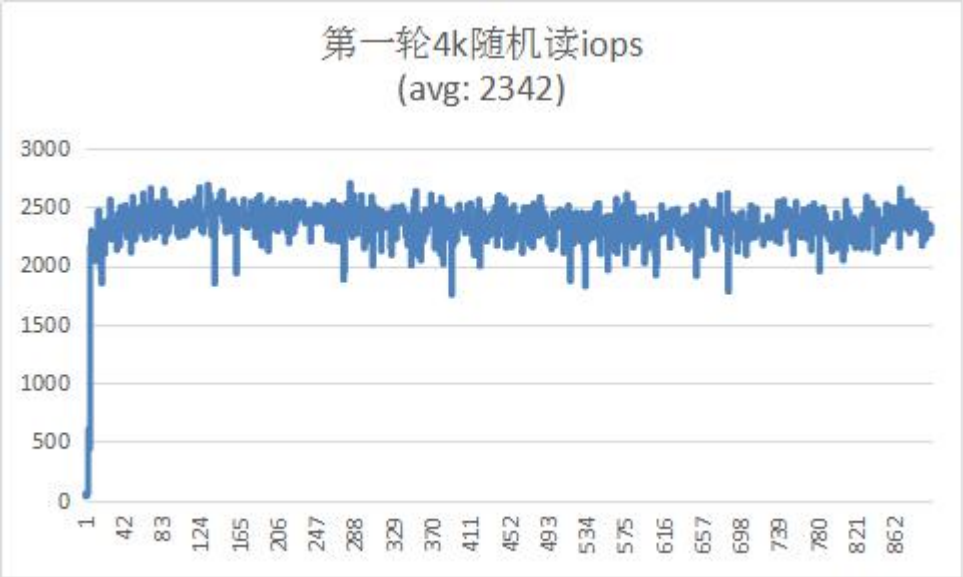
4.4 技术可行性测试报告

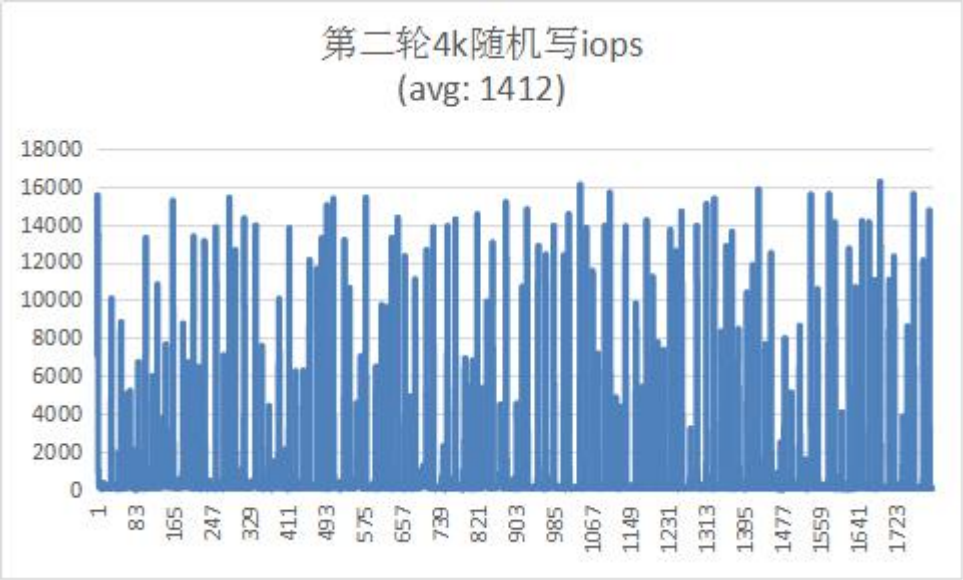
关于测试结果，将从如下两个方面展开：性能优化前后对比测试和对后续工作具有指导意义的性能测试。其中，性能优化前后对比测试包括：升腾测试用例下的绝对性能测试；重放（Redo）速度测试；SSD 写缓存空间不足情况下性能测试；基于区间树的索引性能测试。对后续工作具有指导意义的性能测试包括：索引删除对读性能的影响；hash 方式对性能的影响。

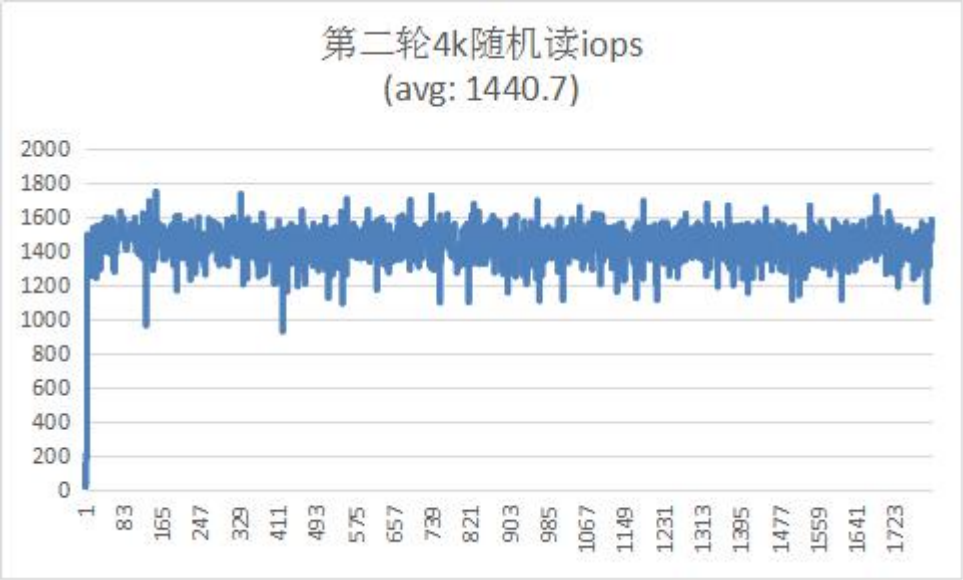
（1）升腾测试用例虚拟化环境绝对性能测试

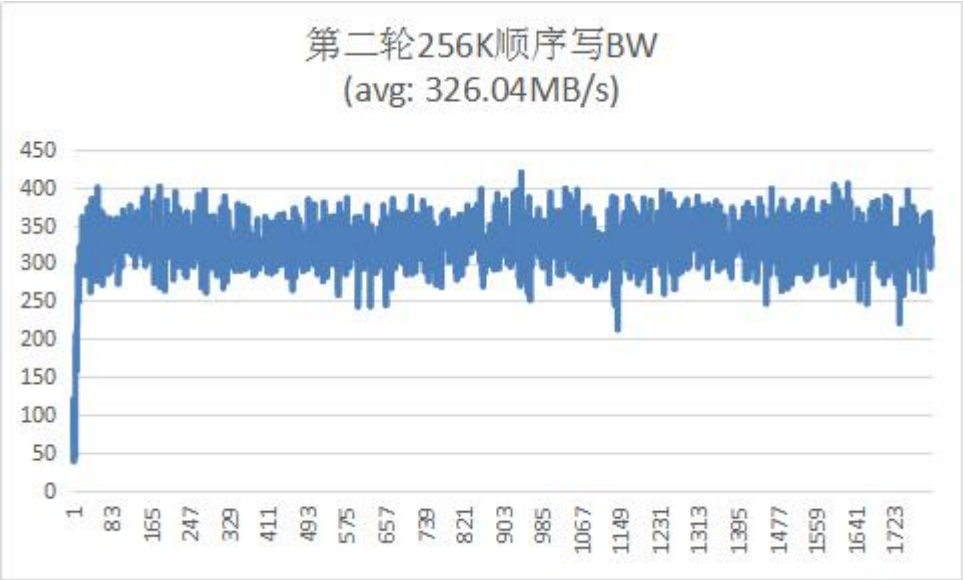


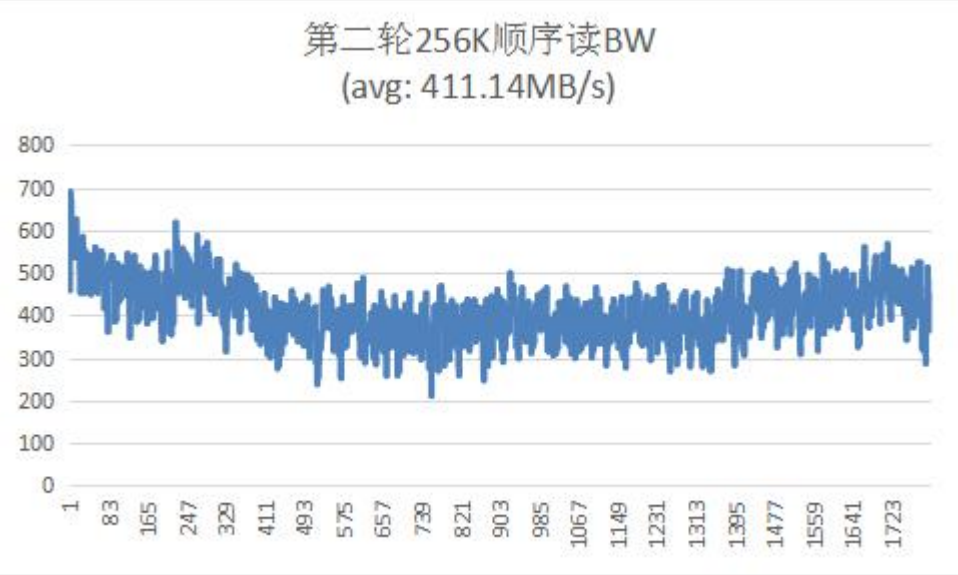


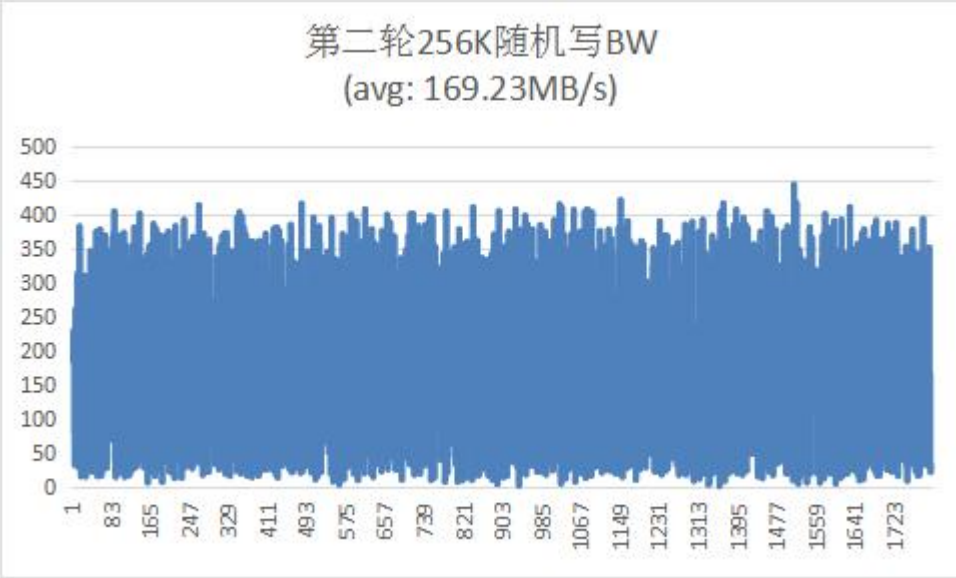

第一轮4k随机读iops
(avg: 2342)
3000
2500
2000
1500
1000
500
0
1
42
83
124
165
206
247
288
329
370
411
452
493
534
575
616
657
698
739
780
821
862

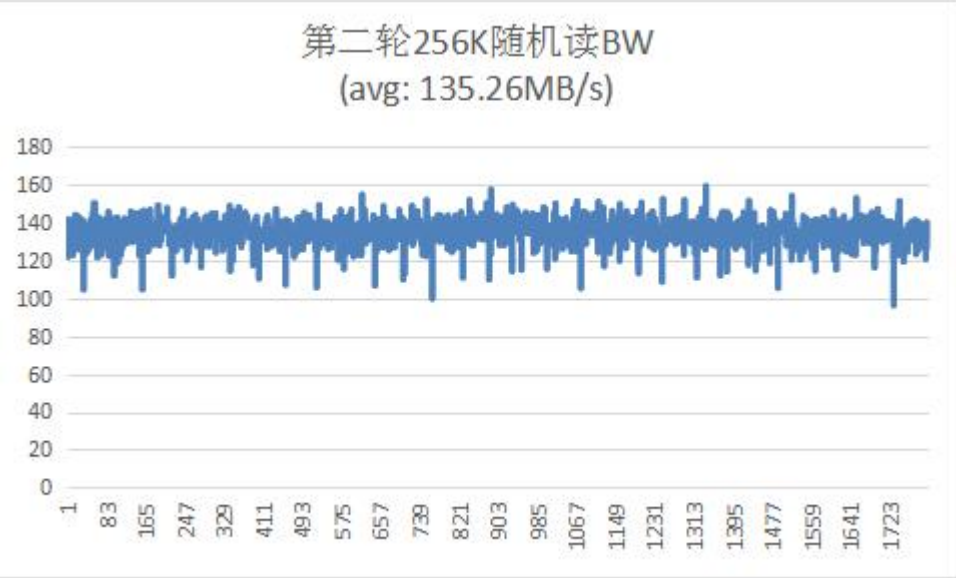

第二轮4k随机写iops
(avg: 1412)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723

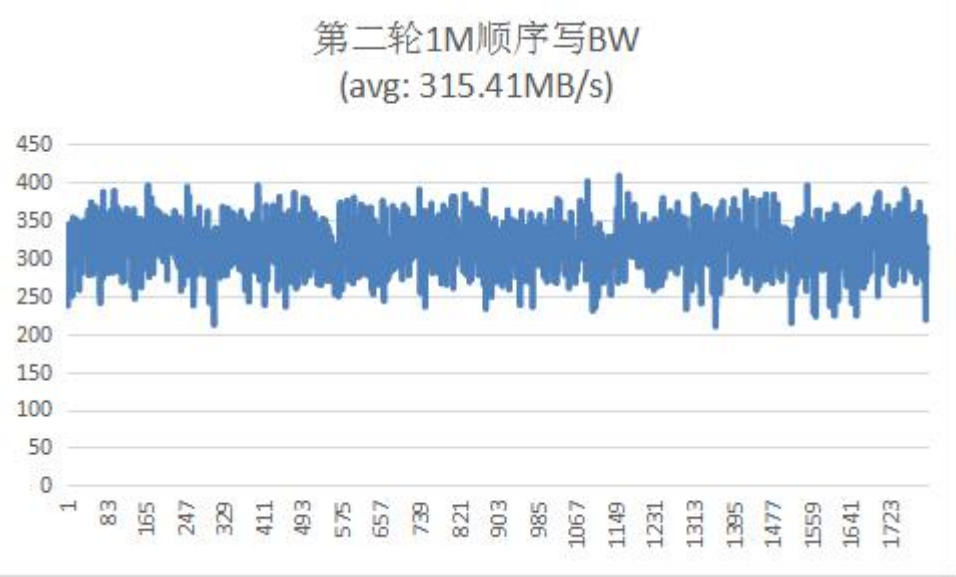

第二轮4k随机读iops
(avg: 1440.7)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723

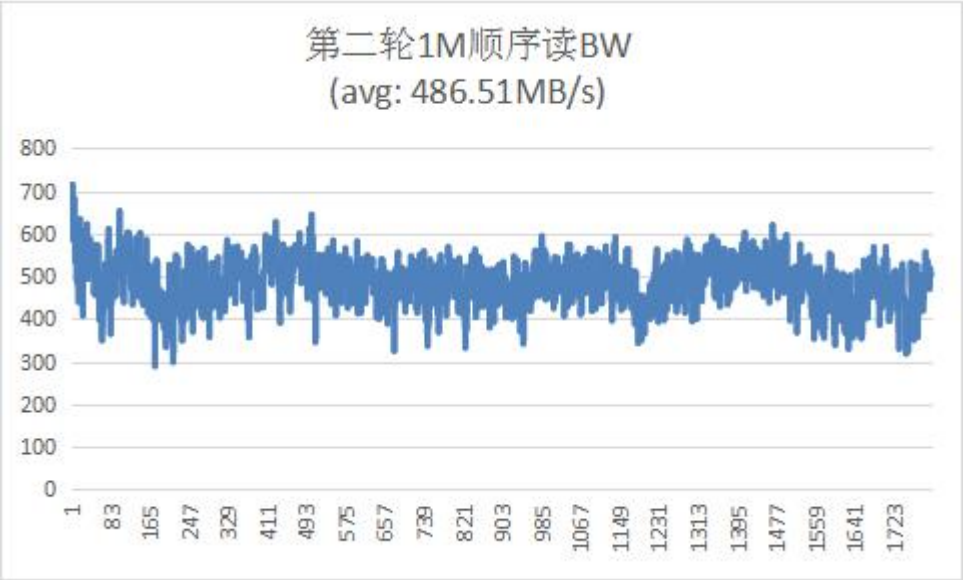

第二轮256K顺序写BW
(avg: 326.04MB/s)
450
400
350
300
250
200
150
100
50
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723


第二轮256K顺序读BW
(avg: 411.14MB/s)
800
700
600
500
400
300
200
100
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723


第二轮256K随机写BW
(avg: 169.23MB/s)
500
450
400
350
300
250
200
150
100
50
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723


第二轮256K随机读BW
(avg: 135.26MB/s)
180
160
140
120
100
80
60
40
20
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723


第二轮1M顺序写BW
(avg: 315.41MB/s)
450
400
350
300
250
200
150
100
50
0
1
83
165
247
329
411
493
575
657
739
821
903
985
1067
1149
1231
1313
1395
1477
1559
1641
1723



改进前后，在升腾测试用例下，虚拟机环境测试性能结果对比：

测试项目		改进前	改进后
第一轮	128K 置备 BW	38.09 MB/s(测试可能会中断)	231 MB/s
	4k 随机写 IOPS	374.73(测试可能会中断)	1418.9
	4k 随机读 IOPS	420.73(测试可能会中断)	2342
第二轮	4k 随机写 IOPS	测试中断无法进行	1412
	4k 随机读 IOPS	测试中断无法进行	1440.7
	256k 顺序写 BW	测试中断无法进行	326.04 MB/s
	256k 顺序读 BW	测试中断无法进行	411.14 MB/s
	256k 随机写 BW	测试中断无法进行	169.23 MB/s
	256k 随机读 BW	测试中断无法进行	135.26 MB/s
	1m 顺序写 BW	测试中断无法进行	315.41 MB/s
	1m 顺序读 BW	测试中断无法进行	486.51 MB/s

（注：测试节点为浪潮 4U 环境）

改进前，虚拟化环境下测试之所以会中断，是因为某些 IO 长时间没有响应所致，但是在改进之后，经过多次测试，都未出现测试中断的现象，而通过对比改进前后在第一轮测试中的结果可以明显看到改进后带来几倍的性能提升，而这些性能提升是分类索引、索引分区、基于范围的批量查询/删除、Redo 限流、ByPass SSD（写缓存空间不足情况下）等综合作用的结果，所以从该角度上来讲，改进工作是卓有成效的。

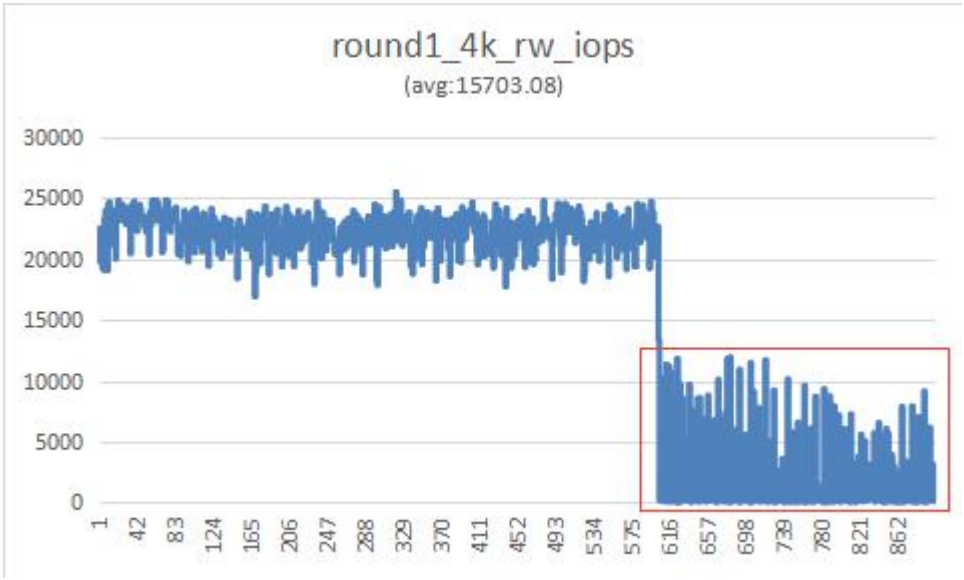
（2）重放速度

关于重放速度的改进，暂无测试数据，听听解决方案工程师（全程参与升腾测试，做了不下数十次测试，重放速度直接决定了解决方案工程师完成一次测试需要多长时间，所以对于重放速度这块解决方案工程师是深有感触的）怎么说的：

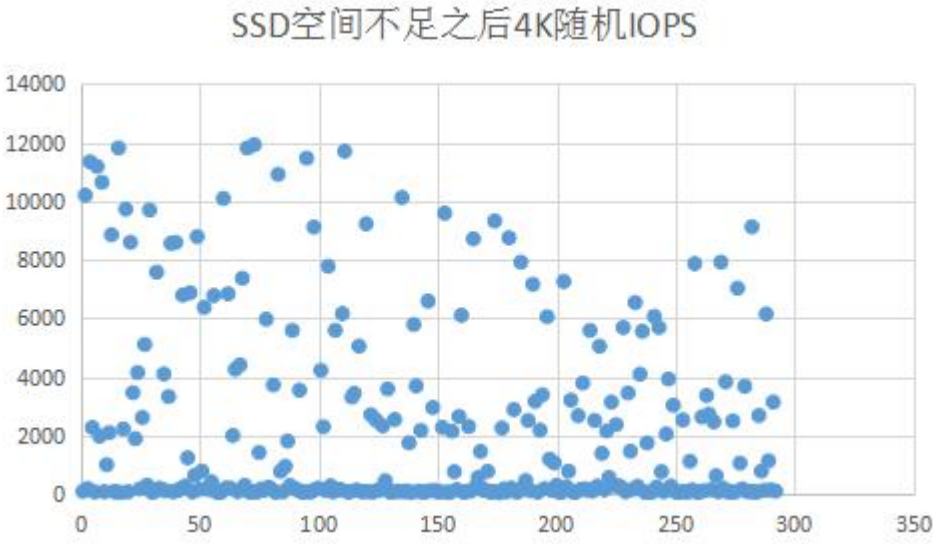
“改进之前，第一轮测试之后需要等待 30+小时方可进行第二轮测试（SSD 写缓存中数据全部 Redo 完毕），改进之后，这个时间缩短到 10 小时左右。”，由此可见，

借助 LevelDB 来存放 WriteVersion 对于重放速度的提升还是卓有成效的。

（3）SSD 写缓存空间不足情况下性能测试

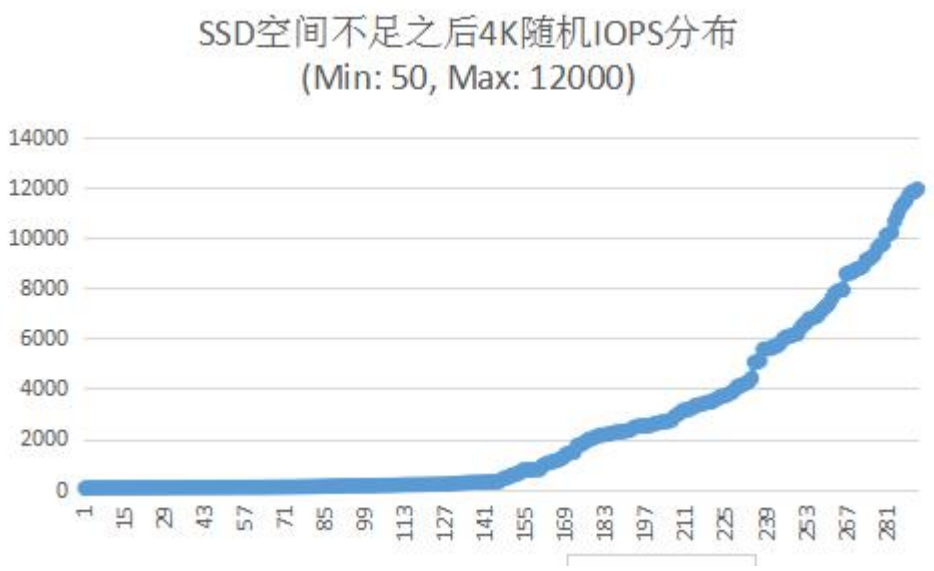


在第 610 个统计周期之后，发生了 SSD 写缓存空间不足的情况，4K 随机写 IOPS 相比之前下降非常大，为了更直观的分析，将第 610-900 个统计周期的性能数据单独在一个图表中展示（按时序关系展示），如下：



从上图可以看出，IO 波动很大，这是因为当检测到 SSD 写缓存空间不足时，到达的写 IO 会直接写入 HDD，HDD 的随机写性能较低，所以表现出的 4K 随机 IOPS 较低，在此期间 Redo 线程正在不遗余力的回收 SSD 写缓存空间，当有可用的 SSD 写缓存空间之后，到达的写 IO 则切换到先写 SSD 的模式，4K 随机 IOPS 则大幅提升，SSD 写缓存空间被很快写满，再次出现 SSD 写缓存空间不足的情况，到达的写 IO 又切换到 ByPass SSD 直写 HDD 的模式，性能又降低到较低水平，如此以往。

SSD 写缓存空间不足情况下 4K 随机写 IOPS 分布情况如下：



从 4K IOPS 分布图（注：不是累积分布）可以看出，在 SSD 写缓存空间不足情况下，4K 随机写 IOPS 最小值为 50，最大值为 12000 左右，将近一半的统计周期内 IOPS 是小于 1000 的（4 块 HDD 盘的随机 4K IOPS 在 1000 左右），但是相较于改进前在 SSD 写缓存空间不足情况下发生大面积的 IO 为 0 的情况不再出现。

（4）基于区间树索引性能测试结果

区间树索引的目的在于加速索引更新和查询操作，为了验证改进效果，进行了基于区间树的（Mutable/Immutable）索引和基于 B*树的（Mutable/Immutable）索引性能对比测试，主要从以下几个方面进行测试：

A. 大粒度更新，大粒度查询
   a) 64K 粒度更新，64K 粒度查询
   b) 128K 粒度更新，128K 粒度查询
   c) 256K 粒度更新，256K 粒度查询
   d) 1M 粒度更新，1M 粒度查询

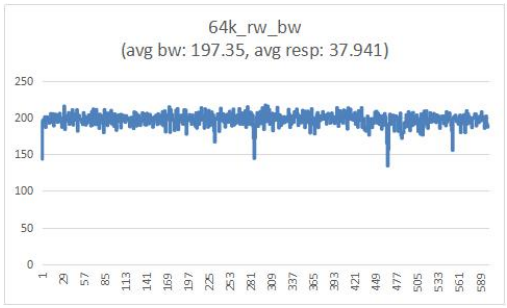
B. 大粒度更新，小粒度查询
   a) 64K 粒度更新，4K 粒度查询

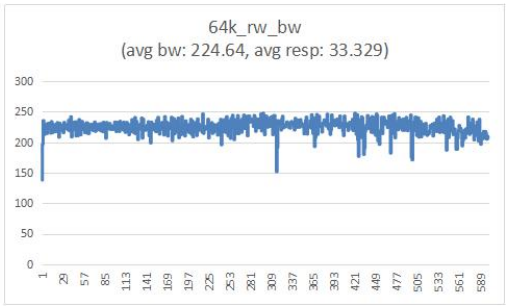
C. 小粒度更新，大粒度查询
   a) 4K 粒度更新，32K 粒度查询
   b) 4K 粒度更新，64K 粒度查询
   c) 4K 粒度更新，128K 粒度查询
   d) 4K 粒度更新，256K 粒度查询
   e) 4K 粒度更新，1M 粒度查询

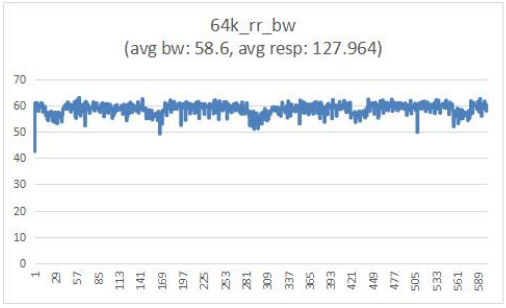
D. 小粒度更新，小粒度查询
   a) 4K 粒度更新，4K 粒度查询

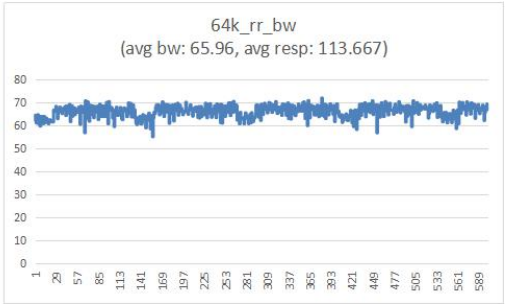
64K 粒度更新，64K 粒度查询

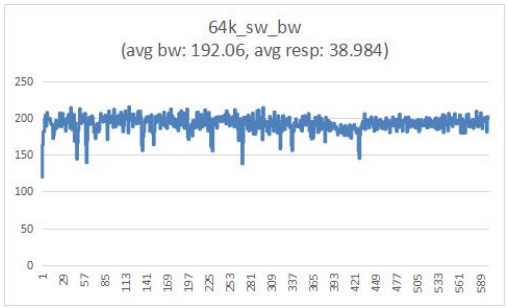
B*树索引 **区间树索引**

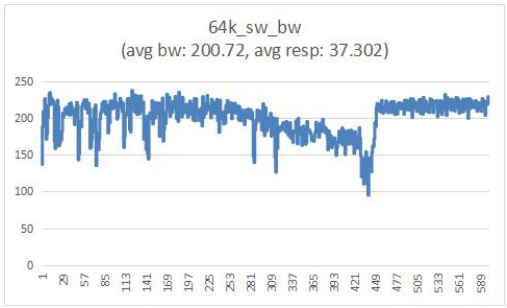


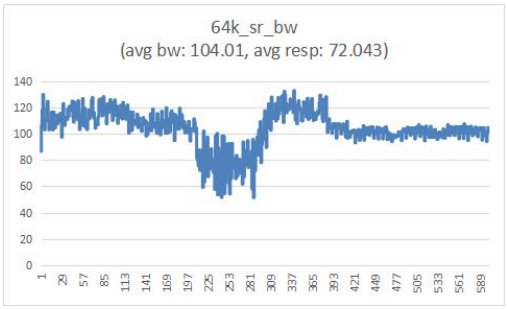


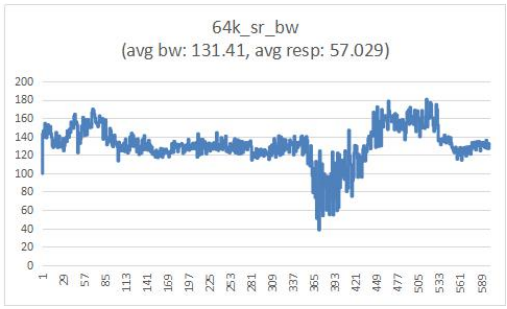






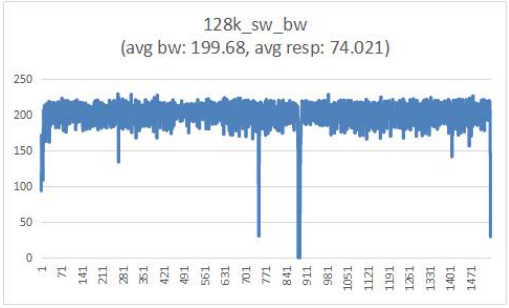




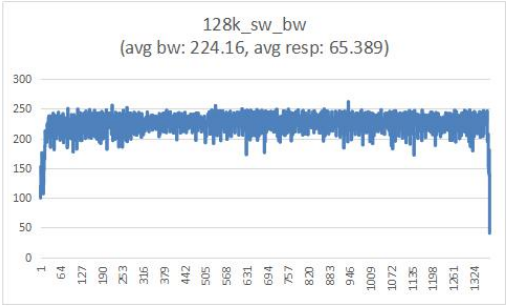


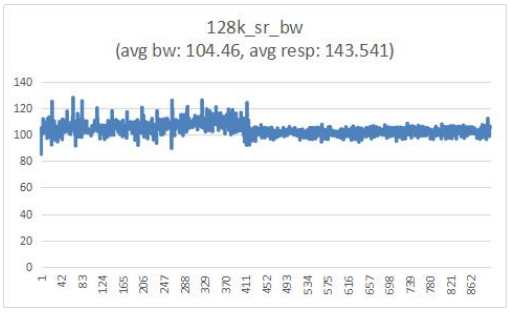
128K 粒度更新，128K 粒度查询

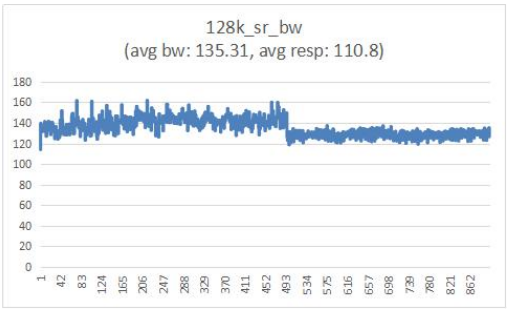
B*树索引



区间树索引



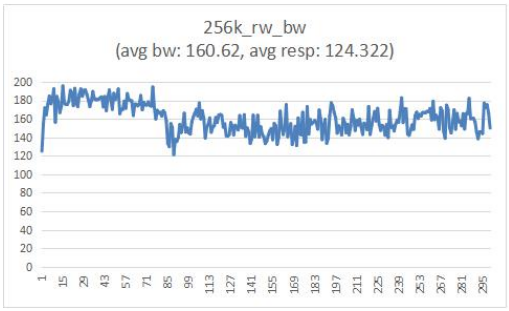


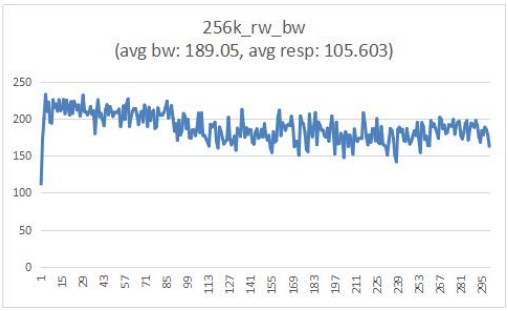


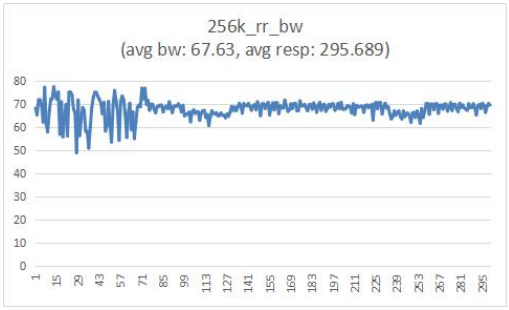
256K 粒度更新，256K 粒度查询

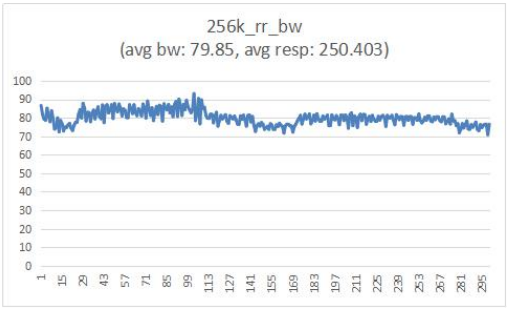
B*树索引

区间树索引





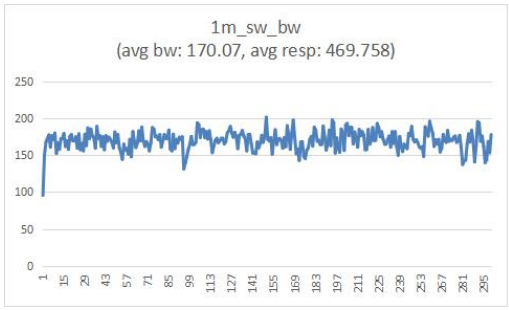


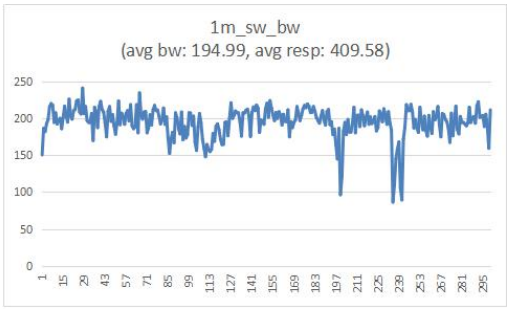


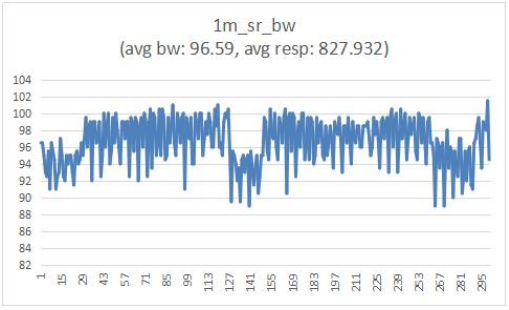
1M 粒度更新，1M 粒度查询

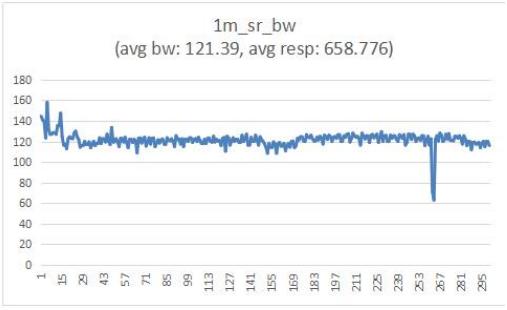
B*树索引

区间树索引



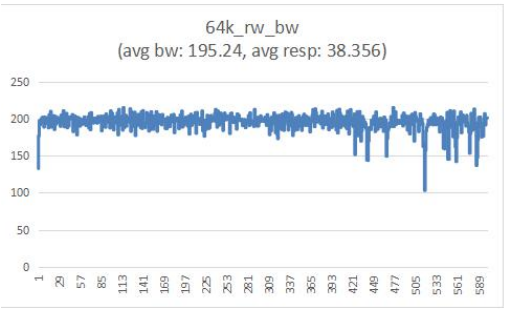


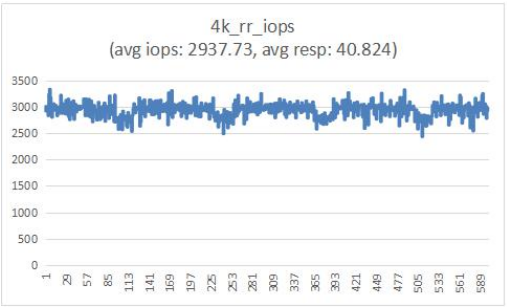




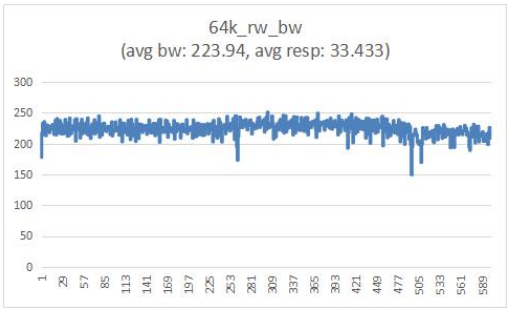
64K 粒度更新，4K 粒度查询

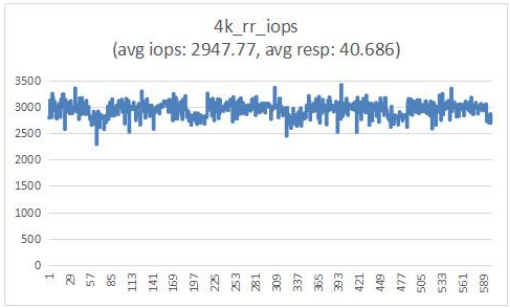
B*树索引





区间树索引

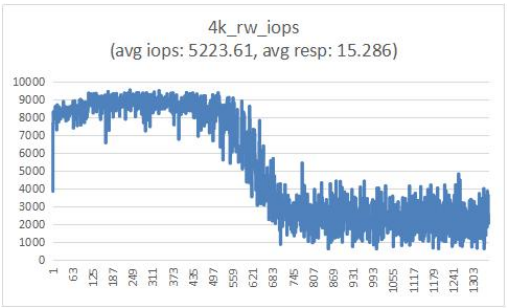




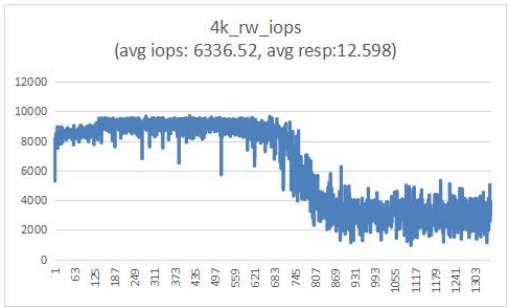
4K 粒度更新，{4K，32K，128K，256K，1M}粒度查询

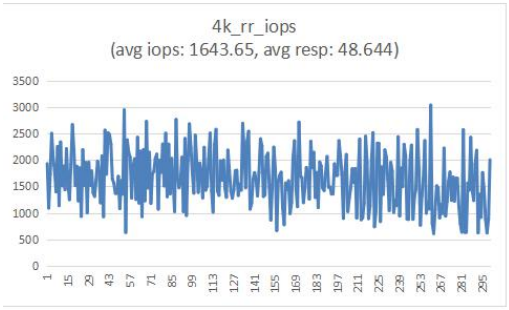
（注：因为测试数据集不同的缘故，4K 粒度更新，64K 粒度查询单独列出）

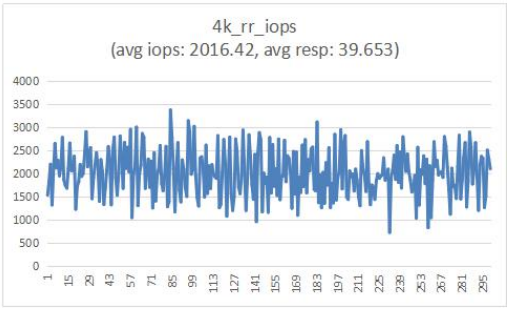
B*树索引

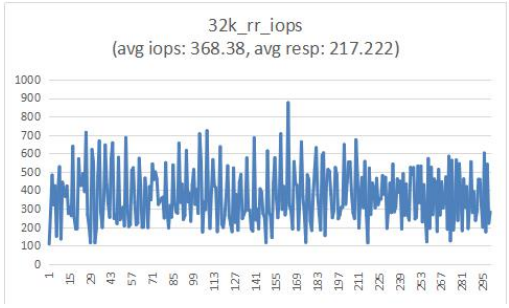


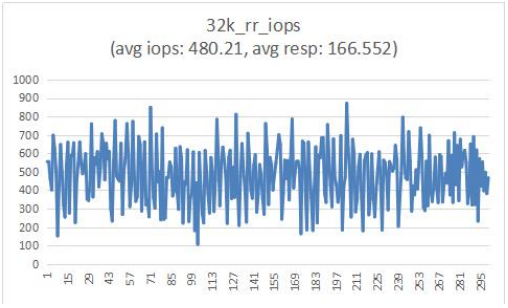
区间树索引

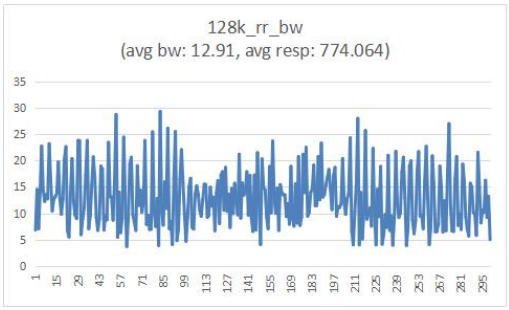


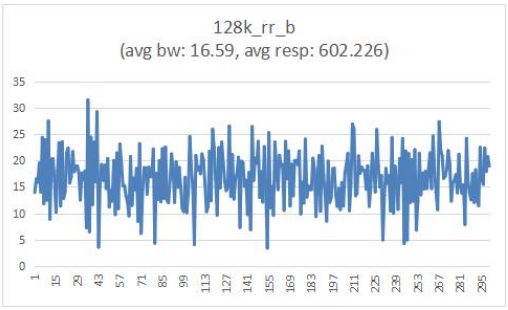


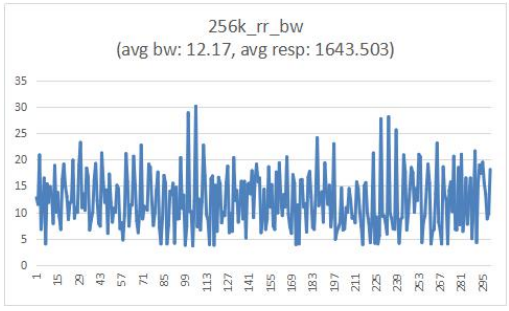


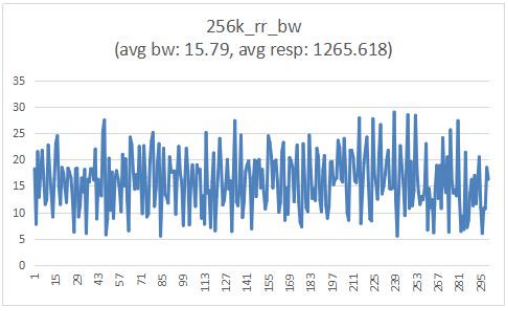


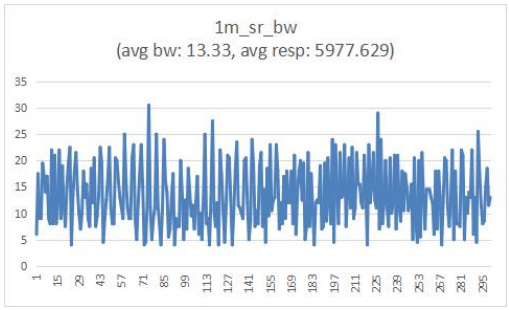


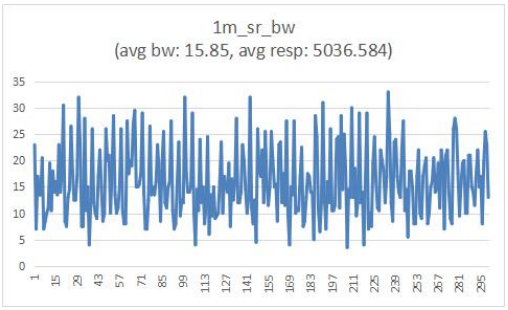








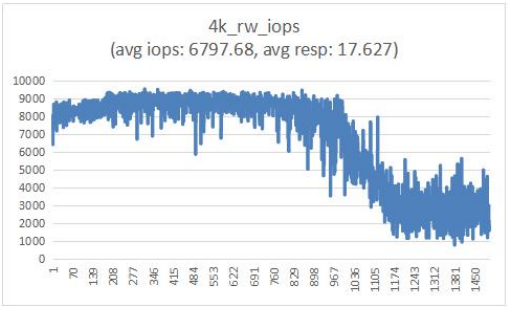


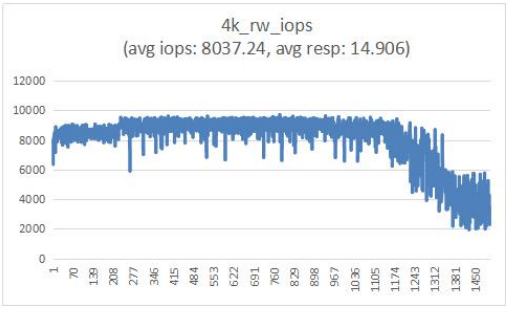


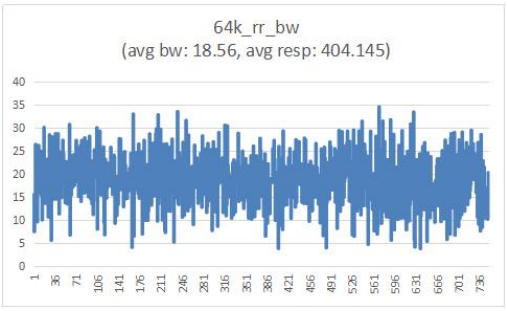
4K 粒度更新，64K 粒度查询

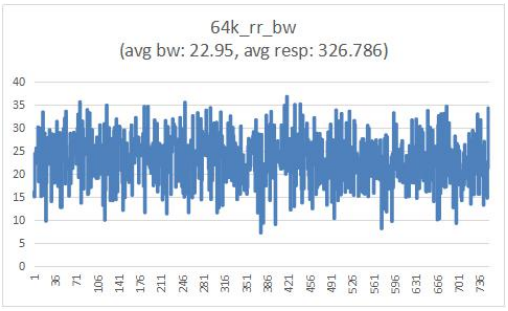
B*树索引

区间树索引









各测试结果汇总：

更新粒度	查询粒度	B*树索引				区间树索引				性能提升(%)	
		更新		查询		更新		查询			
		IOPS	BW(MB/s)	IOPS	BW(MB/s)	IOPS	BW(MB/s)	IOPS	BW(MB/s)	更新	查询
4K	4K	5223.61		1643.65		6336.52		2016.42		21	22
4K	32K	5223.61		368.38		6336.52		480.21		21	30
4K	64K	6797.68			18.56	8037.24			22.95	18	23
4K	128K	5223.61			12.91	6336.52			16.59	21	31
4K	256K	5223.61			12.17	6336.52			15.79	21	29
4K	1M	5223.61			13.33	6336.52			15.85	21	18
64K	4K		195.24	2937.73			223.94	2947.77		14	~0
64K	64K		197.35		58.6		224.64		65.96	13	12
128K	128K		199.68		104.46		224.16		135.31	12	29
256K	256K		160.62		67.63		189.05		79.85	17	18
1M	1M		170.07		96.59		194.99		121.39	14	25

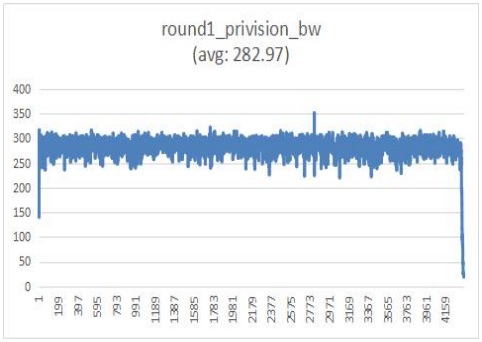
从汇总结果来看， 区间树版本的索引在所有测试用例下，性能上完全占据优势（除了 64K 粒度更新情况下，4K 粒度查询性能几无提升之外），无论是大粒度更新还是小粒度更新，无论是大粒度查询还是小粒度查询，性能都要比 B*树版本高出 10+百分点，证明了区间树索引是有效果的。

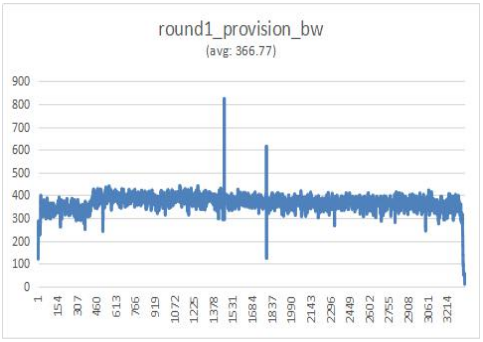
（5）索引删除对读性能的影响

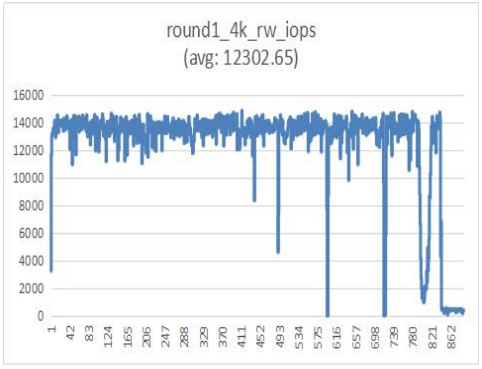
如前文所述，由于 SSD 空间有限，持续的数据写入必然导致 SSD 空间复用，从而导致索引被删除，索引删除必然导致某些读请求在 SSD 写缓存中无法命中，从而降低读性能，那么索引删除对读性能的影响几何呢？我们做了两组对比测试，一组是执行 Redo 和不执行 Redo 进行对比（执行 Redo 会删除索引，不执行 Redo 则不会删除索引），另一组是删除索引和不删除索引进行对比（无论删除索引与否，都会执行 Redo）。

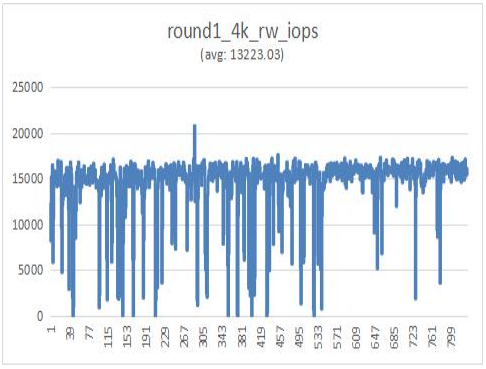
首先进行执行 Redo 和不执行 Redo 的对比测试，结果如下：

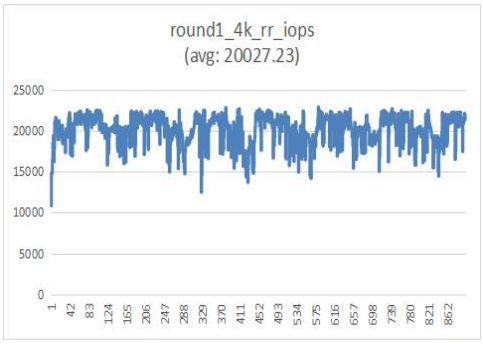
Without-Redo　　With-Redo

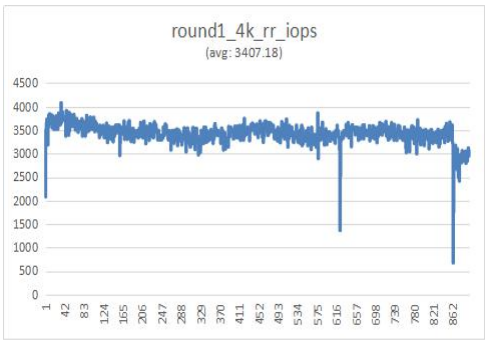

round1_privision_bw
(avg: 282.97)

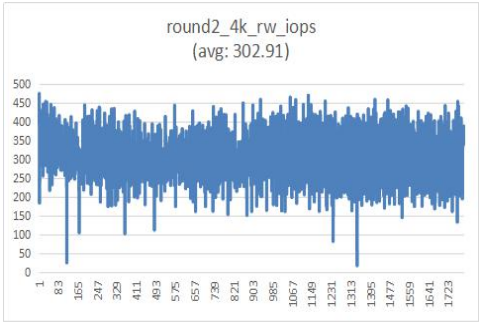

round1_provision_bw
(avg: 366.77)

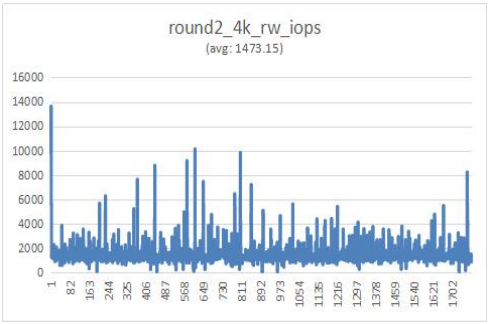

round1_4k_rw_iops
(avg: 12302.65)

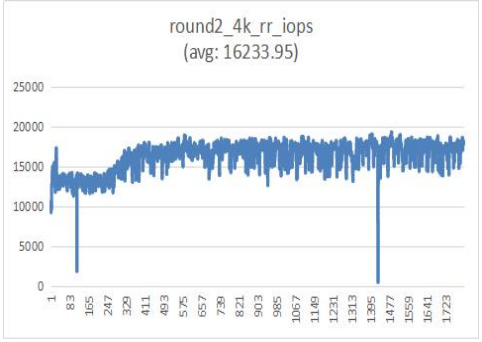

round1_4k_rw_iops
(avg: 13223.03)

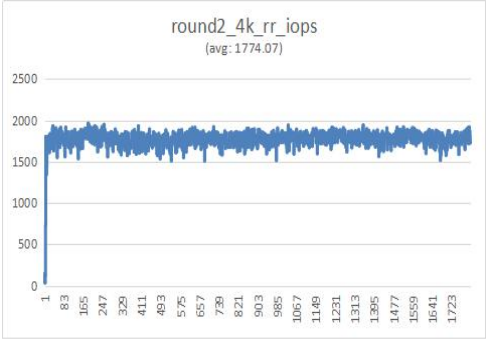

round1_4k_rr_iops
(avg: 20027.23)


round1_4k_rr_iops
(avg: 3407.18)


round2_4k_rw_iops
(avg: 302.91)


round2_4k_rw_iops
(avg: 1473.15)

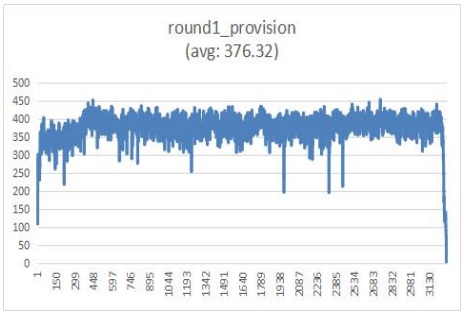
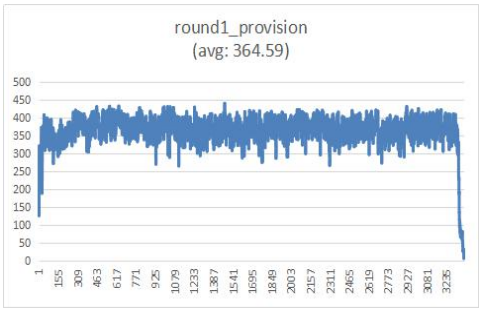
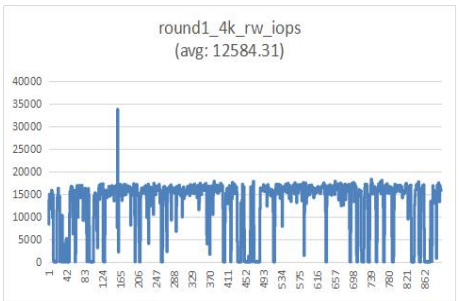
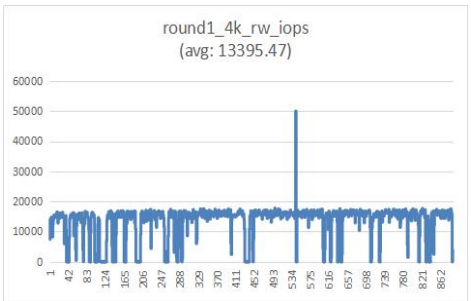
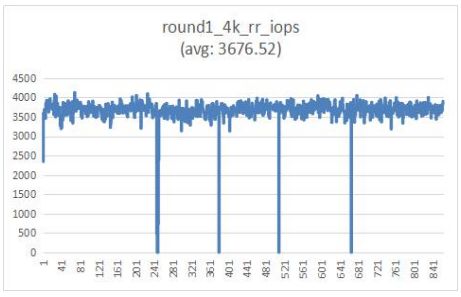
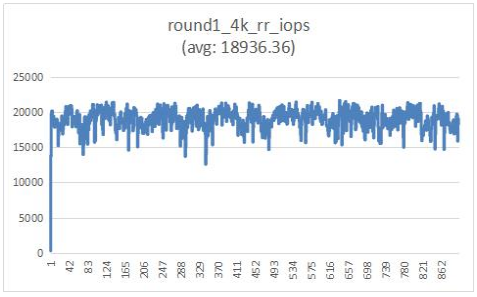

round2_4k_rr_iops
(avg: 16233.95)


round2_4k_rr_iops
(avg: 1774.07)

在 round1 4K 随机读测试中（参考 round1_4k_rr_iops），不执行 Redo 的情况下 IOPS 为 20027.23，而执行 Redo 的情况下，只有 3407.18，两者相差巨大，究其原因，经过 round1 置备阶段（参考 round1_provision_bw）和 round1 4k 随机写测试（参考 round1_4k_rw_iops）之后，SSD 空间使用率已经超过了内置的 SSD 空间回收阈值（这可以通过在不执行 Redo 的情况下，round2 4K 随机写测试性能较低得到佐证），需要逐渐释放那些已经 Redo 过的 LogUnit 对应的索引，以便后续到达的写请求复用 LogUnit，在不执行 Redo 的测试中，既不会执行 Redo，也不会删除索引，所以在 round1 4K 随机读测试中可以全部命中 SSD，然而在执行 Redo 的测试中，如果 4K 随机读的目标是那些被删除的索引对应的页，则不会命中 SSD，需要去后端 HDD 中读取，所以性能会差很多。

在 round2 4K 随机读测试中(参考 round2_4k_rr_iops),不执行 Redo 的情况下 IOPS 为 16233.95，而执行 Redo 的情况下，只有 1774.07，两者相差巨大，原因同上。

在执行 Redo 和不执行 Redo 的对比测试中，着重从删除索引导致读命中率降低的角度进行了分析，但是不执行 Redo 的情况下，读线程和 Redo 线程之间的锁竞争开销也是不可忽略的，为了将锁竞争带来的影响降低，进行了第二组对比测试，即在执行 Redo 的情况下对比测试删除索引和不删除索引情况下性能，结果如下：

With-DropIndex	Without-DropIndex
	
	
	

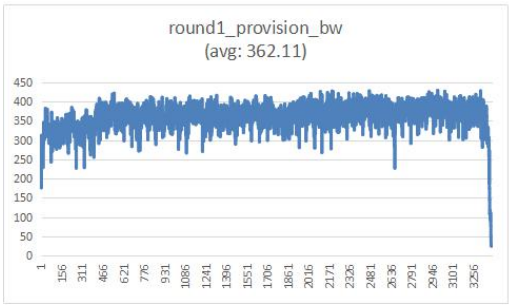
在 round1 4K 随机读测试中（参考 round1_4k_rr_iops），不删除索引的情况下 IOPS 为 18936.36，而删除索引的情况下，只有 3676.52，因为在删除索引的情况下，后续到达的读请求如果发生在那些被删除的索引对应的数据页，则需要去后端 HDD 中读取数据，降低读请求在 SSD 中的命中率，进一步验证了删除索引对于读命中率的影响。

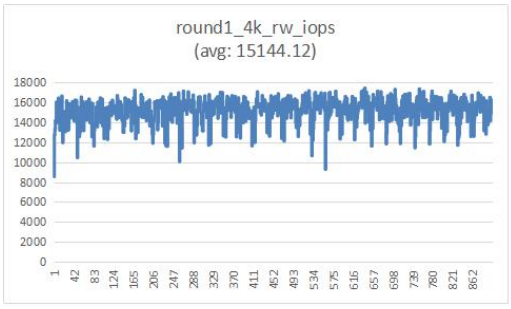
可能细心的人会问，上述两组对比测试中，SSD 空间容量是相同的，无论是否执行 Redo，也无论是否删除索引，存放于 SSD 中的数据总量是相同的，在 4K 随机读测试中，性能应该接近才对啊？这里涉及到“数据密度”问题，在 SSD 写缓存中默认是开启数据压缩功能的，以 snappy 接近 3 的压缩效率，64KB 的数据压缩后约占用 6 个 page（page 大小为 4KB），而 4KB 的数据压缩后占用 1 个 page，相比较而言，64KB 数据的数据密度要高一些，同样的 SSD 空间容量，“数据密度”越高，则存放的真实数据越多，读请求命中 SSD 的概率也越高，读性能也越高，基于此，如何提高“数据密度”对于提高读性能至关重要，初步来说，可以考虑在索引释放和数据合并上做文章。索引释放，可以考虑优先释放那些“数据密度”较低的 LogUnit 对应的索引，以及有效数据较少的 LogUnit 对应的索引。数据合并，即尝试将“数据密度”较低的多个独立的连续数据块进行合并，合并成“数据密度”较高的数据块。未来，SSD 写缓存的进一步优化，可以考虑这些点。

（6）Hash 方式对读写性能的影响

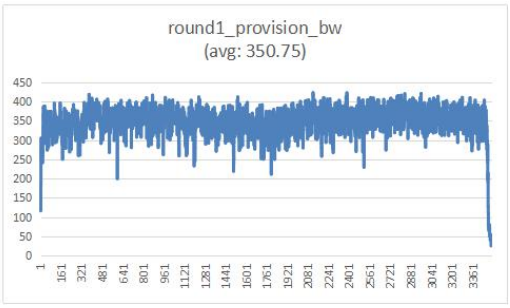
在 7 月上旬的测试中性能相对比较稳定，但是 7 月下旬的测试中性能波动较大，经过代码比对分析，发现两个代码版本中作为索引分区依据的 Hash 值的计算方式有变化（7 月上旬基于 InodeID 和 ChunkIndex 计算 Hash 值，下旬则基于 InodeID 计算 Hash 值），最有可能导致该性能差异，为了验证该分析，专门针对 Hash 方式进行对比测试，测试结果如下：

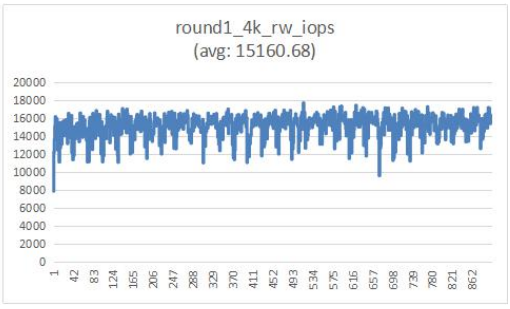
Hash By Inode

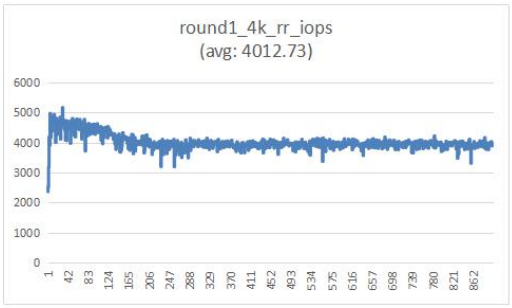


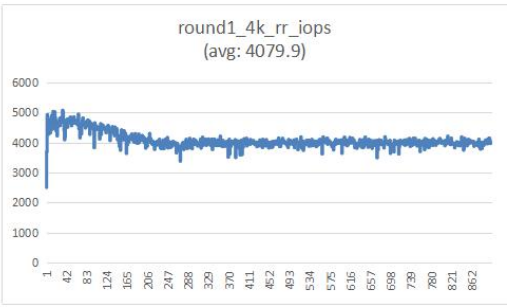


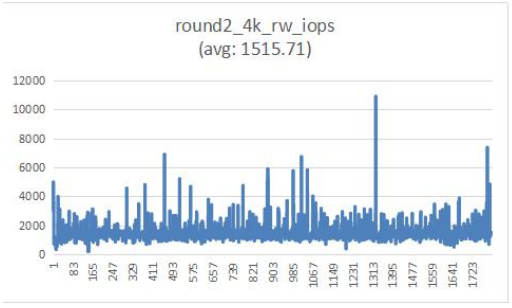
Hash By Inode and Chunk

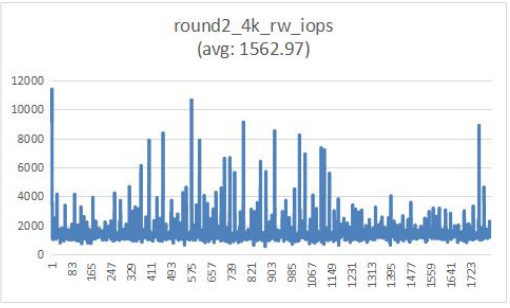


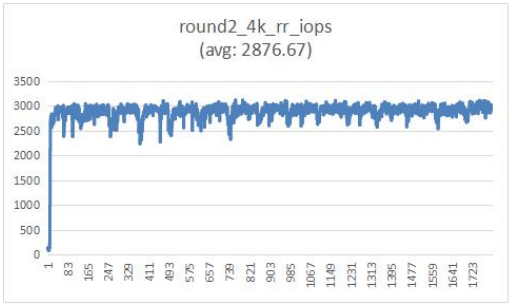


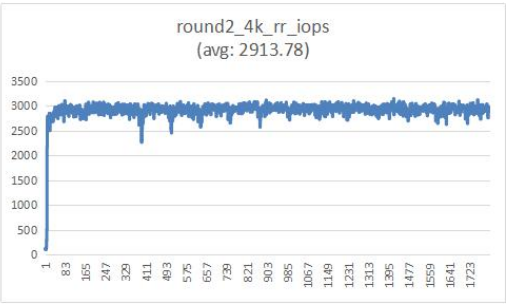

round1_4k_rr_iops
(avg: 4012.73)

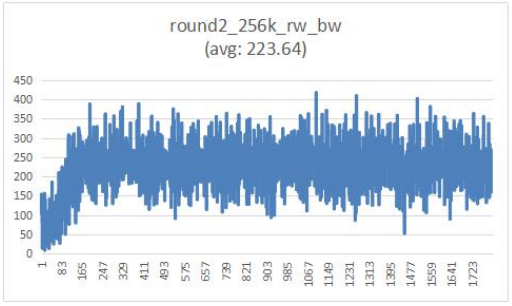

round1_4k_rr_iops
(avg: 4079.9)

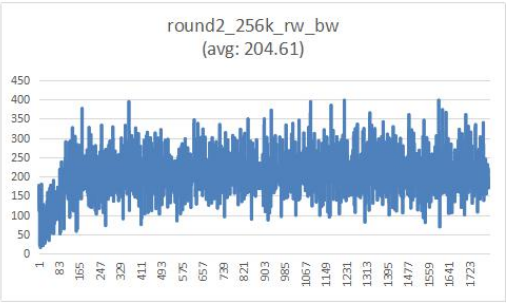

round2_4k_rw_iops
(avg: 1515.71)

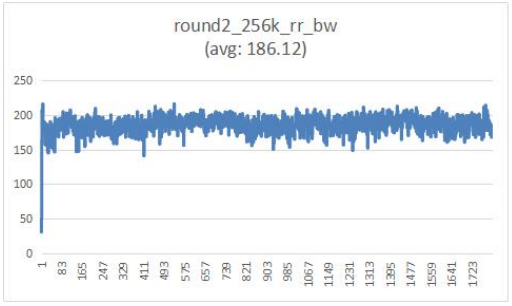

round2_4k_rw_iops
(avg: 1562.97)

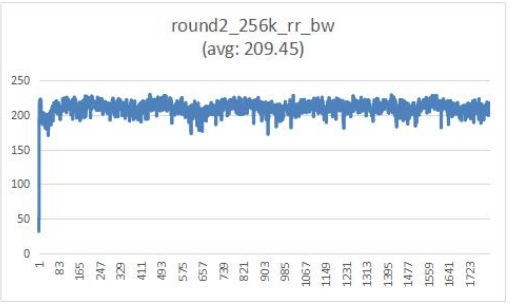

round2_4k_rr_iops
(avg: 2876.67)

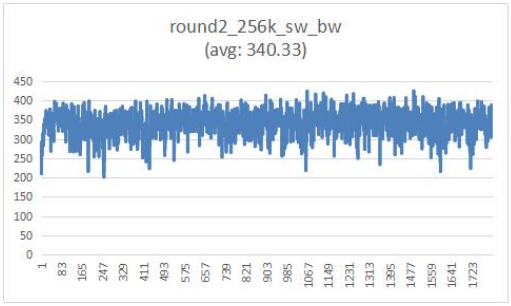

round2_4k_rr_iops
(avg: 2913.78)

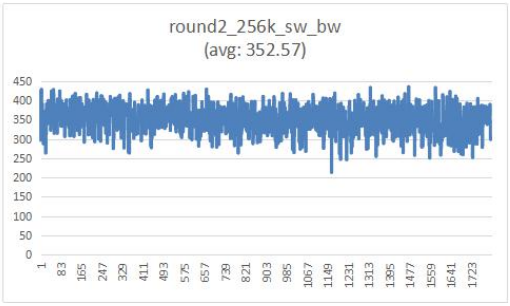

round2_256k_rw_bw
(avg: 223.64)

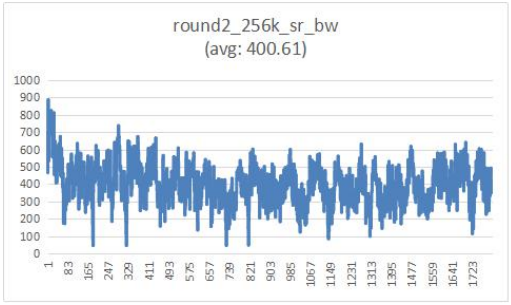

round2_256k_rw_bw
(avg: 204.61)

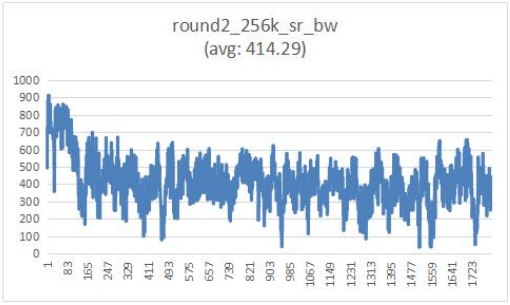

round2_256k_rr_bw
(avg: 186.12)

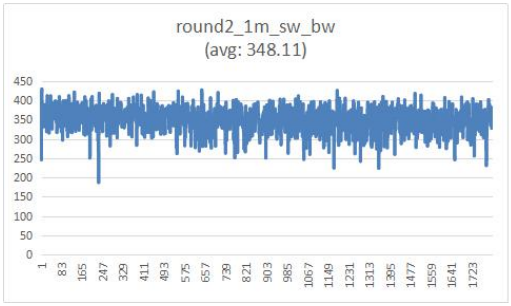

round2_256k_rr_bw
(avg: 209.45)

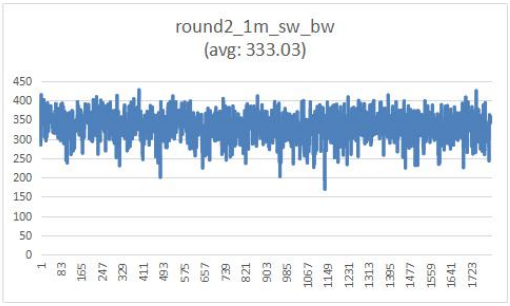


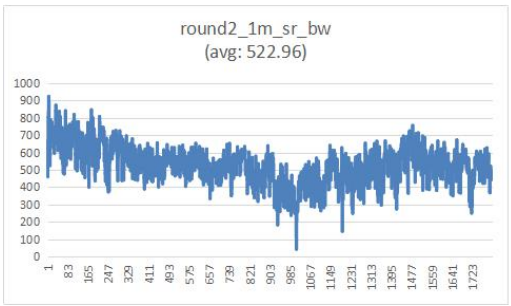


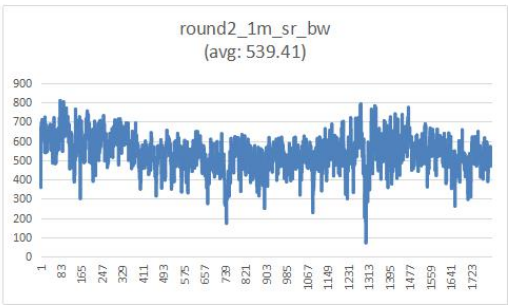












从上述测试结果来看，两种 Hash 计算方式下，性能不分伯仲，所以上述关于七月上旬和下旬性能测试结果差异的分析可能是错误的，Hash 计算方式并非是导致性能差异的原因。但是 Hash 方式的确可能对系统的性能造成影响，因为 hash 方式一定程度上决定了索引的平衡性，以及 SSD 访问的均衡性。

4.5 对系统运行的影响

略。

4.6 对开发环境的影响

略。

5. 经济可行性分析

略。

6. 其它因素分析（社会因素等）

略。

7. 未来工作

在升腾测试过程中暴露出了一系列问题，鉴于时间关系，并没有一一解决，这里将部分已经发现但尚未解决的问题陈列出来，以供后续优化参考。

（1）4K 随机写性能波动较大；

（2）大块顺序读情况下，可以很好的利用预读功能，但是如果存在部分页在 SSD 写缓存中命中，另一部分页在 HDD 命中，则会破坏预读，导致预读失效，从而导致大块顺序读性能较低。

（3）为了避免写线程在复用 LogUnit 时删除该 LogUnit 相关的所有索引的时间开销，将删除 LogUnit 索引这一操作转嫁给 Redo 线程执行，但是相较于写线程的“按需删除”，Redo 线程需要“未雨绸缪”，即写线程可以在真正复用某个 LogUnit 的时候才会删除该 LogUnit 相关的索引，但是 Redo 线程为了确保写线程在需要复用 LogUnit 时有可用的 LogUnit，需要事先删除部分 LogUnit 的索引，这样带来的问题是，较早地删除 LogUnit 的索引，导致后续到达的读请求命中 SSD 写缓存的可能性降低，进而导致读性能降低。

（4）现在的 LogUnit 复用和索引删除没有考虑“数据密度”这一因素，对于读性能会有一定影响。

（5）SSD 写缓存空间不足情况下到达的写 IO，性能虽然不会出现降为 0 的情况，但是性能依然较低。

8. 结论

略。