|  |
| --- |
| Key-value设计要求   * Low latency and fast response * High hit rates(当作为缓存使用的时候)   要求的接口(可选择实现)   * Insert * Get * Delete * Update：可以被insert(key,new value)代替 * Range query |
| Locality metrics:   * How often and how much some keys repeat in requests * **high concentration** of repeating keys provides the justification for caching them in the rst place * The amount of unique keys and how it varies over time   + 不同的workload的特征不一样，这意味着需要针对不同的workload调整参数 * Reuse period, as a measure of temporal locality |

|  |
| --- |
| KVell the design and implementation of a fast persistent key-value store |
| 背景   * 在传统的硬盘上存储Key-value pairs时硬盘的速度是bottleneck，因此可以多利用CPU以减轻对硬盘的负担，例如LSM-tree采用了merge sort的方式使得i>0,L(i)层的所有数据都有序。而在block-addressable NVMe SSD上，由于其读写性能很高，CPU才是bottleneck，即CPU用满的情况下SSD却没有达到最大的throughput，因此需要设计降低CPU负担的策略。 * 设计了有别于传统的基于LSM-tree和B-tree的key-value store。基于LSM-tree和B-tree的系统都需要大量的maintenance，KVell的目的是避免这些maintenance，从而降低CPU的负担。 * KVell是专门针对NVMe SSD设计的，特点是对CPU的负担低。 * NVMe SSD的random accesses性能跟sequential accesses性能很接近。在相同的延迟下，NVMe SSD能同时处理的requests数量也远比以前的SSD多(256 vs 32) * NVMe SSD的读写速度非常快，例如三星的NVMe SSD可以达到顺序读3500MB/s，顺序写3000MB/s   解决方案   * Share nothing: 每一个thread负责所有keys的一个subset   + 减少synchronization带来的CPU开销 * 采用log的方式直接向硬盘上写KV pairs，同时每一个worker都保有一个B-tree作为index，记录每一个KV pair的位置   + 优点：不会引起in-place update，开销小速度快   + 缺点：会导致successive keys不在SSD上连续存储，但是NVMe SSD的随机I/O性能与连续I/O性能很接近，因此这影响不大 * 由于NVMe SSD的random access性能与sequential access的性能很接近，因此KVell对requests做batch的目的是为了减少system call的数量，从而减轻CPU的负担。LSM-tree对requests做batch的目的则是为了利用sequential access。 * 没有commit log(即write ahead log)。这样使得对磁盘的I/O都是写有效数据   数据结构：   * 每一个worker利用各自slabs来存储数据(items)，每一个slab存储一定范围size range中的数据(如同memcache里的slab一样)   + 每一个slab分成很多个blocks(即pages，约为4KB)   + Insert: see page 6 5.2   + Update: see page 6 5.2 * 每一个worker都带有一个B-tree，指示<key,value>在硬盘的哪一个位置。B-tree存有key的prefix，这样可以排序(利用hash value of a key的话没法对key进行排序)。   + B-tree在本篇文章中并未由作者实现仅将部分数据写入硬盘，但是由于是利用mmap分配的，操作系统kernel具有自动将该数据缓冲到硬盘的功能   + B-tree虽然在内存中，但是可以重建 * Page cache: 对page有一个LRU组织的cache，把hot page放在内存中。该page cache的基本结构也为B tree。 * Free list( for garbage collection): 记录了slab文件中哪些区域的item是被删除的，即哪些区域是可以重新利用的。   结果：   * 由于KVell是专门针对最新一代固态设计的，因此在最新一代的固态上KVell优于RocksDB等软件，但是在旧的SSD上则反之。 |
| Benchmark:   * YCSB * 私有数据集 |
| Source code   * https://github.com/BLepers/KVell. |
| 问题：   * KVell的性能提高是不是因为采用了新的异步I/O？ * KVell所占用的内存是不是比其它的例如RocksDB要多得多？ * KVell insert和update数据的时候貌似没有考虑FTL和File system的影响，貌似是可以直接对硬件进行操作 * KVell的update直接将数据写入SSD，实际上会引起大量的in place update。   缺点：   * 把index全都放在内存中，说明适用于较少的数据。 * 虽然mmap可以将index部分置于硬盘上，但是仍然是断电以后就失去index，说明可靠性有问题。 * 由于key不是连续存储的，因此不利于范围查找，但是由于新硬盘的random access的性能与sequential access的性能很接近，因此只要利用好index，范围查找性能几乎没有降低。 * 没有commit log，感觉牺牲了可靠性 |

|  |
| --- |
| SILT a memory-efficient, high-performance key-value store |
| 解决方案：   * Log store   + 将key-value按照log-structured data存在flash上，内存中存有hash table，该hash table的类型为cuckoo hash   + Hash table中仅仅存有key的tag，这是因为tag的长度较小，有利于多存数据到内存中。     - 该tag为alternative bucket for that key，这是因为如果出现哈希冲突，那么为旧的key需要重新计算hash table的地址，而这需要读取一次flash，降低了效率；     - 如果命中，需要从flash中读一次key，进行一一比对   + 一般的cuckoo hash在50%的entries被占后就有可能出现无限kick out。而该log store使用的cuckoo hash中每一个bucket内部都有4个entries。   + 当log store满了以后(例如插入一个tag以后经过maximum displacements后仍结束)，则将该log store冻结，转化为Hash store。同时新开一个log store以吸收新的数据      * Hash store   + 本质上为一个on-flash hash table，存储的为key-value，还是利用cuckoo hash实现，   + Memory-efficient hash filter: 即前面log store对于的在内存中的hash table，但是去掉了offset。见下图      * Sorted store   + 使用prefix tree(trie)来index所有排序完的key-value pairs。排序是根据key来的。该tree每一个节点为key-value pair的index   + 利用了现有的NSort库来生成该sorted store |
| 缺点   * SortedStore的index需要占用内存，内存资源不一定足够 * 内部操作所占用的时间很多 * 写放大很严重，需要读取Sorted Table，将之与Hash Table合并 |

|  |
| --- |
| FlashStore: High Throughput Persistent KeyValue Store |
| 实现：   * 一种Key-value存储方案，该方案利用flash存储处于working set中的key-value pairs，同时随着workload的变化，对flash中key-value pairs进行替换，将cold的key-value pairs存到HDD上(用Berkeley DB存储)。 * 该存储方案是SSD friendly的，通过减少erase操作增加SSD的寿命。实际上即使用log的方式存储数据，每次存约为一page大小的数据 * 作者利用Berkeley DB将key-value pairs存储到HDD上，用Bloom filter以减少对HDD的访问 |
| * 这篇文章里对SSD的key-value的组织跟ChunkStash里很接近 |

|  |
| --- |
| SkimpyStash RAM space skimpy key-value store on flash-based storage |
| 背景：   * 利用flash来存储key-value pairs，在内存中设置一个RAM Hash Table Directory作为index * 对flash的写方案得是flash friendly的，即避免small random writes   Base design：   * 内存中存有哈希表作为index，解决哈希冲突的方式为linear chaining，将该链表存在flash上   + 查找一个Key的时候很有可能会引起多次I/O。设该链表长度为k，则查找一个key平均需要k/2次I/O   Improved design:   * 内存中的哈希表的每一个bucket都带有一个布隆过滤器 * 为了使得buckets对应的chain的长度相似，作者采用了两个哈希函数，将key插入较短的chain * One byte for counting，这也意味着每一个链表中最多放255个key-value pairs * Compaction：将一个bucket中valid key-value pairs都集中存储在一个/一些flash page中，这样每次读取一个flash page就相当于读取读取多个key-value pairs * Garbage collection:由于update，delete以及compaction会引起很多key-value pairs无效化。   + 对于一个key-value，通过查找的方式来判断其是否过时。如果返回的value不等于该value，说明其过时。这样也可以处理掉那些由compaction造成的orphaned key-value。(注：orphaned key-value pairs指无法由HT-directory到达的key-value pairs)   + 如果一个bucket chain的某个record是需要被回收的，则将这整个chain进行compact操作以及垃圾回收。     - 如果不这么做，意味着需要更改上一个record的指针，这会引起flash的in place update。而为了避免in place update，往往需要将上一个record写入新的位置，这又会引起上上个record需要改变，因此索性直接对整个bucket chain进行垃圾回收和compact |
| 不足与改进：   * 感觉不用compaction。每一个Bucket对应一个写缓存，该写缓存大小为一个page size，写满以后就直接写入一个flash page。这甚至可以导致不采用linkedlist结构，去掉地址指针所占的空间。但是这样做会使得写缓存需要超级大的空间 * Garbage collection的时候直接采用稳定的排序算法排序，这样同样的key都会排在一起，取最新的就可以了 * Garbage collection的触发条件是插入超过一个阈值的key-value pairs。这一触发条件对于基本不更新的workload并不适用 * SkimpyStach的range query能力很差，估计需要读取整个数据集 |

|  |
| --- |
| SLM-DB: Single-Level Key-Value Store with Persistent Memory |
| 背景：   * Persistent memory(Non-Volatile Memory)具有非易失性，byte-addressable。 * Write ahead log得名是因为在以前的LSM-tree数据库中，需要先向硬盘上的log写入数据后再将数据写入内存。这样做是为了防止机器故障导致数据丢失 * Persistent Memory(PM)可以通过memory bus与cpu相连，而不是像SSD那样，通过block interface   实现：   * 把MemTable和immutable MemTable都放在persistent memory中。这样一来也可以去掉write ahead log * 其余数据仍旧放在Disk中，数据的组织形式为SSTable。   + 当把一个Immutable MemTable转变为SSTable存入disk后，在Persistent Memory中的B+树内插入对应的<key,location>，都完成以后在MANIFEST中写入对应的log * Global B+-tree用来存储所有key以及其对应的location。   + 由于其为B+-tree，可以很方便实现一个iterator，用以循环访问所有的<key,value> * Selective compaction:   + Candidate list for compaction     - the live-key ratio：将有效率低于一个阈值的SSTable选入candidate list * 这样做是为了Garbage collection   + - the leaf node scans in the B+-tree：采用round-robin的方式对B+-tree扫描一定数量的keys，当这些keys对应的SSTables的数量高于leaf node threshold，将这些SSTables加入candidate list。 * 这样做是为了提前保持一定的sequentiality，利于范围查找。   + - the degree of sequentiality per range query：当每次进行range query的时候，将这个range分成subranges，计算每一个subrange对应的SSTables的数量，如果该数量高于sequentiality degree threshold，将这些SSTables加入candidate list。 * 这样做是为了针对range query优化sequentiality   + 步骤     - 计算candidate list每一个SSTable与其余SSTable的overlapping ratios。     - 选出the sum of overlapping ratios最大的SSTable s，将之与与之overlapping的SSTable进行merge sort * 这样做是为了控制参与merge sort的SSTables的数量   + - compaction时，产生新的SSTable。每产生一个SSTable，都向一个queue中插入对应的<key,location> pairs，然后对B+-tree树进行更新     - Compaction完成后，向MANIFEST中写入对应的数据，然后再删除旧的SSTables |
| 讨论：   * 该篇文章中NVM中的MemTable的数据结构就是Skip List，不是SSTable。这一点与NoveLSM一样 * 有效减少了compaction的数据量，内部操作占用的I/O量下降，利于其余操作   缺点：   * compact的时候需要对每一个key都去查找B+-tree以确定其是否过期，我认为改进方案可以为额外插入一个timestamp段，这样compaction的时候对同样的key，只要保留最新的就可以 * 没有像LevelDB那样保持数据的连续性，从而range query性能下降。而数据的连续性利于range query(因为prefetch策略有效)，这也在实验中可以体现。 |

|  |
| --- |
| Atlas Baidu's key-value storage system for cloud data |
| 背景：   * 百度原先的存储系统是由x86 CPU和9块HDD组成的。而对workload进行分析，requests中的大部分都是对128KB以上的数据进行访问。这导致硬盘才是bottleneck，CPU的使用率很低。   + 这意味着不需要很强大的CPU。便宜9/10的ARM芯片就足够   + 加多硬盘数量虽然可以提高CPU的利用率，但是会造成以下副作用：单个节点存有过量数据，减慢了节点宕机后数据的恢复   + 具体的架构为ARM CPU+4GB RAM+3TB 硬盘，由于内存容量较小，不适用将key-value pair当作文件存在文件系统中，这是因为metadata无法完全存在内存中。作者采用了将数据存储在LSM-tree中的方案。   解决方案：   * 架构如fig3所示。其分为PIS Slice(Patch and Index System Slice)模块和RBS Partserver(RAID-like Block System Partserver)模块。   + 数据首先分为至多256KB大小的块，再通过SHA1 hash计算出该块的key，再根据md5(key) mod n计算出该数据需要发送到哪个PIS Slice   + PIS Slice用以存储metadata以及接收数据缓存到一个block(64MB)中     - 采用了three-copies的方式保证可靠性。     - 当接收的数据达到64MB以后，存到RBS Partsevers中。     - 每64MB才对于一个记录存储位置的metadata，有效的减少了metadata的数量 * RBS Partsserver用于存储数据，利用Reed-Solomon coding algorithm来保证数据的可靠性 * 存储数据时，将64MB数据分为8个，采用Reed-Solomon coding algorithm处理后共有12个数据块，将其分别存到12个服务器中。这是因为该算法可以保证12个数据块中丢失的数量超过5个时才会无法复原数据 |

|  |
| --- |
| ForestDB A Fast Key-Value Storage System for Variable-Length String Keys |
| 背景   * 对于B+-tree来说，variable-sized key会导致一个情况：当key的length较长时，树会很高，不利于查找、插入等操作。   解决方案：   * HB+ tree，如Fig. 9所示，本质为前缀树的变体。其基本原理是将Key分成一个个固定长度的chunk，以位于同一位置的chunk作为key使用，形成一颗B+树。该B+树的叶子节点记录了子B+树的位置或者数据的位置。   + 该树会面临不均匀生长的问题。chunk的长度越长，HB+树不均匀生长的可能性越低。但是block的容量是有限的，chunk的长度越长，HB+树的fanout越小，长度越高，每次读写需要的I/O数越多。   + 为了防止整颗树的高度不均匀生长，将其分为leaf B+树和non-leaf B+树。其中non-leaf B+树即为以固定大小的chunk作为key的index tree，而non-leaf B+树则是以剩下的不固定大小的postfix作为key的index tree。 |

|  |
| --- |
| Towards Building a High-Performance, Scale-In Key-Value Storage System |
| 背景：   * 专门针对NVMe SSD设计的键值对存储系统，其特点是减少了CPU负担   + 现有的一些键值对存储系统直接应用到NVMe SSD上会导致一个现象，CPU利用率很高但是NVMe SSD的Bandwidth利用率并不高(i.e. CPU成为了整个系统的瓶颈) * 现有的一些键值对存储系统在存储数据的时候需要经历一些overheads   + Write Ahead Log(WAL)   + 文件系统   + 操作系统的page cache   + FTL，STL等转换层 * 现有的一些键值对存储系统需要经历read/write amplification，产生原因主要有：   + Background operations like compaction、数据库自带的garbage collection   + STL自带的garbage collection * 现有的一些键值对存储系统的foreground和background的tasks之间会互相竞争资源，导致foreground task的I/O latency变大和bandwidth变小   方案：   * 去掉WAL，而是采用NVMe SSD内部自带的battery backedusing its battery-backed in-memory request buffers. * 如figure 3(c)所示，应用可以直接对SSD发送读写请求，而不用经过文件系统、操作系统、FTL等转换层 |
| * 在key存入前会对其进行hash，这个方案是否是hash collision resistant的呢？ |