Improved Deduplication through Parallel Binning

Zhike Zhang, Univ. of California, Santa Cruz, CA, zhike@cs.ucsc.edu
Deepavali Bhagwat, Hewlett Packard Company, Palo Alto, CA, deepavali.bhagwat@hp.com
Witold Litwin, Université Paris Dauphine, Paris, France, witold.litwin@dauphine.fr
Darrell Long, Univ. of California, Santa Cruz, CA, darrell@cs.ucsc.edu
Thomas Schwarz, S.J., Univ. Católica del Uruguay, Montevideo, Uruguay, tschwarz@ucu.edu.uy

***Abstract*—Many modern storage systems use deduplication in order to compress data by avoiding storing the same data twice. Deduplication needs to use data stored in the past, but accessing information about all data stored can cause a severe bottleneck. Similarity based deduplication only accesses information on past data that is likely to be similar and thus more likely to yield good deduplication. We present an adaptive deduplication strategy that extends *Extreme Binning* and investigate theoretically and experimentally the effects of the additional bin accesses.**

I. Introduction

Deduplication is a popular strategy to compress data in a storage system by identifying and eliminating duplicate data. Its use for backup workloads has shown impressive compression ratios (20:1 as reported by Zhu *et al.* [1], and up to 30:1 as reported by Mandagere *et al.* [2]), and can scale to petabytes [3].

Deduplication works by dividing incoming files or streams into chunks, characterizing the chunks by their signature (hash), storing the chunk signatures in an index, and using the index to find duplicate chunk signatures [4], [5], [6]. Systems that use chunks of a fixed size lose deduplication opportunities when even a single byte is added or deleted in a large file. Content defined chunking [5] sets chunk boundaries based on a condition evaluated in a small window.

Inline deduplication produces a very large number of chunks, so if we were to use chunks of 4 kB and store only 25 bytes for each chunk, then the size of the index would be 0.625% of the total amount of storage dedicated to storing chunks. Even with a compression rate of 1 to 30, a petabyte storage system would require an index of size 208 GB. The need to access an index of this size creates the I/O bottleneck for file deduplication.

Extreme Binning (EB) [3], [7] addresses this issue. EB (Figure 1) divides the index in bins. When EB processes a file, it stores all its chunk signatures in a bin indexed by the minimum chunk signature in the file. A bin accumulates chunk signatures from different files. All these files have at least one and probably many chunks in common. If a new file is processed, EB accesses a single bin (the one with the minimum chunk signature of the file) and updates this single bin. We generalize EB by updating the w bins indexed by the w minimum chunk signatures in the file and looking for similar files in r bins. We call such a scheme wWrR. A scheme with $w > r$ does not make sense as we need to read a bin in order to update it, even if we disregard its contents. While

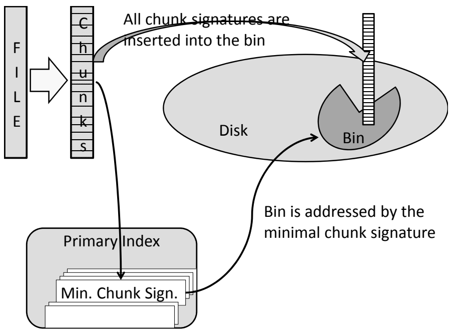
F I L E
Chunks
All chunk signatures are inserted into the bin
Disk
Bin
Bin is addressed by the minimal chunk signature
Primary Index
Min. Chunk Sign.

Fig. 1. Bin creation in Extreme Binning

EB reads one bin and updates this bin (with a total of two IO operations), our scheme uses $r+w$ IO operations. We evaluate EB and our schemes analytically and experimentally how they fall between EB and perfect deduplication where we compare a file signature against the complete index.

II. Related Work

Deduplication needs to solve the problem of finding chunks of data that are already stored in a large repository. This problem is related to finding similar documents in a large repository.

A. File Resemblance

Deduplication makes use of techniques developed for search engines that need to avoid displaying duplicates or near-duplicates of higher ranked search results. In this context, Broder [8], [9] proposed extracting features (substrings) from the document and then to measure resemblance of two documents based on the relative number of features in common. This measure is known as the Jaccard index, which is used in biology to measure species overlap between two sites. Broder then showed that using the s smallest feature hashes gives an unbiased estimate of the resemblance of two documents.

Bhagwat, Eshghi, and Mehra [10] use Broder's feature extraction technique for document retrieval based on similarity. They divide the index into a set of partitions. Each partition is indexed by a feature hash. When a document enters the collection, the hashes of its features are written to the k partitions indexed by the minimum feature hashes in the document. When documents similar to a given document are searched, the search is directed to the k partitions with the minimum feature hashes of the given document. We are analyzing here this technique for deduplication workloads.

B. Overcomming the IO Bottleneck

Chunk-based, inline deduplication divides an incoming file into a stream of chunks and identifies those chunks that are already stored. Unfortunately, the number of chunks in a large system is very large. Current chunk sizes are about 4 kB, so that a terabyte (TB) system contains 2^{28} and a PB system 2^{38} chunks. Even if a petabyte (PB) backup system achieves a deduplication ratio of more than 100, this still amounts to $\sim 2^{31}$ unique chunks. Venti [6] and Jumbo Store [11] find duplicate chunks with a *full chunk index*, a dictionary data structure, where the key is the chunk signature and the value holds the metadata about the chunk such as its location on disk. An incoming file is partitioned into chunks and the index consulted for each chunk in the file. Unfortunately, it is usually impossible to store the complete index in RAM and all resulting disk operations throttle the speed at which the repository can ingest new files.

Zhu, Li and Patterson [1] address this bottleneck by using an in-memory Bloom filter in order to avoid looking up the index if a chunk is not in the system. If we use about one byte per chunk to verify whether a chunk is already in the repository, we would need a Bloom filter of size 2GB for our PB system with 100:1 deduplication rate. In this type of architecture, the Bloom filter resides in memory and secondary storage holds information on chunks that already reside in the system. Zhu, Li, and Patterson structure the index carefully for spatial locality to optimize disk accesses.

Min, Yoon, and Won [12] add to the Bloom filter an index partitioning data structure that uses LRU to exploit temporal locality.

Lillibridge and colleagues [13] propose *Sparse Indexing* that breaks an incoming stream into large (thousands of chunks) segments, statistically samples incoming segments, and determines whether an incoming segment is similar to one already digested. In this case, it loads the (much smaller) index of the already digested segment, which fits into RAM.

C. Extreme Binning

Extreme Binning (EB) [3], [7] uses the concept of file similarity to only consult a small part of the whole index (a single *bin*). Even though it loses some deduplication opportunities, EB has shown deduplication rates for sample workloads that are quite close to optimal. EB splits the chunk index into two levels. The primary (high level) index resides in RAM and contains one chunk ID entry per file, namely the minimum chunk signature. The rest of the chunk IDs are stored in bins, small subsets of the chunk index, which jointly form the second level of the index structure. The bins reside on disk and are accessed using the primary index. The primary index also contains the hash of a complete file to quickly locate duplicate files.

When processing an incoming file, EB determines the minimal chunk signature *s*. It then accesses the primary index in RAM, using *s* as a key. If there is no such entry, EB gives up trying to deduplicate this file. If there is an entry in the primary index, then EB first compares the hash of the whole file with any file hashes it finds in this primary bin entry. This allows EB to find complete duplicate files. Otherwise, EB loads the bin associated with the record in the primary index into memory. It then looks up all chunk signatures from the incoming file in the bin. If it finds the chunk signature in the bin, it has found a deduplication opportunity. The bin entry contains all information necessary for deduplication, in particular the location of the chunk. After this lookup procedure terminates and EB knows which chunks are duplicates and where the other chunks are going to be stored, it generates chunk IDs for the latter chunks and adds this information to the bin. As we have seen, EB never uses more than two disk accesses when processing an incoming file.

The work by Aronovish *et al.* uses a very similar approach to EB [14]. Similarity based detection in their system also uses chunk signatures. They briefly consider and then reject the possibility of using the four maximum chunk signatures as the similarity signature of a segment (which they call chunk). Instead, they use the signature of the four chunks located at a certain offset to the four chunks with maximum signature. They see a trade-off between a lesser likelihood of similarity detection and the elimination of false positives. As they design for an exclusive backup load and not for file-based deduplication, their segments will be on average much larger than for our files and the chunks making up the segment signature almost always exist, but frequently not in our case. Their argument is that the signatures of the chunks at an offset are completely random, whereas the highest bits of the four maxima are likely to be ones. This concern arises because they only compare an incoming segment to one segment in memory in order to process as fast as possible, whereas EB retrieves information on all files with the same file signature to optimize deduplication and recovery. Our extensions considered here follow the same policies. Given these differences that even go beyond just a different target application, an experimental comparison of the work by Aronovich and colleagues and ours is difficult.

Románski *et al.* propose an alternative to content defined chunking [15] by introducing two different levels of chunks, one at 64 KB and the other at 8 KB. On processing an incoming stream, the large chunk index is searched first. It also uses a small "anchor" chunk to indicate that corresponding parts of an on-disk index of small chunks should be fetched. The anchor chunks serve similarly to the maximum signature chunk in extreme binning. An evaluation on our data set showed that anchor- driven subchunk deduplication has worse storage deduplication, but saves some (but not a lot of) RAM. Since it uses sequential prefetching of the small chunk index, it performs better than EB with IO-latency.

III. Architecture

We propose an extension of EB that uses more than a single bin in order to determine whether parts of an incoming file are already stored. As in EB, each bin is a lookup table that associates a set of chunk values with chunk metadata, most importantly the location of the chunk in the storage system. Each bin is labeled by a chunk signature, but unlike EB, this is not necessarily the smallest chunk signature in the bin.

When a file arrives, we determine the minimum r chunk signatures and read the corresponding r bins. After processing, we add all chunk signatures in the file into the w bins labeled by the minimum w signatures. The parameters r and w are small values and not necessarily identical. Since most implementations would need to read a bin before updating it, in practice $w \leq r$.

EB also uses the complete file hash to find quickly exact copies of files already stored. It stores the complete file hash in the bin with the minimum chunk signature. We do the same. The effect on deduplication is the same for EB and our extension.

IV. File Retrieval Probabilities after Alterations

We first assess the effects of extending EB by additional IO operations in a manner independent of the workload. To obtain a measure of efficiency, we use a situation where a new version of a file already stored enters the storage system, and calculate the probability that we deduplicate the altered file against the original. The effect of small changes to a file on its set of chunk signatures are difficult to measure. For instance, changing a single byte can destroy an existing chunk boundary or create a new one and thus alter the number of chunks. We use a simplified model where we represent files as sets of chunk signatures and model changes to the file by removing, inserting, and adding chunk signatures to the set. We concentrate on three scenarios, *insertions*, where we add chunk signatures, *changes*, where chunk signatures change, and *deletions*, where we remove chunk signatures.

Assuming a good hash, we can model a set of N chunk signatures as a set of N uniformly distributed random numbers. We assume a uniform strategy of adding the set of chunk signatures from a file to w bins and of retrieving r bins for deduplication. The original file is represented by a set A of N chunk signatures and the altered file by a slightly different set B. When deduplication processes the altered file, it will find the original file if among the smallest r elements of B is at least one of the smallest w elements of A.

We use a slot model for our calculations. Each slot represents an actual or previous chunk, ordered by chunk signature. For an example, we take a file with five chunks, ordered by ascending signature. Assume that we now add three more chunks to the file. We write n for the three new chunks and r (retained) for the old chunks. A possible configuration is $[n,r,n,r,r,r,r,n]$, indicating that the minimum chunk signature (the left n-marker) is a new chunk and that the previous minimum signature is the original chunk with the minimum signature. If we use a 2W3R scheme, then we deduplicate against the original file if one of the current three minimum signatures is one of the two minimal original chunks. In our example, EB would not have found the original file, but the extension would. As another example, assume that we have again a file with five chunks, but a change to the file deletes one chunk and changes two. In this example, we have five slots representing chunks from the original file and two slots for the "new" chunks created by the change. We use an n marker for these two. Using d for deleted and c for an original, but now changed chunk signature, we can describe a possible situation by $[r,c,n,r,n,d,c]$, in which both EB and 3R2W would deduplicate against the original, because the minimum chunk signature has not changed (this chunk has been "retained").

TABLE I
List of abbreviations and indices used

A	Appends: changing file through chunk insertion
C	Changes: changing the files by changing chunks
D	Deletions: changing the files by deleting chunks
r	Number of bins read
w	Number of bins written
N	Number of chunks in a file
p	Probability of successful retrieval
$\wp$	Limit of probability $N \to \infty$

Retrieval probabilities depend on the model (insertion, change, deletion) and the exact type of extension, as well as the total number N of chunk signatures in the original file and the number x of alterations. We call the probability of retrieving the file p and indicate the extension in form wWrR and the model as a subscript and the chunk numbers as parameters. Thus $p_{(w\mathrm{W}r\mathrm{R},\mathrm{A})}(N,x)$ is the probability of deduplicating against an original of N chunks when x chunks are added using an extension, which writes w bins and reads r. We observed that for $N \to \infty$, this probability quickly depends only on the proportion $\rho = x/N$ of chunks modified, appended or deleted. We calculate this limit as

$$\wp_{(w\mathrm{W}r\mathrm{R},\mathrm{X})}(\rho) = \lim_{N\to\infty} p_{(w\mathrm{W}r\mathrm{R},\mathrm{X})}(N,\rho N).$$

Here X stands for either A, C, or D.

The example in Figure 2 shows an ordered set of chunk signatures of an original file and then the result of changing two of them. The minimal signature 0012763 has been replaced by 3992165 and signature 9481121 has been replaced by 6098712. In the slot model of this alteration, all signatures appear, thus, there are 8 slots. The lowest slot is taken by the minimal signature of the original file, but this signature has been changed. The slot assignment starts with a c-marker. The second slot corresponds to a chunk present in both the original and the altered file and gets an r-marker. The complete slot marking results in $[c,r,r,r,n,n,c,r]$. Incidentally, in this example, EB does not deduplicate against the original file, but perhaps against some other related file.

A. Retrieval using Extreme Binning

We explain our probability calculations in detail for EB. We first treat a changed file and assume that x among N chunk signatures have changed. EB only writes to a single bin and looks up a single bin. There are a total of $N+x$ signatures that we use in our slot model, x changed (c), x new (n), and $N-x$

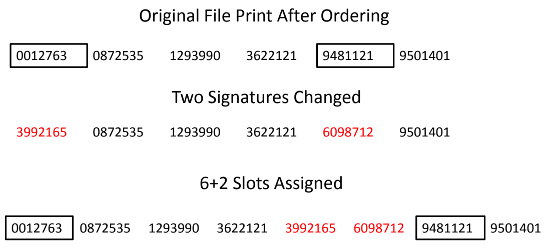

Fig. 2. Example of filling slots.

retained (r) signatures. If the first slot contains a c-signature, then the original file signatures have been stored in certain bin, but it is highly unlikely in practice and impossible in our model that the minimum chunk signature of the altered file is in that bin. Therefore, when we process the altered file, we are going to load a different bin and will not deduplicate against the original file. Similarly, if the first slot contains a n-signature, then the lookup caused by processing the altered file goes to a different bin. Thus, only if the first slot contains an r-marker, do we look up the bin where the chunk signatures of the original file are stored and deduplicate against the original. We describe this situation as $[r,*]$. We determine the probability by counting the distributions of the $N-x$ r-markers in the $N+x$ slots. The number of ways to distribute the remaining $N-x$ r-markers among the remaining $N+x-1$ slots is $\binom{N+x-1}{N-x-1}$ out of $\binom{N+x}{N-x}$ ways to distribute the r-s in the slots, which gives us

$$p_{1\mathrm{W}1\mathrm{R,C}}(N,x) = \frac{N-x}{N+x}$$

If we set $x = \rho N$, we obtain (without having to calculate a limit)

$$\wp_{1\mathrm{W}1\mathrm{R,C}}(\rho) = \frac{1-\rho}{1+\rho}$$

We model a growing file by adding x signatures to a set of N original signatures. Deduplication of the altered file accesses the bin with the chunk signatures of the original file, if the first slot is a r-slot. The expression for the slot assignment is again $[r,*]$, but while the total number of slots remains $N+x$, the number of r-markers is now N as opposed to $N-x$ previously. This gives us

$$\begin{aligned} p_{1\mathrm{W}1\mathrm{R,A}}(N,x) &= \binom{N+x-1}{x-1}\Big/\binom{N}{x} \\ &= \frac{N+x}{N} \end{aligned}$$

If we add a proportion ρ of the signatures, we get

$$\wp_{1\mathrm{W}1\mathrm{R,A}}(\rho) = \frac{1}{1+\rho}$$

as the probability of using the original file for deduplication of the altered one.

If we delete x signatures, then retrieval is successful if and only if the first slot is not a c-slot. We have x changed markers and $N-x$ retained markers. The probability is given by counting the number of ways to distribute the c-markers among the N slots as

$$\begin{aligned} p_{1\mathrm{W}1\mathrm{R,D}}(N,x) &= \binom{N-1}{x}\Big/\binom{N}{x} \\ &= 1-\frac{x}{N} \end{aligned}$$

and $\wp_{1\mathrm{W}1\mathrm{R,D}}(\rho) = 1-\rho$.

B. Retrieval using Minimum and Maximum

We now consider an interesting extension of EB that uses both the minimum and the maximum chunk signature in order to write to and read from two bins. In this version, all chunk signatures of a file entering the system are compared to all signatures in two bins and these two bins are later actualized. The difference to our 2W2R scheme is in the choice of the second bin.

Assume first that we change x out of N signatures in the set constituted by the chunk signatures of a file. Retrieval is successful with the minimum, if the leftmost slot has an r-marker and retrieval is successful with the maximum if the rightmost slot has an r marker. The corresponding slot patterns are $[r,*,r]$, $[r,*,c]$, $[r,*,n]$, $[c,*,r]$, and $[n,*,r]$. The total number of arrangements for the $N-x$ r-markers among the $N+x$ slots is $\binom{N+x}{N-x}$. An arrangement where we do not access one of the two bins is one where all r-markers are in the middle $N+x-2$ slots, and there are $\binom{N+x-2}{N-x}$ ways for this arrangement. Simplifying $1-\binom{N+x-2}{N-x}\big/\binom{N+x}{N-x}$ gives for the probability of retrieval using Minimum and Maximum (MM)

$$p_{\mathrm{MM,\,C}}(N,x) = 1-\frac{2(2x-1)x}{(N+x-1)(N+x)}$$

For the limit, we obtain

$$\wp_{\mathrm{MM,C}}(\rho) = \frac{(1-\rho)(1+3\rho)}{(1+\rho)^2}$$

When we add x to N signatures, then the retrieval fails if all r markers are in the middle $N+x-2$ slots. This happens with probability $\binom{N+x-2}{N}\big/\binom{N+x}{N}$, giving us after simplifying

$$p_{\mathrm{MM,A}}(N,x) = 1-\frac{(x-1)x}{(N+x-1)(N+x)}.$$

For the limit, we obtain

$$\wp_{\mathrm{MM,A}}(N,x) = \frac{1+2\rho}{(1+\rho)^2}.$$

A similar argument is valid for deletions, but now we have N slots of which x are filled with the d-marker. We do not retrieve the original if all r markers are in the middle $N-2$ slots, which happens with probability $\binom{N-2}{N-x}\binom{N}{N-x}^{-1}$ or

$$p_{\mathrm{MM,D}}(N,x) = 1-\frac{(x-1)x}{N(N-1)}$$

and

$$\wp_{\mathrm{MM,D}}(N,x) = 1-\rho^2.$$

As Figure 3 shows, the chances for retrieval are markedly improved over the EB for almost the whole range of ρ.

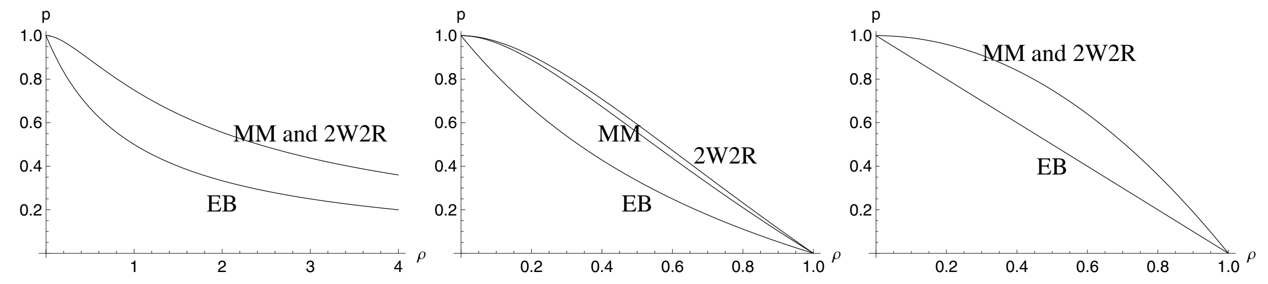

Fig. 3. Probability p of successful retrieval to a file after adding (left) and changing (middle) and deleting (right) a proportion ρ of its chunk signatures.

C. Retrieval with 2 Writes and 2 Lookups

We now consider the 2W2R extension of EB, where we use the two minimal chunk signatures of a file to select the two bins in which we store the file signature and with which we compare the file signature. We first consider changes. Enumerating the slot assignments corresponding to successful retrieval is not simple, for this reason, we verified all our formulae using simulation. A r-marker needs to be in the leftmost or second leftmost position not counting c-markers so that the lookup uses a retained signature. Similarly, a r-marker needs to be in the leftmost or second leftmost position not counting n markers so that we can access one of the two bins with chunks from the original file. This gives us exactly the following possibilities: $[r,*]$, $[c,r,*]$, $[n,r,*]$, and $[c,n,r,*]$. A pattern such as $[n,n,r,*]$ leads to an unsuccessful search, as we use two new signatures for lookup. With a pattern $[c,c,r,*]$, the signatures that lead to the two bins where information on the original is stored are no longer part of the signature set. We use multinomials $\binom{N}{a,b,c} = \frac{N!}{a!b!c!}$ to count the number of arranging three set of a, b, and c markers into $N = a+b+c$ slots.This gives the following retrieval probability and its limit

$$p_{\mathrm{2W2R,C}}(N,x) = \frac{\binom{N+x-1}{N-x-1,x,x}+2\binom{N+x-2}{N-x-1,x-1,x}+\binom{N+x-3}{N-x-1,x-1,x-1}}{\binom{N+x}{N-x,x,x}}$$

$$\wp_{\mathrm{2W2R,C}}\rho) \quad = \quad \frac{1+3\rho-4\rho^3}{(1+\rho)^3}$$

Now we assume that the signature set has x inserts. In this case, the second write has no benefits since the alteration never destroys access to the first bin (with minimum signature as key) while maintaining access to the second bin (with the second minimal signature as key). The only pattern for retrieval failure is given by the pattern $[n,n,*]$ where we do the lookup with new chunk signatures. Thus

$$\begin{aligned} p_{\mathrm{2W2R,A}}(N,x) &= 1-\frac{\binom{N+x-2}{x-2}}{\binom{N+x}{x}} \\ &= 1-\frac{x(x-1)}{(N+x)(N+x-1)} \\ \wp_{\mathrm{2W2R,A}}(\rho) &= \frac{1+2\rho}{(1+\rho)^2} \end{aligned}$$

We now consider a file with N chunk signatures of which x are removed. The retrieval does not function for the pattern $[d,d,*]$. We can obtain the probability of using the original file for deduplication as an opposite probability:

$$\begin{aligned} p_{\mathrm{2W2R,D}}(N,x) &= 1-\frac{\binom{N-2}{x-2}}{\binom{N}{x}} \\ &= 1-\frac{x(x-1)}{(N)(N-1)} \\ \wp_{\mathrm{2W2R,D}}(\rho) &= 1-\rho^2 \end{aligned}$$

Minimum-Maximum functions exactly the same as the 2W2R-scheme for deletions and insertions, whereas a comparison of the patterns shows that for changes, 2W2R is slightly better. Figure 3 compares the three schemes. Retrieval probabilities are up to 28.2% better with 2W2R-scheme than with EB in the case of changes. For insertions and deletions, the improvement attains a maximum of 25% for $\rho = 1$ and $\rho = 0.5$, respectively. We recall that these improvements cost two more IO operations per processed file.

D. Retrievals with One Write

We now explore the effects of additional reads on retrieval probabilities. We first explore this for schemes that insert the chunk IDs of an incoming file into a single bin, *i.e.* schemes with a single write, but where we read r bins (selected by the minimum chunk signatures of the incoming file). If we add x chunk signatures, the retrieval fails if the first k out of $N+x$ slots only contain n-markers. This gives us

$$\begin{aligned} p_{\mathrm{1W}r\mathrm{R,A}}(N,x) &= 1-\frac{\binom{N+x-r}{x-r}}{\binom{N+x}{x}} \\ &= 1-\frac{x(x-1)\ldots(x-r+1)}{(N+x)(N+x-1)\ldots(N+x-r+1)} \\ \wp_{\mathrm{1W}r\mathrm{R,A}}(\rho) &= 1-\frac{\rho^r}{(1+\rho)^r} \end{aligned}$$

If we delete x out of N chunk signatures then retrieval is possible if and only if the minimum chunk signature has not be deleted.

$$\begin{aligned} p_{\mathrm{1W}r\mathrm{R,D}}(N,x) &= x/N \\ \wp_{\mathrm{1W}r\mathrm{R,D}}(\rho) &= 1-\rho^r \end{aligned}$$

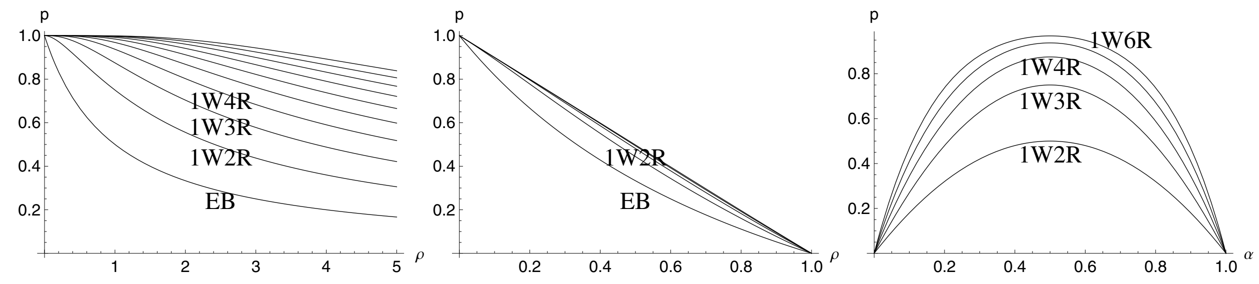

Fig. 4. Probability p of retaining access to a file after adding (left) and changing (middle) a proportion ρ of its chunk signatures. Retrieval probability after deletion do not increase from adding more reads. The right figure gives the retrieval probability against both of two merged files depending on the proportion of the first file's chunks in the merger.

If we change x out of N signatures, then we can retrieve with r reads if an r-marker is among the first r slots and there are no c-markers before this first r-marker. Thus, the slot assignments that allow retrieval are $[r,*]$, $[n,r,*]$, $[n,n,r,*]$ and so on until a pattern with $r-1$ n-markers followed by an r-marker. Thus

$$p_{1\mathrm{W}r\mathrm{R,C}}(N,x) \quad = \quad \sum_{\nu=1}^{r} \frac{\binom{N+x-\nu}{N-x-1,x-\nu+1,x}}{\binom{N+x}{N-x,x,x}}$$

The first limit functions are

$$\begin{aligned}
\wp_{1\mathrm{W1R,C}}(N,x) &= \frac{1-\rho}{1+\rho} \\
\wp_{1\mathrm{W2R,C}}(N,x) &= \frac{1+\rho-2\rho^2}{(1+\rho)^2} \\
\wp_{1\mathrm{W3R,C}}(N,x) &= \frac{1+2\rho-3\rho^3}{(1+\rho)^3}
\end{aligned}$$

Finally, we consider the set of signatures of a file that is the merger of two files. Clearly, even a simple concatenation will likely merge the last and the first chunk of the two files and hence change the set of signatures, but we model the situation by a union of the two sets of signatures. In this idealistic model, we deduplicate the merged file against both originals if signatures from both files are among the k smallest. This probability depends on the relative proportion of signatures belonging to one file, we call this number α. We omit the derivation but present numerical results in Figure 4.

When we compare the numerical results for additional read operations in Figure 4, we see that adding read operations is quite efficient for growing files and merged files, that in the case of changes a second read has some benefits, but that additional reads loose effectiveness rather quickly. For a growing file, the second read yields up to 25% more probability (at $\rho = 1$), adding the third read yields 38.49% more (at $\rho = 1.366$), and adding the fourth read yields up to 47.25% more (at $\rho = 1.702$). The middle graph in Figure 4 shows how adding more reads for a file with a proportion ρ of changed signatures quickly converges to the optimal value of $1-\rho$, which is the probability that the minimum chunk signature under which the file signature was filed has been changed. Additional reads are very effective in case of a merged file. EB can only deduplicate against a single file, but the second read operation deduplicates the merged file against both components with 50% probability if the two components have equal size. We also note that the benefits of additional reads accrue already if the file grows moderately or if a relatively small part of the file is changed.

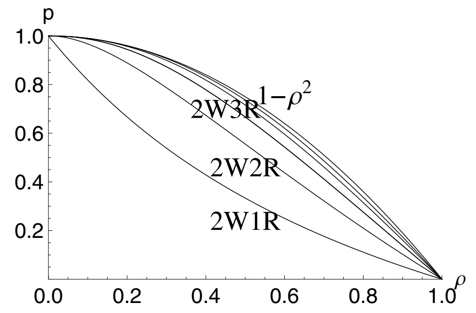

Fig. 5. Retrieval probability after changing a proportion ρ of chunks using schemes with 2 bin writes and 1, 2, 3, 4, and 5 lookups. The highest curve is that of the theoretical limit where we do lookups for all signatures in the file.

E. Retrievals with Two Writes

Increasing the number of bins in which we store chunk IDs from a file does not increase retrieval chances if the alteration results from extending a file or merging two files. When signatures are changed, the situation is different as we can now retrieve even if the minimum signature of the original file is changed. The slot assignments that allow retrieval with r lookups are those were the first r slots have zero or more n-markers, followed by an r-marker, or where the first r slots have one c-marker among one or more n-markers, followed by an r-marker. This gives

$$\begin{aligned}
p_{2\mathrm{W}r\mathrm{R,C}} \quad &= \quad \sum_{s=1}^{r} \binom{N+x-s}{N-x-1,x-s+1,x} \\
&+ \quad \sum_{s=2}^{r} (s-1)\binom{N+x-s}{N-x-1,x-s+2,x-1}
\end{aligned}$$

If we were to look up all bins in the signature of the file, then in the limit, the chances of a successful retrieval are given by the probability that the two signatures under which we stored the chunk IDs of the original file have not changed, *i.e.* $(1-\rho)^2$ if we changed a proportion ρ of signatures.

We display the curves for the limit as $N \to \infty$ in Figure 5, where we display the curves for 2W1L, 2W2L, 2W3L, 2W4L, 2W5L, and the function $\rho \to (1-\rho)^2$. The first scheme, 2W1L, performs in fact exactly as EB, so that writing to the second bin yields no benefit. (It would in the case of a shrinking file.) We can see that improvements of additional lookups are at first substantial. The second lookup gives 25% more retrieval probability (at $\rho = 1/3$), the third 34.99% (at $\rho = 0.393$), and the maximum achievable is 41.86% more (at $\rho = 0.4656$).

If we alter a file by deleting chunks then the number of reads does not improve retrieval probabilities, but the number of writes does. Indeed, if we write the file signature to w bins and delete x chunks from the file, then we retrieve the original file if one of the $N-x$ retained chunks is among the minimal w signatures. If $w > x$, this is always true, otherwise we obtain

$$\begin{aligned} p_{w\mathrm{W}r\mathrm{R,D}} &= 1-\binom{N-w}{N-x}\binom{N}{x} \\ &= 1-\frac{x(x-1)\dots(x-w+1}{N(N-1)\dots(N-w+1} \\ \wp_{w\mathrm{W}r\mathrm{R,D}}(\rho) &= 1-\rho^w \end{aligned}$$

F. Substitution of Secondary Reads

Extending EB with additional bin reads and writes can be implemented in a parallel manner if we use more than one storage device to store the bins. Of course, if we use a reasonable number of storage devices, then often processing a single file includes reading from the same device. We investigate a strategy that only makes one access to each storage device. In this strategy, we always read the bin corresponding to the minimal chunk signature, but select additional bins for reading only if the access is to different devices from which we read a bin with smaller signature. We restrict ourselves to schemes where we always write to one or always write to two bins (wherever they may be located). Since we can delay and bundle writes, we do not care about the location of the bins to which we write.

To assess these schemes, we calculate the probability of retrieving the file signature of the original file with a read to the bin specified by the k^{th} minimal chunk signature in an altered file, assuming that the bin with the minimal chunk signature did not contain the original file.

If we write chunk IDs into a single chunk and consider a change of x out of N signatures, then lookup with the minimum succeeds with slot assignment $[r,*]$. The slot assignments where the lookup with the minimum fails, but the one with the k^{th} minimal signature succeeds are $[n,\dots,n,r,*]$ where the r-marker is at position k. If we write the chunk IDs of a file into two bins, then we obtain additional assignments, where one of the n-markers is replaced by a c-marker. An additional pattern has an additional r-marker before the k^{th} r-marker, but not of course in the first position.

When we add signatures to an existing set, the slot arrangements are simpler. In the case of a scheme writing to two bins, lookup with the second minimal signature succeeds while the one with the minimum does not, has slot arrangements $[n,r,*]$, while lookup with the third succeeds with $[n,n,r,*]$ and $[n,r,r,*]$, which we can simplify borrowing from notation for regular expressions as $[n,?,r,*]$. As a result, retrieval probabilities are the same.

We give the results of our calculations in Figure 6. The curves show that lookup with the minimum and the second-minimal signature are in general quite superior to lookup with minimum and third-minimal signature and even more to lookup with minimum and fourth-minimal signature, but only in case of changes. The effects are less pronounced if the scheme writes chunk IDs of a file to two bins. We conclude that substitution of secondary reads is not likely to have a pronounced effect on deduplication rates.

TABLE II
DATA SETS USED IN EXPERIMENTS

Set	Nr Files	Size
Linux	3186361	35.678 GB
Archive	2090537	53.407 GB
M57	3606473	909.362 GB
HP	17669935	4540.209 GB

V. Experimental Results

To measure the effects of varying extreme binning, we experiment with four different data sets. The experiments allow us to compare deduplication rates, index sizes, and disk read operations. We compared perfect deduplication, extreme binning (1W1R), and our additional strategies 2W1R, 1W2R, 2W2R, and 3W3R. These latter strategies add disk accesses to the deduplication process.

A. Data Sets

Our first data set *Linux* consists of the Linux source code archive, containing versions 1.2.0 to 2.5.74 and representing 564 versions. This data set contains 3.19 million files and has a size of 35.7GB. This data set contains many small files. The second data set *Archive* is a backup of 251 web sites obtained from the Internet archive at www.archive.org. Each web site is captured between once and eight times with a typical value of two or three versions during several years. Since web sites were captured at least several months apart and as some where only captured once, this data set offers little chance for deduplication. The third data set, *M57*, consists of complete, daily backups of four workstations used by engineers and therefore offers excellent deduplication opportunities. Our fourth and final data set, *HP*, is similar, but more ample, consisting of thirty days of backups from 21 engineering workstations at Hewlett Packard. This is the only data set where we only had information on the chunks, but not direct access to the files. We summarize size information in Table II.

B. Deduplication Results

When simulating the deduplication process, we measured the overall storage of the data, the size of the index, and the number of bins accessed. Our numbers for the storage used does not include overhead, such as the index and the pointers to chunks already stored in the file manifest. We give the size

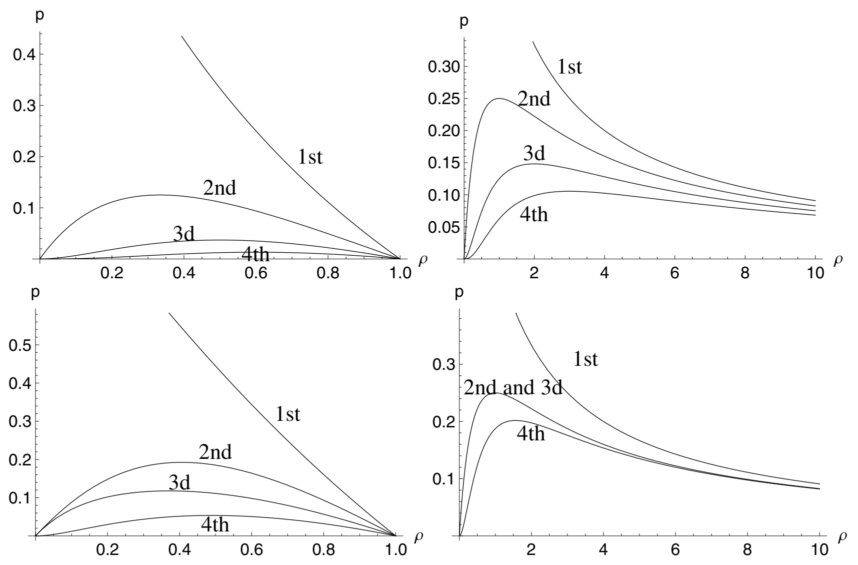

Fig. 6. Retrieval probabilities for schemes with one (top) and two (bottom) bins written, when changing (left) or adding (right) a proportion ρ of chunks. The graphs labeled "1st" are retrieval by minimum signature, whereas "2nd", "3d", and "4th" are retrieval probability of lookup with the 2nd, 3d, or 4th respectively minimal signature when retrieval with the minimal does not succeed.

of the index independently, and observe that the pointers to the manifest have an implementation dependent size, which is about one or two per thousand.

We give the results of our experiments in Table III. As we can see, the 2W2R strategy yields deduplication results that fall midway between EB and perfect deduplication, but more than doubles the number of bins read. They also increase the total size of the index. Our data show the strong dependence on the workload. For example, EB is almost as good as "perfect" for the Archive data set and the additional effort seems hardly worth the additional compression that we can achieve. Where deduplication is performing much better, the picture changes and additional effort adds considerably to the compression rate.

Our analysis in the previous sector has shown that additional reads help when chunks change and are added, while additional writes help when chunks change or vanish. As most systems experience a growth, the 1W2R scheme consistently outperforms the 2W1R scheme.

C. Processing Costs

A deduplication system incurs certain processing costs. We assume that the task of chunking (calculating chunk boundaries and calculating chunk signatures) is given to a different unit. All deduplication methods using variable chunks have to perform these tasks. We can use our simulator to measure processing costs directly attributable to index lookup. These consist of searching the primary index, reading and writing bins from RAM, and creating file manifests. As Table IV shows, the throughput depends on load and algorithm, but is sufficiently high to not become a bottleneck.

D. Comparison of EB Extensions

Extending EB to access more bins for each file processed increases storage savings, but also involves more IO in order to load and write bins. These IO-operations are on the critical path for deduplication performance. Incidentally, higher deduplication rates complicate file retrieval, but lower read performance is of lesser concern for archival systems. In order to assess the various algorithms, we have to balance more involved processing with savings in the storage costs. We make this comparison in terms of hardware saved and expended.

We first discuss a scheme where the secondary index is kept on disks. Access to the index is the primary bottleneck. To maintain a high rate of processing, we need to distribute the secondary index over many disks. In many architectures, we would use the same disks to store the archive as well as the index. The index would consume relatively little space, but be accessed heavily, taxing the interface between disk and system, whereas the archival data would consume much space but be rarely accessed. In this scenario, I/O bandwidth to the dispersed secondary index comes for free and extensions to EB save storage space by increasing the deduplication rate.

However, in a very large storage systems, the contents of the storage system would be "farther away", e.g. in disks in a SAN or in NAS. We then would implement the secondary index in directly attached disks. We compare now the number of disks necessary to store the secondary index, allowing it to sustain the index accesses necessary to process incoming data at a reasonable rate with the number of disks saved by higher deduplication rates. The exact trade-offs depend very much on the engineering of the storage system and the type of load, and our results are meant to prove that sometimes EB is not the best strategy.

We first calculated the time spent on accesses to chunk data for each strategy. We used an enterprise class disk, the Cheetah 15K.7SAS from Seagate [16] in our simulation and determined the amount of disk time in seconds in order to process 10 GB of data. Since we are interested in peak performance, the figure

TABLE III
DEDUPLICATION RESULTS

Linux			
Algorithm	Storage	Index Size	Bin Reads
perfect	4.676%	9.300 MB	0
3W3R	4.850%	10.421 MB	1535525
2W2R	5.121%	8.547 MB	1065138
1W2R	5.841%	5.629 MB	710794
2W1R	5.999%	8.547 MB	570489
EB	6.459%	5.629 MB	474271
2W2RS	5.555%	8.547 MB	819098
1W2RS	6.148%	5.629 MB	591481

Archive			
Algorithm	Storage	Index Size	Bin Reads
perfect	83.292%	234.605 MB	0
3W3R	84.353%	140.737 MB	805585
2W2R	85.231%	114.339 MB	637841
1W2R	85.874%	70.618 MB	590852
2W1R	87.039%	114.339 MB	254407
EB	87.439%	70.618 MB	244327
2W2RS	86.129%	114.339 MB	474309
1W2RS	86.651%	70.618 MB	445439

M57			
Algorithm	Storage	Index Size	Bin Reads
perfect	3.707%	161.830 MB	0
3W3R	4.081%	12.081 MB	822961
2W2R	4.200%	9.731 MB	568526
1W2R	4.538%	6.445 MB	439600
2W1R	4.568%	9.731 MB	292279
EB	4.815%	6.445 MB	257328
2W2RS	4.266%	9.731 MB	428663
1W2RS	4.639%	6.445 MB	347550

HP			
Algorithm	Storage	Index Size	Bin Reads
perfect	6.593%	1787.376 MB	0
3W3R	6.828%	74.952 MB	2944339
2W2R	6.927%	60.064 MB	1996842
1W2R	7.039%	40.175 MB	1585545
2W1R	7.174%	60.064 MB	1070592
EB	7.273%	40.175 MB	958820
2W2RS	7.042%	60.064 MB	1547227
1W2RS	7.139%	40.175 MB	1265663

TABLE IV
THROUGHPUT MEASUREMENTS

Algorithm	Processing throughput (MB/sec)		
	Linux	HP	M57
3W3R	562.058	794.528	554.397
2W2R	702.573	1096.804	720.132
EB	1014.827	2060.147	1475.460

of interest for us is the 99 percentile. We give the numbers in Table V.

If we want to be able to digest 10 GB per hour in the HP data set, one disk suffices to store chunk information, even for the more intensive 3W3R strategy. An hour has $60 \cdot 60 = 3600$ seconds, and the 99 percentile of the throughput requirement for 3W3R is 519.949 seconds. Thus, this single disk storing chunk information sees a maximum utilization of 14.5% at the single disk storing all bins. The storage needs in this case are also very limited. First, a peak of 10 GB per hour

TABLE V
STATISTICS OF DISK ACCESS TIME IN SECONDS PER 10 GB OF PROCESSED DATA FOR CHUNK MANAGEMENT USING A CHEETAH 15K.7SAS (WITH ROTATIONAL LATENCY OF 2.0 SEC AND RANDOM ACCESS TIMES OF 3.4 SEC FOR READS AND 3.9 SEC FOR WRITES).

M57			
Algorithm	Mean (s)	St. Dev.	99 Perc.
3W3R	73.829	74.525	318.233
2W2R	53.076	58.531	246.188
2W2RS	44.531	56.961	237.768
1W2R	38.258	38.672	159.803
1W2RS	32.680	37.959	153.870
2W1R	36.172	55.556	230.192
EB	27.200	37.359	148.642

HP			
Algorithm	Mean (s)	St. Dev.	99 Perc.
3W3R	62.349	88.923	506.860
2W2R	45.146	69.518	387.129
2W2RS	39.680	67.655	379.954
1W2R	32.717	47.468	247.143
1W2RS	28.869	45.925	243.290
2W1R	33.895	66.039	372.144
EB	25.181	44.594	239.497

TABLE VI
NUMBER OF 512B PAGES OF BINS WRITTEN PER 1 GB OF DATA PROCESSED.

M57		
Algorithm	Written Pages	Rewritten Pages
3W3R	7146.906	5961.739
2W2R	5029.127	4224.803
1W2R	2563.771	2166.366
EB	2563.771	2166.366

HP		
Algorithm	Written Pages	Rewritten Pages
3W3R	6606.071	4403.873
2W2R	4303.645	2827.44
1W2R	1329.357	2071.413
1W2RS	1329.357	2071.413
EB	1329.357	2071.413

corresponds to a much lower average rate of ingress. Let's say (arbitrarily) that the average throughput is 1 GB per hour. This corresponds to a yearly storage need of 8.550 TB in year before deduplication, but thanks to deduplication, 3W3R only stores 0.58 TB of data, using less than a complete 1 TB disk for storage. With EB, the peak utilization using our single disk for chunk storage is 6.8% and the total storage after deduplication is 0.62 TB.

We now assume a much larger storage system that stores annually 10 PB of data. This corresponds to an average ingress of 0.325 GB per second. We assume a conservative 10-fold peak load of 3.25 GB per second. The deduplication engine needs to be able to process this peak load. If we use EB, then processing 1 GB needs an accumulated service time of 24.4897 seconds at the disks storing the bins with the chunks in them. If we want a maximum of 50% utilization at these disks, then we need $2 \cdot 24.4896 \cdot 3.25$ disks or 160 disks to store chunk information. By comparison, 3W3R needs a total service time of 51.9949 seconds at these bin carrying disks,

TABLE VII
TOTAL DISKS NEEDED FOR A 10 PB PER YEAR STORAGE SYSTEM USING DISKS OF 1 TB OR DISKS OF INITIAL 1TB CAPACITY, BUT WHOSE CAPACITY INCREASES BY 50% ANNUALLY

M57		
Algorithm	1 TB disks	1 TB+ disks
1W1R	2505	1352
1W2R	2373	1287
2W1R	2434	1340
2W2R	2261	1256
3W3R	2248	1271

HP		
Algorithm	1 TB disks	1 TB+ disks
1W1R	3793	2051
1W2R	3681	1995
2W1R	3829	2111
2W2R	3716	2057
3W3R	3627	2048

TABLE VIII
THE TOTAL NUMBER OF 4 KB PAGES OF BINS WRITTEN AND READ FOR BOTH THE ORIGINAL ALGORITHM OF UPDATING BINS AND THE OPTIMIZED ALGORITHM OF UPDATING BINS. *Opt* REFERS TO THE OPTIMIZED ALGORITHM AND *Ori* TO THE NAIVE ALGORITHM.

M57				
Algorithm	Write#(Opt)	Read#(Opt)	Write#(Ori)	Read#(Ori)
3W3R	464035	4963045	1125403	4471342
2W2R	352078	3048890	820954	2823706
1W2R	211502	1706617	452365	1587703
1W2RS	211502	1468147	452365	1363978
EB	211502	1191989	452365	1103546

HP				
Algorithm	Write#(Opt)	Read#(Opt)	Write#(Ori)	Read#(Ori)
3W3R	3112302	15517182	5543020	14727540
2W2R	2322952	10258881	3880937	9716984
1W2R	1402298	6187802	2133429	5955192
1W2RS	1402298	5297709	2133429	5099018
EB	1402298	4502486	2133429	4332251

which translates to needing $2 \cdot 51.9949 \cdot 3.25$ or 338 disks to store the chunks for the deduplication engine. In a single year, our system stores 10 PB user data. However, thanks to deduplication, we will only need to store 7.273% with EB and 6.828% with 3W3R of these data, amounting to 728 or respectively 683 disks of capacity 1 TB. If we maintain the system for five years, these numbers need to be multiplied by five and added to the number of disks needed for chunking. We can now calculate the total number of disks needed for our 10 PB per year system. We did so for all our basic schemes in Table VII. The second column there shows the total number of disks used assuming that we use 1 TB disks for storing user data. The third column gives the total number of disks, assuming that each year, the storage capacity of these disks increases by 50%. Our results show that all schemes perform within 5% of each other using both measurements, and that the optimal scheme is not EB. These numbers change linearly if we change the yearly amount of data to be stored. For example, in a system with 1 PB of data per year, we would need roughly one tenth of these disks.

Our calculation is somewhat naive. For instance, we completely ignore the processing costs of processing various bins. We also assume that disks with bin information are separate from disks storing user data, whereas in many systems, bins and user data can be stored on the same disks. Finally, we likely are over-provisioning the deduplication engine with disks. Additionally we can use a strategy where we limit the number of bins read in a situation of high load. For example, if we use 2W2R and if the load is high, we forego reading the second bin. This looses some deduplication opportunities, but helps moving the deduplication process along.

Our next task is a similar comparison where we store the secondary index in flash memory. Flash memory is more expensive than disk storage ($1 per GB for flash versus $0.05 per GB for disks), has a limited number of overwrites (depending on the type 1000 to 100000 times), has much faster, but asymmetric access time (200–500 microseconds per read, for writes two or three times more, both on the raw device without FTL). Table VIII gives the number of pages accessed (read and written) for each load. We distinguish between two write strategies, a naive method that reads the complete bucket and then writes its elsewhere, and a more sophisticated method that only writes the new pages that are added to the bucket. Based on these numbers, we can calculate the annual storage requirements for a 10 PB per year system. Table X gives the results. We observe that the storage needs per PB of processed data are surprisingly low. Even with 3W3R, we would need between 165 GB and 84 GB of new flash space for the two backup workloads. The number of overwrites observed is in the lower three digits. While flash can be rated at only a thousand write-erase cycles, these are specifications and actual flash resilience might be ten times higher. In this case, the number of times that we write to this limited amount of memory would be low enough so that flash lasts at least the normal economic life-span of a disk. We also calculated maximum bandwidth requirements for combined read and write operations. Our measurements for the backup workloads show that we read two or three times more from flash memory than we write to it. This matters as the sustained transfer rate of reads can be almost double that of writes. Using the same methodology as before, we assume a peak load of 10 times the average and use the bandwidth requirements at the 99 percentile to calculate the combined read and write bandwidth requirements for each scheme in the last column of Table X. The architecture of flash memory determines the bandwidth that actual flash storage can offer and sustained transfer rate differs. Today (January 2012), a single SSD might suffice and two will suffice for the bandwidth requirements of our hypothetical system. After a possible initial investment in an additional SSD, the storage needs become the limiting factor. We assume that the costs of storage in SSD is 20 times the costs of storage on disk. We then calculate the annual needs for disks for storage and for SSD (in terms of disks) for maintaining the index and obtain a progression, given

TABLE IX
ANNUAL COSTS OF A 10 PB SYSTEM PER YEAR IN TERMS OF DISKS USING SSD FOR INDEX MAINTENANCE.

Algorithm	Costs HP	Costs M57
3W3R	695.698	418.479
2W2R	703.035	428.36
EB	734.213	487.037

TABLE X
FLASH MEMORY NEEDS

M57				
Algorithm	Index Size per PB	Pgs written per PB	Number of Overwrites	Maximum Bandwidth
3W3R	13.285 GB	$1.24 \cdot 10^9$	372.8	390.1 MB/sec
2W2R	10.701 GB	$9.03 \cdot 10^8$	337.6	324.5 MB/sec
EB	7.087 GB	$4.98 \cdot 10^8$	280.9	146.5 MB/sec

HP				
Algorithm	Index Size per PB	Pgs written per PB	Number of Overwrites	Maximum Bandwidth
3W3R	16.509 GB	$1.22 \cdot 10^9$	295.8	492.8 MB/sec
2W2R	13.229 GB	$8.55 \cdot 10^8$	258.5	324.5 MB/sec
EB	8.849 GB	$4.70 \cdot 10^8$	212.4	219.9 MB/sec

in Table IX. These numbers show that 3W3R is uniformly superior in terms of cost to 2W2R and EB.

E. Adaptive Schemes

In our comparisons, we assumed that we access the same number of bins when processing a file. Thus, the secondary index needed to be implemented in a way that allowed the peak demand to be satisfied. However, we can also adopt a mixed strategy. If the load is low, we can read and write 2 bins, but if the load is high and access to the index becomes a bottleneck, we only read and write one bin, i.e. we fall back on EB. We tried out this method by simulation, where we read and write two bins with probability $1-P$ and read and write one bit with probability P and observed the resulting compression. The result is depicted in Figure 7. The results are very encouraging. Basically, the compression rate depends linearly on the probability. In particular, if we have a system

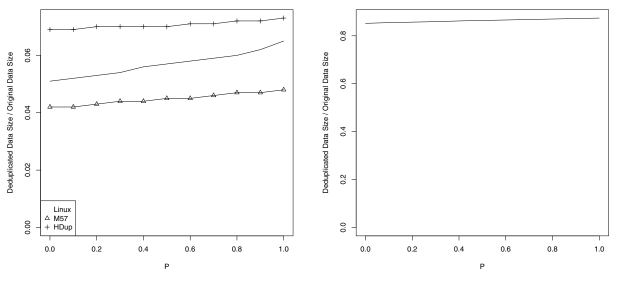

Fig. 7. Deduplication rate using a mixture of 1W1R and 2W2R. The x-axis is the rate P at which the system uses the 1W1R scheme. The left graph contains the data for Linux, M57, and HP data sets and the right one the Archive (with much less impressive compression).

that can do 2W2R most of the time, we get almost all the benefits.

F. Conclusions from the Experiments

Our experimental analysis covered four different data sets, but only the two data sets representing backup loads where extensive enough to obtain meaningful numbers for bandwidth requirements. In this case, we can obtain the costs of two architectures, one based on a disk system in order to solely support the index and one based on flash storage. We notice that the first architecture fails to use the idle bandwidth of disks in a largely archival system. In both cases, we came to the result that EB is not always the optimal strategy. Of course, the designer of an actual system can calculate the bandwidth requirements of all components of the system and might find the network to be the bottleneck. Most intriguing is our last experiments where we picked between EB and 2W2R on a random base and observed practically linear scaling with the probability. This means that if we saturate components with a more involved strategy, there is little harm done with temporarily employing a simpler strategy, such as EB.

VI. COMPARISONS

We briefly report on an experimental comparison with the schemes of Aronovich *et al.* [14] and Románski *et al.* [15]. For the work of Aronovich *et al.* we substituted their mode of determining the equivalent of the file signature for using the k maximum chunk signatures in a file. For the work of Románski *et al.* we implemented their complete scheme, but did not evaluate their strong point, which is addressing the IO bottleneck through prefetching.

Our numbers given in Table XI compare the schemes on compression and index site. Since we did not have access to the original data for the HP data set, we could not evaluate the work by Románski *et al.* for this set. The use of Aronovich's method for defining the file signature does not play out as compression is slightly worse even than EB with the exception of the Archive data set. Possibly, Aronovich's method would be more useful if the load were in the exabyte – petabyte range. In the absence of datasets of this size, a theoretical study should be undertaken, but would not fit into the present work. We plan to do so as future work.

Anchor-driven sub-chunk deduplication yields much worse compression ratios and has in general slightly higher RAM use. It should be much better with regards to I/O, but it is unclear whether a distributed solution as we advocate would not also be very competitive. Since EB was conceived for backup loads with little locality and Románski's method was not, a more favorable picture could emerge with a different backup load.

VII. CONCLUSIONS

We compared the natural extensions of Extreme Binning were we access a larger number of bins. Our theoretical results show the benefits of these extensions when a new file is a moderate alteration of an already existing file. Unfortunately,

TABLE XI
COMPARISON OF EXTREME BINNING (EB), AND ADAPTATION BY THE FILE SIGNATURE DEFINITION OF ARONOVICH *et al.* (AR) AND ANCHOR-DRIVEN SUB-CHUNK DEDUPLICATION BY ROMÁNSKI *et al.* (RO).

Scheme	Compression Rate	Index Size
	Linux	
EB	6.459%	5.902MB
Ar	6.773%	5.780MB
Ro	11.870%	5.314 MB
	Archive	
EB	87.439%	70.618 MB
Ar	87.077%	61.923 MB
Ro	97.412%	56.584 MB
	HP	
EB	7.273%	42.126 MB
Ar	7.295%	41.848 MB
	M57	
EB	4.482%	6.758 MB
Ar	4.875%	6.680 MB
Ro	17.876%	19.318 MB

these theoretical results cannot be applied directly to actual loads. For instance, if essentially the same file is stored many times over (e.g. by taking complete system images over and over again), EB will always do at least a decent job at compressing and accessing more bins is not worthwhile. Nevertheless, the theoretical results give some insight under which type of loads the extensions are more valuable. Our experimental results show that in general, EB is close to perfect deduplication, and that even reading and writing one more bin gives results about midway between EB and perfect deduplication.

Our calculation indicates that depending on the load and the system architecture, the extensions trade better deduplication for higher hardware costs in order to support the higher I/O needs of accessing the secondary index. Our main contribution is proof that mixed schemes, where we switch to EB under high loads only, give deduplication rates close to the ones of the expansion.

In short, while EB appears to be rather close to optimal, *less* extreme binning yields benefits. As is often the case, moderation pays off.

ACKNOWLEDGMENTS

This research was supported in part by the Department of Energy under Award Number DE-FC02-10ER26017/DE-SC0005417, by the National Science Foundation under Grants IIP-0934401 and CCF-1219163, and by the industrial members of the Storage Systems Research Center, including: EMC, Hitachi, Huawei, IBM Research, LSI, NetApp, Northrop Grumman, Permabit and Samsung.

Zhike Zhang was a visiting scholar from Northwestern Polytechnical University, supported by a schoalrship from the government of China.

REFERENCES

[1] B. Zhu, K. Li, and H. Patterson, “Avoiding the disk bottleneck in the data domain deduplication file system,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 269–282.

[2] N. Mandagere, P. Zhou, M. Smith, and S. Uttamchandani, “Demystifying data deduplication,” in *Proceedings of the ACM/IFIP/USENIX Middleware’08 Conference Companion*. ACM, 2008, pp. 12–17.

[3] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, “Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup,” in *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, 2009.

[4] G. Forman, K. Eshghi, and S. Chiocchetti, “Finding similar files in large document repositories,” in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, p. 400.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, “A low-bandwidth network file system,” in *Proceedings of the eighteenth ACM symposium on Operating systems principles*. ACM, 2001, pp. 174–187.

[6] S. Quinlan and S. Dorward, “Venti: a new approach to archival storage,” in *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, vol. 4, 2002.

[7] D. Bhagwat, “Deduplication for large scale backup and archival storage,” Ph.D. Dissertation, University of California at Santa Cruz, September 2010.

[8] A. Broder, “On the resemblance and containment of documents,” in *Proceedings of the Compression and Complexity of Sequences*, vol. 1997, 1997.

[9] ——, “Identifying and filtering near-duplicate documents,” in *Combinatorial Pattern Matching, Springer LNCS-1848*. Springer, 2000, pp. 1–10.

[10] D. Bhagwat, K. Eshghi, and P. Mehra, “Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus,” in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD ’07, 2007, pp. 105–112.

[11] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes, “Jumbo Store: Providing efficient incremental upload and versioning for a utility rendering service,” in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007, pp. 123–138.

[12] J. Min, D. Yoon, and Y. Won, “Efficient deduplication techniques for modern backup operation,” *IEEE Transactions on Computers*, vol. 60, pp. 824–840, 2011.

[13] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, “Sparse indexing: large scale, inline deduplication using sampling and locality,” in *Proccedings of the 7th conference on File and Storage Technologies*. USENIX Association, 2009, pp. 111–123.

[14] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, “The design of a similarity based deduplication system,” in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR ’09. New York, NY, USA: ACM, 2009, pp. 6:1–6:14. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534539

[15] B. Romanski, L. Heldt, W. Kilian, K. Lichota, and C. Dubnicki, “Anchor-driven subchunk deduplication,” in *SYSTOR*, 2011, p. 16.

[16] Seagate Technology LLC, “Product Manual Cheetah 15K.7 SAS,” 2010.