1. 简介

1.1 背景

当前的 OSS 设计中，利用 SSD 来加速读写，但是在 SSD 使用上仍有可以优化的空间，主要表现在以下几个方面：

（1）降低 SSD 上混合读写的比例；

（2）减少无谓的 SSD 读写；

（3）尽可能多的尽可能早的释放 SSD 空间；

（4）基于 SSD 和 HDD 做分层，尽量使热数据可以在 SSD 层命中；

1.2 术语

OpLog：操作日志，它包括 RawLog 和 MemoryCache(For Write)两部分；

RawLog：基于 SSD 的操作日志；

MemoryCache(For Write)：基于内存的操作日志，它包括 Mutable MemoryCache 和 Immutable MemoryCache 两部分；

Mutable MemoryCache：当前正在被写入的 MemoryCache 部分；

Immutable MemoryCache：之前被写入的，当前不能再被修改的 MemoryCache 部分；

UnifiedCache：读缓存，它包括 SSDCache 和 MemoryCache(For Read)两部分；

SSDCache：基于 SSD 的读缓存；

MemoryCache(For Read)：基于内存的读缓存；

TieredStorage：存储分层，暂时包括 SSDTier 和 HDDTier；

SSDTier：以 SSD 作为存储介质的存储层；

HDDTier：以 HDD 作为存储介质的存储层；

1.3 参考资料

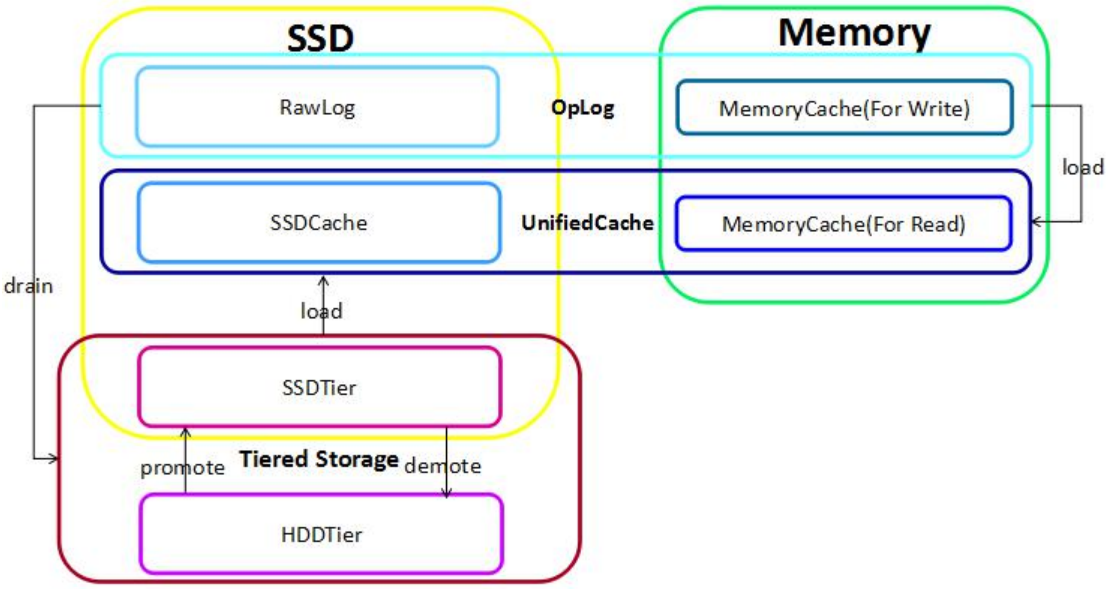
略。

2. 总体设计

2.1 功能需求

2.2 性能需求

2.3 模块结构



2.4 模块设计

2.4.1 OpLog 设计

Features

Hybrid storage;

Dynamical shard by THD/HDD;

Write ahead logging;

Large sequential IO recognition;

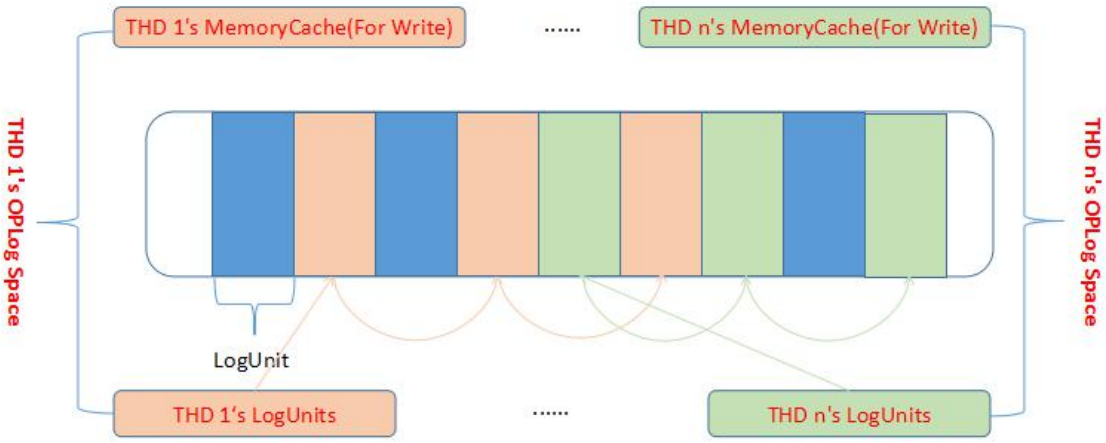
Compaction;

Hybrid storage

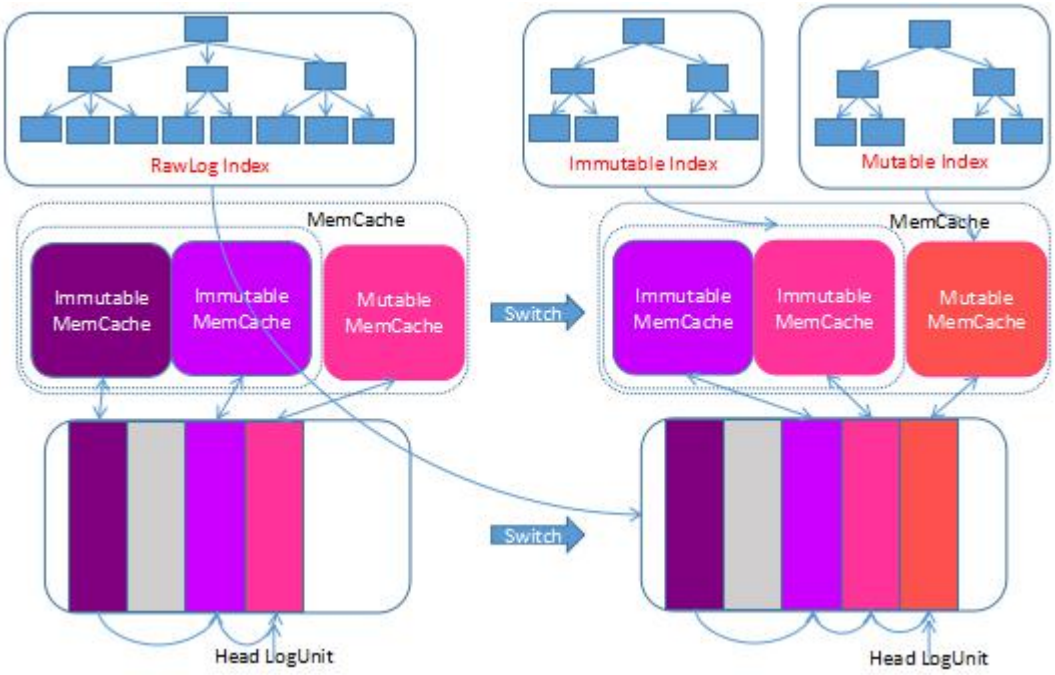
基于内存的 MemoryCache(For Write) 加上基于 SSD 的 RawLog。

Dynamical shard by Thread

所有线程共享可用的 RawLog 空间和 MemoryCache 空间，但是每个线程单独管理自己的那份 RawLog 空间和 MemoryCache 空间。



Single THD's OpLog logic view



Mutable MemCache

Mutable MemCache 用于缓存当前正在被写入的 LogUnit 中的数据，所有存在于当前正在被写入的 LogUnit 中的数据一定同时存在于 Mutable MemCache 中，所有存在于 Mutable MemCache 中的数据也一定存在于当前正在被写入的 LogUnit 中，所以说 Mutable MemCache 是 LogUnit 对齐的，但是 Mutable MemCache 中的数据是不压缩的，LogUnit 中的数据可能是压缩的，所以 Mutable MemCache 占用的内存空间可能远大于数据在 LogUnit 中占用的空间，当然考虑到 Mutable MemCache 中的数据执行 in-place update，Mutable MemCache 占用的内存空间也可能小于数据在 LogUnit 中占用的空间。

伴随着 LogUnit 的分配，Mutable MemCache 也随之分配，如果之前已经存在 Mutable MemCache，则之前的这个 Mutable MemCache 切换为 Immutable MemCache，所以任一时刻，对于某个线程对来说，有且只有一个 Mutable MemCache。

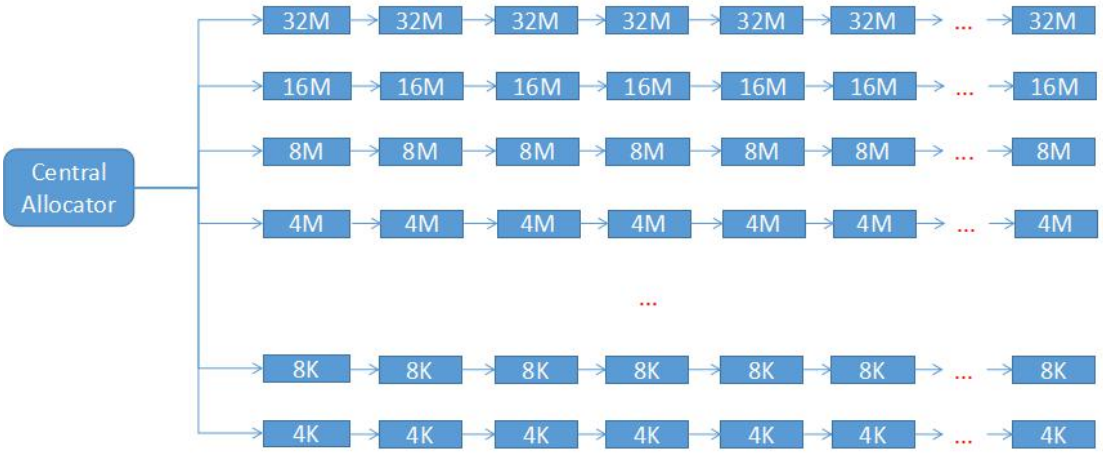
Immutable MemCache

Immutable MemCache 由 Mutable MemCache 转换而来，所以和 Mutable MemCache 一样，Immutable MemCache 也是 LogUnit 对齐的，其中的数据也是不压缩的。但是 Immutable MemCache 可能有多个，至于到底有多少个，则受限于系统中可用的 MemoryCache 空间和当前系统的状态。

Mutable MemCache allocation

因为 Mutable MemCache 是可变大小的，考虑到 snappy 压缩算法的压缩比为 2.5 左右（参考：http://bbs.itcast.cn/thread-85176-1-1.html，将 OSS 中的数据当做大整形数据的文本？），一个 LogUnit 中存放的数据最少为 4KB（全部是 overwrite 的情况下），最多可达 160MB。为了充分利用内存，同时减少内存分配申请次数，采用如下机制：

（1）系统中维护一个中央内存分配器，在该中央内存分配器中维护一系列 MemoryUnit，每种 MemoryUnit 的内存大小依次为 32M，16M，8M，4M，2M，1M，512K，256K，128K，64K，32K，16K，8K，4K。每种 MemoryUnit 有多个（但是数目可能不同），通过链表（链表已经有比较成熟的 lock-free 实现，在多线程并发访问该中央内存分配器的时候可以无须担心锁开销）维护。



（2）中央内存分配器负责维护申请到的所有内存的首地址，以便能够正确将内存释放给系统。

（3）每一个写线程在为 Mutable MemCache 申请内存空间的时候，都采用逐步试探的方式，以期尽可能减少内存浪费，因为分配到的内存是以 MemoryUnit 为单位的。第一次为 Mutable MemCache 分配内存的时候，直接分配 32M，后续分配则根据当前 LogUnit 中可用空间大小以及已写入数据压缩比来估算该 LogUnit 还可以容纳多少数据，进而决定申请多大的内存空间。

（4）可能存在某个写线程向中央内存分配器申请内存可以得到满足，但是只能通过多个 MemoryUnit 拼凑才能得到满足的情况，比如申请 32M 内存，但是现在中央内存分配器中没有 32M 的内存了，但是有很多 16M 的内存，就可以通过 2 个 16M 的内存满足本次分配请求。

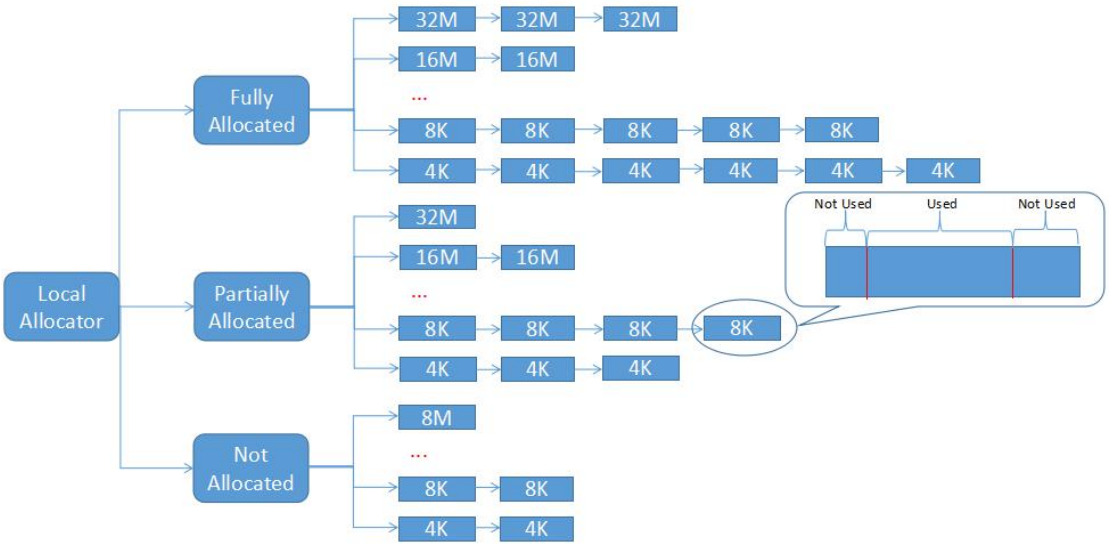
（5）可能存在某个写线程向中央内存分配器申请的内存大小只能部分满足的情况，此时丢弃当前的 Mutable MemCache，只维护 Mutable Index。

（6）可能存在某个写线程向中央内存分配器申请的内存大小无法得到满足的情况，此时丢弃当前的 Mutable MemCache，只维护 Mutable Index。

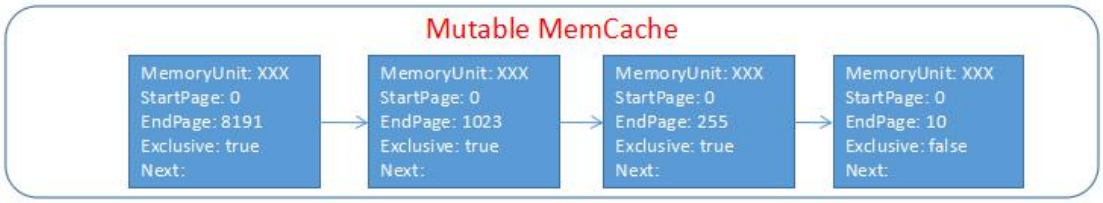
（7）可能存在某个写线程为 Mutable MemCache 申请的内存大于实际需要的内存的情况，申请的内存可能包括 1 个或者多个 MemoryUnit，如果包括多个 MemoryUnit，且直到 Mutable MemCache 即将切换的时刻，至少还剩余一个 MemoryUnit 尚未使用，那么这些尚未使用的 MemoryUnit 将被立刻释放给中央内存分配器。在 Mutable MemCache 即将切换的时刻，正在被使用的 MemoryUnit 也可能是部分使用的，那么该 MemoryUnit 的剩余内存空间将被提供给接下来 Mutable MemCache 使用，该 MemoryUnit 的内存空间将被当做一个环形内存区。

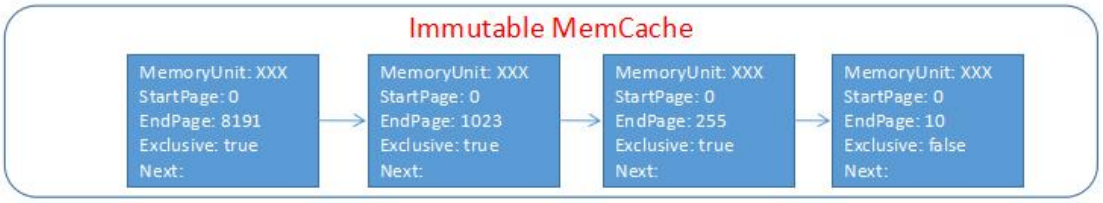
（8）每一个写线程从中央内存分配器申请到的内存空间都会加入本地内存管理器的管理，本地内存管理器记录当前哪些 MemoryUnit 正在被使用（包括完全被使用的和部分被使用的，包括 Immutable MemCache 使用的和 Mutable MemCache 使用的），以及哪些 MemoryUnit 已经被分配了但是尚未被使用，对于那些正在被使用的 MemoryUnit 还要记录

其使用的内存范围（本地内存管理器看到的全局视图）。



（9）在 Mutable MemCache 和 Immutable MemCache 中也要维护它所使用的每一段内存空间，这里每一段内存空间用 MemSegment 表示，一个 MemSegment 一定来自于某一个 MemoryUnit，MemSegment 所代表的内存区间一定是 MemoryUnit 所代表的内存区间的子集。





（10）在剔除 Immutable MemCache 时，本地内存管理器检查该 Immutable MemCache 所使用的 MemoryUnit 是否可以释放，如果可以释放就释放之，否则更新相关 MemoryUnit 在本地内存管理器中的使用范围信息。

Mutable/Immutable MemCache deallocation

（1）对于那些不具备数据局部性特征的线程对，不为其维护 Mutable MemCache，如果它有 Immutable MemCache，这些 Immutable MemCache 也会被迅速释放。当然，这里还是会维护 Mutable Index。

（2）本地内存管理器在向中央内存分配器申请内存的时候，中央内存分配器会给本地内存管理器关于内存使用压力的指示，指示它是否需要释放一些内存，一旦本地内存管理器收到中央内存分配器释放内存的指示，本地内存管理器必须无条件释放部分内存。

（3）对于那些浑然不知中央内存分配器内存压力的线程，中央内存分配器可以主动向

其发送释放部分内存的指示。

Data access locality judgement

无论是 Mutable MemCache 还是 Immutable MemCache，都可以确定其中管理的有效数据页的个数，以及在 Mutable MemCache 或者 Immutable MemCache 的生命周期中 IO 命中的个数，这样就可以得到一个比例，通过判断这个比例和指定的阈值（作为数据是否具备局部性的分水岭），来确定数据访问是否具备局部性特征。

OpLog Index management

索引管理包括 RawLog Index，Immutable Index，Mutable Index 三部分。

RawLog Index 中记录的是除了 Immutable Index 和 Mutable Index 中索引以外的索引。

Mutable Index 用于记录 Mutable MemCache 中的索引情况，它会被写线程和读线程共享访问，为了减少锁竞争，最好被设计成为无锁数据结构，但是目前没有找到经过工业验证的开源的且适合这里的无锁数据结构，所以暂时还是只能使用已有的 B*树来管理（现有的 B*树尚不支持无锁访问），但是无论如何 Mutable Index 在一定程度上都减小了锁的粒度。

在有 Mutable MemCache 的情况下，Mutable Index 既要能够索引到 Mutable MemCache 中的 page，也要能够索引到 RawLog 中的 page。所以索引中的 value 部分包含两个值。在没有 Mutable MemCache 的情况下，Mutable Index 只需要索引到 RawLog 中的 page。

对于 Write 操作，如果是关于某个页在 Mutable MemCache 的第一次写入，则在 Mutable Index 中添加关于该页的索引，如果是关于该页的覆盖写，则 Mutable Index 更新关于该页的索引（in-place update）。

对于 Truncate 操作，首先需要从 Mutable Index 中删除被 Truncate 的区间所包含的页的索引，同时将该 Chunk 及其 Truncate 的区间添加到被 Truncate 的 Chunk 集合中。

对于 Create 操作，将该 Chunk 添加到被 Create 的 Chunk 集合中。

对于 Delete 操作，首先需要从 Mutable Index 中删除关于该 Chunk 的所有索引信息，其次，需要从被 Truncate 的 Chunk 集合中删除该 Chunk，也需要从被 Create 的 Chunk 集合中删除该 Chunk，最后将该 Chunk 添加到被 Delete 的 Chunk 集合中。

对于 Replicate 和 Reselect 操作，首先需要从 Mutable Index 中删除该 Chunk 的索引信息，其次需要将该 Chunk 添加到被 Delete 的 Chunk 集合中。

对于 ReadRangeTable 也需要维护一个 Mutable ReadRangeTable。

对于 Write 操作，直接更新 Mutable ReadRangeTable。

对于 Truncate 操作，直接更新 Mutable ReadRangeTable。

对于 Delete 操作，直接更新 Mutable ReadRangeTable。

对于 Replicate 和 Reselect 操作，直接更新 Mutable ReadRangeTable。

伴随着 LogUnit 的切换，Mutable Index 切换为 Immutable Index，Mutable ReadRangeTable 也切换为 Immutable ReadRangeTable，这些切换都是伴随着 Mutable MemCache 切换为 Immutable MemCache 的切换而切换的，无需额外的工作，当然在切换过程中需要锁保护。

Immutable Index 会被逐渐合并到 RawLog Index，Immutable ReadRangeTable 也会被逐渐合并到 RawLog ReadRangeTable。在合并过程中，被 Create/Delete/Truncate 的集合中的 Chunk 信息不再作用于 Immutable Index 和 Immutable ReadRangeTable，而是作用于 RawLog Index 和 RawLog ReadRangeTable 上，且先将 Delete/Truncate 的集合中的 Chunk 信息作用于 RawLog Index 和 RawLog ReadRangeTable 上，然后再将 Immutable Index 和 Immutable ReadRangeTable 中的信息分别作用于 RawLog Index 和 RawLog ReadRangeTable 上。

迫于内存压力，Immutable MemCache 可能被释放，但是在释放 Immutable MemCache 的时候，该 Immutable MemCache 对应的 Immutable Index 和 Immutable ReadRangeTable 将被保留。假如在释放 Immutable MemCache 的时候，Immutable Index 和 Immutable ReadRangeTable 也被释放，那么在释放之前必须被合并到 RawLog Index 和 RawLog ReadRangeTable 中，这个合并过程也可能在较长时间之内阻塞写线程。那么 Immutable Index 和 Immutable ReadRangeTable 由谁在何时被合并到 RawLog Index 和 RawLog ReadRangeTable 呢？Immutable Index 和 Immutable ReadRangeTable 由 redo 线程负责将其合并到 RawLogIndex 和 RawLog ReadRangeTable 中，具体合并方式请参考“**Immutable MemCache Redo**”。

Immutable MemCache Redo

在执行 Immutable MemCache Redo 之前，首先要判断当前时机是否合适，Immutable MemCache Redo 有以下执行时机：

（1）下一个即将执行 redo 的 LogUnit，恰好处于 Immutable MemCache 中；

（2）下一个即将执行 redo 的 LogUnit 不在 Immutable MemCache 中，但是前一个 LogUnit 已经 redo 结束，下一个 LogUnit 的 redo 尚未开始。

在时机合适的前提下，检查 Immutable MemCache 中有多少可以合并的连续的大块的页（可以直接通过扫描 Immutable Index 获得），对于这些大块页直接执行 redo，并对 Immutable Index 中相关的页的索引打上 redo 标识，标识该页已经执行过 redo 了。一旦该 Immutable MemCache 执行完 redo，redo 线程就将该 Immutable MemCache 对应的 Immutable Index 及其前继 Immutable Index 合并到 RawLog Index 中。

Immutable MemCache Compaction

计算复写率，发生 page 级别 overwrite 的次数除以 Immutable MemoryCache 中总的页数，这里 Immutable MemoryCache 中总的页数包括那些被 overwrite 的页和删除的页，但不包括合并 redo 的页，如果复写率超过某个阈值（比如，至少执行 compaction 之后，至少可以减少使用一个 LogUnit），则执行 compaction，compaction 过程实际上是为了尽快释放 SSD 空间。为简单起见，暂时不支持 compaction。

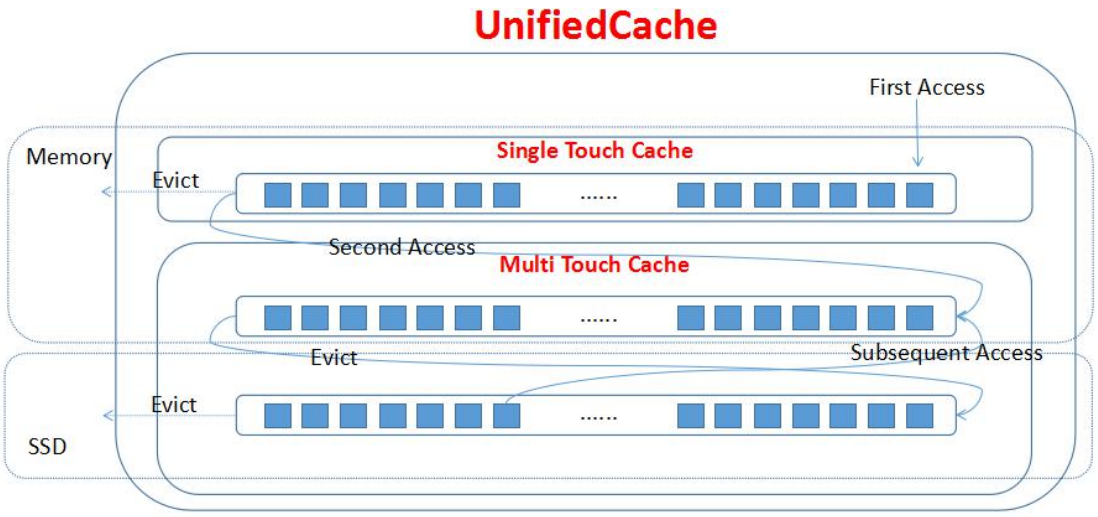
Memory-Based ReadCache for overwrite

为写线程单独开辟一块固定大小的小空间，专门用于应对 non-page-aligned partial overwrite 这种情况，尽可能避免写线程去读取 SSD。

2.4.2 UnifiedCache 设计

Overview

UnifiedCache 是一个动态的读缓存，它包括 Single Touch Cache 和 Multi Touch Cache 两部分，Single Touch Cache 基于内存空间，Multi Touch Cache 则基于内存+SSD 的混合介质空间。对于一次读请求中所涉及到的页，如果不在 UnifiedCache 中，则这些页相关的数据将被添加到 Single Touch Cache 中，处于 Single Touch Cache 中的页，采用基于 LRU 的淘汰策略，如果该页在被淘汰之前再次被访问，则它会被添加到 Multi Touch Cache 的内存部分，否则将不再被保留在 UnifiedCache 中。处于 Multi Touch Cache 内存部分的页也采用基于 LRU 的淘汰策略，但是被淘汰出去的页会依然被保留在 UnifiedCache 中，只不过将该页从 Multi Touch Cache 的内存部分移动到 Multi Touch Cache 的 SSD 部分。处于 Multi Touch Cache 的 SSD 部分的页也采用 LRU 进行管理，那些长时间不被访问的页将被从 Multi Touch Cache 的 SSD 部分删除，而那些在被删除之前再次被访问到的数据页则被移动到 Multi Touch Cache 的内存部分。



Cache sharding

跟 OpLog 类似，UnifiedCache 按照线程进行划分。

Cache granularity

UnifiedCache 中以 page 为单位进行缓存管理。

Cache space management

为简单起见，每个线程使用固定大小的 UnifiedCache 空间。

Cache update and replacement

UnifiedCache 的 Memory 部分采用 LRU-2 进行管理，SSD 部分则采用 LRU 进行管理。UnifiedCache 中包括 Single Touch Cache、Multi Touch Cache 的 memory 部分、Multi Touch Cache 的 SSD 部分三层，每一层单独管理，在插入新元素到 UnifiedCache 的某一层时，如果该层仍

有可用空间，则直接插入，否则剔除 LRU 的尾部的元素，为新插入的元素腾出空间。对于 Single Touch Cache 和 Multi Touch Cache 的 SSD 部分，剔除的元素将不在 UnifiedCache 中占用空间，但是从 Single Touch Cache 的 Memory 部分剔除的元素将被移动到 Multi Touch Cache 的 SSD 部分。

Cache Invalidation

如果一个存在于 UnifiedCache 中的页被覆盖写，则该页仍然保留在原来的存放位置，只更新其内容。

ReadAhead

如果需要读取的数据从 HDD 层读取到，则跟踪该数据的访问模式，确定是否执行预读。

2.4.3 StorageTier 设计

从 IO 栈结构上来说，StorageTier 将作为一个全新的 FinalIOModule，即 StorageTierIOModule。该 Module 包括 SSD 和 HDD 两种介质，主要承担 Redo 和 Read 产生的 IO，Redo 在 SSD 上无需 AIO，而在 HDD 上采用 AIO 效率会更高，所以在 SSDTierIOModule 内部需要能够根据文件所在的位置走不同的接口。

Hot data && cold data

StorageTierIOModule 希望通过分层，将热数据放在 SSDTier，从而提高 Redo 和 Read 的性能，那么对于 Redo 和 Read 来说，哪些数据算是热数据呢？

在 Redo 过程中，至少有三类数据可以作为热数据：第一类，每一个 Chunk 的 Header 部分，因为 redo 之后需要去更新 Header 部分的 WriteVersion 信息，在随机写环境下，由 redo 引起的 Header 的更新可能会更加频繁；第二类，应用数据的元数据部分，这部分数据的更新可能比较频繁；第三类，应用数据的访问比较频繁的数据部分。第二类和第三类数据可以统一作为访问比较频繁的数据。

针对 redo 过程中的第一类热数据，在 redo 过程可以根据统计信息统计出那些即将执行 redo 的且具有大量随机 redo IO 的 Chunk 对应的 Header 提升到 SSDTier 中，当然对于这类热数据是否做提升，可以通过策略指定，比如设置 PromotePolicy.mHeader = true，则对于这些热点数据的 Header 部分做提升，否则不提升。为了方便管理，对于提升到 SSDTier 的 Header 数据，将以独立的文件的形式存在，通过文件名去区分该 Header 是率属于哪个 Chunk 的。

针对 redo 过程中的第二类热数据，需要在统计信息中识别出这类数据，在虚拟化环境中，这类热数据一定存在，对于这类热数据是否做提升，也可以通过策略指定。

针对 redo 过程中的第三类数据，也需要通过统计信息来识别这类数据，但是跟第二类数据不同，这类数据不一定会存在，对于这类数据是否做提升，也可以通过策略指定。

在 Read 过程中，当前只能依赖于统计信息来获取哪些数据被频繁 read，将这类数据作为热数据，并将其提升到 SSDTier 中。

StorageTierIOModule 也希望通过分层，将冷数据放在 HDDTier，从而保证热数据提升到 SSDTier 的时候，有足够的空间可用。在 SSDTier 中，有 Header 和 SubChunk 两类文件，在 StorageTierIOModule 中通过统计信息识别出那些较长时间未被访问的文件，并将其下沉到 HDDTier 中。当前设计上只考虑 SSDTier 和 HDDTier 的组合，暂不考虑层级扩展。

Statistics

StorageTierIOModule 中维护 IO 统计信息，写线程维护关于 SubChunk 的写统计相关的信息，读线程维护关于 SubChunk 的读相关的统计信息，这两种统计信息主要用于识别热数据；读线程、写线程和 Redo 线程共同维护 SSDTier 中 SubChunk 的访问相关的统计信息，Redo 线程维护 Redo 相关的统计信息，写线程和读线程共同维护读相关的统计信息，但是写线程和读线程在这里维护的只是到达 SSDTier 的关于该 SubChunk 的统计信息，该统计信息主要用于识别冷数据。

必须减少热度统计数据集，只统计最近被访问的数据集的热度信息，从这些最近被访问的数据的热度信息中可以找出哪些数据是热数据，从这些热数据中找出那些最热的且存在于容量层中的数据迁移到性能层中，那么冷数据又该如何识别呢，在当前只考虑容量层和性能层两层分层的情况下，只需要识别出性能层的冷数据即可，而当前环境下性能层的容量非常有限，可以单独为其维护一个热度统计视图，从该视图中就可以非常方便的得到哪些数据是冷数据。

因为系统内存空间有限，不可能维护所有 SubChunk 的统计信息，必须减少统计数据集，对于热数据，只统计最近被访问的数据集的热度信息，从这些最近被访问的数据的热度信息中可以找出哪些数据是热数据，从而将这些热数据从 HDDTier 迁移到 SSDTier。对于冷数据，在当前只考虑 HDDTier 和 SSDTier 两层分层的情况下，只需要识别出 SSDTier 的冷数据即可，而当前环境下 SSDTier 的容量非常有限，可以为 SSDTier 中的每一个 SubChunk/Header 维护统计信息，但是考虑到将来也可能会采用一整块盘来做 SSDTier，所以也需要找出那些最近最少访问的 SubChunk 作为冷数据的候选。具体来说，对于热数据，每个线程至多维护最近访问的 N（比如 16K）个 SubChunk 的统计信息，这 N 个 SubChunk 的统计信息采用数组存放，并采用 LRU 进行管理，另外维护一个堆，其中存放最热的 H（比如 2K）个 SubChunk 信息，这个堆起到热度排序的作用；对于冷数据，跟热数据类似，采用数组+LRU+堆进行管理，只不过堆中存放的是 S 个最冷的 SubChunk 信息。

Thread Model

StorageTierIOModule 中除了和其它 IOModule 中一样的读线程、写线程和 Redo 线程以外，另外有一个迁移线程，专门用于执行跨层迁移 SubChunk。

迁移线程在无事可做的情况下进入睡眠状态，等待其它线程唤醒，唤醒它的时机有：

（1）写线程发送关于写的统计信息，写线程在什么时候发送该统计信息？？？

（2）读线程发送关于读的统计信息；

（3）Redo 线程发送关于 redo 的统计信息；

当迁移线程接收到上述统计信息之后，相应的处理如下：

2.4.4 优化

2.4.4.1 Redo 优化

（1）针对 LogUnit Redo 过程中大量随机 IO 的情况，采用紧凑型文件格式，将这些 IO 以 Chunk 为单位进行组织以文件的方式写入 HDD 中（写入过程中走 pagecache + 整个文件写完之后 sync，写入过程采用 AppendOnly 的方式），并在合适的时机将这些文件写回文件真正的位置。

（2）基于 BloomFilter 的 Redo 优化，具体运行是这样的，在系统启动过程之初，仍然

按照现有的逻辑去执行 redo，但是在 redo 过程中如果有比较多的 LogUnit 在等待 redo，且现在 LogUnit 分配的速度大于 LogUnit 释放的速度，则从下一个 LogUnit 分配开始，为后续的所有的 LogUnit 建立 BloomFilter，以期后续 LogUnit 可以在更多 LogUnit 范围内进行合并（当前实现中，redo 只在一个 LogUnit 范围内尝试进行合并），随着 redo 的进行，该轮到 BloomFilter 相关的第一个 LogUnit 执行 redo 的时候，冻结该 BloomFilter，并为后续 IO 生成新的 BloomFilter。对于被冻结的 BloomFilter，针对其所关联的所有 LogUnit 范围，利用 BloomFilter 查找到大范围的连续 IO。关于 BloomFilter，是这样维护的，对于每个 Chunk 都进一步按照 16M 为单位划分为 Sub-Chunk，对于每一个 Sub-Chunk 都以{inodeid, chunkidx, subchunkidx}为 key，以一个 page 为 value，添加到 BloomFilter 中，其中一个 page 中的包含 4096 个 bit 位，根据每一位是 0 还是 1 来确定相应的页是否有效，在查找大范围连续页的过程中，直接判断 page 中连续的 1 的个数来确定有多少可以合并的连续的页。

（3）在 Redo 中即使不存在连续的可以合并的 17 个页，但是存在比如 9 个以上的附近的页，是不是可以考虑先从后端读取上来大块的连续的页，然后和这 9 个页进行合并，然后统一写下去，这样可能这 9 个页只需要两次 IO（一次读，一次写）就 Redo 完毕了，但是不这样做的话，可能需要多次 IO。

2.4.4.2 LogUnit Allocator 优化

当前情况下，如果系统中没有可用的 LogUnit，写 IO 将处于 stall 状态，直到有可用的 LogUnit 为止，改进方案是如果检测到 LogUnit 释放的速度慢于 LogUnit 分配的速度，且当前可用的 LogUnit 的百分比达到某个阈值，则为了快速释放 LogUnit，将这些 LogUnit 给 Dump 到文件中，并在 RawLog 的元数据区记录相关信息，以便在恢复过程中可以正确的恢复之，当然 Dump 之后，如果有关于这些被 Dump 的数据的读的话，也需要能够正确定位到该文件中指定位置，这需要为该文件建立索引，相应的该文件在 RawLog Index 中的索引信息都需要被删除。在 Dump 过程中可以利用 PageCache + sync，当轮到该 LogUnit 执行 redo 的时候，由于是顺序 IO，也可以很快的将该 LogUnit 快速 Load 上来。