An Efficient Data Deduplication Design with Flash-Memory Based Solid State Drive

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Guanlin Lu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Prof. David Hung-Chang Du, Advisor

January, 2012

Acknowledgements

I owe my thanks and gratitude to numerous people who have helped me in accomplishing my PhD study. First of all, I would like express my sincere gratitude to my advisor, Professor David Hung-Chang Du, for his continuous support and truly inspiring guidance throughout my 5-year PhD study. He is not only the person who has brought me into research field, but also devoted so much time and effort to teaching me thinking, presenting skills, and the attitude towards research work. More importantly, he taught me how to be a mature researcher.

Besides my advisor, I would like to thank the rest of my thesis committee members. They are Prof. Jon Weissman, Prof. Abhishek Chandra and Prof. Matthew O'Keefe, for their encouragement and insightful comments. I also hope to take this opportunity to give my appreciation to Prof. Youngjin Nam for his immensely valuable help, guidance, and suggestions in my research. Also, I would like to thank Dr. Weijun Xiao, for his great help on my research. Particularly, without his generosity in offering me his own experimental platform for months, I could not finish this thesis. Also, I thank Cory Devor and other members of CRIS group for their help and support.

I should say thanks to my mentor Dr. Erik Kruus in NEC Laboratories America for his great tutoring on my essential research skills and I will forever value and treasure the friendship with two of my friends, Dr. Biplob Debnath and Dr. Yu Jin. They have given me so much sincere help both in my research collaborations and my life.

I truly enjoy the time and experience I spend with my labmates during my PhD study. Particularly, I would like to give my thanks to Nohhyun Park, Dong-chul Park, Pramod Mandagere, Aravindan Raghuveer, Sarah Sharafkandi, Sunil Subramanya, Vishal Kher and Jim Diehl. I also thank many of my friends, including but not limited to the following: Sheng Yi, Dingchen Li, Lin Zi, Nan Jiang, Peng Peng,

Jicheng Xia and Rui Zhang, for their truly warm friendship and help they gave to me.

Last but not least, I would like to thank my family for being extremely supportive. Thanks to my wife Fan Yang for her full support, patience and love. I am deeply indebted to my parents, Baokang Lv and Lianbi Chen, for their love, support and encouragement through my life.

Dedication

To my wife Fan Yang for years of love and support; to my mother Lianbi Chen for her love and faith in me.

Abstract

Today, a predominant portion of Internet services (e.g., content delivery networks, online backup storage, news broadcasting, blog sharing and social networks) are data centric. A significant amount of new data is generated by these services every day and a large portion of this created data is redundant. Data deduplication is a prevailing technique used to identify and eliminate redundant data, so as to reduce the space requirement for both primary file systems and data backups.

The variety of objectives in a deduplication system design is the primary interest of this dissertation. These objects include maximizing the redundant data removed and achieving a high deduplication read/write throughput with a minimum RAM overhead per chunk. To achieve the first objective, this dissertation proposes a novel chunking algorithm that breaks the input dataset into chunks, with a higher redundancy or with larger sizes, so as to identify the more duplicated data without producing larger numbers of chunks, as compared to other chunking algorithms. To achieve high deduplication throughput while minimizing RAM overhead per chunk, this dissertation proposes a RAM frugal chunk index design along with a chunk filter that is used to filter out index lookups on nonexistent chunks. Both index and filter designs efficiently use a very limited RAM space with flash-memory as persistent storage. In particular, the proposed chunk filter design can dynamically scale up to adapt to the growth of the dataset. In addition, the proposed chunk index design could achieve high throughput, low latency chunk lookup/insert operations with extremely low RAM overhead at the sub-byte-per-chunk level.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today a predominant portion of Internet services, like content delivery networks, online backup storage, news broadcasting, blog sharing and social networks, etc., are data centric. Hundreds of millions of users of these services generate petabytes of new data everyday. For instance, as of April 2011, Dropbox, an online file-sharing and backup service, has more than 25 million 2GB dropboxes (total of 50 petabytes). A large portion of Internet service data is redundant for the following two reasons. Firstly, because of the significant decline in the storage cost per GB, people tend to store multiple copies of a single file for data safety or convenience. For example, it is common for people to send slides and word documents to multiple receivers for work collaborations. Each receiver holds a copy of these files, leading to an extensive amount of data redundancy in the storage system. In addition, daily or weekly full backups are routinely produced by data backup systems, which may contain a significant fraction of duplicated whole-files among a series of full backups. Secondly, while incremental (or differential) data backups or disk image files for virtual desktops tend not to have duplicated whole-file copies, there is still a large fraction of duplicated data portion from the modifications and revisions of the files. [1].

Data deduplication (briefly deduplication) is a prevailing technique used to reduce the space requirement for both primary file systems and data backups. It identifies and eliminates redundant data portions within a given collection of files. For example, on a collection of many primary file systems, deduplication is reported to reduce the space consumption to as little as 32% of its original requirement [2]. On a typical

data backup workload, deduplication performs even better by attaining a 10–30 times duplicate elimination ratio [3]. In addition, when compared with compression techniques such as gzip, deduplication technique could save much more storage space [4].

The deduplication technique divides a collection of files into a number of data chunks, and replaces identified duplicated data chunks with references (such as a SHA1 [5] hash, computed based on the content of the corresponding chunk) to the data chunks already stored on disk. A chunk reference is also referred to as a chunk ID because it is used to globally identify a data chunk.

A certain chunking algorithm is used to divide the input data stream (composed of a collection of files) into a number of data chunks, the length of which could be either fixed (e.g., Fix-size Chunking, FSC) or variable (e.g., Content-Defined Chunking, CDC). For these produced chunks, only one copy of each unique chunk is stored while the rest of the copies are replaced by references to reduce the required total storage space. Among different chunking algorithms, the simplest and fastest approach is FSC which breaks the input stream into a sequence of fixed-sized data chunks. A well known problem with FSC, called *boundary shifting*, is that editing (e.g. inserting/deleting) even a single byte in a file will shift all chunk boundaries beyond that edit point and result in a successive version of the file having very few repeated chunks. On the other hand, the CDC algorithm outputs chunks of variable sizes and addresses the aforementioned boundary-shifting problem by marking each chunk boundary only depending on the local data content, not the offset in the stream, so as to enclose the local edits into a limited number of chunks. Consequently, the CDC algorithm is the state-of-the-art chunking algorithm widely used in deduplication systems.

For each data chunk instance produced by the chunking algorithm, data deduplication process needs to search for the corresponding chunk ID in all stored chunks to determine if this chunk instance is a copy of a stored one. Given a large volume of stored (unique) chunks, an index structure can be used to speed up the chunk ID lookup operation by looking up the chunk ID in the chunk index. If this chunk is an existing one, the lookup operation may further fetch the chunk's metadata in order to retrieve it. Otherwise, the data chunk has to be stored on a disk and a corresponding key-value pair has to be inserted into the chunk index. Typically trees, hash tables, etc. are used to build a chunk index.

A chunk index maps each chunk ID with its corresponding metadata. The metadata contains information like chunk offset, chunk size, chunk location, etc. The size of the metadata is up to 44 bytes. The 20-byte chunk ID is the *key* and 44-byte metadata is the *value,* for a total key-value pair of 64 bytes (as in [6, 7, 8]). Key-value pairs are inserted/searched in a chunk index during the deduplication process. More specifically, for each data chunk instance produced by the chunking algorithm, the chunk hash (key) needs to be looked up in the chunk index to determine whether the chunk instance is a copy of a stored one. If not, the data chunk has to be stored on disk and a corresponding key-value pair has to be inserted into the chunk index. Both lookup and insert operations on the chunk index must be conducted sufficiently fast to avoid becoming the performance bottleneck of the deduplication process. Given the key-value pair size as well as the number of pairs (e.g., billions of distinct chunks) to handle, the chunk index is usually too big to be fully stored in RAM. Thus, it has to be placed on a secondary storage device (e.g., a hard disk) as well. To retrieve an individual file or a backup stream from stored chunks, data deduplication also needs to store a file recipe or a chunk list, comprised of a sequence of chunk references that make up the file or backup stream.

The performance of deduplication is usually measured by the following three well established metrics: duplicate elimination ratio, throughput, and RAM overhead per chunk. The first metric measures the *effectiveness* of the deduplication, that is, how much storage space could be saved by the deduplication. The second metric measures the read/write *throughput* of the deduplication system. The last metric measures the *scalability* of the deduplication system, because the size of the RAM is usually orders of magnitude less than that of the secondary storage on a machine, imposing a constraint on the number of chunks that can be managed. While we hope to maximize both the effectiveness and the throughput, we also hope to minimize the RAM overhead per chunk for the sake of scalability.

Ideally, a deduplication design should maximize all of the three metrics. However, there are two main contentions among these metrics. Firstly, there is a contention between the deduplication effectiveness and the throughput within the existing deduplication systems. For example, in order to increase the duplicate elimination ratio, the

CDC algorithm needs to generate small sized (e.g., 2KB–4KB) chunks, because it determines chunk boundaries randomly. However, this would result in more chunks being produced, increasing the per-chunk RAM overhead as well as the number of chunk (reference) index access (lookup/insert) operations. Since the backup time is usually fixed (e.g., 12 hours), to complete a larger number of chunk lookups in a specified amount of time demands a higher chunk index access (lookup/insert) throughput. Nevertheless, as pointed out by by Zhu et al. [6], chunk index designs within existing deduplication systems could not typically provide sufficiently high chunk index access throughput, thus, becoming the main bottleneck for the deduplicated data write performance. In addition, as smaller chunks are produced, the stored data becomes more fragmented, which causes severe read throughput degradation when the users are trying to retrieve their deduplicated files.

Therefore, the critical challenge is providing the corresponding throughput necessary to move the backup data within the backup time (write throughput) and retrieve the backup data within the recovery window time (read throughput), while maintaining reasonably good deduplication effectiveness [3].

As a key step to achieving considerably good deduplication effectiveness without increasing chunk index accesses, a new chunking algorithm design is recommended. This design should be comparable to, or even better than CDC in terms of deduplication effectiveness, while producing significantly fewer number of chunks, so as to mitigate the chunk index access burden.

The second contention lies between the throughput and the RAM overhead per chunk. For instance, targeted at improving the deduplication throughput, some recent studies recommend using an in-RAM bloom filter to bounce an index lookup before querying the chunk index if the searched chunk does not exist in the bloom filter. While this approach has been proven to be successful in reducing the number of lookup operations on the index stored on secondary storage, the bloom filter consumes tens of GBs of RAM space when the dataset contains large number of chunks. This pure in-RAM bloom filter design imposes a critical scalability bottleneck to the overall deduplication storage. Furthermore, in many cases the dataset size can not be determined in advance. As such, it is inefficient to allocate the maximum possible RAM space for the bloom filter at the beginning of the run.

In addition, the other dominant factor of the deduplication throughput is the chunk index access (lookup/insert) throughput performance. As previously mentioned, a chunk index is usually split over RAM and secondary devices, like hard disks. For instance, a typical chunk index design stores all key-value pairs on a secondary storage device and maintains an in-RAM hash table to map each chunk hash to its associated key-value pair stored. This mapping process is conducted by having each of the hash table entries containing a pointer pointing to the associated key-value pair on the secondary storage device, which makes the index RAM usage linear to the number of chunks stored, with the linearity constant no less than the pointer footprint. Although this chunk index design optimizes the throughput by having one chunk lookup conducted with at most one read to the secondary storage device, it also raises a key scalability issue due to the limited number of chunks that can be indexed in the RAM.

Therefore, a scalable chunk index design should aim at minimizing RAM space usage per chunk when a very large number of chunks needs to be indexed. While some existing designs successfully achieve high index access throughput, they fail to handle a very large number of chunks due to the relatively high RAM space usage [8, 6].

On the other hand, flash-memory based Solid State Drives (SSDs) are seeing widespread adoption as one layer in the storage stack. Due to its 100 – 1000 times lower access time than conventional hard disks, SSD seems to be a good candidate to replace hard disks for chunk index storage. However, many characteristics of SSDs are unique and need to be taken into account. Moreover, the mixing of the RAM and the flash-memory raises even more challenging research questions in how to efficiently use both of them. That is, a desired chunk index design should consider both sides of the story, the RAM and the flash-memory.

Last but not least, the contradictory goals of reducing data redundancy and providing data reliability draw attention to reliability issues of deduplication storage [9, 10, 11, 12]. Intuitively, data reliability techniques protect information against failures by generating more redundant information, while the deduplication process eliminates redundant information from a given collection of files. Many reliability-aware deduplication storage systems have managed both of these issues in a separate manner. More specifically, to provide the demanded reliability for a collection of files, deduplication storage eliminates any duplicated (shared) chunks from the collection by first sharing the

already existing chunks and then by applying the erasure coding [13] for the remaining (unique) chunks.

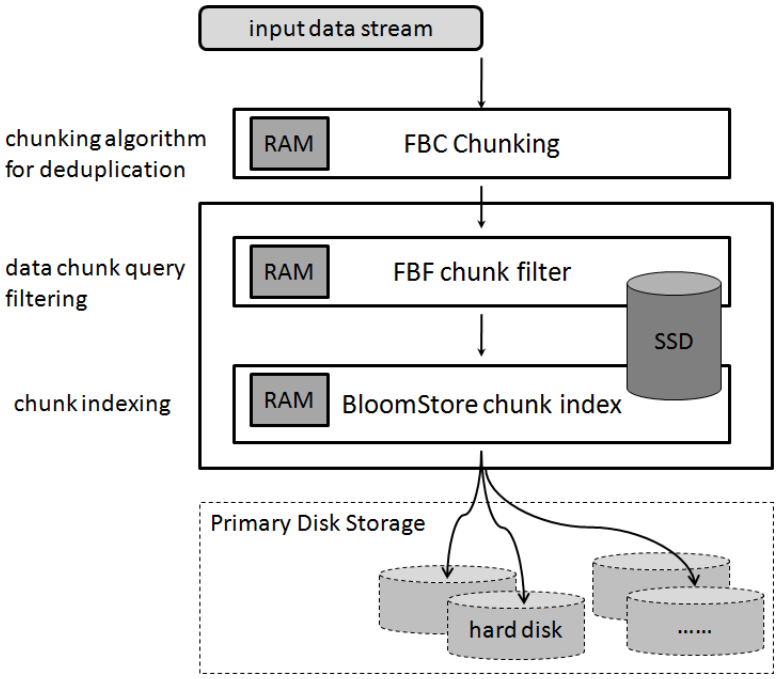

Figure 1.1: Proposed deduplication architecture

This dissertation presents an efficient deduplication design which focuses on maximizing the deduplication effectiveness and achieving high deduplication read/write throughput with minimum RAM overhead per chunk. We address the aforementioned issues with a multi-layer framework (see Figure 1.1), which combines RAM and flash-memory as our main playground. The three components in the solid-line rectangle represents our proposed design while the components in the dashed-line rectangle are the persistent storage layer for the deduplicated data chunks.

We present a novel chunking algorithm for deduplication which aims at identifying more duplicated data without increasing the total number of produced chunks. This proposed chunking algorithm is called Frequency Based Chunking (FBC). The FBF utilizes the occurrence frequency (i.e., an indicator of the redundancy level) of the data segments to help produce the data chunks. The FBC algorithm persistently outperforms the existing chunking algorithms by identifying more of the duplicated data and

producing fewer unique chunks.

As an efficient data chunk query filtering design, the Forest-structured Bloom Filter (FBF) combines the very limited RAM space with a much larger flash-memory space to dynamically adapt to the growth of a dataset, that is, a sequence of chunk IDs are produced by the chunking algorithm.

As for chunk indexing, we propose a key-value store namely BloomStore, which maintains both the key-value pairs and the corresponding indexes together on flash, while efficiently using available RAM space as an index update buffer and data (key-value pairs) buffer. BloomStore is targeted at delivering high throughput lookup/insertion operations to meet the demands of the aforementioned server applications and services, at extremely low RAM space usage. Although BloomStrore is evaluated with deduplication workloads, this key-value store is also suitable for serving a broad range of applications which demand high throughput and low latency key-value pair lookup/insert operations with very low RAM consumption.

The contributions of this dissertation can be summarized as follows:

1. We illustrate that by considering the data chunk frequency in the chunking algorithm design, more duplicated data could be identified without increasing the total number of produced chunks. We propose a RAM space efficient, chunk frequency estimation approach to obtain the chunk frequency information.

2. We propose a novel bloom filter design with flash-memory, which can achieve significantly higher key query and insert throughput with the same RAM usage, compared with the other existing designs. Furthermore, our design could dynamically adapt to the growth of a set of keys. Finally, the proposed BF update buffer scheme is able to reduce the number of flash-memory writes by up to 50%.

3. We propose a new key-value store design on the flash-memory. It can deliver high throughput and low latency lookup/insert operations ($> 10,000$ ops/second) with extremely low RAM overhead per key (< 1 byte), which is lower than all existing key-value store designs with flash memory.

The rest of the dissertation is organized as follows. Chapters 2, 3 and 4 describe the aforementioned chunking algorithm, data chunk query filter design and the chunk index design. Chapter 5 concludes and provides final discussions for this dissertation.

Chapter 2

Frequency-Based Chunking

2.1 Introduction

Today, a predominant proportion of Internet services, like content delivery networks, news broadcasting, blog sharing and social networks, etc., are data centric. A significant amount of new data is generated by these services every day. It is a challenging task for current storage systems to efficiently store and maintain backups for data at the Internet scale.

In comparison to the conventional compression techniques which do not support fast retrieval or modification of a specific data segment, chunking based data deduplication (dedup) is becoming a prevailing technology to reduce the space requirements for both primary file systems and data backups. In addition, it is well known that for certain backup datasets, the dedup technique could achieve a much higher dataset size reduction ratio when compared with compression techniques such as gzip (e.g., the size folding ratio of the chunking based dedup to that of gzip-only is 15:2. [4]).

The basic idea for chunking based data dedup methods is to split the target data stream into a number of data chunks, the length of which could be either fixed (Fixed-Size Chunking, FSC) or variable (CDC). Among these produced chunks, only one copy of each unique chunk is stored. Hence the total required storage space is reduced.The popular baseline CDC algorithm employs a deterministic distinct-sampling technique to determine the chunk boundaries based on the data content. More specifically, the CDC algorithm moves forward a fixed-length sliding window over the data stream byte by

byte. If a data segment covered by the sliding window is sampled by the CDC algorithm meeting certain conditions, it is marked as a chunk cut-point. The data segment between the current cut-point and its preceding cut-point forms a resultant chunk.

In this chapter as a first step to assessing the duplicate elimination performance for chunking based dedup algorithms, we advance the notion of the data dedup gain. Since this metric takes into account the metadata cost for a chunking algorithm, it enables us to compare different chunking algorithms in a more realistic manner. Based on the data dedup gain metric, we analyze the performance of the CDC algorithm in this chapter.

Though the algorithm is scalable and efficient, the content defined chunking is essentially a random chunking algorithm which does not guarantee the appeared frequency of the resultant chunks and hence may not be optimal for the data dedup purpose. For example, in order to reduce the storage space (i.e., to increase the dedup gain), the only option for the CDC algorithm is to reduce the average chunk size. That is, to increase the distinct sampling rate to select more cut-points. However, when the average chunk size is below a certain value, the gains in the reduction of redundant chunks is diminished by the increase in the metadata cost.

Being aware of the problems in the existing CDC algorithm, we propose a novel Frequency Based Chunking (FBC) algorithm in this chapter. The FBC algorithm is a two-stage algorithm. In the first stage, the FBC applies the CDC algorithm to obtain coarse-grained chunks (i.e., the average size of the produced chunks are larger than desired). In the second stage, the FBC identifies the data segments that appear frequently (globally) in each CDC chunk produced in the first-stage and uses these data segments to further selectively split the CDC chunks into fine-grained data chunks. In this way, the resultant data chunks which are output by the FBC algorithm can be categorized into two types: (1) the frequently appearing data segments (fixed-size data chunks); and (2) the chunks formed by the remaining portions of the CDC chunks. An entire CDC chunk may not be split further if it does not contain any frequently appearing data segments. Intuitively, the frequently appearing (globally) data segments are highly redundant and the redundancy of the remaining chunks is close to that of the original CDC chunks. Therefore, the FBC algorithm is expected to have a better data dedup performance than the CDC algorithm. Due to the typically huge volume of chunks in

the data streams, it is prohibitively expensive to obtain the chunks that frequently appeared (globally) by directly counting the number of times each chunk appears (chunk frequency).

In this chapter, we employ a statistical chunk frequency estimation algorithm to identify chunks that frequently appeared. In an observation of the chunk frequency distribution in the data stream, we find that the chunks of high frequency only represent a small portion of all data chunks. Hence, the proposed algorithm utilizes a two-step filtering process (a pre-filtering and a parallel filtering) to eliminate the low frequency chunks, so as to reserve resources (e.g., RAM space and CPU cycles) to accurately identify the high frequency chunks. Due to the powerful filtering mechanism, the proposed statistical chunk frequency estimation algorithm only requires a few megabytes to identify all high frequency chunks with good precision, even with a moderate threshold value (e.g., 15) for defining the high frequency chunks. Therefore, the performance of the proposed algorithm is beyond the capability of most state-of-the-art heavy hitter detection algorithms.

In order to validate the proposed FBC chunking algorithm, we utilize data streams from heterogeneous sources. The experimental results illustrate that the FBC algorithm consistently outperforms the CDC algorithm, either in terms of the dedup gain or in terms of the number of produced chunks. In particular, our experiments verify that the FBC produces 2.5x – 4x less chunks than the baseline CDC with the same Duplicate Elimination Ratio (DER), which measures how large of a portion of the data could be reduced. On the other hand, with the average chunk size no less than that used by the CDC algorithm, the FBC algorithm achieves up to a 50% higher DER.

The contribution of this chapter can be summarized as follows: (1) We advance the notion of the data dedup gain to compare different chunking algorithms in a realistic scenario. The metric takes both the gain in data redundancy reduction and the metadata overhead into consideration. (2) We propose a novel frequency based chunking algorithm (FBC) that explicitly considers the frequency information of data segments during the chunking process. To the best of our knowledge, the FBC algorithm is the first to incorporate frequency knowledge into the chunking process. (3) We design a statistical chunk frequency estimation algorithm with a small memory footprint. This algorithm is able to identify data chunks above a moderate frequency in a stream setting. This

algorithm can be applied to other application domains itself, as a streaming algorithm for cardinality estimation purposes. (4) We conduct extensive experiments to evaluate the proposed the FBC algorithm using heterogeneous datasets. The FBC algorithm persistently outperforms the widely used CDC algorithm.

The rest of the chapter is organized as follows. In the next section, we present an overview of the related work. Section 2.3 provides the motivation for proposing a FBC algorithm. The proposed FBC algorithm is described in Section 2.4 with an extensive analysis presented for each component. Section 2.5 presents the experimental results on several different types of empirical datasets. Finally, we describe our future work and some conclusions in Section 2.6.

2.2 Related Work

As in the domain of data dedup, chunking algorithms can primarily be categorized into two classes: 1. Fixed-Size Chunking (FSC) and 2. CDC. The simplest and fastest approach is FSC. FSC breaks the input stream into fixed-size data chunks. FSC is used by rsync [14]. A major problem with FSC is that editing (e.g., insert/delete) even a single byte in a file will shift all chunk boundaries beyond that edit point and result in a new version of the file having very few repeated chunks. A storage system called Venti [15] also adopts FSC chunking for its simplicity. On the other hand, the CDC algorithm produces chunks of variable sizes and addresses the boundary-shifting problem by marking each chunk boundary, only depending on the local data content, not the offset in the stream. The CDC algorithm will enclose the local edits to a limited number of chunks. Both FSC and the CDC algorithms help data dedup systems to achieve duplication elimination by identifying repeated chunks. For any repeated chunks identified, only one copy will be stored in the system.

The CDC algorithm was first used to reduce the network traffic required for remote file synchronization. Spring et al. [16] adopted the idea of Border's [17], to devise the very first chunking algorithm. It aims at identifying redundant network traffic. Muthitacharoen et al. [18] presented a CDC algorithm based file system called the LBFS which extends the chunking approach to eliminate data redundancy in low bandwidth

networked file systems. You et al. [19] adopt the CDC algorithm to reduce data redundancy in an archival storage system. Some improvements to reduce the variation of chunk sizes for the CDC algorithm are discussed in TTTD [20].

Recent research TAPER [21] and REBL [22] demonstrate how to combine the CDC and delta encoding [23] for directory synchronization. Both schemes adopt a multi-tiered protocol which uses the CDC algorithm and delta encoding for multi-level redundancy detection. In fact, resemblance detection [17, 24] combined with delta encoding [25] is usually a more aggressive compression approach. However, finding similar or near identical files [26] is not an easy task. Comparison results on delta encoding and chunking approaches are also available [27, 28].

The concept of identifying high-frequency identities among a large population is discussed in the area of Internet traffic analysis. Manku et al. [29] proposed an inferential algorithm to determine and count the frequently occurring items in the stream. However, their work is very general and does not consider the particular frequency distribution of the counted items. The idea of using a parallel stream filter to identify high cardinality Internet hosts among existing host pairs has recently been proposed [30]. In this chapter, we adopt the concept of the parallel stream filter and design a special filter to identify the high-frequency chunks and obtain their frequency estimates. We also noticed that the bloom filter [31] was used in data dedup systems [6, 1] to memorize data chunks that have been observed. However, these works used only one bloom filter to tell whether a chunk is observed or not.

Metadata overhead is a big concern in chunking based data dedup [32]. Kruus et al. [1] presents a work similar to ours. Provided chunk existence knowledge, they propose a two-stage chunking algorithm that re-chunks transitional and non-duplicated big CDC chunks into small CDC chunks. The main contribution of their work is that they are able to reduce the number of chunks significantly while attaining the same duplicate elimination ratio as a baseline CDC algorithm does. Alternative work to reduce metadata overhead is to perform local duplicate elimination within a relatively large cluster of data. Extreme binning [33] detects file level content similarity by a representative hash value, and achives a performance that is comparable to the CDC's performacne on datasets with no locality among consecutive files. Another recent work [7] presents a sparse indexing technique to detect similar large segments within a stream.

2.3 Motivation

In this section, we define the data dedup gain as a general metric to measure the effectiveness of dedup algorithms. Based on this metric, we point out the problems in the popular CDC algorithm that hinder it from achieving a better dedup gain. This analysis motivates the proposed FBC algorithm in Section 2.4.

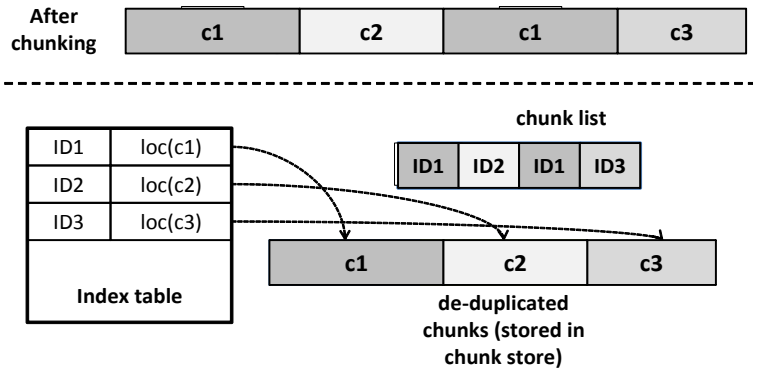

Figure 2.1: Data structures related to chunking dedup

2.3.1 Data Dedup Gain

Let s denote the byte stream that we want to apply the dedup algorithm for. Figure 2.1 shows the results of data dedup using chunking-based algorithm. After a specific chunking algorithm A (e.g., CDC), s is divided into n non-overlapping data chunks $s = \{c_1, c_2, ..., c_n\}$. We then assign each unique chunk an ID (practically, SHA1[5] hash algorithm is used to produce the ID of a chunk based on its content), denoted as $ID(c_i)$. The data stream s is mapped to a chunk list $\{ID(c_1), ID(c_2), ..., ID(c_n)\}$. This is used to support file retrieval from the stored chunks. We store the deduplicated chunks in a chunk store and use an index table to map chunk hashes to the stored chunks. Apparently, the benefit from dedup happens when we only store one copy of the duplicated chunks (e.g., chunk c_1 in Figure 2.1). However, we also need extra storage space for storing the index table and the chunk list, which is referred to as the metadata overhead. Metadata is a system dependent factor that affects the duplicate elimination result. The final gain from applying the chunking-based dedup algorithm, denoted by data dedup gain, is the difference between the amounts of duplicated chunks removed and the cost

of the metadata.

Denote the set of distinct chunks among these n chunks to be $D(s)_A = distinct\,\{c_1, c_2, ..., c_n\}$. Let the set size be m, i.e., $m = |D(s)|_A$. We now formally define the data dedup gain on a data stream s using a specific chunking based dedup algorithm A as follows:

$$gain_A(s) = foldingfactor(s) - metadata(s) \tag{2.1}$$

$$foldingfactor(s) = \sum_{c_i \in D(s)_A} |c_i| \cdot [f(c_i) - 1] \tag{2.2}$$

The first term in eq. 2.1 is called *folding factor* which stands for the gain by removing the duplicated chunks. Because we only need to store one copy in the chunk store for each unique chunk and remove the rest copies, the folding factor defined in eq. 2.2 is calculated as the sum of the products of the number of repeated copies $[f(c_i) - 1]$ and the length of that chunk ($|c_i|$) for every $c_i \in D(s)_A$. Remember, we denote $f(c_i)$ the frequency of chunk c_i.

The product of the chunk frequency and chunk size determines how much a distinct chunk contributes to $gain_A(s)$. In the extreme case where $f(c_j) = 1$, chunk j contributes nothing to $gain_A(s)$.

The second term in 2.1 is called the metadata overhead, which can be expressed as $|index| + |chunklist|$, the sum of the length of the on-disk chunk index and the size of the chunk list. The Index table requires at least $m \cdot log_2 m$ where $log_2 m$ is the minimal length of a chunk pointer that points to a unique chunk. We then rewrite eq. 2.1 as follows:

$$gain_A(s) = \sum_{c_i \in D(s)_A} |c_i| \cdot [f(c_i) - 1] - (|index| + |chunklist|) = \sum_{c_i \in D(s)_A} |c_i| \cdot [f(c_i) - 1] - (\frac{1}{8} m \cdot log_2 m + 20n) \tag{2.3}$$

We assume 20-byte SHA1 hash is used to represent a chunk ID and we divide $m \cdot log_2 m$ by 8 to convert the bit unit to a byte unit. It is worth noting that the index table size we considered here is a theoretically lower-bound, so the corresponding $gain_A(s)$

we calculate is an upper-bound of the dedup gain. (i.e., any real data dedup system cannot gain more than $gain_A(s)$.) Our data dedup gain definition assumes a chunking based data dedup model on top of the file system layer. Thus, the metadata overhead mentioned in this chapter should not be confused with the file system metadata overhead such as *inode*, etc.

2.3.2 Discussion on the Baseline CDC Algorithm

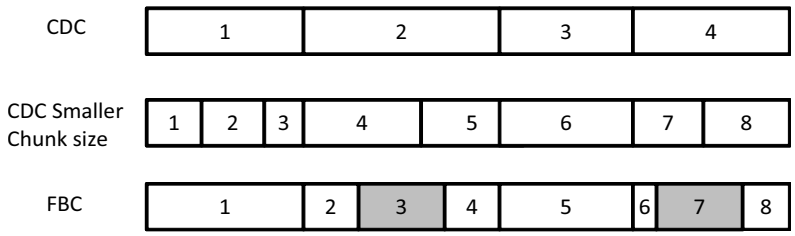

Figure 2.2: Illustration of Chunks Produced by the CDC and the FBC Algorithms

From the formula for the data dedup gain, given a fixed number of chunks, a good chunking algorithm is expected to generate more frequent (and longer) chunks to maximize the dedup gain. Being aware of this, we study the baseline CDC chunking algorithm.

CDC algorithm chops the data stream into chunks based on its local content that meets a certain statistical condition. This condition behaves like a random sampling process. For example, Figure 2.2(top) illustrates a data stream after applying the CDC algorithm. In order to increase the dedup gain, the only way for the CDC algorithm is to reduce the chunk size so that the algorithm could find more repeated (also smaller) chunks in the stream. Figure 2.2(middle) shows the results of applying a fine grained CDC algorithm by halving the average chunk size. Because of the randomness in chunk splitting, the CDC algorithm cannot guarantee the frequency of its produced chunks. Instead, these small chunks are more likely to have low frequencies. Consequently, the increase in the metadata cost may diminish the gain due to the removal of the redundant chunks.

However, assuming that we have the knowledge of the appearing frequency associated with each chunk (of a fixed length) in the data stream, we could design a much better

chunking scheme. As illustrated in Figure 2.2(bottom), after applying a coarse grained CDC algorithm, it performs a second level (fine-grained) chunking by identifying the chunks with a frequency greater than a predefined threshold. The original Chunks 2 and 4 are divided into three smaller chunks, where the new Chunks 3 and 7 are of high frequencies. These frequent chunks provide us with a better dedup gain. In addition, as we shall see later, after extracting these frequent chunks, the remaining ones (smaller segments) have a frequency distribution similar to that of the chunks produced by the fine-grained CDC algorithm.

Although such a chunking method, denoted by the Frequency Based Chunking scheme (FBC), sounds appealing, it makes a strong assumption on the knowledge of the frequency of the individual chunks. To solve this problem, we propose a statistical frequency estimation algorithm to identify the high frequency chunks with a low computation overhead and low memory consumption. In the following section, we introduce the proposed FBC algorithm in detail.

2.4 FBC Algorithm

In this section, we first present an overview of the proposed FBC algorithm. We then introduce the statistical frequency algorithm for identifying the high frequency chunks. In addition, we present counting and bias correction methods. At the end of this section, we discuss our two-stage chunking algorithm.

2.4.1 Overview of Frequency-Based Chunking Algorithm

In this section we describe the frequency-based chunking method, a hybrid chunking algorithm used to split the byte stream based on chunk frequency. Conceptually, the FBC algorithm consists of two steps: chunk frequency estimation and chunking. In the first step, the FBC identifies fixed-size chunk candidates that have a high frequency. The chunking step further consists of coarse-grained CDC chunking and a second-level of (fine-grained) frequency based chunking. During the coarse-grained chunking, the FBC applies the CDC algorithm to chop the byte stream into large-sized data chunks. During the fine-grained chunking, the FBC examines each coarse-grained chunk produced by the CDC algorithm to search for the frequent fixed-size chunk candidates (based on

the chunk frequency estimation results) and uses these frequent candidates to break down the CDC chunks further. If a CDC chunk contains no frequent chunk candidates, the FBC outputs it without re-chunking it. It is worth noting that the coarse-grained chunking step is essentially performing a "virtual chunking" in the sense that it only marks the chunk cut-points in the byte stream. The real cut and the SHA1 calculation only occurs when the FBC outputs a chunk. In the following section, we discuss each step in detail.

2.4.2 Chunk Frequency Estimation

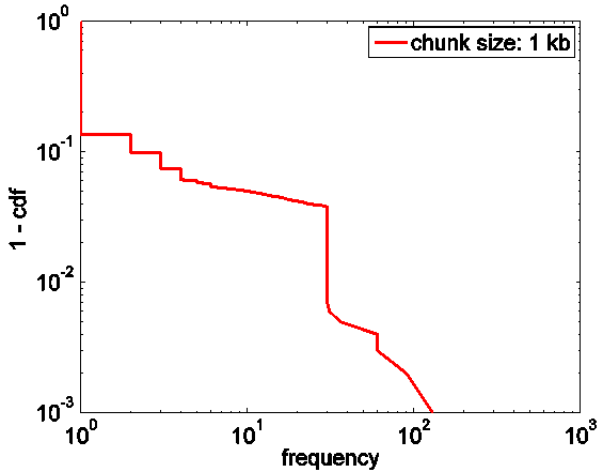

Figure 2.3: Log-log plot for the chunk frequency distribution

Due to the large size of the data stream, directly counting the frequency of each chunk candidate is not feasible. For instance, assuming that we use a sliding window to retrieve all the 4KB length chunk candidates in a 1GB data stream (we shift the sliding window one byte at a time to obtain a new chunk candidate) and the average frequency of the chunk candidates is 4, then there are around 268 million distinct chunk candidates ($1GB/4$). The process of directly counting the frequencies for such a large number of chunk candidates is apparently not scalable. However, one may notice that the FBC algorithm only requires the knowledge of the high-frequency chunk candidates. Figure 2.3 illustrates the log-log plot for the chunk frequency distribution. These chunks are of the size of 1KB from the dataset *Nytimes* (see Section 2.5 for the details of the

dataset). One interesting observation is that a majority of the chunk candidates are of low frequencies. For example, over 80% of the chunks only appear once, while fewer than 5% appear more than 10 times in the stream. Similar results are observed in all other chunk sizes.

This infrequency in the chunk candidates is of little interest to us. However, they do consume most of the memory during a direct counting process due to the large number of infrequent chunk candidates. To address this problem, in this chapter we propose a statistical frequency estimation algorithm to identify the high frequency chunks. The basic concept is to apply a filtering process to eliminate as many low-frequency chunks as possible and hence reserve resources to obtain an accurate frequency estimation for the high frequency chunks.

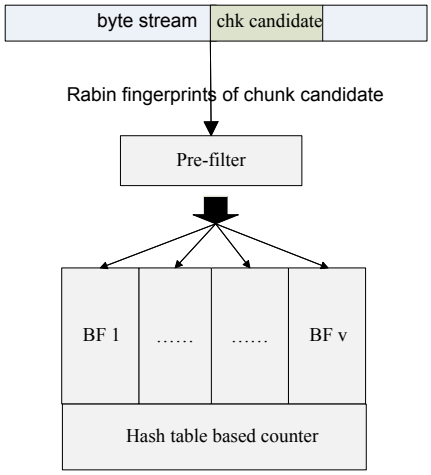

Figure 2.4: Overview of frequency estimation algorithm

The chunk frequency estimation algorithm consists of three components: pre-filtering, parallel filtering and frequency counting. The first two steps aim at reducing the low frequency chunk candidates by as many as possible, while the last step counts the occurrence of high-frequency chunk candidates. Figure 2.4 displays a schematic view of the frequency estimation algorithm. In the next section, we will discuss the implementation of each component.

Pre-filtering

We use a sliding window to extract all fixed-size chunk candidates from the target byte stream. Prior to starting to filter the low frequency chunk candidates with the parallel filter, we apply a pre-filtering step to eliminate the overlapping chunks drawn from the sliding window. For example, suppose there is a data segment of the size of 2KB which appears 100 times in the byte stream. Assume we use 1KB as the sliding window size. As such, this 2KB segment will result in $1,024$ overlapped chunk candidates of the size of 1KB with each having a frequency 100. To avoid deriving the overlapped chunk candidates, we apply a pre-filtering process, a modulo-based distinct sampler with a sampling rate r_0. A chunk instance has to pass the pre-filtering process first before entering the next parallel filtering step. Such a pre-filtering step only requires one XOR (exclusive or) operation for the filtering decision per observed chunk candidate. Essentially, r_0 controls the density of the sampling chunk candidates in the stream. (e.g., $r_0 = \frac{1}{32}$ means a chunk candidate could pass the prefilter if its Rabin's fingerprint [34] value $modulo\, 32 = 1$.)

Parallel Filtering

The parallel filtering is motivated by the idea of dependence filtering proposed in [30]. Before the frequency estimation process begins, the set of v bloom filters must be empty. For each survived chunk candidate out of pre-filtering process, we first check the appearance of the chunk candidate in all of the bloom filters. If we find this chunk in each of the bloom filters, we let it pass through the parallel filter and start counting its frequency. Otherwise, we randomly record the chunk in one of the v bloom filters. Apparently, without considering the false positive caused by the bloom filters, only the chunk candidates with a frequency greater than v can possibly pass the parallel filtering process. Due to the efficiency of the bloom filter data structure that only consumes a small amount of memory, this approach helps to filter out the majority of the low frequency chunk candidates with a small amount of resources. In particular, let X be the number of occurrences before a chunk passes the parallel filtering process. As such, the expected number of X can be calculated as follows:

$$E(X) = v \sum_{i=i}^{v} \frac{1}{i} \tag{2.4}$$

For example, if we use three bloom filters to build the parallel filter ($v = 3$), a chunk is expected to occur 5.5 times in order to pass the parallel filtering process. When the number of bloom filters is increased to 5, then $E(X) = 11.4$. In general, using more bloom filters for the parallel filtering step can help reduce the more low frequency chunks. However, we need to strike a balance between the amount of low frequency chunks reduced and the resource consumed by the bloom filters. In our experiments, we choose $v = 3$, which works best in practice. We present our filtering and counting algorithm in Algorithm 1. Notice the input chunk candidate c_i represents the ones survived pre-filter. In addition, we set the number of bloom filters to be v and the corresponding $E(X)$ to be E. There is a rich amount of literature available for detecting high frequency items in a data stream. However, in our problem, the threshold for high frequency chunks is usually as low as 15. The classical heavy hitter detection algorithms do not scale under such a low threshold. One salient feature of the proposed parallel filtering algorithm is that the parallel stream filter is able to capture the entities with a not-so-high frequency. This can be achieved by the parallel filtering process since it is able to eliminate a majority of the lowest frequency chunks.

Counting and Bias Correction

After parallel filtering, for each survived chunk candidate (i.e., the chunk candidate has appeared in all of the bloom filters), we start to count its frequency. The counting process can be implemented using a simple hash table, in which each of the keys correspond to a chunk and its associated value represents the frequency of the chunk. Notice that our counting process does not start until a chunk candidate appears in all of the bloom filters. Hence a counting bias exists because a number of copies for a particular chunk are skipped during the parallel filtering process. This counting bias can be estimated as $E(X)$. In other words, after estimating the frequency for a particular chunk, we need to add $E(X)$ to the frequency value to correct the estimation bias.

Algorithm 1 Parallel filtering and frequency counting

```
parameters: chunk stream s, v, theta
output: chunks with frequency greater than theta
begin
1:for each c_i belongs s do:
2:      if all v bloom filters contains c_i then
3:              if hash table does not contain c_i then
4:                      add c_i to the hash table
5:                      freq(c_i):=E
6:              endif
7:              freq(c_i):=freq(c_i)+1
8:      else
9:              randomly add c_i to one of the v bloom filters
10:     end-if
11: end-for
12://output freqeunt chunks
13:for each c_i in hash table do
14:     if freq(c_i) > theta then
15:             output c_i
16:     end-if
17:end-for
```

Two-stage Chunking

The concept of combining the CDC algorithm into the FBC in its first-stage of chunking makes the FBC algorithm a hybrid two-stage chunking algorithm. It combines the coarse-grained CDC chunking process with fine-grained frequency based chunking.

Figure 2.5 illustrates the operation on a simple input byte-stream. Figure 2.5(a) depicts the duplicate/nonduplicate patterns in the byte-stream; Figure 2.5(b) represents the coarse-grained chunks produced by applying the CDC algorithm, further examined for fine-grained fixed-size frequent Chunks 2, 4, 8, 9 and 11 in Figure 2.5(c). Coarse (CDC) Chunks 4, 5 and 8 in Figure 2.5(b) without containing any frequent fixed-size chunks are output as individual chunks (Chunk 6, 7 and 13 in Figure 2.5(c)). In some cases, small but infrequent chunks may also be produced. For example, Chunk 6 in Figure 2.5(b) is re-chunked into frequent chunks 8 and 9 in Figure 2.5(c) while the remaining part of Chunk 6 in Figure 2.5(b) becomes Chunk 10 in Figure 2.5(c). However,

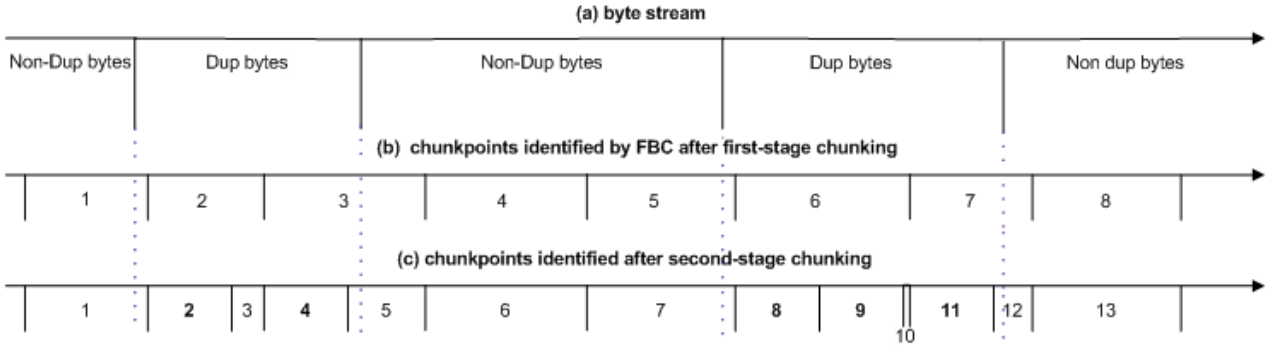

Figure 2.5: FBC chunking steps

throughout our experiments, we found that the impact of outputting the small chunks (e.g., chunk 10 in Figure 2.5(c)) is easily offset by that of outputting large sized (CDC) chunks.

In this chapter, we consider a fixed-size chunk to be frequent if it appears no fewer times than a predefined threshold theta (θ), determined based on domain knowledge of the datasets, including dataset content, dataset type, and number of backups contained in the dataset (if it is a backup dataset). The choice of θ reflects how aggressively the re-chunking should occur. For example, the smaller the θ, the larger the portion of the CDC coarse grained chunks that will be further re-chunked. On the other hand, each re-chunking brings in extra metadata overhead with a longer chunk list and possibly a larger index table.

2.5 Evaluation

In this section, we evaluate the FBC algorithm with extensive experiments with real data dedup workloads. We first introduce the datasets used in the evaluation and discuss the experimental settings. We then compare the dedup performance of the proposed FBC algorithm with the CDC algorithm in terms of the dedup gain and point out the major factors that contribute to the difference between these two algorithms.

2.5.1 Description of Data Dedup Datasets

We use three empirical datasets to evaluate the effectiveness and efficiency of the proposed FBC algorithm. The first two datasets, *Linux* and *Nytimes*, represent the workloads when the data streams exhibit a high degree of redundancy while the *Mailbox* dataset is characterized by a low detectable redundancy.

Nytimes is a web content dataset, which contains 30 days of snapshots of the New York Times website (www.nytimes.com) from June 11th to July 10th, 2009. The *Nytimes* dataset includes a collection of text, images, binaries and video clips. This data was collected using crawling software *wget*, configured to retrieve web pages from the website recursively for a maximum depth of two. We store the web pages in 30 tar balls, one for each day. The total data stream size is 0.92GB. The dataset *Linux* contains the kernel source files of 10 consecutive versions of the *Linux* operating system, from Linux 2.6.30.1 to Linux 2.6.30.10. The size of the *Linux* is 3.37GB. *Mailbox* is a company mail repository used by a group of software engineers. *Mailbox* is treated as a byte stream of 0.48GB in size.

2.5.2 Description of Hardware Platform and Software Implementation

We have implemented both the CDC and the FBC algorithm with GNU C++. We also implemented the parallel bloom filter using Python 2.6. All experiments are carried out on a standard Linux build 2.6 SMP x86_64 platform. The machine has two 2.0GHz cores. The memory size is 2GB.

2.5.3 Evaluation Criteria

We are interested in evaluating the performance of the proposed chunking algorithm, thus we do not compress any resultant chunks after the chunking process is complete. We apply a widely adopted metric 'Duplicate Elimination Ratio' (DER) for evaluation purposes. However, the DER is commonly calculated as the input stream length divided by the sum length of all output chunks after the dedup ($DER = \frac{input\ stream\ length}{sum\ lengths\ of\ all\ output\ chunks}$), without considering the metadata overhead. As we have pointed out in the previous sections, in practice, the metadata overhead is a key factor that affects the duplicate elimination result. A special feature of the proposed FBC

algorithm is that it is designed to explicitly reduce the metadata cost. To this end, we define a new version of the DER metric by taking into account the metadata overhead.

Recall that from eq. 2.3, the metric $gain_A(s)$ represents the amount of storage space saved by applying the dedup algorithm A on the data stream s, after subtracting the cost of the metadata. Denote the length of stream s as $|s|$. Now we define the new DER metric DER_{meta} as follow:

$$DER_{meta} = \frac{|s|}{|s| - gain_A(s)} = \frac{|s|}{\sum_{c_i \in D(s)} |c_i| + (|index| + |chunklist|)} \tag{2.5}$$

Another important metric we used is the *Average Chunk Size* (ACS), defined as the data stream size divided by the total number of produced chunks after applying the chunking algorithm. The total number of produced chunks is inversely proportional to ACS, and hence, the storage overhead of the metadata required to decribe the chunks (chunk-list) is also inversely proportional to the ACS.

We evaluate the FBC algorithm in both the DER and ACS factors so as to see how much advantage the FBC could gain when compared with the baseline CDC algorithms, in terms of the DER and the ACS.

Another reason for considering the ACS as a metric is that it is directly related to the operation cost. Fewer chunks mean less disk accesses, which further implies fewer I/O transactions. In addition, under a networked environment, fewer chunks lead to less network queries and possibly less network transmissions. Lastly, a shorter file reconstruction time should be expected for a bigger ACS. Given a specific DER, we hope to maximize the average chunk size.

We realize that such a comparison may be a little unfair because the FBC requires an extra pass of the byte stream. However, we believe that such a cost is deserved. First, the FBC requires a highly efficient frequency estimation algorithm which entails a very small memory footprint to obtain the desired accuracy. In addition, the estimation speed is comparable to the chunking process. Since in the second-stage the chunking algorithm requires examining at least a portion of the chunks produced in the first-stage, we plan to further reduce the frequency estimation to a smaller portion of the stream in our future work.

2.5.4 Tuning the Parameters

We next discuss the parameter settings for our FBC algorithm in the following experiments. We will group the parameters used in different steps.

Pre-filtering step uses sampling rate r_0 to determine the sampling density. By conducting extensive tests on several of the datasets, we choose r_0 to be $modulo\,32$ (i.e., the prefilter only lets a chunk candidate to pass if its Rabin's fingerprint $modulo\,32$ equals 1). We argue that this value strikes a balance between the size of the identifiable duplicated data and the degree of overlapping for all of the identified chunks. The sliding window size for the pre-filtering step is set with respect to the specific datasets which we will subsequently discuss.

Parallel filtering step takes on the parameters such as the number of bloom filters v, the size of the bloom filter and the chunk candidate frequency threshold θ. As mentioned previously, we set $v = 3$ and the size of each bloom filter to be 0.8MB. We adopt the same parallel bloom filter setting through all the three datasets. The choice of the bloom filter size is chosen according to the spatially optimal bloom filter setting [31] as well as the specific chunk frequency distribution of the datasets. For future work, we may adopt the idea of our work Forest-structured Bloom Filters [35] to accommodate the different data streams of dynamic lengths. We note that when the dataset is highly redundant, most chunks will pass the parallel filtering process. One solution in this case is to apply another layer of random sampling of rate r_1 before a chunk (hash) is inserted into one of the bloom filters. In this way, the output frequency is the frequency estimate obtained from the counting process divided by r_1. The parameter θ is set to 5 in the following experiments.

Two-stage chunking step: the CDC algorithm used in the first stage of the FBC takes on parameters including the minimum and maximum chunk sizes and the expected chunk size. From extensive experiments, we have observed that setting the minimum and maximum chunk sizes to 1Byte and 100KB respectively and varying the expected chunk sizes among 0.25KB, 0.5KB, 1KB, 2KB, 4KB and 8KB under each run, the resulting average chunk size is very close to the expected chunk size. Upon further examination of the distributions of the chunk sizes, it is demonstrated that none of the chunk distributions is pathological (i.e., a significant portion of chunks with a size close to 1.). Hence we apply a 1Byte/100KB minimum/maximum chunk size setting to both

the CDC and the FBC algorithms in all of our experiments. In the meanwhile, the CDC algorithm employs its own 64-byte length sliding window to generate the chunk cut-points.

Since the FBC algorithm chunking step consists of two stages, the chunk-stage-ratio of the expected chunk size produced by the CDC algorithm in relation to the frequent chunk sizes is important. Through extensive experiments on multiple datasets, it was determined that when the chunk-stage-ratio equals 16, it yields the best result. We adopt this ratio value in all of the following experiments.

2.5.5 Experimental Results

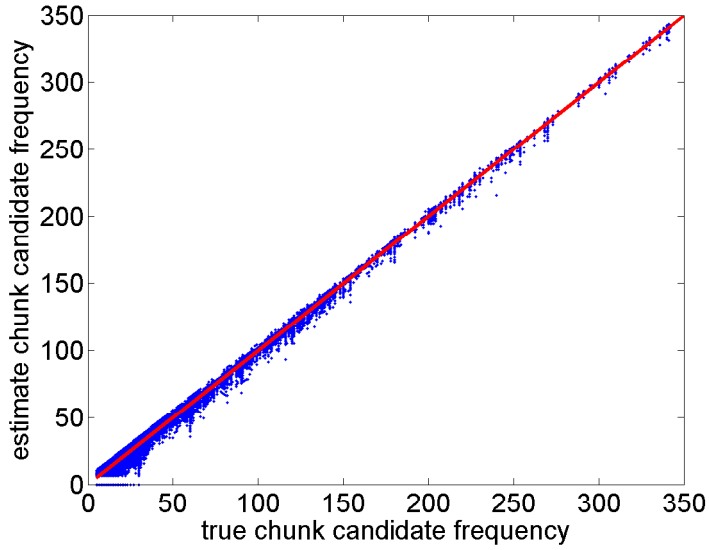

Figure 2.6: Accuracy of the chunk frequency estimation

Chunk candidate frequency estimation: major concerns regarding our candidate frequency estimation include its accuracy and the memory consumption. We examine the parameter settings for our parallel bloom filter through multiple runs on the *Nytimes* dataset. Figure 2.6 plots the estimated chunk candidate frequency against the true chunk candidate frequency under parameter setting $v = 3$; the bloom filter size equals 0.8MB. The chunk candidate frequency threshold is $\theta = 5$. We see that the estimation result is close to the true frequency values, even at the very small filter sizes

(i.e., $3 \times 0.8MB = 2.4MB$). All of other datasets also demonstrated similar results.

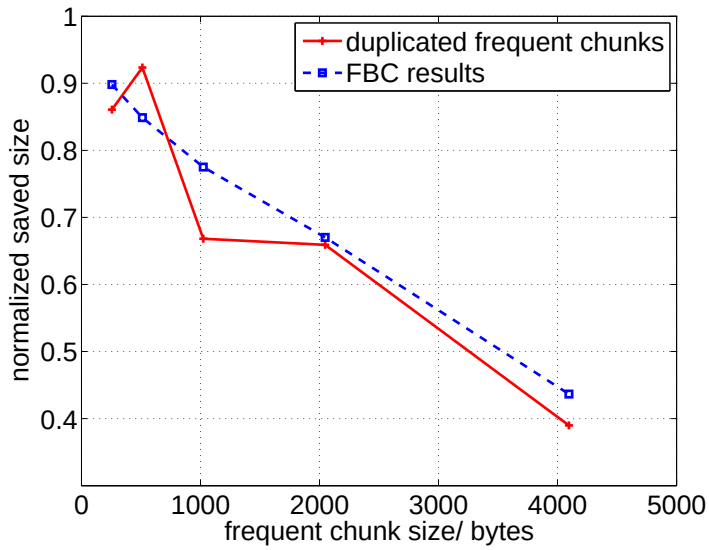

Figure 2.7: Frequent chunk sizes vs. the FBC result

Correlation between frequent-chunk size and total saved size: It is interesting to see the correlation between the frequent-chunk size and its impact on the total saved size. We assume $\theta = 5$ (i.e., all chunks with a frequency of no less than 5 are considered to be frequent.) and fix the expected chunk size produced by the CDC algorithm used in the FBC algorithm to be 8KB. Figure 2.7 plots the saved size vs. the frequent-chunk size used for *Nytimes*. The solid curve indicates the total size (normalized to the size of dataset *Nytimes*) for all detected duplicated frequent chunk candidates (overlapping chunk candidates are removed) while the dashed curve indicates the FBC result. The former is essentially an indicator of the saved size that the FBC algorithm achieves, since it demonstrates how much duplication could be discovered with the corresponding frequent chunk sizes.

We see the positive correlation between the solid and dashed curves, validating that the FBC algorithm does exploit the gain of frequent chunks. Furthermore, it was observed that as the size of the frequent chunks grows from 128byte to 4KB, the saved size (shown on the y-axis) decreases for both curves, indicating that choosing small frequent chunks could result in more duplication being detected.

Comparing results for the FBC algorithm and the CDC algorithm: We

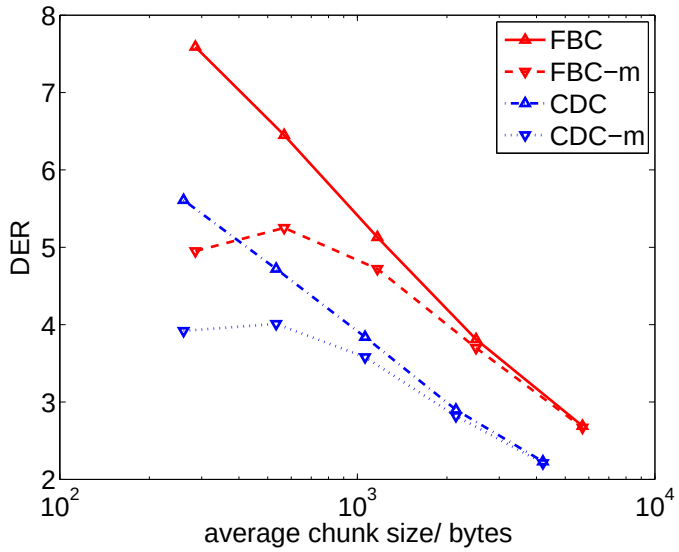

Figure 2.8: Baseline CDC vs. FBC on Linux

compare the FBC and the CDC with respect to the DER and the average chunk size on both the *Linux* and the *Nytimes* datasets. Figure 2.8 illustrates the results on the *Linux* dataset, which summarizes 5 runs for each algorithm. Figure 2.9 shows a similar result on the *Nytimes* dataset, which summarizes 6 runs for each algorithm. More specifically, we compare the baseline CDC and the FBC algorithm under a wide range of chunk sizes: (1) On the *Linux* dataset, we run the CDC with 5 expected chunk sizes of 0.25KB, 0.5KB, 1KB, 2KB and 4KB. We then run the FBC with 5 frequent chunk sizes 0.5KB, 1KB, 2KB, 4KB, and 8KB; (2) On the *Nytimes* dataset, we run the CDC with 6 expected chunk sizes 0.125KB, 0.25KB, 0.5KB, 1KB, 2KB, and 4KB. We also run the FBC with 6 frequent chunk sizes 0.25KB, 0.5KB, 1KB, 2KB, 4KB and 8KB. As we have fixed the chunk-stage-ratio of 16 through all runs with the FBC algorithm, the expected chunk size used in the CDC chunking stage is implicitly determined by the corresponding fixed-size frequent chunk size. We make these settings to assure both algorithms produce the same number of chunks in each comparison.

For each test, we collect two DER values, with respect to whether or not the meta-data overhead is considered. The solid and dashed curves (top 2 curves) represent the results for the FBC while the dash-dotted and dotted curves (bottom 2 curves) represent the results of the CDC. In addition, for either the FBC or the CDC, the curve with the

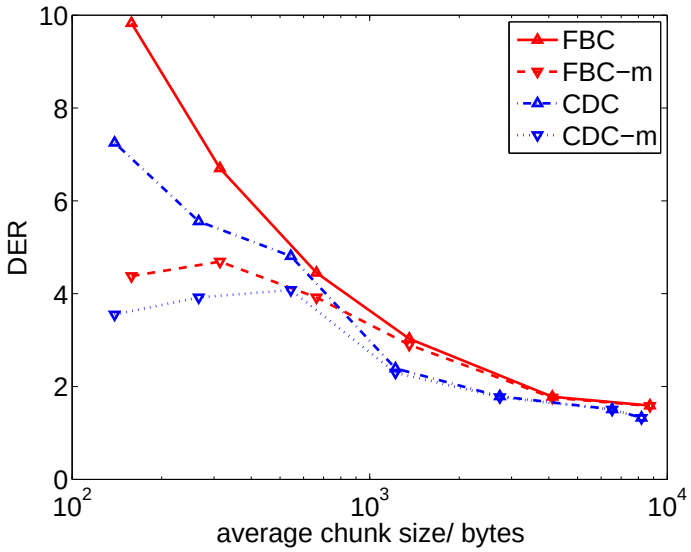

Figure 2.9: Baseline CDC vs. FBC on Nytimes

upward-pointing triangular marker represents the experimental results without considering the metadata overhead, while the curve with the downward-pointing triangular marker represents the results with the metadata overhead considered.

Several important observations could be made from both figures: (1) On the *Linux* dataset, as the average chunk size grows beyond 2KB, the DER that the FBC achieves is almost 1.5x that of what the CDC achieves with the same average chunk size. Moreover, the metadata overhead tends to be negligible as the average chunk size grows beyond 2KB. Thus, we conclude that our FBC algorithm is expected to perform better than the CDC algorithm when large chunk sizes are desired. On the *Nytimes* dataset, the FBC outperforms the CDC algorithm, but the difference is less than that on the *Linux* dataset; (2) For both the FBC and the CDC algorithms, as the average chunk size increases, the result with the metadata overhead considered is getting close to the result without the metadata overhead considered. This observation essentially justifies the preference of the large chunks over small chunks for many chunking based deduplication systems; (3) If we do not consider the metadata overhead, the DER increases monotonically as a smaller chunk size is chosen for both the FBC and CDC algorithms. On the other hand, when the metadata overhead is taken into account, the FBC algorithm performance peaks at the frequent chunk sizes of 1KB and 0.5KB on *Linux*

and *Nytimes* respectively, whereas CDC algorithm performance peaks at the expected chunk size of 0.5KB uniformly on both datasets. These results illustrate that we cannot arbitrarily reduce the chunk size in both the CDC and the FBC algorithms to maximize the DER while considering the metadata overhead, since when the chunk size is too small, the gain loss due to the metadata cost outgrows the gain increase of the identified redundant data. (4) If we view the benefit of the FBC in terms of the number of chunks produced, then an even more significant result can be obtained. This is easy to see from Figure 2.8 and Figure 2.9. Notice that the x-axis in each figure is in log scale. For example, in Figure 2.8, the FBC algorithm with an average chunk size close to 2KB has the same DER value of 4.0 (considering the metadata overhead) as the CDC algorithm with an average chunk size as small as 0.5KB. This is a 4x improvement on the chunk size. We see a similar improvement in the *Nytimes* dataset as well. We see in Figure 2.9 that to achieve the same DER of 3.9 (with/without considering the metadata overhead), the FBC produces chunks of an average chunk size of 663 bytes while the CDC produces chunks with an average chunk size of 266 bytes. Therefore, the FBC enjoys a 2.5x improvement on chunk size. We also examine the results on the *Mailbox* dataset. However, we find that both the CDC and the FBC algorithms perform equally poorly. For example, even if the expected chunk size is chosen to be as small as 0.25KB, the DER for the CDC and the FBC algorithm is only 1.5, which is much worse than what has been achieved (7.8 and 10) on *Linux* and *Nytimes* datasets with the same expected average chunk size. Further investigation into the *Mailbox* dataset illustrates that it primarily consists of very short messages that are carbon-copies from different engineers. As such, the duplicated data pieces are too small to be good candidates for data dedup. Such datasets essentially reveal the limitations of all chunking based algorithms which otherwise require different types of data dedup methods or data compression approaches to apply.

2.6 Conclusions and Future Work

In this chapter we proposed a novel chunking algorithm, the Frequency-Based Chunking algorithm (FBC), for the data deduplication. The FBC algorithm explicitly made use of the chunk frequency information from the data stream to enhance the data dedup

gain, especially when the metadata overhead was taken into consideration. To achieve this goal, the FBC algorithm first utilized a statistical chunk frequency estimation algorithm to identify the globally appearing frequent chunks. It then employed a two-stage chunking algorithm to divide the data stream, in which the first stage applied the CDC algorithm to obtain the coarse-grained chunking results and the second stage further divided the chunks produced by the CDC algorithm with the identified frequent chunks. We conducted extensive experiments on heterogeneous datasets to evaluate the effectiveness of the proposed FBC algorithm. In all experiments, the FBC algorithm persistently outperformed the CDC algorithm in terms of achieving a better dedup gain or producing a smaller number of chunks. More specifically, it produces 2.5 – 4 times fewer chunks than what a baseline CDC algorithm does to achieve the same DER. Another benefit of the FBC over the CDC is that the FBC with the average chunk size greater than or equal to the CDC achieves up to 50% higher DER than what the CDC can do. As mentioned in the end of section 2.4.2, a given frequency threshold θ determines how aggressively the re-chunking should be carried out. In fact, further thinking shows that if some coarse-grained chunks produced by the CDC algorithm already appear to be high frequent (i.e., its frequency substantially exceeds the average chunk frequency), it may be desirable to skip re-chunking those chunks because re-chunking them may result in no gain but increase in the metadata overhead.

In the future, we plan to refine our FBC algorithm by setting the escape frequency for coarse grained chunks produced by the CDC algorithm. Also, our existing approach requires an extra pass on the dataset to obtain frequency information. We plan to design a one-pass algorithm, which conducts the frequency estimation and the chunking process simultaneously. An intuitive idea is to track top k frequent segments, instead of tracking all segments with a frequency greater than a certain threshold. This tracking can be integrated into the FBC algorithm to make an on-line frequency estimation.

Chapter 3

Forest-structured Bloom Filter with Flash-Memory

3.1 Introduction

A Bloom Filter is a bit vector data structure that compactly represents a set of elements (keys). It supports key insertion and membership queries. A membership query is answered by a BF with a certain chance of a false positive (i.e., an affirmative answer to a nonexistent element in the set). In other words, if a BF answers affirmatively with regard to the membership query for a key, this answer may be incorrect. On the other hand, if a BF answers negative to the membership query for a key, it is guaranteed that the key is NOT present. To keep the false positive probability low, the BF size should be a factor of multiple times (e.g., 8, 16, etc.) to the maximum number of keys inserted. For instance, for a maximum number of n keys, a BF size of $8n$ bits would bound the false positive probability to less than 2%. Traditionally, a BF is implemented as an in-RAM data structure. Hence, its size is limited by the available RAM space on the machine.

Initially all bits in the bit vector are set to 0. To insert a key in a BF, k independent hash functions are used to map the key into k different positions in the bit vector, each of which is set to 1. To query a key in a BF, it involves mapping the key into k ($k \geqslant 1$) different positions in the bit vector by k independent hash functions and checking whether each of these bits is 1. If it is, the BF concludes that the key is in the set that

it represents. Otherwise, it conclude that it is not. A false positive occurs during a query when a key not in the set is mapped to k bit positions which are already set to 1 during the insertions of other keys. Further discussions on false positive probability are presented in Section 3.4.4.

A common use of BFs in applications like chunking based data dedup is to identify nonexistent keys so as to avoid high latency caused by looking up a disk-based index for that key. For example, to reduce the frequency of disk access, Zhu et al. [6] adopts an in-RAM Bloom Filter to identify the nonexistent chunk hashes. If a chunk hash is identified as a nonexistent one, the respective chunk is temporarily stored in an in-RAM container without querying the on-disk chunk index. However, in some cases the size of the in-RAM BF is too big to fit in the available RAM. For instance, state-of-the-art dedup systems adopt BFs which consume 1GB RAM for each billion unique chunk hashes stored. Furthermore, when the dataset size could not be determined in advance, the overall space consumed by the BFs should be able to scale up to accommodate the growth of the dataset.

When RAM space is not sufficient, it may be necessary to store a BF into a secondary storage device, like a magnetic disk. However, operations on a BF are randomly spreading over its length. As a recent work [36] illustrates, the random I/O access (seek) time of the magnetic disk is so long (around 10 ms) that it dominates the overall cost and limits the processing speed. Flash-memory (or flash) has a much shorter access time (around $10-100$ μs) in currently available Solid State Drives (SSDs) because there is no mechanical movement for seeking in flash-memory. Thus, it is a good candidate for storing the Bloom Filters. Moreover, the unit price (dollar/GB) for flash-memory is much lower than that of RAM, making flash-memory economically viable as a secondary storage for storing BFs. Flash-memory performs relatively poor in random write operations, as compared with the reads and sequential writes and it exhibits several unique characteristics: (1) Data can be read/written by pages but can only be erased by blocks, where a block is typically $16-32$ times larger than a flash page; (2) a page write is slower than a page read (e.g., 25 μs vs 200 μs) and could not be performed until the block containing that page has been erased, which is multiple times slower (e.g., 1500 μs) than the page read/write operations; (3) the flash page in-place update (overwrite) is not supported as it would require an erase-per-update, resulting in severe performance

degradation; (4) each flash-memory cell allows for a limited number of erase operations to be performed during its lifespan (before the cell wears out). One goal of this chapter is to propose an efficient Bloom Filter design with flash-memory that considers these unique characteristics.

The general idea of building BFs with flash-memory is to utilize a limited amount of RAM space combined with a much larger flash-memory space to form a group of BFs so that the total capacity of BFs could go beyond the available RAM space constraint. Since the BFs are stored in flash-memory, key query/insert operations may trigger flash read/write accesses respectively. Therefore, it is important for our design to consider flash-memory characteristics and use precious RAM resources efficiently to reduce the flash-memory access time. For example, to amortize the relatively expensive flash page write operation, multiple key insertions could be buffered temporarily in the RAM and committed to flash-memory by using one flash write. Similarly, querying a key may require one or multiple flash read operations, depending on how many pages a BF spans in flash as well as how many BFs are required to be checked for a key query operation. Our design aims at minimizing the overhead for both the key query and the insertion so that the overall processing speed (i.e., the number of records processed per second) could be maximized.

Canim et al. [36] and Debnath et al. [37] separately proposed two similar BF designs with flash-memory. Both designs build a BF organization by preallocating flash space and partitioning it into Q small BFs, where each BF is dimensioned to the size of a flash page, e.g., 4KB. Similarly, the available RAM space is equally partitioned into Q buffers as well, one for each BF to access exclusively during the run. To query or insert a key e in the BF organization, a hash function is first used to locate the BF the key should belong to, followed by all k bit positions in that BF hashed by key e using k different hash functions to be checked or set. To insert a key into a given BF, the key will be buffered temporarily at its corresponding buffer. When one BF's buffer becomes full, the buffer is **flushed** to flash as follows: firstly, the associated BF is read into the RAM; next, the loaded BF merges the corresponding updates in the buffer; finally, the updated BF is written back to the flash-memory. We denote both designs as a **single-layer** BF designs in this chapter. This design is very efficient in key query operations, since it consumes just a single flash page read operation for a key query on

the BF organization. In the meanwhile, the number of block erases and page writes will be reduced by buffering. However, if the available RAM space is not big enough, the buffer size could become very small for each BF, resulting in a lot of buffer flushing operations during the run, each of which demands one flash page read and one flash page write.

On the other hand, Guo et al. [38] proposed a *dynamic bloom filter* design to represent a dynamic set instead of rehashing the dynamic set into a new filter as the set size changes. Their idea is to employ a series of s in-RAM BFs of the same size to represent a dynamic set. Among the set of BFs, only one is called *active* since it still allows new key insertions (i.e., allowing more bits to be set to 1) while the rest of the BFs in the series are called *full*. In the beginning, $s = 1$ and the initial BF is active. To insert a key e in the BF, this design uses k independent hash functions to hash e into k bit positions and set them to 1 in the active BF. This design appends a new BF as an active BF when the previous active BF becomes full. To query a key, it searches each BF in the series and returns affirmative when the key is found. It returns negative when the key is not found in any of the BFs in the series. Compared with a traditional single in-RAM BF design, the dynamic bloom filter design achieves capacity-on-demand and is explicitly optimized for representing a dynamic set. However, the maximum cardinality of the dynamic set represented by this design is still constrained by the available RAM space. To overcome the RAM size limitation, we modify the original dynamic bloom filter design to store all full BFs on the flash. Initially, only the active BF is instantiated in RAM to span the entire available space. For each key insertion, corresponding bit positions are directly set to 1 in the active BF. This process is identical to the insertion operation on a traditional in-RAM BF. Upon the BF is becoming full (i.e., the number of inserted keys reaches the maximum allowed value subject to a predefined false positive probability upper-bound), it is written to flash-memory and a new active BF of the same size is instantiated in the RAM. The same procedure repeats when the substitute becomes full again. Eventually, this design forms a chain of BFs with the newest instantiated one residing in the RAM. We denote this design a **linear-chaining** BF design, which has the minimum write operations, because each BF is written to flash only once and never gets modified after that. However, since each full BF on the flash may be searched for a given key, the key query performance would

potentially deteriorate very soon as the chain length increases. In order to minimize the flash page accesses during the key queries, we propose another modification to the original dynamic bloom filter design by localizing k bit positions associated with each key into a flash page (e.g. 4KB). This localization is achieved through another independent hash function that maps each given key to the address range of a certain flash page within a BF. In this way, each key query at a full BF on the flash will only need one flash page read, compared with as many as k flash page reads if the k bits are randomly located within the BF. See Section 3.2.2 for more details of both designs.

In this chapter, we present the design and evaluation of a Forest-structured Bloom Filter (FBF) that combines limited RAM space with a much larger flash space to dynamically adapt to the growth of a dataset. To mitigate the insertion performance bottlenecks, the FBF exploits two key design decisions: (i) in order to cope with a dynamic set, the FBF organizes BFs in a expandable forest-structure. Initially, the FBF allocates the first layer of the BF organization in RAM and divides the layer into a number of small BFs, each of the size of a flash page, so as to localize the bit positions associated with a key. More specifically, an independent hash function is used to map a given key to one of the BFs depending on the hash value of the key. As the set size grows and the first layer becomes full, all BFs in the first layer are then written to the flash-memory and the forest is naturally expanded by allocating a new layer of BFs in the flash-memory. The spared RAM space is then used to buffer the bit updates for BFs allocated at the expanded layer. Each BF at the expanded layer is a child of one of the BFs in its parent layer; and (ii) rather than owning an individual bit update buffer, each BF at one layer shares the RAM buffer space so that the buffer space of a certain BF could grow to hold new bit updates until the overall RAM buffer space is filled up. On the other hand, to improve the key query performance compared with the proposed linear-chaining design, the FBF design achieves much fewer flash reads per key query – let m be the total number of bits taken by the BFs within either design, the required number of flash reads for a key query in a b-branching forest (i.e., each BF has b children except those at the lowest layer) is $\mathcal{O}\left(log_b m\right)$ whereas that for the linear-chaining design is $\mathcal{O}\left(m\right)$.

The key contributions of this chapter could be summarized as follows:

1. To improve the flash access performance, we localize the bit positions associated

with each key into a flash-page sized BF.

2. We first propose a BF design called the linear-chaining BF to cope with a dynamic set and then improve it by proposing a multi-layer design called the forest-structure BF to better adapt the flexible number of key insertions with a much lower false positive probability, as well as to achieve a much smaller query-path-length when compared with the linear-chaining BF design.

3. We efficiently use the available RAM space for buffering the bit updates.

We use two real-world data traces taken from the representative BF applications to drive and evaluate our design. Our experimental results show that the FBF design outperforms the single-layer BF design on a broad range of RAM buffer sizes. In particular, the FBF achieves a 2 times faster processing speed (ops/sec) with 50% less flash write operations.

The rest of the chapter is organized as follows. Section 3.2 provides an overview of flash-memory and introduces two Bloom Filter designs with flash-memory. Section 3.3 presents the proposed FBF design. Section 3.4 discusses the major design issues and choices with respect to our FBF design. Section 3.5 presents an extensive experimental evaluation over our FBF design with two typical real workloads from the data dedup applications. Section 3.6 presents an overview of the related works and Section 3.7 draws some conclusions.

3.2 Overview of Bloom Filter Designs with Flash-Memory

3.2.1 Flash-based Storage Overview

Figure 3.1 illustrates a block-diagram of a NAND flash based SSD. In flash-memory, data is stored in an array of flash blocks. Each block spans $32 - 64$ pages, where a page is the smallest unit of the read and write operation. A distinguishing feature of the flash-memory is that the read operations are very fast, as compared to a conventional magnetic disk drive. Moreover, unlike the disks, random read operations are as fast as sequential read operations, as there is no mechanical head movement. A major drawback of the flash-memory is that it does not allow in-place updates (i.e., overwrite). Page

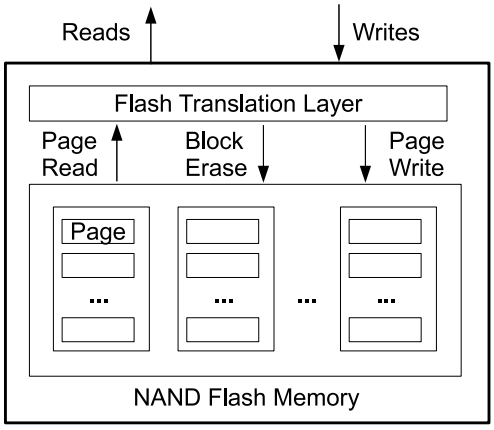

Figure 3.1: Flash-based Solid State Drive (SSD)

write operations in flash-memory must be preceded by an erase operation. Within a block, the pages need to be written sequentially. *The in-place update* problem becomes complicated as the write operations are performed in the page granularity, while the erase operations are performed in the block granularity. The typical access latencies for the read, write, and erase operations are 25 μs, 200 μs, and 1500 μs, respectively [39].

The Flash Translation layer (FTL) is an intermediate software layer inside the SSD, which makes the linear flash-memory device act like a virtual disk. The FTL receives logical read and write commands from the applications and converts them to the internal flash-memory commands. To emulate a disk like in-place update operation for a logical pageL_p, the FTL writes data into a new clean physical page P_p, and maintains a mapping between the logical pages and physical pages. It then marks the previous physical location of L_p as invalid for future garbage collection. Although the FTL layer allows the current disk based application to use SSD without any modifications, it needs to internally deal with the flash physical constraint of erasing a block before overwriting a page in that block. Besides the *in-place update* problem, flash-memory exhibits another limitation – a flash block can only be erased a limited number of times (e.g., 10K−100K) [39]. The FTL uses various wear leveling techniques to even out the erase counts of the different blocks in the flash-memory to increase its overall longevity [40]. Recent studies illustrate that current FTL schemes are very effective for workloads with sequential

access patterns. However, for workloads with random write patterns, these schemes show very poor performance [41][8][42]. One of the design goals of this chapter is to use flash-memory in a FTL-friendly manner in a BF design.

Small random writes need to update data within a page. Since a (physical) flash page could not be updated in place, a new (physical) page will need to be allocated and the unmodified portion of the data page needs to be relocated to the new page. Accordingly, the performance gap between the sequential and random writes on the flash is significant [37]: with 4KB IO request size, the IOPS (I/O operations per second) performance of the sequential write is around 3 times that of the performance of the random write. In addition, the IOPS performance of sequential/random reads is about 6 times that of sequential writes. A slight gap exists between the IOPS performance of sequential and random reads, probably due to the device internal prefetching mechanism.

3.2.2 Two Most Relevant Bloom Filter Designs with Flash-Memory

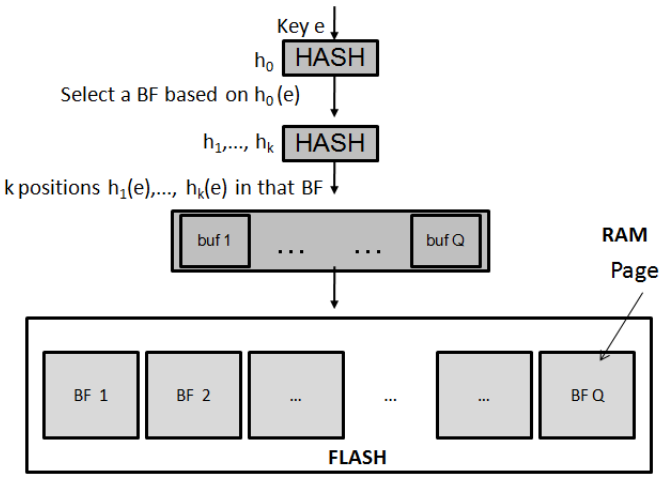

Figure 3.2: Single-layer BF on Flash

The single-layer BF design (Figure 3.2) pre-allocates the flash space for storing all BFs. In order to localize bit positions associated with each key into a single flash page, the single-layer BF design partitions the allocated flash space into Q small BFs, each of the size of one flash page (e.g., 4KB), and uses a hash function to locate the BF whom a given key e (for either query or insert) should belong to. Then, k different bit

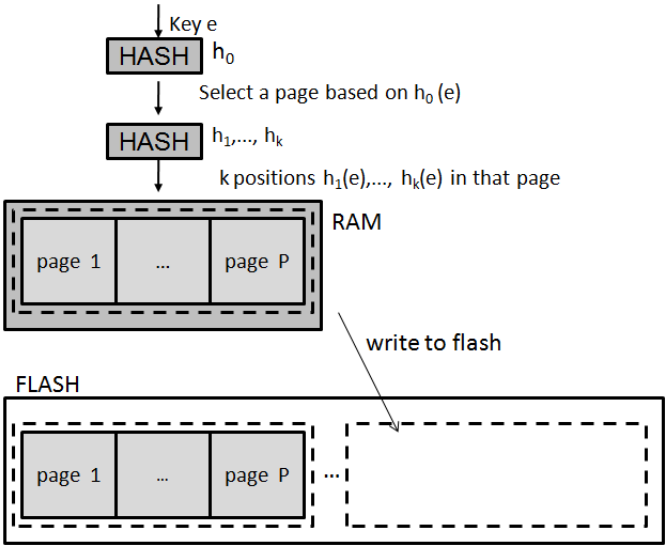

Figure 3.3: Linear-chaining BF on Flash

positions are selected to be checked or set within a BF by another k independent hash functions. Bit position localization is necessary, otherwise the k bit positions associated with a given key may be located in multiple flash pages. Consequently, it needs to take multiple flash page accesses (reads) to check these k bits. The single-layer BF design buffers updates (changes from 0 to 1 as a result of key insertion) in RAM for each of the BFs. Since in-place updates to logical flash pages increases garbage collection activity because of new physical flash pages being allocated and written, buffering BF updates help reduce the write amplification and garbage collection burdens. In the meanwhile, the key query performance could also be improved somehow since a bit position that is buffered in RAM can be read without reading the containing page in flash-memory. To implement this buffering strategy, the single-layer design groups and stores bit position updates for the same flash page in the RAM. Each update records a bit position offset within a page; all updates within a page have the same page id. The available RAM space is equally divided into Q small fixed size buffers, *with each buffer space being accessed exclusively by one BF throughout the run.*

Flushing the buffered updates per BF would cost a random flash page write. In order to leverage faster large (sequential) writes, instead of flushing individual pages, the single-layer design flushes a group of contiguous (logical) pages to flash-memory, where

the page group size can be dimensioned to the size of an erase block (e.g., 128KB). The flushing operation starts when the buffering space for a BF is depleted. At this time, the "dirtiest" page group that contains the largest number of bit position updates (i.e., the maximum number of changed bits) is selected for flushing to flash.

Two limitations exist in this buffering strategy. First, the filling rate deviates non-trivially from one BF buffer to another (due to using a single hash function to locate the BF where an incoming key should belong to), leading to the unbalanced case that one buffer has been filled up while the others still have room for updates to fill. Intuitively, to minimize the total number of write operations, the dirtiest page group should be chosen for flushing to flash-memory. However, the dirtiest page group selected by the single-layer design does not have each of its buffers filled up. This leads to a smaller number of updates being committed per flushing operation, which in turn increases the total number of write operations. Second, if the size gap between the available RAM and flash-memory is big (a reasonable assumption when using a Flash-memory for storing BFs is justifiable), the buffer size could become very small for each BF, resulting in each BF being flushed many times during the run.

The proposed linear-chaining BF design focuses on improving key insertion performance. As Figure 3.3 illustrates, the active BF is instantiated in RAM to span P pages. Each page is aligned to the size of a flash page. To localize bit positions associated with a given key, the linear-chaining design uses a hash function to select the flash page where the bits associated with the key should be located, followed by all k bit positions in that flash page hashed by key e using k independent hash functions to be checked or set. This design is optimal for key insertions because all k bit positions related to an insertion are directly set in RAM. It also maximizes the performance of the flash writes by converting random bit in-place updates into sequential (large) flash writes and guarantees that any (logical) flash page used in the chaining BF organization is to be written only once during the run. However, for a key query, a number of BFs in the chain may need to be searched. In the worst case (i.e., a negative key query), the number of searched BFs for a key query equals the total number of BFs in the chain (chain length). As each BF search demands one flash page read (except for the active one in RAM), the number of flash read operations per key query grows linearly with the chain length. Furthermore, the linear-chaining design suffers a significantly larger false

positive probability than the single-layer BF does due to a much larger number of BFs searched in each key query. This disadvantage is easy to understand, since intuitively, each BF searched during a key query can be a false positive. The effective false positive probability is at least proportional to the expected number of searched BFs per key query.

3.3 Forest-structured Bloom Filter on Flash

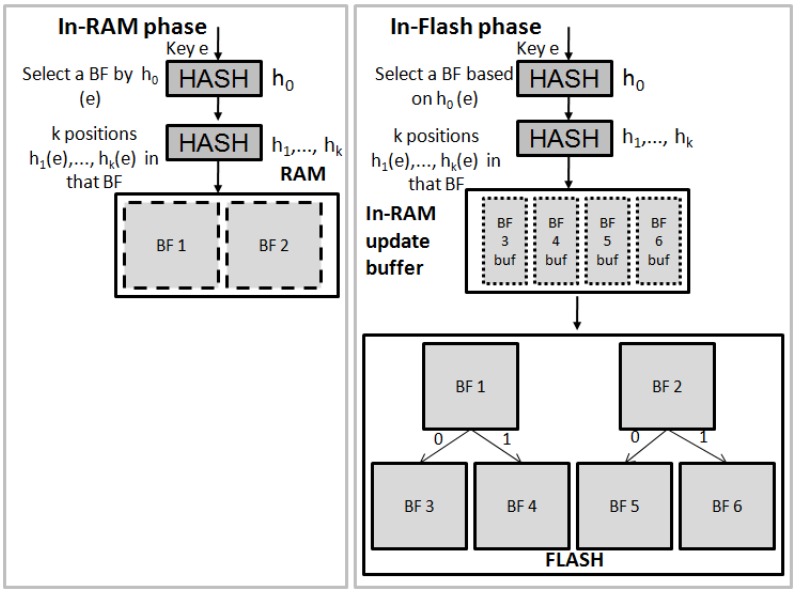

Figure 3.4: Forest-structured Bloom Filter design

Our FBF design is driven by the objectives to overcome the limitations of the existing single-layer BF design and the proposed linear-chaining BF design. In particular, in contrast to the single-layer BF design that could only handle static sets (i.e., the sets that have the maximum number of unique keys predetermined.), one primary goal of the FBF design is to cope with dynamic sets (i.e., the sets that have their sizes increasing and unable to be determined in advance). To this end, the FBF organizes bloom filters into a type of *dynamic-expanding* data structure, which easily adapts to the growth of a set by adding a new layer to the existing structure. Compared with the linear expansion in the proposed linear-chaining BF design, the FBF advances one step further

by organizing bloom filters into a forest structure which could be expanded easily by adding a new layer of BFs to the existing structure. This forest structure explicitly optimizes key query performance by bounding the maximum number of BFs searched during the query to the forest height $\mathcal{O}(log_b m)$ where m is the total number of bits occupied by all BFs and FBF organizes BFs in a b-branching forest. In contrast, the maximum number of BFs checked in the linear-chaining design is $\mathcal{O}(m)$. Furthermore, the Forest structure design achieves a signifiantly lower false positive probability than the linear-chaining design does for the sake of less searched BFs per key query.

In order to localize bit positions associated with a given key, the FBF divides the BF organization at each forest layer into many small BFs, each of the size of a flash page, and locates one BF for a given key query/insert using a hash function.

While this chapter focuses on the FBF design with flash-memory, it is worth noting that its contribution is not limited to a flash-memory environment. In fact, with a pure RAM implementation, the FBF design is expected to achieve a multiple times higher key query speed (ops/sec) compared with our proposed linear-chaining BF design with a multiple times lower false positive probability. In addition, the FBF design is able to cope with dynamic sets, which is beyond the capacity of a single-layer design. In the rest of this section, we describe our FBF design in detail.

The FBF constructs a forest-shaped BF organization with BFs of flash-page size, typically 2KB or 4KB. To cope with dynamic sets, the FBF adaptively expands the BF organization by adding one new layer at a time, making a single-layer evolve into a multi-layer forest. Except for those BFs at the leaf layer (the lowest layer), each BF at a non-leaf layer has b children ($b \geqslant 2$, a constant factor controlling the forest shape). This design does not require statically pre-allocating the entire BF organization on a flash-memory before the run. In particular, the first layer of the forest is directly allocated in RAM to leverage the superior access performance of the RAM. Thus, the number of BFs in first layer of the forest, denoted by N_1, equals the available RAM size divided by the flash page size. As shown in the in-RAM phase in Figure 3.4, to query/insert a key e, a hash function h_0 is used to locate the BF in the first layer to whom key e should belong (i.e., locate the logical flash page id $h_0(e) mod N_1$ in which the BF is stored), followed by the same procedure to check/set the k different bit positions associated with the key mapped by the k independent hash functions. Initially when

there is only one layer of BFs allocated, the k bit positions are directly checked or set in the respective BF in RAM. As the set size grows with more and more key insertions, the first layer becomes *full* (i.e., the number of inserted keys at some BF reaches a pre-calculated threshold determined by the demanded false positive probability, as well as the BF size). Consequently, all BFs in the first layer are consecutively written to flash-memory (logically). The forest is then naturally expanded by a new layer of BFs (e.g., the in-Flash phase in Figure 3.4), with every BF at the expanded layer as a child of a single BF in its upper layer. The expansion routine repeats once the current leaf layer (the lowest layer of the forest) becomes full again.

Multi-layer key locating: Unlike the single-layer design, where a key is allocated to only one possible BF, our forest structure design must allow a key to be allocated to **many possible BFs, one per layer through the height of the forest.** To cope with this extra key location complexity, we propose using an extra hash function h_1 to locate the key through the forest: for a given key e, we first locate it with the hash function h_0 at the first layer, as the single-layer design does. Assume the forest height is greater than 1 and for the key e, BF i is located in the first layer. If the key e is not found in BF i, we use h_1 as well as the branching factor b to locate one of BF i's child that needs to be searched for the key e (i.e., the hash value $h_1(e)$ and b are translated to the child id). If the key is still not found there, one of its children will be searched by the same procedure. This procedure repeats until either the key e is found, or the key is not found and the lowest layer of the forest has been reached.

Also, as each BF at the non-leaf layer has more than one child, the size of an expanded layer would be much larger than that of the available RAM. Therefore, the FBF stores every expanded layer of the BFs on a flash and uses the spared RAM space to buffer the bit updates resulting from the key insertions. It then flushes these buffered bit updates in the units of *page groups* to flash-memory, rather than individual pages, so as to reduce the flash writes as well as the garbage collection overhead resulting from the excessive in-place bit updates to flash. Each page group stores a number of BFs. The size of a page group is dimensioned at the size of an erase block (e.g., 128KB). We have all page groups that allow bit updates share the spared RAM space, rather than assign them an equal-sized exclusive buffer space. This idea of pooling RAM buffer space among page groups is motivated by the existing buffer filling rate deviation problem

(aforementioned in Section 3.2.2) within the single-layer BF design. The FBF achieves this by organizing the overall RAM buffer space with a hash table. The hash table maps each page group id to a **dynamic set** data structure that stores the bit updates applied in the page group. The dynamic set that allows the insertions of elements is already implemented by all mainstream languages, like C, C++, and JAVA, etc. With this organization, the buffer space for a page group could grow to hold new bit updates until the overall available RAM space is filled up. In fact, simply replacing the original buffer strategy with this pooling buffer could further reduce the total number of flash write operations by 50% on the same workload. Result details are reported in Section 3.5.

To further reduce the number of flash writes triggered by bit update flushes, the FBF design designates BFs at the lowest layer instead of any allocated BFs as the possible key insertion locations. On average, this strategy increases the buffer size per buffered BF by $1 + \left(b^{\alpha-1} - 1\right) / [b^{\alpha-1} \cdot (b-1)]$ times, assuming that the forest height is α. In the mean time, the key query performance could also be improved owing to the bit update buffering, because some searched keys that have their bit positions buffered in RAM could be checked without the corresponding BFs being loaded from the flash-memory.

3.4 Design Issues

In this section, we discuss several important design choices that have a great impact on the key query/insert performance of our FBF design.

3.4.1 Choosing Traverse Orders Based on Workload Characteristics

As aforementioned, the FBF organizes the BFs as a forest-structure. Therefore, the traverse order of the forest needs to be considered. In this section, we investigate the performance impact (ops/second) of the traverse order with respect to the workloads of different characteristics. From the introduced multi-layer key locating, we know that given a key e, the FBF deterministically locates a sequence of BFs, one for each layer, as the key's candidate locations. In general, there are two traverse orders when executing a key query in our FBF design: *top-down* order traverses the sequence of the located BFs from the one at the first layer down to the one at the lowest layer (leaf layer) while *bottom-up* order traverses in a reverse direction. The query-path length (i.e., the

number of BFs searched during a key query, denoted as ***qlen***) for any query that returns negative always equals the total number of BFs in the sequence. The qlens for queries that return affirmative may vary remarkably under different traverse orders. Therefore, if the portion of queries that return affirmative are considerable in the entire workload, the performance (ops/sec) impact of the traverse order choice may be significant. The motivation behind choosing traverse orders based on the characteristics of the workloads is that various workloads have different querying patterns. For example, some workloads render a temporal-recency query pattern, that is, it tends to query the recently inserted keys, rather than the ones inserted a long time ago. For this type of workload, it is wise to adopt a bottom-up traverse order to speed up the query performance by opportunistically minimizing the qlens for all queries that return affirmative.

On the other hand, there are certain types of workloads bearing a special pattern that a fraction of inserted keys that were once inserted close to each other temporally are queried many more times than the rest keys during the run. As an example, consider a backup application workload with a data dedup that eliminates redundant data blocks to minimize storage space usage. After the base backup is deduplicated, deduplicating a new backup will likely identify many existing data blocks. With respect to our FBF design, this implies that the keys stored in the upper portion of the forest (e.g., the root-layer) will likely be most frequently queried, as it accounts for a fraction of the keys related to the base or earlier backups. Consequently, to opportunistically minimize the qlens for all queries that returns affirmative, it is reasonable to choose a top-down traverse order. Section 3.5.4 evaluates the choices of traverse orders on the workloads of different query patterns.

3.4.2 Group Size Choice for Bloom Filter Bit Update Flushing

Page group size should be carefully chosen in our FBF design. Page group is the flushing operation unit during which the read and write of a page group are performed. To minimize the total flash I/O accesses needed for bit update flushes, the page group size should be chosen so that the ratio of flushed bit updates per-flush to the page group size is maximized. Otherwise, potentially more flash I/Os are needed to flush the same amount of bit updates to flash during the run on the same workload. Intuitively, to maximize this ratio, the page group size should not be too large, as the number of

flushed bit updates per-flush is constrained by the RAM buffer size. On the other hand, the page group size should not be too small (e.g., only contains one flash page) either. As mentioned in Section 3.2.2, the sizes of a page group should be dimensioned at one or multiple size of an erase block (e.g., 128KB) to mitigate the garbage collection overhead.

To investigate the effect of the page group size on performance (ops/second) of the FBF design, we compare the results of 6 runs on one real workload *vx-9m* (see Section 3.5.2 for details about this workload) with the page group sizes of 1MB, 2MB, 4MB, 5MB, 10MB and 20MB. We fix the overall RAM buffer space to 4MB used in each run. The results illustrate that when the group size is 1MB, it achieves the maximum aforementioned ratio value. Therefore, we choose a 1MB group size in the following experiments.

3.4.3 Choosing the Number of Hash Functions for Flash-based Bloom Filter

To minimize the largest qlen value in a forest, the FBF design flattens the forest-structure by adopting a larger branching factor b. Nevertheless, a larger b value results in more BFs needed to be buffered in the RAM, which, in turn, yields more page group flushes due to the smaller number of bit updates committed per-flush. In order to compensatefor this, we propose a new trade-off dimension that maps each key with fewer hash functions in a BF stored in flash-memory. This design choice would not be *space-wise* in the sense that it asks for more of a fraction of non-set bits (0's) existing in the bit array to achieve the same false positive probability. However, it yields less buffered bits for each buffered key insertion, which allows for more key insertions to be held with the same RAM buffer size. A detailed explanation follows.

Given a key, k different bit positions within a BF are identified using k different hash functions. Denote the optimal value of k to be k_{opt} , which corresponds to the minimized false positive probability $f = 0.5^{k_{opt}}$. k_{opt} is determined by the formula $k_{opt} = \ln 2 \cdot (M/N)$, where M and N are the number of bits in the BF and the maximum number of keys represented with the BF respectively. We call a BF *space-wise* [31] if the value of k is set to the optimal value k_{opt} according to this formula because any other k value would increase the size of the BF (the value of M), provided there is the same false positive probability.

Since each inserted key is represented by k hash values in the buffer, if we choose a smaller value of k, say k' instead of k_{opt}, then on average the same buffer can hold more insertions. In the meanwhile, potentially more key queries will be answered from the RAM buffer since more keys could be held in RAM. Furthermore, this also helps to reduce the number of group flushing operations, because more key insertions will be committed by one page group flushing, which contains the same number of bit updates. On the other hand, the size of each BF increases from M to M' to maintain the same false positive probability f:

$$M' = \frac{-N \cdot k'}{\ln\left(1 - \sqrt[k']{f}\right)}$$

The bigger BFs consume more space in the flash-memory. Section 3.5.6 provides the evaluation results of a real workload with different values of k to validate the legitimate k value choice.

3.4.4 Analysis of False Positive Probability

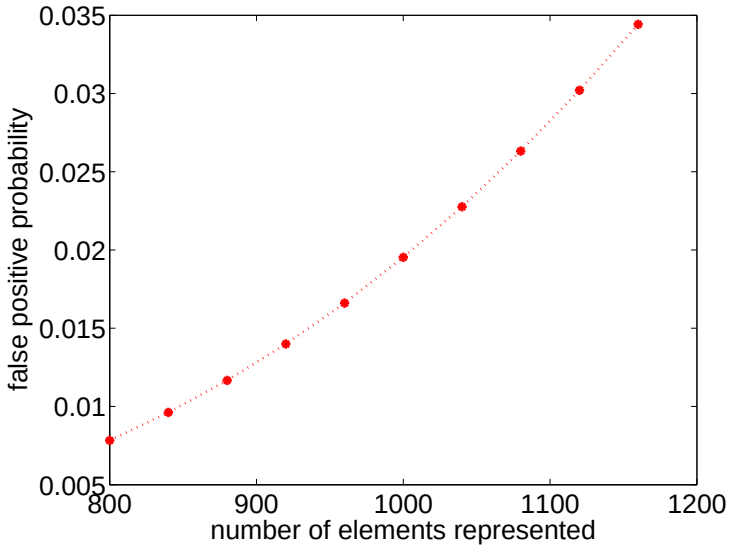

Figure 3.5: False positive probabilities vs. number of elements represented

The FBF organizes multiple flash page-sized BFs in a forest-structure. Each key is assigned to one BF in the forest (using hash functions). We first analyze the false

positive probability of the first layer, then we extend this analysis to the entire forest. Since each given key will only be allocated to one of the BFs in the first layer, all BFs in the first layer essentially form a *composite BF* that returns affirmative or negative in response to any key query and inserts a key into one of the BFs. We calculate the false positive probability of the composite BF as follows.

Suppose that n keys are inserted into the first layer of Q BFs and this leads to n_i key insertions in the i-th BF for $i = 1, 2, ..., Q$, assigned by a hash function h_0. Let the size of each BF be m' (e.g., $m' = 8 \times 4096 = 32768$) bits. As previously stated, k is the number of bit positions checked in each BF for a key query. Suppose the probability of a key hashing to the i-th BF is uniformly random, denoted by $1/Q$. Then, the false positive probability of the composite BF is given by:

$$\frac{1}{Q}\sum_{i=1}^{Q}(1-(1-\tfrac{1}{m'})^{kn_i})^k \approx \frac{1}{Q}\sum_{i=1}^{Q}(1-e^{-kn_i/m'})^k$$

where $\left(1-(1-\frac{1}{m'})^{kn_i}\right)^k \approx (1-e^{-kn_i/m'})^k$ is the false positive probability of that key in i-th BF.

By denoting the expected value of n_i by $\mathbb{E}(n_i)$ and according to Jensen's inequality for convex function, it could be shown that the false positive probability of the composite BF is no less than that of the single BF with the same number of hash function k, number of inserted keys n ($n = Q \cdot \mathbb{E}(n_i)$), and size m ($m = Q \cdot m'$), that is:

$$\frac{1}{Q}\sum_{i=1}^{Q}(1-e^{-kn_i/m'})^k \geqq \sum_{i=1}^{Q}(1-e^{-k\mathbb{E}(n_i)/m'})^k = \sum_{i=1}^{Q}(1-e^{-kn/m})^k$$

with the equality achieved if and only if $n_1 = n_2 = \ldots = n_Q$. However, as demonstrated by previous works [43, 44], with a single load balancer (i.e., hash function h_0), the number of keys (load) assigned to each BF is not balanced. In fact, the more skewed the distribution of keys across the BFs is, the higher the deviation between the false positive probabilities of the composite BF and the same-sized single BF. An intuitive reason for this is explained as follows. Figure 3.5 presents the false positive probabilities with respect to a sequence of numbers of inserted (represented) keys (from to 800 to 1160, with the interval of 40) for the i-th BF of the size of m' bits ($m' = 8192$). In addition, we set the expected number of inserted keys $\mathbb{E}(n_i)$ to be 1000, as well as the number of

hash functions to k_{opt} (calculated with m' and $\mathbb{E}(n_i)$). As illustrated in this figure, the false positive probability changes exponentially as the number of inserted keys deviates from $\mathbb{E}(n_i)$. Therefore, the false positive probabilities of the above-expected occupancy BFs are not averaged out by that of the other below-expected occupancy ones.

Theoretically, in order to achieve a near perfect load balancing across BFs, it could use the *power of two choices idea* [45] that hashes each key to two candidate BFs with two independent hash functions and inserts the key to the one that has the currently lower occupancy. To query a key, both candidate BFs are checked. Applying this idea to the FBF design is much more expensive than applying it to a traditional in-RAM BF design, since it doubles the number of flash reads related to BF queries.

We propose a more practical solution to mitigate the skewness of the distribution of the keys among BFs for each layer. We keep track of the number of inserted keys for all BFs and once the number of inserted keys equals the predefined $\mathbb{E}(n_i)$, we expand the forest by adding another layer for new insertions. This strategy will make sure that for any layer, the false positive probability of composite BF is bounded by that of the single BF.

In addition to the key distribution skewness, the false positive probability of the FBF design relies on the other major dominant factor – the qlen, which measures the number of BFs checked per key query (obviously, the expected qlen is 1 for a single-layer composite BF).

Let the qlen be α, and the false positive probability of a single BF be upper-bounded by f (we could enforce this upper bound false positive probability by throttling the number of inserted keys into BF under a predefined value). The FBF design checks α BFs during a key query, one for each layer. Hence, intuitively, the effective false positive probability increases by a factor of α. Theoretically, by taking into account all combinations of α BFs, we could derive that the probability of exact i false positives is as follows:

$$\Pr(i \text{ false positives}) = \binom{\alpha}{i} \times f^i \times (1-f)^{\alpha-i}, \text{ where } i \leqslant \alpha$$

Consequently, the overall probability of a false positive is

$$\Pr(\text{false positive}) = \sum_{i=1}^{\alpha} \binom{\alpha}{i} \times f^i \times (1-f)^{\alpha-i}$$

Setting $f = 0.1\%$, and $\alpha = 1, 2, 3, \ldots 10$, we can easily see that the overall probability of a false positive grows proportionally as α increases. Luckily, the qlen in our FBF design is *logarithmic* to the number of BFs, whereas the qlen in the linear-chaining design is *proportional* to the number of BFs. Hence the false positive probability of the linear-chaining design is many times higher than that of the FBF design. The single-layer design achieves the minimal false positive probability since its qlen always equals 1. Table 3.1 presents a comparison on false positive probabilities among three designs (single-layer, linear-chaining, and the FBF), with the f set to 0.1% for all cases. Be aware that the FBF design yields the same false positive probability as the single-layer design does when the forest height is 1. From the table we see that the linear-chaining design does suffer a multiple times higher false positive probability than the other two designs, which precludes itself from our candidate list for further discussion.

Table 3.1: False positive probabilities for linear-chaining design, single-layer design and the FBF design over *Linux* and *vx* datasets

workload name	linear-chaining	single-layer	2-layer forest	1-layer forest
Linux	1.04%	0.07%	0.3%	0.07%
vx-20m	1.82%	0.06%	0.3%	0.06%

3.5 Evaluation

In this section, we evaluate the design of the FBF with a moderate-end SSD from OCZ Technology, using two real dedup workloads of different characteristics. Among the evaluations, we measure the performance of the processing speed in terms of **ops/sec** which measures the number of key query/insertion operations accomplished per second. Accomplishing a key query/insertion operation means for the key query either a BF answers affirmative or each BF in the query path (in case of the FBF design) answers negative and a key insertion is made consequently. We also realize that some high-end SSD devices, like the Fusion IO, are available on the market. Nevertheless, the Fusion IO itself takes up hundreds to thousands of megabytes of RAM space as the device cache [46]. For example, as shown in the user-guide for 80GB ioXtreme, for a 4KB recommended file system block size, the drive consumes 800MB of RAM, which

somehow invalidates our purpose of using a Bloom Filter with flash-memory to save RAM space. Also, such a large device cache sitting in the middle between the flash-memory and the in-RAM buffer space would buffer tons of data blocks, making it very difficult to justify the benefit of our own buffering cache design.

3.5.1 Description of Hardware Platform and Software Implementation

We implement the linear-chaining design, the single-layer BF design as well as our proposed FBF design, as a software layer sitting between the application traces and the storage device, using Python. The experiments are carried out on a Linux build 2.6.32 SMP x86_64 machine. The machine has two 2.0 GHz cores, with 1GB RAM. The SSD model is OCZ Agility SATA II [47], with a storage capacity of 120GB.

3.5.2 Description of the Data Deduplication Workloads

Table 3.2: Descriptions of two workloads

workload name	*Linux*	*vx-full*
number of records	57,414,727	164,766,619
number of unique records	4,649,832	90,127,392

Chunking based data deduplication systems heavily involve key query/insert operations. It eliminates data redundancy within either backup data or data stored in the primary storage. A chunking based data dedup system works by splitting files into multiple data chunks using a certain chunking algorithm, and calculating a SHA1 [5] hash signature based on each chunk's content to determine whether two chunks are identical. In in-line data dedup systems, the hashes of the chunks arrive at the deduplication server one-at-a-time and the server has to query an index maintaining all chunk hashes the server had stored. An affirmative querying result implies the incoming data chunk contains redundant data and could be deduplicated. Otherwise, the incoming one is new and its respective chunk hash has to be inserted into the index.

Two real workloads are generated by the content-defined chunking algorithm [18], commonly used in data dedup systems mentioned previously to produce data chunks. Within either workload, each record is a chunk hash value of 160-bit length, calculated

by SHA1. A chunk hash is used to globally identify a data chunk in the storage system. In many practical cases, the storage systems need to scale to tens of terabytes to petabytes of data volume. Thus, the chunk hash index is too big to fit in RAM and has to be stored on the hard disk. Index operations as well as throughput, are limited by slow disk operations (disk seek). Because a backup needs to be finished within a backup window of one-night or one-weekend), high throughput in-line dedup system is demanded. Typically, such systems [6, 7] utilize Bloom Filter to determine if a chunk hash has been seen before so that the disk operations can be avoided for chunks that are seen for the first time.

As illustrated in [6], a client-side Bloom Filter is proven to be very efficient to reduce the chunk look-up latency, because all chunk queries initiated by this particular client will first be answered by its own BF. Only those not-found chunk-ids are sent to the server-side. It is reported that with a Bloom Filter alone, up to 18.6% of the index look-up disk I/Os are reduced. With further optimizations in design, 99% of disk accesses could be avoided. On the other hand, due to the huge number of chunk-ids produced by the chunking algorithm in a streaming manner, traditional in-RAM BF requires pre-allocating up to several GBs of RAM space.

Table 3.2 presents two workloads produced by dedup systems. The *Linux* workload consists of chunk-ids derived from 20 versions of Linux kernel source distributions. (Linux-2.6.30.1 – Linux-2.6.30.10, Linux-2.6.35.1 – Linux-2.6.35.8, and Linux-2.6.36.1, Linux-2.6.36.2). The *vx-full* workload consists of chunk-ids derived from a networked primary file system shared by a group of software engineers. As shown in Table 3.2, 8.1% of chunk-ids from the *Linux* workload are unique ones, while 54.7% of chunk-ids from *vx-full* workload are unique ones. *Linux* is a typical dedup workload for the data backup environment, while the *vx-full* is a typical dedup workload for primary file systems. We called the former query intensive and the latter insertion-intensive, because the percentage of unique chunk-ids determines the key insertion ratio throughout the process.

Intuitively, for a workload with the same number of records, the higher the key insertion ratio is, the more block flushing operations will occur. We plan to investigate our scheme and other schemes under both query-intensive and insertion-intensive workloads. We also obtain some subsets from the *vx-full* workloads and present their notations as

follows: *vx-9m* contains the first 9 million records of *vx-full*; *vx-20m* contains the first 20 million records of *vx-full*; similarly, v*x-25m* contains the first 25 million records of *vx-full*. The number of unique records contained in *vx-9m*, *vx-20m* and *vx-25m* are $5,628,873$, $11,328,914$, and $14,163,022$ respectively. In addition, the ratios of unique chunk-ids are 62.5%, 56.6% and 56.7%, which do not deviate much from 54.7%, the ratio of *vx-full*. In addition, the recency-querying pattern in the *vx* workload is verified and presented in Section 3.5.4. Both features indicate the representativeness of a subset of *vx-full*. Further experimental results illustrate that the results obtained from a smaller workload, such as *vx-9m,* are indeed consistent with the results obtained from *vx-full*, validating the representativeness.

3.5.3 Evaluation of Bloom Filter Buffer Schemes

Table 3.3: Equally-divided (fixed-size) vs. set-dictionary schemes on *vx-20m*

cache structure	equally-divided	set-dictionary
number of cache hits (millions)	3.627	4.185
number of flash reads (millions)	16.373	15.815
number of flash writes	$2,024$	$1,053$
avg. number of bit updates per flash-write	$55,548$	$105,712$
ops/sec	$8,405$	$8,657$

Tables 3.3 and 3.4 compare the fixed-size (also equally-divided) in-RAM buffer scheme with our proposed set-dictionary on the *vx-20m* and the *Linux* workloads. Over 50% of the records in *vx-20m* are unique, making it an example of an insertion-intensive workload. On the other hand, only 8% of the records in *Linux* are unique, making it an example of a query-intensive one. (each unique record in *Linux* on average will be queried over 10 times through the run). For fairness, both buffer schemes are tested

Table 3.4: Equally-divided (fixed-size) vs. set-dictionary schemes o*n Linux*

cache structure	equally-divided	set-dictionary
number of cache hits (millions)	304,413	417,002
number of flash reads (millions)	57.110	56.998
number of flash writes	1145	312
avg. number of bit updates per flash-write	40,334	142,544
ops/sec	12,354	12,510

with the single-layer BF design [36], with the size of a page group of 1MB. For *Linux* , the single-layer design uses 4096 BFs, each of the size of 4KB, and is configured with a 1MB overall buffer space in RAM; for *vx-20m*, it uses 5632 BFs, each of the size of 4KB, and is configured with a 4MB overall buffer space. As shown in the two tables, with a uniformly higher processing speed (8,657 vs. 8,405 on *vx-20m* and 12,510 vs. 12,354 on *Linux*), the set-dictionary buffer scheme requires roughly 50% and 33% of the number of flash write operations taken by the equally-divided buffer scheme respectively. These results validate that the set-dictionary buffer scheme is uniformly better for both type of workloads. Remember that single-layer BF design is equivalent to the FBF design with a single layer allocated in flash. Hence, such a buffer design could also benefit the FBF designs that involve block flushing operations at each layer.

To figure out the impact of the overall buffer size on the processing speed with both buffer schemes, we double the overall buffer size from 1MB up to 16MB in 5 runs on *vx-9m* for both buffer schemes and present the results in Figure 3.6. Each run is configured with 5120 BFs, each of the size of 4KB. Two observations could be made from this figure. Firstly, both processing speeds increase as more RAM space is available for the buffer. Secondly, the set-dictionary scheme uniformly outperforms the fixed-size (also equally-divided) scheme for all sizes, by up to an 11% higher processing speed in terms of ops/sec when a 1MB overall buffer size is used. Further examining the number

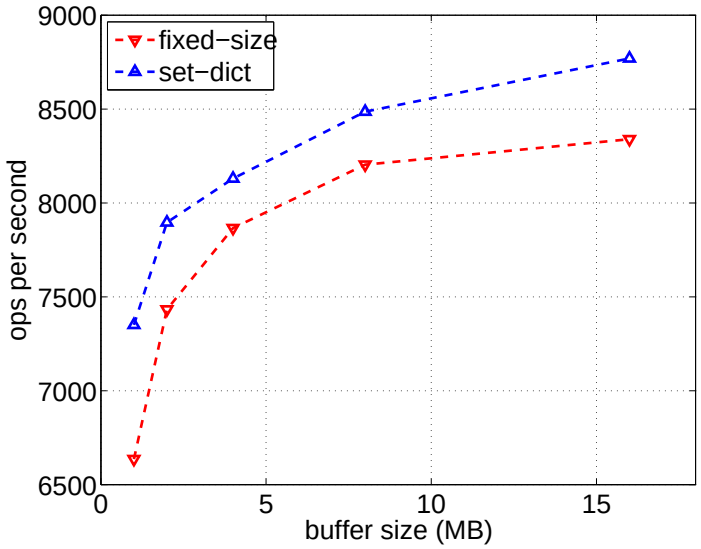

Figure 3.6: Processing speed vs. overall buffer size on vx-9m

of flash write operations illustrates a uniformly 50% reduction for our set-dictionary buffer scheme.

We also evaluate both buffer schemes on the *Linux* workload with the single-layer design (equivalent to the FBF design with one only layer in flash). To figure out the impact of the overall buffer size on the processing speed , we choose a sequence of buffer sizes 1MB, 2MB, 4MB, 6MB and 8MB for 5 runs. Each run is configured with 5120 BFs, each of the size of 4KB. As expected, the set-dictionary buffer scheme attains a uniformly about 50% reduction on the number of flash write operations, compared with the equally-divided buffer scheme.

3.5.4 The Impact of Traverse Order

In this section we first verify the recency-query pattern and long inter-distance query pattern existing in two real workloads. Then we propose an optimal traverse order (e.g., either *bottom-up* or *top-down*, as discussed in Section 3.4.1) with respect to the identified query pattern for each workload.

To verify the recency-querying pattern, we make an FBF structure of a binary-tree shape and run two experiments on *vx-9m* with *top-down* and *bottom-up* traverse orders.

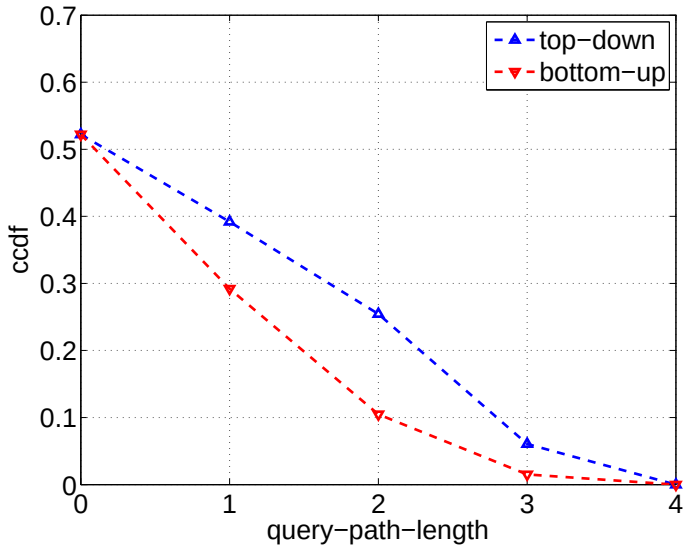

Figure 3.7: The ccdf of query-path-length for successful queries on vx-9m

The FBF is configured with 1MB overall in-RAM buffer. Through each run, we collect the query-path length (qlen) for each key query. We filter out unsuccessful queries, since they are not of interest here. Figure 3.7 plots the complementary cumulative distribution functions (ccdf) of the distribution of qlen for all successful queries. Notice that the dot on the x-axis coordinate α stands for the percentage of queries with qlen strictly greater than α, where $\alpha = 0, 1, 2, 3, 4$. Two important observations could be made from this figure: (1) Most recently inserted keys (e.g., those with qlen of zero) account for 48% percentage of successful queries; and (2) *bottom-up* traverse order (represented by the lower-triangular dotted curve) yields a uniformly shorter qlen than *top-down* traverse order does (represented by the upper-triangular dotted curve). Both of the two observations confirm the existence of recency-querying pattern in the *vx* workload. The second one further indicates that *bottom-up* traverse order could reduce the expected qlen if the workload does render a recency-query pattern.

Linux workload provides a real example bearing long inter-distance query pattern, described in Section 3.4.1. Table 3.5 verifies the existence of the long inter-distance query pattern in the *Linux* workload. In this table, queries with qlen = 0 are answered either from the cache or from the first layer when the first layer was sitting in memory; queries with qlen = 1 are successful queries answered by the first layer after it was committed to

flash (notice that an unsuccessful query could have qlen either to be 0 or 2, but not 1); queries with qlen = 2 are answered by traversing each layer of the forest, provided the overall forest height is 2. It can be seen from Table 3.5 that the dominant portion (40.7 million out of 57.4 million) of queries have qlen of 1 by *top-down* traverse order, while the same portion of queries have qlen of 2 by bottom-up traverse order. This dominant portion of queries with qlen = 1 illustrates that a majority of queries are found in the first layer, which captures keys for the base backup. In addition, it also indicates that this *Linux* workload is a query-intensive one. Provided that the upper portion of the forest (e.g., the first layer) accounts for the keys related to the base backup, it is wise to adopt a *top-down* traverse order to shorten the expected qlen. Table 3.6 illustrates a comparison on *Linux* with *bottom-up* and *top-down* traverse orders. The result supports our argument by demonstrating that the number of flash read operations by *top-down* order is 35% less than that by *bottom-up* traverse order. Thus, it gains a 15% processing speed.

Table 3.5: Statistics of qlen with 4-branching FBF for *Linux*

qlen	number of successful- (millions)	number of failed- (millions)
bottom-up		
0	3.977	2.337
1	8.049	0
2	40.748	2.305
top-down		
0	3.977	2.337
1	40.748	0
2	2.3048	8.048

Table 3.6: Bottom-up vs. top-down traverse order with the 4-branching FBF for *Linux*

workload name	number of reads (millions)	processing speed (ops/sec)
bottom-up	94.153	8,385
top-down	61.454	9,621

3.5.5 Evaluation on Processing Speed over Bloom Filter Designs

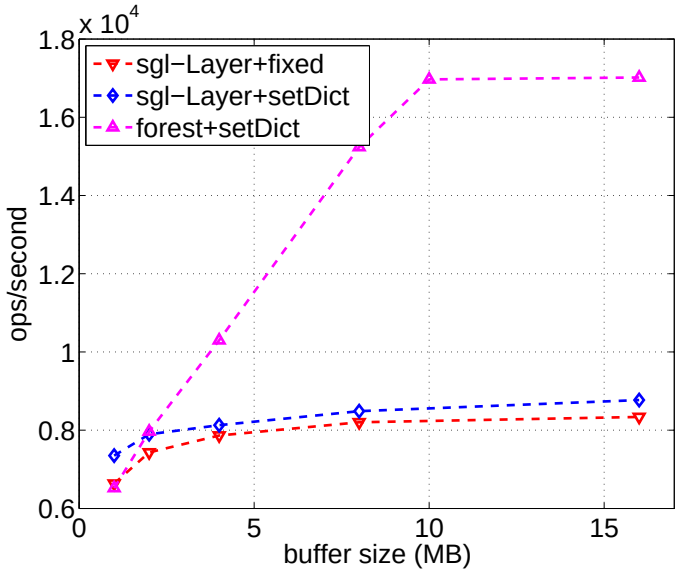

Figure 3.8: Processing speed vs. buffer size on vx-9m

In this section, we compare the processing speeds of the single-layer BF design and the FBF design with respect to both *vx* and *Linux* workloads. We run two sets of experiments with overall buffer sizes ranging from 1MB to 16MB on *vx-9m*. Figure 3.8 plots the processing speed vs. buffering size for the FBF and single-layer BF designs on *vx-9m*. Notice that the single-layer design could also be seen as a particular case of the FBF design of exactly one layer in the flash.

In Figure 3.8, the curve with downward-pointing triangular markers represents the results of the single-layer BF design with a fixed-size (also equally-divided) buffer scheme. The curve with diamond markers represents the results of a single-layer BF design combined with a set-dictionary buffer scheme. The curve with upward-pointing triangular markers represents the results of our FBF design combined with a set-dictionary buffer scheme. Because the single-layer design does not support the dynamic growth of the dataset size, for fairness, we configure each run with 5120 BFs, each of the size of 4KB. In the meanwhile, the size of the page group is set to 1MB across all runs.

Several important conclusions could be drawn from Figure 3.8. Firstly, both designs with set-dictionary buffer schemes uniformly outperform the single-layer design with a

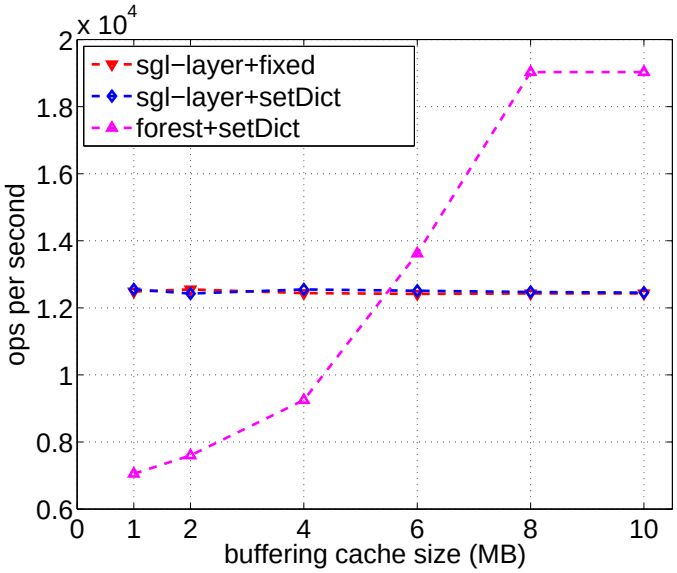

Figure 3.9: Processing speed vs. buffer size on Linux

fixed-size (also equally-divided) buffer scheme for all buffer sizes presented. Secondly, the processing speed increases as the larger buffer size becomes available for all designs. However, the slope (gradient) of the processing speed curve for the FBF design is much higher than that of the single-layer BF design, showing that a much larger processing speed gain could be attained by the FBF design when both are given a larger RAM space. Moreover, if the allowed buffer size is less than 2MB, then the single-layer design with set-dictionary buffer scheme is better. Otherwise, we should choose the FBF design. Finally, the slope of the curve for the FBF design flattens after the overall buffer size is bigger than 10MB. This phenomenon can be explained by the fact that the total amount of records could be buffered solely in RAM when the buffer size goes beyond 10MB. Hence further increasing the buffering cache size would not improve the processing speed.

It is very interesting to see that by the boosting effect of the in-RAM phase, the FBF design, which allows for expansion, could outperform the single-layer design, that could only handle static workloads, significantly (e.g., up to $2x$ faster) with the same amount of RAM space consumed!

We also compare the results of both the single-layer BF design and the FBF design

on *vx-full*. With a 40MB buffer size and 1MB page group size used in both designs, the FBF design achieves $12,105$ ops/sec while the single-layer design achieves 9390 ops/sec. Provided that the total amount of records in *vx-full* would require a 160MB in-RAM BF to process, this 30% higher speed for a forest-structured BF with a 40MB buffering cache size is consistent with the results we obtained from $vx - 9m$.

Similarly, for the *Linux* workload, Figure 3.9 illustrates the comparisons on the processing speed between the single-layer BF design and the FBF design with the overall buffer size ranging from 1MB to 10MB. In particular, for the FBF design, we test both the two-phase scheme and the in-flash only scheme (i.e., the scheme allocates the first layer of BFs in flash directly.) and present the results as the curve with the upwards-pointing triangular markers and the curve with diamond markers respectively. We configure the forest height to be 1 for the in-flash only scheme, making it equivalent to a single-layer BF design with set-diectionary buffer scheme.

Some important observations could be made from the results. Firstly, the single-layer BF design (the curve with downward-pointing triangular marker) does not appear to be a significant improvement when more overall buffer space is used. This could be explained by the fact that the dominant portion of the *Linux* workload is key queries. Hence, a larger buffer for storing more bit updates would not significantly affect the overall performance. Secondly, the processing speed for the two-phase scheme is worse than that of the single-layer BF design with set-dictionary buffer scheme for a buffer size between 1MB and 5MB. In fact, this provides an extreme case where the workload is dominated by queries and the available in-RAM buffer size is small (e.g., less than 5MB in this case). It is wise to choose the in-flash phase only design and try to flatten the forest-shape by as much as possible. Finally, the slope of the curve for the forest-structured design flattens when the allowed memory size exceeds 8MB. This is because the total amount of records could be buffered solely in the memory when the buffer size goes beyond 8MB.

3.5.6 Tuning the Number of Hash Functions

This section presents the evaluation results for tuning the number of hash functions used for the FBF design. In particular, it tunes the number of hash functions used within the BFs in the flash-memory. This tuning trades higher flash space usage for more key

Table 3.7: The number of hash functions

number of hash function	10	2
sum size of 1st layer	1×4	1×4
sum size of 2nd layer	1×16	4.34×16
number of cache hits	$3,225,730$	$4,529,459$
expected qlen	1.152	0.945
number of flushing ops	604	127
process speed (ops/sec)	7828	9755

insertions buffered within the same buffer space.

Table 3.7 presents the experimental results of two different numbers of hash functions on the dataset *vx-20m.* Both running results are collected from a FBF organization configured with a 4-branching factor and a height of 2. The overall buffer size is set to 4MB and the page group size is set to 1MB.

Compared with $k = 10$ (the space-wise configuration), $k = 2$ yields up to an 18% smaller expected qlen, 40% increase on cache-hit ratio and 25% increase in processing speed. Also, since this experiment only changes the sizes of the BFs in the 2^{nd}-layer allocated in the flash-memory, the total size of the FBF is 2.67 times of that of a space-wise one. In addition, as expected, both the configurations result in the same false positive probability 0.26%.

3.6 Related Work

The Bloom Filter structure was initially proposed by B. H. Bloom, as a compact representation of a static set with a certain probability of false positives to serve the set membership queries [48]. Since then, Bloom Filters are widely used in database applications [49] and are drawing increasing attention from the networking community recently [50, 51]. Data Dedup has become another popular application area for Bloom Filters. Zhu et al. [6] utilizes a Bloom Filter to minimize chunk look-up latency; Navendu et al.

[21] adopts Bloom Filters as a feature set of a data chunk; Lu et al. [52] takes a group of Bloom Filters to select out data chunks with more redundancy by filtering out low redundant ones.

Our FBF design is inspired by [53, 38], each of which demonstrates the importance of representing a dynamic growing set, and proposes a solution with RAM-based dynamic Bloom Filters. Nevertheless, two issues are not addressed with their design: (1) the linear look-up on each allocated BF causes the number of false positive errors to be significantly higher than the calculated result; and (2) how to design a Bloom Filter when its size exceeds the capacity of RAM as the dynamic set size grows.

Canim et al. [36] and Debnath et al. [37] propose similar designs to build a Bloom Filter with a flash-memory and use a moderate amount of main memory space to buffer bit updates. In order to increase the flash-memory access locality, Canim et al. limits the size of each BF on the flash-memory to 2MB and accesses the flash at the unit of a BF. In addition, in order to amortize the cost of fetching 2MB from flash each time, it buffers queries into a request queue. Radically different from traditional Bloom Filters, this design does not guarantee that queries will be processed within a certain amount of time. In contrast to their work, our BF design does not buffer requests so as to immediately answer each query request in the order the request was received, in order to be useful for a data dedup application. Furthermore, both designs presented in [36, 37] are only able to tackle static workloads. Our design targets dynamic cases when the dataset size could not be determined in advance.

3.7 Summary and Conclusions

In this chapter, we present the design and evaluation of the FBF, a Bloom Filter that combines a limited RAM space with a much larger flash space to dynamically adapt to the growth of a dataset. In particular, with our design, the querying overhead grows logarithmically as the overall BF size grows. We conduct extensive experiments on two real-world data dedup workloads of different characteristics. Our experimental results show that the forest-structured BF design achieves a 2 times faster processing speed with uniformly 50% fewer flash write operations, compared with the state-of-the-art in-flash BF designs.

Chapter 4

BloomStore

4.1 Introduction

The key-value (KV) store contains a large number of KV pairs and provides two simple operations: key lookup and KV pair insertion. These two operations heavily depend on an internal index structure that maps a key to its associated value. Recently, many applications, such as data dedup [6], on-line multi-player gaming, and Internet services like Amazon and Facebook [54], etc., have preferred to use the KV store, rather than the traditional relational database, because of its simplicity and better scalability. In order to maximize the KV store performance, we need to carefully provide efficient index and KV pair accesses based on the characteristics of the underlying storage media containing the index structure and the KV pairs.

The performance of the KV store often governs the performance of its applications. The KV store is commonly used to implement the chunk index in data dedup, where a chunk ID (SHA1 value computed based on the chunk's content) is the key and its associative chunk metadata (e.g., physical storage location, stream ID, etc.) is the value. Chunk lookup searches a given chunk ID from the KV store, while chunk insertion adds a new chunk ID and its metadata to the KV store. Zhu et al. [6] pointed out that the key performance bottleneck for (in-line) data dedup is its key (chunk) lookup throughput. In addition, applications that detect redundant data transfers across WANs and subsequently send their associated references are [55] recently demanding key lookup throughputs of no less than $10,000$ operations per second.

To design a high-throughput KV store ($> 10,000$ key lookups/second), a typical method is to keep its index structure in RAM to rapidly map each key to its KV pair location on the secondary storage, such as flash or HDD [8]. Also, many KV store designs rely on an in-RAM large-sized hash table to index all KV pairs stored on the flash. Nevertheless, the downside of this approach is that the maximum number of KV pairs in the KV store can be constrained by the available RAM space (scalability constraint).

Provided the high throughput and low access latency requirements, the most cost-effective way to scale up the KV store is to move part of its index structure into the secondary storage. Recently, flash-memory (particularly in the form of SSD) has become one of the popular storage alternatives to the traditional HDD. The flash-memory could persistently store the index and deliver an access speed 100–$1,000$ times faster than the HDD. Compared to RAM (DRAM), however, the flash access speed is 100 times slower. As for the unit price (in terms of \$/GB), flash-memory is 10 times cheaper than RAM, while it is 20 times more expensive than HDDs, thus, positioning it in the middle of these two storage devices. In our design, we use a NAND flash-memory based SSD as the secondary storage. Throughout the rest of this chapter, for simplicity, we refer to the SSD as flash. Accordingly, each read/write operation on the SSD is in the unit of a flash page.

In order to break the scalability constraint (but losing performance benefits to some extent), the index structure (e.g., a large hash table) should be eventually stored in the flash-memory instead of the RAM. However, many index structures that involve intensive small random writes become challenging to be stored on the flash. More specifically, storing a hash-table based index structure on flash causes a few problems: (1) the hash table is randomly accessed (inserted); each KV insertion triggers an expensive random flash page write operation to modify only one hash table entry of several bytes, which is much smaller than the size of a flash page; (2) the hash table is not a garbage-collection friendly data structure, spreading inserted entries across all flash pages occupied by the hash table; even a small portion of invalidated entries (e.g., by update or delete) may lead to excessive in-place updates scattering a large number of pages, which severely aggravates flash write and garbage collection overheads; and (3) the hash table requires a much bigger storage space, as the load factor of an efficient hash table usually needs

to be well below 50% to keep the lookup time bounded.

In this chapter, we aim at designing a flash-based KV store architecture called BloomStore that not only assures an extremely low amortized RAM overhead per KV pair (the consumed RAM space divided by the total number of KV pairs) to be less than 1 byte/key, but also achieves high key lookup/insertion throughput. The BloomStore divides the flash space into a number of logical partitions and indexes each partition separately with a sequence (chain) of Bloom Filters (BFs). It associates a BF with a flash page of KV pairs, where the BF summarizes the keys in the flash page and has a flash pointer to the flash page. For low RAM usage, BloomStore keeps a flash-page sized data buffer and a very small sized BF buffer per partition in RAM, while storing all other BFs and associated KV pairs on the flash. For high lookup/insertion throughput, BloomStore reduces the maximum number of flash page reads by flash partitioning. In each partition, it maintains a dedicated flash page sized KV pair buffer to temporarily buffer inserted KV pairs. The BloomStore maximizes the number of buffered BFs in RAM, so as to minimize the number of flash page reads for loading the BFs from the flash during key lookups and minimize the BF chain update operations that may occur with key insertions. Under two different breeds of data dedup workloads, our experiments illustrates that BloomStore can outperform the state-of-the-art KV store designs in terms of the RAM usage and key lookup throughput.

The rest of the chapter is organized as follows. Section 4.2 presents a detailed survey of existing works, as well as the two most relevant KV store designs. Section 4.3 provides an overview of flash-memory. Section 4.4 presents the BloomStore design. Section 4.5 presents an extensive experimental evaluation of our BloomStore design with two typical real workloads from the data dedup applications. Section 4.6 summarizes our work and draw conclusions.

4.2 Related Work

MicroHash [56] is an index structure designed for memory-constrained embedded devices. It mainly emphasized the optimization of energy usage and the memory footprint, not access latency.

Wu et al. [57] and Nath et al. [58] proposed the on-flash B-tree and B+ tree

(FlashDB) solutions, respectively. However, they are application-specific and efficient under the following limited conditions: (1) keys are distributed in a small numerical range; (2) a small number of leaf-level buckets are active at any given time; (3) access latencies are not critical.

FAWN [59] and ChunkStash [8] store a checksum and a pointer into an index entry of an in-RAM hash table that points to a single KV pair stored in flash. The checksum is used to avoid (with high probability) triggering flash accesses to compare keys for every index entry searched in the hash table during key lookup. The amortized RAM overhead per KV pair of each design is computed as 6 bytes/key. For these designs that index all keys with a single hash table maintained in RAM, the lower-bound RAM overhead is the footprint of the flash pointers (e.g., 4 bytes/key). The KV store size in flash is constrained by the flash pointer size and its KV pair size. For example, the flash space of the KV store with a 4-byte flash pointer and a 64-byte KV pair length can be 256 GB at maximum.

In the meanwhile, two recent KV store designs, BufferHash [60] and SkimpyStash [61], have reduced the RAM overhead per key by efficiently placing their index structures over the RAM and flash. We will look into both designs in more detail in the next section.

4.2.1 Analysis of Recent KV Store Designs

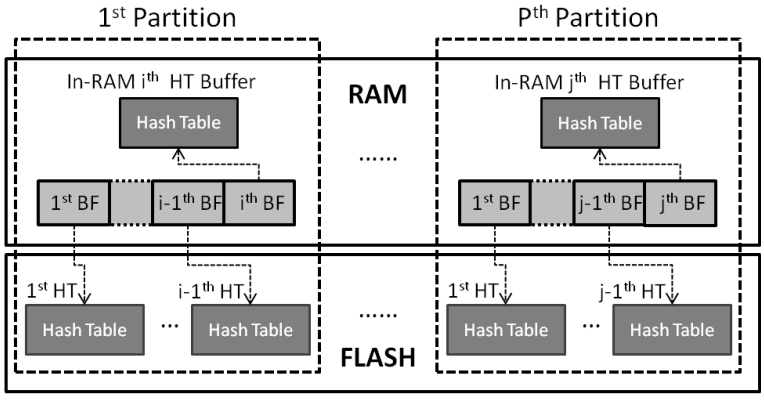

Figure 4.1: BufferHash architecture: multiple partitions, a hash table buffer and a chain of BFs in RAM for each partition, and a chain of corresponding hash tables in flash for each partition (**all BFs in RAM**)

We begin by explaining the properties of a Bloom Filter (BF) that are widely used to devise the index structures of many KV store designs.

Bloom Filters [48]: A BF supports space-efficient membership queries as follows: (1) a set $S = \{e_1, e_2, ..., e_n\}$ of n keys is represented by a vector v of m bits, initially all set to 0; (2) a set of k different hash functions h_1, ..., h_k, are used to set bits at $h_1(e)$, $h_2(e)$, ..., $h_k(e)$ positions for each e_i in set S; (3) to lookup a key e_i, bits at positions at $h_1(e_i)$, $h_2(e_i)$, ..., $h_k(e_i)$ are checked; (4) if any of these are 0, then e_i is not present in the set for sure; otherwise, it concludes that the key e_i is in the set (an affirmative answer); (5) false positive errors may exist; e.g., a key that is not in the set is mapped to k bit positions which are already set to 1 during insertions of other keys; and (6) a false positive probability f is affected by the BF parameters: n, m and k, where f is calculated as

$$(1 - (\frac{1}{m})^{kn})^k \approx (1 - e^{-kn/m})^k$$

BufferHash [60]: It divides the flash space into a number of logical partitions, as shown in Figure 4.1. Each partition maintains a small in-RAM hash table (HT buffer) for the KV pairs stored in the partition. The BufferHash basically employs the hash tables to store its index structure and associated KV pairs. Each hash table is implemented by using the cuckoo hashing[62] with two hash functions that help to improve the space utilization efficiency (e.g., to attain 50% load factor) at the cost of more hash table lookups per key. When the in-RAM hash table of a partition becomes full (i.e., its load factor reaches a predefined threshold), it is written to the flash. Subsequently, a new hash table of the same size is instantiated in RAM for the incoming KV pair insertions. In this way, the multiple hash tables of a partition are being chained with the newest (chronologically) hash table residing in RAM to accommodate newly inserted KV pairs. Suppose that there are P partitions, and each partition on average contains C hash tables. Then, only $1/C$ fraction of the entire hash tables are kept in RAM. The BufferHash keeps a BF in RAM for each hash table stored in either RAM on flash. In order to look up a key in a partition, the BufferHash identifies a specific partition where the key resides by using a hash function. Next, it examines the chain of the BFs linked to the partition in the reverse order of their creation times. For each of the BFs where the key is found (note that there could be multiple BFs that have the key), the

BufferHash looks up the key from the associated hash tables stored either in the HT buffer or on the flash.

However, the BufferHash consumes considerable RAM space for the following reasons: (1) hash tables have low load factors (50% recommended in the BufferHash design); and (2) all the BFs for all the partitions are kept in RAM.

SkimpyStash [61]: It stores all the KV pairs in flash, while maintaining an in-RAM hash table (called the hash table directory) to map keys to their locations in flash, as illustrated in Figure 4.2. Unlike the BufferHash, SkimpyStash stores a flash pointer instead of the actual KV pair in the hash table. The SkimpyStash regards the flash as an append-log. It appends the inserted KV pairs to the log sequentially. It also maintains the single in-RAM data buffer of a flash page size to temporarily hold new KV pairs. When the buffer becomes full, SkimpyStash flushes the KV pairs in the buffer to the flash through an append operation. This has been proven to be an efficient way to maximize the write performance of the flash-memory [63]. To further minimize the RAM usage, it hashes multiple keys into the same bucket in the hash table. It then resolves any collisions with the linear chaining, where the KV pairs in the same bucket are chained in a linked list and stored in the flash. The in-RAM hash table consists of a set of buckets that contains a BF and a flash pointer. Each flash pointer points to the tail (the most recently inserted KV pair) of the corresponding linked list. Each KV pair on the flash contains a flash pointer pointing to its predecessor (the previously inserted KV pair in the same bucket) in addition to its KV pair. Each bucket also keeps a BF in RAM to memorize the inserted keys in that bucket. This BF helps to decide whether the searched key exists in a bucket before blindly following the pointer to search the key from the chain of KV pairs in the flash. The use of the BF is crucial for the SkimpyStash design to reduce flash page reads for key lookups. Otherwise, it has to always traverse the entire linked list to conclude the non-existence of a searched key.

SkimpyStash may incur multiple flash page reads for a key lookup to determine the demanded key from the chained KV pairs. Suppose that the average chain length per bucket is l, then each key lookup needs to have $0.5l$ flash page reads on average. Considering the desirable uniformity of a key distribution into the buckets, the chance becomes extremely small that a series of new keys colliding with the same bucket are stored in the same flash page. Thus, the average number of flash page reads for each

key lookup is proportional to the number of keys hashed into a bucket.

A formula $(1 + \frac{4}{average\,bucket\,length})$ is provided to calculate the amortized RAM overhead per key stored with the SkimpyStash design. For an average bucket length of 10, it pays a 1.4 byte RAM footprint for each KV pair stored in the flash.

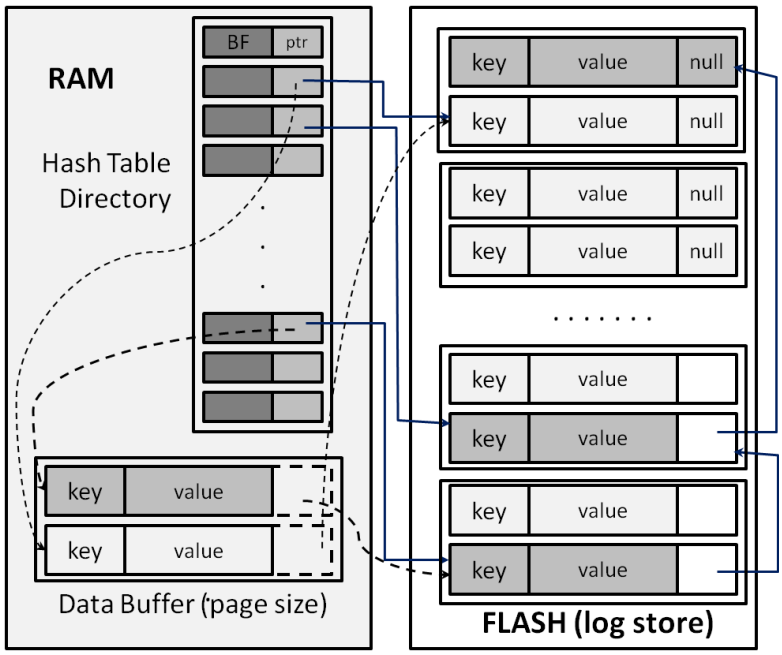

Figure 4.2: SkimpyStash architecture: a hash table directory and a single data buffer in RAM, and linked-listed KV pairs in flash

4.3 Flash-Memory Overview

Figure 4.3 gives a block-diagram of a NAND flash based SSD. In the flash-memory, data is stored in an array of flash blocks. Each block spans $32 - 64$ pages, where a page is the smallest unit of a read and write operation. A distinguishing feature of the flash-memory is that the read operations are faster than those of magnetic disk drives. Moreover, compared with disks, random read operations are much faster on the flash media, as there is no mechanical head movement.

The Flash Translation layer (FTL) is an intermediate software layer inside the flash-based SSD that maintains a mapping table of logical addresses (e.g., those from file

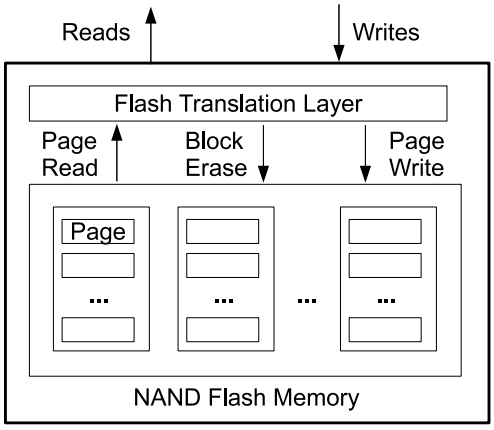

Figure 4.3: Flash-based Solid State Drive (SSD)

systems) to the physical addresses on the flash. It makes a flash-memory device act like a normal block device by only exposing the read/write operations to the external world and hiding the presence of the erase operations, unique to the flash-memory based devices. Flash-memory handles the read and write asymmetrically. While a flash device can read any of its pages, it may only write to one that is *erased*. Erase operations are performed in block granularity, rather than page granularity, since page-level erases are too costly. The typical access latencies for read, write, and erase operations are 25 μs , 200 μs , and 1500 μs respectively [39]. The *in-place updates* would require an erase-per-update, resulting in severe performance degradation. To work around this, for a write operation to a logical page L_p, the FTL writes data into a new physical page P_p, and updates its mapping table to reflect this change. The previous physical page P'_p is marked as invalid. These out-of-place updates bring in the need for FTL to deploy a garbage collection (GC) mechanism.

Besides the *in-place update* problem, the flash-memory exhibits another limitation – a flash cell can only be erased a limited number of times (e.g., 10K–100K) [64, 39]. FTL uses various wear leveling techniques [65, 40, 66] to even out the erase counts of different blocks in the flash-memory to increase its overall longevity. Recent studies show that current FTL schemes are very effective for workloads with sequential access write patterns. However, for workloads with random access patterns, these schemes

illustrate very poor performance [41][8][42]. One of the design goals of this chapter is to use the flash-memory in a FTL-friendly manner in BloomStore design.

Recently, PCI Express (PCIe) SSDs emerged in the market and are proven to be superior in access performance over their SATA counterparts [67]. Consequently, PCIe interfaced SSDs are widely deployed in industrial environments where applications (e.g., MySpace [68]) render more distinct access patterns (high throughput and low latency) than typical personal devices. In this chapter we evaluate our BloomStore design on both the PCIe flash based SSD and the SATA interfaced one.

4.4 BloomStore Design

Our KV store design is driven by the goal to deliver high key lookup/insertion throughput to meet the demands of the recent KV store applications at the minimum RAM space usage. We use the following performance metrics to evaluate the effectiveness of our design:

1. **Amortized RAM overhead per KV pair**: It measures how frugal the RAM space usage is that a KV store design can achieve. It is defined as the consumed RAM space (by the KV store) divided by the total number of KV pairs in the store. This metric can be further decomposed into two parts: (1) the amortized RAM overhead to index the key; and (2) the amortized RAM overhead to buffer the key in the data buffer. The rationale behind this metric is as follows. The RAM size imposes a big scalability challenge to the existing KV store designs. As the growth rate of the flash-memory capacity is nonlinearly faster than that of the RAM capacity (e.g., while the RAM capacity is still in the ten or tens of GBs level, a $1,024$GB SSD is already in market), it becomes impractical for a KV store design to use an amount of RAM space proportional to the overall flash space in the store. Our KV store design overcomes the limitation by partitioning the overall flash space and placing the index structures corresponding to each partition on the flash. With this design choice, our design uses extremely frugal RAM space (e.g., < 1 byte/key) for the index and data buffers. As a byproduct of this design, when an unexpected event like a power failure occurs, our design is expected to experience a much shorter service outage than those designs that

store index structures in the RAM. This is because the latter ones need to scan and reprocess all stored KV pairs for reconstructing the index structure in the RAM.

2. **KV lookup/insertion throughput**: These two metrics measure the performance of the basic KV store operations, crucial in order to meet the demands of the recent KV store applications. Our design should be able to deliver high lookup and insertion throughputs to be comparable to the state-of-the-art design (SkimpStash) with the same amount of RAM space.

4.4.1 Overall Architecture

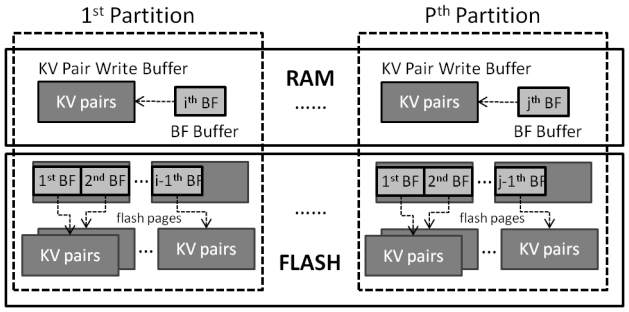

Figure 4.4: BloomStore architecture: multiple partitions, a KV pair write buffer (flash page size) and a BF buffer (one active BF and more) in RAM for each partition, and a chain of BFs and their associated data (KV pairs) pages in flash for each partition (one BF per data flash page, having a pointer to the flash page)

Figure 4.4 shows the overall architecture of our KV store called BloomStore. The BloomStore divides the flash space into a number of logical partitions and uses a Bloom Filter (BF) based index structure. In each partition, inserted KV pairs are first buffered into a flash-page sized in-RAM data buffer (a KV pair write buffer). This KV pair write buffer has its corresponding in-RAM BF (in a BF buffer) that summarizes the inserted keys in that write buffer. When the KV pair write buffer is filled up, all the KV pairs in the buffer are sequentially written to a flash page with a flash write operation. In addition, its corresponding BF is flushed (appended) into a chain of BFs stored in the flash. The BloomStore associates a BF with a single flash page of KV pairs, where the BF has a flash pointer to the flash page.

To reduce the amortized RAM space overhead per KV pair, BloomStore only maintains a flash-page sized data buffer and a very small sized BF buffer per partition in RAM. To achieve high lookup/insertion throughput, BloomStore reduces the number of flash page reads and writes in the following ways: (1) it reduces the maximum number of flash page reads for key lookups by flash partitioning; (2) in each partition, for the KV pair insertion, it maintains a dedicated flash page sized KV pair write buffer to temporarily buffer the inserted KV pairs and writes the KV pairs in a single flash page sequentially; and (3) when extra RAM space is available, it maximizes the number of buffered BFs in the RAM with a RAM-space efficient BF buffer organization scheme. As a result, the key lookup can reduce the number of flash page reads for the associated BFs. The key insertion can reduce the number of the BF buffer flushes into the flash.

Hereafter, we will describe each of the BloomStore components in more detail. To begin with, each BF in BloomStore consists of a flash pointer (4 bytes) and a bit vector for the normal BF operation. The flash pointer points to a logical block address of its associated flash page of KV pairs.

KV Pair Write Buffer (per partition): Each partition has a dedicated fixed-sized in-RAM data buffer to log incoming KV pair insertions for a partition (called a KV pair write buffer). Its size is equal to the underlying flash page size. This buffer is flushed (written) into the flash only when it is filled up with the inserted KV pairs. Thus, the number of flash page writes can be reduced. If more stringent durability is required, a configurable timeout interval (e.g., 1 ms) can be used to ensure that a flash write occurs in the interval.

BF Buffer (per partition): Each partition keeps its very small sized active BFs in the BF buffer. All other BFs for the partition are stored in the flash instead. The BloomStore allows a single BF to hold as many keys as the maximum number of KV pairs stored in a flash page. The active BF in the BF buffer represents the membership of the keys stored in the KV pair write buffer. Thus, when the KV pair write buffer is written into the flash, the active BF (in the BF buffer) is also flushed into the flash. At this time, the flash pointer of the active BF is updated with a valid logical block address of the flash page. This BF buffer flush involves a number of flash page accesses: (1) read the flash page containing a chain of other BFs in the partition into an (temporary) in-RAM buffer (so called remainder of the BF chain); (2) append the active BF into

the buffer; and (3) write the buffer into the flash. The location information for the flash pages of BFs in each partition are kept in the RAM.

The minimum RAM usage required for each partition (per-partition minimum RAM usage) includes a BF size of a BF buffer and a flash page size of a KV pair write buffer. Then, **the minimum RAM usage** can be computed by multiplying the per-partition minimum RAM usage with the number of partitions. With sufficient RAM space (larger than the minimum RAM usage), our current design employs the extra RAM space (the available RAM space minus the minimum RAM usage) for the extra BF buffering (Alternatively, the extra RAM space can be used for accommodating an enlarged BF size; we will investigate this issue in a future work). Thus, each BF buffer can hold the active BF and a number of BFs whose data flash pages of KV pairs have been already written into the flash. By buffering more than one BF in RAM, we can expect performance benefits for key lookup and insertion by reducing the number of flash page reads to fetch the associated BFs into the RAM and by decreasing the number of the BF buffer flushes into the flash. The issue is then how to allocate the extra RAM space to each BF buffer. In our design, we have all BF buffers share the extra RAM buffer space instead of equally assigning each partition to an individual BF buffer space.

BF Chain (per partition): The BloomStore indexes each partition independently with a sequence of BFs (briefly a BF chain), each of which memorizes the inserted KV values in one flash page. For each partition, the associated BF chain is decomposed into two parts: (1) **the active BF** that is most recently instantiated in the chain and is always stored in RAM (the BF buffer); and (2) **the remainder of the BF chain** that consists of the rest of BFs in the chain and is stored in the flash. With the extra RAM space, more than one BF per partition can reside in the RAM. For example, in the first partition of Figure 4.4, the active BF is the i^{th} BF. The remainder of the BF chain is a list of BFs including the 1^{st} BF through the $i-1^{th}$ BF.

The BF chain length is defined as the number of BFs in the BF chain. By definition, the BF chain length is highly correlated with the size of each partition in the flash (briefly the partition size). The partition size is computed by multiplying the BF chain length with the sum of the flash page size and the BF size. As the BF chain length increases, the partition size increases, reducing the number of partitions for the same amount of KV pairs. Thus, the minimum RAM space required for all partitions is

reduced accordingly. On the contrary, a larger BF length causes a higher number of false positive errors, because each partition has a larger number of BFs to examine during each key lookup (See Figure 4.5 in Section 4.4.4).

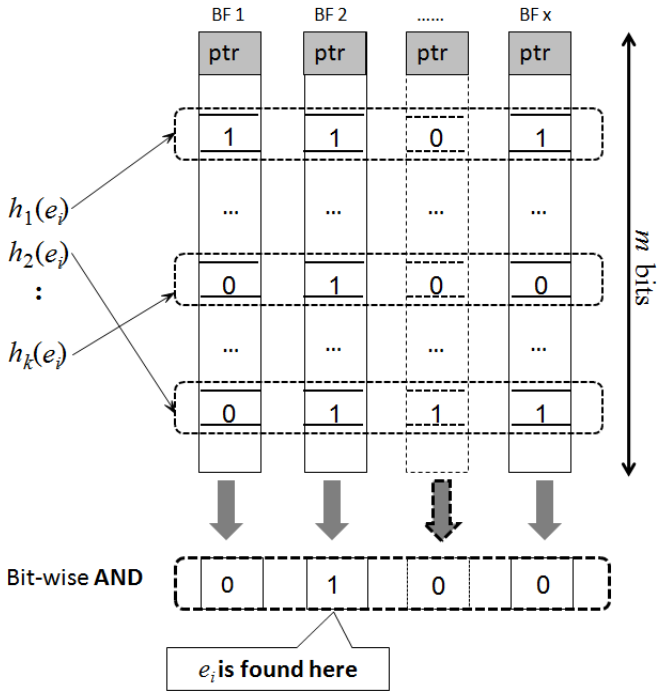

Figure 4.5: Illustration of checking multiple bloom filters in parallel

Bloom Filter Parallel Lookup: To look up a key e_i, BloomStore first uses a hash function to locate an associated partition, followed by checking the key e_i in a chain of BFs associated with the partition. Assume that each BF buffer can contain an active BF, where no extra RAM space is available for the BF buffering. Initially, a BF chain in the partition contains only an active BF in its BF buffer, i.e., no BFs are stored in the flash. Thus, simply one BF lookup operation with the active BF is enough to check the key. Assume that x flash pages of KV pairs in the partition have been written to the flash. It implies that there is a BF chain containing x BFs. With the minimum BF buffer size, only the active BF is in the BF buffer, while the rest of the BF chain (containing other $x-1$ BFs) is in the flash. Thus, we need to read the $(x-1)$ BFs from the flash to an (temporary) in-RAM buffer and then check the key in each of the x BFs. Note that the temporary buffer to hold the BFs can be shared by all the partitions

(causing negligible RAM space overhead). Simply, each of x BFs should be checked separately, requiring x separate BF lookup operations.

However, BloomStore adopts the parallel BF checking scheme proposed in the SegmentedHash [69] to check the buffered BFs in parallel. Figure 4.5 illustrates an example of checking x BFs in parallel. Each m-bit BF (represented as a column) consists of a bit vector and a flash pointer pointing to a flash page of KV pairs. All x BFs use the same group of k different hash functions. Thus, for a given key e_i, the same bit positions are checked in each BF. As such, a row of bits can be composed of one bit from each of the x BFs, as illustrated in the Figure 4.5. To look up a key e_i in x BFs, k different bit positions are selected from a column using k different hash functions h_1, ..., h_k. For each of the k bit positions selected, it accesses a row of the length of x bits and executes bit-wise **AND** operations on k rows. The 1's position in the resultant row of bits indicates the BF where the key was found. If the resultant row of bits is all zero, according to the BF property, the key e_i is not present in any of the x BFs. However, if the resultant row has a single 1, the flash page associated with the BF of the 1's position should be read to search the key from the set of KV pairs contained in the flash page. As an example, Figure 4.5 shows that the key e_i is found in BF 2. Still, a chance exists that no KV pairs have the key (called false positive error). Possibly, the resultant row can have multiple 1's, requiring to read and check more than one flash page (from the largest BF label, implying the most recently written BFs) to find the key. Note that as the number of BFs increases in a partition, the number of false positive errors increases exponentially.

It is worth noting that the key lookup operation bearing such parallel lookup scheme is bottlenecked by the read throughput of the SSD, as the entire remainder of a BF chain (e.g., $x - 1$ BFs of a BF chain containing x BFs) may need to be read from flash-memory for a parallel lookup. If the value x goes beyond the number of internal data channels/buses of the SSD, one possible way to boost the key lookup performance is to pipeline the BF chain retrieval process with the parallel lookup process. In the future, we plan to expand our BloomStore design with this feature.

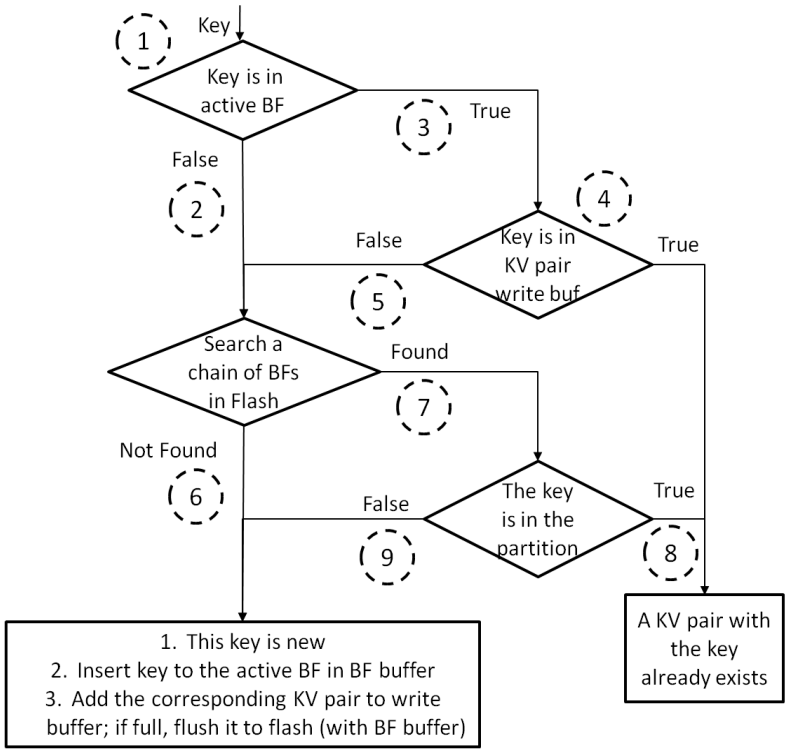

Figure 4.6: Flowchart of the KV store operations: key lookup and KV pair insertion for a partition

4.4.2 KV Store Operations

To help to understand the relationship of different components in BloomStore, we demonstrate the operations of key lookup and KV pair insertion in a partition (see Figure 4.6). In the flowchart, we assume the same data dedup application logic as in our KV store application; i.e., if a key (e.g., a chunk ID) lookup is answered negatively, a successive KV insertion will be triggered to add the corresponding KV pair (e.g., chunk metadata) to the store. For simplicity in our explanation, we assume each BF buffer can hold only one active BF.

Key Lookup: To look up a key in a partition, (1) BloomStore looks up the key at the active BF (stored in the BF buffer) in the associated BF chain; (2) if the key is not found in the active BF, it reads the remainder of the BF chain stored in flash to RAM and performs the BFs parallel lookup for the key; (3) if the key is found in

the active BF, BloomStore proceeds to check the key in the KV pair write buffer; (4) if the key is in the write buffer, the key lookup operation returns an affirmative value; (5) if the key is not found in the write buffer, BloomStore reads the remainder of the BF chain to RAM and performs the BFs parallel lookup on them for the key; (6) if a key is not found in any of the BFs in the BF chain, the key is considered new; (7) for each BF in the BF chain where the key was found, a flash pointer will be extracted and followed to search the key in its corresponding flash page containing the KV pairs; in the case where more than one flash page needs to be searched, BloomStore searches the pages by the reverse order of their write times (from the largest BF label); BloomStore then stops its lookup and returns affirmatively upon finding the first KV pair whose key matches the the searched one; (8) if the key is found in that partition, the lookup operation returns an affirmative value; and (9) if the key is not found in that partition, the lookup operation returns a negative value.

KV Pair Insertion: This operation in a partition inserts (updates) a KV pair into its KV pair write buffer and inserts its key to the active BF associated with the partition. When the KV pair write buffer becomes full, all KV pairs in the buffer are written to flash with a flash page write. In addition, BloomStore updates the associative BF chain for that partition by moving (appending) the currently active BF to the remainder of the BF chain (update operation) and instantiating a new active BF in the BF buffer. This update operation requires BloomStore to fetch the remainder of the BF chain from flash into RAM, append the active BF to the remainder, and write back the updated remainder to the flash. Moreover, if this insertion corresponds to an update operation on an earlier inserted key, the most recent value of the key will be (correctly) retrieved during a key lookup operation as the older value was stored closer to the head in the partition.

KV Pair Deletion: A delete operation on a KV pair is supported by inserting a *null* value for the key (becoming a garbage KV pair). When the flash usage or a fraction of garbage KV pairs in the flash exceeds a predefined threshold, a garbage collection procedure (different from the garbage collection inside the flash-memory) begins to reclaim the flash space in a way similar to that in the log-structured file systems [70]. The garbage collection starts scanning the KV pairs from the earliest written flash pages. It discards the garbage KV pairs as well as the BFs and moves the valid ones (from

the head to the tail of the flash log); the garbage collection stops when the fraction of garbage KV pairs decreases under a certain threshold value.

4.4.3 Design Enhancement: Pre-filter

A key lookup operation, as indicated in the flowchart in Figure 4.6, may generate unnecessary flash page reads caused by loading chains of the BFs and the KV pair data pages from the flash when looking up the non-existent keys. Upon a key lookup miss in the BF buffer, the remainder of the BF chain will be read into RAM to check for the key.

Many real applications frequently perform lookup operations on the keys that do not exist in the KV store (briefly non-existent keys). For example, primary file system deduplicaiton (refer to the property of the *Vx* workload in Section 4.5) usually finds many new chunk IDs through the dedup process. Microsoft LIVE Primetime on-line multi-player game [71] frequently uses lookups on non-existent keys to implement the game logic [37]. These cause many unnecessary flash page reads because BloomStore reads the remainder of the BF chain and its associated data KV pair pages from the flash to look up the non-existent keys.

Therefore, for workloads that search many non-existent keys from the KV store, BloomStore maintains a fixed sized prefilter in RAM that filters out the non-existent keys before reading a chain of BFs from the flash. The rationale behind using a BF as our pre-filter is as follows: (1) The BF has a unique characteristic that it is free of false negative errors (i.e., if a key is not found in a BF, it is for sure a non-existent one), regardless of the size of the BF. Hence, we could freely adjust the size of the pre-filter according to the available RAM space without worrying about missing any keys; and (2) with a fairly small RAM footprint (e.g., 4 bits/key), the BF is able to identify and filter out the major proportion of the non-existent keys observed in the key lookup and insertion process. In statistics theory, this capacity is referred to as the specificity of the tests [72], defined as $\frac{number\,of\,true\,negatives}{number\,of\,true\,negatives+number\,of\,false\,positives}$. As the BF is prominent for its much lower false positive probability over many other data structures with the same RAM footprint [48], its specificity value is expected to be high (close to 1).

For the same total number of keys and fixed available RAM space, assigning a bigger pre-filter will filter out a larger portion of lookups for the non-existent keys, owing to

the less incurred false positive errors. Nevertheless, more RAM space consumed by the prefilter implies that less RAM space can be used for the BF buffering (RAM space taken by all write buffers remains unchanged for the same amount of allocated partitions). This will make potentially more BF chains be retrieved. Therefore, BloomStore has to choose the prefilter size carefully in order to optimize the lookup throughput. We have evaluated the prefilter in extensive experiments with two real workloads of different characteristics and verified its significant impact on one of the workloads.

4.4.4 Understanding Trade-Offs

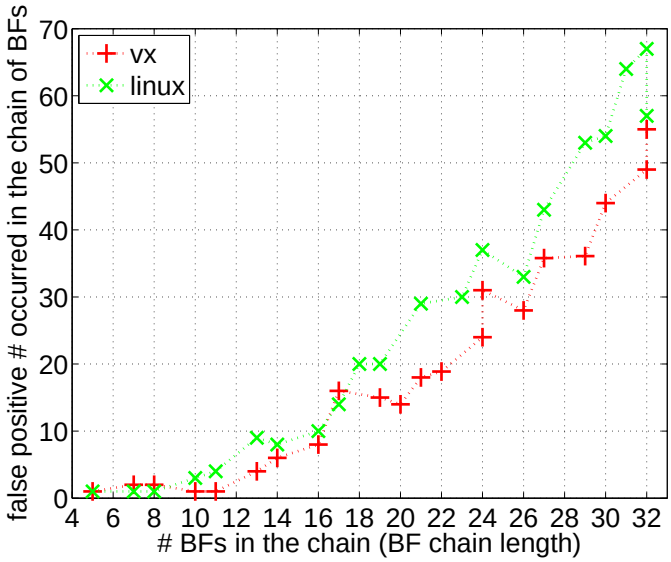

Figure 4.7: The number of false positive errors as BF chain length varies

It is reasonable to assume that the flash-memory size is big enough to hold all the KV pairs in BloomStore. We can obtain the maximum number of partitions (say P_{max}) by simply dividing a given RAM size with the per-partition minimum RAM usage (the sum of a flash page size and a BF size). Simply using the P_{max} can be ineffective in BloomStore, because each BF buffer can hold only a single active BF. Then it frequently issues the BF buffer flush into the flash, requiring many flash page reads and writes. Since the flash page size is fixed, the number of partitions (P) can be decreased by enlarging a BF size or increasing the number of BFs in the BF buffer. However, with

a smaller P, the number of BFs in a partition (the BF chain length) grows linearly, increasing the number of false positive errors exponentially.

Hereafter, we further discuss the performance trade-offs for the BF size and the BF chain length.

BF Size (δ): A BF in BloomStore is associated with a single flash page. Thus, the number of keys for a BF can be computed by dividing the flash page size with a KV pair size. When assigning one byte per key [61], we can compute a reasonable BF size. Still, a larger BF size helps to further reduce the number of false positive errors. However, it consumes more RAM space, reducing the number of BFs that can be held in RAM (the BF buffer). This will potentially retrieve a larger number of BFs from the BF chain in the flash during a key lookup. In addition, buffering a smaller number of BFs in RAM will trigger more frequent BF chain updates for key insertion.

BF Chain Length (M): A larger BF chain length (correlated with each partition size) in BloomStore increases the number of KV pairs stored in each partition. This results in a smaller number of divided partitions. It eventually helps to reduce the RAM usage consumed by the KV pair write buffers for the partitions (or the minimum RAM usage). However, a larger BF chain length can degrade the key lookup performance because more BFs are required to be checked for each key lookup operation. More importantly, as the BF chain length increases linearly, the number of false positive errors occurs potentially nonlinear fast (as illustrated in Figure 4.7) for the following two reasons. Firstly, for any given key search, the effective false positive probability has increased by a factor greater than M since many combinations must be considerered. In fact, assuming M independent BFs, the probability of exact i ($i \leqq M$) false positive errors is as follows:

$\Pr\{i \text{ false positive}\} = \binom{M}{i} \times f^i \times (1-f)^{M-i}$, where $f \approx (1-e^{-kn/m})^k$ is the previously defined false positive probability of a single BF. Consequently, the overall probability of a false positive is:

$\Pr\{\text{false positive}\} = \sum_{i=1}^{M} \binom{M}{i} \times f^i \times (1-f)^{M-i}$, a nonlinear function of M.

Secondly, a key search that results in a number of false positives may re-appear in the workload. At each appearing time, the key will cause the same number of false positives. Data dedup workloads usually contain repeated key searches, making a false positive key likely to trigger multiple times more false positive errors. As each false

positive error wastes a flash page read of KV pairs, this may rapidly worsen the key lookup performance. On the contrary, with a smaller BF chain length, the RAM usage consumed by the KV pair write buffers for the partitions will be very high. Correspondingly, a smaller number of BFs can be kept in the BF buffer for each partition that will degrade the key lookup/insertion throughput.

4.5 Performance Evaluation

We compare our BloomStore with BufferHash and SkimpyStash by using realistic workloads obtained from the data dedup applications in terms of the following performance metrics: (1) amortized RAM overhead per KV pair and (2) key lookup/insertion throughput.

4.5.1 Experiment Setup

Table 4.1: Properties of two data deduplication workloads

Workload name	Total # lookup & insert operations	Lookup:insert ratio	Key/value size (byte)
Linux	$12,427,697$	$4.1:1$	20/44
vx	$14,628,873$	$1.6:1$	20/44

The BufferHash is excluded from throughput comparisons, because the RAM overhead analysis shows that its amortized RAM overhead per KV pair is ten times higher than the others. We will discuss this issue in more detail in the following pages. We implement BloomStore and SkimpyStash by using Python. MurmurHash [73] is adopted to realize the hash functions used in our BF implementation (with different seeds) and to compute a hash table index entry in SkimpyStash. Both BloomStore and SkimpyStash are built on top of a raw block device interface, implying there are no file-system related effects, such as buffering, caching, and prefetch.

We run the implemented BloomStore and SkimpyStash on a typical server having Intel Xeon L5530 Quad Core 2.4GHz and Linux kernel 2.6.32 (ubuntu 10.04). The server is also attached with a prototype Micro 1TB NAND flash based SSD through a high-speed PCIe interface. The physical flash page size of the SSD is 4KB. In addition, to

study the performance of our design on a SATA interfaced SSD, we run both BloomStore and SkimpyStash with an INTEL X25E 32GB SATA interfaced NAND flash based SSD.

We use two real-world data dedup workloads: *Linux* and *Vx* workloads. *Linux* is a typical data dedup workload collected from a data backup environment. This workload was obtained from the Linux kernel source backups which consist of 20 different versions of the Linux kernel source distributions (Linux-2.6.30.1 – Linux-2.6.30.10, Linux-2.6.35.1 – Linux-2.6.35.8, Linux-2.6.36.1, and Linux-2.6.36.2). As with typical data backup streams, this workload contains many duplicates that will exhibit a higher lookup/insertion ratio. The *Vx* stands for another breed of data dedup workload that was obtained from a primary file system environment. We investigate BloomStore performance on *Vx*, because the data deduplication on the primary file systems has recently drawn increasing attention from the data storage community. For example, Meyer et al. [2] studied the data dedup performance on a collection of 857 file systems running Windows at Microsoft. The *Vx* workload was collected by crawling a networked primary file system shared by a group of software engineers. Below are the statistics for the two workloads: (1) the *Linux* workload contains $10,000,000$ total chunks and $2,427,697$ unique chunks, while the *Vx* workload contains $9,000,000$ total chunks and $5,628,873$ unique chunks; and (2) the ratios of the key lookup operations to KV pair insertion operations in the *Linux* and *Vx* workloads are 4.1:1 and 1.6:1, respectively. The *Linux* workload has a higher key lookup/insertion ratio than the *Vx* workload. In addition, the *Vx* workload looks up *many non-existent keys* in the KV store, while the *Linux* workload searches *many already existent keys* due to its high data duplication. The properties of the two workloads are summarized in Table 4.1. In these dedup applications, each key is a 20-byte SHA-1 hash value of the corresponding data chunk, while the value is a 44-byte metadata for the chunk. Thus, the size of each KV pair for the two workloads is 64 bytes.

4.5.2 Amortized RAM Overhead

We analyze the amortized RAM overhead per KV pair for BufferHash, SkimpyStash, and BloomStore. For this analysis, we assume that the flash page size is 4KB and adopt the commonly used 64-bytes KV pair length for data dedup applications [8, 61, 42]. The 64-byte KV pair combines a 20-byte key (a SHA1 hash as a chunk ID) and a 44-byte

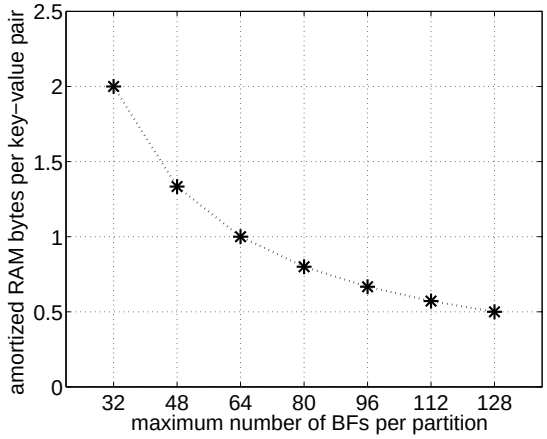

Figure 4.8: Amortized RAM overhead per KV pair for BloomStore as the number of BFs per partition increases

value (chunk metadata for encoding chunk location, chunk size and chunk offset, etc.).

BufferHash (10 bytes/key): With a 16-byte KV pair length and a maximum number of 16 hash tables per partition, BufferHash consumes a 4-byte amortized RAM overhead per KV pair [60]. The amortized RAM overhead grows linearly as the KV pair length increases; i.e., the 64-byte KV pair length increases the amortized RAM overhead per KV to 10 bytes/key.

SkimpyStash (practical range: $(1, 4]$ bytes/key): To mitigate the impact of false positive errors on key lookup throughput, SkimpyStash pays a relatively high RAM overhead; i.e., 1-byte RAM footprint per key represented in a BF. The amortized RAM overhead per KV pair can be computed as follows: $1 + \frac{pointer\ \ size}{average\, bucket\, length}$, where the pointer size is set to 4 bytes [61]. Its smallest amortized RAM overhead per KV pair in theory approaches 1 byte/key, as the average bucket length becomes extremely large. (In our experiments, the smallest amortized RAM overhead was 1.2 bytes/key.)

BloomStore (practical range: $[0.5, 1]$): BloomStore utilizes the RAM space for two purposes: (1) buffering the inserted KV pairs for the partitions and (2) buffering the BFs for the key insertions for the partitions. The amortized RAM overhead for the first is usually several times bigger than that for the second. With the (maximum) BF chain length per partition of $M = 96$, each partition could store up to $\frac{4,096}{64} \times 96 = 6144$ KV pairs. Then, the amortized RAM space overhead of the write buffer per KV pair becomes

$4,096/6,144 = 0.667$ byte/key in a fully filled partition. The amortized RAM overhead of the BF buffer per KV pair can be safely omitted (assuming the minimum size case, where there is only one active BF buffered in RAM for each partition and the BF size is of 64 bytes) because it is tens of times smaller than that of the KV pair write buffer (e.g., assuming with a minimum BF size of 64 bytes and only one active BF buffered in RAM for each partition: $\frac{64}{64\times96} = 0.01$ byte/key vs. 0.667 byte/key). Figure 4.8 illustrates that as the M (the maximum number of BFs in a partition) increases, the amortized RAM space overhead per KV pair decreases non-linearly. Notice that with $M = 128$, the amortized RAM space overhead becomes 0.5 bytes/key. Theoretically, increasing the maximum number of BFs in a partition can further reduce this overhead, whereas it increases the number of false positive errors dramatically. In our experiments, we vary the number of BFs per partition to 64, 96, and 128 BFs to study its effect on the performance.

4.5.3 Experiencing the Trade-Offs

Table 4.2: Summary of Performance results with different configurations of the BF chain length, $M = 128$, 96 and 64 for Vx workload (BF chain read # increases when the remainder of the BF chain is read into RAM; data page read # increases with a flash page read of KV pairs)

	M	# of BF chain reads	# of data page reads	lookup through-put	insert through-put
A	128	$27,800$	$279,700$	$381,400$	$507,600$
B	96	$48,500$	$244,600$	$399,300$	$508,400$
C	64	$119,600$	$247,000$	$325,700$	$500,600$

With the two data dedup workloads and our understanding of the trade-offs in Section 4.4.4, we will empirically investigate the performance trade-offs of the different configurations of the parameters: the BF chain length (M) and the BF size (δ). For our experiments, we vary the BF size starting from 64 bytes based on the following rationale: (1) a KV pair size of the workloads is fixed to 64 bytes, and a flash page size is 4KB; then, a single BF contains 64 keys; (2) to achieve a roughly 1 byte footprint per key as in SkimpyStash, the BF size for BloomStore can be set to 64 bytes, where 4

bytes are occupied by a flash pointer and the remaining 60 bytes are in effect used for the bit vector.

BF Chain Length ($M = 96$ for *Vx* and *Linux*): To study the impact of the BF chain length, each configuration is assigned the same amount of RAM space (including BF buffers and KV pair write buffers). Each BF size is set to the same size (e.g., 64 bytes). In addition, to make sure that BloomStore under each configuration has roughly the same footprint on the flash, the number of partitions is tuned accordingly within each configuration. Table 4.2 summarizes the performance results of the three different configurations of the M parameter. Observe that while the three configurations have almost the same key insertion throughout, the second configuration (B) of $M = 96$ achieves the best key lookup throughput. The second configuration issues a remarkably smaller number of data page reads when compared to the first (A) (As the BF chain length increases, the number of false positive errors creating more flash page reads grows exponentially). The second configuration consumes less than half of the BF chain reads compared with the last one (C). Thus, based on this experiment, we can conclude that the second configuration of $M = 96$ strikes a balance between the total number of BF-containing (BF) page reads and KV pair-containing (data) page reads so as to minimize the overall time consumption of the flash page reads. With a similar procedure, we also observe that $M = 96$ achieves a good performance for the *Linux* workload.

BF Size ($\delta = 64$ for *Vx*, $\delta = 128$ for *Linux*): Based on the obtained BF chain length ($M = 96$) for the two workloads, we further evaluate the impact of the BF size on the key lookup/insertion throughput. Due to space limitations, we only show graphs for the *Vx* workload. The amortized RAM overhead per KV pair is set to 1.3 bytes for all runs, consuming the RAM space of 198KB. Recall that a larger BF size implies that potentially more flash pages are used to store the chain of BFs in a partition. Thus, the choice of a BF size can be also expressed as the choice of a BF chain size ($M \cdot \delta$).

Figure 4.9 illustrates the throughput (lookups or insertions per second) for the different BF chain sizes (4KB, 6KB, 8KB, 10KB, and 12KB) on the *Vx* workload. Figure 4.10 demonstrates the corresponding overhead for the BF chain reads and data (KV pairs) page reads. We can conclude that the minimum overhead is achieved when the (maximum) BF chain size is 6KB, implying that the BF size is 64 bytes.

We also evaluate the choices over the BF size on the *Linux* workload. The best BF

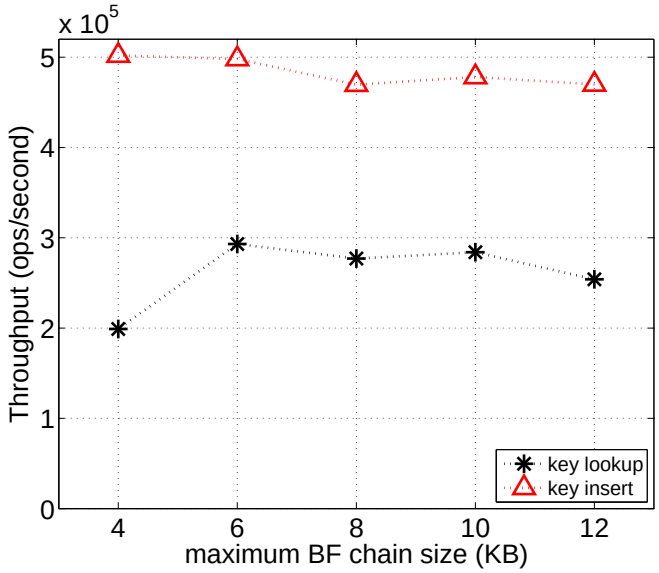

Figure 4.9: *Vx* workload: Throughput (lookup or insert) (ops/sec) for the different values of the BF chain size (KB)

size has been observed as 128 bytes rather than 64 bytes. One reason behind this is that the *Linux* workload has a high key lookup/insertion ratio. Thus, the reduction in the number of flash page read operations benefiting from incurring less false positive errors per lookup operation outweighs the extra flash page read overhead resulting from retrieving a larger size of the BF chain remainder. Recall that the BloomStore retrieves a BF chain remainder only when the key lookup has not been found in the data pages corresponding to the buffered BFs.

4.5.4 Effectiveness of Prefilter

Table 4.3: *Linux* workload: RAM usage (KB) decomposition for different configurations

RAM usage decomposition	BF buffer	prefilter overhead	KV pair write buffer
base	1,302	0	1,648
base+prefilter	807	495	1,648

For fairness in comparing the BloomFilter (*base*) and the BloomFilter with the

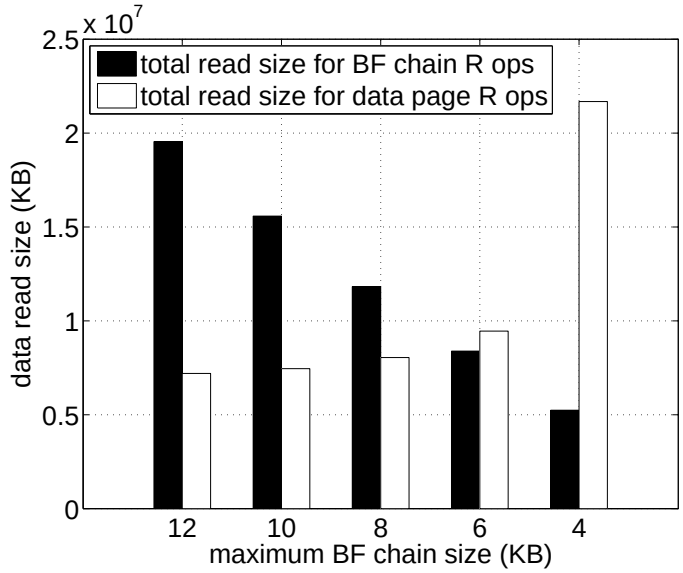

Figure 4.10: *Vx* workload: Data read size decomposition for different values of BF chain size (KB)

Table 4.4: *Vx* workload: RAM usage (KB) decomposition for different configurations

RAM usage decomposition	BF buffer	prefilter overhead	KV pair write buffer
base	3,066	0	3,840
base+prefilter	186	2,880	3,840

prefilter (*base+prefilter*), the *base+prefilter* configuration is given a smaller amount of BF buffer space to compensate for the prefilter RAM usage. Table 4.3 and 4.4 present the RAM usage decomposition of different components (BF buffer and prefilter) with respect to the two different configurations for both workloads. For the two workloads, the amortized RAM overhead per KV pair is 1.2 bytes. Figure 4.11 and 4.12 illustrate the key lookup throughputs for the *Vx* and *Linux* workloads, respectively. In the figures, each group of three bars represents the number of the BF chain reads that represents how many times the remainder of the BF chain in the flash is fetched into RAM (the lower the better), the number of the (KV pair) data page reads (the lower the better), and the key lookup throughput (the higher the better). We omit presenting the insertion throughput, because the prefilter enhancement is only for improving the key lookup

throughput. Notice that the improvement contributed by the prefilter is remarkable in the *Vx* workload. The reason is that a major fraction of the lookup operations is used to check non-existent keys in the workload, which could be avoided by the prefilter. On the other hand, for the *Linux* workload, the prefilter does not help significantly because a major portion of the lookup operations is spent to check those already-existent inserted keys.

4.5.5 Key Lookup & Insertion Throughput

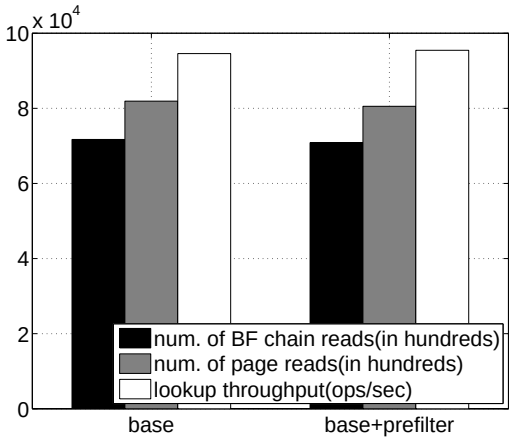

Figure 4.11: *Linux* workload: Impact of prefiltering on key lookup throughput (ops/sec)

Figure 4.13 and 4.14 present the throughput comparisons for BloomStore and SkimpyStash over the *Linux* and *Vx* workloads, respectively. The available RAM space is the same for both designs. For fairness, the additional RAM space required by the extra feature of BloomStore (prefilter) is consumed in the given RAM space. In particular, Figure 4.13 illustrates the results on two different types of SSDs: the upper two curves plot the results obtained from the Micro PCIe SSD, while the lower two curves plot the results obtained from the INTEL SATA SSD. We omit the results of both designs with the INTEL 32GB X25E on the *Vx* workload because they follow the same trend. The BloomStore is configured for the two different workloads as follows: (1) for the *Linux* workload, $M = 96$, $\delta = 128$, and no prefilter (the unused space is used for more BF buffering); (2) for the *Vx* workload, $M = 96$, $\delta = 64$, and with a prefilter.

The following observations are made from the figures. First, both BloomStore and

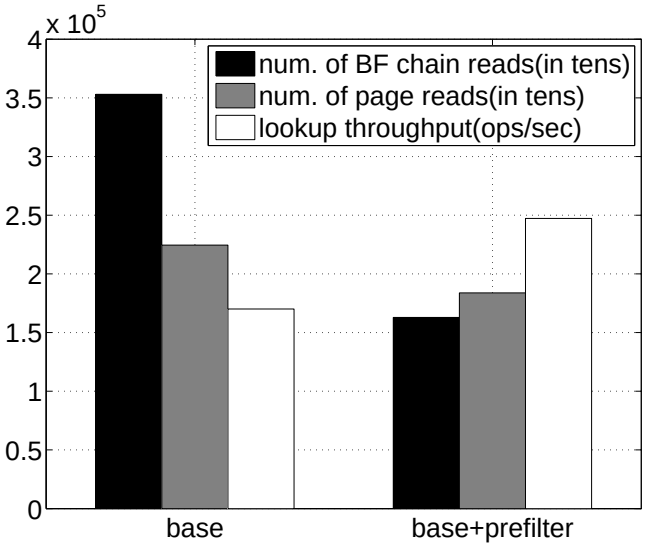

Figure 4.12: *Vx* workload: Impact of pre-filtering on key lookup throughput (ops/sec)

SkimpyStash achieve significantly higher key lookup throughput on the *Vx* workload than on the *Linux* workload. One reason behind this is that the lookup ratio in the *Linux* is much higher than that in the *Vx* (4.1 vs. 1.6, a factor of 2.56 times higher). Secondly, BloomStore delivers a higher key lookup throughput than SkimpyStash. Thirdly, as the available RAM space declines, the lookup throughput drops in both designs. However, the lookup throughput of SkimpyStash drops slightly faster than that of BloomStore. In particular, on the *Linux* workload, as the amortized RAM overhead per KV pair decreases from 1.8 bytes to 1.2 bytes, the lookup throughput of SkimpyStash descends by 2.9 times, which is much steeper than BloomStore. It is worth noting that as the amortized RAM overhead per KV pair grows, the lookup throughput for BloomStore increases monotonically and saturates after reaching a 2.8-byte amortized RAM overhead per KV pair (see Figure 4.13). The reason behind this is that from this point, all the BF chains are completely buffered in the RAM. Only the number of data page reads affects the lookup throughput. In fact, for the *Linux* workload, if more RAM space is allowed beyond the 2.8 bytes/key, one approach to further improve the lookup throughput is to increase the BF size, so as to minimize the wasted flash page reads. To minimize the impact of false positive errors on the key lookup throughput, SkimpyStash pays relatively high RAM overhead for maintaining BFs for each bucket in RAM, e.g.,

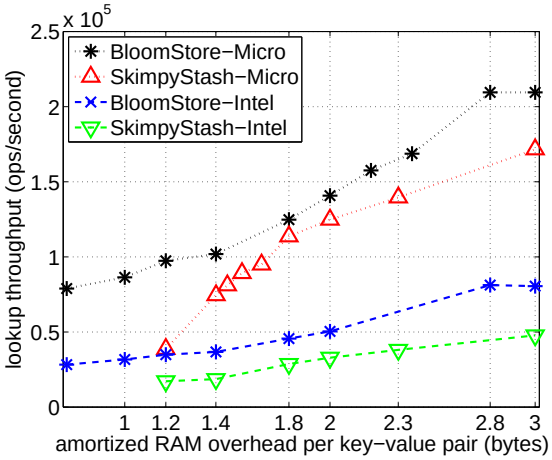

Figure 4.13: *Linux* workload: Key lookup throughput (ops/sec) comparisons between BloomStore and SkimpyStash as the amortized RAM overhead per KV pair varies

1-byte RAM footprint per key. Therefore, there are no results obtained from SkimpyStash corresponding to the ≤ 1 amortized RAM overhead per KV pair. On the contrary, BloomStore could easily achieve reasonably a good lookup throughput with the sub-byte range RAM usage per KV pair. As shown in Figure 4.13, even with 0.72-byte amortized RAM overhead per KV pair, BloomStore is still able to deliver $78,879$ ops/second key lookup throughput, only 22.5% lower than what is achieved by the doubled (1.44 bytes) amortized RAM overhead.

As for the insertion performance with the two workloads, BloomStore performs slightly lower (1.5% and 1.4% for *Linux* and *Vx* than SkimpyStash, which can be explained by the extra overhead of the periodical BF buffer flush operations (i.e., updating the respective chain of BFs on the flash by appending all buffered BFs to the existing BF chain in flash) performed in BloomStore.

Similarly, with the INTEL SATA SSD, BloomStore achieves a uniformly higher key lookup throughput than SkimpyStash for all amortized RAM overheads per KV pair. The lookup throughput results for both BloomStore and SkimpyStash are universally higher on the Micro PCIe SSD than on the INTEL SATA SSD. For example, on the *Linux* workload, BloomStore achieves approximately 2.6–2.8 times higher lookup throughput, while SkimpyStash accomplishes 2.25–4 times higher lookup throughput on the Micro PCIe SSD than on the INTEL SATA SSD.

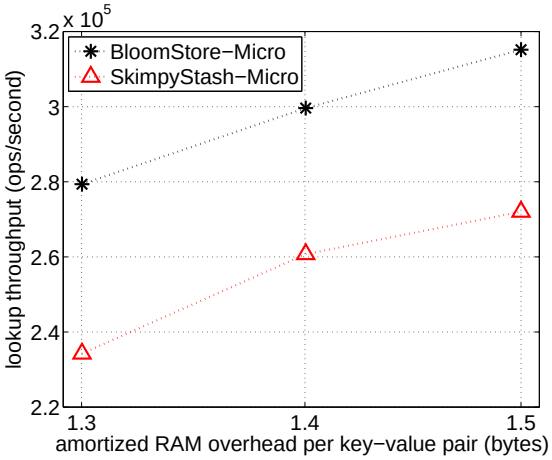

Figure 4.14: *Vx* workload: Key lookup throughput (ops/sec) comparisons between BloomStore and SkimpyStash as the amortized RAM overhead per KV pair varies

4.6 Conclusions Remarks

In this chapter, we designed a flash-based KV store architecture called BloomStore that not only assures an extremely low amortized RAM overhead per KV pair (by keeping a flash-page sized data buffer and a very small sized BF buffer per partition in RAM), but also achieves a high lookup/insertion throughput (by reducing the maximum number of flash page reads with flash partitioning; in each partition by maintaining a dedicated flash page sized KV pair buffer to temporarily buffer the inserted KV pairs; by maximizing the number of buffered BFs in RAM to reduce the BF-containing flash page reads and writes). BloomStore also employs a prefilter to avoid many unnecessary flash page reads for looking up the non-existent keys. Throughput comparisons on the two real-world workloads in data dedup applications illustrated that BloomStore achieved a significantly better key lookup throughput and roughly the same insertion performance with much lower RAM usage – BloomStore provides the same lookup throughput (78,879 ops/second in the *Linux* workload), while consuming 22.5% lower amortized RAM overhead per KV pair, as compared with SkimpyStash.

Chapter 5

Conclusion and Discussion

In this dissertation, we introduced an efficient data dedup system design with a flash-memory based SSD. Our design was explictly optimized for dedup effectiveness and throughput. We also introduced a novel chunking algorithm able to identify more duplicated data by producing longer and more frequent data chunks; a very efficient bloom filter design combing very limited RAM space with much larger flash space to dynamically adapt to the growth of a dataset; a high throughput, low latency KV store design with a flash-memory based SSD and very limited RAM space.

In chapter 2, we specifically proposed a chunking algorithm called the Frequency-Based Chunking (FBC). The FBC algorithm explicitly made use of the chunk frequency information from the data stream to enhance the data dedup gain. This especially occurred when the metadata overhead was taken into consideration. To achieve this goal, the FBC algorithm first utilized a statistical chunk frequency estimation algorithm to identify the frequently appearing chunks. It then employed a two-stage chunking algorithm to divide the data stream, in which the first stage applied the CDC algorithm to obtain a coarse-grained chunking result and the second stage further divided the chunks produced by the CDC algorithm with the identified frequent chunks. We conducted extensive experiments on the heterogeneous datasets to evaluate the effectiveness of the proposed FBC algorithm. In all of the experiments, the FBC algorithm persistently outperformed the CDC algorithm in terms of achieving a better dedup gain or producing a smaller number of chunks. Another benefit of the FBC over the CDC is that the FBC, with an average chunk size greater than or equal to that of the CDC, achieves a

much higher DER than that of the CDC.

As mentioned in the end of section 2.4.2, a given frequency threshold θ determines how aggressively the re-chunking should be carried out. In fact, further thinking shows that if some coarse-grained chunks produced by the CDC algorithm already appear to be highly frequent (i.e., its frequency substantially exceeds the average chunk frequency), it may be desirable to skip re-chunking those chunks, because re-chunking them may result in no gain but more metadata overhead. In the future, we plan to refine our FBC algorithm by setting the escape frequency for the coarse grained chunks produced by the CDC algorithm. In addition, our existing approach requires an extra pass on the dataset to obtain frequency information. We plan to design a one-pass algorithm which conducts the frequency estimation and the chunking process simultaneously. An intuitive idea is to track top k frequent segments instead of tracking all segments with a frequency greater than a certain threshold. This tracking procedure can be integrated into the FBC algorithm to estimate on-line frequency.

In chapter 3, we present the design and evaluation of FBF, a Bloom Filter, that combines a limited RAM space with a much larger flash-memory space to dynamically adapt to the growth of a dataset. Particularly with our design, the querying overhead grows logarithmically as the overall BF size grows. We conduct extensive experiments on two real-world data de-dup workloads of different characteristics. Our experimental results illustrate that the forest-structured BF design achieves two times the processing speed with uniformly 50% of flash write operations of the state-of-the-art in flash BF designs.

In chapter 4, we present a novel KV store architecture on the flash, which efficiently utilizes very limited RAM space combined with much larger flash space to manage the KV pairs and support high throughput, low latency lookup/insertion operations. The design is driven by the need to minimize the RAM space usage for both data buffers and index structures. In order to minimize the RAM overhead of the index structure, BloomStore divides flash space into a number of, say P, logical partitions and indexes each partition separately with bloom filters. We use BFs instead of hash tables as our key index structure because the former further avoid the pointer overhead per bucket, which makes it possible to attain sub-byte level RAM indexing overhead per key. At each partition, one active bloom filter is held in the RAM dedicated to memorizing the

inserted keys in the corresponding data buffer. However, as each partition requires a separate data buffer to temporarily buffer the inserted KV pairs so as to reduce the number of flash writes, the overall RAM usage for data buffers increases by a factor of P times, compared to that in the optimal case, where only one data buffer is held. To overcome this RAM usage increase caused by the partitioning, BloomStore uses a hierarchical partitioning scheme to reduce the number of partitions that require data buffering.

Equipped with both novel BF chain indexing and a hierarchical partitioning design, BloomStore achieves the design goal of extremely low RAM overhead (≤ 1 byte per KV pair stored), which is lower than that of earlier designs. In the meanwhile, compared with the state-of-the-art memory-frugal KV store design SkimpyStash, the BloomStore design achieves a significantly better key lookup operation performance with much lower RAM usage on the two typical dedup workloads that we used to evaluate our KV store design. With less than half of the amortized RAM overhead per KV pair (0.68 bytes), BloomStore can even outperform the SkimpyStash design with 1.44 bytes by 9% in the key lookup throughput on one of the dedup workload. With the same RAM overhead for both designs, BloomStore uniformly outperforms SkimpyStash by 15% - 19% in lookup throughput on the other workload.

Bibliography

[1] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In Randal C. Burns and Kimberly Keeton, editors, *FAST*, pages 239–252. USENIX, 2010. 1, 2.2

[2] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In Gregory R. Ganger and John Wilkes, editors, *FAST*, pages 1–13. USENIX, 2011. 1, 4.5.1

[3] Wei Dong, Fred Douglis, Kai Li, R. Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In Gregory R. Ganger and John Wilkes, editors, *FAST*, pages 15–29. USENIX, 2011. 1

[4] Tony Summers. Hardware compression in storage and network attached storage, 2007. 1, 2.1

[5] 3rd D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1), 2001. 1, 2.3.1, 3.5.2

[6] Benjamin Zhu, Kai Li, and Patterson Hugo. Avoiding the disk bottleneck in the data domain deduplication file system. In *In Proceedings of the 6th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2008. USENIX Association. 1, 2.2, 3.1, 3.5.2, 3.6, 4.1

[7] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In Margo I. Seltzer and Richard Wheeler, editors, *FAST*, pages 111–123. USENIX, 2009. 1, 2.2, 3.5.2

[8] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 16–16. USENIX Association, 2010. 1, 3.2.1, 4.1, 4.2, 4.3, 4.5.2

[9] Deepavali Bhagwat, Kristal T. Pollack, Darrell D. E. Long, Thomas J. E. Schwarz, Ethan L. Miller, and Jehan-François Pâris. Providing high reliability in a minimum redundancy archival storage system. In *MASCOTS*, pages 413–421. IEEE Computer Society, 2006. 1

[10] Chuanyi Liu, Yu Gu, Linchun Sun, Bin Yan, and Dongsheng Wang. R-admad: high reliability provision for large-scale de-duplication archival storage systems. In Michael Gschwind, Alexandru Nicolau, Valentina Salapura, and José E. Moreira, editors, *ICS*, pages 370–379. ACM, 2009. 1

[11] Xiaozhou Li, Mark Lillibridge, and Mustafa Uysal. Reliability analysis of deduplicated and erasure-coded storage. *SIGMETRICS Performance Evaluation Review*, 38(3):4–9, 2010. 1

[12] Youngjin Nam, Guanlin Lu, and David H.C. Du. Reliability-aware deduplication storage: Assuring chunk reliability and chunk loss severity. In *Workshop on Energy Consumption and Reliability of Storage Systems, International Green Computing Conference*. IEEE Computer Society, 2011. 1

[13] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pages 328–338. Springer, 2002. 1

[14] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1996. 2.2

[15] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association. 2.2

[16] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, pages 87–95, 2000. 2.2

[17] ZB Andrei. On the resemblance and containment of documents. In *Compression and Complexity of SEQUENCES*, pages 21–29, 1997. 2.2

[18] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM. 2.2, 3.5.2

[19] Lawrence You, Kristal T. Pollack, and Darrell D. E. Long. Deep store: an archival storage system architecture. In *ICDE*, pages 804–815. IEEE Computer Society, 2005. 2.2

[20] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. Technical Report HPL-2005-30R1, Hewlett Packard Laboratories, October 21 1900. 2.2

[21] Navendu Jain, Mike Dahlin, and Renu Tewari. Taper: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 21–21, Berkeley, CA, USA, 2005. USENIX Association. 2.2, 3.6

[22] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72. USENIX, 2004. 2.2

[23] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference, General Track*, pages 73–86. USENIX, 2004. 2.2

[24] Udi Manber. Finding similar files in a large file system. In *USENIX Winter*, pages 1–10, 1994. 2.2

[25] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX Annual Technical Conference, General Track*, pages 113–126. USENIX, 2003. 2.2

[26] Deepak R. Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage*, 2(4):424–448, November 2006. 2.2

[27] Lawrence You and Christos T. Karamanolis. Evaluation of efficient archival storage techniques. In Ben Kobler and P. C. Hariharan, editors, *MSST*, pages 227–232. IEEE, 2004. 2.2

[28] N. Mandagere, P. Zhou, M.A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17. ACM, 2008. 2.2

[29] G.S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002. 2.2

[30] Jin Cao, Yu Jin, Aiyou Chen, Tian Bu, and Zhili Zhang. Identifying high cardinality internet hosts. *IEEE INFOCOM*, 2009. 2.2, 2.4.2

[31] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003. 2.2, 2.5.4, 3.4.3

[32] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo store: Providing efficient incremental upload and versioning for a utility rendering service. In *FAST*, pages 123–138. USENIX, 2007. 2.2

[33] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS*, pages 1–9. IEEE, 2009. 2.2

[34] M. O. Rabin. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology, Harvard University, 1981. 2.4.2

[35] G. Lu, B. Debnath, and D.H.C. Du. A forest-structured bloom filter with flash memory. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–6. IEEE, 2011. 2.5.4

[36] Mustafa Canim, George A. Mihalia, Bishwaranjan Bhattacharjee, Christian A. Lang, and Kenneth A. Ross. Buffered bloom filters on solid state storage. 2010. 3.1, 3.5.3, 3.6

[37] Biplob Debnath, Sudipta Sengupta, Jin Li, David J. Lilja, and David H.C. Du. Bloomflash: Bloom filter on flash-based storage. In *Proceedings of the 31th International Conference on Distributed Computing Systems*, ICDCS 2011, 2011. 3.1, 3.2.1, 3.6, 4.4.3

[38] Deke Guo, Honghui Chen, and Xueshan Luo. Theory and network applications of dynamic bloom filters. In *In Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2006. 3.1, 3.6

[39] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX*, 2008. 3.2.1, 4.3

[40] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys*, volume 37, 2005. 3.2.1, 4.3

[41] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS*, 2009. 3.2.1, 4.3

[42] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. In *VLDB*, 2010. 3.2.1, 4.3, 4.5.2

[43] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, pages 1094–1104, 2001. 3.4.4

[44] M.J. Luczak and C. McDiarmid. On the power of two choices: balls and bins in continuous time. *The Annals of Applied Probability*, 15(3):1733–1764, 2005. 3.4.4

[45] M.D. Mitzenmacher. *The power of two choices in randomized load balancing*. PhD thesis, Citeseer, 1996. 3.4.4

[46] ioxtreme user guide for linux, version 3 for driver release 1.2.7. page 8, 12 2009. 3.5

[47] Ocz agility sata 2.5 ssd: http://www.ocztechnology.com. 3.5.1

[48] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970. 3.6, 4.2.1, 4.4.3

[49] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom histogram: path selectivity estimation for xml data with updates. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 240–251. VLDB Endowment, 2004. 3.6

[50] Francisco M. Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2003. 3.6

[51] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGOPS Oper. Syst. Rev.*, 34:190–201, November 2000. 3.6

[52] Guanlin Lu, Yu Jin, and David H. C. Du. Frequency based chunking for data deduplication. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, pages 287–296, Washington, DC, USA, 2010. IEEE Computer Society. 3.6

[53] Paulo Sergio Almeida, Carlos Baquero, Nuno Preguica, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6), 2007. 3.6

[54] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010. 4.1

[55] Riverbed steelhead product family: http://www.riverbed.com. 4.1

[56] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST*. USENIX, 2005. 4.2

[57] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embedded Comput. Syst.*, 6(3), 2007. 4.2

[58] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In Tarek F. Abdelzaher, Leonidas J. Guibas, and Matt Welsh, editors, *IPSN*, pages 410–419. ACM, 2007. 4.2

[59] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 1–14. ACM, 2009. 4.2

[60] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *NSDI*, pages 433–448. USENIX Association, 2010. 4.2, 4.2.1, 4.5.2

[61] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: RAM space skimpy key-value store on flash-based storage. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 25–36. ACM, 2011. 4.2, 4.2.1, 4.4.4, 4.5.2

[62] Pagh and Rodler. Cuckoo hashing. *ALGORITHMS: Journal of Algorithms*, 51, 2004. 4.2.1

[63] Shimin Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 73–86. ACM, 2009. 4.2.1

[64] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, June 2005. 4.3

[65] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995. 4.3

[66] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jinsoo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In Taewhan Kim, Pascal Sainrat, Steven S. Lumetta, and Nacho Navarro, editors, *CASES*, pages 160–164. ACM, 2007. 4.3

[67] E. Seppanen, M.T. O'Keefe, and D.J. Lilja. High performance solid state storage under linux. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12. IEEE, 2010. 4.3

[68] Myspace uses fusion powered i/o to drive greener and better data centers. http://www.fusionio.com/case-studies/myspace-case-study.pdf. 4.3

[69] Sailesh Kumar and Patrick Crowley. Segmented hash: an efficient hash table implementation for high performance networking subsystems. In Alan D. Berenbaum, Kai Li, and Jonathan S. Turner, editors, *ANCS*, pages 91–103. ACM, 2005. 4.4.1

[70] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. In *sosp*, pages 1–15, October 1991. 4.4.2

[71] Xbox live 1 vs 100 game: http://www.xbox.com. 4.4.3

[72] G.K. Kanji. *100 statistical tests*. SAGE publications Ltd, 2006. 4.4.3

[73] Murmurhash function: http://en.wikipedia.org/wiki/murmurhash. 4.5.1