Designing a Bloom Filter for Differential File Access

Lee L. Gremillion
Indiana University

SUMMARY: The use of a differential file for a database update can yield integrity and performance benefits, but it can also present problems in providing current data to subsequent accessing transactions. A mechanism known as a Bloom filter can solve these problems by preventing most unnecessary searches of the differential file. Here, the design process for a Bloom filter for an on-line student database is described, and it is shown that a very effective filter can be constructed with a modest expenditure of system resources.

1. Introduction

Allowing multiple users to interactively access and change information in a database is one of the more useful capabilities of modern computer-based systems. This is the basis for such applications as on-line reservation systems and automated bank tellers. Although an on-line database update is a vital feature of these systems, it does involve a qualitative jump in the level of complexity of system design and operations. In particular, an on-line update makes the procedures required to ensure database accuracy, integrity, and recoverability much more difficult.

A process known as *differential file updating* has been suggested as a method of easing these difficulties (Rapaport [3], Severance and Lohman [4], and others). The term "differential file" in conjunction with on-line update refers to a separate data file in which all changes to the database are stored prior to their batch mode application at periodic intervals. Rather than making changes directly to the database, on-line update transactions cause additions to the differential file, while the database proper remains unchanged. Then, at intervals (usually each night after the on-line system has been brought down) a batch processing program applies all the updates in the differential file. When the on-line system comes up again, the database is completely current, and the differential file is empty.

As Severance and Lohman point out, this technique can result in a number of benefits, including performance improvements, greater database reliability, and reduced backup and recovery costs. There does seem to be a negative consequence of using a differential file update, however. Once an update is made to a particular record, the data in the main database for that record is no longer up-to-date. To provide current information in response to subsequent requests against that record, one must obtain whatever part of it is in the differential file, in addition to the main database entry. Since there is no a priori way of knowing whether a record might have already been updated when it is requested, one must search the differential file every time a record is requested. The alternative is to search only the main database upon retrievals, giving users information which is sometimes out-of-date. This appears to be a serious drawback of differential file techniques, particularly if one is in a position in which (1) searching the differential file for every transaction would degrade system performance, but (2) all transactions must be given access to the most current information. However, a solution does exist.

Ideally, when using a differential file, we would like to be able to

CR Categories and Subject Descriptors: H.2.2 [Database Management]: Physical Design–*access methods*; J.1 [Computer Applications]: Administrative Data Processing–*education*.
General Terms: Design, Management, Performance.
Additional Key Words and Phrases: differential file, Bloom filter, database design.
Author's present address: L. L. Gremillion, Operations and Systems Management Department, School of Business, Indiana University, Bloomington, IN 47405.

access that file *only* when we *know* that the record we are seeking is, in fact, in the file. Severance and Lohman suggest that we can approach this ideal through the use of a "Bloom Filter," a technique (first described by Bloom [1]) which uses hash coding to test a series of messages one-by-one for membership in a given set. The technique exploits the fact that a small percentage of erroneous classifications can be tolerated.

As applied to differential file access, the Bloom filter works like this: a set of "switches" (probably bits in main memory) are allocated as an array. All switches are initially "off" (binary 0) before any records are put in the differential file. When a record is put into the differential file, its primary key is transformed through some number n hashing algorithms to switch numbers, and each switch so addressed is turned "on" (binary 1). Anytime a transaction seeks to access a database record, the requested key is transformed through these same n hashing algorithms. If any one of the switches to which this key transforms is off, then we can say with certainty that there is no record with that key value in the differential file. If all the switches for a key value are on, we must look for the record in the differential file, but we cannot be certain that it will be there. It could be that the switches were turned on by synonymous transformations from other key values. If such is the case and we access the differential file in vain, we say that a "filtering error" has occurred.

To use the Bloom filtering technique, one must determine how much memory to allocate to switches and how many and which hashing transformations to use in order to achieve an acceptably low filtering error rate, given the transaction volume, the number of key values accessed and updated, and the characteristics of the keyset. Here, we describe how this was accomplished for a particular operational system and offer some generalizations that can be drawn from our experience.

2. The System

The student database at Indiana University consists of several hundred thousand entries, each representing a current or prior IU student. A large amount of information is stored for each individual so that the entire database occupies several 200-megabyte disk packs. The DBMS is Intel's SYSTEM 2000, running on the university's administrative computer, currently an IBM System/370 model 168-III (8 megabytes), under the MVS operating system.

Many administrative offices, including those of the registrar, bursar, and admissions, throughout the university system have access to the database via a network driven by CICS/VS, an IBM teleprocessing monitor. The network is comprised of several hundred terminals. University officials use the network to retrieve student information and update student records as an integral part of everyday operations. The system runs during normal working hours and logs from 30,000 to 60,000 transactions per day.

When the database was first put on-line several years ago, update transactions were immediately applied to the database records. This soon led to performance problems. Since the DBMS (at least at that time) locked out at the database level during update, all other users were denied access whenever an update transaction occurred. In addition, a direct on-line update often necessitated the revision of the database's secondary indexes, introducing yet more overhead. Backup also became a consideration as the database grew in size to the point where restoring it took several hours.

As a result of all these complications, a differential file approach was adopted. During the course of a day, any database updates were captured in a VSAM key sequenced dataset known as the "indirect update file." (The primary key for both the student database records and the indirect update records is the student's social security number.) This file was empty at the beginning of the day and accumulated records, one for each student whose record was updated, while CICS was up. The records were of variable length, their exact length dependent on the extent to which a student's record had been changed. Every night, after CICS came down, the changes in the indirect update file were applied to the main database, which was then copied to backup.

A compromise was made between searching the differential file for every transaction and giving users noncurrent information. For transactions classified "retrieval only" (that is, they could not cause an update), only the main database was searched. For transaction types which could cause an update, however, the differential file was always searched and its information, if any, merged with that from the main database before it was presented to the user. This was the mode of operation when the use of a Bloom filter was first considered.

A Bloom filter in this case could have two effects. First, it would allow retrieval-only type transactions to have access to current student information. Second, for update transactions, it could eliminate differential file searching in those situations in which the particular record had not been previously updated. If a low enough filtering error rate could be achieved, this technique would allow all transactions access to current information while reducing the total number of differential file searches.

Severance and Lohman developed an algorithm for predicting a filtering error rate given some X transformations which map uniformly and independently over M filter switches, assuming that database update transactions are independent, uniformly distributed over all records, and arrive over time at a fixed rate. In fact, none of these assumptions holds true for our student database. Neither access nor updates are uniformly distributed over the database. Activity on student records tends to be "clumped"—that is, if a particular student's record is accessed today at all, it is probably

accessed several times. Arrival rates vary widely according to the time of day, day of the week or month, and special circumstances (e.g., registration). Finally, there was no compelling reason for choosing a particular filter size or number of transformations, nor was there any certainty about exactly what the "target" filtering error rate should be.

Table I. Filtering Error Rate by Filter Size and Number of Transformations, Given 100,000 Transactions, 30,000 Unique Key Values Accessed, and 7,000 Unique Key Values Updated.

	Number of transformations		
Filter size	4	6	8
24,576	.356	.431	.516
28,672	.273	.333	.407
32,768	.210	.252	.322

Accordingly, it seemed that the best way to start investigating the use of the Bloom technique was to simulate using the filter and observe the effects of varying the filter size, transformations, and transaction characteristics.

3. The Simulations

The first model simulated one heavy day's worth of database transactions, with both the filter size and number of hashing transformations varied in order for us to see the relative effects of these two factors. Figure 1 illustrates the general program logic. During each run, 50,000 transactions were generated, which accessed a total of 10,000 unique key values. Of these, 2,300 unique key values were updated (and therefore put in the simulated differential file). These characteristics of the system were determined by examining system records and interviewing users.

A random sample of actual social security numbers was taken from the student database and used as the keyset for the simulation. Division by prime numbers (several), shifting, folding, and the mid-square technique (all described by Martin [2]) were used for transformations. In total, 64 simulation runs were made, with filter sizes ranging from 4,096 to 32,768 switches (.5K to 4K bytes of memory) and from one to eight transformations.

Figure 2 presents the results of the simulation runs in graphic form. The Y-axis, the filtering error rate, is defined as the number of filtering errors divided by the total number of differential file accesses. This rate drops drastically as the size of the filter increases. It is also affected, but less dramatically, by the number of transformations employed. (Curves for two, four, six and seven transfor-

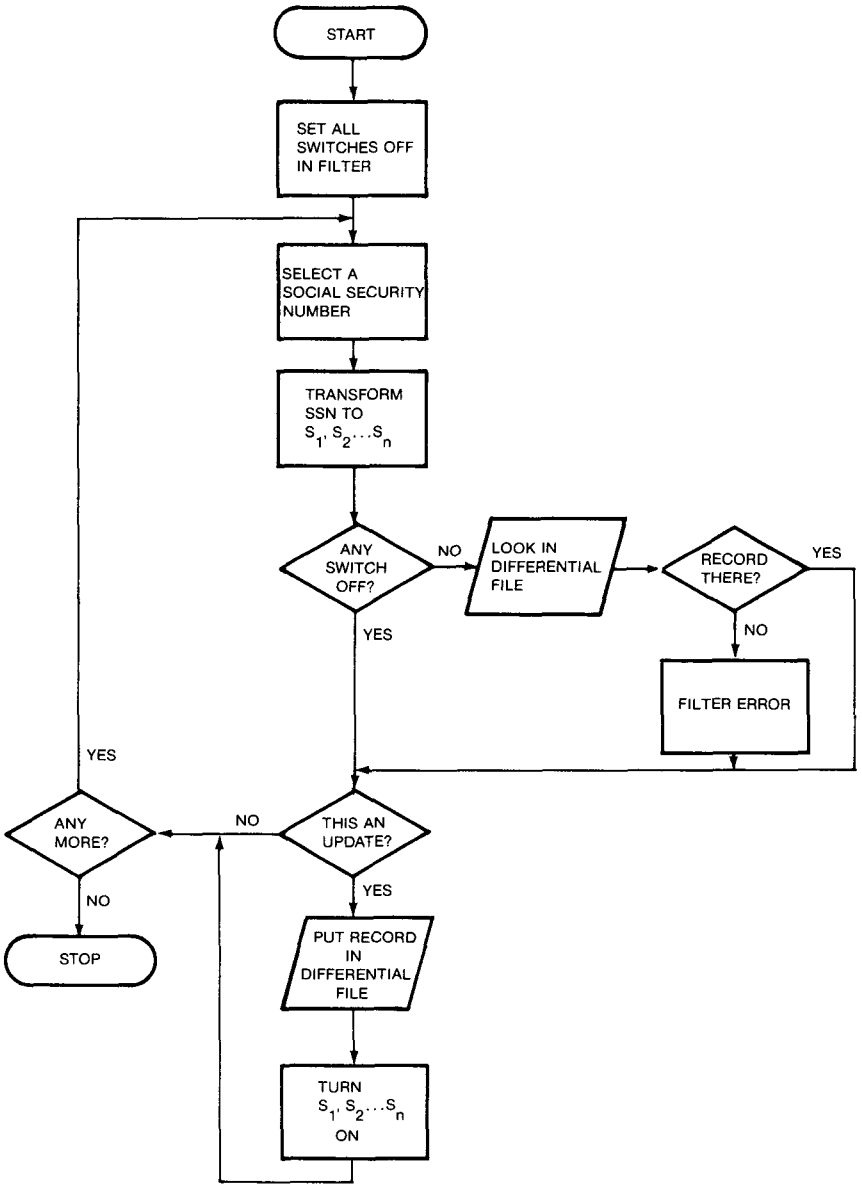

Fig. 1. Simulation Program Logic.

Table II. Filtering Error Rate by Filter Size and Number of Transformations, Given 100,000 Transactions, 30,000 Unique Key Values Accessed, and 7,000 Unique Key Values Updated.

	Number of transformations		
Filter size	4	6	8
40,960	.125	.146	.197
49,152	.078	.082	.107
57,344	.056	.055	.063
65,536	.035	.029	.044

mations follow the same pattern as those shown.)

These results clearly indicate that for most days' activity levels, a filter employing a modest amount of memory (3K-4K bytes) and three or more transformations would be quite effective (a filter error rate ≤ 2 percent). The next step was to see how performance would be affected by events such as registration during which activity levels are very high. To do this, a simulation representing an extremely heavy day's load was run: 100,000 transactions, 30,000 unique key values accessed, and 7,000 unique key values updated. Table I shows the results using filter sizes of 24K, 28K, and 32K switches (3K, 3.5K, and 4K bytes), and four, six, and eight transformations.

As indicated in Table I, extreme activity levels would result in a significantly higher filtering error rate. This rate can be reduced by increasing the filter size. Table II shows the results of using 5K, 6K, 7K, and 8K bytes of main memory for the filter in this same situation.

All of these results suggest that the filter size might be a variable parameter, set at the beginning of each day when the on-line system is brought up. On days when activity is expected to be unusually high, a larger amount of memory may be allocated. This would be relatively easy to implement, and, in fact, the simulation program was written to work this way.

4. Observations and Conclusions

The results presented in Figure 2 and Tables I and II demonstrate that the effectiveness of the Bloom filter is most strongly influenced by what we might call its "loading factor." In other words, for a given number of unique key values which will be transformed into filter switch numbers, the greater the range of switch numbers, the lower the filtering error rate will be. This is exactly parallel to saying that for a hashed direct access file of a given size, the larger the disk space it can be given, the fewer synonyms, and therefore overflow problems, there will be.

It is possible that filter performance could be further improved by the judicious selection of transformation methods. It was beyond the scope of our project to exhaustively examine issues such as, for example, which of the 70 combinations of four of the eight transformation algorithms was the best (i.e., which gave the most uniform, independent distribution). Tuning in this area, by testing various combinations of transformations on actual key val-

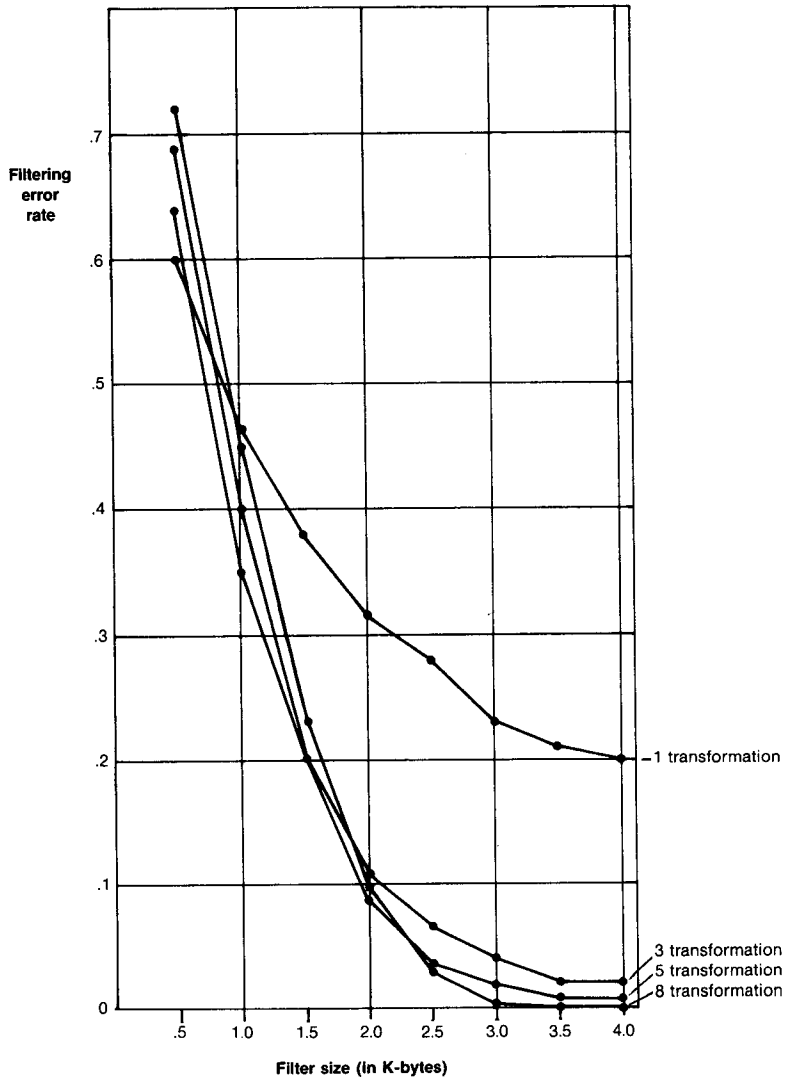

Fig. 2. Simulation Results: The Filtering Error Rate as a Function of Filter Size and the Number of Transformations.

ues, could result in lower error rates than those offered here for a given filter size.

One pragmatic consideration affects the choice of filter size. To avoid page faults when the filter is addressed, additional real storage will have to be fixed in the CICS region. Since the units of storage-fixing under MVS are pages of 4K bytes each, the most likely alternative filter sizes would be 32K (4K bytes) or 64K (8K bytes) switches. (In either case, the code for the filter program could be fit into a few hundred bytes, leaving most of the space for the filter itself.)

Another implementation consideration would be recovery from system failures which cause the loss of the switch array. In such a case, we would want to reconstruct the filter as part of the system restart procedure. The simple way to do this is to read the differential file sequentially, transform each key, and turn on the appropriate switches. If we assume 100 milliseconds per record access (probably a conservative estimate for this hardware configuration), then filter reconstruction would take between three and four minutes for a 2,000-record differential file. If this time was considered too long, then one might design the system so that it maintained a second file which contained only the keys of records stored to the differential file. This second file, which would have a large blocking factor and therefore enable the reading of many keys per physical record access, could be used to more quickly reconstruct the filter.

The Administrative Computing Department at IU is currently considering using this technique to filter access to the student database differential file. Even this preliminary analysis, however, indicates that Bloom filtering is another example of the classic storage/performance tradeoff; that is, the achievement of processing performance improvements by the increased expenditure of storage. Other examples include allocating main memory to store direct access file indexes and increasing the real storage support of a virtual memory system. In this case, we are expending main memory for the filter (plus a modest number of additional program instructions) in exchange for avoiding unnecessary secondary storage accesses. If our goal is to provide all database transactions current information, then the Bloom filter technique reduces the overall processing time required to accomplish this. In a technological environment in which main memory costs continue to drop dramatically, it becomes increasingly attractive to expend main memory on a large enough filter to cut the error rate to practically nothing.

References

1. Bloom, B.H. Space/time tradeoffs in hash coding with allowable errors. *Comm. ACM 13*, 7 (July 1970), 422–426. How hash coding methods which involve a small but tolerable error rate can be used to reduce the amount of space required for hash coded information.

2. Martin, J. *Computer Data Base Organization.* Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 382–385. A comprehensive textbook on the physical and logical design of databases.

3. Rappaport, R.L. File structure design to facilitate on-line instantaneous updating. Proc. 1975 ACM SIGMOD Conf., San Jose, Calif., May 1975, pp. 1–14. Describes a system that uses a type of differential file to facilitate database recovery after power failures.

4. Severance, D.G., and Lohman, G.M. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Systs. 11*, 3 (Sept. 1976), 257–267. Discusses the uses and advantages of differential files for database modifications.