|  |  |
| --- | --- |
| Avoiding the Disk Bottleneck in the Data Domain Deduplication File System | |
| 场景:   * 备份系统，在deduplication的同时需要保证一定的写入速率，否则无法在窗口期完成 * fingerprints无法全部存在内存中，造成disk bottleneck * Backup data with chunk locality   挑战：   * The key performance challenge is finding duplicate chunks. 不引入deduplication的备份系统直接写就可以了。但是引入deduplication后，要先和fingerprints进行比较，然后才决定是否写入。fingerprints的查找效率将直接影响到系统的性能。 * fingerprint的access没有空间相关性   + Since fingerprint values are random, there is no spatial locality in the segment index accesses * fingerprint的access没有时间相关性   + because the backup workload streams large data sets through the system, there is very little temporal locality. Most chunks are referenced just once every week during the full backup of one particular system. | |
| 图 Data Domain File System的结构 | |
| Content Store | 负责objects的byte-level的读写，object可以是file, image, etc |
| Segment Store | Segment Store is essentially a **database** of segments keyed by their segment descriptors.  Segment Indexing: updates the segment index that maps segment descriptors to the container holding the segment, after the container has been appended to the Container Manager  综上：即用数据库存储segment descriptors与container id之间关系 |
| Container Manager | The Container Manager is responsible for allocating, deallocating, reading, writing and reliably storing containers.  每一个container的size是fixed，有以下几个好处   * The fixed container size makes container allocation and deallocation easy. * The large granularity of a container write achieves high disk throughput utilization.(HDD的随机读写性能很差，连续读写比较好) * A properly sized container size allows efficient full stripe RAID writes, which enables an efficient software RAID implementation at the storage layer. |
| Acceleration Methods for fingerprints look up(减少硬盘查询)   * the Summary Vector   + 作用是快速插入、查询fingerprint。DDFS采用布隆过滤器实现，所有fingerprints都要作用一次布隆过滤器 * Stream-Informed Segment Layout(注：这里的segment指的是chunk) * 作用：create spatial locality for both segment data and segment descriptors and to enable Locality Preserved Caching.   + 观察到的现象：segment duplicate locality。即下一篇文章中的chunk locality   + 实现：一个文件相当于一个stream，DDFS分配对应的containers来存这个stream。一个container只能存一个stream的内容，不同streams的内容不能交叉存在同一个container中。 * Locality Preserved Caching(metadata section cache) * 作用：加快辨别重复segment的速度。 * 原理：based on chunk locality. 如果新的流中含有的某chunk也在某个container中，这个container有可能和新的流有更多的重复chunks   + 实现：     - 每一个container都有一个metadata section，记录了其所含的segments对应的fingerprints。     - On a miss, LPC will fetch the entire metadata section in a container, **insert all fingerprints** in the metadata section into the cache, and remove all fingerprints of an old metadata section from the cache together. | |
| 三大技术集成实现   * Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate. * If it is not in the segment cache, check the Summary Vector. If it is not in the Summary Vector, the segment is new. Write the new segment into the current container. * If it is in the Summary Vector, look up the segment index for its container Id(到数据库里去找). If it is in the index, the incoming segment is a duplicate; insert the metadata section of the container into the metadata cache. If the cache is full, remove the metadata cache of the least recently used container first.（布隆过滤器有伪阳性现象，命中表示该fingerprint对应的chunk有可能已经被存储。） * If it is not in the segment index, the segment is new. Write the new segment into the current container. | |

|  |
| --- |
| Sparse indexing Large scale, inline deduplication using sampling and locality |
| 场景：   * Backup stream with locality * fingerprints无法全部存在内存中，造成disk bottleneck   Inline deduplication:   * chunks一到达即进行去重处理，然后根据去重结果是否存在硬盘上 * 缺点:   + 需要将fingerprints存在内存中以迅速比较，而RAM在面对大量数据时无法将所有fingerprints都存在内存中。   + 由于将大部分的index存在硬盘上，每查询一个新来的chunk都有可能触发多次硬盘查询   Outline deduplication:   * chunks在硬盘上积累一定数量后再进行去重处理 * 缺点：   + 额外费硬盘空间：the need for an on-disk holding area large enough to hold an entire backup window’s worth of raw data can substantially diminish storage capacity   + 增加了复杂度：all the functionality that a D2D device provides (data restoration, data replication, compression, etc.) must be implemented and/or tested separately for the raw holding area as well as the deduplicated store   + 往硬盘上积累数据时需要消耗带宽：it is not possible to conserve network or disk bandwidth because every chunk must be written to the holding area on disk.   Chunk locality:   * the tendency for chunks in backup data streams to reoccur together. That is, if the last time we encountered chunk A, it was surrounded by chunks B, C, and D, then the next time we encounter A (even in a different backup) it is likely that we will also encounter B, C, or D nearby   The chunk-lookup disk bottleneck   * 大数据情况下很难将所有的index都存入内存   + a store that contains 10 TB of unique data and uses 4 KB chunks. Then there are  unique chunks. Assuming that every hash entry in the index consumes 40 bytes, we need 100 GB of storage for the full index. * 从硬盘中找index很慢。   + If a seek on average takes 4 ms, this means we can look up only 250 chunks per second for a processing rate of 1 MB/s, which is not acceptable. |
| 方案：   * 将文件流分解成segments，一个segment含有多个chunks。 * 去重步骤： * 对新来的segment，计算其hashes，从中sample出部分hashes作为hooks。 * 内存中存有sparse index。其作用是找出其余含有这些hooks的manifests。 * 从segments中筛选出champions，新的segment针对这些champions去重。选择方法见下： * Each time we choose, we choose the manifest with the highest non-zero score, where a manifest **gets one point for each hook it has in common with S that is not already present in any previously chosen champion**   + - We **don’t** give points for hooks belonging to already chosen manifests because those chunks (and the chunks around them by chunk locality) are most likely already covered by the previous champions. |
| 数据：   * 备份数据，说明不用删除和修改 |
| 缺点：   * 这一方式基于采样，无法保证100%去重率 * 这一方式megabytes scale的chunk locality。如果两个segment之间share一个hook，那么这两者之间可能有更多的相同的chunks。如果chunk locality不好的话，champions与新的segment之间共有的chunks会比较少   + 与Extreme binning不同，这个方案是把文件流分成一个个segments，对segments进行取样，找出最相近的segments进行去重。而Extreme binning则是对文件取representative fingerprints，找出相似文件。 * 硬盘上存储了所有segment的信息，不同segment之间会存有很多重复的hashes。 * 查找champion segments的时候可能需要读取多个segments。虽然作者指出因为可以一次对一个segment中的所有chunks进行去重，从而提升了效率。 |

|  |
| --- |
| ChunkStash Speeding up inline storage deduplication using flash memory |
| 场景：   * fingerprints无法全部存在内存中，造成disk bottleneck * Backup stream with chunk locality   方案：   * 硬盘上存有container，每一个container内约存有1024个chunks   + 内存中存有opened container用以接收新的chunks，当其满时写入disks，同时创建一个新的container(跟LSM-tree的mutable table和immutable table很像) * 每一个chunk都有对应的chunk id，在SSD上存有<chunk id, chunk metadata>对。 * Metadata cache：   + Chunk metadata cache: 内存中存有部分<chunk id, chunk metadata>对。该缓存进出的对象为一个container含有的所有chunks对应的fingerprints(利用了chunk locality)   + Container metadata cache：记录了哪些containers的chunks的fingerprints在内存中，基本单位为<container id, chunk id>。 * Hash table: 在内存中， 采用布谷鸟哈希的形式存储fingerprint对应的<compact key，fingerprint position>（figure 3）。如果命中，需要从flash中读取完整的fingerprint，比较其是否一致。可以看出每个entry只占6个bytes，与SHA1的20bytes相比少了很多。   + 注：可以将SSD上的全部chunk metadata都index到Hash table中，也可以仅仅将部分index到hash table中。后者会导致去重率下降 |
| 缺点   * 该方案一个container中含有的chunks可以来自很多个文件，造成碎片化 * 使用的datasets很小，基本都是一百多GB。只考虑了两次备份 |

|  |
| --- |
| Extreme Binning Scalable, Parallel Deduplication for Chunk-Based File Backup |
| 场景：   * fingerprints无法全部存在内存中，造成disk bottleneck * 需要备份的data的locality几乎不存在。比如incremental backup，邮件的backup等 * 由于Locality不存在，因此缓存和prefetch fingerprints的策略效果就不好   方案：   * 内存中存有文件的一个representative fingerprint，one for each file，硬盘上则存有该文件去重后剩下的所有的chunks的fingerprints. * **The rest of the file’s chunk IDs** are kept on disk in the second tier which is a mini secondary index that we call bin. * 当文件来时，进行chunking和hashing操作，找出representative fingerprint。若该fingerprint在primary index中，则表示该文件以前的版本被存过，从disk中读出对应的bin，进行去重操作，将新的数据写入该bin，将updated bin写回硬盘。若该fingerprint不在primary index中，则认为该文件是新文件，进行去重后将相关数据存入primary index和硬盘。生成该文件的file recipe并存入硬盘。(注：也就是通过representative fingerprint作为其是否是新文件的标准)   + 所有具有相同的representative fingerprint的文件所对应的chunks都会被存在同一个bin中。也就是一个bin中会含有多个文件的chunks。   + 不同的bin之间可能会存在重复文件。     注：内存中只存有representative fingerprint for each bin. 因此存储这些fingerprints的内存空间会小很多 |
| 缺点：   * 以representative fingerprint作为某文件是否为新文件的标准，实际上降低了去重率。同时该系统并未比较不同文件中的重复块。 * 每一个新文件都有可能触发一次bin的读写，如果很多小文件的话，并不利于去重速度。而小文件往往占文件总数的80%以上，体积却只占20%。(见SiLo) * 修改后的大文件的representative fingerprint有可能会与原文件不同。这样一来，即使两者之间仍旧有着很多相同的chunks，也无法实现去重。 * 更新bin时需要进行读出-更新-写回操作。如何保证其不碎片化将影响读写速率(这一条并不一定成立) |

|  |
| --- |
| Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets |
| 思路：   * 用CDC分块的话，重复块前后的非重复块很有可能与原先的块是相似的，因此可以适用delta compression来进一步减少数据 |
| 不足：   * 数据碎片化很严重 |

|  |
| --- |
| Similarity and Locality Based Indexing for High Performance Data Deduplication |
| Similarity利用：   * 20%的文件约占80%的空间：大文件需要分成不同的segment，每一个segment可以利用min-wise来确定与其它segment的相似度 * 80%的文件约占20%的空间：同一目录下的小文件可以聚在一起   数据格式：   * Segment: 含有很多个chunks，用以与其余segments比较，利用similarity   + 对于大文件，直接分成segments   + 对于小文件，把同一个文件夹下的小文件组合成一个个segment * Block：含有多个segments，用以抓住stream流中的locality   + B1中含有S11，S12，S13，B2中含有S21，S22，S23。如果S11与S21相似，那么根据locality，S12与S22，S13与S23很有可能也相似。   原理：   * 某block A的某segment与block B的某segment的RF(representative fingerprint)相同，即使A的其余segments与B的其余segments的RFs并不完全相同，根据chunk locality，剩余的segements也会有很多相同的chunks。 |
| 数据结构：    Workflow：     * 每一个segment都有一个id，每一个block也有一个id * SHTable中的每一项含有一个segment的RF，这个segment所处于的block的block id * 每一个block都有自己的Locality Hash Table(LHTable)，其含有该Block中所有chunk的指纹，基本的存储数据结构为哈希表(为了快速查找) |
| Datasets:   * See page 9. It seems some are available publicly. 而且从中可以看出数据集的特点有很大不同 |
| 发现：   * Read cache不是越大越好。更大的read cache可能会导致读取更多的无效数据 |
| 讨论：   * extreme binning以文件为单位，每去重一个文件都需要触发一次硬盘读取index，并不利于小文件的去重。而SiLo则将小文件组合成了segments，减少了去重小文件时对硬盘的访问量。 * 大文件分成了一个个segment，有利于发现similarity。虽然增加了查询的次数，但是加大了去重率 * 将segments组合成block后可以利用locality。 |
| 缺陷：   * Block的大小决定了去重效果，而这一参数是需要提前确定的 * 利用block进行进一步去重的时候会有可能触发从硬盘中读取block。但是block中仅仅含有LHTable，segmen information等信息，不含有实际的chunk，相对来说比较小 |

|  |
| --- |
| Block Locality Caching for Data Deduplication(unread , a lot text errors) |
| 场景：   * Chunk locality随着时间会改变，旧的chunk locality可能没有用 * 不断更新chunk locality来克服Disk bottleneck * 作者将data stream视为一个整体，切成blocks，每一个block对应一个block recipe。(传统的方法的处理单位是文件，每一个文件对应一个file recipe)   发现：   * Zhu的利用container来提取chunk locality的方式只是suboptimal的。 |

|  |
| --- |
| Building a high-performance deduplication system |
| 设计目标   * Scalability: store and index hundreds of billions of segments. * Deduplication efficiency: best-effort deduplication: if resources are scarce, sacrifice some deduplication for speed and scale. * Throughput: near-raw-disk throughput for data **backup**, **restore**, and **delete**.(不包括**update**)   Reference Management的方案   * Reference counting: 每一个chunk对应一个counter来记录包含该chunk的文件数量。   + 缺点：不可靠 * Reference list: 每一个chunk都保有一个list，该list记录了包含该chunk的所有文件信息   + 缺点：非常占用空间；list是不等长的，更新list涉及到大量的I/O * Mark-and-sweep: 分为mark phase和sweep phase。其实远不需要如文中所说的读取那么多的数据，因为只要读取file recipes就可以了。   解决方案：   * Reference management: reference counting并不可靠，不足以面对生产环境。而实际上，reference需要reliability and good recoverability   + Reference list方案： 较为可靠，对reference list的增删可能会引起周围数据的重写   + Mark and sweep方案：较为可靠，但是从头到尾mark and sweep很费时间。 * 文件的recipe直接记录各个chunks的fingerprints以及其对应的物理位置，形如<fingerprint,location on disk>   + 注意，chunks仍旧存在containers中，因此我估计location on disk实际上是container id * 内存中只存有所有fingerprints中取样出的一部分，作者采用的方案是每一个container中取出一部分fingerprints作为hooks。   + 同时采用了fingerprints cache的方式(见过很多次了)，如果命中某个container中的某个fingerprint，则从硬盘上取出该container的所有fingerprints。这是为了利用chunk locality   + 随着数据的变多，每一个container中取出的fingerprints的数量可以变得更少   实现： |
| 点评：   * 我认为这篇文章不具有什么参考意义1 |

|  |
| --- |
| Efficient Hybrid Inline and Out-of-Line Deduplication for Backup Storage |
| 方法:   * 第一步Inline deduplication: 先用applies coarse-grained inline deduplication at the granularity of large-size units * 第二步outline deduplication: further applies fine-grained out-of-line deduplication on small-size units to improve storage effificiency   方案：   * 将backups分为在live window和archive window内的两种。在live window中的数据是最新的数据，将来最有可能需要restore。在archive window内的数据则将很少会被使用到。 * 见fig1，对整个backups保持一个retention window，在retention window外的backup可以删除掉，retention windown=live window+archive window * 新的backup利用处于live window内的backups经过Inline deduplication(segment级别的)存到硬盘上后，进入live window,。live window, archive window, retention window向后平移。(fig1中X3掉出live window)   + 某backup从live window中掉入archival window后，要执行reverse offline deduplication。即将其较大的segments读出，分割成chunks后去重再写回。   + 掉出retention window的backup例如X0可以删除掉。   + 对segment采用reference进行管理，当一个segment新进入live window后，reference=1；当某含有该segment的backup掉出live window后，该segment的reference--；当且仅当某segment的reference=0时才对其进行chunk level级别的去重 |
| 缺点：   * 第一步的去重率很低，在large segment层次上去重。 * 第二步也无法实现完全去重，因为archive window内的数据没有参与。作者仅将隶属于同一个series的相邻backups做了reverse out-of-line deduplication * 为了从一个older backup中读取某个chunk，可能需要利用indirect reference从旧的backup查找到新的backup直到找到该chunk，这种情况下的查找次数较多 * 我认为不适用于incremental backup。即适用条件有限 |