

Fast File Existence Checking in Archiving Systems

SASO TOMAZIC, University of Ljubljana
VESNA PAVLOVIC and JASNA MILOVANOVIC, University of Belgrade
JAKA SODNIK, ANTON KOS, and SARA STANCIN, University of Ljubljana
VELJKO MILUTINOVIC, IPSI Belgrade

This article presents a new Fast Hash-based File Existence Checking (FHFEC) method for archiving systems. During the archiving process, there are many submissions which are actually unchanged files that do not need to be re-archived. In this system, instead of comparing the entire files, only digests of the files are compared. Strong cryptographic hash functions with a low probability of collision can be used as digests. We propose a fast algorithm to check if a certain hash, that is, a corresponding file, is already stored in the system. The algorithm is based on dividing the whole domain of hashes into equally sized regions, and on the existence of a pointer array, which has exactly one pointer for each region. Each pointer points to the location of the first stored hash from the corresponding region and has a null value if no hash from that region exists. The entire structure can be stored in random access memory or, alternatively, on a dedicated hard disk. A statistical performance analysis has been performed that shows that in certain cases FHFEC performs nearly optimally. Extensive simulations have confirmed these analytical results. The performance of FHFEC has been compared to the performance of a binary search (BIS) and B+tree, which are commonly used in file systems and databases for table indices. The results show that FHFEC significantly outperforms both of them.

Categories and Subject Descriptors: H.3.2 [Information Systems]: Information Storage and retrieval—*Content analysis and indexing*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: File systems management, hash-table, archiving, files backup/recovery, files sorting/searching, performance evaluation

ACM Reference Format:

Tomazic, S., Pavlovic, V., Milovanovic, J., Sodnik, J., Kos, A., Stancin, S., and Milutinovic, V. 2011. Fast file existence checking in archiving systems. *ACM Trans. Storage* 7, 1, Article 2 (June 2011), 21 pages.
DOI = 10.1145/1970343.1970345 <http://doi.acm.org/10.1145/1970343.1970345>

1. INTRODUCTION

The amount of data being created each year increases exponentially. A Berkeley study in 2003 [Lyman et al. 2003] estimates that about four exabytes of new digital material produced in 2002 was stored on magnetic media, and that the growth from 1999 to 2002 was about 78% per year. An updated report will be available at the end of 2008

This work was supported by the Slovenian Research Agency, under research program P2-0246.

Authors' addresses: S. Tomazic, J. Sodnik, A. Kos, and S. Stancin, University of Ljubljana, Faculty of Electrical Engineering, Trzaska 25, Ljubljana, Slovenia; email: {saso.tomazic, jaka.sodnik, anton.kos, sara.stancin}@fe.uni-lj.si; V. Pavlovic, J. Milovanovic, and V. Milutinovic, University of Belgrade, Faculty of Electrical Engineering, Bulevar kralja Aleksandra, Beograd, Serbia; email: {vesna.pavlovic; jasna.milovanovic}@gmail.com; vm@etf.rs.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2011 ACM 1553-3077/2011/06-ART2 \$10.00

DOI 10.1145/1970343.1970345 <http://doi.acm.org/10.1145/1970343.1970345>

[Bohn et al. 2008]. However, if we suppose that the growth rate has not changed since 2002, we can estimate that the amount of new digital material in 2008 will already be 71 exabytes. This explosion of data has motivated researchers to find ways for improving the performance of archival storage systems.

During the archiving process, there are many new submissions that are actually unchanged files not needing to be re-stored in the archive, even though there is a request for the named entity to be archived. However, comparing each new file to be archived to all stored files is time consuming and would make an archive system too slow.

Instead of comparing the entire files, it is sufficient for the digest of a new file to be compared to digests of all stored files. A file digest, or “fingerprint”, is a fixed-length bit pattern that, within a range of probability, uniquely identifies a file. Because files are usually much longer than their digests, many different files can have the same digest, which is referred to as a “collision”. However, if the algorithm for obtaining the file digest is chosen properly, and if the digest is long enough, the probability that two meaningful files would have the same digest is negligible.

Cryptographic hashes such as MD5 [RFC 1321 1992] or SHA-1 [FIPS 180-2 2002] have good collision-resistant properties and can be used for this purpose. For a good hash function, that is, a function with uniformly distributed output values, the probability of collision is (see Appendix A):

$$P_{coll} \doteq \frac{K(K-1)}{2 \cdot 2^n}, \quad (1)$$

where K is the number of hashes stored in the system, and n is the length of the hash, that is, the number of bits. In accordance with equation, for a system that works with one billion 160-bit hashes, the probability of collision would be approximately 10^{-30} .

When a request for archiving a new file arrives, the proposed hash-based procedure would create a hash of the file and submit it to the archiving system before submitting the file itself. The system would then check all stored hashes for a match and respond with the appropriate answer: YES if the hash exists, and NO if it does not. A common approach would be to create an index of all stored hashes and then perform a Binary Index Search (BIS). BIS requires storage only to store hashes and no additional data storage (storage overhead) is required. For this reason, it is a good choice according to the storage-performance criterion. Alternatively, B+tree could be used for storing hashes; it keeps data sorted and allows searches and insertions in logarithmic time, however it requires additional storage to store the tree structure. We propose a new algorithm, which we call “Fast Hash-based File Existence Checking” (FHFEC). The proposed algorithm outperforms significantly both of these algorithms.

This article is organized as follows: In Section 2, we briefly overview the existing systems for file existence checking. In Section 3, we describe the FHFEC algorithm for two different cases: one, when the hashes are stored in random access memory (RAM-FHFEC); and two, when a dedicated hard disk is used for storage (HD-FHFEC). In Section 4, we present the results of a statistical performance analysis and storage-speed trade-off, In Section 5, we present simulation results. The comparison to other algorithms is presented in Section 6. We conclude with Section 7. For clarity of presentation, the derivations of equations needed for the analysis are found in the appendices.

2. BACKGROUND AND RELATED WORK

Hashing [Cormen et al. 2001] has the ability to reduce search space and thus to derive greater efficiency. Therefore, it is primarily used to index and retrieve items in databases, because it is faster to find the item using a shorter hashed key than to find it using the original value [IBM 2010]. For example, OTHER [Jovanov et al. 2002] uses

ordered table hashing operations for the acceleration of non-numeric, database and information retrieval operations. Standard database operations are implemented using ordered table hashing operations that rely on low-level hashing primitives. Hashing is also used in many encryption algorithms.

The identification of duplicate data is of great importance in storage space optimization. During recent years, hashing has also been used in strategies for identifying identical or very similar files, in order to avoid duplicating files [Policroniades and Pratt 2004].

Venti [Quinlan et al. 2002] is a network storage system intended for archival purposes. In this system, files are broken into blocks of fixed size, and for each block a unique SHA hash is calculated. The hash of a block's contents acts as the block identifier for read/write operations. During incremental storing, duplicate blocks (indicated by identical SHA values) are stored only once.

LBFS [Muthitacharoen et al. 2001] is a network file system designed for low bandwidth networks; it saves bandwidth by exploiting similarities between files. It uses the fact that the same chunks of data often appear in multiple files or versions of the same file. Therefore, it divides files into chunks using Rabin fingerprints [Broder 1993] and calculates and stores an SHA hash for each chunk stored in a given file system. Instead of sending the entire file, only the SHA hashes of each chunk in the file are sent. The receiver looks for potential matches among the chunks it possesses, and only the chunks the receiver does not possess are sent.

Pastiche [Cox et al. 2002] is a simple backup system that exploits excess disk capacity to perform peer-to-peer backup with no administrative costs. It stores data on a disk as chunks, and each chunk has an owner list that names the set of nodes interested in a chunk. When a newly written file is closed, it is scheduled for chunking, and each chunk is named by taking an SHA-1 hash of its contents. Before writing a chunk to disk, Pastiche first checks to see if it already exists. If it does, the local host is added to the owner list; otherwise, the chunk is written to the disk.

The storage reduction scheme called REBL [Kulkarni et al. 2004] effectively reduces data sizes. Compression, duplicate-block suppression and delta-encoding are used in order to eliminate redundant data in an efficient manner. REBL uses super-fingerprints to identify similar blocks, while reducing the computational requirements for matching blocks. It achieves more effective data reduction by identifying relationships among similar blocks, rather than only among identical blocks.

One solution for the efficient management of billion file (or chunk) signatures based on the use of hashes is given in Rudan et al. [2006]. Signatures are managed from a specially introduced new file signature management layer; this approach enables handling file signatures in a rapid way. The concept proposed in this article served as a guideline for our own work. Its basic idea to break the whole digital-signature-value domain into regions and to have a bit structure with one bit corresponding to each region is explored in greater depth in this article.

B+tree [Bayer and McCreight 1972] is the most commonly used method in file systems for block indexing, and in databases for table indices. Actually, it represents a tree data structure that allows searches and insertions in logarithmic time, by keeping data sorted. The primary value of B+tree is in storing data for efficient retrieval in a block-oriented storage context. The space required to store the tree is $O(C)$, where C is the capacity of the system, that is, the number of indexes that can be stored in the system.

On the other hand, a binary index search (BIS) [Knuth 1997] is known to be the most efficient non-hash-based method of searching and incremental sorting in respect of storage minimization, as it requires no additional storage beside the storage needed to store values. BIS adds items to a list over time while keeping the list sorted at

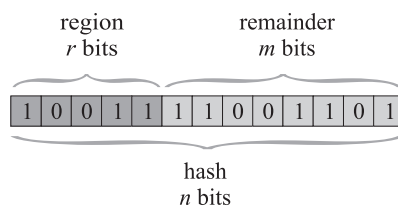


Fig. 1. n bits of a hash are divided into two parts: r bits representing the region the hash belongs to, and m remaining bits specifying the exact hash in the region.

all times. It provides fast searching; however, the update operation is performed very slowly.

In order to improve the performance of our system, we developed an efficient hash-based search which deals with data search alongside updating, which is significantly faster than both BIS and B+tree.

The proposed system for hash checking can be used for hashes either of whole files or of chunks. For convenience, we will refer only to whole files throughout the rest of the article.

3. FAST HASH-BASED FILE EXISTENCE CHECKING ALGORITHM

When a file is archived, its hash is stored in RAM or on HD for later checking of its existence. When a new file is to be archived, its hash is calculated locally and sent to the archiving system. If the same hash already exists in the system, it indicates that this file has already been archived and does not need to be re-archived.

The Fast Hash-based File Existence Checking (FH FEC) algorithm that we propose works with n bits long hashes, which implies that their domain has the size of $N = 2^n$ possible values. The whole domain of hashes, sorted in binary order, is divided into $R = 2^r$ regions. Therefore, the number of all possible hashes in each region is:

$$M = \frac{N}{R} = 2^{n-r} = 2^m, \quad (2)$$

where r is the number of bits of the hash that denote the region and m is the number of remaining bits, as shown in Figure 1.

The region to which the selected hash belongs is easily found, as it can be addressed directly by the first r bits of the hash. In this way, hashes n bits long are actually shortened to a length of r bits, which is necessary for the classification of hashes into regions. This process is equivalent to calculating a new shorter hash from the input hash.

The structures for storing hashes are slightly different in RAM- and HD-based systems.

3.1. RAM-Based System

The structure for storing hashes in a RAM-FH FEC system is shown in Figure 2.

The pointer array consists of R pointers. The first r bits of the hash represent the address of the corresponding pointer. If any of the hashes belonging to a specific region are stored in the system, the corresponding pointer points to the memory location where the first stored hash from that region resides. Otherwise, if no hashes from that region exist in the system, the corresponding pointer has a null value.

As the first r bits of a hash can be determined from the region, only the last m bits of the hash (the hash remainder) need to be stored. Each stored hash remainder has an attached pointer that points to the next stored hash remainder from the same region.

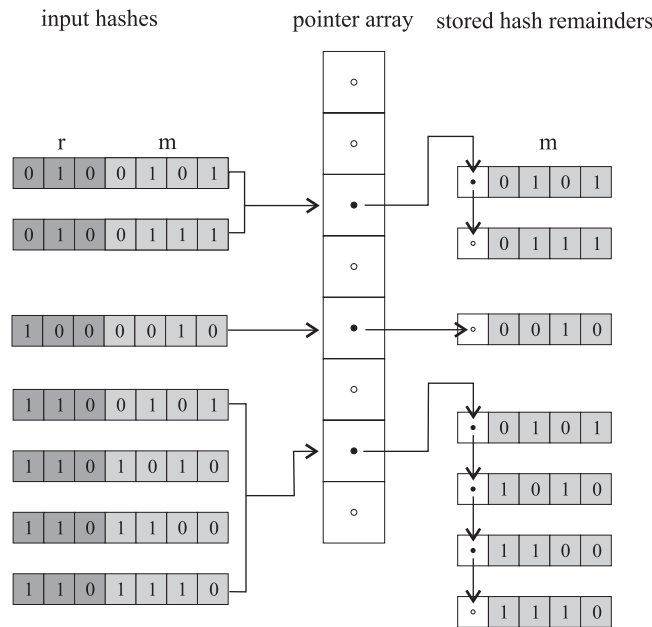


Fig. 2. Structure for storing hashes in RAM-FHFEC. The first r bits of an input hash specify the hash region, and thus the address of the pointer in the pointer array. A null pointer (empty circle) in the pointer array denotes that no hash from that region exists in the system. If at least one hash from the region exists, the corresponding pointer (filled circle) points to the first stored remainder of the hash from that region.

The last pointer has a null value. In this way, stored hashes belonging to one region are connected in a linked list [Parlante 2001].

The procedure of examining if a certain hash already exists in the system is shown in Figure 3. When a new hash arrives, the pointer of the corresponding region is checked. If it has a null value, we get the answer NO (hash does not exist) immediately; otherwise, the answer is POSSIBLE YES (the hash may exist). In the latter case, we know that at least one hash from the corresponding region exists, and we must therefore search for a match through the linked list of stored hash remainders. If a match is found, the final answer is YES; otherwise, the final answer is NO.

When the answer is NO, it indicates that the corresponding file must be archived and its hash added to the system. Since all the hashes belonging to one region are connected in a linked list, the insertion of a new hash is simple. The hash remainder is stored in a free RAM location, and the pointer in the pointer array or the last pointer in the linked list is set to point to this location.

Essentially, RAM-FHFEC is a hash table of hashes. However, there is no need to compute a new hash of the original hash as the first r bits of the original hash can serve for its mapping to the table.

3.2. HD-Based System

In HD-FHFEC, hashes and pointers are stored on a dedicated HD. An HD is a sealed unit containing a number of platters in a stack. Electromagnetic read/write heads are positioned above and below each platter. As the platters spin, the drive heads move in toward the center surface and out toward the edge. Data are stored on the HD in thin concentric bands. A head, while in one position, can read or write a circular ring or band called a track. Sections within each track are called *sectors*. A sector is the

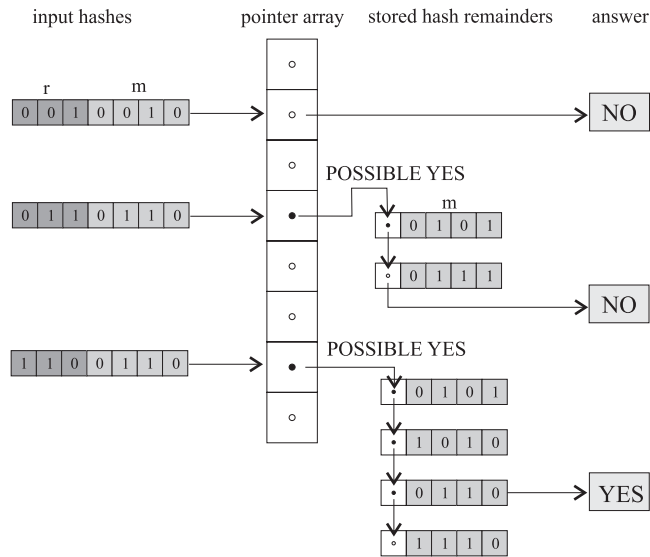


Fig. 3. The FHFEC procedure for examining if a certain hash exists in a RAM-based system. We have two possible final answers: YES if the hash is already in the system, and NO if the hash is not in the system. If the corresponding pointer has a null value the answer is NO; otherwise the answer is POSSIBLE YES and all stored remainders from that region must be checked in order to get the final answer, that is, YES or NO.

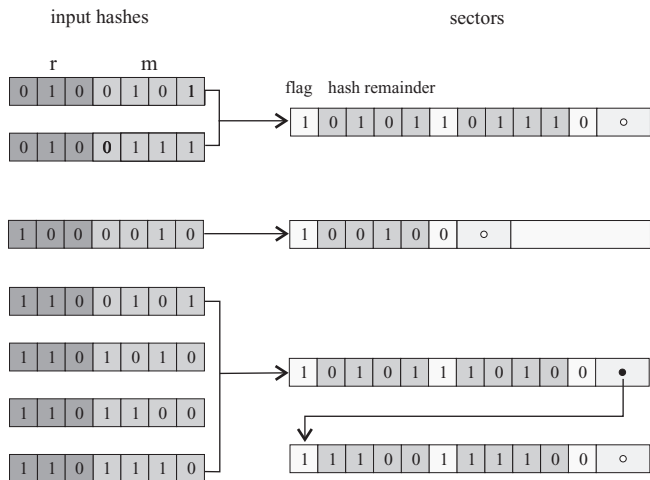


Fig. 4. Hash organization in HD-FHFEC. The first r bits of a hash map the address of the disk sector where the first hash remainder from that region is stored. More than one hash remainder can be stored in one sector. Flags after each hash remainder indicate whether or not there are more stored remainders. When there is not enough space in that sector for the next hash remainder, the pointer at the end of the sector points to the sector where further hash remainders are stored.

smallest physical storage unit on a disk. It is common for each sector to be able to store 512 bytes of data; however, the size of a sector can be modified.

Organization of hashes on the HD is shown in Figure 4. The first r bits of a hash represent the region; they map the address of the disk sector where the first hash remainder from that region is stored (if there is one). For each region there is a bit

Table I. Average Percent of Daily Change

Disk	ACC	PHAR
C - system files	0.49%	0.88%
D - program files	6.71%	2.13%
E - user files	0.02%	0.08%

ACC – academic institution

PHAR – pharmaceutical firm

(flag) in the corresponding sector that is set if at least one hash remainder is stored in it. Initially, all flags are set to zero, meaning that the system is empty. Each one of the stored hash remainders is followed by a flag, which is set if there are more hashes stored in that sector. If the flag is set to zero, there are two possibilities: first, there are no more hashes from that region stored in the system; or second, there is not enough space in the sector to store the remaining hash remainders from the region. In the first case, the null bit is followed by a null pointer, and in the second case, by a pointer that points to the sector where further hash remainders are stored. The Logical Block Addressing (LBA) [PCGuide 2010], that is, specifying a cylinder, head and sector address, can be used for addressing the sectors.

The following procedure is used to examine if a certain hash already exists in the system: first, the region this hash belongs to is determined by observing the first r bits of the hash; then the corresponding sector is checked for emptiness. If it is empty, the answer is NO; otherwise, we must search for a match among all stored hashes from that region, that is, to look through all the sectors that encompass the hash remainders belonging to that region. If the final answer is NO, we need to store the hash in this corresponding sector, or if there is not enough space, to a new sector that will be linked to this one. In the latter case, this implies storing the hash remainder in the new sector, setting the pointer at the end of the old sector to point to the new sector, and setting the flag at the end of the new hash remainder to zero and the following pointer to null.

4. PERFORMANCE ANALYSIS

Performance analysis was carried out by calculating the average number of steps necessary to find a final answer regarding hash existence. In RAM-based systems, with the term “step” we are considering the number of memory reads/writes, while in disk-based systems this term denotes the number of disk head movements. In the latter case, we can disregard the time needed for reading/writing from/to disk sectors, because it is many times shorter than the time needed for the movement of the disk head.

The average number of steps depends, among other things, on the probability P_{new} that a hash is new, that is, that its corresponding file has not yet been archived. P_{new} is a parameter of the archiving system and depends on various properties of the system: user structure, operating system, system load and others. In an experiment carried out in two institutions (one academic institution and one pharmaceutical firm), we monitored the number of files that changed daily on the main servers over a period of two months. The average percentages of files changed daily are presented in Table I. These percentages can be used as estimates of probability P_{new} when backup of the system is performed daily. We observe in this example that the probability that a file is new is quite low; however, other systems with a much higher probability P_{new} certainly exist.

4.1. RAM-Based System

In RAM-FHFEC, the average number of steps S_{RAM} needed to check a hash and, when necessary, to update the pointer array and hash remainder lists, is given by the

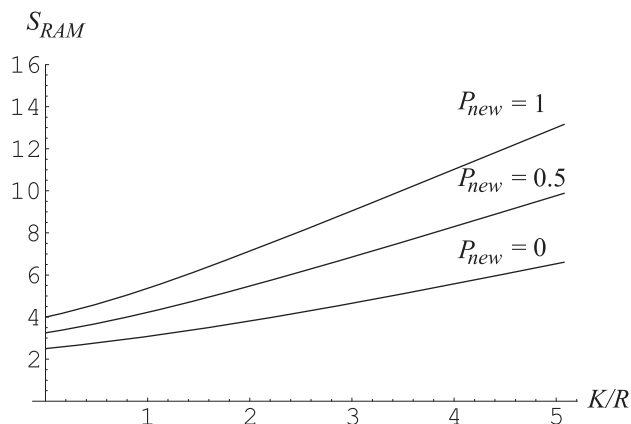


Fig. 5. The average number of steps in RAM-FHFEC for different values of probability P_{new} that a file is new. The algorithm is most efficient when the number of regions R is greater than the number of hashes K . However, even for $K/R = 5$ the number of steps is relatively small.

approximate equation (see Appendix B for details):

$$S_{RAM} \doteq (4 + 2K/R)P_{new} + \left(1 + \frac{K/R}{1 - e^{-K/R}}\right)(1 - P_{new}) \quad (3)$$

It is important to note that S_{RAM} does not depend directly on the number of stored hashes K , but rather on the average number of stored hashes per region K/R . Figure 5 shows the average number of steps in RAM-FHFEC as a function of K/R for different values of the a priori probability P_{new} . We can observe that the algorithm is most efficient when the number of regions is greater than the number of stored hashes ($K/R < 1$). Here, even in the worst case the answer is reached in less than five steps. However, as we shall see later, efficiency can be much higher than in BIS also for values $K/R \gg 1$.

We can also see that the algorithm is more efficient when the probability P_{new} that a file is new is low, because then, in most cases, there is no need for storing the hash. When the probability P_{new} is high, we find out very quickly (often in one step) that the file must be stored, but then we must store the hash and update the pointer array, which is the most time-consuming part of the process.

4.2. HD-Based System

Considering the fact that the time needed for movement of the disk head is many times larger than the time needed for reading the disk sector, in this approach the term "step" denotes the number of disk head movements. The upper bound on the average number of steps S_{HD} in HD-FHFEC is given by (see Appendix C for details):

$$S_{HD} \leq \left(1 + \frac{K/R (1 - e^{-K/R})^{-1} - 2}{2H_d}\right)(1 - P_{new}) + \left(1 + \frac{K/R}{H_d}\right)P_{new} \quad (4)$$

where H_d denotes the number of hashes that can be stored in one sector.

In Figure 6, this bound is presented as a function of the average number of hashes per region K/R for different a priori probabilities P_{new} . It is important to note that the average number of steps is much smaller than in RAM-FHFEC, and is close to one

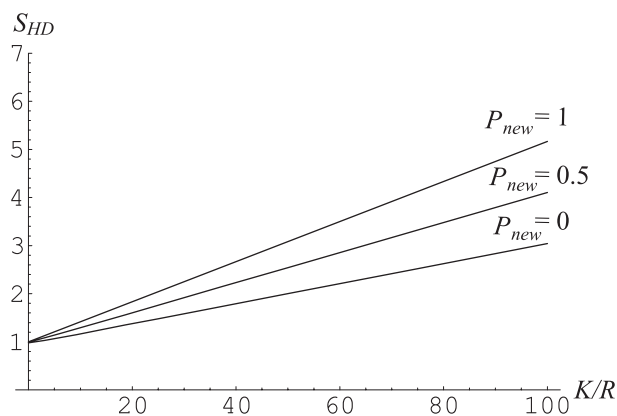


Fig. 6. Average number of steps in HD-FHFEC algorithm. The number of steps depends on the average number of hashes per region K/R and the a priori probability P_{new} that the requested hash does not exist in the system. The number of hashes that can be stored in one sector is taken to be $H_d = 24$.

for small K/R . This is due to the fact that all H_d hashes can be stored to and/or read from a sector in one step. On the other hand, if hashes are stored only on the disk, it is impossible to check the existence of a hash in less than one step. Thus, we can claim that, for $K/R \leq 1$, HD-FHFEC performs nearly optimally.

However, we should not forget that only disk head movements were counted, because disk head movement generally takes much more time than reading/writing all H_d hashes.

Since a single step for HD-FHFEC lasts many times longer than a single step for RAM-FHFEC, we see that in terms of the average time needed for hash existence checking, HD-FHFEC is still much slower than RAM-FHFEC. At present, RAM access times are in the order of nanoseconds, while hard disk access times are in the order of milliseconds, yielding a ratio of 10^6 .

4.3. Storage-Speed Trade-Off

In accordance with the Eq. (3), the number of steps S_{RAM} decreases along with R . However, the amount of RAM needed to store the pointer array increases with R . As is usually the case, we can improve efficiency if we increase RAM size.

When FHFEC is used, $R \lceil \log_2 C \rceil$ bits are needed for storing the pointer array, since $\lceil \log_2 C \rceil$ bits are needed for the addressing of C hashes. We also need Cm bits for storing the hash remainders, which are $m = n - r$ bits long, and $C \lceil \log_2 C \rceil$ bits for the pointers in the linked list of hash remainders. So the amount of required RAM for the FHFEC algorithm is:

$$M_{RAM} = (2^r + C) \lceil \log_2 C \rceil + C(n - r), \quad (5)$$

where C is the capacity of the system and n is the length of the hash.

We define RAM-FHFEC storage increase as the ratio $I_{RAM} = M_{RAM}/M_{HASH}$, where $M_{HASH} = nC$ is the size of RAM needed to store all hashes. For this example, I_{RAM} as a function of r is plotted in Figure 7. Note that I_{RAM} slowly decreases with r to a certain point, where it reaches a minimum. After that point, it starts to increase very rapidly. We can find the optimal value of r , that is, the value which yields minimal I_{RAM} , by solving the equation:

$$\frac{\partial I_{RAM}}{\partial r} = 0, \quad (6)$$

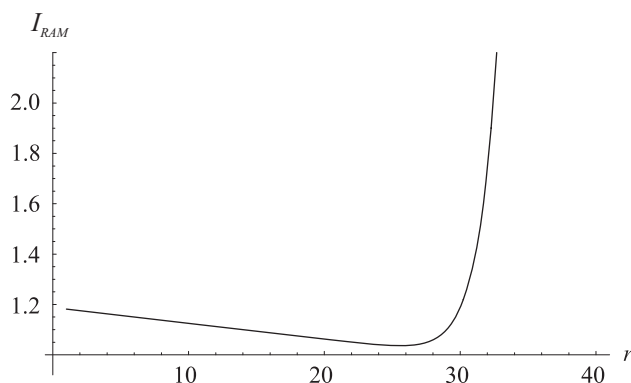


Fig. 7. RAM-FHFEC storage increase I_{RAM} as a function of r ; $C = 2^{30}$, $n = 160$. At first, I_{RAM} slowly decreases with r ; then, after a certain point, it starts to increase very rapidly. The optimal value of r lies between 25 and 26. Values of r below this point do not make sense for practical use, and values above this point represent a compromise between number of steps and storage increase.

which yields:

$$r = \log_2 \left(\frac{C}{\ln 2 \cdot \log_2 C} \right). \quad (7)$$

Note that the optimal value of r depends only on the capacity C and is independent of the hash length n . Decreasing r below its optimal value does not make sense, since the number of steps and the required storage would both increase.

Suppose that the hashes are $n = 160$ bits long, the capacity of the system is one billion ($C = 2^{30}$) hashes, and that we have assigned the first $r = 30$ bits of each hash to represent the region, which implies $R = 2^{30}$ and $K/R \leq 1$. As the optimal value for r , we obtain $r = 25.6$. Because r can have only integer values, we can choose the nearest greater integer value ($r = 26$), which yields $I_{RAM} = 1.037$. Increasing r beyond the optimal value represents a trade-off between required number of steps for finding/storing hash and required storage capacity.

Figure 8 shows performance measured as $1/S_{RAM}(r)$, for a fully loaded system, as a function of $I_{RAM}(r)$ for r above its optimal value. We observe that we can gain in performance at the cost of increasing RAM size above the minimal value.

5. TEST SYSTEM IMPLEMENTATION AND SIMULATION

To verify the analytical results from the previous section, we built the test implementation of our system and used the synthetic load (random generated hashes) to simulate its operation.

For this purpose, we have used modular simulation language, MODSIM III. It is a general-purpose, modular, block-structured language that provides support for object-oriented programming, discrete event simulation and animated graphics. It is intended for use in building process-based discrete event simulation models through modular and object-oriented development techniques. MODSIM's syntax and control mechanisms are similar to those of Modula-2, Pascal and Ada.

The FHFEC test system implementation includes modules for hash generation, as well as for insertion of hashes into bit structure, for hash statistics and probability calculations, for collection of results, and for writing results into standard format files. The system collects, calculates, and generates more than 30 different parameters, including, among others, the number of hashes in the system, the number of all incidents

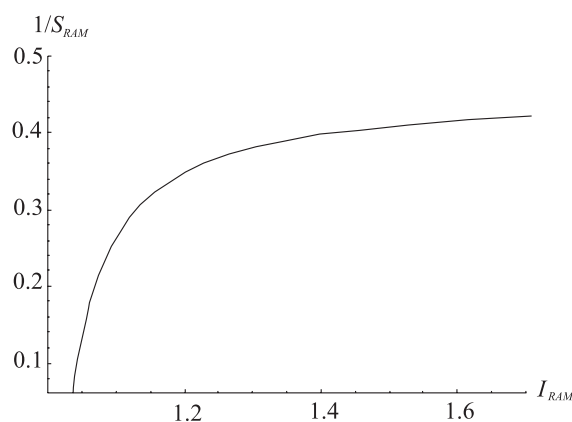


Fig. 8. Trade-off between required number of steps for finding/storing hash and required storage capacity. $C = 2^{30}$, $P_{new} = 0.1$, $K = 2^{30}$.

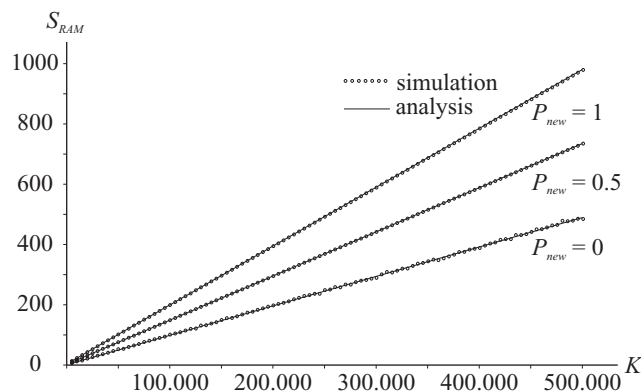


Fig. 9. Comparison between analytical and simulation results for the number of steps S_{RAM} for different values of P_{new} ($C = 500,000$, $r = 10$). The simulation was performed for 100 values of K from 5,000 to 500,000. $2 \cdot 10^4$ new hashes were tested for each K .

resulting in the answer NO in the second step, the probability that the hash already exists in the system, and the number of empty regions.

We simulated the hash checking and insertion process under the FHFEC algorithm from the empty system state at $K = 0$ until its full state at $K = C$ for 100 uniformly distributed points between 0 and C .

The results obtained analytically and by simulation are practically identical for all parameters of FHFEC in question. If plotted to the same diagram, the curves obtained by analytical results and curves obtained by simulations are indistinguishable when the number of tested hashes for each K exceeds 10^5 . By way of illustration, the number of steps for FHFEC algorithm ($C = 10^5$ and $r = 10$) obtained by analysis, plotted alongside simulation results, for different values of the a priori probability P_{new} , is shown in Figure 9. Low values of C and r are due to the limitations of the test system to be precise, to the limited RAM capacity of the computer on which the test system ran. To be able to distinguish simulation and analytical results in the same figure, we intentionally show the simulation results with weaker statistics ($2 \cdot 10^4$ hashes tested for each K).

From this, we can conclude that the simulations confirm analytical results.

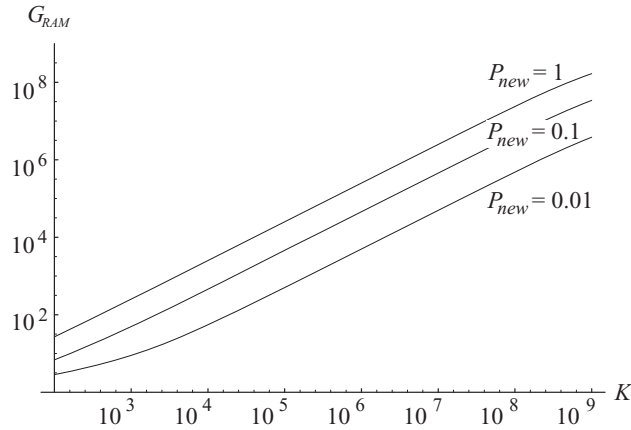


Fig. 10. Performance gain of RAM-based FHFEC with respect to BIS (capacity $C = 2^{30}$, number of regions $R = 2^{30}$). The gain increases with the number of hashes K already stored in the system, and is maximal when the system is full. Higher gains are also obtained for large values of probability P_{new} , that is, when many new files are archived and thus many new hashes must be stored.

6. COMPARISON TO OTHER ALGORITHMS

6.1. Comparison of FHFEC and BIS

Binary index search (BIS) represents a method for finding a requested item in a sorted data array [Knuth 1997]. It is known to be the most efficient non-hash-based method of searching and incremental sorting without storage overhead (there is no need for additional storage capacity beside that needed for storing the values).

Using BIS, the average number of steps needed for searching a hash and storing it, if it has not already been stored, is (see Appendix D for details):

$$S_{BIS} = P_{new}(\lceil \log_2(K+1) \rceil + K + 1) + (1 - P_{new})\left(\log_2 \frac{K}{2} + \frac{1}{K} \log_2 K - 1\right). \quad (8)$$

The average number of steps in BIS depends on the a priori probability P_{new} , and is much greater when the probability P_{new} is high and many new hashes are added to the sorted list. However, even for a low probability P_{new} , the average number of steps in BIS is much greater than in FHFEC.

The performance gain of RAM-FHFEC with respect to BIS is given as $G_{RAM} = S_{BIS}/S_{RAM}$. To compare both algorithms, we observe the same example as in Section 4.3; hashes are $n = 160$ bits long, the capacity of the system is one billion ($C = 2^{30}$) hashes, and $r = 30$ bits of each hash. This implies $K/R \leq 1$ and equality is reached only when the system is full. The RAM-FHFEC performance gain for this case is shown in Figure 10.

Performance gain increases with the number of hashes K already stored in the system. We can observe that for large K , BIS takes significantly more steps than FHFEC, especially for higher values of the a priori probability P_{new} . This is due to the hash-shifting operation needed to insert a hash into the sorted list in BIS.

A similar comparison made for HD-FHFEC would yield even more favorable results. It has been omitted, however, since in practice BIS is never implemented only on HD.

6.2. Speed Comparison to B+Tree

B+tree [Bayer and McCreight 1972] is a tree data structure that keeps data sorted in a way that allows for efficient insertion, retrieval and removal of records. It is widely

Table II. Comparison of FHFEC and B+Tree

Scenario	FHFEC time	B+tree time
A	2.82 s	41.2 s
B	33.4 s	417.0 s

Note: Time needed for searching/storing of $C = 2^{24}$ hashes for scenarios A and B were recorded.

used in file systems, for example, NTFS, ReiserFS, XFS, JFS2, and in databases for table indices.

Unfortunately, we are not aware of any exact analysis of average number of steps for retrieval and insertion of records. It is only known that searching or inserting a new record requires $O(\log K)$ steps and that the space required to store the tree is $O(K)$. On the other hand, FHFEC has complexity $O(1)$ when $K/R < 1$, and $O(\log K)$ in other cases (see 4.1 and 4.2). Hence, we could expect that FHFEC performs much better than B+tree for large K , although it is possible that for relatively small K , B+tree would perform better.

Therefore, we decided to write a simple FHFEC test application and compare its speed to a B+tree application built with the STX B+Tree C++ Template library [Bingmann 2010b], which is available under a GNU Lesser General Public License, and is already fairly optimized.

Both applications were written in C++, compiled with MingW included in the Dev C++ environment and run on a 2.2 GHz PC with 1GB RAM under Windows XP.

As test parameters, the following values were chosen:

- the capacity of the system $C = 2^{24}$,
- the length of hash $n = 32$ (limit imposed by the STX library),
- FHFEC: $r = 24$ and $m = 8$, which yields a good compromise between storage requirements and speed, and
- B+tree: number of node slots = 16, which yields, according to speed tests [Bingmann 2010a], the best results.

The MD5 hash function [RFC 1321 1992] was used for the generation of $C = 2^{24}$ 32-bit pseudo-random hashes, which yielded 16,743,938 new values and 33,278 repeated values. Changing the random seed of hash generation yielded very similar numbers: 16,744,687 new and 32,529 repeated. These C hashes were used for testing in two different scenarios.

- Scenario A.* Loading C different randomly generated hashes, which is similar to the creation of a new archive (no files were in the archive before).
- Scenario B.* This scenario is similar to daily usage of an archive system, when archiving is performed on a daily basis. Every day, all files are submitted to the archive; however, most of them are already archived. In this scenario, we were submitting hashes in groups of $G = 2^{15}$, so that in each session all hashes of a previous group were submitted first and then hashes of the new group added, until the capacity of the system was reached. So, in the first session, we submit G randomly generated hashes, in the second session, we submit G already archived hashes and G new randomly generated hashes, then in the third $2G$ already archived hashes and, again, G new randomly generated hashes, and so on. The total time required was recorded.

We measured the time needed for searching/storing hashes to the system for both scenarios. The results are presented in Table II.

Table III. Comparison of FHFEC and NTFS

Scenario	FHFEC time	NTFS time
A	16 s	866 s
B	18 s	10,99 s

Note: Time needed for searching/storing of $C = 2^{20}$ hashes hybrid FHFEC and NTFS for scenarios A and B were recorded.

As can be seen from the table, for a relatively small system ($C = 2^{24}$), FHFEC performed approximately fourteen times faster for Scenario A and twelve times faster for Scenario B than B+tree. For larger systems, this difference would be greater.

As we were not able to find any implementation of B+Tree that would work solely on HD, we were also unable to compare HD-FHFEC to HD based B+Tree.

The NTFS file system implements B+Tree for checking files stored on HD. Thus, we decided to compare the performance of FHFEC with the performance of NTFS. However, NTFS does not work only with HD. A part of the tree or the whole tree is cached in RAM. To make comparison fair, we compared NTFS with a hybrid implementation of FHFEC where hashes are stored on HD and cached in hash table in RAM. Updates are made to both RAM and HD while checking is performed only in RAM.

To test NTFS, we stored hashes as empty files with names equal to hexadecimal values of hashes. We performed all tests on a HD, which was empty at the beginning of the test. We used the same HD for all tests, that is, both scenarios for hybrid FHFEC and NTFS.

Due to slow performance of HD, we reduced capacity, that is, total number of stored hashes, to $C = 2^{20}$. All other parameters were the same as in the previous test. The results of the test are shown in Table III.

It is worth noting that the increase in time from scenario A to scenario B for hybrid FHFEC is very small, since the most time is used to store new hashes on HD and the number of new hashes is the same for both scenarios. This increase is substantial (more than ten times) for NTFS. We should also note, that two systems are not really equivalent, since FHFEC uses much more RAM than NTFS; however, this is not a problem in achieving system, where a whole server is dedicated for storing hashes.

7. CONCLUSION

The process of identifying a file by an SHA-1 hash of its contents is suitable for archival storage. In this article, we introduce a new fast hash-based file existence checking algorithm, which can be used in archival storage systems. Our proposed algorithm enables fast hash searching, as well as updating. We discuss two versions of the algorithm: one that stores the entire structure needed for FHFEC in random-access memory and one that uses a dedicated hard disk for this purpose.

The comparison to the most commonly used method in file systems and databases for table indices, B+tree, was carried out by measuring time needed for searching/storing hashes to the system for both FHFEC and B+tree. The results show that already for relatively small system FHFEC performs many times faster than B+tree. For larger systems, this difference is more significant.

This article also provides a statistical performance analysis of proposed algorithm. The performance analysis was carried out by calculating the average number of steps necessary to find the final answer regarding hash existence. Comparing to binary index search, FHFEC performs many orders of magnitude faster at the cost of a small increase in hash storage capacity. Moreover, with minor changes, the performance analysis can be applied to database hash based indexing.

So, for systems in which decreasing the time needed for checking the existence of a file is crucial, FHFEC should prove useful. Extensive simulations fully confirmed the analytical results.

APPENDIXES

A. PROBABILITY OF COLLISION

One of the basic features of a good hash function is that it produces uniformly distributed hashes. Consider a system with hashes n bits long, that is, their domain has the size of $N = 2^n$ values, and let K be the number of hashes in the system. Assume that we then add hashes to the system one by one. When the i th hash is added, the probability P_i that collision does not occur at this step is equal to the probability that this hash is different from all $i - 1$ hashes already in the system. Assuming uniform distribution of hashes, this probability is given by:

$$P_i = \frac{N - (i - 1)}{N} = 1 - \frac{i - 1}{N}. \quad (9)$$

The probability that collision does not occur in the process of adding K hashes is then:

$$P_{nocoll} = \prod_{i=1}^K P_i = \prod_{i=1}^K \left(1 - \frac{i - 1}{N}\right). \quad (10)$$

For $N \gg K$, by taking only the first two members of the expansion of this product, the following approximation can be done:

$$P_{nocoll} \doteq 1 - \frac{\sum_{j=1}^{K-1} j}{N} = 1 - \frac{K(K - 1)}{2N}. \quad (11)$$

Finally, the probability of collision is:

$$P_{coll} = 1 - P_{nocoll} \doteq \frac{K(K - 1)}{2N} = \frac{K(K - 1)}{2 \cdot 2^n}. \quad (12)$$

B. AVERAGE NUMBER OF STEPS IN RAM-FHFEC

The number of steps in RAM-FHFEC refers to the number of RAM read/write accesses. The number of steps depends on the type of the answer we get. If we get the answer NO at the beginning, because the corresponding region is empty, the pointer is read in one step and then the hash remainder is stored in three steps: one for updating the original pointer, one for storing the hash remainder, and one for setting the attached pointer to null. Thus, the total number of steps is:

$$s_{no} = 1 + 3 = 4. \quad (13)$$

When the first answer is POSSIBLE YES and the final answer is NO, the corresponding pointer from the pointer array is read, and then the first stored hash remainder is read along with its pointer to the next hash remainder, and so on until the null pointer is reached. Then the new hash remainder is inserted, which requires, as in the previous case, three more steps. So, in this case, it holds:

$$s_{pyes|no} = 1 + 2i + 3 = 4 + 2i, \quad (14)$$

where i is the number of hashes already stored in the region. And, finally, when the answer is POSSIBLE YES and the final answer is YES, hash remainders and pointers

are read only to the point where the match is found. When i hashes are in the region, the average number of readings is:

$$s_{pyes|yes} = 1 + \frac{i}{2} + \frac{i}{2} = 1 + i, \quad (15)$$

where $1 + i/2$ steps correspond to readings of the pointers, and $i/2$ steps correspond to readings of the hash remainders. In this case, there is no need to insert the hash remainder.

The number of steps in (14) and (15) depends on the number of hashes i in the region of the new hash. The average number of steps depends on the average number of i , that is, the average number of hashes in occupied regions, which we denote H_a . From (13), (14), and (15), the average number of steps for different cases is thus:

$$\begin{aligned} S_{no|no} &= 4 \\ S_{pyes|no} &= 4 + 2H_a \\ S_{pyes|yes} &= 1 + H_a. \end{aligned} \quad (16)$$

The overall average number of steps for RAM-FHFEC is:

$$\begin{aligned} S_{RAM} &= (S_{no|no}P_{no|no} + S_{pyes|no}P_{pyes|no})P_{new} \\ &+ S_{pyes|yes}P_{pyes|yes}(1 - P_{new}), \end{aligned} \quad (17)$$

where P_{new} is the a priori probability that a new hash does not already exist in the system, that is, the file is new. P_{new} is a parameter of the system. The a posteriori probability $P_{pyes|yes}$ is the probability that we get the answer POSSIBLE YES when the hash exists, and it is obvious that:

$$P_{pyes|yes} = 1. \quad (18)$$

$P_{pyes|no}$ is the probability that the answer is POSSIBLE YES when the requested hash does not exist, and $P_{no|no}$ is the probability that we get the answer NO at the beginning when the hash does not exist. So it holds:

$$P_{no|no} = 1 - P_{pyes|no}. \quad (19)$$

Substituting (16), (18), and (19) into (17) now yields:

$$\begin{aligned} S_{RAM} &= (4(1 - P_{pyes|no}) + (4 + 2H_a)P_{pyes|no})P_{new} \\ &+ (1 + H_a)(1 - P_{new}). \end{aligned} \quad (20)$$

To arrive at the final result, we still have to determine $P_{pyes|no}$ and H_a .

B.1. Probability $P_{pyes|no}$

In the FHFEC system, we have R regions to be considered. Suppose that i hashes are already stored in the system. If we choose an empty region, the probability that the next hash does not fall into that region is:

$$P_i = \frac{MR - M - i}{MR - i}, \quad (21)$$

where M is the number of hashes in each region. The probability that a chosen region remains empty after adding K hashes one by one is then:

$$P_{re} = \prod_{i=0}^{K-1} P_i = \prod_{i=0}^{K-1} \frac{MR - M - i}{MR - i}. \quad (22)$$

Suppose now that K hashes are already in the system at the point when we want to check if a randomly chosen hash exists. The a priori probability that it already exists in the system is:

$$P_{yes} = \frac{K}{MR} \quad (23)$$

and the probability that it does not is:

$$P_{no} = 1 - P_{yes} = \frac{MR - K}{MR}. \quad (24)$$

The probability that we get the answer NO at the beginning is equal to the probability that the chosen region (the region to which the hash belongs) is empty, that is, equal to P_{re} . The probability of getting the answer POSSIBLE YES is then:

$$P_{pyes} = 1 - P_{re} = 1 - \prod_{i=0}^{K-1} \frac{MR - M - i}{MR - i}. \quad (25)$$

In accordance with the Total Probability Theorem [Papoulis 1984], we have:

$$P_{pyes} = P_{pyes|yes}P_{yes} + P_{pyes|no}P_{no} \quad (26)$$

and

$$P_{pyes|no} = \frac{P_{pyes} - P_{pyes|yes}P_{yes}}{P_{no}}. \quad (27)$$

Finally, substituting (18), (23), (24), and (26) into (27), after rearrangement yields:

$$P_{pyes|no} = 1 - \frac{MR}{MR - K} \prod_{i=0}^{K-1} \frac{MR - M - i}{MR - i}. \quad (28)$$

B.2. The Average Number of Hashes in Occupied Regions H_a

A region is occupied if there is at least one hash stored in it. The average number of occupied regions is:

$$R_{occ} = P_{pyes}R \quad (29)$$

because P_{pyes} is actually the probability that a region is occupied. The average number of hashes in occupied regions is equal to the number of all stored hashes divided by the number of occupied regions; thus:

$$\begin{aligned} H_a &= \frac{K}{R_{occ}} = \frac{K}{P_{pyes}R} \\ &= \frac{K}{R(1 - \prod_{i=0}^{K-1} \frac{MR - M - i}{MR - i})}. \end{aligned} \quad (30)$$

B.3. Approximation

The product that appears in (28) and (30) is computationally demanding and non-intuitive, especially for large K . We can see that the product already approaches a limit value at a relatively small M , for constant K and R (see Figure 11). Let Π be the

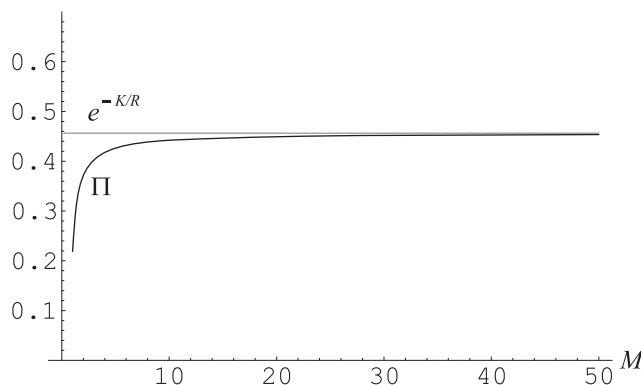


Fig. 11. Approximation of product Π ($R = 2^7$, $K = 100$). It can be seen that Π depends weakly on M even for relatively small M , and that $e^{-K/R}$ is a good approximation, even for relatively small R .

product and Γ its limit, when $M \rightarrow \infty$. We have:

$$\begin{aligned} \Gamma &= \lim_{M \rightarrow \infty} \Pi \\ &= \lim_{M \rightarrow \infty} \prod_{i=0}^{K-1} \frac{MR - M - i}{MR - i} \\ &= \left(\frac{R-1}{R} \right)^K. \end{aligned} \quad (31)$$

Since, in all practical cases, M is very large, this equation is a very good approximation. Furthermore, when R is large enough (which is also true for all practical purposes), Γ can be approximated with:

$$\Gamma = \left(\frac{R-1}{R} \right)^K = \left(\left(1 - \frac{1}{R} \right)^{-R} \right)^{-K/R} \doteq e^{-K/R}. \quad (32)$$

Figure 11 shows the dependency of the product Π on the parameter M , along with the approximation for Γ in (32). We can see that curve Π tends toward $e^{-K/R}$ very rapidly, even for M and R much smaller than in all practical cases of interest.

For large M and R and for $K \ll RM$, the probability $P_{pyes|no}$ in (28) can now be approximated as:

$$P_{pyes|no} \doteq 1 - e^{-K/R}. \quad (33)$$

and the average number of hashes in occupied regions H_a in (30) as:

$$H_a \doteq \frac{K/R}{1 - e^{-K/R}}. \quad (34)$$

Substituting now (33) and (34) into (20) yields, after rearrangement:

$$\begin{aligned} S_{RAM} &\doteq (4 + 2K/R) P_{new} \\ &\quad + \left(1 + \frac{K/R}{1 - e^{-K/R}} \right) (1 - P_{new}) \end{aligned} \quad (35)$$

C. AVERAGE NUMBER OF STEPS IN HD-FHFEC

In HD-FHFEC, steps are counted as the number of disk head movements. The average number of steps for HD-FHFEC is obtained in a similar fashion to RAM-FHFEC in

(17), (18), and (19):

$$S_{HD} = (S_{no|no}(1 - P_{pyes|no}) + S_{pyes|no}P_{pyes|no})P_{new} + S_{pyes|yes}(1 - P_{new}) \quad (36)$$

When the answer is NO at the beginning, the time spent to get the answer is actually the time needed for positioning the disk head to the track that holds the corresponding sector. The time for inserting a new hash is negligible, since the disk head is already positioned on the corresponding sector. Thus:

$$S_{no|no} = 1 \quad (37)$$

When the answer at the beginning is POSSIBLE YES and the final answer is NO, the disk head must be moved to the sector that is addressed with the first r bits of the hash, which requires one step. After that, we need to go through all additional sectors, if there are any, one step for each sector (although, of course, this can be minimized if we are able to select subsequent sectors that are contiguous). Let H_d be the number of hash remainders that can be stored in one sector. If the number of hash remainders that are already stored in the region is equal to or greater than H_d , we have to move the head to the next sector, and so on. When equality holds, we have to store the new hash remainder in a new sector, which adds an additional step. For region i that has H_i stored hashes, we thus need

$$s_i = 1 + \left\lfloor \frac{H_i}{H_d} \right\rfloor \quad (38)$$

steps. Because hashes are uniformly distributed, the average number of steps in this case is:

$$S_{pyes|no} = 1 + \frac{1}{R_{occ}} \sum_{i=1}^{R_{occ}} \left\lfloor \frac{H_i}{H_d} \right\rfloor \leq 1 + \frac{H_a}{H_d}, \quad (39)$$

where R_{occ} is the number of occupied regions. The right-hand side of this inequality represents the upper bound to average number of steps. The inequality is due to the floor operation in the sum, so the bound is quite tight. In fact, subtracting 1 from this upper bound would yield the lower bound, although this is of lesser interest for our purposes.

When the final answer is YES, on average we will need to check half the hashes in the region. The same reasoning as shown, further taking into account that there is no need for storing a hash remainder in a new sector, yields the necessary number of steps:

$$S_{pyes|yes} \leq 1 + \frac{H_a/2 - 1}{H_d}. \quad (40)$$

From (33), (36), (37), (39), and (40), we can now write the upper limit for the average number of steps in HD-FHFEC as:

$$S_{HD} \leq \left(1 + \frac{K/R(1 - e^{-K/R})^{-1} - 2}{2H_d}\right)(1 - P_{new}) + \left(1 + \frac{K/R}{H_d}\right)P_{new} \quad (41)$$

D. PERFORMANCE OF BIS

BIS operates on an array of sorted hashes. It checks the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner.

When the answer is NO, the hash is not found in the index table, which implies that the procedure must be repeated until only one hash remains after bisection. This requires S_r readings from RAM:

$$S_r = \lceil \log_2(K + 1) \rceil, \quad (42)$$

where K is the number of hashes stored in the system. However, the new hash must be stored in the index table. Storing a new hash in the sorted array of hashes implies, on average, shifting half of the hashes. Hash shifting takes one read and one write. An additional step is needed to store the new hash. Thus, the average number of steps S_s needed for shifting and storing a new hash is:

$$S_s = 2 \frac{K}{2} + 1 = K + 1. \quad (43)$$

The total number of steps when the answer is NO is then:

$$S_{no} = S_r + S_s = \lceil \log_2(K + 1) \rceil + K + 1. \quad (44)$$

When the answer is YES fewer readings are needed for searching, because the search stops after the hash is found. On average, we need

$$S_{yes} = \log_2 \frac{K}{2} + \frac{1}{K} \log_2 K - 1 \quad (45)$$

steps to find out that the hash is already in the system [Corwin 2010]. For large values of K , this result can be approximated by $\log_2(K/2)$. In this case, there is no need to store the hash.

The average number of steps needed for searching a hash and storing it, when it is not already stored, is then:

$$S_{BIS} = P_{new}(\lceil \log_2(K + 1) \rceil + K + 1) + (1 - P_{new}) \left(\log_2 \frac{K}{2} + \frac{1}{K} \log_2 K - 1 \right). \quad (46)$$

REFERENCES

- BAYER, R. AND MCCREIGHT, E. M. 1972. Organization and maintenance of large ordered indices. *Acta Informatica* 1, 173–189.
- BINGMANN, T. 2010a. Speed test results. <http://idlebox.net/2007/stx-btree/stx-btree-0.8-doxygen/speedtest.html>.
- BINGMANN, T. 2010b. Stx b+ tree c++ template classes. <http://idlebox.net/2007/stx-btree/>.
- BOHN, R., ET AL. 2008. How much information? At the global information industry center. <http://hmi.ucsd.edu/howmuchinfo.php>.
- BRODER, A. Z. 1993. Some Applications of Rabin's Fingerprinting Method, Sequences II: In *Methods in Communications, Security and Computer Science*, Springer-Verlag.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd Ed. MIT Press and McGraw-Hill.
- CORWIN, E. M. 2010. Average case of binary search. <http://www.mcs.sdsmt.edu/ecorwin/cs251/binavg-/binavg.htm>.
- COX, L. P., MURRAY, C. D., AND NOBLE, B. D. 2002. Pastiche: Making backup cheap and easy. *ACM SIGOPS Oper. Syst. Rev.* 36, 285–298.
- FIPS 180-2 2002. Secure hash standard. National Institute of Standards and Technology.
- IBM 2010. Grouping hash implementation. <http://publib.boulder.ibm.com/infocenter/series/v5r3/index.jsp?topic=rzajq/groupopt.htm>.
- JOVANOVIĆ, E., MILUTINOVIC, V., AND HURSON, A. R. 2002. Acceleration of nonnumeric operations using hardware support for the ordered table hashing algorithms. *IEEE Trans. Comput.* 51, 9.
- KNUTH, D. 1997. *The Art of Computer Programming*, Vol. 3: Sorting and Searching, 3rd Ed. Addison-Wesley.

Fast File Existence Checking in Archiving Systems

2:21

- KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. 2004. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Technical Conference*.
- LYMAN, P., VARIAN, H. R., SWEARINGEN, K., CHANLES, P., GOOD, N., JORVAN, L. L., AND PAL, J. 2003. How much information? 2003. <http://www2.sims.berkeley.edu/research/projects/how-much-info-2003>.
- MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. 2001. A low-bandwidth network file system. In *Proceedings of the Symposium on Operating Systems Principles*.
- PAPOULIS, A. 1984. *Probability, Random Variables and Stochastic Processes*, 2nd Ed. McGraw-Hill.
- PARLANTE, N. 2001. *Linked List Basics*. Stanford University.
- PCGuide 2010. Logical block addressing (LBA). <http://www.pcguides.com/ref/hdd/bios/modesLBA-c.html>.
- POLICRONIADES, C. AND PRATT, I. 2004. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Conference*.
- QUINLAN, S., AND DORWARD, S.. 2002. Venti: A new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*.
- RFC 1321 1992. The MD5 message-digest algorithm. IETF.
- RUDAN, S., KOVACEVIC, A. Z., BABOVIC, D. J., MILLIGAN, C., AND MILUTINOVIC, V. 2006. One approach to efficient management of zillion signatures. *PSI Trans. Internet Res.* 2, 2, 17–21.

Received December 2009; revised April 2010; accepted July 2010