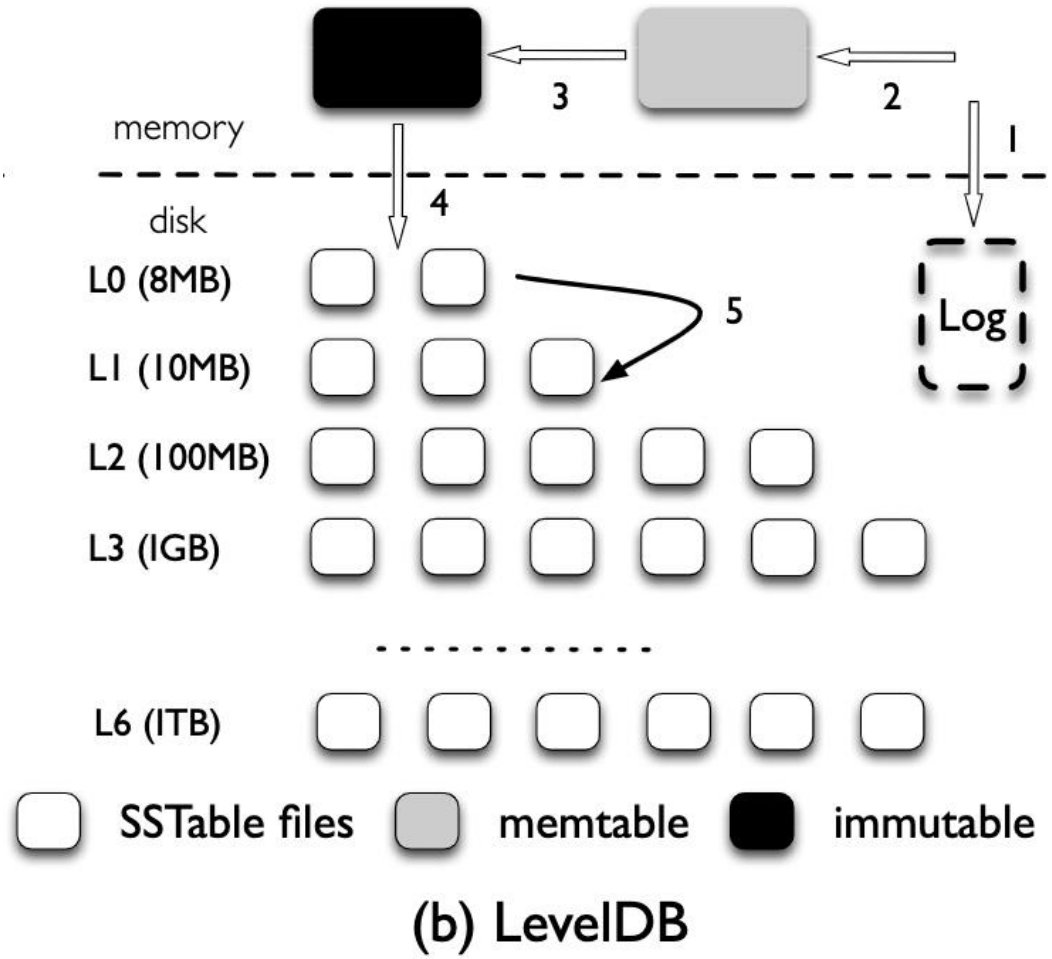
WiscKey 论文阅读笔记

背景

基于 LSM 树（Log-Structured Merge-Trees）的键值存储已经广泛应用，其特点是保持了数据的顺序写入和存储，利用磁盘的顺序 IO 得到了很高的性能（在 HDD 上尤其显著）。但是同一份数据会在生命周期中写入多次，随之带来高额的写放大。



(b) LevelDB

以 LevelDB 为例，数据写入的整个流程为：

1. 数据首先会被写入 memtable 和 WAL
2. 当 memtable 达到上限后，会转换为 immutable memtable，之后持久化到 L0（称为 flush），L0 中每个文件都是一个持久化的 immutable memtable，多个文件间可以有相互重叠的 Key

3. 当 L0 中的文件达到一定数量时，便会和 L1 中的文件进行合并（称为 compaction）
4. 自 L1 开始所有文件都不会再有相互重叠的 Key，并且每个文件都会按照 Key 的顺序存储。每一层通常是上一层大小的 10 倍，当一层的大小超过限制时，就会挑选一部分文件合并到下一层

由此可以计算出 LevelDB 的写放大比率：

1. 由于每一层是上一层大小的 10 倍，所以在最坏情况下，上一层的一个文件可能需要和下一层的十个文件进行合并，所以合并的写放大是 10
2. 假设每行数据经过一系列 compaction 最终都会落入最终层，每层都需要重新写一次，那么从 L1 到 L6 的写放大为 5
3. 加上 WAL 和 L0，最终写放大可能会超过 50

另一方面，由于数据在 LevelDB 中的每一层（memtable/L0/L1~L6）都有可能存在，所以对于读请求，也会有一定的读放大：

1. 由于 L0 的多个文件允许有数据重叠，所以最坏情况需要查询所有文件
2. 对于 L1 到 L6，因为数据有序且不重叠，所以每层需要查询一个文件
3. 为了确认 Key 是否存在，对于每个文件都需要读取 index block、bloom-filter blocks 和 data block

论文中提供了一个实际的数据：

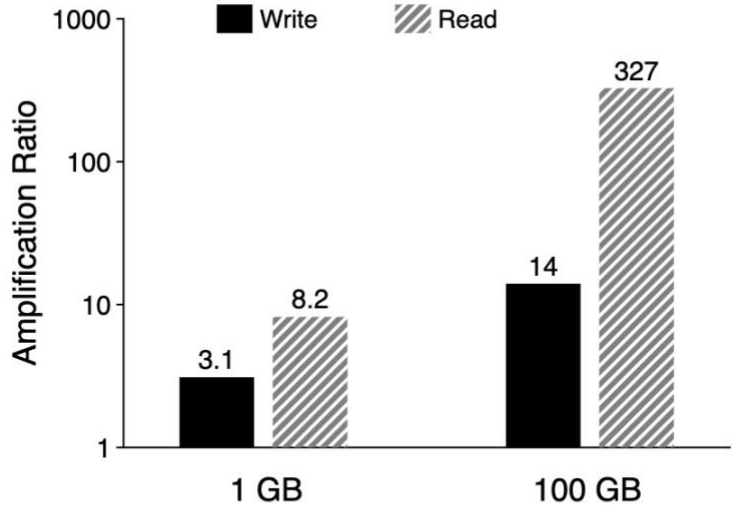


Figure 2: **Write and Read Amplification.** *This figure shows the write amplification and read amplification of LevelDB for two different database sizes, 1 GB and 100 GB. Key size is 16 B and value size is 1 KB.*

WiscKey 介绍

设计目标

WiscKey 的核心思想是将数据中的 Key 和 Value 分离，只在 LSM-Tree 中有序存储 Key，而将 Value 存放在单独的 Log 中。这样带来了两点好处：

1. 当 LSM-Tree 进行 compaction 时，只会对 Key 进行排序和重写，不会影响到没有改变的 Value，也就显著降低了写放大
2. 将 Value 分离后，LSM-Tree 本身会大幅减小，所以对应磁盘中的层级会更少，可以减少查询时从磁盘读取的次数，并且可以更好的利用缓存的效果

另外，WiscKey 的设计很大一部分还建立在 SSD 的普及上，相比 HDD，SSD 有一些变化：

1. SSD 的随机 IO 和顺序 IO 的性能差距并不像 HDD 那么大，所以 LSM-Tree 为了避免随机 IO 而采用了大量的顺序 IO，反而可能会造成了带宽浪费
2. SSD 拥有内部并行性，但 LSM-Tree 并没有利用到该特性
3. SSD 会因为大量的重复写入而产生硬件损耗，LSM-Tree 的高写入放大率会降低设备的寿命

下图展示了在不同请求大小和并发度时，随机读和顺序读的吞吐量，可以看到在请求大于 16KB 时，32 线程的随机读已经接近了顺序读的吞吐：

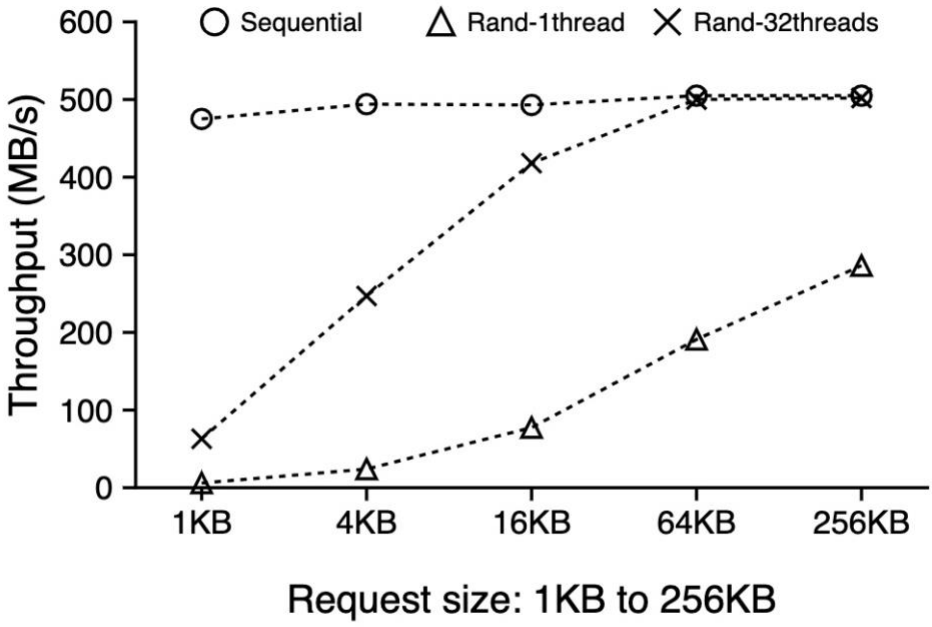


Figure 3: **Sequential and Random Reads on SSD.** *This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100-GB file on ext4.*

实现

在 LSM-Tree 的基础上，WiscKey 引入了一个额外的存储用于存储分离出的值，称为 Value Log。整体的读写路径为：

1. 当用户添加一个 KV 时，WiscKey 会先将 Value 写入到 Value-Log 中（顺序写），然后将 Key 和 Value 在 Value Log 中的地址写入 LSM-Tree
2. 当用户删除一个 Key 时，仅在 LSM-Tree 中删除 Key 的记录，之后通过 GC 清理掉 Value Log 中的数据
3. 当用户查询一个 Key 时，会先从 LSM-Tree 中查询到 Value 的地址，再根据地址将 Value 真正从 Value-Log 中读取出来（随机读）

假设 Key 的大小为 16 Bytes，Value 的大小为 1KB，优化后的效果为：

1. 如果在 LSM-Tree 中的单层的写放大率是 10，那么使用 WiscKey 后单层的写放大率将变为 ((16 x 10) + (1024 x 1)) / (16 + 1024) = 1.14，远小于之前的 10 倍

2. 如果一个标准的 LSM-Tree 的大小为 100G，那么将 Value 分离后 LSM-Tree 本身的大小将会减少到 2G，层级会减少 1~2 级，并且缓存到内存中的比例会更高，从而降低读放大

看上去实现很简单，效果也很好，但是背后也存在了一些挑战和优化。

挑战一：范围查询

在标准的 LSM-Tree 中，由于 Key 和 Value 是按照顺序存储在一起的，所以范围查询只需要顺序读即可遍历整个 SSTable 的所有数据。但是在 WiscKey 中，每个 Key 都需要额外的一次随机读才能读取到对应的 Value，因此效率会很差。

论文中的解决方案是利用上文中所提到的 SSD 内部的并行能力。WiscKey 内部会有一个 32 线程的线程池，当用户使用迭代器迭代一行时，迭代器会预先取出多个 Key，并放入到一个队列中，线程池会从队列中读取 Key 并行的查找对应的 Value。

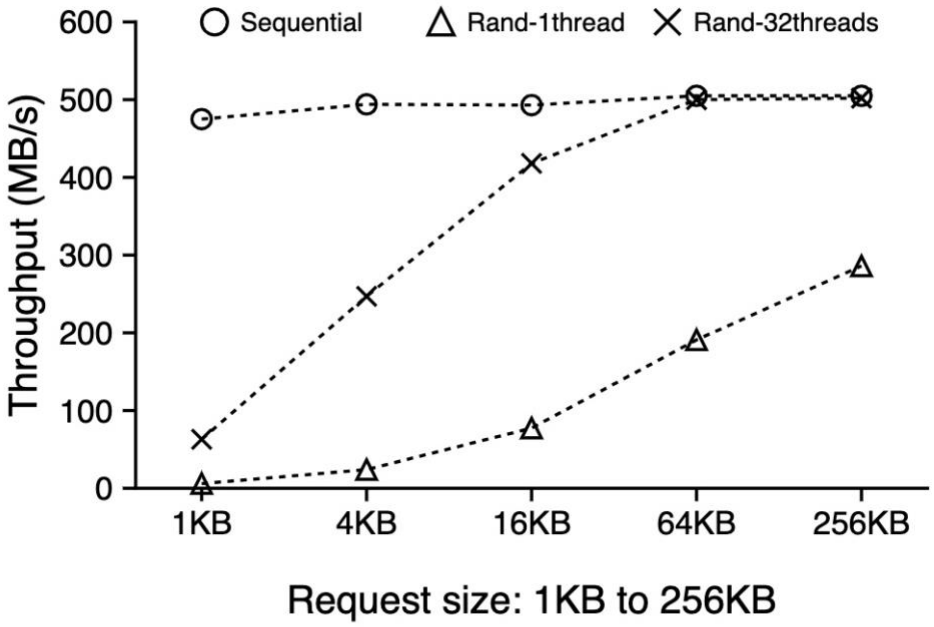


Figure 3: **Sequential and Random Reads on SSD.** *This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100-GB file on ext4.*

疑问：

1. *预取在某些场景是否会有浪费？（用户不准备迭代完所有数据的场景，例如 Limit 或是 Filter）*
2. *为什么用线程池 + 队列，而不是直接用异步 IO？*

挑战二：垃圾收集（GC）

上文中提到了，当用户删除一个 Key 时，WiscKey 只会将 LSM-Tree 中的 Key 删除掉，所以需要一个额外的方式清理 Value-Log 中的值。

最简单的方法是定期扫描整个 LSM-Tree，获得所有还有引用的 Value 地址，然后将没有引用的 Value 删除，但是这个逻辑非常重。

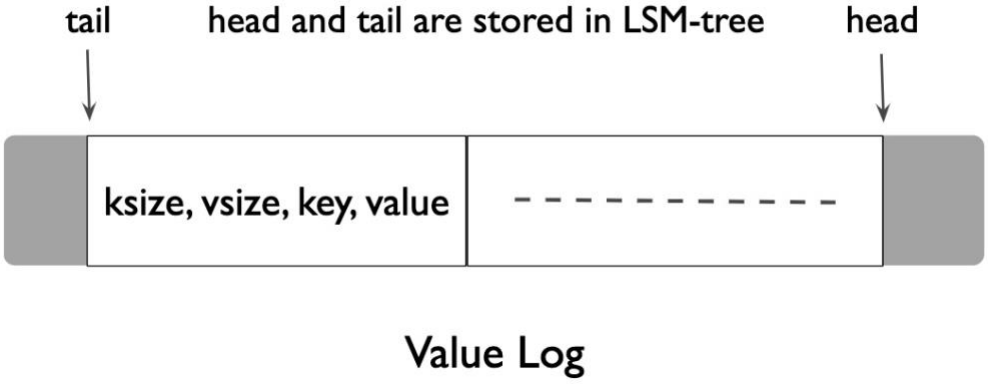


Figure 5: WiscKey New Data Layout for Garbage Collection. *This figure shows the new data layout of WiscKey to support an efficient garbage collection. A head and tail pointer are maintained in memory and stored persistently in the LSM-tree. Only the garbage collection thread changes the tail, while all writes to the vLog are append to the head.*

论文中介绍的方式是通过维护一个 Value Log 的有效区间（由 head 和 tail 两个地址组成），通过不断地搬运有效数据来达到淘汰无效数据。整个流程为：

1. 对于 Value-Log 中的每个值，需要额外存储 Key，为了方便从 LSM-Tree 中进行反查（相对 Value，Key 会比较小，所以写入放大不会增加太多）
2. 从 tail 的位置读取 KV，通过 Key 在 LSM-Tree 中查询 Value 是否还在被引用
3. 如果 Value 还在被引用，则将 Value 写入到 head，并将新的 Value 地址写回 LSM-Tree 中
4. 如果 Value 已经没有被引用，则跳过这行数据，接着读取下一个 KV
5. 当已经确认数据写入 head 之后，就可以将 tail 之后的数据都删除掉了

因为需要重新写入一次 Value，并且需要将 Key 回填到 LSM-Tree 中，所以这个 GC 策略会造成额外的写放大。并且即使不做 GC，也只会影响到空间放大（删除的数据没有真正清理），所以感觉可以配置一些策略：

1. *根据磁盘负载和 LSM-Tree 的负载计算，仅在低峰期执行*
2. *计算每一段数据中被删除的比例有多少，当空洞变得比较大的时候才触发 GC*

挑战三：崩溃一致性

当系统崩溃时，LSM-Tree 可以保证数据写入的原子性和恢复的有序性，所以 WiscKey 也需要保证这两点。

WiscKey 通过查询时的容错机制保证 Key 和 Value 的原子性：

1. 当用户查询时，如果在 LSM-Tree 中找不到 Key，则返回 Key 不存在
2. 如果在 LSM-Tree 中可以找到 Key，但是通过地址在 Value-Log 中无法找到匹配的 Value，则说明 Value 在写入时丢失了，同样返回不存在

这个前提建立在于 WiscKey 通过一个 Write Buffer 批量提交 Value Log（下面有详细介绍），所以才会出现 Key 写入成功后 Value 丢失的场景，用户也可以通过设置同步写入，这样在刷新 Value Log 之后，才会将 Key 写入 LSM-Tree 中。

另外，WiscKey 通过现代的文件系统的特性保证了写入的有序性，即写入一个字节序列 b1, b2, b3…bn，如果 b3 在写入时丢失了，那么 b3 之后的所有值也一定会丢失。

优化一：Write Buffer

为了提高写入效率，WiscKey 首先会将 Value 写入到 Write Buffer 中，等待 Write Buffer 达到一定大小再一起刷新到文件中。所以查询时首先也要先从 WriteBuffer 中查询。当崩溃时，Write Buffer 中的数据会丢失，此时的行为就是上文中的崩溃一致性。

疑问：

1. *根据这个描述，Value-Log 似乎是异步写入？结合上文中崩溃一致性的介绍，会有给用户返回成功但是数据丢失的情况？*

优化二：WAL 优化

LSM-Tree 通过 WAL 保证了在系统崩溃时 memtable 中的数据可恢复，但是也带来了额外的一倍写放大。

而在 WiscKey 中，Value-Log 和 WAL 都是基于用户的写入顺序进行存储的，并且也具备了恢复数据的所有内容（前提是基于上文中的 GC 实现，Value Log 里存有 Key），所以理论上 Value-Log 是可以同时作为 WAL 的，从而减少 WAL 的写放大。

由于 Value Log 的 GC 比 WAL 更加低频，并且包含了大量已经持久化的数据，直接通过 Value-Log 进行恢复的话可能会导致回放大量已经持久化到 SST 的数据。所以 WiscKey 会定期将已经持久化到 SST 的 head 写入到 LSM-Tree 中，这样当恢复时只需要从最新持久化的 head 开始恢复即可。

疑问：

1. *Delete 操作只需要写 LSM-Tree，但如果需要 Value Log 作为 WAL，则 Delete 也需要写入到 Value Log 中*
2. *如果不应用这个优化，则可以做到只将大 Value 分离出 LSM-Tree，应用此优化后，小 Value 也必须要额外存到 Value Log 中了*
3. *与其说是用 Value Log 替代 WAL，不如说是让 WAL 支持读 Value…*

效果

说完实现再看看效果，论文中有 db_bench 和 YCSB 的数据，为了节约篇幅，只贴一部分 db_bench 的数据。

db_bench 的场景分两种，一种是所有 Key 按顺序写入（这样写放大会更低，数据在每一层会更紧凑），另一种是随机写入（写放大更高，数据在每一层分布更均匀）。

顺序写入

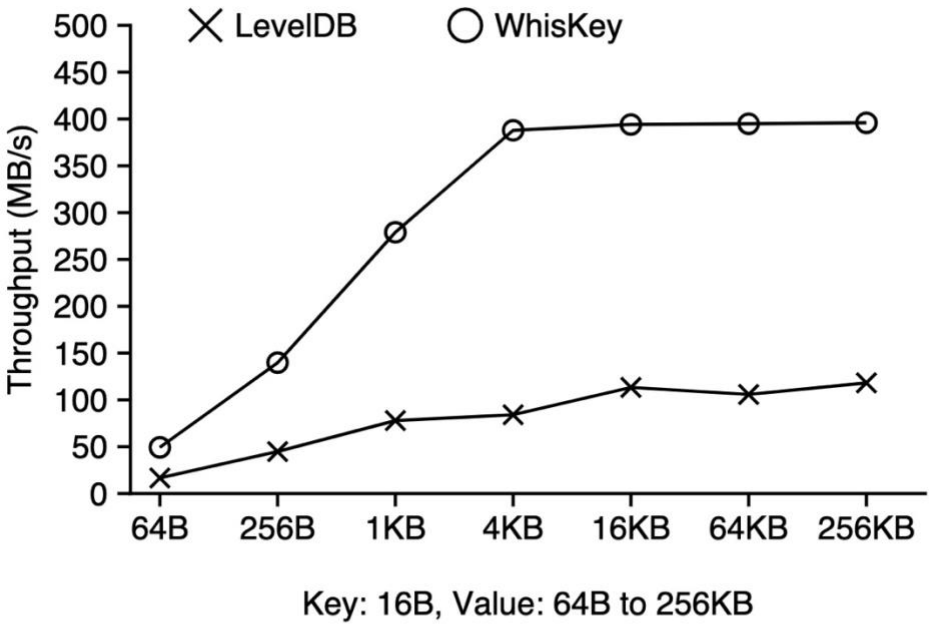


Figure 7: **Sequential-load Performance.** *This figure shows the sequential-load throughput of LevelDB and WiscKey for different value sizes for a 100-GB dataset. Key size is 16 B.*

效果应该来自两部分：

1. *WAL 没了直接省了一倍写放大*
2. *顺序写入，每一层合并可以认为没有写放大，但是数据依旧要在每一层写一次，100 G 可能是 4~5 次（对应 L5 的大小）*

随机写入

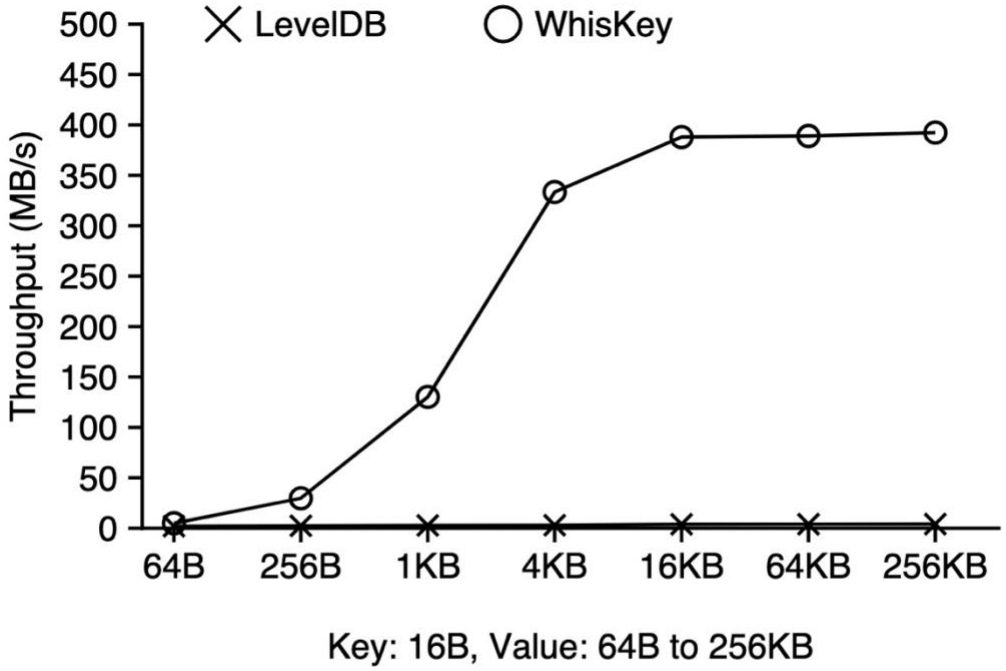


Figure 9: **Random-load Performance.** *This figure shows the random-load throughput of LevelDB and WiscKey for different value sizes for a 100-GB dataset. Key size is 16 B.*

效果对比顺序写入，如果说为什么差距会这么大，只有可能是每一层合并造成的写放大了。

点查

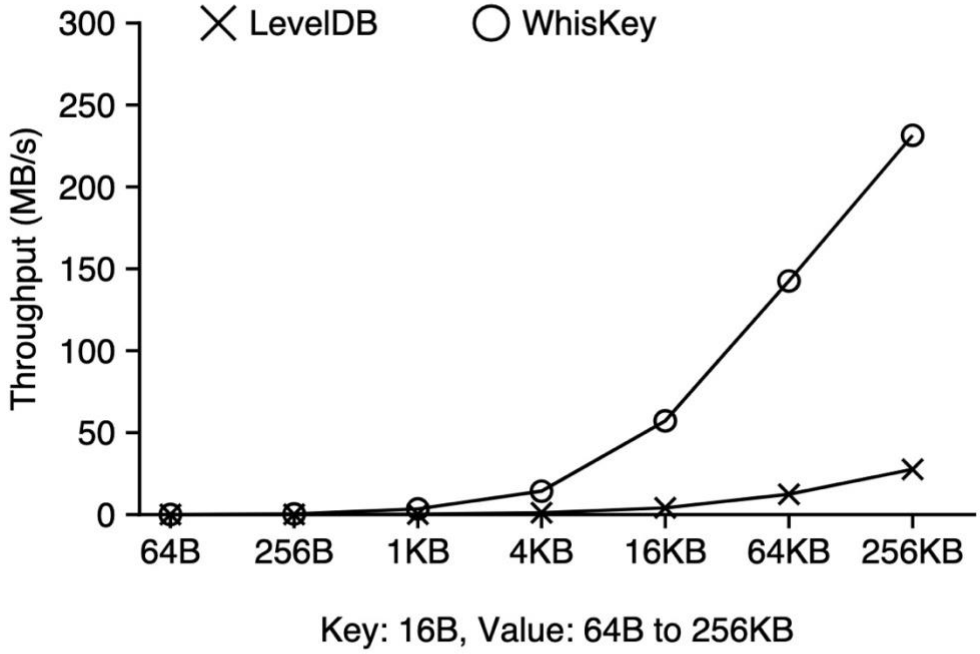


Figure 11: **Random Lookup Performance.** *This figure shows the random lookup performance for 100,000 operations on a 100-GB database that is randomly loaded.*

1. *当 Value 比较小时，WiscKey 的劣势在于额外的一次随机读，而 LevelDB 的劣势在于读放大。当 Value 变得更大时，基于 SSD 内部的并行能力，随机读依旧能读满带宽，但是 LevelDB 读放大造成的带宽浪费却没有改善。*
2. *另外这个测试场景是数据库大小为 100G，对于 LevelDB 来说，层级和 KV 大小挂钩，对于 WiscKey 来说，层级和 Key 大小挂钩，所以当 Value 越大，WiscKey 中的 LSM-Tree 反而更小，层级也就更低，甚至可能仅在内存中 (例如 Value 为 256KB 时，Key 加起来才 100G / (16 + 256* 1024) *16 ~= 610KB)*

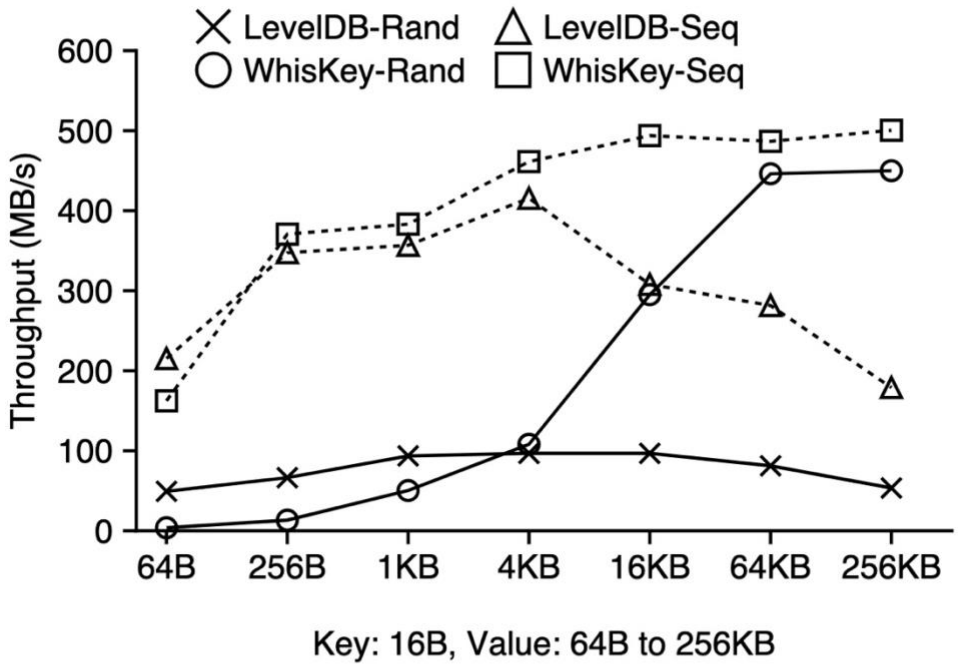


Figure 12: **Range Query Performance.** *This figure shows range query performance. 4 GB of data is queried from a 100-GB database that is randomly (Rand) and sequentially (Seq) loaded.*

这个有点看不懂...：

1. *在数据集是顺序写的场景下，LevelDB 的性能随着 Value 的增大反而降低了，这个不太理解原因（理论上读放大不会很大，而且是顺序读，很容易就能读满带宽），WiscKey 因为是随机读，并且有上文中提到的 LSM-Tree 本身很小，随着 Value 变大性能越高是符合预期的*
2. *在数据集是随机写的场景下，一开始 WiscKey 性能低是因为随机读的延迟，随着 Value 增大，优势应该和点查类似*

GC

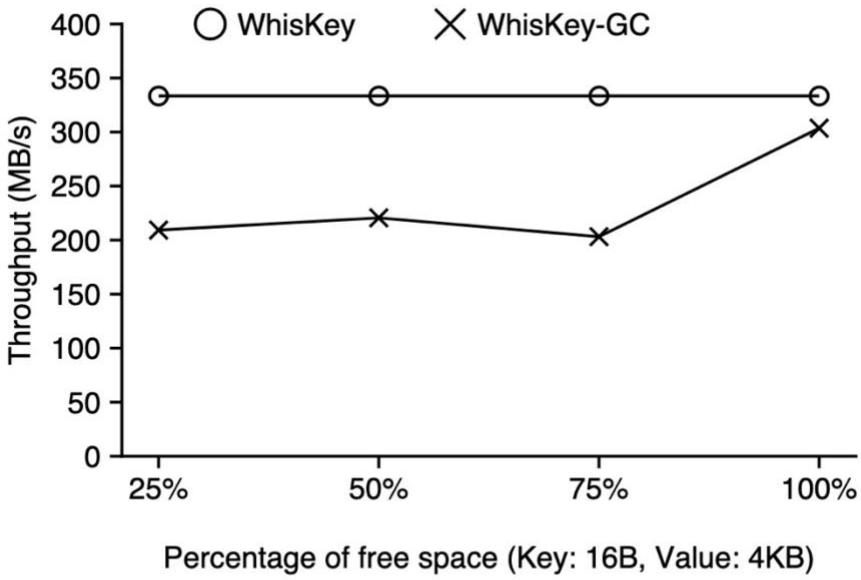


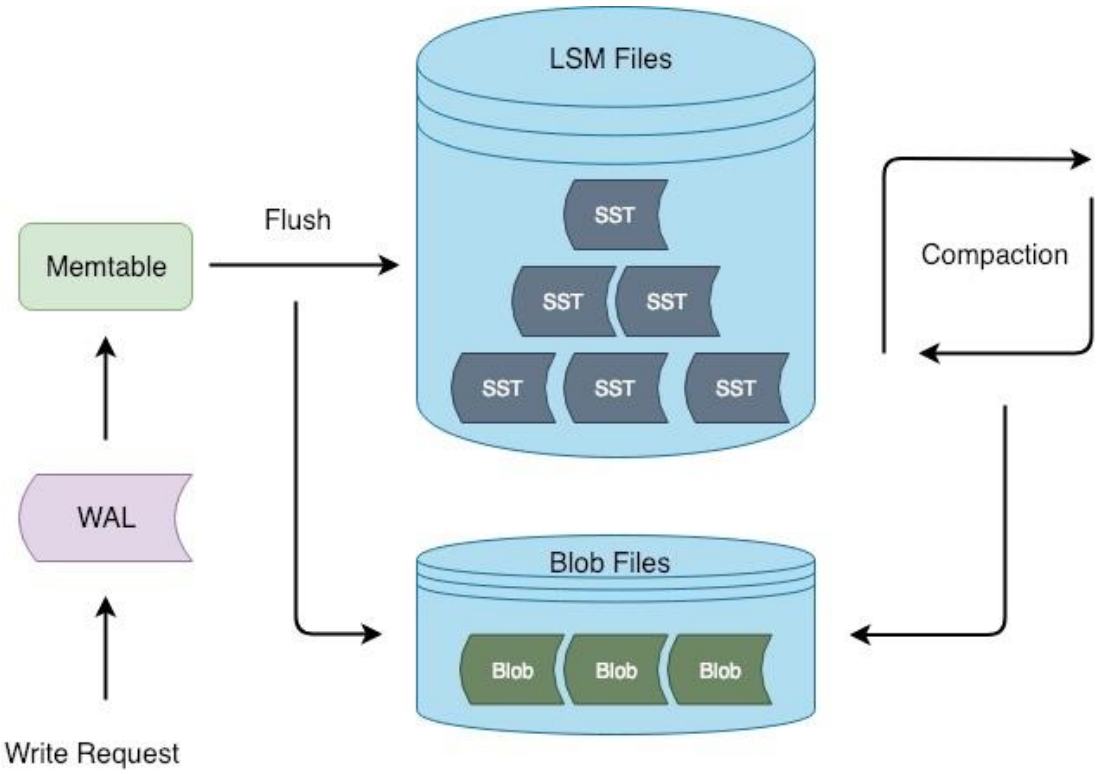
Figure 13: **Garbage Collection.** *This figure shows the performance of WiscKey under garbage collection for various free-space ratios.*

上文提到了 GC 会重写 Value 以及写回 LSM-Tree，造成额外的写入。当空余空间的占比越高时（大部分数据都已经被删了），回写的数据越少，对性能的影响也就越小。

Titan 的实现

BlobDB 和 Badger 的实现都和论文比较接近，并且也都是玩具。反而 TiKV 的 Titan 有一些独特的设计可以学习和讨论，所以下面只介绍这一案例。

核心实现



和 WiscKey 的主要区别在于：Titan 在 flush/compaction 时才开始分离键值，并且用于存储分离后 Value 的文件（BlobFile）会按照 Key 的顺序存储，而不是写入的顺序（其实在这个阶段，已经没有写入顺序了）。

因此导致实现上的差异有：

1. 范围查询：由于 Value-Log 没有按照 Key 排序，所以 WiscKey 需要将一个范围查询拆解为多个随机读。而 Titan 保证了局部有序，在单个 BlobFile 内部可以顺序读，但是会有多个 BlobFile 的范围有重叠，需要额外做归并。另外对于预取策略，WiscKey 建立在 SSD 并行的优势上，可以靠增加并发预取增加吞吐，而 Titan 暂时没有如此激进的预取策略
2. WAL 优化：在 WiscKey 的实现中通过 Value Log 替代 WAL 减少了一倍写放大，而 Titan 在 flush/compaction 时才进行键值分离，肯定是没办法做这个优化的，不过这一点在 Titan 的设计文档里也提到了：「假设 LSM-tree 的 max level 是 5，放大因子为 10，则 LSM-tree 总的写放大大概为 1 + 1 + 10 + 10 + 10 + 10，其中 Flush 的写放大是 1，其比值是 42 : 1，因此 Flush 的写放大相比于整个 LSM-tree 的写放大可以忽略不计。」，个人觉得还是还比较信服的
3. GC 策略：Titan 目前有两个版本的 GC 策略，会在下面详细介绍

GC 策略

第一种策略（传统 GC）：

1. 首先挑选一些需要合并的 BlobFile，在 flush/compaction 时可以统计出每个 BlobFile 中已经删除的数据大小，从而挑选出空洞较大的文件
2. 迭代这些 BlobFile，通过 Key 查询 LSM-Tree，判断 Value 是否还在被引用
3. 如果 Value 还在被引用就写到新的 BlobFile 中，并把更新后的地址回填到 LSM-Tree 中

这个实现和论文中的 GC 方案类似，只不过论文为了 WAL 需要写入一条完整的 Value Log，所以需要维护 head 和 tail。Titan 的实现只需要每次都生成新的 BlobFile 即可。

不同点在于：WiscKey 是随机读，Value Log 的大小不会影响到读 Value 的成本。GC 策略在于写放大和空间放大的权衡，所以 GC 可以更加低频。而 BlobFile 是顺序读，如果 BlobFile 中的无效数据太多，会影响到预取的效率，间接也会影响到读的性能。

第二种策略（Level-Merge）：

1. 不存在单独的 GC，由 LSM-Tree 的 compaction 触发
2. compaction 时，如果遍历到的值已经是一个 BlobIndex（代表值已经写入了某个 BlobFile），依旧将其读出来重新写入新的 BlobFile，也就是说每次 compaction 都会生成一批与新的 SST 完全对应的 BlobFile
3. 目前 Level Merge 只在最后两层开启

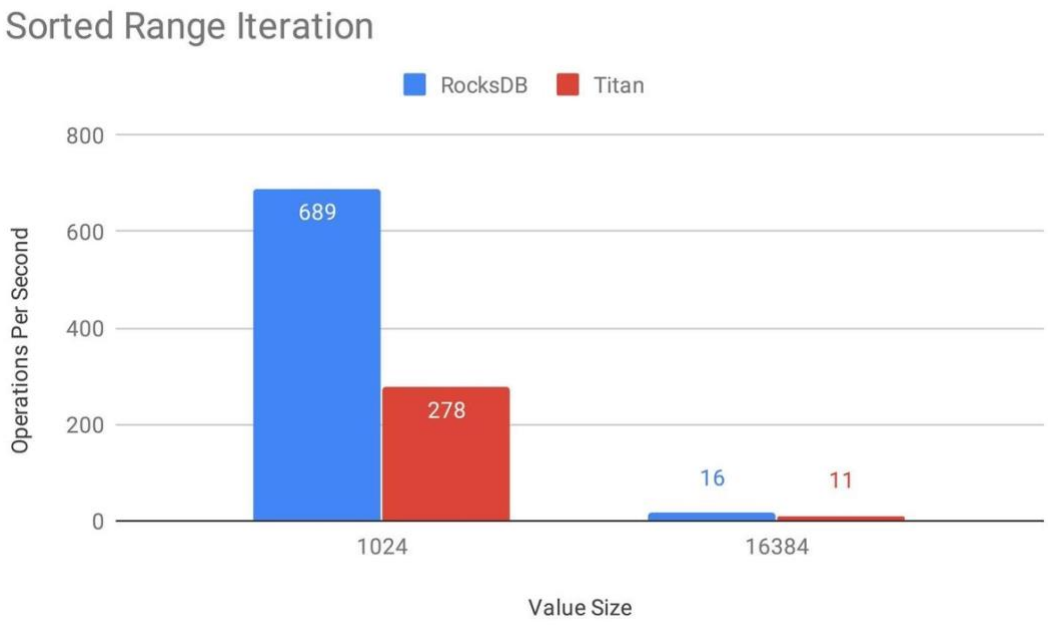
开启 Level Merge 后相当于 GC 频率和 compaction 频率持平了（GC 频率最多也只能和 compaction 持平），并且在这个基础上，直接在 compaction 里做 GC，可以减少一次回写 LSM-Tree 的成本（因为在 compaction 的过程中就能将老的 Value 地址替换掉）。

这种策略的优点在于 BlobFile 中不再有无效数据，可以用更加激进的预取策略提高范围查询的性能，缺点是写放大肯定会比之前更大（个人觉得开启后，写放大就和标准 LSM-Tree 完全一样了吧（一次 compaction 需要合并的 Key 和 Value 都需要重写一遍）？），所以只在最后两层开启。

效果

Titan 的性能测试结果摘自官网的文章，大部分结论都和 WiscKey 类似，并且文章中也分析了原因，就不在此赘述了。

因为文章是 19 年初的，所以还没有上文中的 Level Merge GC，不过 GC 策略理论上只影响范围查询的性能，所以在此贴一下范围查询的性能：



在实现 Level Merge GC 的策略之前，Titan 的范围查询只有 RocksDB 的 40%，主要原因应该还是分离后需要额外读一次 Value，以及没办法并行预取增加吞吐。这点文章最后也提到了：

我们通过测试发现，目前使用 Titan 做范围查询时 IO Util 很低，这也是为什么其性能会比 RocksDB 差的重要原因之一。因此我们认为 Titan 的 Iterator 还存在着巨大的优化空间，最简单的方法是可以通过更加激进的 prefetch 和并行 prefetch 等手段来达到提升 Iterator 性能的目的。

另外在 TiDB in Action 也提到了 Level Merge GC 可以「大幅提升 Titan 的范围查询性能」，不知道除了完全去掉无效数据之外，是否还有其他的优化，还需要再看下代码。

感受

个人认为 WiscKey 的核心思想还是比较有意义的，毕竟适用的场景很典型而且还比较常见：大 Value、写多读少、点查多范围查询少，只要业务场景命中一个特点，效果应该就会非常显著了。

对于论文中的具体实现是否能套用在一个真实的工业实现中，我觉得大部分实现还是简单有效的，但是也有一些设计个人不太喜欢，例如使用 Value Log 替代 WAL 的方案，感觉有些过于追求减少写放大了，可能反而会引入其他问题，以及默认的 GC 策略还要写回 LSM-Tree 也有些别扭。

在和其他同事讨论内部项目的实现时，也畅想过一些其他玩法，例如只将 Value 中的一部分分离出来单独存储，或是一个分布式的 WAL 是否也能转换为 Value

Log，会有哪些问题。包括看到 Titan 的实现时，我也很好奇设计成 BlobFile 这种顺序读的方式是否有什么深意（毕竟论文都把利用 SSD 写到标题里了），或者只是因为从 compaction 才开始分离键值最简单的做法就是按顺序存储 KV。

总之，期待将来能有更多工业实现落地，看到更多有趣的案例。