DedupSearch: Two-Phase Deduplication Aware Keyword Search

Nadav Elias, *Technion - Israel Institute of Technology;* Philip Shilane, *Dell Technologies;* Sarai Sheinvald, *ORT Braude College of Engineering;* Gala Yadgar, *Technion - Israel Institute of Technology*

https://www.usenix.org/conference/fast22/presentation/elias

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

DedupSearch: Two-Phase Deduplication Aware Keyword Search

Nadav Elias*, Philip Shilane†, Sarai Sheinvald§, Gala Yadgar*
*Computer Science Department, Technion †Dell Technologies §ORT Braude College of Engineering

Abstract

Deduplication is widely used to effectively increase the logical capacity of large-scale storage systems, by replacing redundant chunks of data with references to their unique copies. As a result, the logical size of a storage system may be many multiples of the physical data size. The many-to-one relationship between logical references and physical chunks complicates many functionalities supported by traditional storage systems, but, at the same time, presents an opportunity to rethink and optimize others. We focus on the offline task of searching for one or more byte strings (keywords) in a large data repository.

The traditional, *naïve*, search mechanism traverses the directory tree and reads the data chunks in the order in which they are referenced, fetching them from the underlying storage devices repeatedly if they are referenced multiple times. We propose the *DedupSearch* algorithm that operates in two phases: a physical phase that first scans the storage sequentially and processes each data chunk only once, recording keyword matches in a temporary result database, and a logical phase that then traverses the system's metadata in its logical order, attributing matches within chunks to the files that contain them. The main challenge is to identify keywords that are split between logically adjacent chunks. To do that, the physical phase records keyword prefixes and suffixes at chunk boundaries, and the logical phase matches these substrings when processing the file's metadata. We limit the memory usage of the result database by offloading records of tiny (one-character) partial matches to the SSD/HDD, and ensure that it is rarely accessed.

We compare DedupSearch to the naïve algorithm on datasets of different data types (text, code, and binaries), and show that it can reduce the overall search time by orders of magnitude.

1 Introduction

Deduplication first appeared with backup storage systems holding weeks of highly redundant content [32, 43, 48], with the purpose of reducing the physical capacity required to store the growing amounts of logical backup data. This is achieved by replacing redundant chunks of data with references to their unique copies, and can reduce the total physical storage to 2% of the logical data, or even less [43]. Deduplication has recently become a standard feature of many storage systems, including primary storage systems that support high IOPS and low latency accesses [18, 41]. Even with the lower redundancy levels in such systems, deduplication may reduce the required physical capacity to 12%-50% of the original data's size [18].

Most storage architectures distinguish between the logical view of files and objects and the physical layout of blocks of data on the storage media. In deduplicated storage, this distinction further creates multiple logical pointers, often from different files and even users, to each physical chunk. This many-to-one relationship complicates many functionalities that are supported by traditional storage systems, such as caching, capacity planning, and support for quality of service [25, 33, 40]. At the same time, it presents an opportunity to rethink other functionalities to be deduplication-aware and more efficient.

Keyword search is one such functionality, which is supported by some storage systems and is a necessary operation for numerous tasks. For example, an organization may need to find a document containing particular terms, and if the search is mandated by legal discovery [37], it has to be applied to backup systems [45] and document repositories that may include petabytes of content. Virus scans and inappropriate content searches may also include a phase of scanning for specified byte strings corresponding to a virus signature or a pirated software image [29, 44]. Finally, data analysis and machine learning tools often rely on preprocessing stages to identify relevant documents with a string search. Our focus is on *offline* search of large, deduplicated storage systems for legal or analytics purposes.

Logging and data analytics systems support fast keyword searches by constructing an index of strings during data ingestion [2, 8]. While they provide very fast lookup times, such indexes carry non-negligible overheads: their size is proportional to the logical size of the data, and thus they may consume a large fraction of the physical storage capacity [6, 31]. In addition, their data structures must be continually updated as new data is received. Thus, an index is typically not maintained in systems where search is a rare operation, such as backups. Another limitation of index structures is that they often assume a delimiter set such as whitespace, which is not useful for binary strings or more complex keyword patterns.

When an index is unavailable or cannot support the search query, an exhaustive scan of the data is required. A *naïve search* algorithm would process a file system by progressing through the files, opening each file, and scanning its content for the specified keywords. Even without the effects of deduplication, traversing the file system in its logical 'tree' order is inefficient due to fragmentation and resulting random accesses. With deduplication, a given chunk of data may be read repeatedly from storage, once for every file that references it.

We propose an alternative algorithm, *DedupSearch*, that progresses in two main phases. We begin with a *physical phase* that performs a physical scan of the storage system and scans each chunk of data for the keywords. This has the twin benefits of reading the data sequentially with large I/Os as well as reading each chunk of data only once. For each chunk of data, we record the exact matches of the keyword, if it is found, as well as prefixes or suffixes of the keyword (partial matches) found at chunk boundaries. We use a widely-used [11] string-matching algorithm to efficiently identify multiple keywords in a single scan, as well as their prefixes and suffixes.

We then continue with a *logical phase* that performs a logical scan of the filesystem by traversing the chunk pointers that make up the files. Instead of reading the actual data chunks, we check our records of exact and partial matches in those chunks, and whether partial matches in logically adjacent chunks complete the requested keyword. This mechanism lends itself to also supporting standard search parameters such as file types, modification times, paths, owners, etc.

The database of chunk-level matches generated during the physical scan can become excessively large when a keyword begins or ends with common byte patterns or characters, such as 'e'. Our experiments show that very short prefix and suffix matches can become a sizable fraction of the database even though they are rarely part of a completed query. We separate records of "tiny" partial matches into a dedicated database which is written to SSD/HDD and is accessed only when the tiny prefix/suffix is needed to complete the keyword match.

We implemented DedupSearch in the Destor open-source deduplication system [21], and evaluated it with three real-world datasets containing Linux kernel versions, Wikipedia archives, and virtual machine backups. DedupSearch is faster than the naïve search by orders of magnitude: its search time is proportionate to the physical size of the data, while the naïve search time increases with its logical size. Despite its potential overheads, the logical phase becomes dominant only when the number of files is very large compared to the size of the physical data, as is the case in the archives of the Linux kernel versions. Even in these use cases, DedupSearch outperforms the naïve search thanks to its efficient organization of the partial results, combined with reading each data chunk only once. These advantages are maintained when searching for multiple keywords at once and when varying the average chunk size and number of duplicate chunks in the system.

2 Background and Challenges

Data in deduplicated systems is split into chunks, which are typically 4KB-8KB in average size. Duplicate chunks are identified by their *fingerprint*—the result of hashing the chunk's content using a hash function with very low collision probability. These fingerprints are also used as the chunks' keys in the fingerprint-index, which contains the location of the chunk on the disk. When a new chunk is identified, it is written into a *container* that is several MBs in size to optimize disk writes. A container is written to the disk when it is full, possibly after its content is compressed. A file is represented by a *recipe* that lists the fingerprints of the file's chunks. Reading a file entails looking up the chunk locations in the fingerprint index, reading their containers (or container sub-regions) from the disk, and possibly decompressing them in memory.

Consider, for example, the four files in Figure 1(a). Each file contains two chunks of 5 bytes each, where some of the chunks have the same content. The total logical size of these files is eight chunks, and this is also their size in a traditional storage system, without deduplication. Figure 1(b) illustrates how these files will be stored in a storage system with deduplication. We assume, for simplicity, that the files were written in order of their IDs, and that the chunks are all of size 5 bytes.[1] When deduplication is applied, only four unique chunks are stored in the system, in two 10-Byte containers.

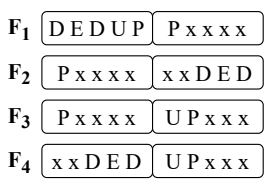
A keyword search in a traditional storage system would scan each files' chunks in order, with a total of eight sequential chunk reads. The same naïve search algorithm can also be applied to the deduplicated storage: following the file recipes it would scan the chunks in the following order: $C_0, C_1, C_1, C_2, C_1, C_3, C_2, C_3$—a total of eight chunk reads. If this access pattern spans a large number of containers (larger than the cache size), entire containers might be fetched from the disk several times. Moreover, the data in each chunk will be processed by the underlying keyword-search algorithm multiple times—once for each occurrence in a file.

Our key idea is to read and process each chunk in the system only once. Our algorithm begins with a *physical phase*, which reads all the containers in order of their physical addresses, and processes each of their chunks. In our example, we will perform two sequential container reads, and process a total of four chunks. The challenges in searching for keywords in the physical level result from the fact that most deduplication systems do not maintain "back pointers" from chunks to the files that contain them. Thus, we cannot directly associate keyword matches in a chunk with the corresponding file or files. Furthermore, keywords might be split between adjacent chunks in a file, preventing the identification of the keyword when searching the individual chunks.

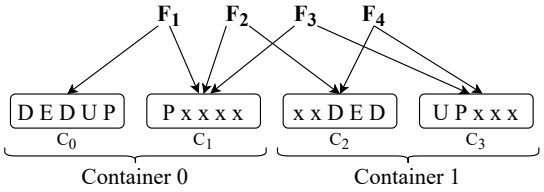
Consider, for example, searching for the keyword DEDUP in the files in Figure 1. The naïve search will easily identify the matches in files F_1 and F_4, even though the word is split between chunks C_2 and C_3. The physical search will only identify the exact match of the word in chunk C_0 but will not be able to correlate it with file F_1 or identify F_4 as a match.

To address these challenges, we add a *logical phase* following the completion of the physical phase, that collects the matches within the chunks and identifies the files that contain them. To identify keywords split between chunks, we must also record *partial matches*—prefixes of the keyword that appear at the end of a chunk and suffixes that appear at the beginning of

[1] At the host level, files are split into blocks. We assume, for this example, that each host-level block corresponds to a deduplication-level chunk.



(a) Traditional system



(b) Deduplicated system

FP	|Prefix|	|Suffix|	# Exact	Offset
FP_0	0	0	1	0
FP_1	0	1	0	0
FP_2	3	0	0	0
FP_3	0	2	0	0

(c) Chunk results

Figure 1: Four files containing four unique chunks in a traditional storage system (a) and in a deduplicated system (b), with their corresponding chunk-result records (c).

a chunk. For example, in addition to recording the full match in chunk C_0, the physical phase will also record the prefix of length 3 in the end of chunk C_2, and the suffix of size 2 in the beginning of chunk C_3. We must also record the suffix of length 1 in chunk C_1, to potentially match it with the prefix DEDU, even though this prefix does not appear in any chunk.

This introduces an additional challenge: some prefixes and suffixes might be very frequent in the searched text. Consider, for example, a keyword that begins with the letter 'e', whose frequency in English text is 12% [23]. Recording all prefix matches means we might have to record partial matches for 12% of the chunks in the system. In other words, the number of partial matches we must store during the physical phase is not proportionate to the number of keyword matches in the physical (or logical) data. This problem is aggravated if we search for multiple keywords during the same physical scan. In the worst case, we might have to store intermediate results for all or almost all the chunks in the system. In the following, we describe how our design addresses these challenges.

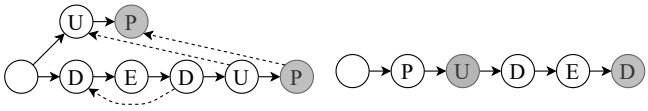
3 The Design of DedupSearch

We begin by describing the underlying keyword-search algorithm and how it is used to efficiently identify partial matches during the physical search phase. We then describe the data structures used to store the exact and partial matches between the two phases. Finally, we describe how the in-memory and on-disk databases are accessed efficiently for the generation of the full matches during the logical phase.

3.1 String-matching algorithm

To identify keyword matches within chunks, we use the *Aho-Corasick* string-matching algorithm [11]. This is a trie-based algorithm for matching multiple strings in a single scan of the input. We explain here the details relevant for our context, and refer the reader to the theoretical literature for a complete description of the algorithm and its complexity.

The *dictionary*—set of keywords to match—is inserted into a trie, which represents a finite-state deterministic automaton. The root of the trie is an empty node (*state*), and the edges between consecutive nodes within a keyword are called *child links*. Each child link represents a state transition that occurs when the next character in the input matches the next character in the keyword. Thus, each node in the trie represents the occurrence in the input of the substring represented by the path



(a) Search trie (b) Reverse trie

Figure 2: The Aho-Corasick trie (a) and reverse trie (b) for the dictionary {DEDUP,UP}

to that node. Specifically, each leaf represents an exact match of its keyword in the dictionary and is thus an accepting state in the automaton.

In addition to the child links, a special link is created between node u and node v whenever v is the longest strict suffix of u in the trie. These links are mainly used when the matching of an entire keyword fails, and are thus referred to in the literature as *failure links*. For example, Figure 2(a) illustrates the trie created for the dictionary {DEDUP,UP}, where the dashed arrows represent the failure links.

The characters in the input are used to traverse the automaton. If an accepting state is reached, the algorithm emits the corresponding keyword and its location in the input. If the search fails in an internal node (i.e., when the next character in the input does not correspond to any child link) with a failure link, this means that the substring at the end of the link occurs in the input, and the search continues from there. For example, if the input is DEDE, then after reading the first three characters we will reach the node corresponding to DED. After the next character, E, we will backtrack to the node corresponding to D, continuing the search from the same input location, immediately transitioning to the next node by traversing the child link E. There is an implicit failure link to the root from every node that does not have an explicit failure link to another node.

The failure links guarantee the linear complexity of the algorithm: they prevent it from having to backtrack to earlier positions in the input whenever one keyword is found, or when the search fails. For example, when the string DEDUP is identified in the input, the failure link to the node representing UP allows the algorithm to emit *all* the keywords that occur in the input so far, continuing the search from the current location. The overall complexity of the Aho-Corasick search is linear in the total length of the dictionary plus the length of the input plus the number of keyword matches.

We use the Aho-Corasick algorithm with minimal modification to identify keyword prefixes. When the end of a chunk is reached and the current state is an internal node, then this

node's corresponding substring is the longest substring of at least one keyword. We can traverse the path of failures links starting from this node and emit all the longest prefixes found. For example, if the chunk ends with the string DEDU, then the current state corresponds to this prefix of DEDUP. The failure link points to U, which is the longest prefix of UP.

To identify suffixes at the beginning of a chunk, we construct a trie for the *reverse dictionary*—the set of strings which are each a reverse of a string in the original dictionary. We use it to search, in reverse order, the first n bytes of the chunk, where n is the length of the longest string in the dictionary. For example, Figure 2(b) shows the trie for the reverse dictionary of {DEDUP,UP}. To find the suffixes in chunk C_3 from Figure 1(b), we use this trie on the (reverse) input string "XXXPU".

Partial matches. As demonstrated in Figure 1, keywords might be split between adjacent chunks. Let n denote the length of the keyword, and p_i and s_i denote a prefix and a suffix of length i, respectively. p_i and s_i are considered *prefix or suffix matches* if they constitute the last or first i characters in the chunk, respectively. A *full match* occurs if the jth chunk in the file contains a prefix match of length i and the $(j+1)$th chunk (likely not stored consecutively with the jth chunk) contains a suffix match of length $n-i$.

In some cases, a chunk may contain several prefix or suffix matches. For example, chunk C_2 in Figure 1(b) contains p_3=DED as well as p_1=D. Thus, this prefix can be part of two possible full matches if the following chunk contains either s_2=UP or s_4=EDUP. To minimize the size of the partial results generated by the physical phase, we record only the longest prefix and longest suffix in each chunk, if a partial match is found. Note that if a chunk contains a prefix match of length i (e.g., DED) and some suffix of this prefix is itself a prefix of size $j < i$ of the keyword (e.g., D), then the partial match of p_j is implied by the record of the match p_i.

To facilitate the identification of all possible full matches, we construct, for each keyword, the set of all prefix and suffix matches. For example, for the word DEDUP, a full match can be generated by combining the following pairs of longest partial matches: D+EDUP, DE+DUP, DED+UP, DEDU+P, and DED+EDUP. The pairs can be represented by a set of integer pairs corresponding to the substring lengths: $\{(1,4),(2,3),(3,2),(4,1),(3,4)\}$. This set is constructed offline, before the start of the logical phase. We store it in the *partial-match table*, which is kept in memory for the duration of the logical phase. It is implemented as a two dimensional array such that cell (i,j) holds a list of all match offsets found in $p_i + p_j$. For example, Table 1 is the partial-match table for keyword DEDUP, where the offsets are calculated with respect to the beginning of the prefix. For example, the entry $(3,4)$ indicates that a match begins two characters after the beginning of the partial match DED. During the logical phase, when adjacent chunks contain a prefix p_i and a suffix s_j, we check the table for the pair (i,j) to determine if and where a full match is found.

	$j=1$	2	3	4
$i=1$				0 [D+EDUP]
2			0 [DE+DUP]	
3		0 [DED+UP]		2 [DED+EDUP]
4	0 [DEDU+P]			

Table 1: Partial-match table for DEDUP

3.2 Match result database

Exact matches. Exact matches are identified within individual chunks during the physical phase. We record the existence of an exact match by the offset of its first character.A chunk may contain several exact matches, which would require recording an arbitrarily large number of offsets. In practice, however, the vast majority of the chunks contain at most one exact match. This led us to define our basic data structures as follows.

Chunk-result record: this is the basic record of search results in a single chunk. It contains five fields: `fingerprint` (20 bytes), `longest prefix length` (1 byte), `longest suffix length` (1 byte), `number of exact matches` (1 byte), and `offset` of the first exact match (2 bytes). The total (fixed) size of this object is 26 bytes, although it might vary with the system's fingerprint and maximum chunk sizes. Figure 1(c) shows the content of the chunk-result records for the chunks in Figure 1(b), when searching for the keyword DEDUP.

Location-list record: this is a variable-sized list of the locations which is allocated (and read) only if the chunk contains more than one exact match. The first field is the `fingerprint` (20 bytes), and the remaining fields contain one offset (within the chunk), each. The number of offset fields is recorded in the `number of exact matches` field in the corresponding chunk-result record. The value 255 is reserved to indicate that there are more than 254 exact matches in the chunk. In that case, we use the following alternative record.

Long location-list record: this object is identical to the location-list record, except for one additional field. Following the chunk fingerprint, we store the precise number of exact matches, whose value determines the number of offset fields in the record.

Tiny substrings. Keywords that begin or end with frequent letters in the alphabet might result in the allocation of numerous chunk-result records whose partial matches never generate a full match. To prevent these objects from unnecessarily inflating the output of the physical phase, we record them in a different record type and store them in a separate database (described below). Each *tiny-result record* contains three fields: `fingerprint` (20 bytes) and two Booleans, `prefix` and `suffix`, indicating whether the chunk contains a prefix match or a suffix match, respectively.

The tiny-result records are allocated only if this is the only match in the chunk, i.e., the chunk does not contain any exact match nor a partial match longer than one character. For example, the chunk-result record for chunk C_1 in Figure 1(c) will be replaced by a tiny-result record. Tiny-result records are accessed during the logical phase only if the adjacent chunk

contains a prefix or suffix of length $n-1$.[2]

We use tiny-result records for substrings of a single character: our results show that this captures the vast majority of tiny substrings, as we expected from text in a natural language [23]. However, when searching for non-ASCII keywords, we might encounter different patterns of tiny frequent substrings. The tiny-result records could then be used for variable-length substrings which are considered short. In this case, the record would contain an additional field indicating the length of the substring. To improve space utilization, several Boolean fields can be implemented within a single byte.

Multiple keywords. When the dictionary includes multiple keywords, we list them and assign each keyword its serial number as its ID. We then replace the individual per-chunk records with lists of <keyword-ID,result-fields> pairs. The structure of the records (chunk-result, locations-list, and tiny-result) is modified as follows. It includes one copy of the chunk fingerprint, followed by a list of <keyword-ID,result-fields> pairs. The result fields correspond to the fields in each of the three original records, and a pair is allocated for every keyword with non-empty fields. For example, if we were searching for two keywords, DEDUP and UP, then the chunk-result object for chunk C_3 in Figure 1(b) would include the following fields:

FP	ID	|Prel	|Sufl	#Exact	Off	ID	|Prel	|Sufl	#Exact	Off
FP_3	0	0	2	0	0	1	0	0	1	0

Database organization. We store the output of the physical search phase in three separate databases, where the chunk fingerprint is used as the lookup key. The *chunk-result index*, *location-list index*, and *tiny-result index* store the chunk-result records, location-list records, and tiny records, respectively. The first two databases are managed as in-memory hash tables. The tiny-result index is stored in a disk-based hash table. In a large-scale deduplicated system, chunks can be processed (and their results recorded) in parallel to take advantage of the parallelism in the underlying physical storage layout.

3.3 Generation of full search results

After all the chunks in the system have been processed, the logical phase begins. For each file in the system, the file recipe is read, and the fingerprints of its chunks are used to lookup result records in the database. The fingerprints are traversed in order of their chunk's appearance in the file. The process of collecting exact matches and combining partial matches for each fingerprint is described in detail in Algorithm 1, which is performed separately for every keyword.

This process starts by emitting the exact match in the chunk-result record, if a match is found (lines 4-5). If the chunk contains more than one match, it fetches the relevant location-list record and emits the additional matches (lines 6-9). If the chunk contains a suffix, it attempts to combine it with a prefix in the previous chunk (lines 10-14). If the chunk contains a prefix or a suffix of length $n-1$, then the tiny-result index is queried for the corresponding one-character suffix or prefix (lines 15-22). Thus, regular prefixes and suffixes (or tiny suffixes recorded in a regular chunk-result record) are matched when the suffix is found, while tiny substrings are matched when the respective $(n-1)$-length substring is found.

[2]This optimization is not effective for keywords of length 2. We do not include specific optimizations for this use case in our current design.

Algorithm 1 DedupSearch Logical Phase: handling FP_i in File F

Input: FP_i, FP_{i-1}, FP_{i+1}, res_{i-1}

```
1:  res_i ← chunk_result[FP_i]
2:  if res_i = NULL then
3:      return
4:  if res_i.exact_matches > 0 then
5:      add file name, match offset to output
6:      if res_i.exact_matches > 1 then
7:          locations ← list_locations[FP_i]
8:          for all offsets in locations do
9:              add file name, offset to output
10: if res_i.longest_suffix > 0 then
11:     if res_{i-1} ≠ NULL then
12:         if res_{i-1}.longest_prefix > 0 then
13:             for all matches in partial-match_table
                [res_{i-1}.longest_prefix, res_i.longest_suffix]
                do
14:                 add file name, match offset to output
15:     else if res_i.longest_suffix = n − 1 then
16:         tiny ← tiny_result[FP_{i-1}]
17:         if tiny ≠ NULL & tiny = prefix then
18:             add file name, match offset to output
19: if res_i.longest_prefix = n − 1 then
20:     tiny ← tiny_result[FP_{i+1}]
21:     if tiny ≠ NULL & tiny = suffix then
22:         add file name, match offset to output
```

The logical phase can also be parallelized to some extent: while each file's fingerprints must be processed sequentially, separate backups or files within them can be processed in parallel by multiple threads. Even for a large file, it is possible to process sub-portions of the file recipe in parallel. Both physical and logical phases can be further distributed between servers, requiring appropriate distributed result databases. This extension is outside the scope of this paper.

4 Implementation

We used the open-source deduplication system, Destor [21], for implementing DedupSearch (*DSearch*). The physical phase of DedupSearch is composed of two threads operating in parallel: one thread sequentially reads entire containers and inserts their chunks into the chunk queue. The second thread pops the chunks from the queue and processes them, as described in Sections 3.1 and 3.2: it identifies exact and partial matches of all the keywords, creates the respective result records, and stores them in their respective databases.

We used Destor's restore mechanism for implementing the logical phase. Destor's existing restore operates in three parallel threads: one thread reads the file recipes and inserts them into the recipe queue. Another thread pops the recipes from their queue, fetches the corresponding chunks by reading their containers, and inserts the chunks in order of their appearance in the file to the chunk queue. The last thread pops the chunks from their queue and writes them into the restored file.

The logical phase uses the second thread of the restore mechanism. It uses the fingerprints to fetch chunk-result records, rather than the chunks themselves, and inserts them into the result queue with the required metadata. An additional thread pops the result records from the queue, processes them according to Algorithm 1, and emits the respective full matches.

The implementation of the chunk-result index and location-list index is similar to Destor's fingerprint index. This is an in-memory hash table, whose content is staged to disk if the memory becomes full. The tiny-result index is implemented as an on-disk hash table using BerkeleyDB [5, 39]. We used BerkeleyDB's default setup with transactions disabled, because, in our current implementation, accesses to the tiny-result index are performed from a single thread in each phase.

We modified a publicly available implementation of the Aho-Corasick algorithm in C++ [22] to improve its data structures, memory locality, and suffix matching, and to support non-ASCII strings. For best integration of this implementation into Destor, we refactored the Destor code to use C++ instead of C. Our entire implementation of DedupSearch consists of approximately 1600 lines of code added to Destor, which is available online [1].

5 Evaluation Setup

For comparison with DedupSearch, we implemented the traditional (*Naïve*) search within the same framework, Destor. Naïve uses Destor's restore mechanism by modifying its last thread: instead of writing the chunk's data, it is processed with the Aho-Corasick trie of the input keywords. To identify keywords that are split between chunks, the last $n-1$ characters (where n is the length of the longest keyword) of the previous chunk are concatenated to the beginning of the current chunk.

We ran our experiments on a server running Ubuntu 16.04.7, equipped with 128GB DDR4 RAM and an Intel® Xeon® Silver 4210 CPU running at 2.40GHz. The backing store for Destor was a DellR 8DN1Y 1TB 2.5" SATA HDD, and the tiny-result index was stored on another identical HDD. We remounted Destor's partition before each experiment, to ensure it begins with a clean page cache.

5.1 Datasets

Our goal was to generate datasets that differ in their deduplication ratio and content type. To that end, we used data from three different sources—Wikipedia backups [9, 10], Linux kernel versions [4], and web server VM backups—and used Destor to create several distinct datasets from each source. Destor ingests all the data in a specified target directory, creating one backup file. This file includes the data chunks and the metadata required for reconstructing the individual files and directory tree of the original target directory. We created two or four versions of each of our datasets, each with a different average chunk size: 2KB, 4KB, 8KB, and 16KB.

Dataset	Logical size (GB)	Physical size + metadata size (GB) 2KB	4KB	8KB	16KB
Wiki-26 (skip)	1692		667+16 40.4%	861+9 51.4%	
Wiki-41 (consecutive)	2593		616+22 24.6%	838+12 32.8%	
Linux-197 (Minor versions)	58	10+1 19%	10+1 19%	11+1 20.7%	13+1 24.1%
Linux-408 (every 10th patch)	204	10+4 6.9%	10+4 6.9%	15+2 7.4%	16+2 8.8%
Linux-662 (every 5th patch)	377	10+7 4.5%	11+5 4.2%	13+4 4.5%	17+3 5.3%
Linux-1431 (every 2nd patch)	902	10+18 3.1%	11+13 2.7%	10+13 2.5%	17+8 2.8%
Linux-2703 (every patch)	1796	10+34 2.5%	10+26 2.0%	13+20 1.9%	17+17 1.9%
VM-37 (1-2 days skips)	2469	145+33 7.2%	129+18 6.0%	156+10 6.7%	192+5 8.0%
VM-20 (3-4 days skips)	1349	143+19 12.0%	125+10 10.0%	150+6 11.6%	181+3 13.6%

Table 2: The datasets used in our experiments. 2KB-16KB represent the average chunk size in each version. The value below the physical size is its percentage of the logical size.

The Linux version archive includes tarred backups of all the Linux kernel history, ordered by version, major revision, minor revision, and patch. The size of the kernel increased over time, from 32 MB in version 2.0 to 1128 MB in version 5.9.14 (the latest in our datasets). Naturally, versions with only a few patches between them are similar in content. Thus, by varying the number of versions included, we created five datasets that vary greatly in their logical size, but whose physical size is very similar, so the effective space savings increases with number of versions. All our Linux datasets span the same timeframe, but vary in the "backup frequency", i.e., the number of patches between each version. They are listed in Table 2.

The English Wikipedia is archived twice a month since 2017 [9, 10]. We used the archived versions that exclude media files, and consist of a single archive file, each. We created two datasets from these versions. Our first dataset includes 41 versions, covering three consecutive periods of 4, 5, and 15 months between 2017 and 2020 (chosen based on bandwidth considerations). To create the second dataset, we skipped every one or two versions, resulting in roughly half the logical size and almost the same physical size as the first dataset. The list of backups in each dataset appears in [19].

For experimenting with binary (non-ASCII) keywords, we created a dataset of 37 VM backups (`.vbk` files) of two WordPress servers in the Technion CS department over two periods of roughly two weeks each. The backups were generated every

one or two days, so as not to coincide with the existing, regular backup schedule of these servers. The first dataset consists of all 37 backups. The second consists of 20 of these backups, with longer intervals (three to four days) between them. Table 2 summarizes the sizes and content of our datasets.

5.2 Keywords

We created dictionaries of keywords with well-defined characteristics to evaluate the various aspects of DedupSearch. Specifically, we strived to include keywords that appear sufficiently often in the data, and to avoid outliers within the dictionaries, i.e., words that are considerably more common than others. We also wanted to distinguish between keywords with different probabilities of prefix or suffix matches, and different suffix and prefix length. Our dictionaries consist of multiple keywords, to evaluate the efficiency of DedupSearch in scenarios such as virus scans or offline legal searches.

We sampled 1% of a single Wikipedia backup (~1GB), and counted the occurrences of all the words in this sample, using white spaces as delimiters. As we expected, the frequency distribution of the keywords was highly skewed. We chose approximately 1000 words whose number of occurrences was similar (between 500 and 1000), and whose length is at least 4. We counted the number of occurrences of each keyword's prefixes and suffixes in the sample. We also calculated the average prefix and suffix length, which were less than 1.2 for all keywords, confirming that the vast majority of substring matches are of a single character. We then constructed the following three dictionaries of 128 keywords each: *Wiki-high*, *Wiki-low*, and *Wiki-med* contain keywords with the highest, lowest, and median number of prefixes and suffixes, respectively.

We repeated the process separately for Linux using an entire (single) Linux version, resulting in the corresponding dictionaries *Linux-high*, *Linux-low*, and *Linux-med*. We created an additional dictionary, *Linux-line*, that constitutes entire lines as search strings, separating strings by EOL instead of white spaces. We chose 1000 lines with a similar number of occurrences, sorted them by their prefix and suffix occurrences, and chose the lines that make up the middle of the list.

For the binary keyword dictionary, we sampled 1GB from both of the VM backups, and counted the number of occurrences of all the binary strings of length 16, 64, 256 and 1024 bytes. We chose strings with similar number of occurrences and the median number of prefix and suffix matches. The resulting dictionaries for the four keyword lengths are *VM-16*, *VM-64*, *VM-256*, and *VM-1024*. The statistics of all our dictionaries are summarized in Table 3.

Dictionary	Avg. pre/suf length	Avg. # pre/suf	Avg. # occurrences	Avg. keyword length
Wiki-high	1.09	85.3 M	722	8.4
Wiki-med	1.10	42.2 M	699	7.8
Wiki-low	1.08	5.7 M	677	6.0
Linux-high	1.09	64.8 M	653	10.5
Linux-med	1.20	32.8 M	599	10.4
Linux-low	1.13	5.7 M	583	10.4
Linux-line	1.22	31.4 M	63	25.9
VM-16	1.00	8.7 M	31	16
VM-64	1.00	8.6 M	29	64
VM-256	1.00	8.6 M	27	256
VM-1024	1.00	8.6 M	27	1024

Table 3: Characteristics of our keyword dictionaries.

6 Experimental Results

The goal of our experimental evaluation was to understand how DedupSearch (*DSearch*) compares to the Naïve search (*Naïve*), and how the performance of both algorithms is affected by the system parameters (dedup ratio, chunk size, number of files) and search parameters (dictionary size, frequency of substrings). We also wanted to evaluate the overheads of substring matching in DedupSearch, and how it varies with these system and search parameters.

6.1 DedupSearch performance

Effect of deduplication ratio. In our first set of experiments, we performed a search of a single keyword from the 'med' dictionaries, i.e., with a median number of substring occurrences. We repeated this search on all the datasets and chunk sizes detailed in Table 2. Figure 3 shows, for each experiment, the total search time and the time of the physical and logical phases of DedupSearch as compared to Naïve. The result of each experiment is an average of four independent experiments, each with a different keyword. The standard deviation was at most 6% of the average in all our measurements except one.[3]

We first observe that DedupSearch consistently outperforms Naïve, and that the difference between them increases as the deduplication ratio (the ratio between the physical size and the logical size) decreases. For example, with 8KB chunks, DedupSearch is 2.5× faster than Naïve on Linux-197 and 7.5× faster on Linux-2703. The total time of Naïve increases linearly with the logical size of the dataset, as the number of times chunks are read and processed increases. The total time of DedupSearch also increases with the number of versions. However, the increase occurs only in the logical phase, due to the increase in the number of file recipes that are processed. The time of the physical phase remains roughly the same, as it depends only on the physical size of the dataset.

Effect of chunk size. Chunk sizes present an inherent tradeoff in deduplicated storage: smaller chunks result in better deduplication, but increase the size of the fingerprint index. This tradeoff is also evident in the performance of both search algorithms. The search time of Naïve on the Linux datasets and most of the VM datasets decreases as the average chunk size increases. While this increases the physical data size, it reduces the number of times each container is read on average, as well as the number of times each chunk is processed. On the Wikipedia datasets and on the VM-37 dataset with

[3]The standard deviation of time of the logical phase in the Linux datasets was as high as 15%, due to the variation in the number of prefix and suffix matches for the different keywords.

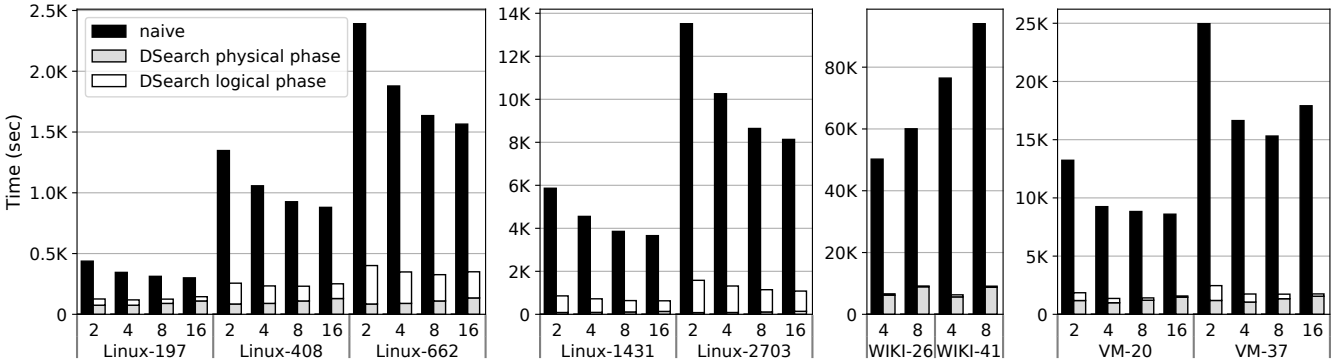


Figure 3: Search times of DedupSearch and Naïve with one word from the 'med' dictionary. The numbers 2-16 in the x-axis indicate the average chunk size in KB.

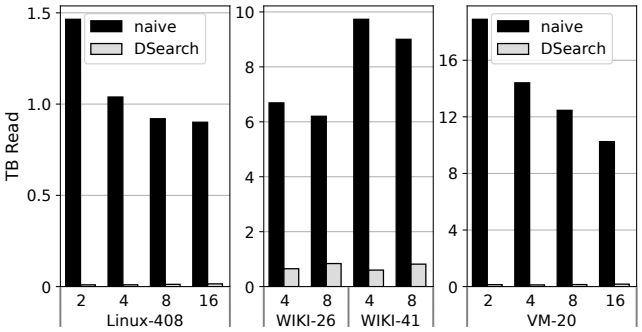


Figure 4: Amount of data read in DedupSearch and Naïve with one word from the 'med' dictionary. The numbers 2-16 in the x-axis indicate the average chunk size in KB.

16KB chunks, however, the increase in chunk size increases the search time of Naïve. The reason is their physical size, which is much larger than the page cache: although fewer containers are fetched by Destor, more of their pages miss in the cache and incur additional disk accesses.

The time of the physical phase in DedupSearch increases with the chunk size due to the corresponding increase in the data's physical size. This increase is most visible in our Wikipedia datasets, which are our largest datasets. In contrast, the logical phase is faster with larger chunks. The main reason is the reduction in the size of the file recipes and the number of chunk fingerprints they contain. Larger chunks also mean fewer chunk boundaries, which reduce the overall number of partial results that are stored and processed. These results were similar in all our datasets.

Figure 4 shows the amount of data read by both search algorithms on representative datasets. It confirms our observations that the main benefit of DedupSearch comes from reducing the amount of data read and processed by orders of magnitude, compared to Naïve. For Naïve, the amount of data read increases with the logical size and decreases with the chunk size. For DedupSearch, the amount of data read is proportionate to the physical size of the dataset, regardless of its logical size.

Effect of dictionary size. To evaluate the effect of the dictionary size on the efficiency of DedupSearch, we used subsets of different sizes from the 'med' dictionary. Figure 5 shows the results for the Linux-408 and Wikipedia-41 workloads with 8KB chunks (the results for the other datasets are similar). We repeated this experiment with two underlying keyword-search algorithms: Aho-Corasick, as explained in Section 3.1, and the native C++ `find`, described below. Both implementations use the result records, data structures, and matching algorithm described in Sections 3.2- 3.3, but differ in the way they handle multiple keywords and partial matches.

When the Aho-Corasick algorithm is used, the chunks' processing time (denoted as 'search chunks' in the figure) increases sub-linearly with the number of keywords in the search query. Nevertheless, the processing time is lower than the time required for reading the chunks from physical storage, which means that the time spent in the physical phase does not depend on the dictionary size. The logical phase, however, requires more time as the number of keywords increases: more keywords result in more exact and partial matches generated in the physical phase. As a result, more time is required to process the result records and to combine potential partial matches. We observe this increase only when the dictionary size increases beyond eight keywords. For smaller dictionaries (e.g., when comparing two keywords to one) increasing the number of keywords means that each thread of the logical phase processes more records per chunk. This reduces the frequency of accesses to the shared queues, thus reducing context switching and synchronization overheads. For example, the logical phase of Linux-408 with two keywords is five seconds faster than that with one keyword.

C++ `find` [3] scans the data until the first character in the keyword is encountered. When this happens, the scan halts and the following characters are compared to the keyword. If the string comparison succeeds, the match is emitted to the output. Regardless of whether a match was found or not, the scan then resumes from where it left off, which means the search backtracks whenever a keyword prefix is found in the data. This process is more efficient than Aho-Corasick when the number of keywords is small (see the difference in the 'search chunks' component): it's overhead is lower and its implementation is likely more efficient than our Aho-Corasick implementation. However, its search time increases linearly

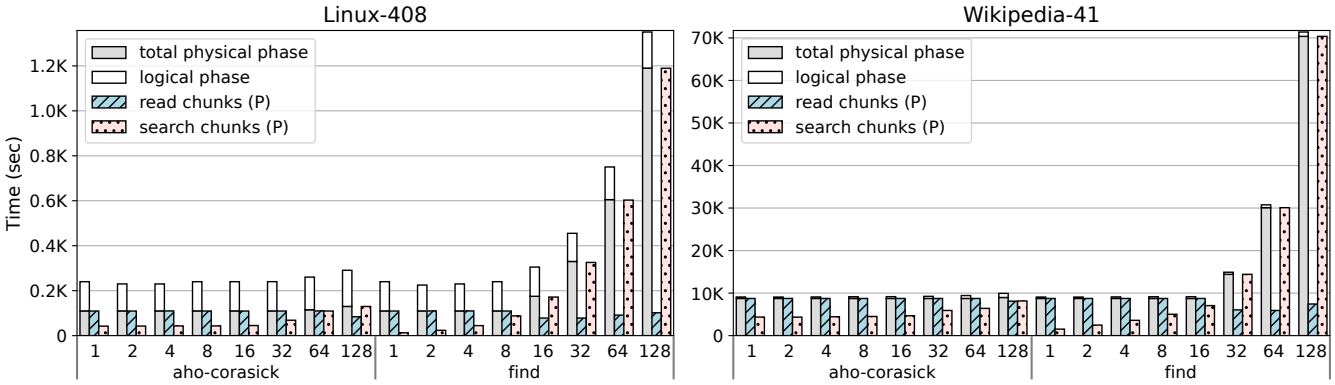


Figure 5: DedupSearch times of different number of keywords from the 'med' dictionary with Aho-Corasick vs. C++ `find` and 8KB chunks.

with the number of keywords: it exceeds the time used by Aho-Corasick when the dictionary size is 8 or higher, and its processing time exceeds the time required to read the physical chunks when the dictionary size exceeds 16 and 64, in Linux-408 and Wikipedia-41, respectively. The difference between the datasets stems from their different content: the prefixes in the Linux dictionaries are longer, which causes `find` to spend more time on string comparison.

Effect of keywords in the dictionary. To evaluate the effect of the type of keywords, we compared the search times of DedupSearch and Naïve (Figure 6) when using the full (128-word) dictionaries from Table 3 on four representative datasets: Linux-408, Linux-2703, Wikipedia-41, and VM-20, all with 8KB chunks. The results for all four binary (VM-*) dictionaries were identical, and so we present only results with 64-byte keywords. Our results show that in the physical phase, the time spent searching for keywords within the chunks increases with the number of substring occurrences: it is shortest for the 'low' dictionary and longest for the 'high' dictionary, where all the keywords start and end with popular characters (e, t, a, i, o, and '_'). The duration of the logical phase increases slightly with the number of substrings in the database, because more partial results are fetched and processed.

Surprisingly, as the chunk processing time increases, the time spent waiting for disk reads decreases. This reduction is a result of the operating system's readahead mechanism: the next container is being read in the background while the chunks in the current one are being processed. The page cache also explains the results of Naïve: it processes each chunk several times, but the processing time, which is higher with more prefixes and suffixes, is not masked by the reading time: many chunks already reside in the cache. Thus, Naïve is more sensitive to the dictionary type when searching the Linux datasets because they are small enough to fit almost entirely in memory.

6.2 DedupSearch data structures

Index sizes. Figure 7(top) shows the number of chunk-result, list-locations and tiny-result records that are generated by the physical phase when searching for a single keyword. Comparing the datasets to one another shows that the number of search results (rightmost, white bar) increases with the logical size, while the number of result records (i.e., objects stored in the database) depends only on the physical size. The results of each dataset are an average of four experiments, with four different words from the 'med' and '64' dictionaries. Unlike the performance results, the standard deviation here is larger because the results are highly sensitive to the number of substring matches of each keyword. However, the trend for each keyword is similar to the trend of the average in all the datasets.

This figure also shows that, in all the datasets, a large percentage of the records are tiny-result records (note the log scale of the y-axis). Figure 7(bottom) shows the size of each of the databases: the memory-resident chunk results and list locations, and the on-disk tiny results. The tiny results constitute 62%, 84% and 98% of the space occupied by the result databases in Linux-408, Wiki-41 and VM-20, respectively. Storing them on the disk successfully reduces the memory footprint of both logical and physical phases. The location lists occupy a small portion of the overall database size: 3%, 4% and 0% of the database size of Linux-408, Wiki-41 and VM-20, respectively. There are, on average, 3.3 offsets in each location list. Separating these offsets into dedicated records allows us to minimize the size of the more dominant chunk-result records.

Figure 8 shows the number of result records for a representative dataset, Linux-408 (the trend for the other datasets is similar), when varying the chunk size and the keyword type. When the number of chunks increases (chunk size decreases), more keyword matches are split between chunks. As a result, there are fewer exact matches and fewer list locations, but more chunk-result records with prefixes and suffixes, and more tiny-result records. The overall database size increases with the number of records, from 0.82 MB for 16KB chunks to 2.28 MB with 2KB chunks.

The results of the different dictionaries show the sensitivity of DedupSearch to the keyword type. Although the number of search results for the entire high, med, and low dictionaries is similar, the number of result records generated during the search varies drastically. For example, there are 4% more keyword matches when searching for the Linux-high dictionary than for Linux-med, but 55% [120%] more records [tiny records] in the database. Thanks to the compact representation

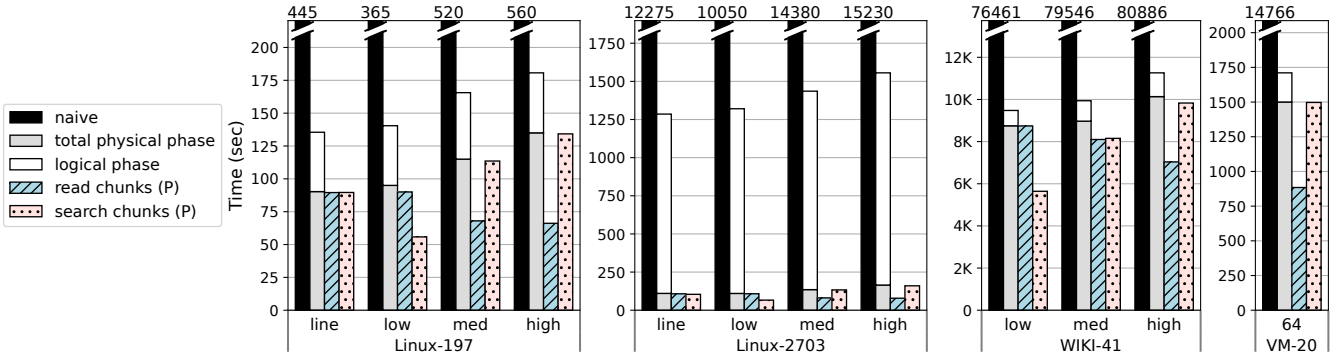


Figure 6: Breakdown of DedupSearch times of 128 words from all dictionaries and 8KB chunks.

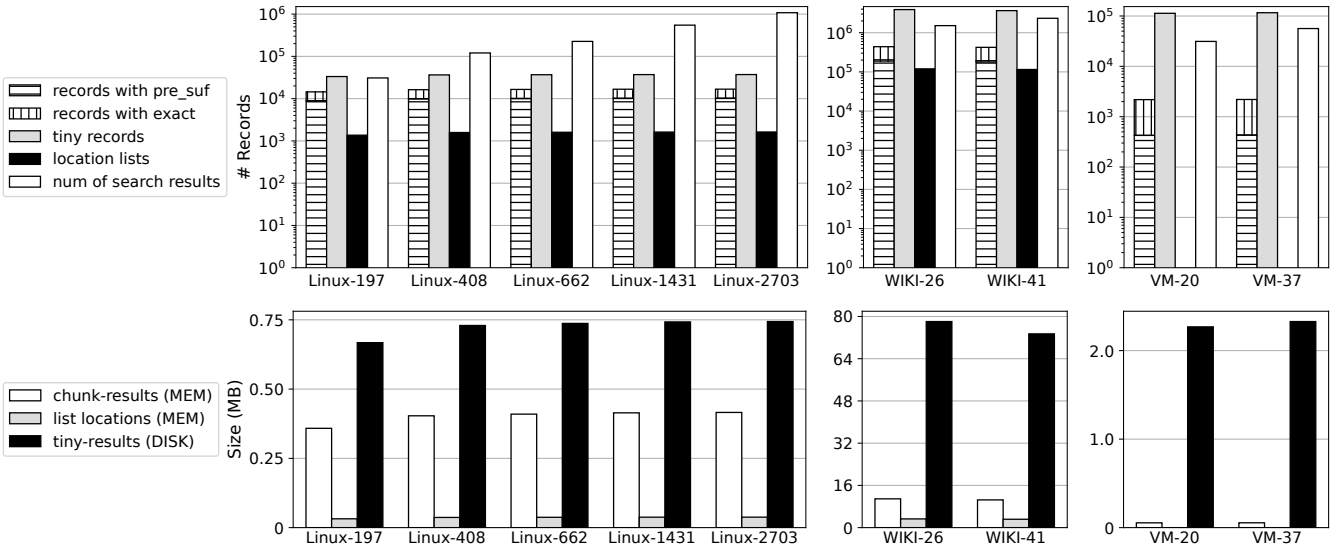


Figure 7: The number of search results and result objects during the search of a single keyword from the 'med' dictionary (top, note the log scale of the y-axis), and the corresponding database sizes (bottom).

of the tiny records (and their location on the disk), the database for Linux-high is only 16% larger than that of Linux-med, and its memory usage is also only 15% higher.

All the tiny-result databases in our experiments were small enough to fit in our server's memory. The largest tiny-result database, 3.6GBs, was created when searching for the Wiki-high dictionary on the Wikipedia-41 dataset with 4KB chunks. Nevertheless, we designed and implemented DedupSearch to avoid memory contention in much larger datasets.

Database accesses. Table 4 presents additional statistics of database usage and access during the search of keywords from the 'med' and '64' dictionaries (on the 8KB-chunk datasets). The top line for each dataset presents an average of four experiments, each with a different word from the dictionary. The bottom line presents results for searching the entire dictionary. Less than 0.2% of the keyword matches were split between chunks in the textual (Linux and Wikipedia) datasets. The percentage of split results was higher in the binary datasets because the keywords in the dictionary were considerably longer.

The number of accesses to the tiny-result index increases with the dataset's logical size and with the number of keywords. However, it is still several orders of magnitude lower than the number of records in the database: the tiny-result index is accessed only when the rest of the keyword is found in the chunk. The probability that the missing character is found in the adjacent chunk ('tiny hit') depends on the choice of keywords. For comparison, the percentage of successful substring matches out of all attempts is approximately 5% in the Linux datasets and 30% in the Wikipedia datasets. These differences are due to the different text types in the two datasets, and to some short (4-letter) keywords in the Wikipedia dataset.

Although the number of accesses to the tiny-result index can be as high as hundreds of thousands when searching large dictionaries, these numbers are orders of magnitude smaller than the random accesses that Naïve performs when fetching the data chunks in their logical order. Furthermore, repeated accesses to the index result in page-cache hits, as the operating system caches frequently accessed portions of the index. Thus, even though our on-disk index represents the worst-case performance of DedupSearch, we expect that moving it to memory would not add significant performance gains.

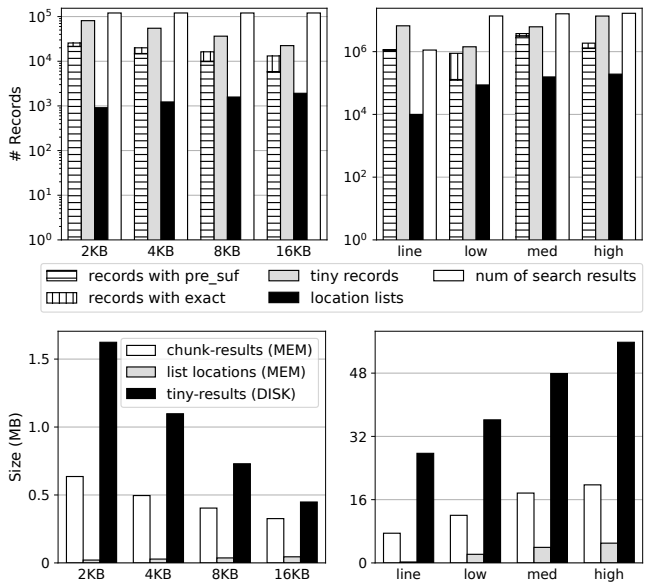


Figure 8: The number of search results, result records and the corresponding database sizes in Linux-408 during a search of one keyword (left) and entire dictionaries (right).

Dataset	# results (M)	% matches split	# tiny records (M)	# tiny accesses	tiny hit rate
Wiki-26	1.52	0.05	3.90	1167	0.10
	208.50	0.10	490.31	44,719	0.94
Wiki-41	2.34	0.05	3.67	1780	0.10
	321.07	0.09	459.96	69,094	0.94
Linux-197	0.03	0.19	0.03	59	0.08
	5.08	0.12	4.19	1,665	0.73
Linux-408	0.12	0.19	0.04	197	0.15
	16.08	0.11	4.57	5,986	0.71
Linux-662	0.23	0.19	0.04	360	0.16
	29.16	0.11	4.63	11,101	0.70
Linux-1431	0.55	0.18	0.04	855	0.16
	68.96	0.11	4.67	26,682	0.70
Linux-2703	1.08	0.18	0.04	1673	0.17
	134.65	0.11	4.68	52,391	0.69
VM-20	0.03	0.00	0.11	0	N/A
	4.02	1.61	14.62	0	N/A
VM-37	0.06	0.00	0.12	0	N/A
	7.24	1.61	14.96	0	N/A

Table 4: Percentage of keywords split between chunks and usage of the tiny-result index. The numbers are from searching one (top) and 128 (bottom) keywords from the 'med' and '64' dictionaries.

Dataset	Logical size	Physical size	Dedup ratio	Naïve time	DSearch time (logical)
Wiki-1	76	76	99.8%	616	620 (11)
LNX-1	1	0.80	80%	7.4	6.7 (0.6)
LNX-1-merge	0.82	0.78	95%	6.2	6.1 (0.1)
LNX-408	204	17	7.4%	926	231 (121)
LNX-408-merge	169	19	11.2%	768	203 (28)

Table 5: The size (in GB) and dedup ratio of the datasets created from a single archived version with 8KB chunks, and the time (in seconds) to search a single keyword from the 'med' dictionary.

6.3 DedupSearch overheads

Physical phase. To measure the time of the extra processing per chunk, we created five mini-backups with no duplicates of container-sized (4 MB) samples from Wikipedia. We read the data in advance and kept it in memory for the duration of the experiment to eliminate I/O delays. We measured the time of the processing thread of the physical phase (containing the keyword search algorithm and storing the records) and compared it to the processing time of Naïve. We performed a search with one and 128 keywords from the 'high' dictionary. With one word, DedupSearch and Naïve spent the same time processing the data. However, with 128 keywords, the processing thread of DedupSearch ran 32% longer than that of Naïve. The reason is that the number of records stored by DedupSearch is not proportional to the number of full matches, especially in the 'high' dictionary. With 128 keywords, there are hundreds of records, compared to dozens with one keyword. This experiment represents the worst-case for DedupSearch, with the highest number of partial matches and zero I/O cost. In practice, we expect such situations to be rare.

Logical phase. In addition to the datasets described in Table 2, we created three small datasets, each consisting of a single archived Linux/Wikipedia version. Table 5 shows the characteristics of these datasets. They exhibit the least amount of deduplication, allowing us to evaluate the overheads of DedupSearch in use-cases where its benefits are minimal. The table also shows the time spent by Naïve and by DedupSearch when searching a single keyword from the 'med' dictionary.

In the Wikipedia dataset, which exhibits minimal deduplication, DedupSearch is slower than Naïve by 0.8%. The reason is that Naïve emits its search results as soon as the chunks are processed, while DedupSearch requires the additional logical phase. DedupSearch reads and processes 20% and 0.02% less data, respectively (recall that data is read and processed in the granularity of containers and chunks, respectively).

In the Linux dataset, the physical size is 20% smaller than the logical size, and thus the physical phase of DedupSearch is shorter than Naïve's total time. The logical phase on this dataset, however, is 600 msecs, which are 9% of the total time of DedupSearch. The reason is the large number of files (64K) in the single Linux version. The logical phase parallelizes reading the file recipes from disk, fetching chunk results from their database, and collecting the full matches for the files. As the number of files increases, the overhead of context switching and synchronization between the threads increases.

To illustrate this effect, we include a merged dataset of the same Linux version, where the content of the entire archived version is concatenated into a single file. The physical size of Linux-1 and Linux-1-merge is similar and so is the time of the physical phase when searching them. The logical phase, however, is six times shorter, because it has to process only a single file recipe. We repeated this experiment with a larger number of versions: we created the Linux-408-merge dataset

by concatenating each of the versions in the Linux-408 dataset into a single file. This dataset contains 408 files, compared to a total of 15M files in Linux-408. The logical phase when searching the merged dataset is 4.3× faster. This effect also explains the long times of the logical phase when searching the Linux datasets (see Figure 3). The number of files in each Linux dataset increases from 4.3M in Linux-197 to 133M in Linux-2703. We conclude that the overheads of DedupSearch are low, even when the deduplication is very low. When deduplication ratios are high, these overheads become negligible as DedupSearch is faster than Naive by orders of magnitude.

7 Discussion

Extended search options. DedupSearch lends itself to several extensions that can enhance the functionality of the search. The first is the use of "wildcards"—special characters that represent entire character groups, such as numbers, punctuation marks, etc. Grep-style '*' wildcards can be supported by creating a dictionary that includes all the precise (non-*) substrings in the query. The logical phase would have to ensure that they all appear in the file in the correct order.

It would be more challenging to support keywords that span more than two chunks, since our prefix/suffix approach is insufficient. We would have to also identify chunks whose entire content constitutes a substring of the keyword, which means attempting to match the chunk content starting at all possible offsets within the keyword. Supporting regular expressions is similarly challenging, because the matched expression might span more than two chunks.

Approximate search. Some applications of keyword search only require the list of files containing the keyword, without the offset of all occurrences within the file, so the logical phase can stop processing a file's recipe as soon as a keyword is found. This eliminates the need for the location-list records. Alternatively, a best-effort search could focus on exact matches within a chunk for a faster, though imperfect search.

Additional applications. Dividing the search into physical and logical phases can potentially accelerate keyword search in highly fragmented or log-structured file systems as well as copy-on-write snapshots where logically adjacent data blocks are not necessarily physically adjacent.

The benefit of DedupSearch might be smaller when a large portion of the chunks can be marked irrelevant a priori by Naïve. Examples include binary data in textual search, file-system metadata, or chunks belonging to files that are excluded from the search for various reasons. Additional aspects that may affect the performance of DedupSearch and its advantage over Naïve include the underlying storage media (i.e., faster SSD), and parallel processing of chunks and file recipes. We leave these for future work.

8 Related Work

Deduplication is a maturing field, and we direct readers to survey papers for general background material [35, 46]. Our search technique follows on previous work that processed post-deduplication data sequentially along with an analysis phase on the file recipes, which has been applied to garbage collection and data migration [16, 17, 33]. We leverage this basic concept by processing the post-deduplication data with large, sequential I/Os instead of a logical scan of the file system with random I/O. Thus far, we have not found previous research that optimized string search for deduplicated storage.

String matching. String matching is a classical problem with a rich family of solutions that are used in a variety of areas. The longstanding character-based exact string matching algorithms are still at the heart of modern search tools. These include the Boyer-Moore algorithms [14], hashing-based algorithms such as Rabin-Karp [27], and suffix-automata based methods such as Knuth-Morris-Pratt [28] and Aho-Corasick [11]. ACCH [15] accelerates Aho-Corasick on Compressed HTTP Traffic by recording partial matches in referenced substrings. GPU-based string matching is used in network intrusion detection systems [42, 47].

Indexing. Offline algorithms use indexing to achieve sub-linear search time. Indexing methods include suffix-trees [24], metric trees [12] and *n*-gram methods [34], and the rank and select structure for compressed indexing [20]. Indeed, many systems scan the data in advance to map terms to their locations. Examples include Elasticsearch [2], Splunk [8], and CLP [38] for log searches and Apache Solr [7] for full-text search. An index is useful when queries are frequent and latency must be low. Its downside is that it precludes searching keywords that are not indexed, such as full sentences or arbitrary binary strings. Moreover, its size might become a substantial fraction of the dataset size: 5-10% in practice [6, 31]. Our approach is thus more appropriate when queries are infrequent and moderate latency is acceptable such as in legal discovery [37, 45].

DedupSearch can be viewed as a form of **near-storage processing**, where the storage system supports certain computations to reduce I/O traffic and memory usage. For example, YourSQL [26] and REGISTOR [36] offload functions to the SSD. BAD-FS [13] and Quiver [30] coordinate batch workloads or jobs to minimize I/O in large-scale systems. DedupSearch shares their underlying principle of fetching and/or processing data once for use in several contexts.

9 Conclusions

String search is a widely-used storage function and can be redesigned to leverage the properties of deduplicated storage. We present a two-phase search algorithm. The physical phase scans the storage space and stores matches per chunk. Most of the results are stored on the disk while only the popular are in memory. The logical phase goes over all file recipes and uses the chunk results to collect matches, including matches that span logically consecutive chunks. Our evaluation demonstrates significant savings of time and reads in DedupSearch in comparison to the Naïve search, thanks to the physical scan that reads duplicated chunks only once.

Acknowledgments

We thank the reviewers and our shepherd, Andrea Arpaci-Dusseau, for their feedback and suggestions, Amnon Hanuhov for help with the Aho-Corasick implementation, and Dita Jacobovitz for help with the VM datasets. This research was supported by the Israel Science Foundation (grant No. 807/20).

References

[1] DedupSearch implementation. https://github.com/NadavElias/DedupSearch.

[2] Elasticsearch: The heart of the free and open Elastic Stack. //https://www.elastic.co/elasticsearch/.

[3] libstdc++. https://gcc.gnu.org/onlinedocs/gcc-7.5.0/libstdc++/api/a00293_source.html#l01188.

[4] Linux Kernel Archives. https://mirrors.edge.kernel.org/pub/linux/kernel/.

[5] Oracle Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html.

[6] Search indexing in Windows 10: FAQ. https://support.microsoft.com/en-us/windows/search-indexing-in-windows-10-faq-da061c83-af6b-095c-0f7a-4dfecda4d15a.

[7] Solr. https://solr.apache.org/.

[8] splunk. https://www.splunk.com/.

[9] Wikimedia data dump torrents. https://meta.wikimedia.org/wiki/Data_dump_torrents.

[10] Wikimedia downloads. https://dumps.wikimedia.org/enwiki/.

[11] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[12] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207)*, pages 14–22, 1998.

[13] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control a batch-aware distributed file system. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[14] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[15] Anat Bremler-Barr and Yaron Koral. Accelerating multipattern matching on compressed HTTP traffic. *IEEE/ACM Transactions on Networking*, 20(3):970–983, 2012.

[16] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[17] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[18] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication—large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[19] Nadav Elias. Keyword search in deduplicated storage systems. Technical report, Computer science department, Technion, 2021.

[20] Antonio Fariòa, Susana Ladra, Oscar Pedreira, and Ángeles S. Places. Rank and select for succinct data structures. *Electron. Notes Theor. Comput. Sci.*, 236:131–145, April 2009.

[21] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[22] Christopher Gilbert. Aho-Corasick implementation (C++). https://github.com/cjgdev/aho_corasick.

[23] Gintautas Grigas and Anita Juskeviciene. Letter frequency analysis of languages using latin alphabet. *International Linguistics Research*, 1:p18, 03 2018.

[24] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, USA, 1997.

[25] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen Kat. Sketching volume capacities in deduplicated storage. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[26] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[27] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[28] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, March 1977.

[29] Geoff Kuenning. How does a computer virus scan work? *Scientific American*, January 2002.

[30] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[31] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Transactions on Information Systems (TOIS)*, 19(3):217–241, 2001.

[32] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[33] Aviv Nachman, Gala Yadgar, and Sarai Sheinvald. GoSeed: Generating an optimal seeding plan for deduplicated storage. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[34] Gonzalo Navarro, Ricardo Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng Bull*, 24:19–27, 11 2001.

[35] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):1–30, 2014.

[36] Shuyi Pei, Jing Yang, and Qing Yang. REGISTOR: A platform for unstructured data processing inside SSD storage. *ACM Trans. Storage*, 15(1), March 2019.

[37] Martin H Redish. Electronic discovery and the litigation matrix. *Duke Law Journal*, 51:561, 2001.

[38] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[39] Margo I. Seltzer and Ozan Yigit. A new hashing package for UNIX. In *USENIX Winter*, 1991.

[40] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[41] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[42] Giorgos Vasiliadis and Sotiris Ioannidis. *GrAVity: A Massively Parallel Antivirus Engine*, pages 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[43] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[44] Jau-Hwang Wang, Peter S Deng, Yi-Shen Fan, Li-Jing Jaw, and Yu-Ching Liu. Virus detection using data mining techinques. In *IEEE 37th Annual 2003 International Carnahan Conference on Security Technology, 2003. Proceedings.*, pages 71–76. IEEE, 2003.

[45] Kenneth J Withers. Computer-based discovery in federal civil litigation. *Fed. Cts. Law Rev.*, 1:65, 2006.

[46] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[47] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *2006 Symposium on Architecture For Networking And Communications Systems*, pages 93–102, 2006.

[48] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.