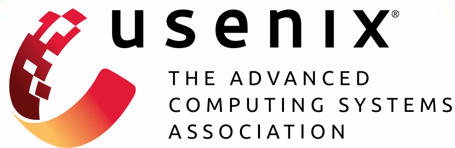


Physical vs. Logical Indexing with IDEA: Inverted Deduplication-Aware Index

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

https://www.usenix.org/conference/fast24/presentation/levi

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

NetApp®

Physical vs. Logical Indexing with IDEA: Inverted Deduplication-Aware Index

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

Abstract

In the realm of information retrieval, the need to maintain reliable term-indexing has grown more acute in recent years, with vast amounts of ever-growing online data searched by a large number of search-engine users and used for data mining and natural language processing. At the same time, an increasing portion of primary storage systems employ data deduplication, where duplicate logical data chunks are replaced with references to a unique physical copy.

We show that indexing deduplicated data with deduplication-oblivious mechanisms might result in extreme inefficiencies: the index size would increase in proportion to the logical data size, regardless of its duplication ratio, consuming excessive storage and memory and slowing down lookups. In addition, the logically sequential accesses during index creation would be transformed into random and redundant accesses to the physical chunks. Indeed, to the best of our knowledge, term indexing is not supported by any deduplicating storage system.

In this paper, we propose the design of a deduplication-aware term-index that addresses these challenges. *IDEA* maps terms to the unique chunks that contain them, and maps each chunk to the files in which it is contained. This basic design concept improves the index performance and can support advanced functionalities such as inline indexing, result ranking, and proximity search. Our prototype implementation based on Lucene (the search engine at the core of Elasticsearch) shows that IDEA can reduce the index size and indexing time by up to 73% and 94%, respectively, and reduce term-lookup latency by up to 82% and 59% for single and multi-term queries, respectively.

1 Introduction

One of the most effective ways to address growing storage requirements in datacenters is data deduplication: duplicate *chunks* of data are identified and replaced by references to a single unique copy of each chunk. The mechanisms involved in data deduplication have been optimized in numerous studies and commercial systems. As a result, most backup and archival systems [25,80], as well as many *primary* (non-backup) storage systems and appliances [20,24,35,43], currently support data deduplication.

Data deduplication entails a distinction between the user's *logical* data and the *physical* chunks stored in the system. This additional level of abstraction introduces new challenges in data management. The implicit sharing of content between files complicates, for example, garbage collection [39,40,62], load balancing between volumes [30,37,38,49,51], caching [44,55,56], and charge-back [69]. Fragmentation, which results from newly written files referencing a combination of 'old' chunks and newly written chunks, transforms logically-sequential data accesses to random I/Os in the underlying physical media. This has been addressed in the context of file-read and restore performance [33,45,57,63] and in full-system scans [42].

In this paper, we address *keyword indexing*, an important functionality that is supported by many storage systems [17,26,27] and is severely complicated by deduplication. Specifically, we refer to term-to-file indexing (also known as *inverted indexing*), which supports queries that return the files containing a *keyword* or *term*. Inverted indexes are widely used for simple queries, e.g., by users on personal computers, as well as for complex and batch queries involving multiple terms in a large-scale repository, e.g., by search engines [36], data analytics jobs [61,64,70], and legal discovery [67,77]. The searched data might be deduplicated, e.g., in shared file systems, code repositories, or systems storing similar VM images.

Two aspects of keyword indexing are affected by deduplication. The first is initial index creation time: the system is scanned by processing the logical files, generating random accesses to physical chunks. In addition, chunks are processed redundantly when there are multiple references to a chunk due to deduplication. The second aspect is the index size, which is proportional to the logical data size rather than to the physical size stored in the system: each term must point to all the files containing it, even if the files' content is almost identical. The inflated index size can result in poor lookup performance and also overshadow any capacity savings achieved by deduplication.

Indeed, to the best of our knowledge, systems with high deduplication ratios (i.e., a large number of references to each unique chunk) typically *do not support full keyword indexing*. For example, VMware vSphere [25] and Commvault [19] support file indexing, which only identifies individual files within a backup according to their metadata. Dell-EMC Data Protection Search [21] supports full content indexing, but warns that "processing the full content of a large number of files can be time consuming" and recommends performing targeted indexing on

specific backups or file types.

We address these challenges by *deduplication-aware keyword indexing*. We introduce *IDEA*, which replaces the term-to-file mapping in traditional indexes with a term-to-chunk mapping, whose size is proportional to the unique content physically stored in the system. An additional chunk-to-file mapping records references from chunks to the files they are contained within. This 'reverse mapping' is significantly smaller than the term-to-chunk map and can be stored in a smaller and faster storage device such as SSD or NVRAM. IDEA focuses on textual data. It uses a *white-space aware* content-defined chunking algorithm that creates chunk boundaries that align with white-space characters. This ensures that terms are not split between adjacent chunks.

IDEA creates the index by sequentially processing the physical data instead of the logical data. The term-to-chunk mapping is created by standard term-indexing software, which scans all the physical chunks in the system, disregarding their logical order in the containing files. The chunk-to-file mapping is created by scanning the file metadata, which is typically stored separately from the data chunks. Term lookup in IDEA begins with querying the term-to-chunk mapping. The set of resulting chunks is then used for lookup in the chunk-to-file map, producing a set of matching files.

This basic design of IDEA can support additional functionalities provided by traditional indexes, including low-overhead, incremental indexing of incoming data streams as part of their ingestion, ranking of documents with metrics such as TF-IDF, and returning, for each file in the query result, the offsets in which the keywords were found.

We make the following contributions in this paper:

- We identify and demonstrate the challenges involved in indexing deduplicated data.
- We propose IDEA, the first design of a deduplication-aware term index.
- We describe a prototype implementation of IDEA. For this prototype, we integrated Lucene [2], an open-source single-node inverted index software (similar to that used by the distributed Elasticsearch [4]), into the Destor deduplicating storage system [46].
- We compare the performance of IDEA to a naïve, deduplication unaware, index. On datasets of Linux kernel versions and of English Wikipedia archives, IDEA significantly reduced the indexing and lookup times. For the datasets with a high deduplication ratio, it also reduced the index size.

The rest of this paper is organized as follows. Section 2 gives background and surveys relevant related work, and Section 3 identifies the challenges involved in indexing deduplicated data. Sections 4 and 5 describe the design and implementation of IDEA. Our evaluation setup is described in Section 6, and our evaluation results are analyzed in Section 7. We discuss possible extensions of IDEA in Section 8 and conclude in Section 9.

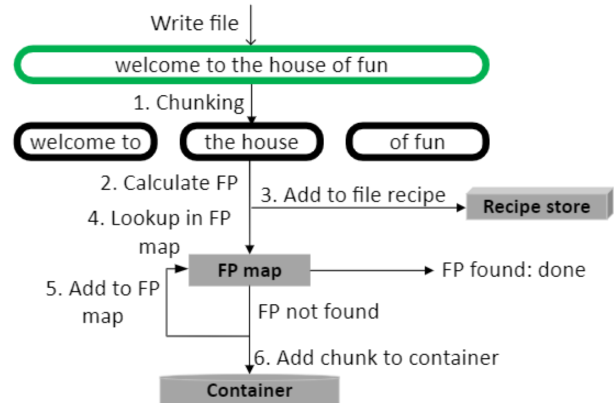


Figure 1: The basic deduplication process.

File	*Content*
F1	Let me into the house
F2	The house of the opera
F3	Welcome to the house of fun

Term	*File (Offset)*
fun	F3(24)
house	F1(16),F2(4),F3(15)
into	F1(7)
let	F1(0)
...	...

Term	*File only*
fun	F3
house	F1,F2,F3
into	F1
let	F1
me	F1
of	F2,F3
opera	F2
the	F1,F2,F3
to	F3
welcome	F3

Figure 2: A toy example of an index of three files.

2 Background and Related Work

Data deduplication. Deduplicating storage systems process incoming data to identify duplicate content and replace it with references to content already stored in the system. Figure 1 gives a schematic view of the main mechanisms of the deduplication process. The data is first split into *chunks* whose average size is typically 4KB-8KB, in a process referred to as *chunking*.

A chunk is represented by a cryptographic hash of its content, referred to as its *fingerprint*. The fingerprint map is queried to determine whether an incoming chunk is already stored in the system. If the chunk is new, it is written and its fingerprint is added to the fingerprint map. Each file is represented by a *file recipe* which contains the file metadata, a list of its chunks' fingerprints, and their sizes. Thus, to read (or *restore*) a file, its recipe is read and its chunks are located by searching in the fingerprint map or a cache of its entries.

The unique physical chunks are written in a log-structured manner, in the order in which they are added to the system. Backup and archival systems usually aggregate chunks, compress them, and pack the compressed data into *containers*, which are the unit of I/O. Containers are several MB in size, and decompression is necessary when restoring chunks. In contrast, deduplication systems for primary storage [34, 41, 43, 72], and especially deduplicating file systems [20, 24], might support direct access to individual chunks.

Keyword indexing. An *inverted index* or a *keyword index*, is a data structure which points from *terms* to their *occurrences*

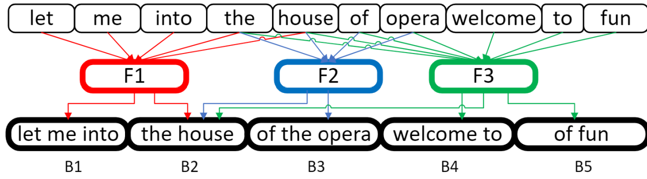


Figure 3: A naive implementation of an index in a deduplicated file system. The resulting inverted index is similar to that in Figure 2.

in a collection of documents. Terms (or *keywords*) can be any searchable strings and are typically natural language words. Any data structure which implements a key-value mapping can be used as an inverted index. From here on, we refer to a keyword index as simply an index, and use *map* to describe individual data structures in the system. Figure 2 shows an example of a small inverted index of a dataset containing three files, where each file is an indexed document. An *index lookup* or *query* returns, for each included term, the list of files containing this term, and optionally the byte offsets in which the term appears. For example, looking up the term "house" in this example will return $\{F_1, F_2, F_3\}$. If offsets are stored, the query will return $\{F_1(16), F_2(4), F_3(15)\}$. Storing the term offsets increases the index size, and is thus supported as an option.

The process of building an index is referred to as *indexing*, and includes the following steps [58]: (1) collecting the documents, i.e., reading the indexed files, (2) identifying the terms within each document, (3) linguistically normalizing the terms (e.g., eliminating plural form and capitalization), and (4) creating the list of documents, and optionally offsets, containing each term. An index can also be built incrementally by a series of *index updates*, where a new set of documents is processed and the existing index is updated to reflect the terms appearing in them. Creating an inverted index is known to be time and resource consuming. Distributing the index and its creation was thus included as a use case in the seminal MapReduce paper [36]. Most index designs support the removal of documents by marking deleted documents and garbage collecting index entries. Some designs avoid the resulting fragmentation in the index structure by batching deletions and rebuilding the index later [2, 58].

Indexing is a key mechanism in information retrieval and presents several challenges that have attracted a wide range of research efforts. One challenge is handling a high rate of incoming new data from sources, such as social media platforms and news services, which needs to be indexed [74, 75], possibly while simultaneously remaining responsive to user queries. Another challenge is supporting not only direct search, but also *similarity search*, in order to provide users with additional related search results [29, 52, 71]. Indexing is used in a wide range of contexts. For example, in natural language processing, an index is used for pre-training and fine-tuning the language model [48, 54]. Indexed pattern matching also plays a key role in bioinformatics [61, 64], data mining [70] and multimedia retrieval [73]. Since the data held in the index itself may be very large [74], an extensive body of work addresses *compressed indexes* with the goal of fitting them in memory [60, 68, 78].

Of available commercial indexing products, the most well-known is Elasticsearch—a distributed search engine supporting full-document indexing and real-time analytics [4, 47]. Elasticsearch is built on top of the single-node Apache Lucene [2]—an open-source full-text search-engine library. Lucene combines a document store with an inverted index that supports searching within any field of the indexed documents, simple lookups, complex queries, analytics jobs, and offsets. Lucene's underlying data structure is based on a hierarchy of skip-lists, which enable sequential access when a query contains multiple terms.

Lucene and its variations serve as the underlying engine of many more commercial indexing products, such as Apache Solr™ [13] and Amazon OpenSearch [1]. IBM Watson [5] is based on distributed Lucene and Indri [6] for indexing large corpora as well as semantic entries and relations between words. Other products support similar document and search interfaces with alternative data structures. For example, Meilisearch [10] is based on LMDB [8] which is implemented with B+ trees, and TypeSense [14] uses the LSM-tree-based RocksDB [12] for its mapping.

Specialized solutions enable search inside compressed structured data such as logs or time-series data. Examples include rapid exhaustive search [22, 23], lazy on-demand indexing of log fields [18], and highly effective in-memory caching of logs [65]. These special-purpose solutions are tightly coupled with the structure of the data and are not directly applicable to the general case of unstructured deduplicated data.

Result ranking. To maximize their relevance, lookup results are typically ranked by index systems, using a *scoring* formula on each result. Among the most popular such formulas, which we use in this paper, is *TF-IDF* [66], used by Lucene (and therefore in Elasticsearch). TF-IDF is commonly defined as follows. Given a document d in which a lookup term t is found, the score TF-IDF(t,d) is defined as $TF(t,d) \cdot IDF(t)$ where $TF(t,d) = \sqrt{\frac{\text{\# occurrences of t in d}}{\text{\# words in d}}}$ and $IDF(t) = 1 + log(\frac{\text{\# docs in the system}}{1+\text{\# docs in which t appears}})$.

Intuitively, TF measures how frequently a term appears in the document, and IDF measures the term significance, based on its occurrence in the entire corpus. Keeping the byte offsets of the terms allows measuring additional attributes such as proximity between multiple terms [32].

3 Challenges

When it was first commercialized, deduplication was primarily applied to backup and archival storage of 'cold' data, which is only rarely read and processed [80]. Since then, however, two separate trends have changed the way deduplicated data is accessed. The first is the growing need to process cold data, including old backups. Common scenarios include full-system scans for malware and anomaly detection [53, 76], as well as

keyword searches for legal disclosure [67, 77]. Enterprise applications might perform complex analytics queries, involving multiple-term lookups, on cold storage [31]. In disaster recovery situations, needed VMs may be identified with search terms and then run directly from backup storage until a primary system is restored [28]. These scenarios were addressed in a recent study of deduplication-aware search [42]. However, in the absence of an index, these exhaustive searches might be prohibitively slow.

The second trend is the growing application of deduplication on primary storage of 'hot' and 'warm' data that is accessed regularly [69]. As a result, deduplicating storage appliances are expected to support various functions, including keyword search. For example, users might perform single-term searches for files within their deduplicated personal workstation or their home directory on a deduplicated shared storage partition.

Since indexing software operates at the file-system level, it is unaware of the underlying deduplication at the storage system. A deduplication-unaware (*naïve*) index would thus schematically resemble the structure in Figure 3. The index will map terms to files, independently of how the files' content is stored in the underlying media. However, this oblivious design is inefficient due to the following three challenges.

Challenge 1: index size. The size of the index grows with the number of distinct terms as well as the number of files in the system. In traditional storage systems, this size is roughly proportional to the size of the stored data. As the data size grows, the storage capacity is scaled accordingly, accommodating the growing index. In deduplicated storage, however, the index grows with the logical data, as every new file must be reflected in the terms' document list, even if its content is almost identical to that of files already in the system. An increased index size might also increase the latency of lookups due to logarithmic search complexity and because smaller portions of it will fit into DRAM.

Challenge 2: indexing time. The indexing process scans all the files in the system to create the list of terms in each document and then the list of documents for each term. Performing such a scan on deduplicated data will result in random I/Os for reading the chunks in the order they appear in the files. For example, creating the index in Figure 3 would perform the following series of chunk reads: $[B_1, B_2, B_2, B_3, B_4, B_2, B_5]$. Chunk B_2 is accessed and processed three times, once for every file it is contained in. Furthermore, recall that chunks are read by fetching their entire container or a compression region within a container. Reading chunks in random order might thus cause high read amplification.

Challenge 3: split terms. Although the chunks in our example contain entire words, the chunking process will likely split the incoming data into chunks at arbitrary positions, splitting words between adjacent chunks. Thus, the terms in the beginning or end of a chunk can be correctly identified only when considering the chunks adjacent to it in each file. Therefore, even if the chunk is identified as duplicate, it must be processed in the context of each file that contains it.

As a result of these challenges, to the best of our knowledge, current deduplicating storage systems do not support indexing of their entire content.

4 IDEA

In this section, we describe the design of our deduplication-aware index, IDEA. We begin with an overview of the key concepts, and then describe each component in detail.

4.1 Overview

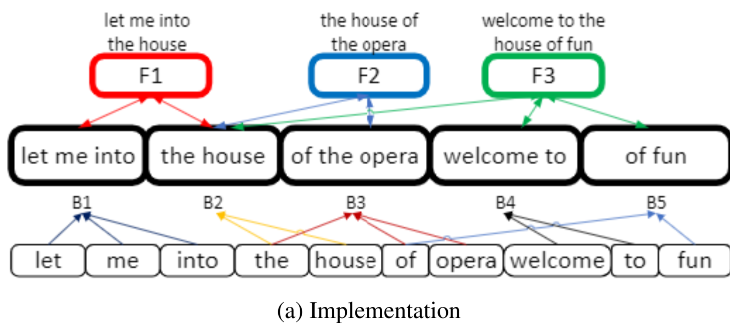
The key idea of deduplication-aware indexing is to map terms to the unique physical chunks they appear in, instead of the logical documents whose number might be disproportionately high. We replace the term-to-file mapping of the traditional index with two complementing maps: a *term-to-chunk map* and a *chunk-to-file map*. The lookup process first finds all the chunks containing the queried terms, and then finds the files containing these chunks. Figure 4(a) depicts the deduplication-aware index that replaces the naïve index in Figure 3. The logical term-to-file mapping from Figure 2 is realized by the combination of the two maps in Figures 4(b) and 4(c), with the file paths resolved by the *file-to-path map* in Figure 4(d). We use file IDs (generated as an internal serial number) in the term-to-file map because they are much smaller than the full file paths. From hereon, we refer to the file IDs as files.

This design allows deduplicating storage systems to provide index functionality to their users. The system can construct The term-to-chunk map with standard indexing software (e.g., Lucene), by passing the chunks as documents for indexing. The chunk-to-file map is based on information from the file recipes, and can be implemented by any standard key-value store. The only modification required in the deduplication system is the chunking process, to ensure that chunk boundaries do not split terms between chunks. We modify the chunking procedure to be *white-space aware* and enforce chunk boundaries only between words.

Properties. The term-to-chunk map is the largest part of the deduplication-aware index. Its size and creation time are proportional to the number of physical chunks. In systems with a high deduplication ratio, this map will be smaller than the term-to-file map in traditional indexing, and will incur lower lookup latency. On the other hand, many optimizations within the traditional index data structures are most effective when files are large (e.g., compressed encoding of file IDs or offsets within files). Processing individual chunks instead of entire files eliminates some of their benefits in a deduplication-aware index. We discuss these cases in detail in Section 7.

4.2 White-space aligned chunking

Deduplication systems employ two types of chunking mechanisms. *Fixed-sized chunking* splits the incoming data into fixed-sized chunks and is typically used in primary storage to align the deduplicated chunks with those of the storage interface. In



(a) Implementation

Term	*Chunk (Offsets)*
fun	B5(3)
house	B2(4)
into	B1(7)
let	B1(0)
me	B1(4)
of	B3(0),B5(0)
opera	B3(7)
the	B2(0),B3(3)
to	B4(8)
welcome	B4(0)

(b) Term-to-chunk

Chunk	*File (Offsets)*
B1	F1(0)
B2	F1(11),F2(0), F3(10)
B3	F2(10)
B4	F3(0)
B5	F3(21)

(c) Chunk-to-file

File	*Path*
F1	home\file1
F2	shared\file3
F3	fun\lyrics\file3

(d) File-to-path

Figure 4: An illustration of a deduplication-aware index (a) and its related maps (b-d) for the files and terms in Figure 2.

	Chunk 1	Chunk 2	Chunk 3
Incoming data	welcome to the house of fun		
Traditional	welcome t	o the hous	e of fun
Whitespace-CDC	welcome to	the house	of fun
Whitespace-fixed	welcome	to the	house of fun

Figure 5: The effect of white-space alignment on chunk content.

other words, the chunks are aligned to the operating-system pages and to the storage-device blocks [20, 24, 34, 43, 56]. *Content-defined chunking (CDC)* splits the data into variable-sized chunks where the hash produced over a rolling window matches a predefined mask. This indicates the start of a new chunk [50, 79, 80]. This method ensures that small differences between similar files will be contained within a small number of chunks and has been shown to achieve better deduplication efficiency than fixed-size chunking [59, 80].

Both techniques are agnostic to word boundaries and will likely end a chunk in the middle of a word. Thus, we modify both to be *white-space aware* and create chunk boundaries that align with white-space and other characters that preserve the locality of words within chunks. These characters are the delimiters used by the indexing software to parse terms during document processing. Thus, white-space awareness is not restricted to a specific encoding or language. In our implementation, the delimiters are defined by the C function `isspace()`.

Content-defined chunking. Systems that use content-defined chunking are designed to handle variable-sized chunks. Thus, extending this mechanism to be white-space aware is relatively straightforward: when it identifies a chunk boundary, instead of immediately triggering a new chunk, we continue scanning the following characters until a white-space character is encountered. This character ends the current chunk and starts the next chunk at the character immediately after it. If a white-space character is not encountered within a sufficiently long distance from the original boundary (512B in our implementation), we leave it unchanged—we assume the split string is irrelevant for indexing anyway.

Fixed-size chunking. Systems that use fixed-size chunking require chunks to fit into fixed-sized memory buffers and/or storage blocks. Thus, if the chunk boundary splits a term in two, we cannot extend this chunk until the end of this term. Instead, when an end of a chunk is identified (by calculating the offset from the beginning of the chunk), we scan this chunk *backwards* until a white-space character is encountered. This character ends the current chunk and starts the next chunk at the character immediately after it. Figure 5 demonstrates the effect of white-space aligned chunking on a small file example.

Although white-space alignment converts fixed-sized chunks to variable-sized chunks, it does not interfere with the deduplication system's operation. The resulting chunks are always smaller than the fixed size, and can thus be stored in a single block (i.e., hard-disk sector or flash page). In addition, recall that file recipes record the size of each chunk, and can thus handle chunks smaller than the fixed size. In case of a deduplicating file system, it can trim the block in memory to the chunk boundary. Additional file system changes might be required to support variable-sized chunks within larger fixed-sized blocks, e.g., recording the chunk size in the inode. These changes are beyond the scope of this project.

Non-textual content. Aligning chunks to white-spaces is only effective in case of textual content, and will have little effect on arbitrary binary content. We thus apply white-space alignment only to chunking of textual content. We identify this content by the file extension of the incoming data, e.g., `.txt`, `.c`, `.h`, and `.htm` files. This distinction during the chunking step allows us to also identify candidate chunks for indexing, and to exclude non-textual content from the indexing process. This is similar to how traditional indexing excludes files based on their extension, e.g., executable files.

The modifications to the deduplication mechanism are minimal. In our implementation, we add a Boolean field to the metadata of each chunk in the file recipe and in its container, indicating whether it is a 'text' chunk or not. During index creation, described in the following subsections, we only process chunks marked as textual.[1] We note, however, that our

[1]We experimented with workloads that contained a mix of textual and non-textual data, and found that the non-textual data does not affect the performance of IDEA. We thus omit those results from our evaluation.

chunking and indexing approaches do not preclude processing of binary content. Non-textual strings are identified by the indexing software (e.g., by the 'tokenizer' in Lucene) and are excluded from the mapping.

Overhead. White-space aligned chunking requires additional processing during the chunking step, and alters the location of chunk boundaries. We verified that enabling white-space awareness increases the chunking time by no more than 0.6% for content-defined chunking. The fixed-size chunking time increased by up to 3×, although it was still only 0.5% of the content-defined chunking time. The resulting number of chunks was within 0.4% and 0.15% of the number of chunks created by content-defined and fixed-size chunking, respectively. The difference in average chunk size was similarly negligible. White-space alignment reduced the deduplication ratio (percentage of data removed by deduplication) of content-defined chunking by no more than 0.4%, and marginally improved that of fixed-size chunking. The experiments were done on the LNX-198 and Wiki-4 datasets, described in Section 6.

4.3 Term-to-chunk mapping

The term-to-chunk map is an inverted index whose documents are physical chunks instead of logical files. The white-space aligned chunking described above ensures that chunks include complete terms, preventing arbitrary prefixes or suffixes from being incorrectly indexed. The number of documents in the index is the number of physical chunks, which might be higher than the number of logical files. The effect of this design choice on the size of the index is evaluated in Section 7.

During indexing, the chunks are read sequentially by fetching entire containers or compression regions, and each chunk is processed only once, regardless of the number of files containing it. Since each chunk is processed independently, processing the chunks is easily parallelizable. We leave related optimizations for future work. A term-lookup in the term-to-chunk map returns the fingerprints of the chunks this term appears in (and optionally its offsets within them). The fingerprints are used for lookup in the chunk-to-file map, described later in this section.

4.4 Chunk-to-file mapping

The mapping from chunks to files is independent of the term-to-chunk mapping, both in structure and in its construction. The mapping is constructed from two complementing maps: in the chunk-to-file map, each chunk fingerprint points to the IDs of all the files that contain this chunk (and optionally its offsets within each file). The file-to-path map connects each file ID to the file's full pathname. This is equivalent to the mapping between document IDs to user-defined document names in traditional inverted indexes.

We implement the chunk-to-file and file-to-path maps as separate key-value stores, whose keys are the chunk fingerprints and file IDs, respectively. Both maps are created from the metadata in the file recipe. For each file, a <fileID,path> pair is added to the file-to-path map, and a <fingerprint,fileID> pair is added to the chunk-to-file map for each fingerprint in the recipe. If the index support offset lookup, then the <fingerprint,fileID> pair also carries the list of offsets in which the chunk appears. This information can be derived from the file recipe, which contains the size of each chunk.

4.5 Keyword/term lookup

IDEA performs keyword lookup in three phases: (1) a lookup in the term-to-chunk map yields the fingerprints of all the relevant chunks and optionally the term offsets within them, (2) a series of lookups in the chunk-to-file map retrieves the IDs of all the files containing these chunks, and optionally the chunk offsets within them, and (3) a lookup of each file ID in the file-to-path map returns the final list of file names. When requested, the offsets of the terms within the files are derived from the combination of the term and chunk offsets.

Phases (2) and (3) are oblivious to the number of keywords in the original search query. The term-to-chunk map, implemented as an inverted index, returns a set of unique chunks, even in complex search queries that lookup multiple keywords. However, when a term appears in multiple chunks belonging to the same file, some of the files returned from the chunk-to-file map will be redundant. When offsets are not supported or not requested in the query, we collect the results from phase (2) in a set data structure that eliminates duplicate entries, ensuring that each file is searched in the file-to-path map only once.

4.6 Ranking results

As a proof of concept, we extended IDEA to support document ranking with the TF-IDF metric. Recall (from Section 2) that the score of a <document,term> pair is calculated using four values. The number of *files in the system* is a global system value. The number of *words in the file* is calculated during index creation: IDEA sums the number of words in each of its chunks, which are counted by indexing software when the chunks are processed.

The remaining values are calculated during the lookup of the term, as follows. The number of *files containing the term* is the number of files in the query result. Calculating the number of *appearances of the term in the file* is equivalent to counting the offsets of this term within the file. If offsets are not supported, IDEA supports ranking by recording number of appearances instead of offsets. The term-to-chunk map records the number of appearances of the term in each chunk, and the chunk-to-file map records the number of appearances of each chunk in the file. These values are combined for the files in the query result, to return the number of appearances of the term in each file.

Supporting ranking for a query with multiple terms requires calculating the number of appearances of each term in each file, separately. This can be done by maintaining a temporary data structure that collects, for each term, the chunks it appears in. This information can be combined with the number of appearances of each chunk in each file in the query result. IDEA can directly support ranking with any metric that is

based on term occurrences or positions. Other metrics may also be supported, but are outside the scope of this paper.

5 Implementation

We implemented a prototype of IDEA to show how deduplicating storage systems can provide indexing functionality with our approach. Storage systems have direct access to file recipes and physical chunks, and can use an existing engine for indexing. In our implementation, we integrated Apache Lucene [2] into the Destor open-source deduplicating backup system [46]. Destor was built for academic purposes and includes all fundamental deduplication mechanisms and data structures. It supports *backup* and *restore* operations, creating deduplicated backups of entire directories. We used the open-source C++ implementation of Apache Lucene, LucenePlusPlus [9], and the C++ version of Destor [3] that was used for deduplication-aware exhaustive scans [42].

Lucene assigns an internal *document ID* to every document that it processes. Its index consists of two major data structures. The *term-to-doc map* returns, for each term, a list of document IDs it is contained in. The *document store* contains, for each document ID, attributes such as its size and file path, and possibly its content. In IDEA, we realize the term-to-chunk map using Lucene's term-to-doc. The document store realizes the chunk-to-file map: the document representing each chunk contains the list of files this chunk is contained in. We use Berkeley-DB [11] for the file-to-path map. It is implemented with the recno data structure, which is a flat text format optimized for sequential integer keys. Figure 6 illustrates the data structures used by Lucene and by IDEA. By default, IDEA uses an SSD for the data structures which are external to Lucene, as Lucene uses the main memory to cache the parts of the index required for fast access. We evaluate the effect of the SSD below.

The full indexing process in IDEA proceeds as follows. We first scan all the file recipes from Destor and create the list of files containing each chunk using a key-value store, which may spill to disk. Each list is added as a document (which is immutable in Lucene[2]) to the document store, where the document ID is the chunk ID. Then, we read the containers from Destor's on-disk container store to memory. The chunks in the containers are passed to Lucene for indexing in the term-to-document map, with their respective IDs.

IDEA must ensure that its separate maps remain consistent. In other words, all the chunks returned from the term-to-chunk map must be present in the chunk-to-file map, and all the file IDs must map to full file pathnames. Thus, IDEA ensures that the <fileID,path> pair is persisted in the file-to-path map before it uses this ID in the chunk-to-file map. IDEA currently does not support lookups to be issued in parallel with index creation. IDEA records, in the file recipe, whether the file has been indexed or not, and a similar record marks containers

[2]The documents are stored sequentially in the index segments, and are immutable to facilitate efficient direct access to them via the skip-lists.

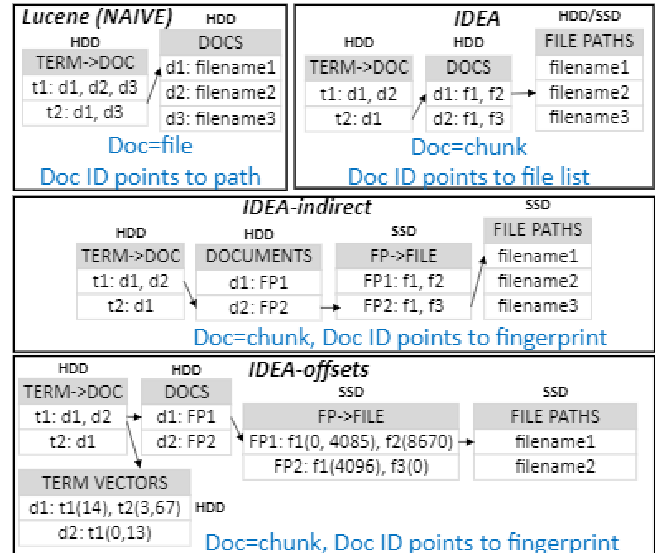


Figure 6: Data structures for Naïve and IDEA, including variants of IDEA that support offsets and ranking.

whose chunks have been indexed. In case of a system crash during index creation, IDEA can be restored to a consistent state by re-processing the unindexed containers and file recipes. Duplicate entries in the term-to-chunk map will be handled by Lucene. In the chunk-to-file map, we can look up each chunk before adding its <chunk,fileID> pair, to find out whether it was already inserted to the map.

We perform the three phases of keyword lookup described in Section 4.5 sequentially. We use the set data structure of C++ (`std::set`, implemented as a red-black tree) to store the unique sets of files returned from chunk-to-file map. Lucene uses a similar structure to return unique file-IDs in the term-to-chunk lookup of multiple keywords. For multiple-keyword lookups, we use Lucene's OR query with all the keywords.

IDEA-indirect. To support additional index functions, we implemented an alternative, more modular, version of IDEA, by adding another level of indirection to its maps. The main advantage of IDEA-indirect is its ability to support *inline indexing*, as part of the system's processing of incoming data: the separation between the document store and the mapping of chunks to files precludes the need to create an immutable list of files containing each chunk.

In IDEA-indirect, the chunk-to-file map is split into two maps: the document store holds, for each document ID (representing a chunk), this chunk's fingerprint. An additional map returns, for each fingerprint, the files containing its chunk. Figure 6 illustrates these data structures. The FP-to-file and file-to-path maps are stored on SSD for faster lookup. We use Berkeley-DB [11] for the additional FP-to-file map. We implement it as a hash-table based multi-map, which supports efficient additions of <chunk,fileID> pairs whenever a new reference to a chunk is identified. In inline indexing, new chunks are passed to Lucene for indexing as soon as they are identified

Dataset	Logical (GB)	Physical (GB)	Recipe (GB)	Files	Chunks (M)
LNX-198	51	11	0.7	4.3M	1.6
LNX-409	181	13	2.4	15.3M	1.7
LNX-662	334	14	4.7	28.2M	1.8
Wiki-4	242	114	0.884	2.3K	10.3
Wiki-8	487	180	1.8	4.7K	14.9
Wiki-12	736	255	2.8	7K	20.1
Wiki-12-1MB	736	259	2.9	686K	20.1
Wiki-24-1MB	1370	478	5.4	1.3M	36.9

Table 1: The datasets used in our experiments

Dictionary	LNX-198		Wiki-12	
	Files	Chunks	Files	Chunks
file-low	1.4	1.3	1.3	1.11
file-med	9.5	3.3	9.35	3.57
file-high	93.7	14.4	95.4	40.1
chunk-low	11.6	1.22	8.25	1.43
chunk-med	28	9.6	16.1	9.36
chunk-high	148	94.8	208.9	94.6

Table 2: Average number of files and chunks per keyword in dictionaries used in our experiments.

as unique (in parallel to step 6 in Figure 1). The <chunk, path> pairs are added as <chunk, fileID> and <fileID, path> to their respective maps after fingerprint calculation, during the creation of the file recipe. Lucene organizes its index in segments, and automatically creates a new segment when new documents are added to an existing index. Segment merging and splitting is controlled by a set of Lucene's internal triggers.

IDEA-offsets. In LucenePlusPlus [9], which is equivalent to Lucene version 3.0.3, term offsets within files are maintained in dedicated data structures called *term vectors*: for each document, the term vector lists the terms in this document, and the offsets each term appears in [3]. We rely on these existing structures to support offset lookups in IDEA-offsets, which extends IDEA-indirect as follows. IDEA-offsets uses the term vectors to record the offsets of terms within the chunks they appear in. It also extends the chunk-to-file map to record, for each file-ID, the list of offsets of the chunk within the file. Figure 6 illustrates these data structures.

IDEA-rank. LucenePlusPlus supports ranking by recording term frequencies within the term-to-doc map. This version of Lucene couples support for ranking with support for proximity search: the term frequency is recorded alongside the list of its positions in the files.[4] This increases the size of the term-to-doc map beyond what is necessary for ranking alone.

We implement IDEA-rank by extending IDEA-indirect to use the frequency records of Lucene. It stores the frequency of the term in each chunk in the term-to-chunk map, and the frequency of the chunk in each file in the FP-to-file map. The number of terms in each file is stored in the file-to-path map, and the global counter of files is maintained as an independent counter. IDEA-rank currently supports a single-term lookup, with multi-term queries deferred to future work.

6 Experimental Setup

Baseline. In addition to IDEA, IDEA-indirect, IDEA-offsets and IDEA-rank, we evaluated a deduplication-oblivious index, **Naïve**, which uses Lucene to index the logical files as documents. To implement Naïve, we extended Destor's restore process: we use it to read all the files in the system in their logical order. Instead of writing the restored files, they are passed to Lucene as documents for indexing. Lookup in Naïve is performed by a simple OR-query lookup. After retrieving the document IDs from the inverted index, Lucene converts them to file names and returns the names as the query result (see Figure 6). We also implemented versions supporting additional functionality, **Naïve-offsets** and **Naïve-rank**.

Datasets. We used two types of datasets for evaluating indexing and lookup times, similar to those used in [42]. The **Linux** datasets contain versions of the Linux kernel source code [7], from version 2.0 to version 5.9. The datasets contain 198, 409, and 662 versions, including all the minor versions, every 10th patch, and every 5th patch, respectively. The **Wikipedia** datasets contain archived versions of the English Wikipedia [15, 16], from January 2017 to March 2018. The datasets contain 4, 8, 12, and 24 consecutive XML dumps, which we split into files of 100MB (at page boundaries). We created two additional datasets from the 12 and 24 versions, with files of 1MB, to evaluate the effect of the number of files on the index performance. The datasets were created with variable-sized chunks (using Rabin fingerprints) with an average size of 8KB. The full details appear in Table 1.

Keyword dictionaries. We created six sets (*dictionaries*) of keywords that vary in the number of chunks and files they appear in. We sorted the terms in Wiki-12 in order of the number of chunks they appear in, and retrieved all the terms that appear in the ranges of 1-2, 9-10, and 90-100 chunks. We then chose 128 random terms from each range, creating the Wiki-chunk-low, Wiki-chunk-med, and Wiki-chunk-high dictionaries. We repeated this process, counting appearances of terms in entire files instead of chunks, to create Wiki-file-low, Wiki-file-med, and Wiki-file-high. We created a similar set of dictionaries from LNX-198 using the same process. The resulting average number of chunks and files containing each term in each dictionary are summarized in Table 2.

Hardware. For our experiments, we used a server running Ubuntu 16.04.7, equipped with 128GB DDR4 RAM and an Intel Xeon Silver 4210 CPU running at 2.40GHz. The backup store for Destor was a Dell 8DN1Y 1TB 2.5" SATA HDD. The maps of all the index alternatives (Naïve as well as the term-chunk map of IDEA and all the maps of IDEA-Direct) were stored on a separate identical HDD. The chunk-to-file and file-to-path maps of IDEA were stored on a Dell T1WH8

[3] Later versions of Lucene also support the embedding of offsets within the term-to-doc map, eliminating the additional data structure.

[4] A position is the term's offset counted in terms, rather than bytes.

240GB 2.5" SSD. We cleared the page cache and restarted Lucene before each indexing and lookup experiment.

7 Evaluation

The goal of our experimental evaluation was to understand how deduplication-aware indexing (IDEA) compares to the traditional deduplication-oblivious indexing (Naïve) in its storage requirements (index size), memory usage, indexing time, and lookup performance. We designed our evaluation to demonstrate how these aspects are affected by the characteristics of the indexed data (deduplication ratio, number and size of files) and of the searched keywords.

Indexing time. Figure 7 shows the offline indexing times of Naïve and IDEA. It shows that deduplication-aware indexing can reduce indexing time compared to Naïve, and that the reduction is proportional to the *deduplication ratio*—the portion of data removed by deduplication. The recipe-processing time is negligible compared to the chunk-processing time in all except very extreme cases (LNX-662 with 28M files).

Recall that the Linux datasets have a very high deduplication ratio (between 78% in LNX-198 and 95% in LNX-662), with many small files. In these datasets, the indexing time of IDEA is shorter than that of Naïve by 76% to 94%. This reduction results from processing each chunk only once, and fetching the chunks sequentially from the underlying HDD. The Wikipedia datasets have lower deduplication ratios (between 53% in Wiki-4 and 65% in Wiki-12 and Wiki-24) and a considerably smaller number of large files. In these datasets, the reduction in indexing time is substantial but smaller: the indexing time of IDEA is shorter than that of Naïve by 49% to 76%.

The results for the Wiki-12 versions further illustrate the effect of the number and the size of the files on the indexing times. When Lucene creates the term-to-file mapping, multiple occurrences of a term in a document are heuristically replaced (in memory) by a single pointer/counter. Thus, as the size of the files decreases (from 100MB in Wiki-12 to 1MB in Wiki-12-1MB), there are fewer replacements and their processing time in Naïve increases. In contrast, the chunk-processing time of IDEA depends on the appearances of terms in chunks, not files, and thus remains similar for all these versions.

Index size. Figure 8 compares the size of the different indexes. In the Linux datasets, IDEA is always smaller than Naïve, and the difference between them increases with the deduplication ratio—for LNX-662, the index size of IDEA is 73% smaller than that of Naïve. The reason is the large number of small files, combined with a very high deduplication ratio: there are more files than chunks in all these datasets (see Table 1). In Naïve, each term points to a large set of files it occurs in, while in IDEA, the files are recorded for each chunk rather than term. The file-to-path map occupies a significant portion of IDEA's index for these datasets. However, it is still considerably smaller (27%-44%) when compared to Naïve.

The Wikipedia datasets have a much lower deduplication ratio than Linux. In this case, the benefit from mapping term to chunks instead of files depends on the size of the files. The index of IDEA is larger than that of Naïve in the datasets with 100MB files (Wiki-4, Wiki-8, Wiki-12), and is smaller in the datasets with 1MB files. The advantage of IDEA compared to Naïve can be seen when comparing the different versions of the Wiki-12 dataset: the size of Naïve grows considerably with the number of files, while the size of IDEA is almost unchanged. The reason is that when the data is split into more files, Naïve must record more files for all the terms included in them. In IDEA, however, this additional information is recorded per chunk, not per term.

The memory requirements of each index are related but not directly proportional to the index size. During startup, Lucene loads an internal data structure called the *term-info-index (Tii)*, which contains statistics regarding each term, including a compressed counter of its frequency in the entire dataset. The size of the Tii is roughly half of the size of the data loaded into memory during Lucene's startup. The Tii in IDEA is smaller than the Tii in Naïve (except in Wiki-4, which has an unusually low deduplication ratio) by 52% to 76%. The peak memory usage of IDEA is proportionately lower than that of Naïve for the Wikipedia datasets. In the Linux datasets, more memory is used for chunk-to-file lookups, and thus its consumption is comparable to that of Naïve.

Lookup times. Figure 9 shows the lookup time for a single keyword from two dictionaries in three representative datasets. Each bar represents an average of four experiments (the standard deviation was at most 0.2%), each with a different keyword, with the latency divided into startup time and lookup times in each map. The results show that the additional lookups due to the indirection in IDEA have a minor effect on the latency. IDEA is faster than Naïve by up to 82%, 47%, and 45% in LNX-198, Wiki-12, and Wiki-12-1MB, respectively.

The advantage of IDEA is the smaller size of its term-to-doc map, which incurs shorter lookup latency. The latency of each step in the lookup process depends on the size of the respective data structure. The startup time, which is dominant when searching for a single keyword, is proportional to the size of the Tii, which is much smaller in IDEA. The lookup times in the different inverted term-indexes is also proportional to their size, due to the logarithmic search complexity in Lucene's skip-lists. The remaining latency is incurred when converting document IDs to names. In the Wikipedia datasets, this specific map of IDEA is larger than that of Naïve by orders of magnitude, but its overall lookup time is still smaller than that of Naïve.

Figure 10 shows the lookup times with increasing numbers of terms from the file-med dictionary. Each bar shows the average time of three independent executions, and the standard deviation was at most 0.09%. As the number of terms increases, the weight of the startup time S_{index} in the overall lookup latency decreases, and the time to convert the document IDs to their names increases. IDEA outperforms Naïve by up to 59%

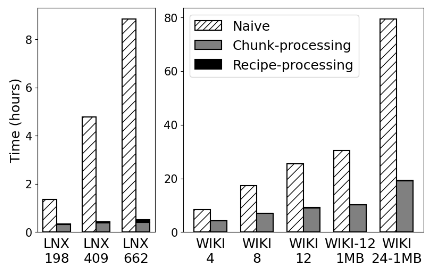


Figure 7: Offline indexing times.

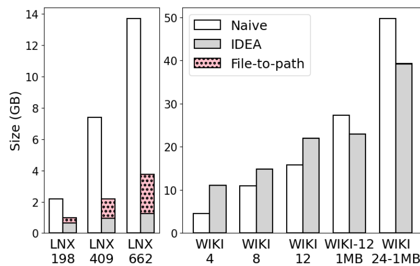


Figure 8: Index sizes.

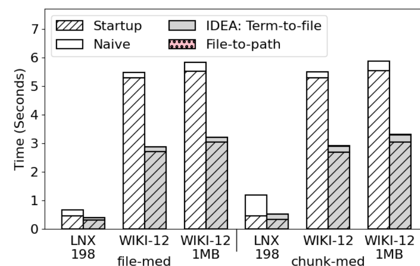


Figure 9: lookup times of a single keyword.

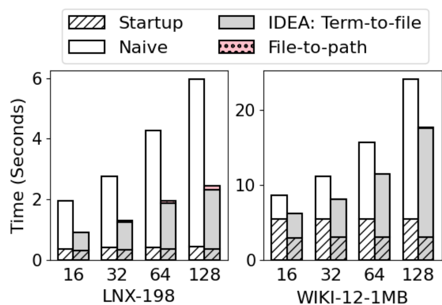


Figure 10: Lookup times with different numbers of keywords, with the file-med dictionary.

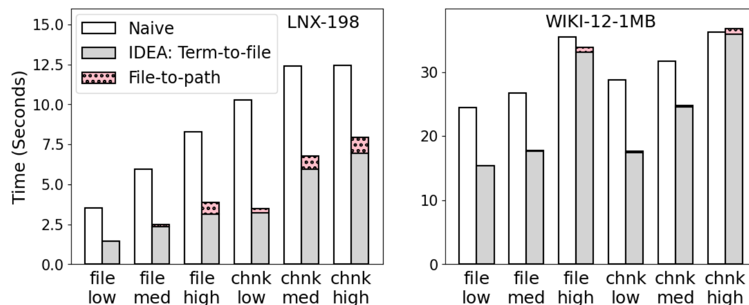


Figure 11: lookup times with 128 keywords of different dictionaries, in the LNX-198 (left) and WIKI-12-1MB (right) datasets.

and 27% on the Linux and Wikipedia datasets, respectively.

Figure 11 compares the effect of different dictionaries on the lookup times. Each bar shows the average time of three independent executions, and the standard deviation was at most 0.09%. The lookup times of Naïve as well as IDEA increased with the number of chunks and files in the query result. The Linux dataset contained more files than chunks, and thus IDEA was faster than Naïve by up to 59% (in the file-low dictionary). The Wikipedia dataset, on the other hand, contained many more chunks than files. As a result, there are considerably more chunks that contain each term, especially in the chunk-high dictionary, for which the lookup time of both indexes was comparable. For all other dictionaries, the lookup time of IDEA was shorter than that of Naïve.

We repeated all the lookup experiments of IDEA without an auxiliary SSD. This means that its file-to-path map was stored on HDD. As expected, this increased the total lookup time, and this increase was proportional to the number of different files in the query result. For example, with a single keyword from the chunk-med dictionary, the increase compared to IDEA with an SSD was 1.1% and 16% in the Wiki-12 and LNX-198 datasets, respectively. The biggest increase was 168% when looking up 128 keywords from the LNX-662 dataset. Nevertheless, the lookup of IDEA was faster than that of Naïve, even without the SSD, in all experiments.

IDEA overheads. We created two datasets to evaluate the worst-case overheads of IDEA in a system without deduplication. LNX-1 and Wiki-1 contain a single version of Linux and Wikipedia, respectively. With almost no deduplication, IDEA has no advantage when compared to deduplication-oblivious indexing, while incurring the overhead of processing a large number of small documents, and looking up terms and chunks in an additional mapping layer. Table 3 lists the characteristics of each dataset, as well as the indexing and lookup times of the different indexes.

Indeed, IDEA is larger than Naïve, due to the larger number of documents in the index: IDEA must record, for each term, all the chunks it appears in, even though many of them point to the same file. The indexing time is similar for both indexes in the Linux dataset that consists of many small files. In the Wikipedia dataset, IDEA cannot optimize index construction by eliminating recurring terms in a document because its documents are small chunks rather than Wikipedia's large files. The lookup times of IDEA are longer for both datasets, due to the reasons discussed in detail above. However, this increase is negligible for Wikipedia and is a modest 10% for Linux.

This experiment emphasizes the tradeoffs of deduplication-aware indexing. Namely, that the additional layer of indirection incurs non-negligible overheads that are masked in systems where the deduplication ratio is sufficiently high. In our future work, we will identify the minimal deduplication ratio for which deduplication-aware indexing is more efficient that the traditional approach, and how this minimum depends on the average file size.

The effect of indirection. IDEA-indirect is the basis for the additional functionality of deduplication-aware indexing (offsets and ranking). To evaluate the effect of the additional layer of indirection (chunk-to-file mapping), we repeated the indexing and lookup experiments with IDEA-indirect. The offline indexing times were within 1% and 6% of those of IDEA for the Wikipedia and Linux datasets, respectively. The size of IDEA-indirect is always larger than that of IDEA: by

Dataset	Dataset size			Indexing time		Index size		Lookup time	
	Logical	Physical	# files / # chunks	Naïve	IDEA	Naïve	IDEA	Naïve	IDEA
WIKI-1	60.4GB	60.3GB	574 / 6M	2.02H	2.25H (11.3%)	2.5GB	6.1GB (152%)	11.67	11.78 (0.1%)
LNX-1	0.91GB	0.86GB	74K / 165K	86 secs	85 secs (N/A)	49MB	60.8MB (24%)	0.47	0.52 (10%)

Table 3: Worst-case overhead of IDEA (in parentheses, with respect to Naïve) for systems without deduplication. The lookup times (in seconds) refer to the average of the file-med and chunk-med dictionaries.

up to 22% and 49% for the Wikipedia and Linux datasets, respectively. Despite this increase, it is still smaller than Naïve in the Linux datasets.

The additional level of indirection also increases the lookup time compared to IDEA. This increase grows with the number of chunks in which the queried keywords appear: in the LNX-198 dataset, the lookup time of 128 keywords with IDEA-indirect was 32% and 23% higher than that of IDEA, in the file-low and chunk-high dictionaries, respectively. In the Wiki-12-1MB dataset and the file-high and chunk-high dictionaries, this increase caused IDEA-indirect to be slower than Naïve. In all other experiments, however, IDEA-indirect was faster than Naïve, despite the additional layer of indirection.

Inline indexing. We compared the inline and offline indexing times of Naïve and IDEA-indirect on two of the datasets, LNX-198 and Wiki-12. Recall that inline indexing is integrated into the deduplication process, referred to as 'backup' in Destor. In this experiment, the original data was read from one HDD and backed-up by Destor on a second HDD. We include the backup time in the results for offline indexing, for a meaningful comparison. Inline indexing is more efficient in terms of memory usage – the chunks are processed while they are still in memory, and do not need to be fetched from the disk. At the same time, the backup process is slowed down by the additional processing.

The results in Figure 12 show that the slowdown of the backup process is detrimental with Naïve indexing: 6.1x and 11.5x in the Linux and Wikipedia datasets, respectively. Indeed, to the best of our knowledge, no deduplicating system currently supports inline indexing. In contrast, IDEA-indirect slows down the backup process by only 1.7x and 4.5x in the Linux and Wikipedia datasets, respectively, thanks to its ability to process only new unique chunks. Although this overhead is not negligible, it presents, for the first time, a realistic opportunity to index deduplicated data inline with writes.

The effect of term offsets. We repeated the indexing and lookup experiments of IDEA with IDEA-offsets. For brevity, we present here only the results of representative datasets and workloads. Figure 13 shows the index size for Naïve-offsets and IDEA-offsets. These sizes are larger than the sizes without offsets, due to the additional information stored in the term vectors. The increase is higher for Naïve than for IDEA: offsets increase the index size by up to 20.9x and 7.1x for Naïve and IDEA, respectively. As a result, the size of IDEA-offset is always smaller than Naïve-offset, even for datasets in which the situation was reversed without offsets (see Figure 8).

The reason for this difference is that the number of offsets stored by Naïve-offset depends on the logical occurrences of each term. This eliminates the "advantage" that Naïve had over IDEA in datasets with large files. In IDEA-offsets, the offsets are recorded only within chunks: their number as well as their values are smaller, occupying less space in the term-vectors. The additional size of the chunk offsets in the FP-to-file is much smaller than that of the term offsets.

Dataset	LNX-198	Wiki-12-1MB
Naïve-rank	10.2GB (332%)	173GB (490%)
IDEA-rank	3.8GB (178%)	80GB (172%)

Table 4: Index sizes with ranking. The number in parentheses is the increase compared to the version without ranking.

IDEA's smaller size also results in faster indexing, shown in Figure 14. Offsets increase the indexing time by up to 51% and 47% for Naïve and IDEA, respectively. The increase is higher for Naïve-offsets due to the larger index that must be persisted on the HDD.

Figure 15 shows the lookup times for 128 words from the different dictionaries for two representative datasets. Lookups with offsets are naturally slower than without them, due to the need to fetch the term vectors for each file in the query result. Comparing the results to those in Figure 11 shows the increase in the lookup time due to the added offsets: the time increased by as much as 33x and 7.5x for Naïve and IDEA, respectively. The increase is higher in Naïve due to different reasons: in LNX-198, it fetches more term vectors from HDD: one for each file the term appears in, rather than one for each chunk. In Wiki-12-1MB, the term vectors are much longer, because each 1MB file contains many more terms than each chunk.

The effect of result ranking. We evaluated the effect of result ranking on two representative datasets, LNX-198 and Wiki-12-1MB. Recall that recording term frequencies in LucenePlusPlus is coupled with the recording of term positions. As a result, the size of both Naïve-rank and IDEA-rank is larger than their versions that do not support ranking. Table 4 shows that this increase is higher for Naïve, similarly to its increased size when offsets are recorded.

The increase in indexing time (shown in Table 5) is milder, and is higher for IDEA than for Naïve. The reason is the additional in-memory processing required for generating the term counters in the file-to-path map of IDEA-rank. Nevertheless, the indexing time of IDEA-rank is still shorter than that of Naïve-rank. Further optimization of the data structures of IDEA-rank is possible and is left for future work.

Figure 16 shows the lookup times with one keyword (averaged over four runs with different keywords) from the file-med

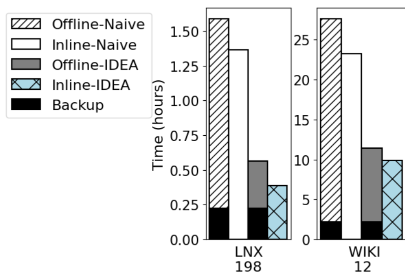


Figure 12: Inline indexing times.

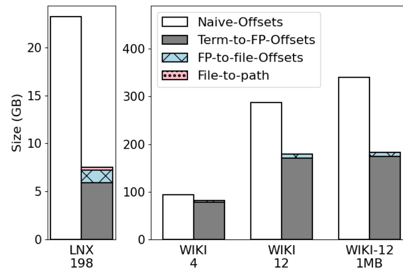


Figure 13: Index sizes with offsets.

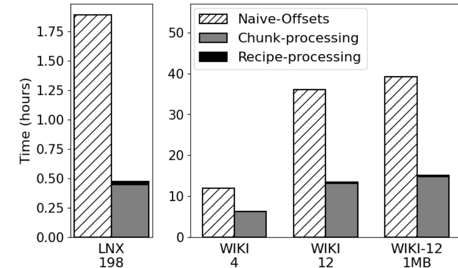


Figure 14: Offline indexing times with offsets.

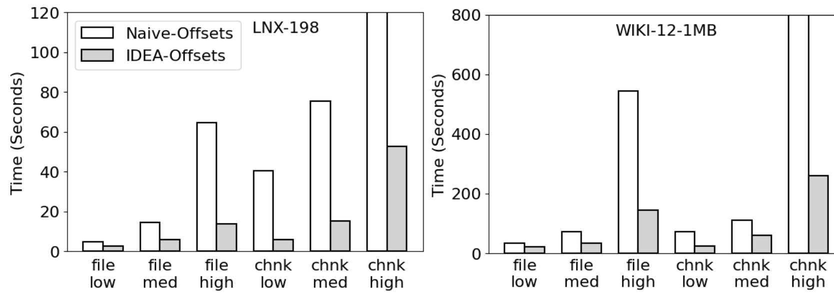


Figure 15: lookup times with 128 keywords of different dictionaries, in the LNX-198 (left) and WIKI-12-1MB (right) datasets, with offsets.

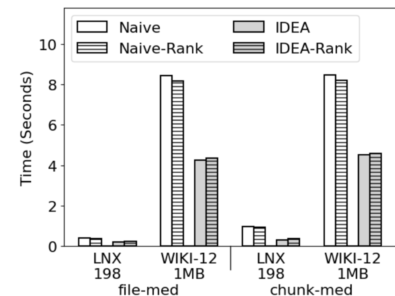


Figure 16: lookup times of 1 keyword of different dictionaries with TF-IDF ranking.

Dataset	LNX-198	Wiki-12-1MB
Naïve-rank	1.57H (15%)	36H (19%)
IDEA-rank	0.4H (18%)	13H (25%)

Table 5: Indexing times with ranking. The number in parentheses is the increase compared to the version without ranking.

and chunk-med dictionaries. The standard deviation was at most 0.4%. The lookup time of IDEA-rank is longer than that of IDEA by up to 25% (except a 50% increase in the "-low" dictionaries). This increase is due to the storage of the term offsets, which increases the term-to-chunk map. Interestingly, in Naïve, this increase in size triggered a segment split in Lucene, which results in slightly faster lookups. This inverse effect causes one anomaly: IDEA-rank is slower than Naïve-rank for words in the file-low dictionary and LNX-198. In all other cases, IDEA is much faster than Naïve.

8 Discussion and Open Challenges

Deduplication-aware indexing opens up several additional venues for improving search performance and applicability. The major advantage of our approach is its generality: it is orthogonal to the specific design details of both deduplication and indexing mechanisms. Deduplication-aware indexing can be integrated into any deduplication system that chunks incoming data streams. The design of the index itself relies on the basic search functionality of Lucene, and could use any other search engine. The chunk-to-file and file-to-path maps can be realized with any data structure or external database. Furthermore, the lookup in the two maps, term-to-chunk and chunk-to-file can be pipelined to reduce some of its overhead: looking up the unique chunks does not require that all of them are identified in advance.

IDEA can support file deletion similarly to existing index designs. The index must maintain the property that it returns all non-deleted files in the storage system that contain the query terms. This can be realized by marking each file as live or deleted, and returning only live files in the query result. A long series of file deletions, can, as in existing index designs, trigger garbage collection and an update of the term-to-chunk and chunk-to-file maps.

While our deduplication-aware indexing approach lends itself to many extensions and improvements, its dependency on white-space aware chunking might prevent it from being applicable when the system receives chunked data and does not perform chunking internally (such as in the case of existing deduplicating storage devices). When terms might be split between chunks, IDEA will have to process each chunk in the context of the chunks adjacent to it in each file.

A similar challenge is presented by files containing compressed text, such as `.pdf` or `.docx`. Their textual content can only be processed after the file is opened by a suitable application or converted by a dedicated tool. Thus, the individual chunks cannot be processed during offline index creation. Both challenges might be addressed by inline indexing, but will require adjusting the indexing process and data structures accordingly. We leave such extensions for future work.

Finally, the overhead of creating and storing an index might be prohibitively high, for deduplicated as well as non-deduplicated data. The choice between indexing and exhaustive search depends on the context of each specific system: its data type and the frequency and type of queries it is expected to serve. IDEA and IDEA-Direct introduce additional design choices between inline and offline indexing, and using HDD or SSD for external map structures.

9 Conclusions

Since most storage in large-scale systems is or will be deduplicated, standard storage functionality can be made more efficient by taking advantage of deduplicated state. In this paper, we presented the first design of a deduplication-aware term index. Our evaluation showed the advantages of this approach, as well as its flexibility in supporting advanced search functions.

Acknowledgments

We thank our shepherd, Vasily Tarasov, for his valuable feedback and suggestions. We thank Amnon Hanuhov and Nadav Elias for their help with the IDEA prototype, and Dafna Sheinvald for insightful discussions. This research was supported, in part, by the Israel Science Foundation (grant No. 807/20).

Artifact Appendix

Abstract

IDEA is a *deduplication-aware keyword index* which addresses the inherent challenges of indexing deduplicated data. We make IDEA's code, scripts, keywords, and configurations publicly available to allow the reproducability of our results and to facilitate further research of deduplication-aware indexing. This section describes the artifact that is made available with the publication of this paper.

Scope

The artifact includes code, tools, and instructions sufficient for reproducing the results presented in the paper. The instructions in the repository can be followed for generating result data equivalent to that used for Figures 7– 16. They can be used to verify the main claims of the paper:

- Indexing is faster with IDEA than with Naïve. This claim holds in setups with and without offsets and ranking, for inline as well as offline indexing.
- The Naïve index is larger than that of IDEA when the deduplication ratio and the number of files are high.
- When offsets are supported, the index size of Naïve is always larger than that of IDEA.
- Index size and indexing time increase when offsets or ranking are supported.
- Lookup times are faster in IDEA than in Naïve, except in several extreme cases.
- Offsets and ranking increase lookup times.

In addition to the reproducible environment, the repository contains instructions for modifying and recompiling the code.

Contents

The main content of the artifact is the source code of IDEA, which is a fork of Destor [46]. The artifact contains source and header files (`src/`), binary libraries of LucenePlusPlus [9] (`libs/`), and compilation scripts for building the IDEA executable. The same executable is also used to run the versions of the Naïve index. The repository also contains resources for reproducing the experiments described in this paper: instructions for downloading and creating the Linux and Wikipedia datasets (`dataset_details/`), the respective keywords (`keywords/`), relevant system configurations (`configs/`), and scripts for running the experiments (`scripts/`).

Hosting

Our artifact is hosted in GitHub and is available here: https://github.com/asaflevi0812/IDEA. The *main* branch is stable for installation and is up-to-date. To modify the code, either open the repository in an IDE in the host machine (e.g., over SSH), or fork the repository and use git to transfer changes between the virtual and host machines.

Requirements

Our prototype is based on Destor, which requires Ubuntu version 16.04. To be consistent with the evaluation setup described in this paper, follow the instructions in GitHub for creating the backup and index on HDD, with IDEA's external data structures on SSD. The required storage capacity depends on the dataset (see Table 1 for details). However, we reproduced our main results also in a server with HDD only, and on a server with Amazon AWS EBS SSD storage.

During the artifact evaluation process, our artifact was evaluated using an *M5.large* instance on AWS with 8 GB DRAM. The image name was *ubuntu-xenial-16.04-amd64-pro-server-20230912* and the storage was configured as GP3 100GB with 3000 IOPS.

References

[1] Amazon OpenSearch™. https://aws.amazon.com/what-is/opensearch/.

[2] Apache Lucene. https://lucene.apache.org/. Accessed: 2022-05-14.

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

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

[5] IBM Watson. https://www.ibm.com/watson.

[6] Indri. http://www.lemurproject.org/indri/.

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

[8] LMDB. http://www.lmdb.tech/doc/.

[9] LucenePlusPlus. https://github.com/luceneplusplus/LucenePlusPlus. Accessed: 2022-12-01.

[10] Meilisearch. https://www.meilisearch.com/.

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

[12] RocksDB. http://rocksdb.org/.

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

[14] TypeSense. https://typesense.org/.

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

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

[17] Microsoft search server 2010 expres. https://www.microsoft.com/en-us/download/details.aspx?id=18914, 2019.

[18] Fast and reliable schema-agnostic log analytics platform. https://www.uber.com/en-CA/blog/logging/, 2021.

[19] Commvault documentation pdf: Protect. access. comply. share. https://documentation.commvault.com/commvault/index.html, 2022.

[20] Deduplication: btrfs wiki. https://btrfs.wiki.kernel.org/index.php/Deduplication, 2022.

[21] Dell EMC Data Protection Search 19.6.1 deployment and administration guide. https://www.dell.com/support/home/en-il/product-support/product/data-protection-search/docs, 2022.

[22] Reducing logging cost by two orders of magnitude using CLP. https://www.uber.com/en-US/blog/reducing-logging-cost-by-two-orders-of-magnitude-using-clp, 2022.

[23] Searching 1.5tb/sec: Systems engineering before algorithms. https://www.dataset.com/blog/systems-engineering-before-algorithms/, 2022.

[24] Solaris ZFS administration guide: The dedup property. https://docs.oracle.com/cd/E19120-01/open.solaris/817-2271/gjhav/index.html, 2022.

[25] Veeam backup & replication 11: User guide for VMware vSphere. https://helpcenter.veeam.com/docs/backup/vsphere/overview.html?ver=110, 2022.

[26] Efficient search in netapp data storage solutions. https://intrafind.com/en/blog/efficient-search-in-netapp-data-storage-solutions, 2023.

[27] Windows search overview. https://learn.microsoft.com/en-us/windows/win32/search/-search-3x-wds-overview, 2024.

[28] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? Redesigning protection storage for modern workloads. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[29] Alexandr Andoni. Nearest neighbor search: the old, the new, and the impossible. Ph.d. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2009.

[30] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, 2009.

[31] Renata Borovica-Gajić, Raja Appuswamy, and Anastasia Ailamaki. Cheap data analytics using cold storage devices. *Proceedings of the VLDB Endowment*, 9(12):1029–1040, 2016.

[32] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[33] Zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[34] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[35] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI 04)*, 2004.

[37] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[38] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *25th International Conference on Large Installation System Administration (LISA 11)*, 2011.

[39] 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.

[40] 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.

[41] 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.

[42] Nadav Elias, Philip Shilane, Sarai Sheinvald, and Gala Yadgar. DedupSearch: Two-Phase deduplication aware keyword search. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[43] EMC Corporation. *Introduction To The EMC XtremIO Storage Array (Ver. 4.0)*, rev. 08 edition, April 2015.

[44] Jingxin Feng and Jiri Schindler. A deduplication study for host-side caches in virtualized data center environments. In *29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, 2013.

[45] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[46] 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.

[47] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. O'Reilly Media, Inc., USA, 2015.

[48] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International Conference on Machine Learning*, pages 3929–3938. PMLR, 2020.

[49] 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.

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

[51] Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The what, the from, and the to: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[52] Naama Kraus, David Carmel, and Idit Keidar. Fishing in the stream: similarity search over endless data. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 964–969. IEEE, 2017.

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

[54] Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. Pre-training via paraphrasing. *Advances in Neural Information Processing Systems*, 33:18470–18481, 2020.

[55] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[56] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[57] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[58] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

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

[60] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems (TOIS)*, 14(4):349–379, 1996.

[61] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-*

Throughput Sequencing. Cambridge University Press, 2015.

[62] 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.

[63] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications (HPCC 11)*, 2011.

[64] Enno Ohlebusch and Giuseppe Ottaviano. 3.16 bioinformatics algorithms: Sequence analysis, genome rearrangements, and phylogenetic reconstruction. *Indexes and Computation over Compressed Structured Data*, page 33, 2013.

[65] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, Aug 2015.

[66] Juan Ramos. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48, 2003.

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

[68] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '02, 2002.

[69] 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.

[70] Fabrizio Silvestri. *Mining Query Logs: Turning Search Usage Data into Knowledge*, volume 4. Now Publishers Inc., Hanover, MA, USA, Jan 2010.

[71] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008.

[72] 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.

[73] Ja-Hwung Su, Yu-Ting Huang, Hsin-Ho Yeh, and Vincent S Tseng. Effective content-based video retrieval using pattern-indexing and matching techniques. *Expert Systems with Applications*, 37(7):5068–5085, 2010.

[74] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6:1930–1941, 09 2013.

[75] Ke Tao, Fabian Abel, Claudia Hauff, and Geert-Jan Houben. Twinder: A search engine for twitter streams. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering*, pages 153–168, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[76] 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.* IEEE, 2003.

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

[78] Ian H Witten, Ian H Witten, Alistair Moffat, Timothy C Bell, Timothy C Bell, Ed Fox, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[79] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[80] 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.