A FRACTIONAL DIFFERENCING METHOD FOR FILE SYSTEM DATA STORAGE AND TRANSMISSION

VASILE GABURICI

Abstract. We introduce a method for computing the difference between a pattern string (data block or file) and a much larger, iteratively updated set of strings (an entire file system or cache thereof), while only examining a small fraction of this set.

We show how our method generalizes existing algorithms used in a unique block storage system (venti), and a low bandwidth network file system (LBFS), and a differencing incremental backup system (by Burns & Long). We also provide improved algorithms for all these systems, and introduce a new file system (wonkyfs) that can offer better compression than building on top of an improved block systemi. Finally, we combine the compression and bandwidth reduction improvements in the case of a versioning client-server wide area file system (motefs).

1. The Basic Method

1.1. **Description.** Let $\Phi = \mathcal{T}_0 \subset \mathcal{T}_1 \subset \ldots \subset \mathcal{T}_n \subset \ldots$ be a sequence of string sets, such that $\forall i \in \mathbb{Z}^*, \mathcal{T}_{i+1} = \mathcal{T}_i \cup \{P_i\}$, and let $T_i = P_0 \ldots P_{i-1}$ be a string obtained by concatenating all the strings in $\mathcal{T}_i$ in a predetermined order. Our goal is to compute at each stage i an approximation for a shortest[1] edit path (SEP) between P_i and T_i while reading only a small fraction of the strings in $\mathcal{T}_i$. Informally, the sequence $\mathcal{T}_\rangle$ models the data in WORM (Write Once Read Many) file systems. At each stage we extend the file system data T_i by adding a new chunk of data P_i, which, depending on the file system, may be a data block, the data of a single file, or even the data of a file set. E.g., in a multisession CD-R each session corresponds to a stage, and P_i is the data from all the files added in session i. The goal is to write the new chunk of data P_i as a difference with respect to the existing data T_i, but without having to read the entire T_i which may be very large.

In classic approximate string matching problems (k-difference, ϵ-matching) a special class of algorithms addresses the problem of finding all substrings of a text string T that approximately match a pattern string P, whith the practical constraint that T is much longer than P. The basic idea of the exclusion method [Gus97] (called filtration in the more recent [NR02]) for these classic problems is to somehow efficiently exclude large regions of T, so the more expensive approximate matching algorithms only have to be performed on some surviving regions of T.

Building on the exclusion method idea, our fractional differencing method (i) first excludes a large fraction of the strings from $\mathcal{T}_i$ based on an exclusion criteria, then (ii) extends the surviving strings according to an extention algorithm, and (iii) selects the most promising survivors based on a selection criteria. Depending on the

[1]At this point we defer a precise definition for the weights in the edit graph. To define then precisely we need to take into account the exact encoding for the edit path.

exension algorithm used at (iii), it is possible that a good enough approximation for the SEP is readily available after (iii). If this is not the case, we may add another step (iv) in which the strings from $\mathcal{T}_i$ identified at (iii) are used to determine the SEP approximation using a more precise (perhaps even optimal) differencing algorithm.

Some classic algorithms for inexact string matching based on the exclusion method (e.g. Myers ϵ-matching algorithm [Mye94]) build an indexing structure $\mathfrak{S}$ for the large string T. This approach is especially effective if repeated matches of various pattern strings P_i are done on the same database string T. E.g., Myers algorithm runs in sublinear time wrt. to the length of T. As formulated herein, our file system problem differs from this traditional inexact string match against a database in one simple but important aspect: after every matching query, the pattern string is added to the database. This observation leads to the introduction of a sequence of indexing structures $\mathfrak{S}_i$, that facilitates the crucial step (i) in our method. Because $\mathfrak{S}_i$ grows as T_i grows (although it may grow slower), we would like to construct $\mathfrak{S}_{i+1}$ from $\mathfrak{S}_i$ and P_i while examining only a fraction of $\mathfrak{S}_i$. This is the last step (v) in our method. Fortunately, for most index structures (e.g. hashtables, suffix trees), this insertion is a simple problem, and has running times that are sublinear or may not even depend on the size of $\mathfrak{S}_i$, thus satisfying our requirement.

1.2. **Applications.**

1.2.1. *Venti and venti*$^+$. Venti [QD02] is a system that allows file data to be stored as a tree of variable size unique blocks. Blocks are identfied, and their uniqueness is enforced by computing a hash function on block data. Leaf blocks contain actual file data, while the blocks at other levels describe sequences of blocks by enumerating block identifiers. The venti system itself does not deal with splitting files into blocks, but the suggested application usage is to determine anchors that are resistent to insertions and deletions using the technique initially employed in the tool *sif* [Man94].

Venti may be seen as an application of our method in the following way: the block hashtable is the index structure $\mathfrak{S}_i$, and P_i is a block to be written. At step (i) if the hash of P_i is found in $\mathfrak{S}_i$ then the block allready stored in for that hash value is the survivor, otherwise there are no survivors. We then skip to step (iv) where in the case of a survivor, we do nothing and assume based on the low proability of hash collision that our differencing has found an exact match. Step (v) obviously consists of adding the hash of the new block to the hashtable.

There are two ways to extend venti, one which preserves the existing architecture and the current choice for P_i, which we shall present first, and another less direct approach that assumes integration with a filesystem sitting on top of venti. Because the second approach is not really a block storage system anymore we shall call it wonkyfs[2]. The purpose of this discussion is to illustrate simple applications of the method, and also to emphasize the importance of an appropriate choice for P_i.

The direct approach for building a venti$^+$ is to go beyond the dichotomous differencing by identifying more survivors than the exact identifying hash match. In simple terms, we need to identify similar blocks. A simple index structure that allows that is a hashtable that, in the true spirit of [Man94], indexes block subregions. Because this second hashtable is only used for finding similar blocks, its

[2] WORM chunky file system

hash function may output shorter fingerprints, and be less collision resistant. This is proabably a good indea also because the number of block subregions is higher than the number of blocks, so storage space for this hashtable may be of concern. Having less collision resistance on this secondary hash function does not threaten data integrity, it merely increses the probability that a surviving region may not actually be similar to P_i. Thus, the new indexing structure $\mathfrak{S}'_i$ is comprised of two hashtables, the original one $\mathfrak{S}_i$, which allows identity to be estabilished with high probabillity, and the new subregion indexing hashtable $\mathfrak{R}_i$ which maps block subregion hashes to the block byte ranges they belong to.

We need to allow files to make use of partial venti$^+$ blocks. To do this, we extend the intermediate "inode" blocks to allow them to contain block ranges. While more complex solutions are possible, for the purpose of simplicty we limit the use of block byte ranges to block identifiers that point to data blocks. So an intermediate block contains a list of blocks, identifiers, and, for those blocks that contain actual data, they also contain a block byte range. In order to benefit from these partial blocks, the read interface for venti$^+$ would have to be changed to allow block byte ranges to be requested. Given that the maximum venti block size is 52Kb, and that that the system uses hard drives which normally have 512 byte blocks, reading only partial venti$^+$ blocks is worthwhile.

Writing a new block P_i proceeds as follows: (i) process P_i calculating hashes for use in $\mathfrak{R}_i$, and the identifying hash. If the identifying block hash is found in $\mathfrak{S}_i$, there is only one survivor, and we proceed, as in the regular venti, to step (iv-a). If the identifying hash in not in $\mathfrak{S}_i$, lookup in $\mathfrak{R}_i$ all block regions that match, and asseble the list of survivors as the blocks which those regions map to in $\mathfrak{R}_i$. At step (ii) we use a simple linear extension algorithm to extend these matching subregions. At (iii) we select for each subregion of P_i the longest corresponding subregion in the surviving set if such a region exists and proceed to step (iv-b). Step (iv-a) is identical to the original step (iv) in venti. At step (iv-b) we create new venti$^+$ blocks for the regions of P_i that do not have a matched survivor. We write an intermediate block (or tree of blocks if only one block does not have enogh space) that contains identifying hashes and byte ranges for the surviving region blocks, and the identfiers for the new blocks. Finally, at step (v) we add the appropriate entries to $\mathfrak{S}'_i$.

1.2.2. *WonkyFS.* Venti$^+$ suffers from two flaws when a file system is the layer above it: (1) it does not use the fact that blocks belong to files, and (2) the hashtable $\mathfrak{S}_i$ is redundant for similarity finding purposes. The strict layering imposed by venti, transforms a file composed of a linear succession of n blocks into a set of n blocks. If one of these blocks is found to be identical to a block already stored, chances are that other blocks in the file, perhaps adjacent ones have some partially matching peers already stored. But, we have to discover this information the hard way, using the hashtable $\mathfrak{R}_i$ for each block, and some blocks may actually fail to find any similar peers, despide having a non-trivial amount of similar data. Also, in venti$^+$ we cannot cross block boundaries when the matching regions are extended. This particular issue may be alleviated by choosing anchors in a way that makes anchors for $\mathfrak{S}_i$ fall on subset of the anchors for $\mathfrak{R}_i$ [3].

[3]This can be done by using the same Rabin polynomial for both, and choosing the anchor defining values for $\mathfrak{S}_i$ be a subset of the defining values for $\mathfrak{R}_i$.

We correct this flaws in proposing a new file system wonkyfs, that uses the file-block relationship for similarity prediction instead the block-sublock relationship. Furhtermore, we eliminate the redundant hashtable $\mathfrak{S}_i$ which also happens to make data retreival determistic. The key difference from venti that allows these improvements is to include the file splitting layer in the system.

A file inode in wonkyfs is identified in the traditional manner by an identifier that does not depend on the file contents. For simplicity, the file inode stores a list of block byte ranges. For this encoding to be effective, there is minmum threshold for a byte range to be worth specifying, but we will ignore this detail in here. Blocks are also identified in the traditional way (perhaps based on their position disk). The regions used for finding similarity are no longer included in blocks, but may span blocks. We shall call them s-regions. The hashtable $\mathfrak{R}_i$ maps an s-region to a set of inode byte ranges that exactly match that region.

Writing a new file P_i in wonkyfs follows the our method: (i) the file is split in s-regions that are matched against $\mathfrak{R}_i$. For the surviving regions, the files (and byte ranges) that correspond to them in the file system are retrieved. (ii) These surviving regions are extended linearly. (iii) For each region in P_i we select the larges corresponding surviving region. (iv) The resulting matched file byte ranges are mapped to block byte ranges using the file inodes, and for any unmatched region in P_i we create a new block (or multiple blocks if we want to limit block size). The new file inode is written with the appropriate blocks and byte ranges. (v) $\mathfrak{R}_i$ is extended with all unmatched regions from P_i.

1.2.3. *Burns & Long backup system.* The backup system of Burns & Long [BL97], build by modifing IBM ADSM, uses a single file to diff against. Any file name changes would throw off the algorithm. We can trivially use the wonkyfs algorithm to find more that one file to diff against based on file contents.

2. An Extension for Caches

2.1. **Description.** We extend the method from section 1 in two ways: first, we add support for removal of strings from $\mathcal{T}_i$, and second, we adapt our method to the case where only a subset of $\mathcal{T}_i \supseteq \mathcal{C}_i$ is readable directly. These extensions allow us to apply our method to caches. Note that supporting deletions from $\mathcal{T}_i$ does equate support for a "fully fledged" file system, because (partial) overwrites are still not supported.

Formally, adding support for deletions makes the sequence $\Phi = \mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n, \ldots$ be defined by choosing for each i one of the two rules $\mathcal{T}_{i+1} = \mathcal{T}_i \cup \{P_i\}$ or $\mathcal{T}_{i+1} = \mathcal{T}_i - \{D_i\}$ s.t. $D_i \in \mathcal{T}_i$. The only addition that our method requires is that deletion from $\mathfrak{S}_i$ be performed efficiently w.r.t. the size of $\mathcal{T}_i$. This requirement is trivially met by the classic index structures.

The extension for a cache is a bit more involved. We need to define writing, reading and deletion. At each step i we chose to perform one of these three operations. For writing, we choose to add a new element P_i to $\mathcal{T}_i$, so we also add P_i to $\mathcal{C}_i$, but at the same time may to choose to delete an element from $\mathcal{C}_i$ (i.e. either simply add to the cache ore perform a cache replacement). For reading, an element $P_i \in \mathcal{T}_i$ is to be added to $\mathcal{C}_i$, again possibly replacing an element in $\mathcal{C}_i$. For deletion, we choose to delete an element from $\mathcal{T}_i$, and we also delete it from $\mathcal{C}_i$ if contained in it. The two sequences $\mathcal{C}_i, \mathcal{T}_i$ model a client-server system where the cache $\mathcal{C}_i$ is on the client and the full file system $\mathcal{T}_i$ is on the server. We define two problems

in this context: the write problem is to send a new pattern P_i to the server using minimal bandwidth, and the read problem is to send a pattern from the server to the client, which may be new w.r.t. $\mathcal{C}_i$. Except for the inclusion property that hold between $\mathcal{C}_i$ and $\mathcal{T}_i$, the problem is symmetrical. Thus, it makes sense to consider two index structures $\mathfrak{S}_i^{\mathcal{C}}$ and $\mathfrak{S}_i^{\mathcal{T}}$.

If the index structures are hashtables using a sensible hash function which map s-regions to file byte ranges, then we can send the hash values as high probability proxies of the s-regions. LBFS [MCM01] uses exactly this solution for reducing network traffic. In the case that the opposite side hashtable does not have the exact s-region (called chunk in LBFS), a negative acknowledgement (nak) is sent back, and the actual data of the s-region is sent.

2.2. **Applications.**

2.2.1. *LBFS.* A first way to extend LBFS is use the approach we took for venti$^+$, i.e. keep two additonal index structures at sub-chunk level $\mathfrak{R}_i^{\mathcal{C}}$ and $\mathfrak{R}_i^{\mathcal{T}}$. For writes, when nak come back for a chunk hash, simply find similar chunks using $\mathfrak{R}_i^{\mathcal{C}}$ and send a difference against those. In the case of a read, the problem is complicated by the fact the server may choose to difference against chunks not present in $\mathcal{C}_i$. There are several solutions to this problem, the most practical is for the server to maintain an approximation of the client cache set, e.g. using a Bloom filter. Any false positives in the Bloom filter will require that the regions in P_i corresponding to false positive hits in the Bloom filter be actually sent to the client.

The exact approach used for wonkyfs will not work for improving LBFS because $\mathfrak{S}_i^{\mathcal{C}}$ and $\mathfrak{S}_i^{\mathcal{T}}$ cannot be dropped without losing the ability to compute (coarse) differences agains the remote party. We can however mitigate this by dropping $\mathfrak{R}_i^{\mathcal{C}}$ and $\mathfrak{R}_i^{\mathcal{T}}$, and using somewhat smaller chunk size. Furthermore, we modify $\mathfrak{S}_i^{\mathcal{C}}$ and $\mathfrak{S}_i^{\mathcal{T}}$ to map a chunk not to only one file byte range, but to all the ranges that match it. After a nak, the offending chunk may thus be mapped to all the files that contain it using the local $\mathfrak{S}_i$. Using the wonkyfs approach, we find similar files locally and for naked hashes, we send file differences restricted to the byte range of the chunk in P_i. Unlike the wonkyfs case, it is not clear wheter is approach is superior to the venti$^+$ approach. It does however have the advantage that the Bloom filter on the server needs to map files, not chunks, so it may be smaller in size.

2.2.2. *Motefs.* The original idea in motefs is to perform only local diffing, thus in a way it is the opposite of LBFS. Because $\mathfrak{S}_i^{\mathcal{C}}$ and $\mathfrak{S}_i^{\mathcal{T}}$ are no longer required, we can use the wonkyfs solution and keep only $\mathfrak{R}_i^{\mathcal{C}}$ and $\mathfrak{R}_i^{\mathcal{T}}$. This allows for a unfied differencing solution both for storage and network transmission; the observation that Bloom filter is need on the server still applies. Motefs proposed the use file metadata rules to determine candidate files for differencing. Our method uses actual file data, but the two approaches may be combined by including in the surviving set files based on our method and metadata rules.

2.2.3. *Backup system.* The backup system of Burns & Long [BL97] can be improved probabilistically by using a hashtable $\mathfrak{S}_i^{\mathcal{T}}$ to keep hashes for (some) blocks no longer available. Thus, precise differencing can be done against the cached subset, and less precise against the larger hashed subset.

3. Related and Future Work

We only covered very simple algoritms and data structures in here. Specifically, we need to evaluate other: algorithm and data structure (when a hashtable is not mandatory like in LBFS) for filtering (i), algorithm for extending survivors (ii). For (ii), Myers ϵ-matching algorithm [Mye94], and more recent algorithms covered in [NR02] are interesting candidates. The problem with Myers algorithm is that it performs poorly for large alphabet sizes; 256 is large in this case.

References

[BL97] Randal C. Burns and Darrell D. E. Long. Efficient distributed backup with delta compression. In *I/O in Parallel and Distributed Systems*, pages 27–36, 1997.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[Man94] Udi Manber. Finding similar files in a large file system. In USENIX, editor, *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 1–10, Berkeley, CA, USA, Winter 1994. USENIX. Also as TR 93-33.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001.

[Mye94] Eugene W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[QD02] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pages 89–102, Berkeley, CA, January 28–30 2002. USENIX Association.