

Chapter 2

Existing Deduplication Techniques

Abstract Though various deduplication techniques have been proposed and used, no single best solution has been developed to handle all types of redundancies. Considering performance and overhead, each deduplication technique has been developed with different designs considering the characteristics of data sets, system capacity and deduplication time. For example, if the data sets to be handled have many duplicate files, deduplication can compare files themselves without looking at the file content for faster running time. However, if data sets have similar files rather than identical files, deduplication should look inside the files to check what parts of the contents are the same as previously saved data for better storage space savings. Also, deduplication should consider different designs of system capacity. High-capacity servers can handle considerable overhead for deduplication, but low-capacity clients should have lightweight deduplication designs for fast performance. Studies have been conducted to reduce redundancies at routers (or switches) within a network. This approach requires the fast processing of data packets at the routers, which is of crucial necessity for Internet service providers (ISPs). Meanwhile, if a system removes redundancies directly in a write path within a confined storage space, it is better to eliminate redundant data before storage. On the other hand, if a system has residual (or idle) time or enough space to store data temporarily, deduplication can be performed after the data are placed in temporary storage. In this chapter, we classify existing deduplication techniques based on granularity, place of deduplication and deduplication time. We start by explaining how to efficiently detect redundancy using chunk index caches and bloom filters. Then we describe how each deduplication technique works along with existing approaches and elaborate on commercially and academically existing deduplication solutions. All implementation codes are tested and run on Ubuntu 12.04 precise.

2.1 Deduplication Techniques Classification

Deduplication can be divided based on granularity (the unit of compared data), deduplication place, and deduplication time (Table 2.1). The main components of these three classification criteria are chunking, hashing and indexing. Chunking is a process that generates the unit of compared data, called a chunk. To compare

Table 2.1 Deduplication classification

Methods based on granularity	Place	Time
File-level deduplication	Server-based deduplication	Inline deduplication
Fixed-size block deduplication	Client-based deduplication	Offline deduplication
Variable-sized block deduplication	Redundancy elimination (end-to-end RE, network-wide RE)	

duplicate chunks, hash keys of chunks are computed and compared, and a hash key is saved as an index for future comparison with other chunks.

Deduplication is classified based on granularity. The unit of compared data can be at the file level or subfile level, which are further subdivided into fixed-size blocks, variable-sized chunks, packet payload or byte streams in a packet payload. The smaller the granularity used, the larger number of indexes created, but the more redundant data are detected and removed.

For place of deduplication, deduplication is divided into server-based and client-based deduplication for end-to-end systems. Server-based deduplication traditionally runs on high-capacity servers, whereas client-based deduplication runs on clients that normally have limited capacity. Deduplication can occur on the network side; this is known as *redundancy elimination* (RE). The main goal of RE techniques is to save bandwidth and reduce latency by reducing repeating transfers through the network links. RE is further subdivided into end-to-end RE, where deduplication runs at end points on a network, and network-wide RE (or in-network deduplication), where deduplication runs on network routers.

In terms of deduplication time, deduplication is divided into inline and offline deduplication. With inline deduplication, deduplication is performed before data are stored on disks, whereas offline deduplication involving performing deduplication after data are stored. Thus, inline deduplication does not require extra storage space but incurs latency overhead within a write path. Conversely, offline deduplication does not have latency overhead but requires extra storage space and more disk bandwidth because data saved in temporary storage are loaded for deduplication and deduplicated chunks are saved again to more permanent storage. Inline deduplication mainly focuses on latency-sensitive primary workloads, whereas offline deduplication concentrates on throughput-sensitive secondary workloads. Thus, inline deduplication studies tend to show trade-offs between storage space savings and fast running time.

First we explain chunk index caches and bloom filters that are used to identify redundant data based on indexes and small arrays, respectively. We then go into detail about classified deduplication techniques, discussing each one by one, in the order of granularity, place and time. Note that a deduplication technique can belong to multiple categories, such as a combination of variable-sized block deduplication, server-based deduplication and inline deduplication.

2.2 Common Modules

2.2.1 *Chunk Index Cache*

Deduplication aims to find as many redundancies as possible while maintaining processing time. To reduce processing time, one typical technique is to check indexes of data in memory before accessing disks. If the data indexes are the same, deduplication does not involve accessing the disks where the indexes are stored, which would reduce processing time. An index represents essential metadata that are used to compare data (or chunks). In this section, we show what can be indexed and how indexes are computed, stored and used for comparisons.

2.2.1.1 Fundamentals

To compare redundant data, deduplication involves the computation of data indexes. Thus, an index should be unique for all data with different content. To ensure the uniqueness of an index, one-way hash functions, such as message digest 5 (MD5), secure hash algorithm 1 (SHA-1), or secure hash algorithm 2 (SHA-2) are used. These hash functions should not create the same index for different data. In other words, an index is normally considered a hash key that represents data. Indexes should be saved to permanent storage devices like a hard disk, but to speed up the comparison of indexes, they are prefetched in memory. The indexes in memory should provide temporal locality to reduce the number of evictions of indexes from memory owing to filled memory as well as a decrease in the number of prefetches. In the same sense, to prefetch related indexes, the indexes should be grouped by spatial locality. That is, indexes of similar data are stored close to each other in storage.

An index table is a place where indexes are temporarily located for fast comparison. Such tables can be deployed using many different methods, but mainly they are built using hash tables, which allows comparisons to be made very quickly due to the time complexity of $O(1)$ with the overhead of hash table size. In the next section, we present a simple implementation of an index table using an `unordered_map` container.

2.2.1.2 Implementation: Hash Computation

We show an implementation of an index computation using an SHA-1 hash function. The whole code for this example is in Appendix A. The codes in the appendix are written in C++. The unit of data can be a file or a byte stream data (like chunk). Thus, we show codes to compute a SHA-1 hash key from a file and data. We use the FIPS-180-1-compliant SHA-1 implementation created by Paul Bakker. We developed a wrapper class with two functions, such as `getHashKeyOfFile(string`

filePath) and getHashKey(string data). Following are code snippets that use the two functions.

```
string hashKey;
hashKey = sha1Wrapper.getHashKey(data);
```

```
string hashKey;
hashKey = sha1Wrapper.getHashKeyOfFile(fileName);
```

We provide a main function to test the computation of a hash key and a Makefile to make compilation easy. In the main function, the first paragraph shows how to compute a hash key of a file, and the second paragraph shows how to calculate a hash key of a string block:

```
#ifdef SHA-IWRAPPER_TEST

int main() {

    Sha1Wrapper obj;
    string filePath = "hello.dat";
    string data = "hello_danny_how_are_you_??_";

    string hashKey;

    // get hash key of a file
    hashKey = obj.getHashKeyOfFile(filePath);
    cout << "hashkey_of_" << filePath << "_:" << hashKey << endl;
    cout << endl;

    // get hash key of data
    cout << data << endl;
    hashKey = obj.getHashKey(data);
    cout << "hashkey_of_data_:" << hashKey << endl;

    return 0;
}

#endif

## make
all:
    g++ -DSHA1WRAPPER_TEST -o SHA-1 sha1.cc sha1Wrapper.cc

clean:
    rm -f *.o SHA-1
```

We compile and build an executable file to test SHA-1 as follows:

```
root@server:~/lib/SHA-1# make
g++ -DSHA1WRAPPER_TEST -o SHA-1 sha1.cc sha1Wrapper.cc
root@server:~/lib/sha1# ls -l
-rw-r--r-- 1 root root 12 Jul 20 20:28 hello.dat
-rw-r--r-- 1 root root 98 Jul 20 20:28 Makefile
-rwxr-xr-x 1 root root 37383 Jul 20 20:33 SHA-1
```

```

-rw-r--r-- 1 root root 20297 Jul 20 20:28 sha1.cc
-rw-r--r-- 1 root root 4606 Jul 20 20:28 sha1.h
-rw-r--r-- 1 root root 1187 Jul 20 20:28 sha1Wrapper.cc
-rw-r--r-- 1 root root 522 Jul 20 20:28 sha1Wrapper.h

```

Following are the results of running a SHA-1 executable file. We retrieve an index string with 40 characters created from 20 bytes; 1 byte is denoted by two hexadecimals. Thus, the size of the index amounts to 40 bytes (40 characters). The first hash key that starts with 49a32... is computed from a file (here, hello.dat). The second hash key starting with e69927 is computed from a string “hello danny how are you??”:

```

root@server:~/lib/sha1# SHA-1
hashkey of hello.dat : 49a32112d754917ca799d684895c5bbc4e25828b

hello danny how are you ??
hashkey of data : e69927c529b145fa729ae2664c07929853f59994

```

2.2.1.3 Implementation: Index Table

We show an implementation of an index table using an unordered_map. The implementation codes are in Appendix B. We compile and build a cache executable file. To compile using an unordered_map, we need to add ‘-std=c++0x’ at compilation:

```

root@server:~/lib/cache# make
g++ -DCACHE_TEST -o cache cache.cc -std=c++0x

root@server:~/lib/cache# ls -l
total 72
-rwxr-xr-x 1 root root 54227 Jul 20 21:34 cache
-rw-r--r-- 1 root root 2235 Jul 20 20:28 cache.cc
-rw-r--r-- 1 root root 4079 Jul 20 20:28 cache.h
-rw-r--r-- 1 root root 1278 Jul 20 20:28 cacheInterface.h
-rw-r--r-- 1 root root 91 Jul 20 21:34 Makefile

root@server:~/lib/cache# cat Makefile
## make
all:
    g++ -DCACHE_TEST -o cache cache.cc -std=c++0x

clean:
    rm -f *.o cache

```

What follows shows how to test the implementation codes of an index table. First, an index table is created with a pair consisting of a key and a value. ‘cache.empty()’ is used to check whether the index table is empty. To save an index to the table, we use the set() method, for example, ‘cache.set(<key>, <value>)’. To obtain an index from the table, we use ‘cache.get(<key>)’. ‘cache.size()’ retrieves the number of indexes. To check whether an index with a key exists, the ‘cache.exist(<key>)’ function is used:

```

UMapCache<string , string> cache;

string key = "1";
string value = "Danny";
string key2 = "2";
string value2 = "Kim";
string key3 = "3";

// check if cache is empty
cout << "****_current_cache_****" << endl;
if (cache.empty())
    cout << "empty" << endl;
else
    cout << "filled" << endl;
cout << endl;

// save an entry
cout << "****_save_entries_****" << endl;
cout << "<" << key << "," << value << ">" << endl;
cout << "<" << key2 << "," << value2 << ">" << endl;
cache.set(key, value);
cache.set(key2, value2);
cout << endl;

// check if cache is empty
cout << "****_current_cache_****" << endl;
if (cache.empty())
    cout << "empty" << endl;
else
    cout << "filled" << endl;
cout << endl;

// get an entry
cout << "****_get_an_entry_****" << endl;
cout << "key=_ " << key << "_ ";
cout << cache.get(key) << endl;
cout << endl;

// get number of entries
cout << "****_get_number_of_entries_****" << endl;
cout << "size:_ " << cache.size() << endl;
cout << endl;

// check if an entry with key exists
cout << "****_existence_of_a_key_****" << endl;
string tmp = key2;
if (cache.exist(tmp))
    cout << tmp << "_exists" << endl;
else
    cout << tmp << "_doesn't_exist" << endl;
cout << endl;

```

To show all entries, 'cache.showAll()' is used. We determine the size of the index table using various functions, such as 'sizeOfAllEntries()', 'sizeOfAll-

EntriesDouble()', 'sizeOfKeys()', 'sizeOfKeysDouble()' and 'sizeOfValues()'. That is, 'cache.sizeOfAllEntriesDouble()' shows the size of the index table, including all pairs; 'cache.sizeOfKeys()' and 'cache.sizeOfValues()' return the size of keys or values in the index table respectively; 'cache.sizeOfKeysDouble()' and 'cache.sizeOfValuesDouble()' return the size of double data type; and 'cache.removeAll()' removes all indexes in the index table.

```
// show all entries
cout << "****_show_all_entries_****" << endl;
cache.showAll();
cout << endl;

// show size of all entries in bytes
cout << "size_of_all_entries_(bytes):_" << cache.sizeOfAllEntries() << endl;
cout << "size_of_all_entries(double_value)_(bytes):_" << cache.sizeOfAllEntriesDouble() << endl;

// show size of keys of all entries in bytes
cout << "size_of_keys_(bytes):_" << cache.sizeOfKeys() << endl;
cout << "size_of_keys(double_value)(bytes):_" << cache.sizeOfKeysDouble() << endl;

// show size of values of all entries in bytes
cout << "size_of_values_(bytes):_" << cache.sizeOfValues() << endl;
cout << endl;

// remove all entries
cout << "****_remove_all_entries_****" << endl;
cache.removeAll();
cout << "size_of_all_entries:_" << cache.sizeOfAllEntries() << endl;
cout << endl;
```

The following code shows the results of running a 'cache' executable file. In the following result, for the first time the index table is empty, and then two entries are saved. The keys are '1' and '2', and the values are 'Danny' and 'Kim'. There are two entries. Keys occupy 2 bytes of two characters, and values have 8 bytes of eight characters.

```
root@server:~/lib/cache# cache
**** current cache ****
empty

**** save entries ****
<1,Danny>
<2,Kim>

**** current cache ****
filled
```

```

**** get an entry ****
key = 1 Danny

**** get number of entries ****
size : 2

**** existence of a key ****
2 exists

**** show all entries ****
1, Danny
2, Kim

size of all entries (bytes) : 10
size of all entries(double value) (bytes) : 10
size of keys (bytes) : 2
size of keys(double value)(bytes) : 2
size of values (bytes) : 8

**** remove all entries ****
size of all entries : 0

```

2.2.2 Bloom Filter

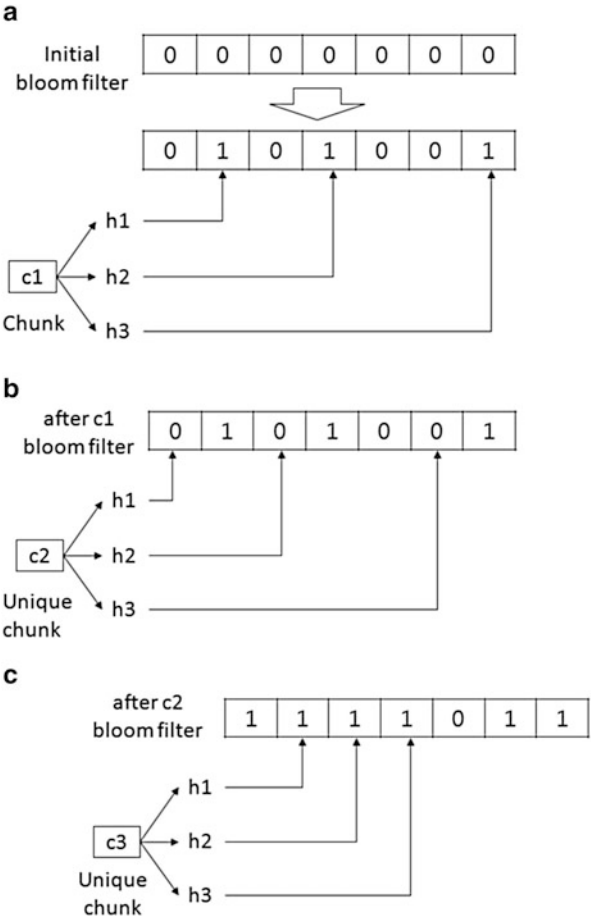
To prevent an index table occupying memory as the number of indexes grows in the index table, a small summary vector, called a Bloom filter, is used to quickly check whether data are unique using small sized metadata. In this section, we see how Bloom filter codes are implemented.

2.2.2.1 Fundamentals

A Bloom filter is used to see whether duplicate chunks of data exist in storage. The Bloom filter is a bit array of m bits initially set to 0. Given a set U , each element u ($u \in U$) of the set is hashed using k hash functions h_1, \dots, h_k . Each hash function $h_i(u)$ returns an array index in the bit array that ranges from 0 to $m-1$. Subsequently, a bit of the index is set to 1. This Bloom filter is used to check whether an element was already saved to a set. When an element attempts to be added to a set, if one of the bits corresponding to the return values of hash functions h_1, \dots, h_k is 0, then the element is considered a new one in the set. If bits corresponding to return values of hash functions are all 1, the element is considered to exist in the set.

Let us explain how the Bloom filter works in an example as shown in Fig. 2.1. The Bloom filter initially has all 0 bits. When a chunk $c1$ is saved, the array indexes of the Bloom filter are computed using three different hash functions ($h1, h2, h3$). Here, $h1, h2$ and $h3$ functions return the second, fourth and seventh indexes respectively.

Fig. 2.1 How the Bloom filter works. **(a)** Bloom filter after *c1* chunk is saved. **(b)** Bloom filter when *c2*, a unique chunk, is compared. **(c)** Bloom filter when *c3*, a unique chunk, is compared (false positive). A unique chunk is found to be redundant



Subsequently, the indexes of the Bloom filter are set to 1. Suppose the same chunk *c1* is saved again. The chunk is found to be redundant because all three indexes by hash functions are already set to 1. As shown in Fig. 2.1b, when a unique chunk (*c2*) is saved, indexes by three hash functions are computed again. Now, the elements of the three indexes are all 0. Thus, a chunk *c2* is determined to be unique. However, in Fig. 2.1c, the Bloom filter can have a false positive, that is, the Bloom filter says that a chunk is redundant, but the chunk is actually unique. The array indexes for *c3* are the second, third and fourth indexes, which were set by other chunks. In this case, we will lose a unique chunk without saving it. Thus, the Bloom filter guarantees that a chunk is unique with one 0 index, but it does not guarantee that a chunk is redundant with all three 1 indexes. Thus, in this case, the chunk index cache should be checked after the Bloom filter is used.

2.2.2.2 Implementation

We show the implementation of a Bloom filter using four hash functions. The size of a Bloom filter bit array is calculated based on the SHA-1 hash key. The first step in implementing a Bloom filter is to determine the size of the Bloom filter bit array. Considering a 2 % false positive, we calculate the size of the Bloom filter bit array as shown in [50]. m is the number of bits in a Bloom filter array, n is the number of bits of a fingerprint (which means a hash key in this case), and k is the number of hash functions. To achieve a 2 % false positive, the smallest size of the Bloom filter is $m = 8 * n$ bits ($m/n = 8$), and the number of hash functions is four. Thus, we compute the size of the Bloom filter bit array (m) to 1280 bits as follows. We choose 1283 rather than 1280 for the size of the Bloom filter bit array because the prime number shows a good uniform distribution, reducing the primary cluster as shown in Weiss' book [47] when the mod() function is used for the hash function:

```
n = 160 bits (SHA-1 hash key)
m = 8 * 160 = 1280 bits
k = 4 (four hash functions)
```

The codes are compiled and built by typing 'make', and bf is the executable file used to test the Bloom filter codes:

```
root@server:~/bf# make
gcc -DBF_TEST -DDEBUG -o bf bf.c

root@server:~/bf# ls -l
-rwxr-xr-x 1 root root 12861 Jul 21 08:59 bf
-rw-r--r-- 1 root root 5627 Jul 21 00:39 bf.c
-rw-r--r-- 1 root root 1585 Jul 21 00:38 bf.h
-rw-r--r-- 1 root root 51 Jul 21 08:59 Makefile
```

Following are the results of the testing program of the Bloom filter codes. First, we assign two fingerprints (which are considered to be hash keys). The Bloom filter is initialized with a bit array with all 0s in each bit. We use 11 bits for the Bloom filter; readers can extend the size if needed. When data with the first hash key (fingerprint1) are saved, deduplication checks the Bloom filter. The bit indexes that four hash functions compute are 2, 4, 7 and 8. Please note that the bit index starts at 0. The data are found to be unique because 0 is found among the bit values. Then the values of the bit indexes (2, 4, 7, 8) are changed to 1, and the Bloom filter has 00101001100 bit arrays:

```
root@server:~/bf# bf
#####
# Test : Input Data          ##
#####
[bloom filter] fingerprint1   : 4543863031426141731
[bloom filter] fingerprint2   : 4543863041425141743

#####
[bloom filter] initialize
#####
```

```

>>> Bloom Filter (11 bits)
0 0 0 0 0 0 0 0 0 0 0

#####
[bloom filter] insert      : 4543863031426141731
#####
hash1() : fringerpt = 4543863031426141731
hash1() : temp      = 454386303142614
hash1() : bf_index  = 7
hash2() : fringerpt = 4543863031426141731
hash2() : temp      = 45438630314261
hash2() : bf_index  = 8
hash3() : fringerpt = 4543863031426141731
hash3() : temp      = 4543863031426
hash3() : bf_index  = 4
hash4() : fringerpt = 4543863031426141731
hash4() : temp      = 454386303142
hash4() : bf_index  = 2

>>> Bloom Filter (11 bits)
0 0 1 0 1 0 0 1 1 0 0

#####
[bloom filter] lookup : 4543863031426141731
#####
hash1() : fringerpt = 4543863031426141731
hash1() : temp      = 454386303142614
hash1() : bf_index  = 7
hash2() : fringerpt = 4543863031426141731
hash2() : temp      = 45438630314261
hash2() : bf_index  = 8
hash3() : fringerpt = 4543863031426141731
hash3() : temp      = 4543863031426
hash3() : bf_index  = 4
hash4() : fringerpt = 4543863031426141731
hash4() : temp      = 454386303142
hash4() : bf_index  = 2
[bloom filter] exist      : 4543863031426141731

#####
[bloom filter] lookup : 4543863041425141743
#####
hash1() : fringerpt = 4543863041425141743
hash1() : temp      = 454386304142514
hash1() : bf_index  = 7
hash2() : fringerpt = 4543863041425141743
hash2() : temp      = 45438630414251
hash2() : bf_index  = 8
hash3() : fringerpt = 4543863041425141743
hash3() : temp      = 4543863041425
hash3() : bf_index  = 4
hash4() : fringerpt = 4543863041425141743
hash4() : temp      = 454386304142
hash4() : bf_index  = 1

```

```
[bloom filter] doesn't exist: 4543863041425141743
root@server:~/bf#
```

When data with the same hash key (fingerprint1) are saved, four hash functions compute the bit indexes, including 2, 4, 7 and 8, which were set to 1 already. Thus, the Bloom filter finds the current data to exist and to be redundant. Now, when the new data with a different hash key (fingerprint2) are saved, four hash functions again calculate the 1, 4, 7 and 8 bit indexes. Though the bit values of the 4, 7, 8 indexes are found to be 1, the bit values of the 1 bit index (second bit) is still 0. This means the current data are unique. Therefore, the Bloom filter says the current data do not exist in the previously saved data. Then the bit value of the 1 bit index (second bit) is changed to 1.

2.3 Deduplication Techniques by Granularity

2.3.1 File-Level Deduplication

File-level deduplication uses file-level granularity, which is the most coarse-grained granularity. File-level deduplication compares entire files based on a hash value of a file, like SHA-1 [34], to avoid saving the same files. In this section, we demonstrate how file-level deduplication works and its implementation.

2.3.1.1 Fundamentals

We begin by explaining how file-level deduplication works. As shown in Fig. 2.2, suppose we have two identical files. When we save the first file, deduplication computes an index that is a hash value using a one-way hash function. If the index is not found in the index table, the file is unique. In this case, the index and the file are saved to the index table and storage respectively. For the second file, the index of the file is found in the index table, so the corresponding file is not saved.

File-level deduplication has been used to remove redundancies of identical files in storage, email systems and cloud-based storage systems. For storage, EMC Corporation's (EMC's) Centera [17] uses file-level deduplication to reduce redundancies in storage. For email systems, Microsoft Exchange 2003 [29] and 2007 [30] use file-level deduplication, called the Single Instance storage (SIS) [5]. An email with multiple recipients is copied to multiple mailboxes, resulting in multiple copies of the email. In this case, SIS saves only one copy of an email in the recipient's mailbox and saves only the pointers of the email in other recipients' mailboxes without storing the email redundantly in the individual recipients' mailboxes. Many cloud-based storage services such as JustCloud [22] and Mozy [32] also use file-level deduplication. One study [28] on corporate users' file systems showed that

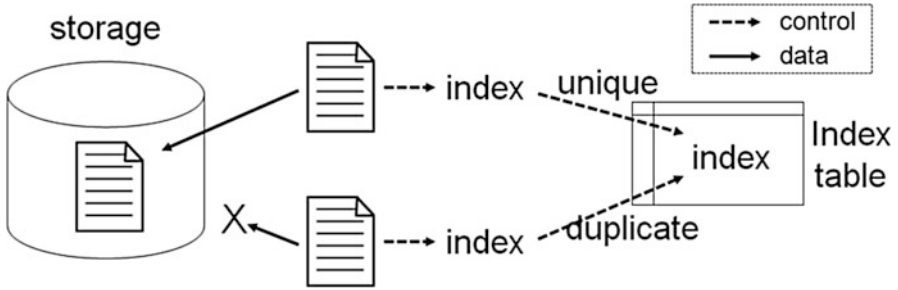


Fig. 2.2 File-level deduplication

simple file-level deduplication can achieve three-quarters of the space savings of aggressive, expensive block deduplications (to be discussed in the next two sections) at a lower cost in terms of performance and complexity.

2.3.1.2 Implementation

The first step of file-level deduplication is to compute an index (hash key) of a file, and the hash key and data of the file are fed into a file-level deduplication function, called the `dedupFile()`. The SHA-1 hash key is used as an index computed by the `getHashKey()` with the data of the file:

```
FileOper fileOper;
ShalWrapper shalWrapper;
string data, hashKey;

data = fileOper.getData(filePath); // path of the file to be
    saved
hashKey = shalWrapper.getHashKey(data);

dedupFile(hashKey, data);
```

`getData()` in `FileOper` class read a file and load the content to a string typed variable. `getData()` is implemented using `ifstream()` as follows:

```
string
FileOper::getData(string filePath) {

    string result, line;

    ifstream file((char*)filePath.c_str());
    if (file.is_open()) {
        while (file.good()) {
            getline(file, line);
            if (!file.eof())
                result += line + "\n";
            else
                result += line;
        }
    }
```

```

    }
} else {
    cout << "getData()_:_" << filePath << "_open_error_" << endl;
    return "";
}
file.close();

return result;
}

```

The dedupFile() function begins by comparing the hash key in arguments with pre-existing indexes in a hash table, checking whether data corresponding to the hash key are unique or duplicates. In code, we use a flag variable, called 'isUnique', that has 'false' initially. File-level deduplication checks the Bloom filter with an index (hash key). If the Bloom filter returns 'true', then we further check the chunk index cache because there may be false positives. If the Bloom filter returns 'false', the current data are determined to be unique with 100 % certainty, and 'isUnique' is changed to 'true':

```

void
SDedup::dedupFile(string fileHashKey , string data) {

    boolean isUnique = false;

    //-----
    // check bloom filter
    //-----
    if (existInBloomFilter(fileHashKey)) {
        // due to false positive , we check chunk index subsequently.

        //-----
        // check chunk index cache
        //-----
        // duplicate data
        if (!isDuplicateInCache(fileHashKey)) {
            isUnique = true;
        }
    } else {
        isUnique = true;
    }

    if (isUnique) {
        saveInCache(fileHashKey);

        // save to storage
        sm.setBufferedData(fileHashKey , data);
    }
}

```

existInBloomFilter() is a wrapper function of bf_lookup(<bloom filter array>, <index>) of the Bloom filter implementation; that is, the Boolean result of bf_lookup() is forwarded to existInBloomFilter(). isDuplicateInCache() is a wrapper function of 'cache.exist(key)' in the index table implementation as follows:

```

bool
SDedup::isDuplicateInCache(string key) {
    if (cache.exist(key))
        return true;
    else
        return false;
}

void
SDedup::saveInCache(string key) {
    cache.set(key, "");
}

```

If the current data are determined to be unique by the Bloom filter or chunk index cache, the index is saved to the chunk index cache using the `saveInCache()` function. `'sm.setBufferedData(fileHashKey, data)'` buffers data contents, compresses the data, and saves them to storage. `'sm'` is an object of the `'StoreManager'` class.

2.3.1.3 Existing Solutions

File-level deduplication is used for Microsoft Exchange 2003 and 2007 based on a SIS [5]. SIS stores file contents to a `'SIS Common Store'`. In SIS, a user file is managed by a SIS link that is a reference to a file called the `'Common Store File'`. Whenever SIS detects duplicate files, SIS links are created automatically and file contents are saved to the common store. SIS consists of a file system filter library that implements links and a user-level service detecting duplicate files that are replaced by links. SIS can find duplicate files but not large redundancies within similar files. We address this issue by developing the Hybrid Email Deduplication System (HEDS) [23].

File-level deduplication is used for popular cloud storage systems, such as JustCloud [22] and Mozy [32], to reduce latency in a client. Cloud storage system client applications run file-level deduplication that computes an index (hash key) of each file and checks whether the index exists in a server. If the server has the index, the client does not send the duplicate file. Running the file-level deduplication in the client before sending data to a server allows cloud storage systems to consume less storage space and bandwidth. One study [20] measured the performance of several cloud storage systems including Mozy.

One study [28] evaluated the trade-off in space savings between file-level deduplication and block-based (fixed-size and variable-size) deduplications, claiming that file-level deduplication provided a simpler complexity and reduced more file fragmentation than block-based deduplications. The study collected file system contents from almost 1000 desktop computers in a corporation and measured file redundancies and space savings. The authors showed that File-level deduplication saves less space than block-level deduplication. Figure 2.3 shows the evaluation setup of the study. A file system scanner computes the indexes of blocks or chunks by running fixed-size and variable-sized block deduplication with the minimum and

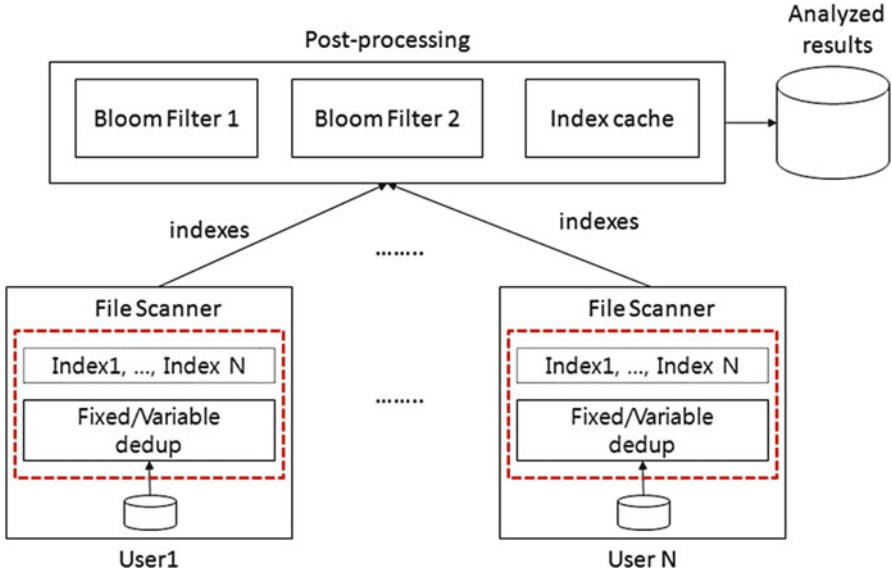


Fig. 2.3 A study of practical deduplication: evaluation set-up

maximum chunk sizes, 4 KB and 128 KB respectively. The expected chunk size ranges from 8 to 64 KB. The computed indexes are collected by a post-processing module that checks the redundancies of indexes using two Bloom filters. The size of each Bloom filter is 2 GB. The analysed results are saved to a database. The computed total size of the files is 40 TB, and the number of files is 200 million. File duplicates are found in post-processing by identifying files where all chunks matched. This study also mentions that a semantic knowledge of file structures would be useful to reduce redundancies with less overhead, and our Structure-Aware File and Email Deduplication for Cloud-based Storage Systems (SAFE) approach exploits the semantic information of file structures, as shown in Chap. 4.

2.3.2 Fixed-Size Block Deduplication

File-level deduplication can find redundancies of identical files but not redundancies within similar files. To find redundancies in similar files, fixed-size block deduplication has been proposed and uses fixed-size blocks for the granularity. In this section, we show how fixed-size block deduplication works and the implementation codes.

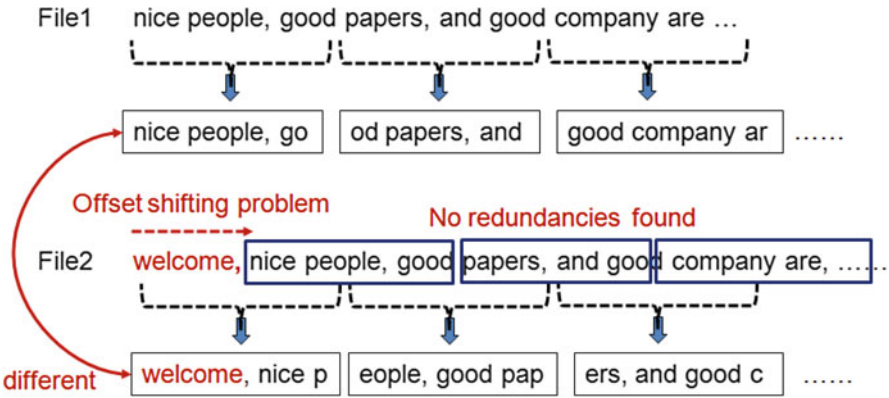


Fig. 2.4 Fixed-size block deduplication

2.3.2.1 Fundamentals

Fixed-size block-level deduplication separates a file into the same sized blocks and finds redundant blocks by comparing the indexes of the blocks. It runs fast because it only relies on offsets in a file to separate a file into blocks. However, fixed-size block deduplication has an issue when it comes to finding matching contents in similar files when the content at the beginning of the files is changed. For example, as shown in Fig. 2.4, suppose deduplication uses a 15 byte fixed-size block as granularity. When we save an original file *File1*, deduplication splits the file into 15 byte fixed-size blocks. Likewise, when we save an updated file *File2*, in which we add the small text ‘welcome’ at the beginning of the original file, deduplication again splits the file into fixed-size blocks. However, blocks split from the updated second file are totally different from blocks split from the original first file. This is because the contents are shifted in the file; this is called the *offset-shifting problem*.

Fixed-size block deduplication has been used for archival storage systems like Venti [39]. Venti uses fixed-size blocks as the granularity level and compares SHA-1 hash keys of blocks with previously saved hash keys following an on-disk index hierarchy. A popular cloud storage system, Dropbox [12], uses very large fixed-size (4 MB) block deduplication. Dropbox reduces network redundant traffic and redundant savings in the server by communicating with indexes between clients and servers before sending the data. Detailed information on how Dropbox works is explained in Chap. 4.

2.3.2.2 Implementation

Fixed-size block deduplication requires three arguments, including an index (a hash key) of a file, the data (content of a file) and a block size based on which data are

split into blocks. To retrieve an index, we can use the `getHashKey()` function in the `sha1Wrapper` class (Sect. 2.2.1.2):

```
hashKey = sha1Wrapper.getHashKey(data);
dedupBlock(hashKey, data, blkSize);
```

The following is an example of a fixed-size block deduplication function, `dedupBlock()`. `dedupBlock()` has three parameters corresponding to the called method, `dedupBlock()`. The first local variable `block` is a reference variable to point to the array of blocks whose type is the string. We also need a variable, ‘`numOfBlocks`’, that shows the number of blocks in a file. The `hashKey` variable indicates the index of each block. In this code, the Bloom filter is not shown, but the checking of redundancy by the Bloom filter can be located before checking using the `isDuplicateInCache()` function with ‘`chunk index cache`’.

```
void
SDedup::dedupBlock(string fileHashKey, string data, int blkSize){

    string *blocks;
    int numOfBlocks = 0;
    string hashKey;
    int i;

    //-----
    // check duplicate file
    //-----
    // *** A duplicate file does not need to be de-duplicated
        to blocks
    if (isDuplicateInCache(fileHashKey)) {
        return;
    }
    else {
        saveInCache(fileHashKey);
    }

    //-----
    // check duplicate blocks
    //-----
    // set block size
    chunkWrapper.setAvgChunkSize(blkSize);

    // get blocks from a data
    blocks = chunkWrapper.getBlocks(data, numOfBlocks);

    for (i=0; i < numOfBlocks; i++) {

        // get hash key of a block
        hashKey = sha1Wrapper.getHashKey(blocks[i]);

        if (!isDuplicateInCache(hashKey)) {
            // save an index of a block
            saveInCache(hashKey);
```

```

        // save to storage
        sm.setBufferedData(hashKey, blocks[i]);
    }

}

// clear memory
delete[] blocks;
}

```

In pure fixed-size block deduplication, a file is directly split into blocks without checking whether the file itself exists, causing redundant processing overhead and memory overhead. Thus, the `dedupBlock()` function first checks whether there is a duplicate file with an index of the current file. If a file is redundant, the file is not separated into blocks. An index of the file is saved to the index table using the `saveInCache()` function, and the `dedupBlock()` function ends.

If the current file is not a duplicate, there could be similar files with the same blocks. First, the program sets the block size using `chunkWrapper.setAvgChunkSize(blkSize)`. The `chunkWrapper` object maintains all environment variables related to chunking. We go into detail on the `chunkWrapper` class in the next section. The `getBlock()` function in `chunkWrapper()` splits the data (file) into blocks of string and returns the split blocks and the number of blocks (`numOfBlocks`). Then, for each block, we check that the block is a duplicate based on the index (or hash key) of the block on the index table. If the block is unique [that is, if `isDuplicateInCache(hashKey)` returns ‘false’], the index of the block is saved to the index table, and each block is filled into buffer so that buffered data are stored when the buffer is full or when it reaches the time threshold. After deduplication is done, the memory allocated for blocks is deleted (the memory was allocated as the type of dynamic array).

```

string *
ChunkWrapper::getBlocks(string data, int & numOfChunks) {

    string *chunks;
    int chunkIndex = 0;
    size_t beginOffset = 0, endOffset = 0;

    int chunkSize = getAvgChunkSize();

    // get number of fixed chunks
    numOfChunks = (data.length() / chunkSize) + 1;

    // get fixed chunks
    chunks = new string[numOfChunks];
    endOffset = beginOffset + (chunkSize - 1);
    while (endOffset < data.length()) {

        chunks[chunkIndex++] = data.substr(beginOffset, chunkSize);

        beginOffset = endOffset + 1;
        endOffset = beginOffset + (chunkSize - 1);
    }
}

```

```

}

// get fixed last chunk
chunks[chunkIndex] =
    data.substr(beginOffset, data.length() - beginOffset);

return chunks;
}

```

The preceding code shows the `getBlocks()` function implementation. Please note that chunk and block terms are used interchangeably. In `getBlocks()`, the number of blocks is computed by dividing the size of the data by the block size (`chunkSize`). We maintain `beginOffset` and `endOffset` for each block, and each block is split from the data and ultimately contained in the string element of the chunk array using a `substr()` function. After all blocks are contained in a chunk string array, the reference variable of the chunks are returned.

2.3.2.3 Existing Solutions

Venti [39] is a fixed-size block deduplication system and uses a write-once policy, preventing data being inconsistent or causing malicious data loss. The main idea is that a file is divided into several blocks, and the index (hash key) of each block is created by a SHA-1 hash function. If the index of the block is the same as a previously saved index, the block is not saved. The index is arranged into a hash tree for reconstructing a file that contains the block. To improve performance, Venti uses three techniques: caching, striping and write buffering. The block and index are cached. Venti shows the possibility of using a hash key to differentiate each block in a file. Most deduplication applications that have been published split a file into several blocks (or chunks) and save each block based on the index (hash key) of each block.

Figure 2.5 shows how files are saved into the tree structure of Venti. A data block is pointed to by an index (hash key) of the block, and the indexes are packed into a pointer block with pointers. As shown in Fig. 2.5a, Venti creates a hash key of a pointer block P_0 that is a root pointer block of *file1*. Venti creates new pointer blocks P_1 and P_2 that subsequently point to D_0, D_1, D_2 and D_3 . Thus, data blocks of *file1* are retrieved following on the tree structure of pointer blocks starting from P_0 . Figure 2.5b demonstrates how the tree structure is changed when a similar file (*file2*) is saved. Suppose *file2* has two identical data blocks (D_0 and D_1), like *file1*, but two unique data blocks (D_4 and D_5). Venti does not change the pointer blocks but instead creates new pointer blocks (P_3 and P_4) for *file2*. *File2* can be retrieved using pointer blocks P_3, P_1 , and P_4 .

Dropbox [12] uses fixed-size block deduplication with a 4 MB fixed block as its granularity. One study [11] discovered internal mechanisms of Dropbox by measuring and analysing packet traces between clients and Dropbox servers. Dropbox is accessed by Web UI (<http://www.dropbox.com>) or a Dropbox client.

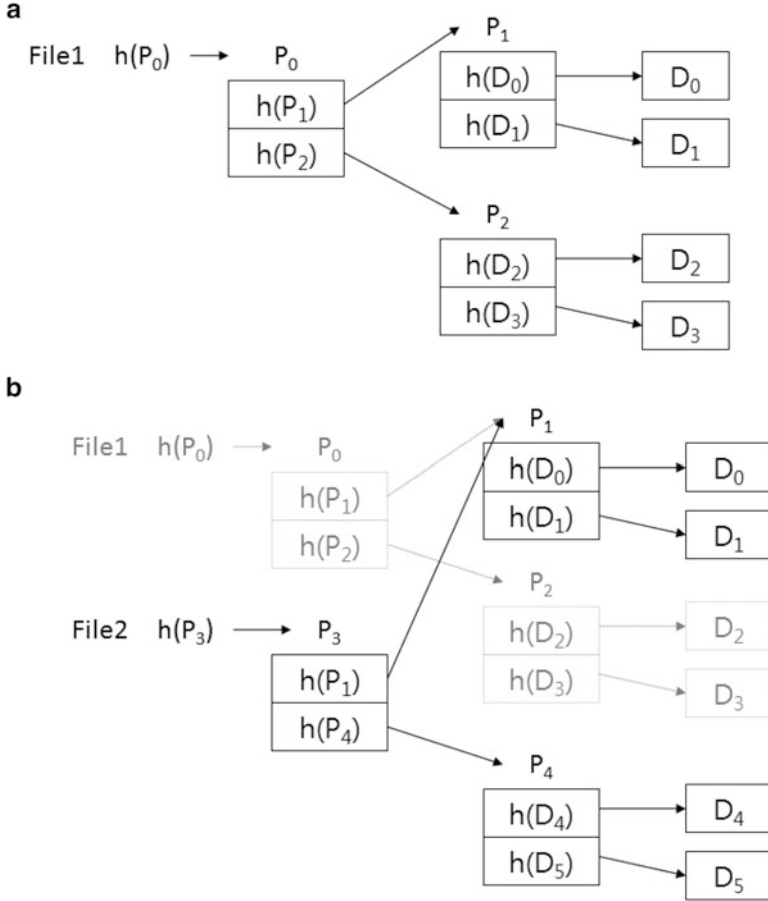


Fig. 2.5 Venti tree structure of data blocks [39]. (a) Tree structure of an original file (File1). *File1* consists of four data blocks, including D_0 , D_1 , D_2 , and D_3 . (b) Tree structure of a similar file (File2). *File2* consists of four data blocks, including D_0 , D_1 , D_4 , and D_5

We leverage SAFE into a Dropbox client to deduplicate structured files on the client side. Dropbox consists of two type of servers, one a control server and the other a storage server. Control servers hold metadata of files such as the hash value of individual blocks and mapping between a file and its blocks. Storage servers contain unique blocks in Amazon S3 [2]. Dropbox client synchronizes its own data and indexes with Dropbox servers.

Figure 2.6 shows how Dropbox works. Circles with numbers represent the order in which a file is saved. *File-A* is a file and *Blk-X* is a block that is separated from a file. $h(Blk-X)$ denotes the hash value of a block. Thick $h(Blk-X)$ and $Blk-X$ are considered hash values and blocks that already existed before a file was saved. The user device is a mobile phone, tablet, laptop or desktop. Dropbox goes through

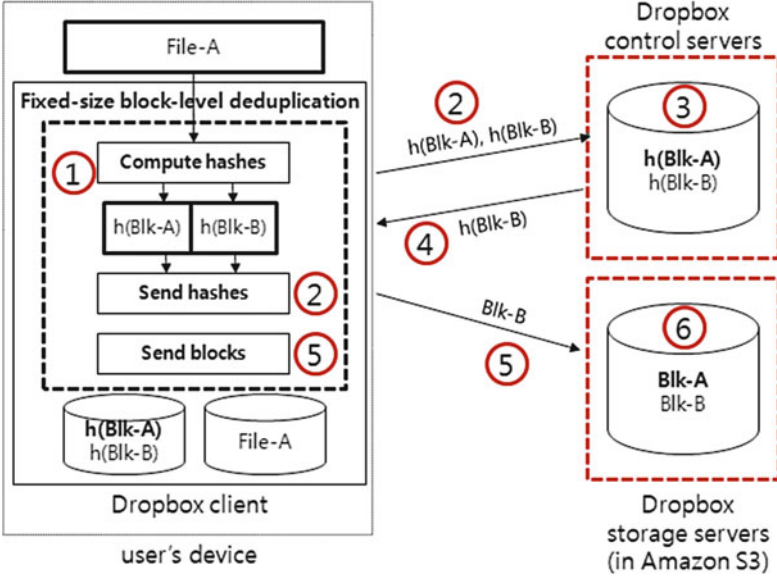


Fig. 2.6 Dropbox internal mechanism

the following steps to save a file. (1) As soon as a user saves *File-A* to a shared folder in a Dropbox client, the fixed-size block deduplication of Dropbox splits the file into blocks based on 4 MB granularity and computes hashes of the objects. If a file is larger than 4 MB, then the file is the same as an object and a hash value of the file is computed. Dropbox uses SHA256 [35] to compute a hash value. (2–4) The Dropbox client sends all computed hash values of a file to a control server that returns only unique hash values after checking previously saved hash values. In this example, the hash key of *Blk-B* is returned to a client because the hash key of *Blk-A* is found to be a duplicate. (5–6) The Dropbox client sends the blocks of returned indexes to the storage server. Ultimately, storage servers have unique blocks across all Dropbox clients. Note that storage saving occurs in a server (thanks to not saving *Blk-A* again), and the incurred network load is reduced because only *Blk-B* is sent.

2.3.3 Variable-Sized Block Deduplication

Variable-sized block deduplication resolves the offset-shifting issue in fixed-size block deduplication, finding more redundant data in similar files. variable-sized block deduplication relies not on a fixed-size offset but on content-based chunking. In this section, we show how variable-sized block deduplication works and present the implementation codes.

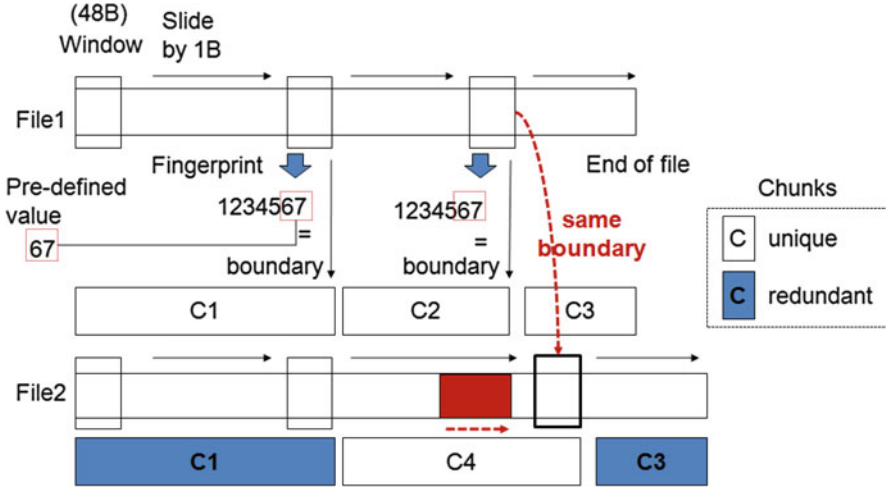


Fig. 2.7 Variable-sized block deduplication

2.3.3.1 Fundamentals

Variable-sized block deduplication has been proposed to solve the *offset-shifting problem* of fixed-size block deduplication. Variable-sized block deduplication relies on contents rather than a fixed offset. Figure 2.7 illustrates how variable-sized block deduplication works. Suppose we have two files. *File1* is an original file and *File2* is an updated file in which we add brief texts in the middle of the file. When we save *File1*, deduplication slides a small window from the beginning of the file. While the window is sliding byte by byte, a fingerprint [40] of each window is computed and the two lowest digits are compared with a pre-defined value. If they are the same, the window is set to a chunk boundary. Then the contents ranging from the previous chunk boundary to the current chunk boundary is treated as a chunk. The window keeps sliding and finding chunk boundaries in the same manner. As a result, three unique chunks (*C1*, *C2*, *C3*) and the corresponding indexes are saved. When we save the updated second file, deduplication again slides a window and finds chunks. *C4* is found to be unique, and *C1* and *C3* are found to be redundant. Here, we see that chunk boundaries are maintained, though the contents are shifted in a file. Thus, content-based variable-sized block deduplication can find more redundancies than offset-based fixed-size block deduplication.

Since variable-sized block deduplication provides fine-granularity chunking techniques to achieve high storage space savings, it has been used for backup [10, 13, 19, 26, 48, 50] or file systems [6, 42]. However, to speed up the processing time by reducing the number of disk accesses, this approach, like the DDFS [50], exploits efficient caching schemes, like the Bloom filter and the chunk index cache, and locality-based disk layout.

2.3.3.2 Implementation: dedupChunk()

We show the dedupChunk() function where variable-sized deduplication is used. Like fixed-size block deduplication, the Bloom filter is not shown, but it can run before an index is checked in the index table using the isDuplicateInCache() function. The dedupChunk() function is almost the same as the dedupBlock() function, except that dedupChunk() uses chunkWrapper.getChunks() rather than chunkWrapper.getBlocks(). The getChunks() function is explained in more detail in Sect. 2.3.3.5. Overall, the unique chunk is passed to a buffer where the buffered data are saved in storage.

```

void
SDedup::dedupChunk(string fileHashKey, string data, int avgChunk
    Size,
                    int minChunkSize, int maxChunkSize) {

    string *chunks;
    int numOfChunks = 0;
    int i;
    string hashKey;
    string filePath;

    //-----
    // check duplicate file
    //-----
    // *** A duplicate file does not need to be de-duplicated to
    blocks
    if (isDuplicateInCache(fileHashKey)) {
        return;
    }
    else {
        saveInCache(fileHashKey);
    }

    //-----
    // check duplicate chunks
    //-----

    // get chunks from a data
    chunks = chunkWrapper.getChunks(data, numOfChunks,
        avgChunkSize, minChunkSize, maxChunkSize);

    for (i=0; i < numOfChunks; i++) {
        // get hash key of a block
        hashKey = sha1Wrapper.getHashKey(chunks[i]);

        if (!isDuplicateInCache(hashKey)) {
            saveInCache(hashKey);
            sm.setBufferedData(hashKey, chunks[i]);
        }
    }
    // clear memory
    delete [] chunks;
}

```


2.3.3.3 Implementation: Rabin Fingerprint

The Rabin fingerprint [40] is used to find chunk boundaries, resulting in the identification of a chunk. The Rabin fingerprint is a 64 bit key. When we compute fingerprints in data (byte stream using sliding windows), a fingerprint of each window can be computed quickly based on the previous fingerprints using the following equation. Detailed information can be found in [43]. The full implementation codes are in Appendix D:

$$\text{RF}(t_{i+1} \dots t_{\beta+i}) = (\text{RF}(t_i \dots t_{\beta+i-1}) - t_i \times p^\beta) + t_{\beta+i} \bmod M. \quad (2.1)$$

We can compile and build a test program to compute the Rabin fingerprint by typing ‘make’. The results from running the executable file, rabin, show a fingerprint (with long integer type) for the ‘hello tom danny’ string.

```
root@server:~/rabin# make
g++ -o rabin rabinpoly.cc rabinpoly_main.cc

root@server:~/rabin# ls -l
total 40
-rw-r--r-- 1 root root 83 Jul 24 16:47 Makefile
-rwxr-xr-x 1 root root 13503 Jul 24 16:47 rabin
-rw-r--r-- 1 root root 9662 Jul 24 16:46 rabinpoly.cc
-rw-r--r-- 1 root root 2756 Jul 24 16:46 rabinpoly.h
-rw-r--r-- 1 root root 1399 Jul 24 16:47 rabinpoly_main.cc

root@server:~/rabin# rabin
rabinpoly_main : input data
hello tom danny
rabinpoly_main : fingerpt = 379718595532164463
```

2.3.3.4 Implementation: Chunking Core

Chunking is the first of three steps in deduplication (the other steps are hashing and indexing). The snippet codes for chunking are found in Appendix E. The core function in chunking is process_chunk in chunk_sub.cc. The process_chunk() function slides a small window byte by byte on the data, finds the chunk boundaries, and saves the beginning and ending indexes for all chunks to the ‘begin_indexes’ and ‘end_indexes’ integer arrays respectively. That is, the goal of the process_chunk() function is to identify the boundaries of the chunks. Based on the boundaries, the chunking Wrapper class, as shown in Appendix F, splits the data into chunks.

The following code is the function call to process_chunk(), where ‘buf’ and ‘size’ are the data and the size of the data to be separated, num_of_breakpoints is the number of break points, BOUNDARY_SIZE, defined in rabinpoly.h, is 48 bytes and the size of the sliding windows avg_chunk_size, min_chunk_size and max_chunk_size are average, minimum and maximum chunk size respectively.

num_of_chunks is literally the number of chunks. begin_indexes and end_indexes are integer arrays, where the beginning and ending indexes of all chunks are to be held after the breakpoints are found.

```
process_chunk(buf, size, &num_of_breakpoints, BOUNDARY_SIZE,
              avg_chunk_size, min_chunk_size, max_chunk_size,
              &num_of_chunks, begin_indexes, end_indexes);
```

The entire code of process_chunk() is in chunk_sub.cc of Appendix E.3. We explain how process_chunk() works using snippet codes. The first *for* loop slides a window by one byte based on cur_pos. At STEP1, b_region means each window, and at STEP2, fingerprint is computed for each b_region using the fingerprint() function in rabinpoly.cc. Readers should note that they can further optimize the running time not by copying window to the b_region char array but by using the offset of window in data. At STEP3, low-order bits are calculated as the remainder of fingerprint divided by avg_chunk_size. At STEP5, the low-order bits of fingerprint are compared with BREAKMARK_VALUE, which is defined at 0x78 in chunk.h. If they are the same, then the current window is determined to be a chunk boundary. The beginning (chunk_b_pos) and ending (chunk_e_pos) indexes of the chunk are saved to begin_indexes and end_indexes integer arrays by the set_breakpoint() function. Note that if the chunk size is less than the predetermined min_chunk_size ('if (cur_chunk_size < min_chunk_size)'), process_chunk() continues to slide a window to find the next chunk boundaries. Also, as at STEP4, if the window keeps sliding without finding chunk boundaries and the size of a chunk is supposed to be larger than max_chunk_size, process_chunk() forcibly sets a chunk boundary (break mark) and creates a chunk.

```
:
for (cur_pos=0; cur_pos < data_size; cur_pos += ONE_BYTE)
{
    :
    //-----
    // STEP1. get boundary region
    //-----
    :
    // allocate boundary region
    b_region = (unsigned char *)malloc(sizeof(unsigned char)*
        b_size);
    memset(b_region, '\0', b_size);

    // get boundary region
    strncpy(b_region, data + cur_pos, b_size);
    b_region[b_size] = '\0';
    :
    //-----
    // STEP2 compute rabin fingerprint
    //-----
    fingerprint = fingerprint(b_region, strlen(b_region), FINGER
        PRINT_PT);

    //-----
```

```

// STEP3. compare to breakpoint value
//          to extract chunk
//
//          fingerprint % K(avg_chunk_size) == BREAKMARK_VALUE
//_____
low_order_bits = fingerprint % avg_chunk_size;

//_____
// STEP4. chunk size is larger than maximum chunk size
//_____
cur_chunk_size = get_chunk_size(chunk_b_pos, cur_pos,
b_size);
if (cur_chunk_size >= max_chunk_size)
{
    //get chunk_b_pos and chunk_e_pos
    set_breakpoint(num_of_breakpoints, &chunk_b_pos,
                  &chunk_e_pos, &cur_pos,
                  (char*)"MAX_CHUNK_SIZE", b_size, data_size,
                  num_of_chunks, begin_indexes, end_indexes);

    // set position for the next chunk
    chunk_b_pos = chunk_e_pos + 1;
    cur_pos      = chunk_b_pos;
}

//_____
// STEP5 chunk size is less than minimum chunk size or
//          is in between minimum chunk size and maximum chunk
//          size
//
//          (f(A) mod K == x)
//          (chunk size is less than minimum or
//          is in range of minimum and maximum chunk size)
//
//          : f(A) → fingerprint
//          : K    → expected average chunk size
//          : x    → BREAKMARK_VALUE
//_____
if (low_order_bits == BREAKMARK_VALUE)
{
    if (cur_chunk_size < min_chunk_size)
    {
        // do not set breakpoint
        ;
    }
    else if ( (cur_chunk_size >= min_chunk_size)
              && (cur_chunk_size < max_chunk_size) )
    {

        // get chunk_b_pos and chunk_e_pos
        set_breakpoint(num_of_breakpoints, &chunk_b_pos,
                      &chunk_e_pos, &cur_pos,
                      (char*)"BREAKMARK", b_size, data_size,
                      num_of_chunks, begin_indexes, end_indexes);
    }
}

```

```

    // set position for the next chunk
    chunk_b_pos = chunk_e_pos + 1;
    cur_pos     = chunk_b_pos;

  }
}
}

```

To compile and build a ‘chunk’ executable, we type ‘make’. Following are the results of running the ‘chunk’ executable. ‘body’ is data that are split into chunks. The chunking core computes 14 boundaries from the body data. The beginning and ending indexes of each chunk are shown in the results. For example, the second chunk has the 3493rd byte as the beginning index and 12,427th byte as the ending index.

```

root@server:~/lib/chunk/chunk_lib# make
g++ -o chunk chunk_main.cc chunk_sub.cc rabinpoly.cc util.cc

root@server:~/lib/chunk/chunk_lib# ls -l
total 356
-rw-r--r-- 1 root root 106342 Jul 20 20:29 body
-rwxr-xr-x 1 root root 24070 Jul 25 13:49 chunk
-rw-r--r-- 1 root root 5417 Jul 20 20:29 chunk.h
-rw-r--r-- 1 root root 4592 Jul 20 20:29 chunk_main.cc
-rw-r--r-- 1 root root 30079 Jul 20 20:29 chunk_sub.cc
-rw-r--r-- 1 root root 409 Jul 20 20:29 common.h
-rw-r--r-- 1 root root 209 Jul 20 20:29 Makefile
-rwxr-xr-x 1 root root 13503 Jul 24 17:36 rabin
-rw-r--r-- 1 root root 9662 Jul 20 20:29 rabinpoly.cc
-rw-r--r-- 1 root root 2756 Jul 20 20:29 rabinpoly.h
-rw-r--r-- 1 root root 1399 Jul 24 17:36 rabinpoly_main.cc
-rwxr-xr-x 1 root root 8845 Jul 24 17:30 util
-rw-r--r-- 1 root root 3088 Jul 20 20:29 util.cc

root@server:~/lib/chunk/chunk_lib# chunk body 8192 2048 65535
### set_breakpoint [1] : size(3493) : 0 ~ 3492, BREAKMARK
### set_breakpoint [2] : size(8935) : 3493 ~ 12427, BREAKMARK
### set_breakpoint [3] : size(23575) : 12428 ~ 36002, BREAKMARK
### set_breakpoint [4] : size(5917) : 36003 ~ 41919, BREAKMARK
### set_breakpoint [5] : size(9126) : 41920 ~ 51045, BREAKMARK
### set_breakpoint [6] : size(3076) : 51046 ~ 54121, BREAKMARK
### set_breakpoint [7] : size(4246) : 54122 ~ 58367, BREAKMARK
### set_breakpoint [8] : size(8408) : 58368 ~ 66775, BREAKMARK
### set_breakpoint [9] : size(18804) : 66776 ~ 85579, BREAKMARK
### set_breakpoint [10] : size(2109) : 85580 ~ 87688, BREAKMARK
### set_breakpoint [11] : size(11416) : 87689 ~ 99104, BREAKMARK
### set_breakpoint [12] : size(2180) : 99105 ~ 101284, BREAKMARK
### set_breakpoint [13] : size(4326) : 101285 ~ 105610, BREAKMARK
### set_breakpoint [13] : size(731) : 105611 ~ 106341, LAST_CHUNK

```

2.3.3.5 Implementation: Chunking Wrapper

The chunkWrapper class defines functions [getChunks()] to separate data into variable-sized chunks based on beginning and ending indexes computed by the chunking core class. The chunkWrapper class also defines functions to obtain fixed-size blocks [getBlocks()]. The chunkWrapper class requires other libraries, including the chunking core class (Sect. 2.3.3.4), Rabin fingerprint class (Sect. 2.3.3.3) and SHA-1 hashing (Sect. 2.2.1.2). The chunkWrapper class also requires a file operation class based on the C++ Boost library. The file operation class is not shown in this book owing to the large code size.

```
root@server:~/lib/chunk# make
g++ -DCHUNK_WRAPPER_TEST -o chunk chunkWrapper.cc
chunkWrapperTest .cc
fileOper/fileOper.cc chunk_lib/chunk_sub.cc chunk_lib/rabinpoly
.cc
chunk_lib/util.cc -I. -Iboost_1_58_0 -L/usr/local/lib
-lboost_filesystem -IfileOper -LfileOper -Ichunk_lib -Lchunk_lib
-Isha1 -Lsha1 sha1/sha1.cc sha1/sha1Wrapper.cc

root@server:~/lib/chunk# ls -l
drwx----- 9 501 staff 4096 Jul 25 15:37 boost_1_58_0
-rw-r--r-- 1 root root 83581760 Apr 16 03:58 boost_1_58_0.tar.gz
-rwxr-xr-x 1 root root 198084 Jul 25 16:01 chunk
-rw-r--r-- 1 root root 1355 Jul 20 20:28 chunkInterface.h
drwxr-xr-x 2 root root 4096 Jul 25 13:49 chunk_lib
-rw-r--r-- 1 root root 4162 Jul 20 20:28 chunkWrapper.cc
-rw-r--r-- 1 root root 915 Jul 20 20:28 chunkWrapper.h
-rw-r--r-- 1 root root 2536 Jul 20 20:28 chunkWrapperTest.cc
-rw-r--r-- 1 root root 18666 Jul 20 20:28 document.xml
.changed
drwxr-xr-x 2 root root 4096 Jul 25 16:01 fileOper
-rw-r--r-- 1 root root 934 Jul 25 16:01 Makefile
-rw-r--r-- 1 root root 1509 Jul 20 20:28 readme
drwxr-xr-x 2 root root 4096 Jul 25 16:01 SHA-1

root@server:~/lib/chunk# ls -l chunk_lib
-rw-r--r-- 1 root root 106342 Jul 20 20:29 body
-rwxr-xr-x 1 root root 24070 Jul 25 13:49 chunk
-rw-r--r-- 1 root root 5417 Jul 20 20:29 chunk.h
-rw-r--r-- 1 root root 4592 Jul 20 20:29 chunk_main.cc
-rw-r--r-- 1 root root 30079 Jul 20 20:29 chunk_sub.cc
-rw-r--r-- 1 root root 409 Jul 20 20:29 common.h
-rw-r--r-- 1 root root 209 Jul 20 20:29 Makefile
-rwxr-xr-x 1 root root 13503 Jul 24 17:36 rabin
-rw-r--r-- 1 root root 9662 Jul 20 20:29 rabinpoly.cc
-rw-r--r-- 1 root root 2756 Jul 20 20:29 rabinpoly.h
-rw-r--r-- 1 root root 1399 Jul 24 17:36 rabinpoly_main.cc
-rwxr-xr-x 1 root root 8845 Jul 24 17:30 util
-rw-r--r-- 1 root root 3088 Jul 20 20:29 util.cc

root@server:~/lib/chunk# ls -l SHA-1
```

```

-rw-r--r-- 1 root root 98 Jul 25 16:01 Makefile
-rwxr-xr-x 1 root root 37383 Jul 25 16:01 SHA-1
-rw-r--r-- 1 root root 20297 Jul 25 16:01 sha1.cc
-rw-r--r-- 1 root root 4606 Jul 25 16:01 sha1.h
-rw-r--r-- 1 root root 1908 Jul 25 16:01 sha1_test.cc
-rw-r--r-- 1 root root 1187 Jul 25 16:01 sha1Wrapper.cc
-rw-r--r-- 1 root root 522 Jul 25 16:01 sha1Wrapper.h

root@server:~/lib/chunk# ls -l fileOper
-rw-r--r-- 1 root root 52812 Jul 25 16:01 fileOper.cc
-rw-r--r-- 1 root root 22577 Jul 25 16:01 fileOper.h

```

We show three results obtained from running the program. The first variable-sized chunking extracts 14 chunks based on 8 KB (8192 bytes) average chunk size. Lines show the chunk boundaries and the chunk sizes after showing the boundaries. The second variable-sized chunking extracts 45 chunks from the same data. This is because an average chunk size (2 KB) that is smaller than the first chunking is used. The smaller the average chunk size used, the more chunks are created. The third result shows blocks extracted by `getBlocks()`, which means fixed-size blocks.

```

root@server:~/lib/chunk# chunk

***** variable sized chunking *****
average chunk size = 8192
minimum chunk size = 2048
maximum chunk size = 65535
### set_breakpoint [1] : size(3493) : 0 ~ 3492, BREAKMARK
### set_breakpoint [2] : size(8935) : 3493 ~ 12427, BREAKMARK
### set_breakpoint [3] : size(23575) : 12428 ~ 36002, BREAKMARK
### set_breakpoint [4] : size(5917) : 36003 ~ 41919, BREAKMARK
:
### set_breakpoint [12] : size(2180) : 99105 ~ 101284, BREAKMARK
### set_breakpoint [13] : size(4326) : 101285 ~ 105610, BREAKMARK
### set_breakpoint [13] : size(731) : 105611 ~ 106341, LAST_CHUNK
chunk [0] 3493
chunk [1] 8935
chunk [2] 23575
chunk [3] 5917
chunk [4] 9126
:
chunk [11] 2180
chunk [12] 4326
chunk [13] 731

***** variable sized chunking (with parameters) *****
average chunk size = 2048
minimum chunk size = 512
maximum chunk size = 65535
### set_breakpoint [1] : size(1329) : 0 ~ 1328, BREAKMARK
### set_breakpoint [2] : size(2164) : 1329 ~ 3492, BREAKMARK
### set_breakpoint [3] : size(3284) : 3493 ~ 6776, BREAKMARK
### set_breakpoint [4] : size(2974) : 6777 ~ 9750, BREAKMARK
:

```

```

### set_breakpoint [41] : size(2180) : 99105 ~ 101284, BREAKMARK
### set_breakpoint [42] : size(609) : 101285 ~ 101893, BREAKMARK
### set_breakpoint [43] : size(1674) : 101894 ~ 103567, BREAKMARK
### set_breakpoint [44] : size(1623) : 103568 ~ 105190, BREAKMARK
### set_breakpoint [44] : size(1151) : 105191 ~ 106341,
  LAST_CHUNK
chunk [0] 1329
chunk [1] 2164
chunk [2] 3284
chunk [3] 2974
:
chunk [40] 2180
chunk [41] 609
chunk [42] 1674
chunk [43] 1623
chunk [44] 1151

***** fixed sized chunking *****
chunk [0] 8192
chunk [1] 8192
chunk [2] 8192
:
chunk [8] 8192
chunk [9] 8192
chunk [10] 8192
chunk [11] 8192
chunk [12] 8038

```

2.3.3.6 Existing Solutions

Variable-sized block deduplication involves expensive chunking and indexing for finding large redundancies, requiring an efficient in-memory cache and on-disk layout on high-capacity servers. DDFS [50] exploits three techniques to relieve a disk bottleneck, reducing processing time. A summary vector, which is a compact in-memory data structure, is used to find new data. Stream-informed segment layout, on-disk layout, is used to improve spatial locality for both data and indexes. The idea of a stream-informed segment layout is that a segment tends to reappear in similar sequences with other segments. This spatial locality is called *segment duplicate locality*. Locality-preserved caching uses segment duplicate locality to acquire a high hit ratio in the memory cache. The study removes 99 % of disk accesses and achieves 100 MB/s and 210 MB/s for single-stream throughput and multi-stream throughput respectively.

Sparse indexing [26] uses sampling and a sparse index to reduce the number of indexes, decreasing RAM requirements. Sparse indexing chooses small portions of chunks in the byte stream as a sample and avoids full chunk indexes, unlike DDFS. This approach employs *chunk locality*, the tendency of chunks in backup data streams to reoccur together. Figure 2.8 shows the deduplication process of sparse indexing. In sparse indexing, a segment is the unit of storage and retrieval and

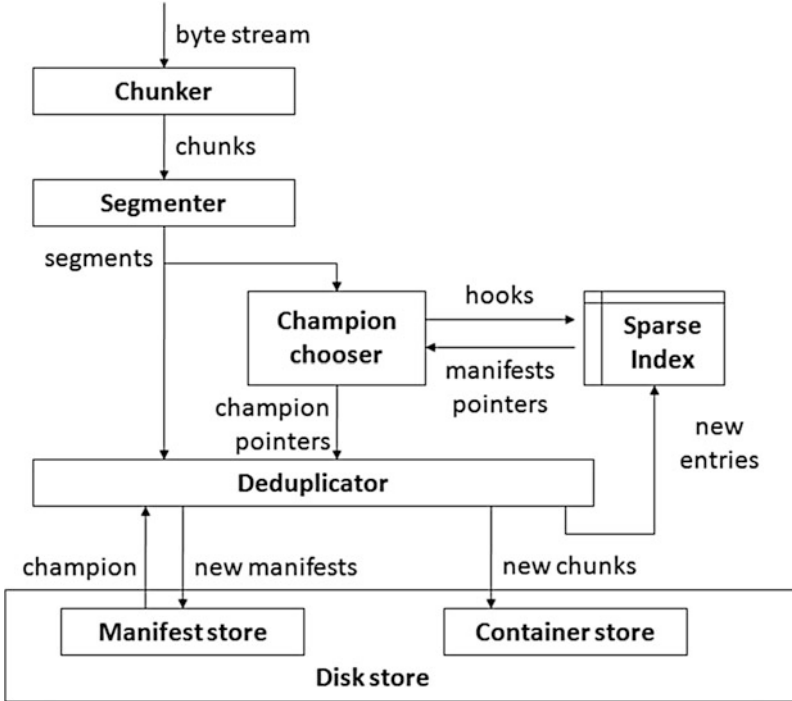


Fig. 2.8 Sparse indexing: deduplication process [26]

a sequence of chunks. A byte stream is split into chunks by Chunker using variable-sized chunking, and a sequence of chunks becomes a segment by Segmenter. Two segments are similar if they share a number of chunks. The Champion chooser chooses sampled segments, called champion, from a sparse index (in-memory index). Deduplicator compares chunks in incoming segments with chunks in champions (selected segments). Unique segments are saved to the sparse index for future comparison, and new chunks are saved to the Container store.

2.3.4 Hybrid Deduplication

Hybrid approaches have been proposed by adaptively using variable-sized block-level deduplication and file-level deduplication, based on either a fixed policy or dynamically changed file information [23, 31]. Min et al. [31] employed context-aware chunking using a file-level deduplication for multimedia content, compressed files or encrypted content and uses variable-sized block-level deduplication for

text files. Our approach, HEDS [23], first separates the message body and individual attachments and performs a variable-sized block-level deduplication if the object size exceeds a predefined threshold. Otherwise, a file-level deduplication is used.

2.3.5 *Object-Level Deduplication*

Fixed-size block deduplication and variable-sized block deduplication can be used for all types of files because they rely on the physical byte-string format of a file. However, for specific file formats, they may be inefficient owing to expensive chunking. Thus, object-level deduplication that splits a file based on the semantic (or logical) format of a file has been proposed. A few *structure-aware data deduplication* techniques [24, 25, 27, 49] have been proposed to simplify the chunking mechanism by using objects. Our approach, SAFE [24], splits structure files, including compressed files, document files (docx, pptx and pdf) and emails, based on files' structured formats. ADMAD [27] separates a file into variable-sized semantic segments, called meaningful chunks (MCs), based on the metadata of each file. Although the idea of ADMAD decomposing a file into objects according to the object structure is similar to that of SAFE, ADMAD is limited to a specific file format. For example, ADMAD does not deal with document file types such as docx, pptx and pdf. In addition, ADMAD does not handle emails with multiple attachments. Similar concepts involving the deduplication of structured objects are presented in [25] and [49].

2.3.6 *Comparison of Deduplications by Granularity*

Overall, as shown in Fig. 2.9, the deduplication ratio indicates how many redundancies are removed, and variable-sized block deduplication is much better than others. In terms of processing time, variable-sized block deduplication is the worst owing to expensive chunking. In terms of index overhead, fixed-size and variable-sized block deduplication is much worse than file-level deduplication, and the index overhead of fixed-size and variable-sized block deduplication changes depending on the block or chunk size. Thus, variable-sized block deduplication is good for the deduplication of updated files or server-based deduplication because high-capacity servers can handle excessive processing time and index overhead. On the other hand, file-level deduplication is good for the deduplication of copied files or client-based deduplication given low-capacity clients.




	Good for client-based Or copied files		Good for server-based Or updated files	
 Deduplication ratio	File-level	< Fixed size << better	Variable size	
 Processing time	File-level	< Fixed size <<<< worse	Variable size	
 Index overhead	File-level	<< Fixed size worse	Variable size	

Fig. 2.9 Comparisons of deduplications

2.4 Deduplication Techniques by Place

2.4.1 Server-Based Deduplication

Server-based deduplication has emerged as a disk-based substitute for tape storage and backs up large amounts of data at fast speeds using high-performance and dedicated backup systems. There are many commercial products [18, 36, 45] that can be used for this type of deduplication.

In this approach, clients send backup data to servers where data are deduplicated. Clients have lightweight backup through which data are sent to servers, avoiding large CPU computation and memory overhead of sources for backup purposes. Figure 2.10 shows how server-based deduplication works. A file is transferred to a server through a client application. On the server, the file is separated into chunks typically using variable-sized block deduplication. Indexes of chunks are computed and compared with indexes previously saved using a Bloom filter or a chunk index cache. Suppose a chunk *c1* is redundant and a chunk *c2* is unique in this example. Then, chunk *c2* and its corresponding index are saved to storage.

Server-based deduplication finds significant redundancies but incurs excessive redundant data traffic because duplicate data are delivered to servers to be deduplicated. What is worse, servers have large CPU computation and memory overhead for chunking and indexing of all backup data. To handle backup quickly with this overhead within a limited backup window, efficient in-memory and on-disk layout is required, such as in DDFS [50].

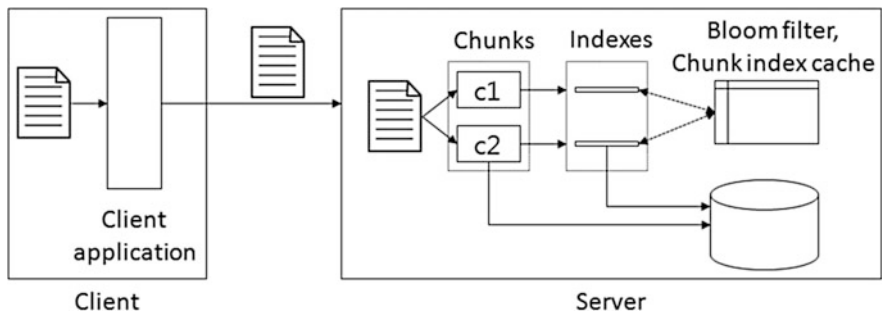


Fig. 2.10 Server-based deduplication

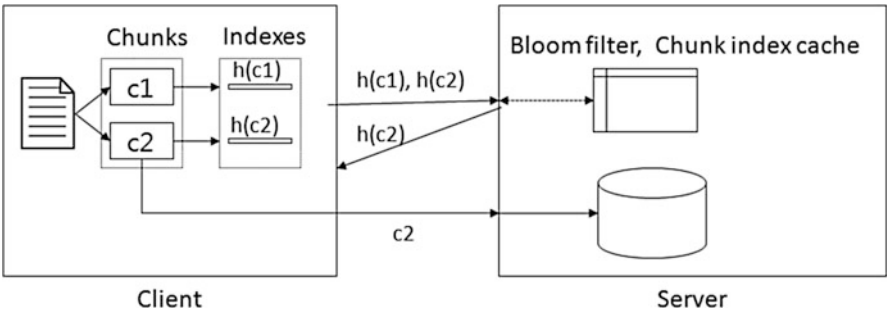


Fig. 2.11 Client-based deduplication: $c1$ and $c2$ are chunks. $h(c1)$ and $h(c2)$ are hash keys (indexes) of chunks

2.4.2 Client-Based Deduplication

In *client-based deduplication*, clients can keep indexes of deduplicated data or have a backup agent to check indexes that exist on servers. In either case, clients check the uniqueness of data in local indexes or in remote indexes through a backup agent. Only unique data are then delivered to servers. Client-based deduplication [16, 46] removes excessive redundant network traffic by performing deduplication at the client before data are transmitted. However, clients incur CPU computation and memory overhead for backup.

Pure client-based deduplication, where a client removes redundant data before sending data to a server, does not collaborate with a server (or servers); redundant data among clients are transferred to a server, which increases data traffic on the network. Thus, client-based deduplication typically communicates with a server, and Fig. 2.11 illustrates how client-based deduplication works with the help of a server. The client splits a file into chunks ($c1$ and $c2$) and computes indexes ($h(c1)$, $h(c2)$). Then the client sends all the indexes of the file to a server, which then returns indexes ($h(c2)$) of unique chunks that have not yet been saved. In this way, the client then can send only unique chunks ($c2$).

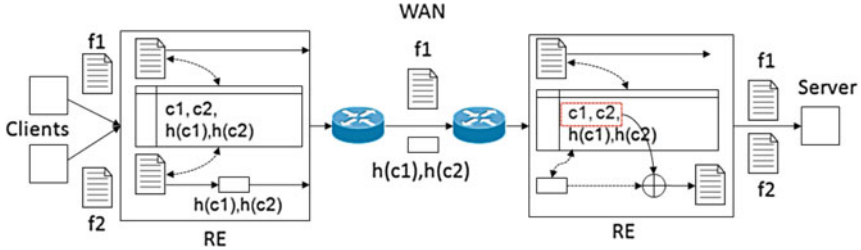


Fig. 2.12 End-to-end RE: $c1$ and $c2$ are chunks; $h(c1)$ and $h(c2)$ are hash keys (indexes) of chunks

A low-bandwidth network file system (LBFS) [33] improves space savings by adding a communication protocol that sends indexes to a server before sending a real data chunk. However, it introduces latency to run the protocol. Overall, a client-based deduplication system has difficulties associated with the limited capacity of clients to carry out an expensive deduplication process.

2.4.3 End-to-End Redundancy Elimination

End-to-end RE, like WAN optimizers [7, 8, 41], removes redundant network traffic at two end points (e.g. branch to headquarter and data centre to data centre). Figure 2.12 illustrates how end-to-end RE works. An end-to-end RE device, like a WAN optimizer, is located just before an ingress router (sending side) and just after an egress router (receiving side). Suppose clients send the same files ($f1$ and $f2$) to a server. When a unique file $f1$ is transferred, the file is split into chunks (here $c1$ and $c2$) and corresponding indexes ($h(c1)$ and $h(c2)$) are saved to the cache; subsequently, chunks and indexes are saved to disk (shown here). The file is compressed and delivered to the server side, where chunks and indexes of the received file are saved to the cache.

Now, when another client sends the same file ($f2$), chunks of $f2$ are split and indexes of the chunks are compared with previously saved indexes. The file $f2$ is found to be a duplicate because the same indexes $h(c1)$ and $h(c2)$ are found in the cache. Thus, the contents of the file are replaced (or encoded) by small indexes $h(c1)$ and $h(c2)$, which reduces packet size. When an encoded packet arrives at the server side, a file $f2$ is reassembled with chunks $c1$ and $c2$ based on indexes in the packet. The reassembled file is directed to a specific server.

A LBFS [33] reduces latency and network bandwidth through the collaboration of the client and server. That is, LBFS avoids sending data over the network when the same data can already be found in the server's file system or the client's cache. To reduce the bandwidth requirement, LBFS exploits cross-file similarities. As shown in Fig. 2.13a, a LBFS consists of a LBFS client and server, and both sides maintain chunk indexes in a chunk database.

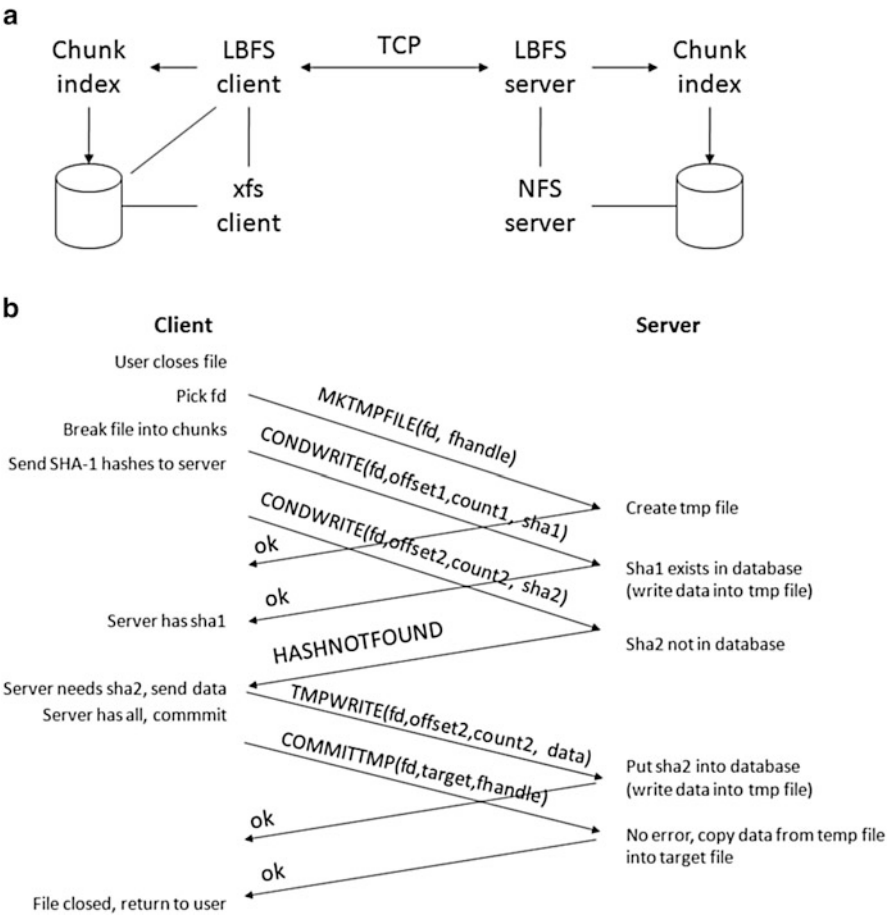


Fig. 2.13 Low-bandwidth file system (LBFS) [33]. (a) LBFS implementation. (b) LBFS: write a file

Figure 2.13b shows how a LBFS works when a file is written to a server from a client. When a user closes a file, a client chooses a file descriptor and calls MKTMPFILE RPC; subsequently, a server creates a temporary file. A client splits a file into chunks (chunk1 and chunk2), computes the hash keys of the chunks and calls CONDWRITE RPCs with hash keys. Suppose the server has SHA-1 (hash key for chunk1) but not SHA-2 (hash key for chunk2). The server returns HASHNOTFOUND for the SHA-2 request; that is, the server does not have chunk2. The client sends only chunk2 to the server, and the server creates a file with chunk1 (previously saved chunk) and chunk2 (chunk received by TMPWRITE RPC). A LBFS can be considered client-based deduplication because the client splits the file

into chunks and saves the indexes. Also, LBFS can be considered end-to-end RE because the client and the server hold the same chunks and indexes, and only unique chunks are transferred through the network, with both sides (client and server) maintaining chunks for unique and redundant files.

2.4.4 *Network-Wide Redundancy Elimination*

Network-wide RE runs deduplication at routers on the network rather than running it at a host (either a client or a server). The unit of deduplication becomes a payload of the packet or byte string on the payload, which is generally smaller than a file (or block or chunk) used for storage data deduplication. This section shows how network-wide RE works and implementation code. For implementation, the first step in the deduplication of a payload of a packet or a byte stream of the payload is to capture a packet on the fly in a router (or at network end points in a router). We show how to intercept a packet using a user space library called `libnetfilter_queue`. We also show how to intercept a packet using a kernel-level module, `libnetfilter`, to achieve better performance.

2.4.4.1 Fundamentals

Network-wide RE [3, 4, 43] eliminates repeating network traffic across network elements such as routers and switches. Network-wide RE computes indexes [40] for the incoming packet payload and eliminates redundant packets by comparing indexes with the packets saved previously. Redundant payload is encoded by small shims and decoded before exiting the network. However, this approach suffers from high processing time owing to sliding fingerprinting on routers and high memory overhead for saving packets and indexes.

The goal of network-wide RE is to remove redundancies of packet payloads, and the granularity is byte strings in a payload. Figure 2.14 shows how network-wide RE works. In network-wide RE, there are special routers (or switches) called RE devices. When an RE device receives a packet, it slides a small window on the payload and computes the fingerprints of all windows. Then some fingerprints are compared with fingerprints in the local cache. If they are the same, the indicated byte regions are expanded to the left and to the right while being compared with a packet in the local cache. The expanded byte region is replaced by a small shim header with a fingerprint and byte offsets. These processes are encoding processes. The encoded payload is reconstructed by a RE device on a path, called decoding. Decoded packets are delivered to a server.

As we see here, network-work RE saves bandwidth in links between an encoder and a decoder. However, as shown in Fig. 2.15, sliding fingerprinting requires excessive processing time, and packets that are saved in cache increase memory requirements. More importantly, redundancies removed in the network are restored

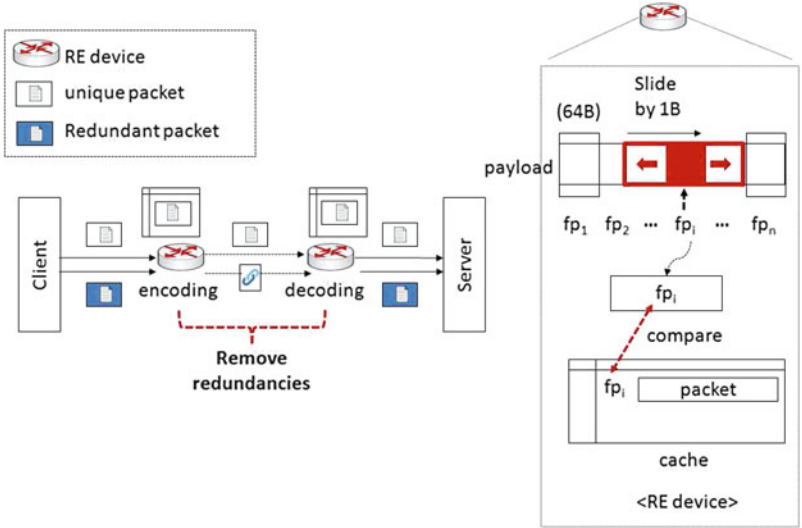


Fig. 2.14 Network-wide redundancy elimination

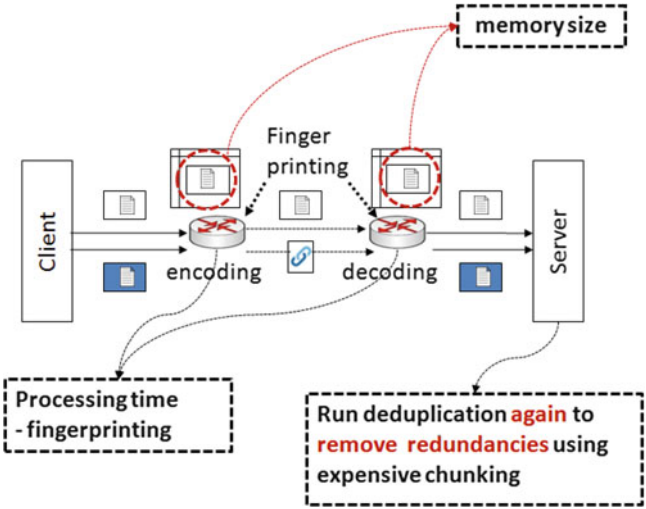


Fig. 2.15 Network-wide redundancy elimination: issue

in a decoder before reaching the server. Thus, the server should run deduplication again to remove redundancies using expensive chunking. That is, there are redundant deduplication operations in the network as well as on the server. We address this issue by developing SoftDance in Chap. 5.

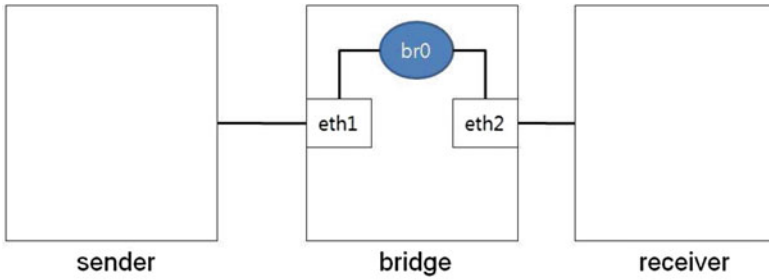


Fig. 2.16 Example deployment – Linux Bridge

2.4.4.2 Implementation: Linux Bridge

To deduplicate payloads in packets, a packet should be captured on a router, and the redundant payload (or byte strings in the payload) is (are) replaced by small indexes. The current router itself does not support these processes, so we need to use different types of router, like software-defined switches or middlebox based on a generic server. We show that a middlebox acts as a router and performs deduplication. In this example of implementation, we use a Linux system as a middlebox. To capture a packet, we set up a Linux bridge in a Linux system, capture an incoming packet and check whether the payload is redundant based on the index of the payload of packets that have been passed. This section shows how to set up a Linux bridge.

Figure 2.16 shows a basic Linux bridge that forwards a packet in Layer 2. A bridge is not a router. A sender and a receiver are in the same network. We show how to deploy a Linux bridge. The deployed environment is one in which there are three Linux computers. One has two network interface cards (NICs) for a bridge and one NIC for Internet access on which ‘brctl’ is installed. The two other computers have just one NIC. This example deployment is based on Ubuntu 12.04 LTS. The IP address of the bridge is 192.168.2.4, and those of the other two computers (a sender and a receiver) are 192.168.2.11 and 192.168.2.5 respectively. In this example deployment, we set an IP address on a bridge. However, if we do not have to access the bridge, it is okay not to set the IP address on the bridge. The first step is to install ‘brctl’ on a computer to be used as a Linux bridge as follows.

```

root@bridge:~# apt-get update
:
root@bridge:~# apt-get install bridge-utils
:
root@bridge:~# brctl show
bridge name      bridge id        STP enabled      interfaces

```

The next step is to create and configure a bridge, called br0, on the Linux computer, and connect two ports to the new bridge. That is, two ports communicate through the bridge. One port acts as an incoming or outgoing port interchangeably. After tying two ports to a bridge (br0), the bridge is set up with an IP address.

We enable promiscuous mode for two ports (eth1 and eth2). By typing ‘brctl show’, we find that a bridge (br0) is bound to two ports such as eth1 and eth2.

```
root@bridge:~# brctl addbr br0
root@bridge:~# brctl show
bridge name      bridge id          STP enabled      interfaces
br0              8000.000000000000  no
// ← we create a bridge (br0)

root@bridge:~# brctl addif br0 eth1
root@bridge:~# brctl addif br0 eth2
root@bridge:~# ifconfig br0 192.168.2.4 netmask 255.255.255.0 up
root@bridge:~# ifconfig eth1 0 promisc up
root@bridge:~# ifconfig eth2 0 promisc up

root@bridge:~# brctl show
bridge name      bridge id          STP enabled      interfaces
br0              8000.001018076b3d  no               eth1
                                                         eth2
```

The next step is to set a Linux parameter to forward traffic as follows. We change ip_forward under the ‘/proc/sys/net/ipv4/’ directory from 0 to 1.

```
root@bridge:~# cat /proc/sys/net/ipv4/ip_forward
0
root@bridge:~# echo 1 > /proc/sys/net/ipv4/ip_forward
root@bridge:~# cat /proc/sys/net/ipv4/ip_forward
1
```

What follows is the result showing that all connections work properly. First, it shows a connection from a bridge to both end systems (a sender and a receiver). Second, we check whether a bridge is working properly by pinging from a sender to a receiver, and vice versa.

```
// Connection test to both ends
- bridge → receiver
root@bridge:~# ping 192.168.2.5
PING 192.168.2.5 (192.168.2.5) 56(84) bytes of data:
64 bytes from 192.168.2.5: icmp_req=1 ttl=64 time=0.678 ms
64 bytes from 192.168.2.5: icmp_req=2 ttl=64 time=0.369 ms
— 192.168.2.5 ping statistics —
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.369/0.523/0.678/0.156 ms

- bridge → sender
root@bridge:~# ping 192.168.2.11
PING 192.168.2.11 (192.168.2.11) 56(84) bytes of data:
64 bytes from 192.168.2.11: icmp_req=1 ttl=64 time=254 ms
64 bytes from 192.168.2.11: icmp_req=2 ttl=64 time=1.46 ms
— 192.168.2.11 ping statistics —
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 1.467/127.972/254.478/126.506 ms

// check if a bridge works
```

```

- sender -> receiver
[root@sender ~]# ping 192.168.2.5
PING 192.168.2.5 (192.168.2.5) 56(84) bytes of data.
64 bytes from 192.168.2.5: icmp_seq=1 ttl=64 time=83.9 ms
64 bytes from 192.168.2.5: icmp_seq=2 ttl=64 time=2.17 ms
64 bytes from 192.168.2.5: icmp_seq=3 ttl=64 time=1.57 ms
:

- receiver -> sender
root@receiver:~# ping 192.168.2.11
PING 192.168.2.11 (192.168.2.11) 56(84) bytes of data.
64 bytes from 192.168.2.11: icmp_req=1 ttl=64 time=126 ms
64 bytes from 192.168.2.11: icmp_req=2 ttl=64 time=2.32 ms
64 bytes from 192.168.2.11: icmp_req=3 ttl=64 time=1.47 ms

```

2.4.4.3 Implementation: Packet Flow in Netfilter

After we deploy a Linux bridge, the next question is how and where to capture a packet. For this purpose, we need to understand packet flow in Netfilter within the Linux network stack. As shown in Fig. 2.17, an incoming packet flows through Netfilter modules in the Linux operating system. Each rectangle means a process for a packet. For example, a rectangle with ‘filter’ (upper part of the rectangle) and ‘forward’ (lower part of the rectangle) means that a packet is processed in a forward chain based on the iptable rule (‘filter’). Thus, to capture (intercept) a forwarding packet, we add an iptable rule of the forward chain as follows:

```

iptables -I FORWARD -p tcp -j NFQUEUE --queue-num 0

```

-I selects a chain among INPUT, FORWARD, and OUTPUT chains. -p indicates a protocol by which a packet is delivered. -j means a target at which a packet is intercepted; in this case, a packet is forwarded to NFQUEUE where packets are buffered before being forwarded to a packet processing application. --queue-num means the number of the NFQUEUE. The valid queue numbers are 0 to 65536, and NFQUEUE 0 indicates the first NFQUEUE.

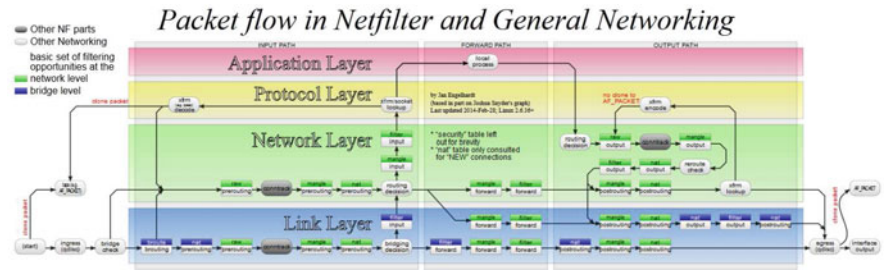


Fig. 2.17 Packet flow in Netfilter [37]

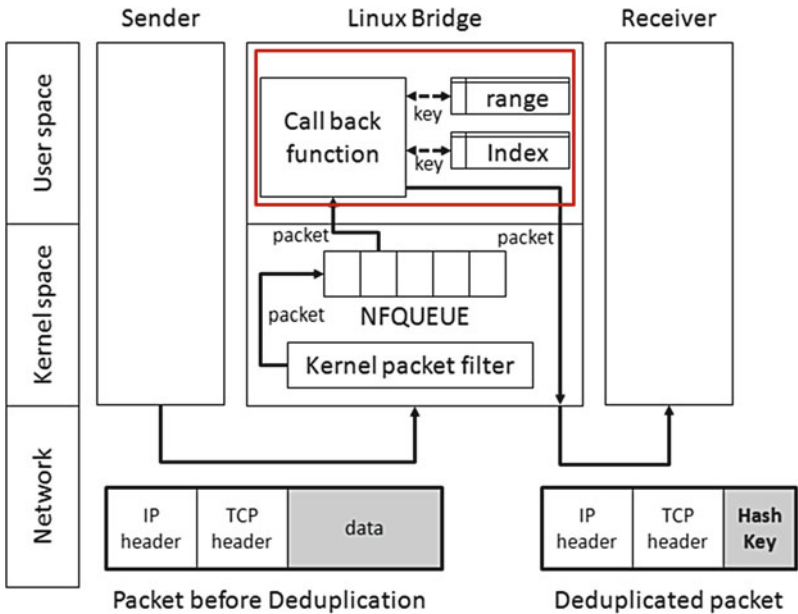


Fig. 2.18 NFQUEUE in Linux bridge

Figure 2.18 illustrates a Linux bridge intercepting an incoming packet. An incoming packet is intercepted from Netfilter (forward chain) and forwarded to NFQUEUE. The packet in NFQUEUE is sent to a call-back function (explained in the next section, libnetfilter_queue) which changes the duplicate payload of a packet to a small index (or changes duplicate byte strings in the payload to small indexes). Then, a deduplicated packet is sent out through an outgoing interface to a receiver.

2.4.4.4 Implementation: Packet Capture: libnetfilter_queue

libnetfilter_queue is a user space library providing an API to packets that have been queued to a kernel packet filter. We can implement a network deduplication program using the libnetfilter_queue library. In this section, we show how to install the libnetfilter_queue library and then how to run a simple network deduplication program based on the library. This installation is done with Ubuntu 12.04.

First, we need to install prerequisite packages such as nfnetlink and libmnl for libnetfilter_queue. To install an nfnetlink, ‘apt-get install’ is used with the package name, libnfnetlink-dev. For the libmnl package, libmnl higher than 1.0.3 is needed for libnetfilter_queue. (In default, Ubuntu 12.04 has a lower version of libmnl than 1.0.3.) Thus, we download libmnl_1.0.3 source codes and install by typing ‘./configure’, ‘make’ and ‘make install’ in that order. To extract the bzip2 file, ‘tar -xjvf <file name>’ is used.

```

root@server:~# apt-get install libnfnctlink-dev
:
Unpacking libnfnctlink-dev
Setting up libnfnctlink-dev (1.0.0-1) ...
root@server:~#

root@server:~# wget https://launchpad.net/ubuntu/+archive/
primary/+files/libmnl_1.0.3.orig.tar.bz2
:
Length: 337375 (329K) [application/octet-stream]
Saving to: 'libmnl_1.0.3.orig.tar.bz2'

100%[=====>]

'libmnl_1.0.3.orig.tar.bz2' saved [337375/337375]

root@server:~# ls
:
libmnl_1.0.3.orig.tar.bz2
:

root@server:~# tar -xjvf libmnl_1.0.3.orig.tar.bz2
libmnl-1.0.3/
:
libmnl-1.0.3/examples/rtnl/rtnl-route-dump.c

root@server:~# ls
:
libmnl-1.0.3

root@server:~# cd libmnl-1.0.3
root@server:~/libmnl-1.0.3# ./configure
root@server:~/libmnl-1.0.3# make
root@server:~/libmnl-1.0.3# make install

```

We install `libnetfilter_queue` library. We download the source code of the library using the `'wget'` command. The latest version is 1.0.2. To extract the downloaded compressed file with `bzip2`, `'tar -xjvf <file name>'` is used. We type `'./configure'`, `'make'` and `'make install'` to compile and build `libnetfilter_queue` library. The final step is to set `'LD_LIBRARY_PATH'` to the `'/usr/local/lib'` directory where shared library files are located.

```

root@server:~# wget http://www.netfilter.org/projects/
libnetfilter_queue/files/libnetfilter_queue-1.0.2.tar.bz2

root@server:~# ls
libnetfilter_queue-1.0.2.tar.bz2 ...

root@server:~# tar -xjvf libnetfilter_queue*.bz2
libnetfilter_queue-1.0.2/
:
root@server:~# ls
libnetfilter_queue-1.0.2 ....

```

```

root@server:~# cd libnetfilter_queue-1.0.2
root@server:~/libnetfilter_queue-1.0.2# ./configure
root@server:~/libnetfilter_queue-1.0.2# make
root@server:~/libnetfilter_queue-1.0.2# make install

// set up LD_LIBRARY_PATH
root@server:~/nfqueue# export LD_LIBRARY_PATH=/usr/local/lib

root@server:~/nfqueue# env | grep LD_LIBRARY_PATH
LD_LIBRARY_PATH=/usr/local/lib

```

2.4.4.5 Implementation: Network dedup Sample Program Using libnetfilter_queue

Network deduplication sample programs using the libnetfilter_queue library are in Appendix G. The sample program works as a call back function. In this section, we show how to compile and build an executable file for the sample program. Then we demonstrate testing of this program to intercept and process a packet. The process is to change the lowercase letters of a payload in a packet to uppercase letters. To compile and build, we type ‘make’. Then an executable file, nd, is created.

```

root@server:~/nfqueue# make
g++ -DNDEDUP_TEST -o nd ndedup_main.cc ndedup.cc -lnetfilter_queue
-lnfnetlink

```

Prepare Testing For testing, we add a rule in iptable using the ‘INPUT’ chain as follows. ‘-dport 50000’ means that a packet only destined to the 50000 port is captured. In this example, we use the ‘INPUT’ chain for a receiver, but we can change the ‘INPUT’ chain to a ‘FORWARD’ chain for a forwarder. In either case, packets are intercepted and forwarded to the sample program through NFQUEUE. To check whether a rule has been added, we type ‘iptables -L -n’. We see there is a rule starting from ‘NFQUEUE tcp ...’. Then we run the sample program by typing the executable file ‘nd’.

```

iptables -I INPUT -p tcp -j NFQUEUE --dport 50000 --queue-num 0
--queue-bypass

```

```

root@server:~/nfqueue# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source      destination
NFQUEUE    tcp  --  0.0.0.0/0   0.0.0.0/0   tcp dpt:50000 NFQUEUE
num 0 bypass

```

```

Chain FORWARD (policy ACCEPT)
target     prot opt source      destination

```

```

Chain OUTPUT (policy ACCEPT)
target     prot opt source      destination

```

```

root@server:~/nfqueue# nd
opening library handle
unbinding existing nf_queue handler for AF_INET (if any)
binding nfnetlink_queue as nf_queue handler for AF_INET
binding this socket to queue '0'
setting copy_packet mode
// <— network dedup module is waiting for packets.

```

Initial Connection As an example, we open two terminals. One terminal is used for a sender, and the other terminal is used for a receiver. Please note that we need three terminals: one for a sender, another for a receiver and the last one for a network deduplication sample program. We run a TCP server with port 50000 using 'nc -l 50000' at a receiver. We also run a TCP client connecting to a receiver in a localhost.

```

// receiver
root@server:~# nc -l 50000
// waiting to receive a message

// sender
root@server:~# nc localhost 50000
// waiting to send a message

```

The following shows standard output that is printed by the network deduplication sample program when a sender and a receiver are connected through an initial TCP connection. The output contains an IP header and a TCP header. The IP header information includes the IP version, header length, total packet length, IP identification number, TTL, protocol (TCP), checksum, source IP address and destination IP address. The TCP header information contains the port numbers of the sender and receiver, sequence number, acknowledgement of sequence number, TCP header length, flags (urgent, ack, push, rst, syn, fin), window size and checksum. As we see, when a sender is connected to a receiver, two packets are transferred to each other: one is a SYN packet from a sender to a receiver, and the other is an acknowledge (ACK) packet from a receiver to a sender.

```

// output by network deduplication sample program
pkt received
**** IP header ****
version      : 4
header length : 20 (byte)
total length  : 60 (byte)
id           : 14648
ttl          : 64
protocol     : 6
checksum     : 0x382
source       : 127.0.0.1
destination  : 127.0.0.1
**** TCP header ****
sport        : 55492
dport        : 50000
seq          : 2993819844
ack seq      : 0
header length : 40 (byte)

```

```

flag (urgent) : 0
flag (ack)    : 0
flag (push)   : 0
flag (rst)    : 0
flag (syn)    : 1
flag (fin)    : 0
window size   : 43690 (byte)
checksum      : 0x1301
entering callback

pkt received
**** IP header ****
version       : 4
header length : 20 (byte)
total length  : 52 (byte)
id            : 14649
ttl           : 64
protocol      : 6
checksum      : 0x389
source        : 127.0.0.1
destination   : 127.0.0.1
**** TCP header ****
sport         : 55492
dport         : 50000
seq           : 2993819845
ack seq       : 1922593124
header length : 32 (byte)
flag (urgent) : 0
flag (ack)    : 1
flag (push)   : 0
flag (rst)    : 0
flag (syn)    : 0
flag (fin)    : 0
window size   : 342 (byte)
checksum      : 0xce56
entering callback

```

Packet Payload Change by Sample Program After a connection between sender and receiver is established, we send a message with ‘hello deduplication’ from a sender to a receiver. The following results show that a sender sends the ‘hello deduplication’ message (which has all lowercase letters) to a receiver. Then a receiver receives a ‘HELLO DEDUPLICATION’ message (with all uppercase letters) that was changed by the network deduplication sample program in a Linux bridge.

```

// sender side
root@server:~# nc localhost 50000
hello deduplication

// receiver side
root@server:~# nc -l 50000
HELLO DEDUPLICATION

```

```
// output by network deduplication sample program
pkt received
**** IP header ****
version      : 4
header length : 20 (byte)
total length  : 72 (byte)
id           : 14650
ttl          : 64
protocol     : 6
checksum     : 0x374
source       : 127.0.0.1
destination  : 127.0.0.1
**** TCP header ****
sport        : 55492
dport        : 50000
seq          : 2993819845
ack seq      : 1922593124
header length : 32 (byte)
flag (urgent) : 0
flag (ack)    : 1
flag (push)   : 1
flag (rst)    : 0
flag (syn)    : 0
flag (fin)    : 0
window size   : 342 (byte)
checksum      : 0x681d
**** Block ****
begin offset  : 52
end   offset  : 71
size         : 20
**** nf_payload(IP header + TCP header + TCP data) ****
>>> before modification
ip   checksum = 0x374
tcp  checksum = 0x681d
45000048393A4000400603747F0000017F000001D8C4C350B27210C
57298716480180156681D00000101080A08E6708E08E629FC68656C
6C6F2064656475706C696361746966F6E0A
HELLO DEDUPLICATION
>>> after modification
ip   checksum = 0x374
tcp  checksum = 0xa91e
45000048393A4000400603747F0000017F000001D8C4C350B27210C
57298716480180156A91E00000101080A08E6708E08E629FC48454C
4C4F2044454455504C49434154494F4E0A
entering callback
```

The output of the sample program shows the information of a payload (shown as a block) including ‘begin offset’, ‘end offset’ and ‘size’. ‘begin offset’ and ‘end offset’ are literally the beginning and ending offsets of a packet. ‘size’ is the size of the payload (‘hello deduplication’ including a newline character), which is 20 bytes. After the ‘**** Block ****’ section, it shows some useful information, including the change of checksum after modification of the payload. IP checksum (0x374) is not changed because the size of the payload is not changed, which means no

change is made to the IP header. However, TCP checksum (0x681d —> 0xa91e) is changed owing to the change in payload. The output also shows hexa codes of a packet before modification and after modification.

2.4.4.6 Existing Solution

One study [3] proposes the network-wide deployment of RE technology. Authors assume that the routers have the ability to detect and encode redundant content from network packets on the fly by comparing packet contents that were stored in a cache previously. In this approach, unique packets and the corresponding fingerprints of bytes in the packet payload are saved to a packet store and fingerprint store. When a packet arrives at a router, a small window slides on the payload in a packet, and fingerprints are computed for all windows. From among all the fingerprints representative ones are selected randomly. If the same fingerprints are found in the cache, the matched region from the pointed byte regions on a payload are expanded both to the left and to the right while comparing the two packets (incoming packet and packet in cache). The expanded region is replaced by a small shim header.

Figure 2.19 illustrates how many redundant packets are removed. Figure 2.19a shows the traditional shortest path routing where 18 packets are transferred from a sender to two destinations, D_1 and D_2 . Using RE on the routers, packet P_1 on each link is removed, as shown in Fig. 2.19b, which is a 33 % reduction of the total packets. This study proposes redundancy-aware routes based on the redundancy profile (which explains how often content is replicated across different destinations) for intra- and inter-domain routing. Figure 2.19c supports the idea that redundancies are further reduced using redundancy-aware routing, which amounts to a 44 % reduction of the total packets.

2.5 Deduplication Techniques by Time

2.5.1 Inline Deduplication

Inline deduplication is a deduplication technique that removes redundancies before data are stored on disk. Inline deduplication can be applied to primary workloads like email, user directories, databases and secondary workloads like archives and backups. Figure 2.20 elaborates how inline deduplication works for primary workloads (latency sensitive) as well as secondary workloads (throughput sensitive). For primary workloads, as shown in Fig. 2.20a, deduplication runs on a direct write and read path. When a user or client writes data, deduplication intercepts the data and checks for redundancies. Only unique data and indexes are saved to storage along with cache. Applications using primary workloads are highly latency sensitive; thus, deduplication typically uses in-memory cache to reduce disk I/O

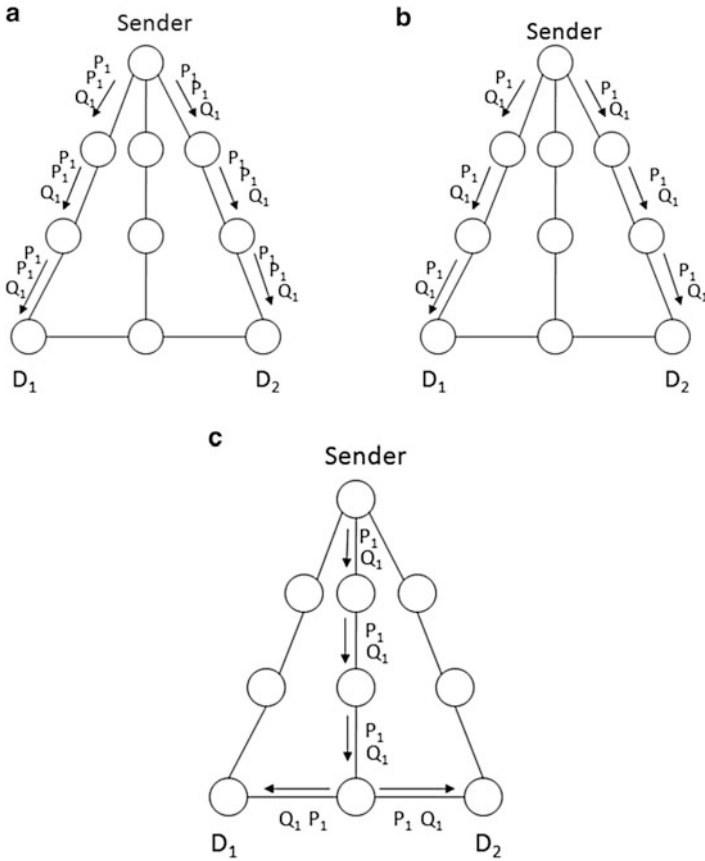


Fig. 2.19 Redundant traffic elimination with packet caches on routers. (a) No RE. (b) RE. (c) RE with redundancy-aware routing

requests. Figure 2.20b shows how deduplication works for secondary workloads. In these workloads, deduplication runs when data are archived or backed up on a backup server. The backup server does not maintain additional storage.

Inline deduplication has been proposed to remove redundancies for the primary workload [14, 44] and secondary workload [9, 26, 36, 50] without incurring extra space overhead or requiring more disk bandwidth. However, this approach requires latency overhead in a write path. iDedup [44] exploits temporal locality and spatial locality to maintain fast processing times in a write path. Content address storage (CAS) systems [17, 39] run inline deduplication because blocks are addressed by their fingerprints. A few file systems [6, 42] use inline deduplication for primary storage.

Inline deduplication [9, 13] runs deduplication before data are saved to disk storage. iDedup [44] has been proposed for inline deduplication for a primary workload.

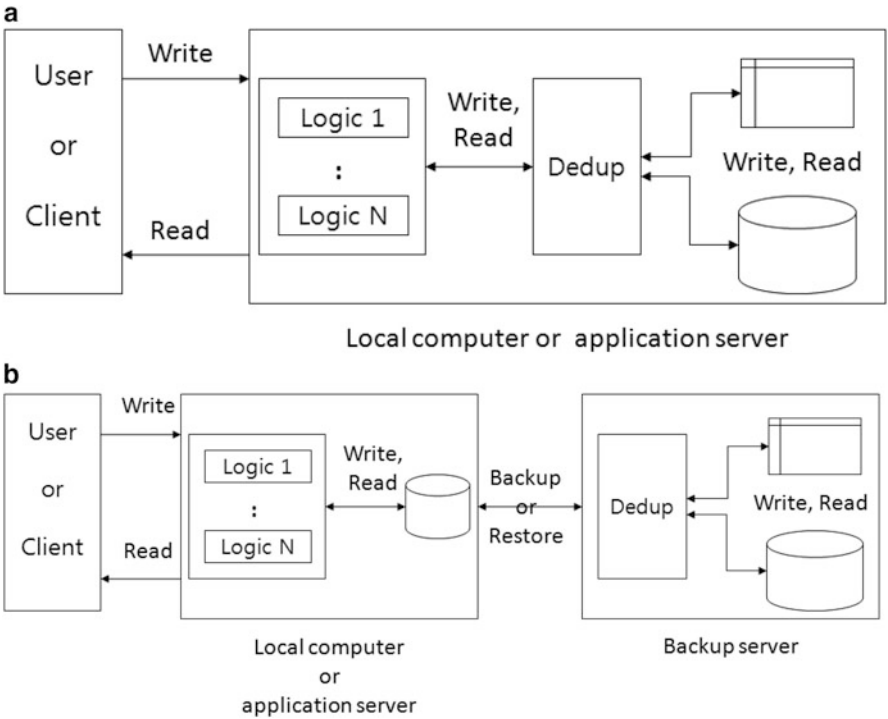


Fig. 2.20 Inline deduplication. (a) Inline deduplication for primary workloads. (b) Inline deduplication for secondary workloads

iDedup exploits spatial locality and temporal locality to achieve high performance (running time). For spatial locality, iDedup performs selective deduplication and mitigates the extra seek time for sequentially read files. For this purpose, iDedup examines blocks at write time and deduplicates full sequences of file blocks if and only if the sequences of blocks are (1) sequential in the file and (2) have duplicates that are sequential on disk. For temporal locality, iDedup maintains dedup-metadata as a *Least Recently Used (LRU)* cache by which iDedup avoids dedup-metadata I/O.

2.5.2 Offline Deduplication

Offline deduplication [1, 15, 21] runs deduplication after data are stored on disk; thus, it does not involve latency overhead in a write path but requires extra storage space. As shown in Fig. 2.21, data are saved to storage without deduplication. Offline deduplication runs out of a critical write and read path using already saved data, which does not hurt latency to write and read data. However, offline deduplication has several drawbacks: (1) extra disk space is needed to hold data

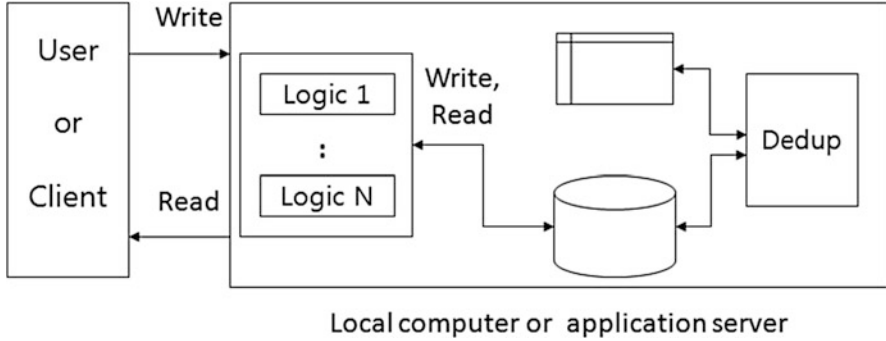


Fig. 2.21 Offline deduplication

temporarily before deduplication, (2) deduplication runs on system idle time, so deduplication can be very delayed if the system is running almost all the time, and (3) data on disk are loaded to memory for deduplication, so disk bandwidth is unnecessarily consumed.

ChunkStash [9] is a flash-assisted inline deduplication system where chunk metadata (with chunk index as key, and with chunk location and length as value) are saved to flash memory rather than disk. Considering that flash memory is 50 times faster than disk, ChunkStash reduces the penalty of index lookup misses in RAM, which increases inline deduplication throughput. ChunkStash also uses in-memory hash tables using the variant of cuckoo hashing [38], and compact key signatures rather than full keys are stored in the hash table, which reduces RAM size.

HYDRAsstor [13] is a grid of storage nodes. It works based on a distributed hash table (DHT) to save blocks to distributed storages. HYDRAsstor uses inline deduplication based on immutable and content-addressed and variable-sized blocks, data resilience by erasure coding, load balancing, and preservation of locality of data streams by prefetching. HYDRAsstor achieves scalability (by DHT), efficient utilization (by deduplication), fault tolerance (by data resiliency) and system performance (by load balancing, locality and prefetching).

2.6 Summary

In this chapter, we showed techniques for deduplication. We classified deduplication techniques based on various criteria, including granularity, deduplication place and deduplication time. We explained fundamental deduplication components, including chunk index cache and Bloom filters, along with implemented codes. Based on these criteria, we illustrated deduplication techniques such as file-level deduplication, fixed-size block deduplication, variable-sized block deduplication, server-based deduplication, client-based deduplication, end-to-end RE, network-wide RE, inline deduplication and offline deduplication.

References

1. Alvarez, C.: Netapp deduplication for FAS and v-series deployment and implementation guide (TR-3505). <http://www.netapp.com/us/media/tr-3505.pdf> (2011)
2. Amazon: Amazon simple storage service. <http://aws.amazon.com/s3/>
3. Anand, A., Gupta, A., Akella, A., Seshan, S., Shenker, S.: Packet caches on routers: the implications of universal redundant traffic elimination. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (2008)
4. Anand, A., Sekar, V., Akella, A.: SmartRE: an architecture for coordinated network-wide redundancy elimination. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (2009)
5. Bolosky, W., Corbin, S., Goebel, D., Douceur, J.: Single instance storage in Windows 2000. In: Proceedings of the 4th USENIX Windows Systems Symposium (2000)
6. Bonwick, J.: ZFS deduplication. https://blogs.oracle.com/bonwick/entry/zfs_dedup (2009)
7. Cisco: Wide area application services. <http://www.cisco.com/c/en/us/products/routers/wide-area-application-services/index.html>
8. Citrix: Cloudbridge. <http://www.citrix.com/products/cloudbridge/overview.html>
9. Debnath, B., Sengupta, S., Li, J.: ChunkStash: speeding up inline storage deduplication using flash memory. In: USENIX Annual Technical Conference (2010)
10. Dong, W., Douglass, F., Li, K., Patterson, R.H., Reddy, S., Shilane, P.: Tradeoffs in scalable data routing for deduplication clusters. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2011)
11. Drago, I., Mellia, M., Munafo, M., Sperotto, A., Sadre, R., Pras, A.: Inside dropbox: understanding personal cloud storage services. In: Proceedings of the 2012 ACM Conference on Internet Measurement Conference (IMC), pp. 481–494 (2012)
12. Dropbox: <http://www.dropbox.com>
13. Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M.: HYDRASor: a scalable secondary storage. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2009)
14. ElShimi, A., Kalach, R., Kumar, A., Oltean, A., Li, J., Sengupta, S.: Primary data deduplication-large scale study and system design. In: USENIX Annual Technical Conference (2012)
15. EMC: Achieving storage efficiency through EMC celerra data deduplication. <http://china.emc.com/collateral/hardware/white-papers/h6265-achieving-storage-efficiency-celerra-wp.pdf> (2009)
16. EMC: Avamar. <http://www.emc.com/backup-and-recovery/avamar/avamar.htm>
17. EMC: Centera: Content Addresses Storage System, Data Sheet. <http://www.emc.com/collateral/hardware/data-sheet/c931-emc-centera-cas-ds.pdf>
18. EMC: Networker. <http://www.emc.com/domains/legato/index.htm>
19. Guo, F., Efstathopoulos, P.: Building a high-performance deduplication system. In: USENIX Annual Technical Conference (2011)
20. Hu, W., Yang, T., Matthews, J.N.: The good, the bad and the ugly of consumer cloud storage. ACM SIGOPS Oper. Syst. Rev. **44**(3), 110–115 (2010)
21. IBM: IBM white paper: IBM storage tank - a distributed storage system. <https://www.usenix.org/legacy/events/fast02/wips/pease.pdf> (2002)
22. JustCloud: <http://www.justcloud.com/>
23. Kim, D., Choi, B.Y.: HEDS: hybrid deduplication approach for email servers. In: 2012 Fourth International Conference on Ubiquitous and Future Networks (ICUFN) (2012)
24. Kim, D., Song, S., Choi, B.Y.: SAFE: structure-aware file and email deduplication for cloud-based storage systems. In: Proceedings of the 2nd IEEE International Conference on Cloud Networking (2013)
25. Li, J., He, L.W., Sengupta, S., Aiyer, A.: Multimodal object de-duplication. Microsoft Corporation (2009). Patent

26. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., Camble, P.: Sparse indexing: large scale, inline deduplication using sampling and locality. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2009)
27. Liu, C., Lu, Y., Shi, C., Lu, G., Du, D., Wang, D.: ADMAD: Application-driven metadata aware de-duplication archival storage system. In: Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI), pp. 29–35 (2008)
28. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2011)
29. Microsoft: Exchange server 2003. <http://technet.microsoft.com/en-us/library/bb123872%28EXCHG.65%29.aspx>
30. Microsoft: Exchange server 2007. <http://www.microsoft.com/exchange/en-us/exchange-2007-overview.aspx>
31. Min, J., Yoon, D., Won, Y.: Efficient deduplication techniques for modern backup operation. *IEEE Trans. Comput.* **60**, 824–840 (2011)
32. Mozy: <http://mozy.com/>
33. Muthitacharoen, A., Chen, B., Mazières, D.: A low-bandwidth network file system. In: SOSP (2001)
34. National Institute of Standards and Technology (NIST): Secure Hash Standard 1 (SHA-1). <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf> (2015)
35. National Institute of Standards and Technology (NIST): Secure hash standard 256 (sha256). <http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf>
36. NEC: Hydrastor. <https://www.necam.com/hydrastor/>
37. Netfilter: Packet Flow. <https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg>
38. Pagh, R., Rodler, F.F.: Cuckoo hashing. *J. Algorithms* **51**(2), 122–144 (2004). doi:10.1016/j.jalgor.2003.12.002. <http://dx.doi.org/10.1016/j.jalgor.2003.12.002>
39. Quinlan, S., Dorward, S.: Venti: a new approach to archival storage. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2002)
40. Rabin, M.O.: Fingerprinting by random polynomials. Tech. Rep. Report TR-15-81, Harvard University (1981)
41. Riverbed: Steelhead for wan optimization. <http://www.riverbed.com/products/wan-optimization/>
42. Silverberg, S.: SDFS. <http://opendedup.org>
43. Spring, N.T., Wetherall, D.: A protocol-independent technique for eliminating redundant network traffic. In: Proceedings of the ACM SIGCOMM 2000 Conference on Data Communication (2000)
44. Srinivasan, K., Bisson, T., Goodson, G., Voruganti, K.: iDedup: latency-aware, inline data deduplication for primary storage. In: Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST) (2012)
45. Symantec: Netbackup. <http://www.symantec.com/netbackup>
46. Symantec: Puredisk. <http://www.symantec.com/netbackup-puredisk>
47. Weiss, M.A.: Data Structures and Algorithm Analysis in C++, 3rd edn. Addison Wesley, Reading, MA (2005)
48. Xia, W., Jiang, H., Feng, D., Hua, Y.: SiLo: a similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In: USENIX Annual Technical Conference (2011)
49. Yan, F., Tan, Y.: A method of object-based de-duplication. *J. Netw.* **6**(12), 1705–1712 (2011)
50. Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2008)

Data Deduplication for Data Optimization for Storage
and Network Systems

Kim, D.; Song, S.; Choi, B.-Y.

2017, XIII, 262 p. 89 illus., 61 illus. in color., Hardcover

ISBN: 978-3-319-42278-7