# Buffered Bloom Filters on Solid State Storage

Mustafa Canim
University of Texas at Dallas
Richardson, TX, USA
canim@utdallas.edu

George A. Mihaila
IBM Watson Research Center
Hawthorne, NY, USA
mihaila@us.ibm.com

Bishwaranjan Bhattacharjee
IBM Watson Research Center
Hawthorne, NY, USA
bhatta@us.ibm.com

Christian A. Lang
IBM Watson Research Center
Hawthorne, NY, USA
langc@us.ibm.com

Kenneth A. Ross
Columbia University
New York, NY, USA
kar@cs.columbia.edu

## ABSTRACT

Bloom Filters are widely used in many applications including database management systems. With a certain allowable error rate, this data structure provides an efficient solution for membership queries. The error rate is inversely proportional to the size of the Bloom filter. Currently, Bloom filters are stored in main memory because the low locality of operations makes them impractical on secondary storage. In multi-user database management systems, where there is a high contention for the shared memory heap, the limited memory available for allocating a Bloom filter may cause a high rate of false positives. In this paper we are proposing a technique to reduce the memory requirement for Bloom filters with the help of solid state storage devices (SSD). By using a limited memory space for buffering the read/write requests, we can afford a larger SSD space for the actual Bloom filter bit vector. In our experiments we show that with significantly less memory requirement and fewer hash functions the proposed technique reduces the false positive rate effectively. In addition, the proposed data structure runs faster than the traditional Bloom filters by grouping the inserted records with respect to their locality on the filter.

## 1. INTRODUCTION

The Bloom filter is a probabilistic data structure optimized for very efficient membership queries [4]. It provides a very compact representation of sets. This efficiency comes at the price of reduced precision: specifically, the Bloom filter may incorrectly indicate that an element is present in the original set, thus occasionally generating false positives (but no false negatives). The false positive rate is directly proportional to the compression ratio. Thus, in order to lower the false positive rate, a larger amount of memory is required.

Bloom filters are used in a wide variety of application areas, such as databases [1], distributed information retrieval [20], network computing [5], and bioinformatics [15]. Some of these applications require large Bloom filters to reduce the false positive rate. In memory limited environments dedicating large memories for Bloom filters may not be feasible. A naive approach to tackle this issue would be to put the Bloom filter on secondary storage, either explicitly or by using memory-mapped I/O. Each probe would then require one or more random I/O operations, which would become a severe performance bottleneck. Therefore, simply placing the filters on disk will not provide a sufficient quality of service particularly for time critical applications.

A less naive approach can be used when probes come in bulk. Instead of one large filter, we store $\delta$ disjoint filters. A record is hashed to one of the $\delta$ filters during the build or probe, and the access is performed on the smaller filter. However, instead of performing the access immediately, we instead store the access request in a main-memory buffer. Only when a buffer becomes full (or when we flush a buffer periodically) do the accesses actually get applied to the corresponding filter partition. Because all of the accesses to the component filter get applied at once, we improve the access locality, amortizing the I/O cost and cache-miss cost of retrieving the filter into RAM/cache over a large number of accesses. This idea is motivated by a similar idea used by Zhou and Ross for buffering accesses to B-trees to improve locality of reference [23].

As we shall see experimentally, even this buffering approach performs poorly when the secondary storage device is a hard disk. The random I/O-cost is sufficiently large that it dominates the overall cost and limits throughput. However, more recent secondary storage devices based on solid state flash memory have demonstrated much better random I/O performance. We therefore study whether such devices have the potential to provide cost-effective solutions for accessing large Bloom filters.

To perform this study, we have chosen to use what is today a high-end flash memory device, namely the Fusion-IO card [10]. See Table 1 for the specifications of this device. As of May 2010, the retail price[1] of this device is about \$38 per gigabyte. A higher-capacity MLC device by the same vendor has 60% of the throughput, but a price of \$15 per gigabyte. Note that such flash devices are in a relatively early stage of development, and prices are expected to drop as they become more widely adopted. Industry experts seem to agree (see e.g., [11]) that within the next few years the price per Gigabyte of SSDs will be more than an order of magnitude below the price of main memory, and approaching the price of hard disks.

[1]All quoted prices were obtained from the Dell web-site. For comparison purposes, the retail price of server RAM is about \$82 per gigabyte using 8GB DIMMs, or \$25 per gigabyte using 4GB DIMMs.

The main contribution of the paper is an SSD-aware Bloom filter algorithm that reduces the main memory usage by a factor of at least four without any performance or quality loss.

## 1.1 Motivating example

To motivate the need for Bloom filters utilizing flash storage, consider the following example based on the TPC-H schema. Suppose a user poses the following query

```
Select *
From Lineitem L, Orders O
Where L.Orderkey=O.Orderkey
  And L.Quantity>1
  And O.Orderstatus='P'
```

In relational algebra, we might write this query as

$$\sigma_{C1}(L) \bowtie_J \sigma_{C2}(O)$$

where $C1$ and $C2$ are the local selection conditions on each table, and $J$ is the join condition on `Orderkey`. Suppose that the selectivities of $C1$ and $C2$ are $\theta_1$ and $\theta_2$ respectively, and that $C1$ and $C2$ are statistically independent over the join of `L` and `O`. Note that `Orderkey` is a foreign key from `L` to `O`.

If this database was stored in a distributed fashion, with `L` and `O` stored at different sites, then data transmission can be reduced by employing a semijoin or Bloom filter [18]. For example, one could ship

$$P = \pi_{\texttt{Orderkey}}(\sigma_{C2}(O))$$

or a Bloom filter of $P$ from the site hosting $O$ to the site hosting $L$ in order to reduce the amount of data from $L$ that needs to be transmitted over the network. When the network is the bottleneck, such data reduction improves performance significantly.

Now imagine that both $O$ and $L$ are stored in a complex storage subsystem. This subsystem contains a variety of resources, including multiple persistent devices of various kinds, including hard disk drives and solid state drives. The subsystem contains a limited amount of processing power and memory, sufficient to manage and optimize the storage system, but not designed to perform computations that require extensive CPU or memory resources. The storage subsystem serves as the storage layer for many processing nodes, connected to the storage subsystem via a shared high-bandwidth channel such as infiniband [12].

Even a high-bandwidth channel can be a performance bottleneck. Since the storage subsystem abstracts a large number of devices, and since data is requested by many processing nodes, there can be contention for the channel. In such a situation, it would be advantageous to preprocess data in the storage subsystem to reduce the bandwidth required. One straightforward way to reduce bandwidth is to compress the data.

Another way to reduce bandwidth is to filter out useless data so that it is not transmitted in the first place. An example of this kind of approach is the Netezza TwinFin system [19]. That system uses FPGAs in the storage layer to apply local selections such as $C1$ and $C2$ above so that only rows satisfying the local selections are transmitted to compute nodes. The data volume is reduced by a factor of $\theta_1$ for $L$ and $\theta_2$ for $O$.

There still remain opportunities for further data reduction. In the query mentioned above, the join induces a further data reduction because the local conditions on the two tables are independent. In fact, the fraction of each table that contributes to an output row for this query is $\theta_1\theta_2$. It would be unrealistic to expect the storage subsystem to perform the full join. In the absence of special clustering properties, the join operation would require substantial CPU and memory resources. What is more, this join might be part of a larger query plan in which these tables are not joined directly, and performing the full join of these two tables would be suboptimal.

Nevertheless, we argue that the storage subsystem could effectively optimize the bandwidth requirements by applying a Bloom filter to $L$ and $O$ before transmitting the data. Two Bloom filters would be created, one for $L$ and one for $O$, representing the set of `Orderkey` values satisfying the local selection conditions. These filters would then be used to further reduce the data in the other table before data is transmitted across the data channel to the compute nodes.

The challenge in such a system is performing the Bloom filter reduction in a low-resource environment such as a storage subsystem. We expect to have a small amount of RAM relative to the total disk capacity, and much of that RAM could be devoted to other functions such as caching. On the other hand, the storage subsystem does have plentiful persistent storage. We therefore propose an approach that leverages the persistent storage, particularly solid state storage that has high random-I/O rates. Our approach allows efficient Bloom filter reduction to scale to much larger configurations than could be handled with RAM alone.

In the remainder of this paper, we first recap the characteristics of traditional Bloom filters (TBF). In section 3, we introduce our extensions to the TBF that allows it to be SSD-resident without sacrificing creation and probing efficiency. The resulting data structure is called "*buffered Bloom filter*" (BBF). Finally, section 4 presents detailed experiments for a variety of BBF usage scenarios. Section 5 discusses related work and section 6 concludes the paper.

## 2. TRADITIONAL BLOOM FILTERS

To introduce the traditional Bloom filters we use the framework and notations described in [9, 17, 13]. A Bloom filter is a method to represent a set $S = \{R_1, R_2, ..., R_n\}$ to support membership queries. Each element of $S$, $R_i$, is any item that can be an argument of a hash function. It was invented by Burton Bloom in 1970 [4] and has been used in numerous applications since then.

A Bloom filter consists of a vector of $\beta$ bits, initially all set to 0. To update the filter, k independent hash functions $h_1, h_2, ..., h_k$ all with range $\{1, ..., \beta\}$ are used. For each element $R \in S$, the bit locations $h_i(R)$, $i \in \{1, 2, ..., k\}$, are set to 1. Note that a particular bit location could be set to 1 several times since the hash results of different elements may collide. To check if an element $e$ is in $S$, the bit locations $h_i(e)$, $i \in \{1, 2, ..., k\}$, are checked. If any of them is not set to 1, we conclude that $e$ is not an element of $S$. Otherwise, we assume that $e$ is an element of $S$, with a certain probability that the assumption is wrong. If we conclude that the element is in the set although it is not, that element is called a false positive.

## 2.1 The Locality Problem

The probability of cache misses has a significant impact on the execution time of TBF. Each bit location on the filter has the same access probability during the execution of both build and probe phases. In the worst case the distance between two consecutive bit locations that are accessed could be as large as the size of the filter. For small datasets for which the filter size is smaller than the CPU cache, cache misses are not expected to occur since the filter is small enough to fit in the cache. For larger datasets the filter size should be increased so as to keep the false positive rate below an acceptable threshold. As the filter size increases, the probability of cache misses will grow proportionally. If the cache size is $c$ bytes and the filter size is $f$ bytes, then the probability that an accessed bit of the filter is not cached is $\frac{f-c}{f}$. As the cache miss probability increases, the execution cost of the Bloom filter would increase gradually.

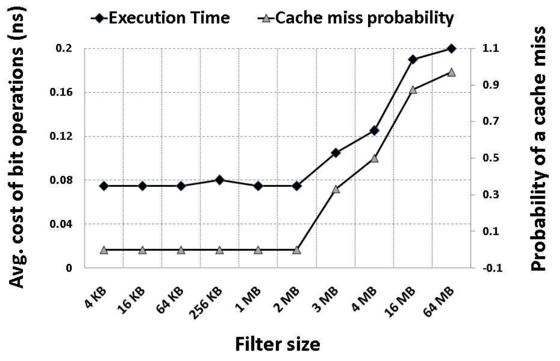

**Figure 1: Effect of filter size on the cost of bit operations**

To observe the impact of cache misses on the performance of Bloom filters, we implemented the build phase and probe phase of the Bloom filter. For both of these phases we have used 100 million records and a single hash function. We run the experiment with different filter sizes in the range [4KB, 64MB] on a platform with 2MB CPU cache (L2 cache). In another experiment setup, we measured how much time is spent in the other steps of the algorithms and subtracted that amount from the previous measurements to isolate the cost of bit operations. The experiment results are given in Figure 1. The x axis represents the Bloom filter size which is stored in the memory[2]. The vertical axis on the left indicates the average cost of a single bit operation for different filter sizes. The vertical axis on the right indicates the probability of cache misses that is computed using the formula $\frac{f-c}{f}$. For instance, if the filter size is 16MB and the CPU cache is 2MB, 14 out of 16 bit accesses are expected to cause a cache miss. Note that, as the filter size increases, the probability of cache miss increases as well and rapidly converges to 1. Moreover, the experimental results are precisely parallel with the analytical observations.

In the following section we introduce buffered Bloom filters (BBF) which prevent cache misses effectively regardless of the size of the dataset. Even though the total size of the filter exceeds the CPU cache size, the cache misses are prevented by partitioning the Bloom filter into sub-Bloom filters. The memory partition in each sub-Bloom filter on which the bit operations are performed does not exceed the cache size. This, in turn, prevents the cache misses effectively and yields a significant performance gain.

[2]In Figure 1 the step size has a finer granularity between 1MB and 4MB

## 3. BUFFERED BLOOM FILTERS

Due to the locality problem in the Bloom filters, storing the filter in lower storage layers such as conventional hard drives would entail excessive random access cost. SSDs alleviate this problem by eliminating mechanical head movements. Based on this observation, one can suggest moving the filter to the SSD without any modification in the traditional Bloom filter algorithm. However, this will yield a significant performance loss since random access speed of the SSDs is about one order of magnitude slower than that of memories. In this paper, we propose a modified version of the traditional Bloom filter so as to implement it on a solid state disk. To compensate for the performance loss, we propose the idea of deferring the read and write operations during the execution of both build and probe phases. With this improvement the number of SSD page accesses is reduced by up to three orders of magnitude depending on the buffer size. As a second improvement, we propose dividing the Bloom filter into virtual sub-Bloom filters to improve the locality of the bit operations. This modification reduces the CPU cache misses and helps to compensate for the cost of IO operations.

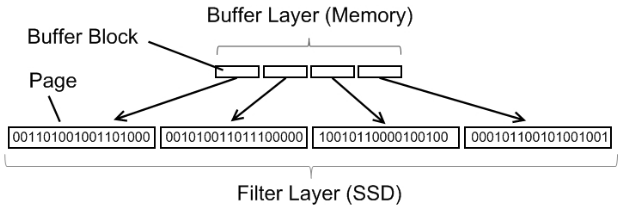

**Figure 2: Structure of Buffered Bloom Filter**

The structure of the BBF is depicted in Figure 2. The major difference between TBF and BBF is that the latter has a hierarchical structure. TBF consists of a single layer, stored in the main memory, which keeps the filter. In contrast, there are two layers in the BBF. The upper layer is called *buffer layer* and the lower layer is called *filter layer*. The buffer layer is stored in the main memory while the filter layer is stored in the SSD. As the name implies, the buffer layer is used to buffer the auxiliary information pertaining to the deferred reads and writes. The filter layer, on the other hand, is used to store the filter (bit vector). The filter is divided into virtual pages such that each page keeps the bit vector of a sub-Bloom filter. If the filter is divided into $\delta$ pages, the buffered Bloom filter can be considered as a combination of $\delta$ separate traditional Bloom filters. Each sub-Bloom filter has a dedicated buffer block; therefore, the number of buffer blocks is equal to the number of filter pages.

### 3.1 The Build Phase

The build phase of the BBF is described in Algorithm 1. Similar to the TBF, the input parameters include the list of records, $BuildArray$, the size of the build array, $n_b$, and the number of hash functions, $k$.

**Algorithm 1** Buffered Bloom Filter - Build Phase:

```
1: Given BuildArray, $n_b$, $k$
2: for $i = 0$ to $n_b$ do
3:    $B_i \leftarrow h_0$(BuildArray[i])
4:    for $j = 1$ to $k$ do
5:       $\Omega_j \leftarrow h_j$(BuildArray[i])
6:       addOffsetValueToBufBlock($\Omega_j$, $B_i$)
7:       if isBufferBlockFull($B_i$) then
8:          retrievedBlock $\leftarrow$ readPageFromSSD($B_i$)
9:          for each offsetVal in BufferBlock[$B_i$] do
10:            retrievedBlock[offsetVal] $\leftarrow 1$
11:            removeOffsetValFromBufBlock(offsetVal, $B_i$)
12:         end for
13:         writeUpdatedPageBackToSSD(retrievedBlock)
14:      end if
15:   end for
16: end for
```

While processing the records of the build array, an initial hash function, $h_0$, is used to determine the sub-Bloom filter to which the records will be assigned. For each record $R_i$ in the $BuildArray$, the block number $B_i$ is equal to $h_0(R_i)$. If the filter is divided into $\delta$ virtual pages, $B_i$ would be a value in the range [0, $\delta - 1$].

Once the block number is determined (line 3), subsequent hash functions, $h_1, .., h_k$, are used to find the *offset values* for $R_i$, that is, $\Omega_j = h_j(R_i)\,\forall j \in (1, ..., k)$, in each iteration of the for loop starting at line 4. The range of $\Omega_j$ is [0,$\beta$] where $\beta$ is the size of the filter page (i.e. total number of bits in a filter page). If $\Omega_j$ is equal to $x$, the $x^{th}$ bit of the filter page is expected to be set to 1. However, instead of updating the ${B_i}^{th}$ filter page immediately, the offset value, $\Omega_j$, is placed to the available position at the ${B_i}^{th}$ buffer block (line 6). For each record, $k$ offset values are computed and stored in the buffer blocks. Note that in TBF, for any record, $x$ would be any bit location in the filter. In contrast, in BBF the range of $x$ is limited to the size of a filter page. If the same record is processed in the probe phase, the record will be forwarded to this sub-Bloom filter to check the bit values corresponding to the hash values of the record.

At any given time each buffer block can be either in "Growing Phase" or "Shrinking phase". Initially, all buffer blocks are assumed to be in the growing phase. As new records are processed, the buffer blocks are filled with the offset values (line 6). Once a block gets full, the growing phase of the block ends and the shrinking phase starts by retrieving the corresponding page from the SSD to the memory (line 8). In the shrinking phase, for each offset value waiting in the buffer block, the corresponding bit position of the retrieved page is set to one (line 10) and the offset value is removed from the buffer block (line 11). After processing all of the offset values in the buffer block, the page is written back to the SSD (line 13). Once the offset values are removed, the buffer block gets ready for the next growing phase.

The growing and shrinking phases of the buffer blocks continue as long as new records are read from the stream. Each buffer block enters the growing phase and the shrinking phase consecutively. At any time, at most one buffer block could be in the shrinking phase. The growing phase does not require any disk accesses. All of the updates in this phase are performed on the main memory. On the other hand, the shrinking phase requires retrieval of an SSD page and writing this updated page back to the SSD.

## 3.2 The Probe Phase

The probe phase of the BBF is described in Algorithm 2. The input parameters include the list of records, $ProbeArray$, the number of records in the probe list, $n_p$ and the number of hash functions, $k$.

Initially, all buffer blocks are assumed to be in the growing phase. Similar to the build phase, for each record $R_i$ in ProbeArray, the initial hash function is used to determine the sub-Bloom filter to which the record will be assigned. If $B_i$ is equal to $h_0(R_i)$, $R_i$ is placed to the available position at the ${B_i}^{th}$buffer block (line 4). If the buffer block gets full, the shrinking phase starts and the corresponding page is retrieved from the SSD to the memory (line 6). For each record waiting in the buffer block, subsequent offset values are computed using the hash functions. If all of the bits in the retrieved page corresponding to the computed hash values are set to 1, the record is output as positive. If any of these bits is set to 0, the record is output as negative indicating that the record ID does not exist in the build list.

The same procedure is applied as long as new probe records are queried.

**Algorithm 2** Buffered Bloom Filter - Probe Phase:

```
1: Given ProbeArray, $n_p$, $k$
2: for $i = 0$ to $n_p$ do
3:    $B_i \leftarrow h_0$(ProbeArray[i])
4:    addRecordIDToBufBlock(ProbeArray[i], $B_i$)
5:    if isBufferBlockFull($B_i$) then
6:       retrievedBlock $\leftarrow$ readSSDpage($B_i$)
7:       for each recordID in BufferBlock[$B_i$] do
8:          for $j = 1$ to $k$ do
9:             $\Omega_j \leftarrow h_j$(recordID)
10:            if retrievedBlock[$\Omega_j$] = 0 then
11:               outputNegative(recordID)
12:               break
13:            else
14:               if j = $k$ then
15:                  outputPositive(recordID)
16:               end if
17:            end if
18:         end for
19:         removeRecordIDFromBufBlock(recordID, $B_i$)
20:      end for
21:   end if
22: end for
```

## 3.3 Probability of False Positives in BBF

In this section we show that $P_{FP}$ in BBF is independent of the number of sub-Bloom filters and therefore is equal to $P_{FP}$ in TBF.

Let $b$ be the number of bits in a filter page and $\delta$ be the number of buffer blocks ( i.e. the Bloom filter is divided into $\delta$ sub-Bloom filters). In the shrinking phase of a buffer block $B_i$, after processing an element $R_i$, the probability that a single bit of the filter page is not set to 1 by any of the k functions is:

$$\left(1 - \frac{1}{b}\right)^k \tag{1}$$

Assuming that the hash function, $h_0$, used to assign the records to the buffer blocks is perfectly random, the expected number of records assigned to each buffer block will be:

$$\upsilon = n_b \times \frac{1}{\delta} \tag{2}$$

After inserting $\upsilon$ elements to the ${B_i}^{th}$ buffer block, the probability that a single bit of the filter page is not set to 1 by any of the k functions is:

$$\left(1 - \frac{1}{b}\right)^{k \times \upsilon} \tag{3}$$

Hence, at the end of the build phase, the probability that a certain bit in the ${B_i}^{th}$ filter page is set to 1 by at least one hash function is:

$$1 - \left(1 - \frac{1}{b}\right)^{k \times \upsilon} \tag{4}$$

Now, $P_{FP}$, the probability of getting a false positive for an element assigned to $B_i$ is:

$$P_{FP} = \left(1 - \left(1 - \frac{1}{b}\right)^{k \times \upsilon}\right)^k \approx \left(1 - e^{\frac{-k \times \upsilon}{b}}\right)^k \tag{5}$$

Let $\beta$ be the total number of bits in the filter. Then $\delta$, the number of buffer blocks, is equal to $\frac{\beta}{b}$, where $b$ is the number of bits in a filter page. Substituting for $\delta$ in equation 2, we get $\upsilon = n_b \times \frac{b}{\beta}$. The formula for the probability of false positives is now given by the following equation:

$$P_{FP} \approx \left(1 - e^{\frac{-k \times n_b \times \frac{b}{\beta}}{b}}\right)^k = \left(1 - e^{\frac{-k \times n_b}{\beta}}\right)^k \tag{6}$$

## 4. EXPERIMENTS

In this section, we first give the hardware and software specifications of the system where we conducted the experiments. In Section 4.2, we then discuss the starvation problem in response time critical applications. Then, in Section 4.3 we compare the performance of the proposed algorithm running on an SSD versus an HDD. In Section 4.4, we examine the impact of the buffer size on the performance of the BBF. We conclude this section by presenting the observations on the impact of the ratio of probe to build records.

### 4.1 Hardware & Software Specifications

All experiments are conducted on a 64 bit Fedora 8 (Linux kernel 2.6.24) operating system. IBM DB2 V.9 is used as the DBMS software. The system that is used to run all the experiments has an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz with 2MB L2 cache per each core and 4GB memory. Hardware specifications for the solid state disk are given in Table 1.

**Table 1: Hardware specifications of the SSD**

| | |
|---|---|
| Brand: | Fusion IO |
| NAND Type | Single Level Cell (SLC) |
| Storage capacity: | 80GB |
| Interface: | PCI-Express x4 |
| Write Bandwidth: | 550 MB/s (random 16K) |
| Read Bandwidth: | 700 MB/s (random 16K) |
| IOPS: | 88,000 (70/30 random 4k mix) |
| Access Latency: | 50$\mu$s Read |
| Wear Leveling: | 24yrs (@ 5-TB write-erase/day) |

### 4.2 Starvation Avoidance

Bloom filters are used in numerous applications. In some of these the filter is unaltered over time and is mostly used for processing the incoming stream of queries. Using Bloom filters as spell checkers in word processors is an example where the filter remains almost the same as the dictionary words are not expected to change frequently [16]. In some applications the response time could be critical. Once the probe requests of the client application(s) are inserted into a request queue, the applications may expect to get the result of the query within an acceptable time period. This is not a concern for the traditional Bloom filter as the probing requests are processed and output in the same order of insertion into the *request queue*. In contrast to TBF, the buffered Bloom filter does not guarantee that the output order of the requests will be similar to the order of the request queue. Therefore, the data structure does not guarantee that the queries will be processed within a certain amount of time. Even though a perfectly random hash function is used to distribute the probe requests to the buffer blocks, the processing latency of the request will depend on the distribution of the probes. If a specific record is probed frequently, the buffer block corresponding to this record will fill up faster than other buffer blocks. This, in turn, may cause some of the probes to wait for a long time while some of them are processed and output faster. To avoid starvation the buffer blocks should be flushed if a certain timeout period, $t_{max}$, is exceeded even though the block has not been filled up. A dedicated timer is assigned to each buffer block to keep track of time since the last flushing. Once a probe record is inserted into an empty buffer block, the timer is reset. After a while the buffer block is flushed, provided that the block is not flushed within $t_{max}$ unit of time.

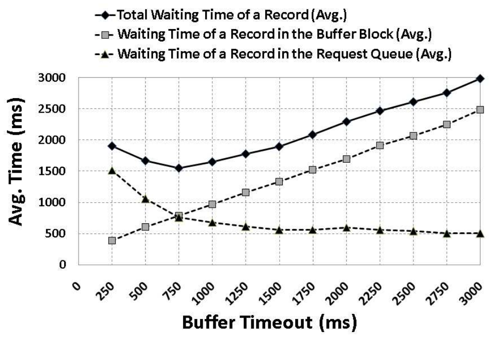

**Figure 3: The impact of $t_{max}$ on the average waiting time of the probe requests**

One can suggest choosing $t_{max}$ as small as possible in order to reduce the average waiting time of the probe requests. However setting $t_{max}$ below a certain threshold would increase the average waiting time of the probe requests. If $t_{max}$ is very small, fewer records will be processed while flushing the buffer blocks. Due to the cost of I/O operations, this will decrease the overall throughput even though the waiting time of the records in the buffer blocks is reduced. As a side effect, the waiting time of probe requests in the request queue will be much longer. Therefore $t_{max}$ should be chosen deliberately so as to minimize the waiting time of the records in both the request queue and the buffer blocks.

We conduct an experiment in order to analyse the impact of $t_{max}$ on the average waiting time of the probe requests. In this experiment 50 million records are inserted into the buffered Bloom filter and 50 million records are probed. The total buffer size is set to 32 MB and a 256MB SSD space is used for the filter. Two hash functions are used for probe operations. The request queue size is set to 3MB. A dedicated timer is assigned to each record to measure the average latency of the records. The experiment results are shown in Figure 3.

The x axis represents the timeout period, $t_{max}$, for the buffer blocks. The upper line in the graph represents the average time of a record including the time spent in the request queue and the time spent in the buffer blocks. The other two dashed lines represent the components of the total latency. The time spent in the buffer blocks exhibits an upward trending pattern while the waiting time in the request queue exhibits a downward trend. As $t_{max}$ increases, the time spent in the buffer blocks increases since the frequency of flushes gets smaller. Since the IO cost is proportional to the frequency of flushes, the throughput increases and the time spent in the request queue declines. In this configuration, setting $t_{max}$ to 750 milliseconds minimizes the total waiting time of the probe requests as seen in the figure. Using this configuration, we can obtain good throughput while still achieving a response time of less than a second for any individual probe.

## 4.3 Hard Disk Experiments

We conducted two experiments to compare the performance of the proposed algorithm running on an SSD versus an HDD. The hardware specifications of the HDD used in these experiments are as follows: 500GB Hitachi Ultrastar A7K1000, 8.2 ms average seek time and 4.17 ms average latency, 1070 Mb/sec maxs media transfer rate.

The experiment parameters are provided in Table 2 under the BBF column (the buffer size is set to 16MB).

In the first experiment we run the BBF on both SSD and HDD without modifying any part of the BBF algorithm. As a result of this experiment we observed that BBF runs 8 times faster on the SSD than on the HDD. This is not a surprising result as the buffer blocks fill up in a random order and trigger random disk accesses while processing the records.

One might try to address the random I/O difficulties of hard disks by converting some of the random I/O to sequential I/O. For example, one could store a single large buffer of access requests, organized according to the physical order of the partitioned filters on the disk. The entire Bloom filter would be scanned sequentially each time this buffer is filled. We implemented a variant of this hard disk based method and observed that the running time in a heavily loaded scenario is 4.8 times slower than the running time of the BBF on the SSD. This is actually quite reasonable, since we used just one hard disk that was more than 4.8 times cheaper than the SSD. Nevertheless, the hard disk based algorithm is much less flexible, and would perform poorly in situations where response time is being optimized ahead of throughput. Because the SSD's random I/O is so much faster than that of a disk, it can handle many more requests in a random I/O-limited setting.

## 4.4 Impact of Buffer Size on Execution Times

Our goal in this experiment is to examine the impact of buffer size on execution time of the BBF. The parameters used in this experiment are given in Table 2. We specify an upper limit for the false positive ratio and use 128 MB of memory space for the TBF with 3 hash functions in order to keep the ratio below this upper limit. As for the BBF, 256 MB of SSD space is used with two hash functions. In each run of the experiment we change the buffer size for the BBF while keeping the remaining parameters in the table constant.

**Table 2: Parameters used in Experiment 1**

| Parameter | Buffered BF | Traditional BF |
|---|---|---|
| Num. of Build Records | 50 Million | 50 Million |
| Num. of Probe Records | 50 Million | 50 Million |
| False Positives | $< 0.25$ % | $< 0.25$ % |
| Num. of hash functions | 2 | 3 |
| Page Size (MB) | 2 | N/A |
| Buffer Size (MB) | Variable | N/A |
| Filter Size (MB) | 256 (SSD) | 128 (Memory) |

As long as the buffer size and the number of hash functions are kept constant, the buffer size has no impact on the false positive ratio. However, changing the buffer size affects the execution cost since the number of disk accesses is inversely proportional to the buffer size.

The execution times of each run for different buffer sizes are given in Figure 4. Note that the execution times corresponding to the shaded bars are constant because none of the parameters for the TBF changes. The first black column corresponds to the run where we use 8 MB of memory space for the BBF. In this case the TBF runs much faster than the BBF. We gradually increase the allocated memory size for the BBF. The BBF runs faster than the TBF provided that the buffer size is at least 32 MB of memory space.

We also measured the false positives in another experiment setting where we used 32 MB of memory space for BBF and 64 MB for TBF and set all the other parameters to the values given above. We observed that TBF yields at least an order of magnitude more false positives than BBF does.

## 4.5 Different Ratios of Probe to Build Records

We conducted some experiments to see how sensitive the overall execution time is to the ratio of the probe records to the build records. We observed that the ratio has a slight impact on the overall execution time. As we increase the number of probe records while reducing the number of build records proportionately, the overall execution time slightly decreases. In one setting we run the BBF algorithm with 10 million build records and 90 million probe records and observed that it takes about 6 % less time than it takes to run the same algorithm with 50 million build records and 50 million probe records. This is mainly because the build phase includes both read and write operations while the probe phase consists of read-only disk accesses.

## 5. RELATED WORK

The Bloom filter has been introduced in [4], as a probabilistic data structure for set membership queries. Since then, it has been used extensively in a variety of application areas, such as spell-checkers [16], distributed information retrieval [20], network computing (e.g. distributed caches, P2P networks, resource routing, packet routing) [5, 7], stream computing [22], gene sequence analysis [15, 8], as well as database query processing [18, 14, 21].

Motivated by the diverse requirements of such a large set

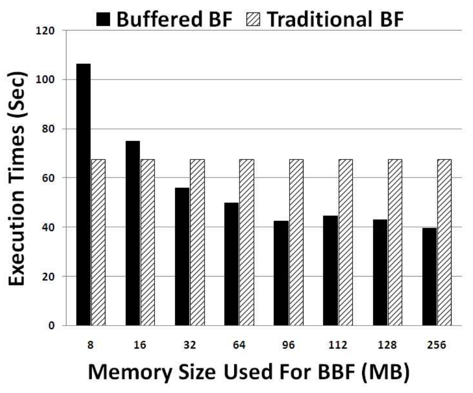

**Figure 4: Execution Times for Different Buffer Sizes for BBF**

of applications, numerous variants of the original data structure have been proposed. For example, the need to insert as well as remove elements lead to the counting Bloom filter, which stores multiplicity counts instead of bits [9]. Furthermore, in order to achieve the best accuracy for a given bit vector size the compressed Bloom filter [17] has been proposed. The idea of the compressed Bloom filter is to use the least message size for transmitting a Bloom filter over a network. In order to use the compressed Bloom filter for probing, the entire filter needs to decompressed, and that typically takes more memory than a TBF. Therefore it is not comparable with the BBF in terms of the reducing memory footprint. For environments that can tolerate small false negative rates the Bloomier filter [6] offers better accuracy compared to the traditional Bloom filter. In situations where the size of the build set is not known in advance, an adaptive Bloom filter [3] has been proposed: the idea is to create additional Bloom filters whenever the set bits become too dense and thus bring the false positive rate above an acceptable threshold. All these variants are assumed to be main memory resident, because the lack of access locality would make them impractical to store on a hard disk.

Closer to our work is the cache-aware Bloom filter proposed in [2]. This work addresses the following locality problem of the traditional Bloom filter: if $k$ independent hash functions are used, for every build or probe operation $k$ different cache lines need to be accessed. This filter improves the processor cache hit ratio by partitioning the bit vector into cacheline-long segments and making sure that all the $k$ hash functions map into the same segment for any input element. This is accomplished by dispatching the input to one of the segments via an initial hash function, and then selecting bit locations inside that segment using the $k$ hash functions. Although the goal of this work is related to ours, there is one important difference: our buffered Bloom filter achieves reference locality for multiple build or probe operations instead of just one. This is precisely what enables the buffered Bloom filter to operate on an SSD, the fact that the I/O cost is amortized over several requests.

Another attempt to improve the access locality is the hierarchical Bloom filter array [24]: the idea there is to employ a two-level hierarchy of Bloom filters, where the first level Bloom filter is small and consequently not very accurate and the second level filter is larger and therefore more accurate. The first level filter, by virtue of its smaller size will have better locality, and will reduce the number of requests that reach the second filter. Moreover, the nature of the application targeted by this solution, namely distributed file lookups in meta-data servers, provides additional temporal locality of file access patterns, which improves the locality of reference in the second layer filter. For a generic-purpose use, though, one may not expect such temporal locality of access patterns, and thus the hierarchical Bloom filter may not alleviate the locality problem enough to enable, for example, the storage of the second layer filter on an SSD.

As mentioned previously, our work is motivated in part by the use of buffering for B-trees to improve cache performance [23]. Unlike the present paper, that paper deals exclusively with cache miss latencies, and does not consider secondary storage costs.

## 6. CONCLUSIONS

In this paper we proposed the Buffered Bloom Filter (BBF) as an adaptation of the familiar Bloom filter data structure that enables it to take advantage of solid state storage, as a cheaper alternative to main memory. Since SSDs are block devices, simply porting the traditional Bloom filter to SSD would not be feasible because of the low locality of reference of the bit operations. The BBF circumvents this problem by making two, somewhat independent, locality-improving changes to the original algorithm: 1) breaking a large filter into multiple small (page-sized) sub-filters and dispatching each element to one sub-filter in a deterministic way; and 2) amortizing the I/O cost by buffering the write and read requests for the same sub-filter. The first change, by itself, improves the locality of bit operations for a single element processing in the build or probe phase. The second change further improves the locality by grouping together a set of requests for the same sub-filter so they can be applied in bulk once the sub-filter is brought to main memory.

Each of these two ideas can be applied independently, but neither will provide the same benefit as their combined use. Thus, just partitioning the filter into sub-filters, while definitely improving the processor cache hit ratio, is only suitable to a main memory resident filter. Similarly, one can just group the access requests by the referenced filter page and read the page only when enough requests have been collected, without requiring all hash functions to map to the same sub-filter. The build phase algorithm will work unchanged, even though the bit updates for the same inserted element will be carried out asynchronously. The probe phase algorithm would need to be adapted so that the results of asynchronous bit read operations can be merged to output the pass/fail status of each probe request. The extra amount of bookkeeping information and merge processing would reduce the efficiency of the algorithm without providing any benefit in terms of the false positive ratio.

While we have only focused on the Bloom filter data structure for this paper, we believe that similar partitioning and buffering techniques can be employed for other memory constrained data structures such as hash tables or search trees, thus enabling them to spill to solid state disks without any performance degradation. In fact, any large data structure which is expected to be frequently probed can be spilled to SSD, provided that the requests can be buffered and served in an out-of-sequence fashion.

## 7. REFERENCES

[1] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. E. Simmen, M. Wang, and C. Zhang. Cost-based optimization in db2 xml. *IBM Systems Journal*, 45(2):299–320, 2006.

[2] K. S. Beyer and S. Rajagopalan. System and method for generating a cache-aware bloom filter. U.S. patent application no. 2008/0155229.

[3] K. S. Beyer, S. Rajagopalan, and A. Zubiri. System and method for generating and using a dynamic bloom filter. U.S. patent application no. 2008/0154852.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003.

[6] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *in Proc. 15th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA*, pages 30–39, 2004.

[7] Y. Chen and O. Oguntoyinbo. Power efficient packet classification using cascaded bloom filter and off-the-shelf ternary cam for wdm networks. *Computer Communications*, 32(2):349–356, 2009.

[8] Y. Chen, B. Schmidt, and D. L. Maskell. A reconfigurable bloom filter architecture for blastn. In M. Berekovic, C. Müller-Schloer, C. Hochberger, and S. Wong, editors, *ARCS*, volume 5455 of *Lecture Notes in Computer Science*, pages 40–49. Springer, 2009.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998.

[10] Fusionio drive specifications. `www.fusionio.com/PDFs/Fusion%20Specsheet.pdf`.

[11] J. Handy. Flash vs dram price projections - for ssd buyers. `www.storagesearch.com/ssd-ram-flash%20pricing.html`.

[12] Infiniband architecture. `www.intel.com/technology/infiniband`.

[13] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[14] Z. Li and K. A. Ross. Perf join: an alternative to twoway semijoin and bloomjoin. In *In Proceedings of the International Conference on Information and Knowledge Management (CIKM*, pages 137–144. ACM Press, 1995.

[15] K. Malde and B. O'Sullivan. Using bloom filters for large scale gene sequence analysis in haskell. In A. Gill and T. Swift, editors, *PADL*, volume 5418 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 2009.

[16] M. D. McIlroy. Development of a spelling list. *IEEE Trans. on Communications*, 30:91–99, 1982.

[17] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[18] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Softw. Eng.*, 16(5):558–560, 1990.

[19] Netezza TwinFin system. `www.netezza.com/data-warehouse-appliance-products/twinfin.aspx`.

[20] A. Singh, M. Srivatsa, L. Liu, and T. Miller. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. In *SIGIR Workshop: Distributed Multimedia Information Retrieval*, volume 2924 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2003.

[21] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 240–251. Morgan Kaufmann, 2004.

[22] Z. Yuan, J. Miao, Y. Jia, and L. Wang. Counting data stream based on improved counting bloom filter. In *WAIM*, pages 512–519. IEEE, 2008.

[23] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, pages 405–416, 2003.

[24] Y. Zhu, H. Jiang, and J. Wang. Hierarchical bloom filter arrays (hba): a novel, scalable metadata management system for large cluster-based storage. *Cluster Computing, IEEE International Conference on*, 0:165–174, 2004.