u表格 LSM-tree的机理

|  |
| --- |
| LSM-Tree的写过程以及compaction    写过程：   * 首先写到MemTable中，如果MemTable满了则转化为Immutable MemTable，flush到硬盘上的L0层(仅仅是flush，没有merge sort)。L0层中的各个SSTable之间的key range是可能会有重叠的(毕竟没有merge sort)。而对于i>0的各层，由于merge sort的关系，SSTables之间的key ranges是不重叠的。 * 如果L0满了，则进行L0->L1的compaction，由此可能引发链式compaction   Compaction  (1) an SSTable in(called a victim SSTable) and multiple SSTables in  who has **overlapping**  **key ranges** (called overlapped SSTables) are picked as the compaction data.   1. Other SSTables in  that fall in this compaction key ranges are selected reversely.   Those SSTables identified in steps (1) and (2) are fetched into memory, to be merged and sorted( older values will be discarded during this process).   1. The regenerated SSTables are written back to .   注意：L0-L1层的compaction有显著不同。Since L0 is unsorted(so that tables in L0 have overlapping key ranges) and each SSTable in L0 spans a wide key range, the L0-L1 compaction performs step (1) and (2) back and forth involving almost all SSTables in both levels, leading to a large all-to-all compaction.  写放大：   * 在写入新的key时，可能会引起compaction，造成写放大现象。这是因为每一次compaction相当于读出原sstables中数据，merge sort以后再写回硬盘。这样一来，系统实际写入的数据要远大于用户写入的数据。(特别是链式compaction，比如L0->L1的compaction导致L1的大小超过阈值，又引起L1->L2的compaction，又导致L2的大小超过阈值，引起L2->L3的compaction, etc)。 |
| LSM-Tree查找key的过程(read 过程)   * 查找过程需要借助manifest，manifest记录了每一个SSTable在哪一层，key range怎么样，如下图所示      * 流程 * 第一，查MemTable，Immutable MemTable * 第二，根据manifest来查找key range包含该key的SSTable.在i>0的每一层，由于有序性的原因，最多只有一个SSTable可能含有该key。而在L0层，则可能有0到多个SSTable含有该key * 第三, 对于key range包含该key的每一个SSTable, 首先检查每一个SSTable对应的布隆过滤器，如果命中，则表示该SSTable可能含有该key，此时读出数据，再一一比对。   Read amplification:   * 查找一个数据可能需要查找多个levels的SSTables * 对于单个SSTable查找一个key-value pair，需要查找index block+bloom-filter block+data block |
| 更新：   * 更新过程转化为写，即将<key, new value>写入LSM-tree中。查找的时候只返回最新值，在compaction的过程中，只保留<key, newest value>，这样就达到了删除旧值的目的 |
| 删除：   * 转化为插入<key,deletion tag>，将该key标记为删除 |
| SSTable的结构:   * 有一个index block，来记录每一个data block的首个key，每一个data block常常为4KB * 有一个filter block，来指示每一个data\_block中可能含有的keys |

|  |
| --- |
| LSM-based Storage Techniques A Survey(这篇文章没有正式发表，里面已经发现了矛盾) |
| 背景：   * 对数据的更新方式有两种，一种是in-place update，例如B+树。一种是out-place update，例如LSM-tree(这是从数据结构的角度出发的，实际在文件系统中完全可以采用write-on-copy方式对数据进行更新)   + B+树     - 优点 * 利于读，因为根据index马上找到值   + - 缺点 * 不利于写，因为可能会引起random I/O * update和delete很有可能会造成碎片化   + LSM-tree     - 优点 * 不会有random I/O，而且将update，delete都转换为写，碎片化程度很低。   + - 缺点 * 不利于读，因为每次读都需要逐层查找。 |
|  |

|  |
| --- |
| LSM-Tree Managed Storage for Large-Scale Key-Value Store |
| 文件系统的文件组织：   * FS index: file metadata(e.g. inode) * User data: resource allocation map(e.g. bitmap)   File system types以及特点   * Update-in place file system: ext4   + 需要更新FS index，而FS index与user data存在不同的地方，需要random access * Log-based file system: f2fs[23],NILFS2[27]   + 更新时FS index和user data都是采用log的方式写，因此更新速度很快。   + 由于采用log的方式更新user data，碎片化严重，以及需要垃圾回收 * Copy-on-write file system: btrfs   问题:借助文件系统存储LSM-tree会有额外开销   * 更新LSM-tree chunk需要进行以下操作   + updates to FS blocks for the LSM-tree chunk data (e.g., 4 I/Os for a 4MB chunk)   + updates to FS blocks for the FS index (at least 2 I/Os, depending on the file system organization). * 更新LSM index需要进行以下操作   + updates to FS blocks for the LSM index   + updates to FS blocks for the FS index   除此以外，文件系统自身可能带有log操作(see page 5 3.3.1)  作者解决方案：   * 从以上可以看出，将LSM-tree通过文件系统存到硬盘上需要多次更新FS index。因此作者提出LSM-tree Directly managed Storage，作用是直接将LSM-tree的数据和它的物理位置对应，去除掉FS inderx。 |
| Benchmarks:   * db\_bench(LevelDB自带的) |

|  |
| --- |
| Mutant Balancing Storage Cost and Latency in LSM-Tree Data Stores |
| Properties of today’s data store   * Access pattern: strong temporal locality, but the popularity fades over time. * LSM-tree designs imply that data that arrives in succession is grouped into the same SSTable, being split off when full. Because of the access patterns, the frequency of which an SSTable is accessed **decreases with the SSTable’s age.** * Each SSTable of which the database is comprised is a portable unit, allowing them to be readily migrated between various cloud storage media. |
| 作者研究对象：   * 将LSM tree存到混合文件系统上，利用高速SSD存hot的SSTable，利用HDD存cold的SSTable。即LSM-tree+混合文件系统。   优化目标：   * Find a subset of SSTables to be stored in the fast storage (optimization goal) such that the sum of fast storage SSTable accesses is maximized, (constraint) while bounding the volume of SSTables in fast storage * SSTable是存在云上的，因此作者的优化目的是为了使得混合存储系统的开销最小。 |

|  |
| --- |
| GearDB A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction |
| 背景   * LSM-tree需要时不时的compaction。 * Garbage collection overhead: 将LSM-tree应用在HM-SMR disks上会导致不用层的SSTable混在同一个zone中，换一句话来说，每一层的SSTables分布在不同的zones中。随着数据的输入，每一个zone的有效利用率有可能下降，直到达到一定阈值/年龄条件(page 4 equation 1)后被垃圾回收。这种方式需要读写的数据会很多，挤占了其余操作的带宽。 * Zone utilization distribution: 综上所述，最好的情况是bimodal distribution，即有数据的zone内的数据都是有效数据，剩下的zone都是空的/充满无效数据。 |
| 解决方案：   * 每一个zone对应一个level，只能存储这一个level的SSTable。无效化这个zone后才能给这个zone分配一个新的level。 * 没有随机写，对一个Level的所有zones选出一个writing zone。需要写入data时，仅仅向这个zone顺序写入 * 给每一个Level的zones设置一个compaction window，该window包含这个level内的部分zones。   具体过程(这里还不是很清晰)：   * 对于i>0，Li内的所有数据都是有序的，**按顺序**分布在不同的zone中   + compaction以后该层的所有数据也要按顺序分布在不同的zones中，实现的方式见下   + 由于L0仅有一个SSTable，实际上L0内也是有序的 * 每一层都有各自的compaction window，compaction window指定了一定的key range内的数据参与compaction，即对应的几个zones。 * 在L(i)->L(i+1)compaction的过程中，回收过期数据，同时将compaction后的数据分为几块：   + Out of L(i+2)的compaction window: 写入L(i+1)   + Out of L(i+2)的key range: 写入L(i+2)   + Within L(i+2)的compaction window: 继续参与compaction     - 当这一部分data不存在时即触发compaction的停止条件     - 当抵达最高层后compaction也停止了。 * 当L(i)层的k个compaction window都走了一遍以后，L(i+1)层的compaction window走一步；当L(k+1)的k个compaction window走一遍后，L(i+2)层的compaction window走一步 |
| Implementation:   * libzbc interface：HGST. Libzbc version 5.4.1. <https://github:com/hgst/libzbc,> 2017. * 13TB HM-SMR disk from Seagate(experimental product, unavailable for customers)   Benchmark   * db bench microbenchmark released with LevelDB * YCSB macro-benchmark suite |
| Open source code:   * https://github.com/PDSLab/GearDB |
| 缺点：   * 适用HM-SMR disk * 写放大问题？ |

|  |
| --- |
| MatrixKV Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM |
| 目标：   * 作者希望同时解决write stall和write amplification。   Write stall的产生原因   * Insertion stall: MemTable已满，此时插入新数据受阻 * Flush stall: L0层已满，将Immutable MemTable写入硬盘受阻 * Compaction stall: 由于compaction占用了带宽引起。在该篇文章中作者发现对于LevelDB，compaction stall主要是由L0-L1层的compaction引起(figure 2)。   + 由于L0是unsorted的，很有可能需要L1层的所有SSTable参与compaction(即all-to-all compaction).   Write amplification的产生原因   * WA与LSM-tree的深度呈正相关，深度越深，WA rates越大。   + 一个数据最终会到达底层，每走到下一层都意味着一次读写   + Compaction的时候，需要对下一层参与compaction的数据进行一次读写。有可能是层数越深，参与compaction的总体数据越多(未证实)。   NVM的读写特点   * compared to DRAM, NVM still has 5 times lower bandwidth and 3 times higher read latency * 可以做到byte level access，而不是像SSD那样只能做到page level |
| 解决方案：   * 将LSM-tree的深度变浅，以减少写放大，但是会导致每一层的数据量变大，这主要带来以下几个后果：   + 导致L0的数据量变大，由于L0的SSTables之间的key range是overlapped，不利于查找。     - 解决方案：作者将L0放在NVM中，并且采用了新的数据结构RowTable。RowTable的forward pointer可以对L0内的数据查找进行加速   + 导致L0->L1的数据量变大，使得write stall的时间变长。作者利用column compaction来弥补。 * 由于作者发现write stall主要是由L0->L1的compaction引起的，因此他希望利用NVM来减少L0->L1的compaction的数据量   + L0放在NVM中，由matrix container管理(为了利用NVM的byte-addressability以及fast random accesses)，L1及以上放在SSD中   + L0的数据结构RowTable中增加forward pointer，用于加速L0中数据的查找。   + L0与L1的compaction变为column compaction，以避免传统的L0->L1的all-to-all compaction，减少compaction的数据量。   Forward pointer:   * 第i个RowTable中的某个key x对应的forward pointer指向第i-1个RowTable中第一个不小于x的key k(k>=x) * 查找的过程从底部的RowTable开始找，forward pointer可以减少上一层RowTable的查找范围   Matrix manager组成：   * Receiver: NVM中利用receiver接受flush下来的memtable，将之变成RowTable(仅有一行的table)，如figure 7a所示，其中Pi为forward pointer，用在cross-row hint search中。当receiver中的rowtables的数量达到一定阈值以后，receiver转变成为compactor，同时创建一个新的receiver用于接收数据。 * Compactor: 如figure 6，compactor的一个column包含在一定key-range内的所有数据。每次L0->L1的compaction使用了L0中的一个column和L1中对应的SSTables。释放出的空间可以由receiver重新利用。由于column不断轮转的关系，最后compactor中的数据会全部利用完。       Column compaction:   * 由于L1层的SSTables是整体有序的，因此两张相邻的SSTable的首个key就组成了一个key range，利用该Key range从compactor的所有row tables中寻找落于该key range之内的KV pairs，形成logical column * 设立了Lower bound和upper bound，当一个logical column内的数据量不够时，利用下一个key range找到下一个logical column，直到满足lower bound和upper bound * 将选出的logical columns与L1中对于的SSTables进行merge sort，将形成的新的SSTable写回硬盘 * Key ranges是轮转的，这样可以充分将compactor中的数据利用完。   注：NVM具有byte-addressability，所以才可以将matrix container竖向分为一个个column |
| 实现：   * 利用PMDK library来访问NVM * 利用POSIX api来访问SSD |

|  |
| --- |
| SILK Preventing Latency Spikes in Log-Structured Merge Key-Value Stores |
| 目标：   * 解决LSM-tree的write stall问题   平滑write stall的方法：   * dynamically allocates bandwidth between client and internal operations according to fluctuations of workloads * SILK allocates less I/O bandwidth to compactions on higher levels during peak client load, and exploits transient low-load periods to boost the processing of internal operations. * Advantages:   + - limit interference between internal operations and client-facing ones     - avoid accumulating over time too large a backlog of internal work, **preventing overload conditions in the long term**(作者发现：短期抑制compaction会导致compaction需求累积，在长期一旦需要compaction，工作量剧增) * gives preference to internal operations that may block client operations(对内部操作引入优先级) * SILK gives first priority to flushes. It ensures that the flushes are fast, making room in memory to absorb incoming updates, which directly affects write latency * SILK gives second priority to L0 to L1 compactions, ensuring that L0 does not reach its full capacity, so that flushes can proceed * SILK gives third priority to compactions on the levels below L1 because, while they maintain the structure of the LSM tree, their timely execution does not significantly affect client operation latencies in the short term. * allows preemption of less critical internal operations(抢占式优先级队列方式) * SILK implements a new compaction algorithm that allows internal operations on lower levels of the tree to preempt compactions on higher levels. |
| Open source code：   * <https://github.com/theoanab/>SILK-USENIXATC2019 |

|  |
| --- |
| Optimization of LevelDB by Separating Key and Value |
| 方法：   * 把key,value分开，分为<key, (offset,length)>，和value，其中前者仍然调用levelDB来存储，后者专门存储在value file中，采用log的方式增加，更新数据。   优点：   * 由于<key,(offset,length)>大小比<key,value>要小很多，因此compaction时的数据量就会小很多，减小了写放大，也会带来write stall的减轻 |
| 缺点：   * 删除value的时候向LSM-tree中插入了<key,deletion flag>值。更新value的时候则是插入了<key, new (offset, length)>，并且向value file的尾部插入了新的value。这两种情况都会造成value file中很多旧的value无效化。如果要对value file进行空间回收的话，则会引起大量<key,(offset,length)>中的(offset,length)更改。作者因此并未进行空间回收 * 只有一个value file。在写的时候需要向LSM-tree和value-file写，多一次I/O。读的时候也是需要额外从硬盘上读一次value-store(但是采用了读缓存的方式缓解) |
| 改进：   * 每一个SSTable对应一个key file，一个value file。compaction的时候也要compact value file。但是我估计这样random write性能会下降，这是因为key file里额外增加了内容，总的参与compaction的数据的量多了 |

|  |
| --- |
| WiscKey: Separating Keys from Values in SSD-Conscious Storage |
| 背景：   * 由于读写放大的原因，将LSM-tree直接应用在SSD上会造成大量的bandwidth被浪费，同时大量增加的写会导致SSD的寿命减少。   方案：   * 跟上一篇文章一样，将key和value分开，由于<key,addr>的大小比<key,value>小很多，因此写放大也减少了。   + 注意，这里的value实际上为<key size, value size, key, value>。      * 由于<key,addr>很小，因此可以全部放在内存中，这样读的时候只要读一次硬盘即可。 * Range query：采用parallel random reads来提高读性能   + 由于value采用log的方式存在SSD上，range query会导致大量的random reads，而SSD的random read的性能比sequential read差一些(最新的SSD非常接近)。需要指出，可以使用多线程的方案提高random read的性能，甚至可以接近sequential read。 * Garbage collection:value log的head负责写入新数据，垃圾回收从尾部开始，每次从尾部读取数据，检查其是否有效，然后将有效数据写到head处。   + 需要查询LSM-tree来确定某个value是否有效   + GC时将数据写到head后需要去改对应的<key,addr>中的addr的内容。 * 由于Value Log记录了key和value，因此不再需要LevelDB的write ahead log。 |
| Benchmark:   * db\_bench，YCSB |
| 不足：   * GC还是需要不断写回旧数据，说明仍旧有写放大现象。并且GC的时候需要不断地查询数据是否有效，实际效率很低   改进：   * 将database碎片化，每一个碎片含有一定范围内的keys，这样一来就可以同时有多个线程对数据库写 |

|  |
| --- |
| Enabling Efficient Updates in KV Storage via Hashing Design and Performance Evaluation |
| 背景   * 基于LSM-tree的KV存储是专门将各种操作转化为sequential writes，利于写。但是这样一来write amplification和read amplification都很大。 * 将key value分开存储的WiscKey并不利于update-intensive workloads，主要原因有两点   + 利用circular log来存储values。GC总是从tail开始的，遇到valid values需要将其写到log的head处，并且向LSM-tree中更新其位置，这引起了额外的写，对写放大有贡献。   + GC需要通过查询来检查一个<key,value>的有效性，而每次查询开销很大 * 将key value分开存储的WiscKey在遇到update-intensive workloads时也会产生接近LevelDB的写放大，而这写放大很大一部分来自Garbage Collection(GC)   + 作者使用的Update-intensive workloads是hot-cold mixed，且符合zipfan分布，这意味大量的数据并不更新，但是需要在GC的时候从tail移到head，同时更新这些数据的位置 * 目的：解决write amplification   方案：  如figure 3所示，仍旧采用Key value分开存储的方案，但是进行了以下几点改进   * 将每一个key都进行hash，由哈希后的值决定其对应的value存在哪一个segment group中   + segment group是在不断增长的。当其用于吸收数据的一个log segment满了以后，则再分配一个新的log segment用以存储新的数据   + 由segment table记录每一个segment group的end pos，这样写的时候就不用再查询LSM-tree了，实现了高效的写。 * 冷热数据分开存储，这样GC的时候就不用重写冷数据了   + 采用了cold data log用来专门存储cold data * 对于value较小的key-value pair，直接将其存在LSM-tree中，不再分开存储   + Compaction overheads for small values并不如big values那样显著   + 查询的时候不用又查LSM-tree又查vLog(value log) * Range scan: 采用read ahead来对范围查找进行加速。如果不采用，那么其查找速度很有可能低于其余LSM-tree based数据库 * 垃圾回收   + 每次选取一segment group来进行垃圾回收，对于key重复的key-value pairs，只保留最新的即可，这是因为写的时候保留了重复key的相对顺序。随后向LSM-tree中更新最新的位置。   + 这样的垃圾回收方式无需借助LSM-tree即可完成，比WiscKey减少了查找开销 |
| 缺点：   * 数据结构不利于range query。但是用了read ahead来进行弥补 |

|  |
| --- |
| PebblesDB Building Key-Value Stores using Fragmented Log-Structured Merge Trees |
| 背景：   * 解决LSM tree的写放大问题，写放大是由compaction带来的，因为每一次compaction相当于读出原sstables中数据，merge sort以后再写回硬盘。这样一来，系统实际写入的数据要远大于用户写入的数据。   目标：   * Low write amplification * High write throughput * good read performance   方案：   * 每一层引入guards，guard用将整一个level的key range分为很多不相重叠的subranges，例如分割成[0,guard1)，[guard1，guard2)...。每一个subrange对应一些sstables，处于一个subrange之中的sstables的key ranges可能会有重叠。 * Compaction的时候对一个subrange对应的sstables进行merge sort，然后再分为一个个sstables，写入到下一层的各个subranges中。   + 这样一来下一层的数据就不用像LevelDB中那样对下一层的数据先读再写了，因此减少了写放大。 * Guards的选择：   + 当一层只有一个subrange的时候，相当于每一层都是无序的，对read和range query很不友好。   + 方案：     - 每插入一个key，这个key有一定概率被选为guard。越高层的level这个概率越高。     - 第i层的key被选为guard以后，第i+1，i+2等更高的层也需要将这个key选为guard * 改进：由于subrange之内的sstables的key ranges之间有重叠，因此get()和range query操作需要检查一个range内的所有sstables，效率较低。注意的是，range operation的底层是seek()和next()   + Bloom filter for each sstable: 这样可以提高get()的性能(但是这个技术也在RocksDB和LevelDB中使用到)   + Parallel seeks：range query的时候用多线程，每个线程负责一个sstable   + Range query在一定条件下会触发subrange内的merge sort(例如seek的次数达到10次)   Limitation:   * See page 9, limitations。说明PebbleDB并不是所有地方都优于HyperlevelDB或者LevelDB，而这是作者用实际数据测出来的，事先不一定想得到 |
| Source code:   * <https://github.com/utsaslab/pebblesdb>   Bench mark   * Yahoo Cloud Serving Benchmark * Db\_bench |

|  |
| --- |
| MemcacheDB Persistent Implementation using LevelDB |
| 背景：   * 比较了LevelDB和BerkleyDB作为Memcached持久层时的性能 * BerkeyDB的底层是B+ tree，而LevelDB的底层LSM tree。 |
| 评价：   * 感觉没啥特别重要的创新，但是发出来了 |

|  |
| --- |
| Redesigning LSMs for Nonvolatile Memory with NoveLSM |
| 背景   * NVM特点   + 可以通过memory bus与cpu连接，这样可以避免使用POSIX interface   + provide 100x lower read and write latency and up to 5x-10x higher bandwidth compared to SSDs   + the read (load) latency of NVMs is comparable to DRAM, but the write latency is expected be to 5x slower. * 将LevelDB直接使用在NVM上无法充分使用NVM的byte * ，capacity。   + 将数据写入NVM的时候仍旧进行了serialization(即将skip list转变成了SSTab)，引起overheads。换句话说，没有利用NVM的byte-addressability，即NVM与内存的访问方式可以很像   + 读取数据的时候需要逐层查找，查找到某block后需要将其读入内存并且deserialize，引起overheads * LevelDB采用了write-ahead log，即数据写入内存前先要写入硬盘上的log，这样做是为了避免system failure造成的数据遗失   解决方案：   * 由于DRAM和NVM都是byte-addressable的，因此作者提出将immutable MemTable存入NVM时无需进行serialization转变为SSTable，这样一来从NVM中读取数据的时候也旧不再需要deserialization。 * 在NVM中设置一个较大的mutable memtable(例如变成四倍)，用以辅助add和update数据 * 原因 * 传统的SSD和HDD上存的数据结构是SSTable，为immutable的，这是因为对其更改需要read-modify-write，并且延迟很高 * 由于NVM是byte-addressable，并且可以通过memory bus与cpu直连，因此可以直接对NVM中的数据进行更新，无需借助memory buffer，也不再需要batch writes。 * 操作方式 * 当内存中的mutable memtable未满时，将数据写入log后再写入内存中的mutable memtable * 当内存中的mutable memtable满后，将其转变成immutable memtable后进行flush操作。 * 在flush时，将新的数据写入NVM中较大的mutable memtable。这样对延迟的影响最小。 * 减少write ahead log * 对内存中的mutable memtable的插入数据需要先log再写入 * 对NVM中的mutable memtable则直接写入，不需要先log * LevelDB是从内存开始一层层查找的，而NoveLSM则是利用了NVM的高速和低延迟，采取了多线程同时查找。 |
| 缺点： |

|  |
| --- |
| LSM-trie An LSM-tree-based Ultra-Large Key-Value Store for Small Data |
| 背景：   * 在单机上用LSM-tree存储大量数据很流行，这样一来导致metadata的数据量很大，不一定可以全部放在内存中，这会导致以下结果：   + LSM-tree查找数据时，需要先从内存开始，再逐层查找，检查一个SSTable的时候需要先读取一个index来确定目标可能所在的block，再检查该block的bloom filter。这意味着查找一个key-value pair很有可能需要多次reads，显著降低了效率。 * 写放大的产生   + 每次将上一层的数据写入下一层都多写了一次上一层的数据   + compaction的时候要不断重复读写下一层的数据，即旧的数据不断被读出并写回 * 由于workload不太具有locality，因此对metadata进行cache的策略不一定有用。对于LSM-tree，每次查找一个KV pair都可能需要读取多层的metadata，造成了严重的都放大 * 目的：解决write amplification，read amplification，very large metadata   方案：   * 每一层内细分为sublevels并采用linear growth，层之间仍旧采用exponential growth   + 某层的sublevels进行merge sort，形成下一层的sublevel。     - 由于没有下一层的数据参加，因此有效减少了write amplification * 采用LSM-trie的方式组织数据，见figure 2。   + 每一层都由诺干个TableContainer组成，相当于sublevel。每一个TableContainer含有多个HashTables(不再是SSTables)   + 将每一个key计算哈希值，按照该哈希值的prefix将key value pair置入不同的TableContainer中，并且随着compaction不断下沉     - 采用哈希值意味着key hash分布比较均匀 * 将SSTable替代为HashTable(减少indices)，该HasHTable有以下几个特征。   + 每一个block视为一个bucket，根据key的哈希值的suffix将key value pair放入不同的bucket中     - 采用suffix是因为prefix已经用于决定该key-value pair位于哪个TableContainer中   + 对于较大的KV pair，则是用特殊的一个bucket来存储，并做好index   + 由于负载不均衡，需要将overloaded buckets内的数据移动到underloaded buckets中，同时写入移动信息，并将负载最重的前20%的buckets的移动信息存在内存中，方便查询。   + 每一个block有一个对应的bloom filter，用来加快查询。 * 采用BloomBuster的方式管理bloom filters，即将一个TableContainer中各个HTable内的同一个序号的bucket所对应的bloom filter存储在一个block中，方便一次读取。   + 这是因为一个KV pair在不考虑迁移的情况下，只可能出现在同一序号的buckets中。 |
| 缺点   * 不是有序的数据结构，不利于range query，但是作者指出并不是所有Key-value storage都需要支持 * 貌似每次compaction都要计算hash，这样会造成大量CPU资源的浪费。但是也可以通过额外存储hash的方式解决。 * 没有仔细考虑哈希冲突的问题。但是哈希冲突貌似不会对基础操作有影响 * HashTable里各个bucket之间的负载均衡的方式很复杂，需要先按照负载量排序再迁移。由于存在迁移，查询的时候需要查询多个bucket，实际上开销很大。 * HashTable里对各个bucket的追加数据相当于in-place update，在文件系统和FTL的参与下行为模式无法确定 |

|  |
| --- |
| BuildingWorkload-Independent Storage with VT-Trees |
| 背景：   * 存储有两种基本的workloads，一种为file workload，一种为database workload(small tuples)。其分别用file system和database存储。 * 作者发明了一个文件系统，可以同时支持file system和KV storage   解决方案：   * 实现了一个文件系统，前台叫做KVFS(Key Value File System)，后台叫做KVDB(Key Value Database)。   + KVFS的一部分使用FUSE实现的，为了POSIX compliant，作用是将文件系统的操作命令例如ls, remove等转换为KVDB的get, update, delete等操作。同时KVFS也通过socket实现了key-value storage的接口，来直接处理get，update，delete等操作。   + KVDB则是利用mmaps直接操作一个存储在ext4上的文件 |

|  |
| --- |
| TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores |
| 背景：   * 减少LSM-tree的写放大 * 由于将MemTable冲入硬盘的触发条件是log达到一定大小，因此当数据冷热不均比较严重时，log的大小达到临界条件，但是MemTable由于吸收了大量相同key的更新，大小远小于maximum size，此时将MemTable冲入硬盘是不合算的。 * L0->L1的compaction   + 当L0仅有一个SSTable时，每一次flush都会导致一次L0->L1的compaction   + 当L0有多个SSTable时，不同的SSTable之间很有可能共有同样的keys，而这意味着对处于不同L0 SSTables中的同一key执行多次L0->L1 compaction。这一现象在数据冷热不均更加严重时较为突出。 * Commit log和flush意味着同一数据很有可能写两次(可能写两次是因为有的数据在更新的过程中被MemTable覆盖了)   方案：   * 把hot key保存在内存中，这样就可以显著减少compaction的次数 * 延迟L0->L1的compaction，仅当L0的SSTables与L1的SSTable的重叠度达到一定程度或者L0的SSTables的数量达到上限时才触发L0->L1的compaction * 为了不将MemTable冲入L0，作者直接将Commit Log内的数据转变成CL-SSTable。由于Commit Log不是有序的，因此还需要将key-value pairs对应的indices冲入L0层。   实现：   * 在RocksDB上实现 |
| Available at:   * https://github.com/epfl-labos/TRIAD |

|  |
| --- |
| A Light-weight Compaction Tree to Reduce IO Amplification toward Efficient Key-Value Stores |
| 方案：   * L(i)的SSTable中需要保存有L(i+1)层的overlapping的SSTables的metadata。方式是L(i+1)层的SSTables中的metadata更新后，将更新后metadata写入上一层的overlapping的SSTables * Compaction的时候根据victim SSTable的自己的metadaa和保存的overlapping的SSTables的metadata，将数据写入对应的下一层的SSTables，写入的时候需要覆盖掉原始的metadata   + 这样做是为了减少compaction的时候需要读取overlapping SSTables的metadata(这一读取过程是随机读)   + After finishing the light-weight compaction, the updated metadata of overlapped DTables in level L(i+1) which has been merged recently in memory is aggregated and stored in the **victim DTable** in level L(i)   + 只有每一层的victim DTable才有下一层overlapping SSTables的metadata * 如何保持SSTable之间的大小差别不太大：相邻两个SSTable之间转移data(感觉与后面的SMR disk的应用出现了描述矛盾)   应用于SMR：   * 应用在SMR disk上时，直接用数据库负责管理空间的分配，每一个DTable都专门分配一个band。 * 由于每一个band都不适合随机写，而更适合用log的方式写。因此该篇文章中的写方案可以避免L(i+1)层的数据重写，又可以避免对SMR disk的随机写。换一句话说，该实施方式既可以减少数据库内部的write amplification，又避免了SMR disk自身引起的write amplification * 当某个DTable内的数据过多时，数据库读出该DTable，将其分解为多个小的DTable，写入对应的band * 不采用对该DTable进行compaction的原因是为了避免引发更高层级连锁compaction |
| 缺点：   * segments之间会有overlapping，不利于读，例如figure 19 b 中的random get * 重分已满的DTable也会导致额外的写 |
| 硬件：   * 1TB Seagate ST1000DM003 HDD * a 5 TB Seagate ST5000AS0011 SMR drive * a 400GB Intel P3700 SSD   注：   * SMR disk是用于leveldb-smr这个对照组的 * LWC-smr中的smr是用skylight模拟出来的，因为host-managed smr还没有 |
| * 总感觉这篇文章前面的描述和应用于SMR上出现了矛盾 |

|  |
| --- |
| Leveraging NVMe SSDs for Building a Fast, Cost-effective, LSM-tree-based KV Store |
| 背景   * 更改RocksDB，使之可以用于NVMe SSD(Speed Disk)和大容量硬盘(Capacity Disk)组成的存储系统，这样既可以利用NVMe SSD的high bandwidth、low latency，也可以利用普通硬盘的大容量和低造价 * Storage Performance Development Kit (SPDK)：由于SSD的速度越来越快，file system和Linux kernel所引起的latency overheads再也无法忽略，因此intel开发了这一工具来避免以上的overheads |

|  |
| --- |
| An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD |
| 背景：   * LevelDB的实现方案是针对HDD的，HDD仅有一个磁头负责读写，而且顺序读写速度比随机读写快很多。而现在的SSD都是multichannel的，想当于多个独立的磁头同时对硬盘进行读写。 * 过去的multichannel SSD内部的channels对外部不可见，仅仅提供了一个block device interface。这也意味着其内部的scheduling策略对LSM-tree而言不是最优的，会影响性能。而SDF作为一种特殊的SSD，其内部的channel对外部的应用可见。 * 作者专门针对SDF这一种SSD，改进了lsm-tree。   SDF(Baidu内部使用的一种SSD)   * SDF的内部channel对外部可见，一共有44个channels(/dev/ssd0,/dev/ssd1,.../dev/ssd43)。 * SDF的读写   + Write unit大小为2MB，刚好为一个block的大小，且写的时候需要提供block的地址(即每次写一个至少一个Block的数据)。这样一来，每个Block内部不会出现valid data和invalid data互相交织的情况，但是会出现block没有写满。   + SDF的Read unit大小为8KB，这样可以利用SDF的随机读的性能。 * SDF的erase操作是可以人为操作的，可以更好的scheduling   方案：   * 由于多个channels是可见的   + 可以增加background threads来负责flush和compaction   + 增加immutable memtable的数量     - 当memtable中的数据达到2MB后(1个block的大小)，将其转换成immutable的。将write request插入某一个channel的request queue中等待写入   + 增加L0中SSTables的数量，从而减少flush操作对写性能的影响     - 由于L0中的SSTables之间的值是可能互相重叠的，过去为了减少从L0中读取数据的开销，限制了L0中SSTable的数量。而由于SDF存在多个channels的关系，可以同时读而不影响性能，所以可以增加L0中SSTable的数量。   + 同时运行多个compaction，但是过多会互相干扰，因此作者仅增加了一个。 * 利用PCM或者batttery backup DRAM来存储Write Ahead Log，这对write intensive workloads有效 * Compaction的时候注意将产生的SSTables尽量分配到含有的下一层SSTables不与之overlap的channels中，这样下一次compaction的时候可以提升读取效率。 |

|  |
| --- |
| SpanDB A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage |
| 背景   * 优化了LSM-tree在NVMe SSD上的实现，采用了Intel SPDK interfaces而不是ext4文件系统来绕过Linux I/O stack，从而提升性能 * SPDK (Storage Performance Development Kit) : a set of user-space libraries/tools for accessing high-speed NVMe devices(用来避免linux I/O stack带来的开销)   方案：     * 将L4及以上的数据存在CD(Capacity Disk)上，将L0~L3文件以及WAL area存在Optane SSD上，并且采用了SPDK来读写 |