Similarity-based Deduplication for Databases

Lianghong Xu[⋆], Andrew Pavlo[⋆], Sudipta Sengupta[†], Gregory R. Ganger[⋆]
[⋆]Carnegie Mellon University, [†]Microsoft Research

CMU-PDL-16-101
April 2016

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

dDedup is a similarity-based deduplication scheme for on-line database management systems (DBMSs). Beyond block-level compression of individual database pages or operation log (oplog) messages, as used in today's DBMSs, dDedup uses byte-level delta encoding of individual records within the database to achieve greater savings. dDedup's single-pass encoding method can be integrated into the storage and logging components of a DBMS to provide two benefits: (1) reduced size of data stored on disk beyond what traditional compression schemes provide, and (2) reduced amount of data transmitted over the network for replication services. To evaluate our work, we implemented dDedup in a distributed NoSQL DBMS and analyzed its properties using four real datasets. Our results show that dDedup achieves up to 37× reduction in the storage size and replication traffic of the database on its own and up to 61× reduction when paired with the DBMS's block-level compression. dDedup provides both benefits with negligible effect on DBMS throughput or client latency (average and tail).

Acknowledgements: We thank the members and companies of the PDL Consortium (including Avago, Citadel, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Two Sigma, Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and MongoDB Incorporated. Experiments were enabled by generous hardware donations from Intel, NetApp, and APC.

Keywords: Deduplication, Databases, Delta Encoding

1 Introduction

The rate of data growth outpaces the decline of hardware costs. Database compression is one solution to this problem. For database storage, in addition to space saving, compression helps reduce the number of disk I/Os and improve performance, because queried data fits in fewer pages. For distributed databases replicated across geographical regions, there is also a strong need to reduce the amount of data transfer used to keep replicas in sync.

The most widely used approach for data reduction in DBMSs is block-level compression [29, 36, 44, 41, 3, 16]. Although this method is simple and effective, it fails to address redundancy across blocks and therefore leaves significant room for improvement for many applications (e.g., due to app-level versioning in wikis or partial record copying in message boards). Deduplication (dedup) has become popular in backup systems for eliminating duplicate content across an entire data corpus, often achieving much higher compression ratios. The backup stream is divided into chunks and a collision-resistant hash (e.g., SHA-1) is used as each chunk's identity. The dedup system maintains a global index of all hashes and uses it to detect duplicates. Dedup works well for both primary and backup storage data sets that are comprised of large files that are rarely modified (and if they are, the changes are sparse).

Unfortunately, traditional chunk-based dedup schemes are unsuitable for operational DBMSs, where many update queries modify a single record. The duplicate data in records is too fine-grained unless the system uses small chunk sizes. But, relatively large chunk sizes (e.g., 4–8 KB) are the norm to avoid huge in-memory indices and large numbers of disk reads.

This paper presents **dDedup**, a lightweight scheme for on-line database systems that uses *similarity-based deduplication* [62] to compress individual records. Instead of indexing every chunk hash, dDedup samples a small subset of chunk hashes for each new database record and then uses this sample to identify a similar record in the database. It then uses byte-level delta compression on the two records to reduce both online storage used and remote replication bandwidth. dDedup provides higher compression ratios with lower memory overhead than chunk-based dedup and combines well with block-level compression, as illustrated in Fig. 1.

We implemented dDedup in the MongoDB DBMS [5], and we evaluate its efficacy using four real-world datasets. Our results show that it achieves upto 37× reduction (61× when combined with block-level compression) in storage size and replication traffic, significantly outperforming chunk-based dedup, while imposing negligible impact on the DBMS's runtime performance.

dDedup introduces and combines several novel techniques in order to achieve such efficiency, in addition to borrowing from the recent sDedup system [62] its sampling-based and cache-aware approaches to selecting source records. It uses novel *two-way encoding* to efficiently transfer encoded new records (forward encoding) to remote replicas, while storing unencoded new records with encoded forms of selected source records (backward encoding). As a result, no decode is required for the common case of accessing the most recent record in an encoding chain (e.g., the latest Wikipedia version). To avoid performance overhead from updating source records, dDedup introduces a *lossy write-back delta cache* tuned to maximize compression ratio while avoiding I/O contention. dDedup also uses a new technique called *hop encoding* to minimize the worst-case number of decode steps required to access a specific record in a long encoding chain. Finally, dDedup adaptively skips dedup effort for databases and records where little savings are expected.

This paper makes the following key contributions: First, to our knowledge, it describes the first dedup system for operational DBMSs that reduces both database storage and replication bandwidth usage. It is also the first database storage dedup system that uses similarity-based dedup. Second, it introduces several novel techniques that are critical to achieving acceptable dedup efficiency, enabling use for online database storage. Third, it describes and evaluates a full implementation of the system in a distributed NoSQL DBMS, using four real-world datasets, quantifying the efficacy of dDedup's approach.

The rest of this paper is organized as follows. Section 2 motivates use of similarity-based dedup for

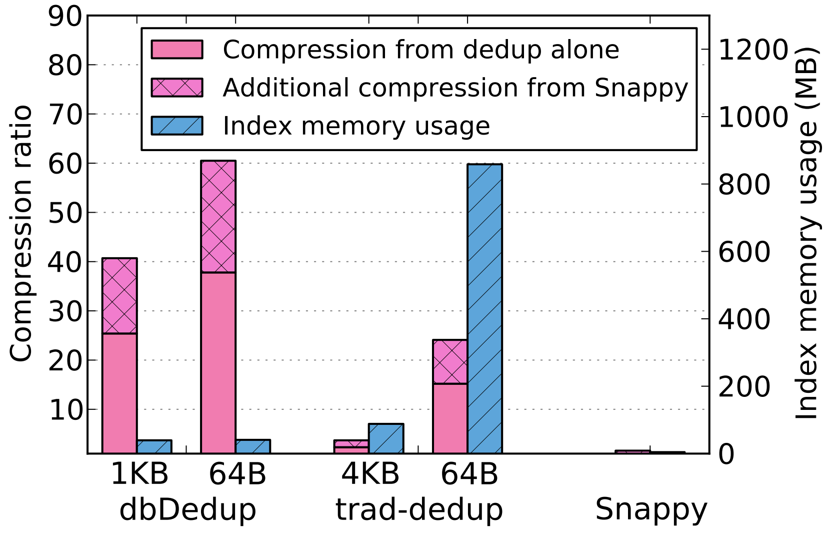


Figure 1: Compression ratio and index memory usage for Wikipedia data stored in five MongoDB configurations: with dDedup (1KB chunk size and 64B), with traditional dedup (4KB and 64B), and with Snappy (block-level compression). dDedup provides higher compression ratio and lower index memory overhead than traditional dedup. Snappy provides the same 1.6× compression for the post-dedup data or the original data.

database applications and categorizes dDedup relative to other dedup systems. Section 3 describes dDedup's dedup workflow and mechanisms. Section 4 details dDedup's implementation, including its integration into the storage and replication frameworks of a DBMS. Section 5 evaluates dDedup, and Section 6 discusses additional related work.

2 Background and Motivation

Deduplication consists of identifying and removing duplicate content across a data corpus. This section motivates its potential value in DBMSs, explains the two primary categories (exact match and similarity-based) of dedup approaches and why similarity-based is a better fit for dedup in DBMSs, and puts dDedup into context by categorizing previous dedup systems.

2.1 Why Dedup for Database Applications?

While dedup is widely used in file systems, it has not been fully explored in databases—most data reduction in DBMSs is based on block-level compression of individual database pages. The primary reason is that database records are usually small compared to typical dedup chunk sizes (4–8 KB), so applying traditional chunk-based dedup would not yield sufficient benefits.

We observe, however, that many database applications could benefit from dedup due to resemblance between un-collocated records whose relationship is not known to the underlying DBMSs. In addition, we find that the benefits from dedup are complementary to those of compression—combining deduplication and compression yields greater data reduction than either alone.

For many applications, a major source of duplicate data is application-level versioning of records. While multi-version concurrency control (MVCC) DBMSs maintain historical versions to support concurrent transactions, they typically clean up older versions once they are no longer visible to any active transaction. As a result, few applications take advantage of versioning support provided by the DBMS to perform "time-travel queries”. Instead, most applications implement versioning on their own when necessary. A common feature of these applications is that different revisions of one data item are written to the DBMS as completely unrelated records, leading to considerable redundancy that is not captured by simple page compression. Examples of such applications include websites powered by WordPress, which comprise 25% of the entire web [12], as well as collaborative wiki platforms such as Wikipedia [14] and Baidu Baike [1].

Another source of duplication in database applications is inclusion relationships between records. For instance, an email reply or forwarding usually includes the content of the previous message in its message body. Another example is on-line message boards, where users often quote each other’s comments in their posts. Like versioning, this copying is an artifact of the application that cannot be easily exposed to the underlying DBMS. As a result, effective redundancy removal also requires a dedup technique that identifies and eliminates redundancies across the entire data corpus.

It is important to note that there are also many database applications that would not benefit from dedup. For example, some do not have enough inherent redundancy, and thus the overhead of finding opportunities to remove redundant data is not worth it. Typical examples include most OLTP workloads, where many records fit into one database page and most redundancies among fields can be eliminated by block-level compression schemes. For applications that do not benefit, dDedup automatically disables dedup functionalities to reduce its impact on system performance.

2.2 Similarity-based Dedup vs. Exact Dedup

Dedup approaches can be broadly divided into two categories. The first and most common (“exact dedup”) looks for exact matches on the unit of deduplication (e.g., chunk) [64, 39, 27, 33, 34]. The second (“similarity-based dedup”) looks for similar units (chunks or files) and applies delta compression to them [58, 50, 22]. For those database applications that do benefit from dedup, we find that similarity-based dedup outperforms chunk-based dedup in terms of compression ratio and memory usage, though it can involve extra I/O and computation overhead. This section briefly describes chunk-based dedup, why it does not work well for DBMSs, and why similarity-based dedup does. Section 3 details dDedup’s workflow and its techniques for mitigating the potential overheads.

A traditional file dedup scheme based on exact matches of data chunks (“chunk-based dedup”) [42, 46, 64] works as follows. An incoming file (corresponding to a new record in the context of DBMS) is first divided into chunks using Rabin-fingerprinting [47]; Rabin hashes are calculated for each sliding window on the data stream, and a chunk boundary is declared if the lower bits of the hash value match a pre-defined pattern. The average chunk size can be controlled by the number of bits used in the pattern. Generally, a match pattern of n bits leads to an average chunk size of 2^n B. For each chunk, the system calculates a unique identifier using a collision-resistant hash (e.g., SHA-1). It then checks a global index to see whether it has seen this hash before. If a match is found, then the chunk is declared a duplicate. Otherwise, the chunk is considered unique and is added to the index and the underlying data store.

While chunk-based dedup generally works well for backup storage workloads, it is rarely suitable for database workloads. From our observations, duplicate regions for database workloads are usually small (on the order of 10’s to 100’s of bytes) and spread out within a record. At this small size, chunk-based dedup with a typical chunk size on the order of KBs is unable to identify many duplicate chunks. Reducing chunk size to match up with duplication length may improve the system’s compression ratio, but the chunk tracking index becomes excessively large and negates any performance benefits gained by I/O reduction.

In contrast, dDedup’s similarity-based dedup identifies one similar record from the database corpus and performs delta compression between the new record and the similar one. As shown in Fig. 2, dDedup’s byte-level delta compression is able to identify much more fine-grained duplicates and thus provide greater compression ratio than chunk-based dedup.

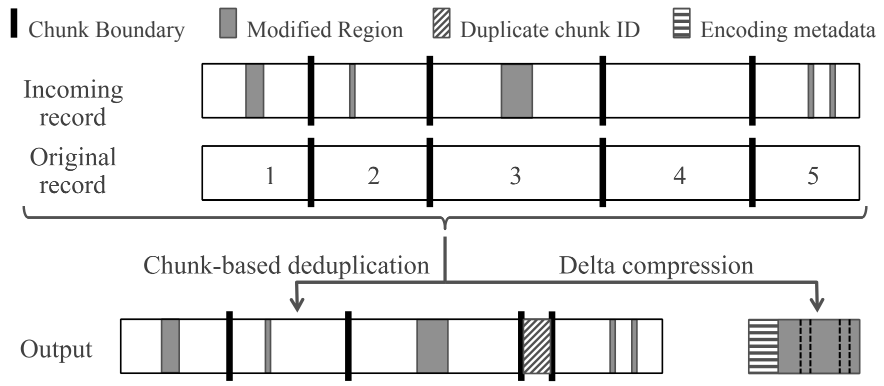


Figure 2: Comparison between chunk-based deduplication and similarity-based deduplication using delta compression for typical database workloads with small and dispersed modifications.

2.3 Categorizing Dedup Systems

Table 1 illustrates one view of how dDedup relates to other systems using dedup, based on two axes: dedup approach (exact match vs. similarity-based) and dedup target (primary storage vs. secondary/backup data). To our knowledge, dDedup is the first similarity-based dedup system for primary data storage, as well as being the first dedup system for on-line DBMSs addressing both primary storage and secondary data (the oplog).

Much prior work in data deduplication [64, 39, 20, 50, 51] was done in the context of backup data (as opposed to primary storage) where dedup does not need to keep up with primary data ingestion nor does it need to run on the primary (data-serving) node. Moreover, such backup workloads often run in appliances on premium hardware. dDedup, being in the context of operational DBMSs, must run on primary data-serving nodes on commodity hardware and be frugal in its usage of CPU, memory, and I/O resources.

There has been recent interest in primary data dedup on the primary (data-serving) server but the solutions are mostly at the storage layer (and not at the data management layer, as in our work). In such systems, depending on the implementation, dedup can happen either inline with new data (Sun's ZFS [17], Linux SDFS [4], iDedup [52]) or in the background as post-processing on the stored data (Windows Server 2012 [34]), or provide both options (NetApp [19], Ocarina [7], Permabit [8]).

Systems in the lower middle column use a combination of exact and similarity-based dedup techniques at different granularities, but are in essence chunk-based dedup systems because they store hashes for every chunk. To the best of our knowledge, dDedup is the first similarity-based dedup system for primary storage workloads that achieves data reduction on storage and network bandwidth requirement at the same time. This is because byte-level delta compression is traditionally considered expensive for on-line databases, due to the extra I/O and computation overhead relative to hash comparisons. As a result, previous systems either completely avoid it or use it when disk I/O is not a major concern. For example, SIDC [50] and sDedup [62] use delta compression for network-level deduplication of replication streams; SDS [20] applies delta compression to large 16 MB chunks in backup streams retrieved by sequential disk reads. While dDedup takes advantage of delta compression to achieve superior compression ratio, it uses a number of techniques to reduce the overhead involved, making it a practical dedup engine for on-line DBMSs.

3 dDedup Design

This section describes dDedup's dedup workflow, encoding techniques, I/O overhead mitigation mechanisms, and approaches to avoiding wasted effort on low-benefit dedup actions.

	Exact Dedup		**Similarity-based Dedup**
Primary	iDedup [52]		
	ZFS [17]		
	SDFS [4]		
	Windows server 2012 [15]		***dDedup***
	NetApp ASIS [19]		
	Ocarina [7]		
	Permabit [8]		
Secondary	DDFS [64]	Extreme binning [22]	
	Venti [46]	Sparse Indexing [39]	
	ChunkStash [30]	Silo [61]	SDS [20]
	DEDE [27]	SIDC [50]	sDedup [62]
	HydraStor [32]	DeepStore [63]	

Table 1: Categorization of related work

3.1 Deduplication Workflow

dDedup uses similarity-based dedup to achieve good compression ratio and low memory usage simultaneously. Fig. 3 shows the dedup encode workflow used when preparing updated record data for local storage and remote replication. During insert or update queries, new records are written to the local oplog, and dDedup encodes them in the background, off the critical path. Four key steps are (1) extracting similarity features from a new record, (2) looking in the deduplication index to find a list of candidate similar records in the database corpus, (3) selecting one best record from the candidates, and (4) performing delta compression between the new and the similar record to compute encoded forms for local storage and replica synchronization.

3.1.1 Feature Extraction

As a first step in finding similar records in the database, dDedup extracts similarity features from the new record using a content-dependent approach. dDedup divides the new record into several variable-sized data chunks using the Rabin Fingerprinting algorithm [47] that is widely used in many chunk-based dedup systems. Unlike these systems that index a collision-resistant hash (e.g., SHA-1) for every unique chunk, dDedup calculates a (weaker, but computationally cheaper) MurmurHash [6] for each chunk and only indexes a representative subset of the chunk hashes. dDedup adapts a technique called *consistent sampling* [45] to select representative chunk hashes, which provides better similarity characterization than random sampling. It sorts the hash values in a consistent way (e.g., by magnitude from high to low), and chooses the top-K[1] hashes as the *similarity sketch* for the record. Each chunk hash in the sketch is called a *feature*—if two records have one or more common features, they are considered to be similar.

By indexing only the sampled chunk hashes, dDedup bounds the memory overhead of its dedup index to be at most K index entries per record. This important property allows dDedup to use small chunk sizes for better similarity detection while not consuming excessive RAM like in chunk-based dedup. Moreover, because dDedup does not rely on exact match of chunk hashes for deduplication, it is more tolerant of hash collisions. This is why it can use the MurmurHash algorithm instead of SHA-1 to reduce the computation overhead in chunk hash calculation. While this may lead to a slight decease in compression rate due to more false positives, using a weaker hash does not impact correctness since dDedup performs delta compression in the final step.

[1]We find $K = 8$ strikes a reasonable trade-off between compression ratio and memory usage, and we use it as a default value for all experiments unless otherwise noted.

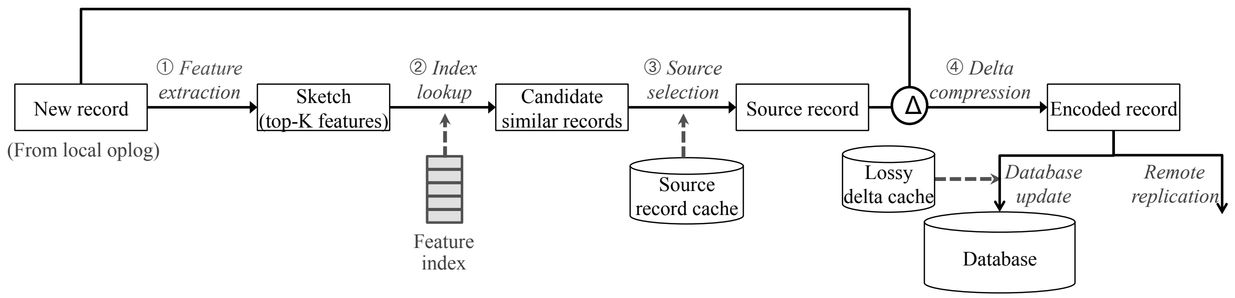


Figure 3: dDedup Workflow – (1) Feature Extraction, (2) Index Lookup, (3) Source Selection, and (4) Delta Compression.

3.1.2 Index Lookup

For each extracted feature, dDedup finds existing records that share that feature with the new record. Since dDedup is an online dedup system, it is imperative that this index lookup process is fast and efficient. dDedup achieves this by building an in-memory feature index that uses a variant of Cuckoo hashing [43, 30] to map features to records. This approach uses multiple hashing functions that map a key to multiple candidate slots, which increases the table's load factor while bounding lookup time to a constant. In the feature index, each entry is comprised of a 2-byte key that is a compact checksum of the feature and a 4-byte value that is a pointer to the database location of the corresponding record.

On feature lookup, dDedup first calculates a hash of the feature value using one of the Cuckoo hashing functions that maps to a candidate slot containing multiple index entries (buckets). It then iterates over the buckets, compares their checksums with the given feature, and adds any matched records to the list of similar records. This process repeats with the other hashing functions until it finds an empty bucket indicating the end of search. dDedup then inserts the feature and a reference to the new record to the empty bucket for future lookup. Finally, dDedup combines the lookup results for all top-K features and generates a list of existing similar records as input for the next step. To further reduce CPU and memory usage, dDedup limits the maximum number of similar records that examines for each feature. Once the threshold is reached, the lookup process terminates and the entry containing the least-recently-used (LRU) record is evicted from the feature index.

3.1.3 Source Selection

The index lookup results may contain multiple candidate similar records, yet dDedup only chooses one of them to delta compress the new record in order to minimize the overhead involved. While most previous similarity selection algorithms make such decisions purely based on the similarity metrics of the inputs, dDedup adds consideration of system performance, giving preference to candidate records that are present in the source record cache (see Section 3.3). We refer to this selection technique as *cache-aware selection*. Specifically, dDedup first assigns an initial score for each candidate similar record based on the number of features it has in common with the new record. Then, dDedup increases that score by a reward if the candidate record already resides in the cache. The candidate with the highest score is then selected as the input for delta compression. While cache-aware selection may end up choosing a record that is sub-optimal in terms of similarity, we find it greatly reduces the I/O overhead to fetch source records from the database. We evaluate the effectiveness of cache-aware selection and its sensitivity to the reward score in Section 5.4.

3.1.4 Delta Compression

The last step in dDedup workflow is to perform delta compression between the new record and the selected similar record. We describe the details of the encoding techniques in Section 3.2 and the compression algorithms in Section 4.2.

3.2 Encoding Techniques

dDedup uses a *two-way encoding* technique that reduces both remote replication bandwidth and database storage, while optimizing for common case queries. It also uses *hop encoding* to reduce worst-case source retrievals for reading encoded records.

3.2.1 Two-way Encoding

After a candidate record is selected from the data corpus, dDedup generates the byte-level difference between the candidate and the new record in reverse directions, using a technique that we call *two-way encoding*. In the first pass, dDedup performs *forward encoding*, as shown in Fig. 4a, which uses the older (i.e., the selected candidate) record as the source and the new record as the target. After the encoding, the source remains in its original form, while the target is encoded as a reference to the source plus the delta from the source to the target. dDedup sends the encoded data, instead of the original new record, to remote replicas. Using forward-encoding for network-level deduplication is a natural design choice, because it allows the replicas to easily decode the target record using the locally stored source record.

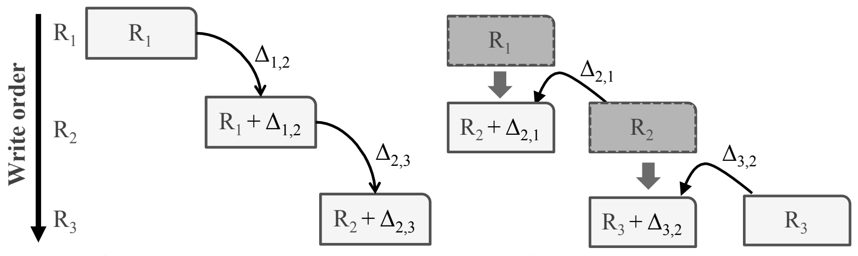
dDedup could simply use the same encoded form for local database storage. Doing so, however, would lead to significant performance degradation for read queries to the newest record in the encoding chain, which we observe to be the common case with app-level versioning and inclusions. Because the intermediate records in a forward chain are all stored in the encoded form using the previous one as the source, decoding the latest record requires retrieving all the deltas along the chain, all the way back to the first record, which is stored unencoded.

Instead, dDedup uses *backward encoding* (Fig. 4b) to optimize for read queries to recent records. That is, for local storage, dDedup performs delta compression in the reverse temporal order, using the new record as the source and the similar candidate record as the target. As a result, the most recent record in an encoding chain is always stored unencoded. While backward encoding is optimized for reads, it amplifies the number of writes, since the older records selected as candidates need to be updated to the encoded form. To mitigate the write amplification, dDedup caches backward-encoded records to be written back to the database and delays the updates until system I/O is relatively idle, which we will discuss in more detail in Section 3.3.

While dDedup performs delta encoding between new and candidate records in two directions, it only incurs the computation overhead of one encoding pass. Specifically, dDedup first generates the forward-encoded data and then uses it to efficiently compute the backward-encoded data at memory speed. We call this process *re-encoding* and detail the algorithm in Section 4.2.

3.2.2 Hop Encoding

Using backward encoding minimizes the decoding overhead for reading recent records, but it may incur excessive source retrieval time for occasional queries to older records (e.g., a specific version of a Wikipedia article). Some previous work on delta encoding [26, 40] uses a technique called *version jumping* to cope with this problem. The idea is to divide the encoding chain into fixed-size clusters, where the last record in each cluster, termed *reference version*, is stored in its original form and the other records are stored as backward-encoded deltas. Doing so bounds the worst-case retrieval times to the cluster size, but at the cost of lower compression ratio, because the reference versions are not compressed. As the encoding cluster size



(a) Forward encoding **(b)** Backward encoding

Figure 4: Illustration of two-way encoding.

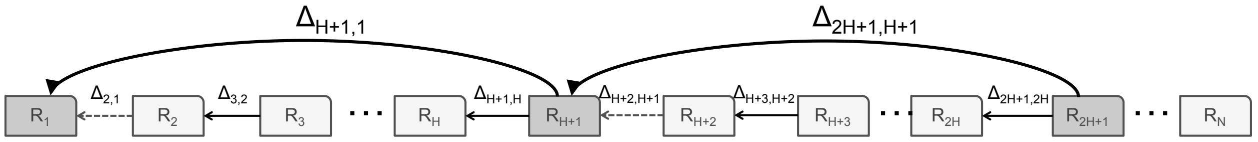


Figure 5: An encoding chain of N records with a hop distance of H. Shaded records (R_1, R_{H+1}, etc.) are hop bases.

decreases, the compression loss can increase significantly, especially when the the size of encoded data is much smaller than the original records.

dDedup uses a new technique we call *hop encoding* that provides higher compression ratio while achieving reasonable worst-case retrieval times. As illustrated in Fig. 5, extra deltas are computed between particular records and others some distance back in the chain. We call these records *hop bases* and the interval between them *hop distance*. Decoding a record involves first tracing back to the nearest hop base and then following the encoding chain starting with it. Assuming a chain of N records and a hop distance of H, the worst-case retrieval times is $\frac{N}{H} + H$, as compared to H in version jumping. Because hop bases are stored in encoded form, the compression ratio using hop encoding is close to that of standard backward encoding. We evaluate the comparison between hop encoding and version jumping in Section 5.

3.3 Caching Mechanisms

Most chunk-based dedup systems focus on optimizing RAM usage to increase the amount of the dedup index that can reside in memory. For these systems, the major system performance bottleneck is the I/O overhead to access the on-disk dedup index, rather than to access the actual data content. In contrast, as a similarity-based dedup system for delta-encoded databases, dDedup's challenge is extra disk I/O involved with reading and updating source records. To minimize this overhead, dDedup uses two specialized caches: a source record cache that reduces the number of database reads during encode and a lossy write-back delta cache that mitigates write amplification caused by backward encoding.

3.3.1 Source Record Cache

A key challenge in delta-encoded storage is the I/O overhead to read the base data from the disk as input for delta compression or decompression. Likewise, in dDedup, reading a selected similar record can involve an extra disk read, contending with client query processing and other database activities.

dDedup uses a small yet effective record cache to avoid most disk reads for similar records. The design of the record cache exploits the high degree of temporal locality in record updates of workloads that dedup well. For instance, updates to a Wikipedia article, forum posts to a specific topic, or email exchanges in the same thread usually occur within a short time frame. So, the probability of finding a recent similar record in the cache is high, even with a relatively small cache size. Another key observation is that the updates are usually incremental (based on the immediate previous update), meaning that two records tend to be more similar if they are closer in creation time. Based on these observations, dDedup only keeps the latest record of an encoding chain in the cache. When a new record arrives, if dDedup identifies a similar record in the cache (which is the normal case due to the cache-aware selection technique discussed in Section 3.1), it replaces the existing record with the new one. Otherwise, it simply adds the new record to the cache, and evicts the oldest record in an LRU manner if the cache becomes full.

3.3.2 Lossy Write-back Delta Cache

As mentioned in Section 3.2, backward encoding optimizes for read queries, but introduces some write amplification—the similar source record needs to be updated to its encoded form. For heavy insertion bursts, this could significantly increase the number of disk writes, leading to visible performance degradation.

dDedup uses a novel *lossy write-back cache* to mitigate this problem. On record insertion, dDedup writes the new record to the database in its original form, but it stores the backward-encoded similar record in a cache and postpones the storage update until system I/O is relatively idle. The idleness metric can vary but we use the I/O queue length as an indication of idleness in our experiments. The term "lossy" means that if the cache becomes full, an entry can be safely evicted without writing the delta back to the database. Because the database storage is backward encoded, not applying the update simply leaves the corresponding record in its unencoded form, which reduces the compression ratio but does not affect correctness.

To minimize the compression loss, dDedup sorts deltas in the cache by the absolute amount of space saving they contribute. When system I/O becomes idle, more valuable deltas are written out first. When the cache grows to a threshold before the system gets idle enough, the entry with the least contribution to data reduction is evicted. By prioritizing the update and eviction orders, dDedup more effectively reaps the compression benefits from cached deltas.

3.4 Avoiding Unproductive Dedup Work

dDedup uses two approaches to avoid applying dedup effort with low likelihood of yielding significant benefit. First, a dedup governor monitors the runtime compression ratio and automatically disables deduplication for databases that do not benefit enough. Second, a size-based filter adaptively skips dedup for smaller records that contribute little to overall compression ratio.

3.4.1 Automatic Deduplication Governor

Database applications exhibit diverse dedup characteristics. For those that do not benefit much, dDedup automatically turns off dedup to avoid wasting resources. In our experience, most duplication exists within the scope of a single database, that is, deduplicating multiple different databases usually yields little marginal benefits as compared to deduplicating them individually. Therefore, dDedup partitions its in-memory dedup index by database and internally tracks the compression ratio for each. If the compression rate for a database stays below a certain threshold (e.g., $1.1\times$) for a long enough period (e.g., 100k record insertions), the dedup governor disables dedup for it and deletes its corresponding index partition. Future records belonging to that database are processed as normal, bypassing the deduplication engine, while already encoded data remains intact. dDedup does not reactivate a database for which dedup is already disabled, because we do not notice dramatic change in compression ratio over time for any particular workload, which we believe is the norm.

3.4.2 Adaptive Size-based Filter

In our observation of several real-world database datasets (see Section 5.1), we find that most dedup savings come from a small fraction of the records that are larger in size. Fig. 6 shows the cumulative distribution

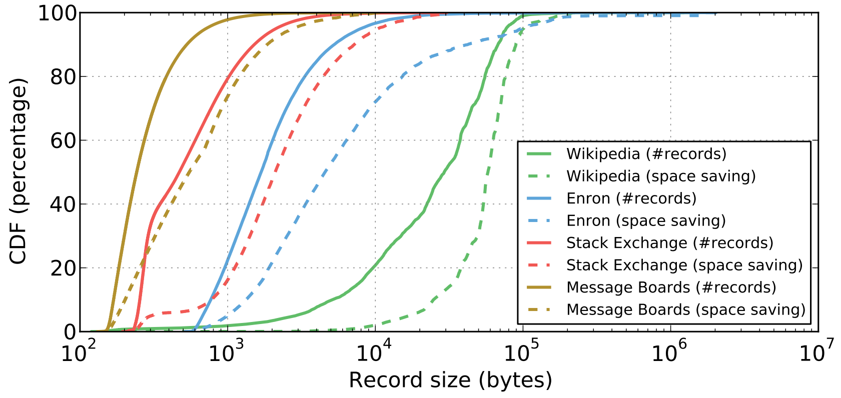


Figure 6: Size-based deduplication filter.

function (CDF) of record size and the weighted CDF by contribution to space saving for the four workloads used in our experiments. For these datasets, the 60% largest records account for approximately 90–95% of data reduction. In other words, if we only deduplicate records larger than the 40%-tile record size, we can reduce dedup overhead by 40% while only losing 5–10% of the compression ratio.

dDedup exploits this observation, using a size-based dedup filter that bypasses (treats as unique) records smaller than a certain threshold. Unlike specialized dedup systems whose workload characteristics are known in advance, dDedup determines the cut-off size on a per-database basis using a simple heuristic. For each database, the dedup threshold is first initialized to zero, meaning that all incoming records are deduplicated. This value is then periodically updated with the 40%-tile record size of the database every 1000 record insertions.

4 Implementation

This section describes dDedup implementation details, including how it fits into DBMS storage and replication frameworks and internals of its delta compression algorithm.

4.1 Integration into DBMSs

While implementation details vary across DBMSs, we illustrate the integration of dDedup using a simple distributed setup consisting of one client, one primary node and one secondary node, as shown in Fig. 7. For simplicity, we assume that only the primary node serves write requests,[2] and it pushes updates asynchronously to the secondary node in the form of oplog batches. Below, we describe dDedup behavior for primary DBMS operations.

Insert: Normally, record insertion works as follows. The primary writes the new record into its local database and appends the record to its oplog. Each oplog entry includes a timestamp and a payload that contains the inserted record. When the size of unsynchronized oplog entries accumulates to a certain amount, the primary pushes them in a batch to the secondary node. The secondary receives the updates, appends them to its local oplog, and replays the new oplog entries to update its local database.

With dDedup, a new record is first stored in the local oplog. Later, when preparing to store the record or send it to a replica, it is processed by the dDedup encoder following the deduplication steps described in

[2]When secondaries also serve write, each of them would maintain a separate dedup index. These indices would be updated during replica synchronization and eventually converge.

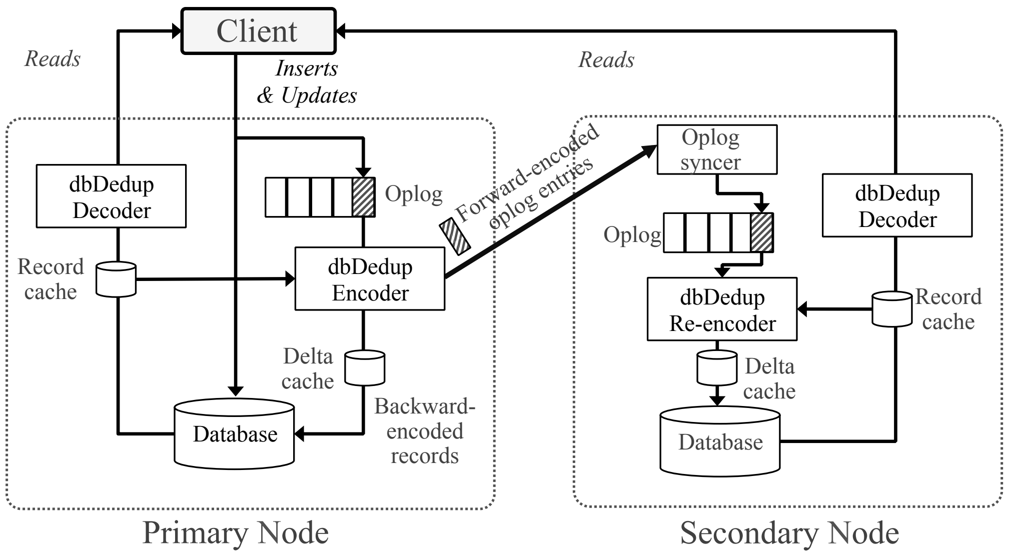


Figure 7: Integration of dDedup into a DBMS.

Section 3.1. If dDedup successfully selects a similar record from the existing data corpus, it retrieves the content of the similar record by first checking the source record cache. On cache misses, it directly reads the record from the underlying storage. It then applies bidirectional delta compression to the source and target records to generate the forward-encoded form of the new record and the backward-encoded form of the similar record. dDedup inserts the new record to the primary database in its original form and caches the backward-encoded similar record in the lossy write-back cache until system I/O becomes idle. Then, dDedup appends the forward-encoded record to the primary oplog that is transferred to the secondary during replica synchronization.

On the secondary side, an oplog syncer receives and propagates the encoded oplog entries to the dDedup re-encoder. The re-encoder first decodes the new record by reading the base similar record from its local database[3] (or the source record cache, on hits) and applying the forward-encoded delta. It then delta compresses the similar record using the newly reconstructed new record as the source, like in the primary, and generates the same backward-encoded delta for the similar record. Finally, dDedup writes the new record to the secondary database and updates the similar record to its delta-encoded form. These steps ensures that the secondary stores the same data as the primary node.

dDedup internally keeps track of a reference count for each stored record to indicate the number of records referencing it as a base for decode. Because dDedup uses backward encoding for database storage, after insertion, the reference count of the new record is set to one, while that of the similar record is unchanged. The reference count of the original base of the similar record, if existing, is reduced by one.

Update: Upon update, dDedup first checks the reference count of the queried record. If the count is zero, meaning no other records refer to it for decoding, dDedup directly applies the update as normal. Otherwise, dDedup keeps the current record intact and appends the update to it. Doing so ensures that other records using it as a reference can still be decoded successfully. When the reference count drops to zero, dDedup compacts all the updates to the record and replaces it with the new data.

dDedup uses a write-back cache to delay the update of a similar source record to its encoded form. To prevent it from overwriting any subsequent updates to the source record, dDedup always checks the

[3]Because the secondary and primary nodes are mostly synchronized, the base record used in the primary to encode the new record is almost always also present in the secondary. In rare cases where it is not, the secondary directly queries the primary node for the new record to avoid extra decoding overhead.

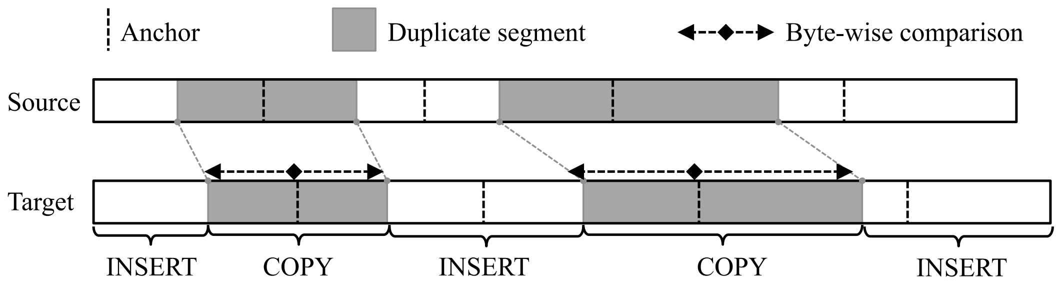


Figure 8: Illustration of delta compression in dDedup.

write-back cache before applying a client update. If it finds a record with the same ID (to be written back later), it simply evicts it from the cache, ensuring transactional correctness at the cost of compression loss for the corresponding record.

Delete: If the specific record has a zero reference count, the deletion proceeds as normal. Otherwise, dDedup marks it as deleted but retains its content. Any client reads to a deleted record returns an empty result, but it can still serve as a decoding base for other records referencing it. When dDedup detects that the reference count becomes zero, the record is removed from the database.

Read: On read, if the queried record is stored in unencoded form, it is directly sent to the client just like the normal case. If encoded, the record is first decoded by the dDedup decoder before returned to the client. During decoding, the decoder fetches the base record from the source record cache (or storage, on cache miss) and reconstructs the queried record using the stored delta. If the base record itself is encoded, the decoder repeats the step above iteratively until it finds a base record stored in its entirety.

4.2 Delta Compression

To ensure lightweight dedup, it is important to make dDedup's delta compression fast and efficient. The delta compression algorithm used in dDedup is adapted from xDelta [40], a classic copy/insert encoding algorithm using a string matching technique to locate matching offsets in the source and target byte streams. The original xDelta algorithm mainly works in two steps. In the first step, xDelta divides the source stream into fixed-size (by default, 16-byte) blocks. It then calculates an Alder32 [31] checksum (the same fingerprint function used in gzip) for each byte block and builds a temporary in-memory index mapping the checksums to their corresponding offsets in the source. In the second step, xDelta scans the target object byte by byte from the beginning, using a sliding window of the same size as the byte blocks. For each target offset, it calculates a Alder32 checksum of the bytes in the sliding window and consults the source index populated in the first step. If it finds a match, xDelta extends the search process from the matched offsets, using bidirectional byte-wise comparison to determine the longest common sequence (LCS) between the source and target streams. It then skips the matched region to continue the iterative search. If it does not find a match, it moves the sliding window by one byte and restarts the matching. Along this process, xDelta encodes the matched regions in the target into COPY instructions and the unmatched regions into INSERT instructions.

The delta compression algorithm used in dDedup, as shown in Algorithm 1 and Fig. 8, modifies xDelta based on the observation that a large fraction of computation time is spent in source index building and lookups. To mitigate this overhead, in the first encoding step, dDedup samples a subset of the offset positions called *anchors*, whose checksums' lower bits match a pre-determined pattern. The interval between anchors indicates the sampling ratio and is controlled by the length of the bit pattern, similar to how variable-sized chunking algorithms determine the average chunk size. In the second step, dDedup performs index lookups only for the anchors in the target, avoiding the need to consult the source index at every target offset. The

Algorithm 1 Delta Compress

```
1:  function DELTACOMPRESS($src, tgt$)
2:      $i \leftarrow 0$                                              ▷ Initialization
3:      $j \leftarrow 0$
4:      $pos \leftarrow 0$
5:      $ws \leftarrow 16$
6:      $sIndex \leftarrow empty$
7:      $tInsts \leftarrow empty$
8:      while $i + ws <= src.length$ do                           ▷ Build index for src anchors
9:          $hash \leftarrow$ RABINHASH($src, i, i + ws$)
10:         if ISANCHOR($hash$) then
11:             $sIndex[hash] \leftarrow i$
12:         end if
13:         $i \leftarrow i + 1$
14:     end while
15:     while $j + ws <= tgt.length$ do                           ▷ Scan tgt for longest match
16:         $hash \leftarrow$ RABINHASH($tgt, j, j + ws$)
17:         if ISANCHOR($hash$) and $hash$ in $sIndex$ then
18:             $(soff, toff, l) \leftarrow$ BYTECOMP($src, tgt, sIndex[fp], j$)
19:             if $pos < toff$ then
20:                 $insInst \leftarrow INST(INSERT, pos, toff - pos)$
21:                 $memcpy(insInst.data, tgt, toff - pos)$
22:                 $tInsts.append(insInst)$
23:             end if
24:             $cpInst \leftarrow INST(COPY, soff, l)$
25:             $tInsts.append(cpInst)$
26:             $pos \leftarrow toff + l$
27:             $j \leftarrow toff + l$
28:         else
29:             $j \leftarrow j + 1$
30:         end if
31:     end while
32:     return $tInsts$
33: end function
```

anchor interval provides a tunable trade-off between compression ratio and encoding speed, and we evaluate its effects in Section 5. Of course, some details are omitted in the pseudo-code given above. For example, contiguous and overlapping COPY instructions are coalesced; short COPY instructions are converted into equivalent INSERT instructions when the encoding overhead exceeds space savings.

As discussed in Section 3.2, after computing the forward-encoded data using the algorithm above, dDedup uses delta re-encoding (Algorithm 2) to efficiently generate the backward-encoded source record. Instead of switching the source and target objects and performing delta compression again, dDedup reuses the COPY instructions generated before and sorts them by their corresponding source offsets. It then fills the unmatched regions in the source with INSERT instructions. While it may result in slightly sub-optimal compression rate (e.g., due to overlapping COPY instructions that are merged), the re-encoding process is extremely fast (at memory speed), since there are no checksum calculations or index operations.

Delta decompression in dDedup is straightforward. It simply iterates over the instructions generated by the compression algorithm and concatenates the matched and unmatched regions to reproduce the original target object.

Algorithm 2 Delta Re-encode

```
function DELTAREENCODE($src, tgt, tInsts$)
    $sPos \leftarrow 0$
    $tPos \leftarrow 0$
    $copySegs \leftarrow empty$
    $sInsts \leftarrow empty$
    for each $inst$ in $tInsts$ do
        if $inst.type = COPY$ then
            $copySegs.append(inst.sOff, tPos, inst.len)$
        end if
        $tPos \leftarrow tPos + inst.len$
    end for
    $copySegs.sortBy(sOff)$
    for each $seg$ in $copySegs$ do
        if $sPos < seg.sOff$ then
            $insInst \leftarrow INST(INSERT, sPos, sOff - sPos)$
            $memcpy(insInst.data, src, sOff - sPos)$
            $sInsts.append(insInst)$
        end if
        $cpInst \leftarrow INST(COPY, seg.tOff, seg.len)$
        $sInsts.append(cpInst)$
        $sPos \leftarrow seg.sOff + seg.len$
    end for
    return $sInsts$
end function
```

5 Evaluation

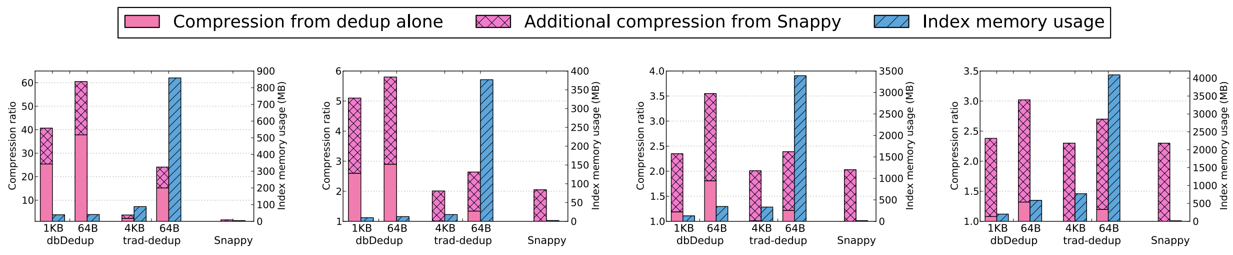
This section evaluates dDedup using four real-world datasets. For this evaluation, we implemented both dDedup and traditional chunk-based dedup (trad-dedup) in MongoDB (v3.1). The results show that dDedup provides significant compression benefits, outdoes traditional dedup, combines with block-level compression, and imposes negligible overhead on DBMS performance.

Unless otherwise noted, all experiments use a replicated MongoDB setup with one primary, one secondary, and one client node. Each node has four CPU cores, 8 GB RAM, and 100 GB of local HDD storage. We use MongoDB's WiredTiger [16] storage engine with the full journaling feature turned off to avoid interference.

5.1 Workloads

The four real-world datasets represent a diverse range of database applications: collaborative editing (Wikipedia), email (Enron), and on-line forums (Stack Exchange, Message Boards). We sort each dataset by creation timestamp to generate a write trace, and then generate a read trace using public statistics or known access patterns to mimic a real-world workload, as detailed below.

Wikipedia: The full revision history of every article in the Wikipedia English corpus [13] from January 2001 to August 2014. We extracted a 20 GB subset via random sampling based on article IDs. Each revision contains the new version of the article and metadata about the user that made the edits (e.g., username, timestamp, comment). Most duplication comes from incremental revisions to pages. We insert the first 10,000 revisions to populate the initial database. We then issue read and write requests according to a public Wikipedia access trace [59], where the normalized read/write ratio is 99.9 to 0.1. 99.7% of read requests are



(a) Wikipedia **(b)** Enron **(c)** Stack Exchange **(d)** Message Boards

Figure 9: Compression Ratio and Index Memory – The compression ratio and index memory usage for dDedup (1 KB chunks or 64 byte chunks), trad-dedup (4 KB and 64 byte), and Snappy. The upper portion of each dedup bar represents the added benefit of compressing after dedup.

to the latest version of a wiki page, and the remainder to a specific revision.

Enron: A public email dataset [2] with data from about 150 users, mostly senior management of Enron. The corpus contains around 500k messages, totaling 1.5 GB of data. Each message contains the text body, mailbox name, message headers such as timestamp and sender/receiver IDs. Duplication primarily comes from message forwards and replies that contain content of previous messages. We insert the sorted dataset into the DBMS as fast as possible. After each insertion, we issue a read request to the specific email message, resulting in an aggregate read/write ratio of 1 to 1. This is based on the assumption that each user uses a single email client that caches the requested message locally, so each message is written and read once to/from the DBMS.

Stack Exchange: A public data dump from the Stack Exchange network [10] that contains the full history of user posts and associated information such as tags and votes. Most duplication comes from users revising their own posts and from copying answers from other discussion threads. We extracted a 10 GB subset (of 100 GB total) via random sampling. We insert the posts into the DBMS as new records in temporal order. For each post, we read it for the same number of times as its view count. The aggregate read/write ratio is 99.9 to 0.1.

Message Boards: A 10 GB forum dataset containing users' posts crawled from a number of public vBulletin-powered [11] message boards that cover a diverse range of threaded topics, such as sports, cars, and animals. Each post contains the forum name, thread ID, post ID, user ID, and the post body including quotes from other posts. This dataset also contains the view count per thread, which we use to generate synthetic read queries. Duplication mainly originates from users quoting others' comments. To mimic users' behavior in a discussion forum, for each post insertion, we issue a certain number of "thread reads" that request all the previous posts in the containing thread. The number of thread reads per insertion is derived by dividing the total view count of the thread by the number of posts it contains.

5.2 Compression Ratio and Index Memory

We first evaluate dDedup's compression ratio and index memory usage and compare them to trad-dedup and Snappy [9], MongoDB's default block-level compressor. For each dataset, we load the records into the DBMS as fast as possible and measure the resulting storage sizes, the amount of data transferred over the network, and the index memory usage.

Fig. 9 shows the results for five configurations: (1) dDedup with chunks of 1 KB or 64 bytes, (2) trad-dedup with chunks of 4 KB or 64 bytes, and (3) Snappy. The pink (left) bar shows storage compression ratio, indicating the contribution of dedup alone and compression after dedup. The compression ratio is defined as original data size divided by compressed data size, so a value of one means no compression achieved. The network transfer compression ratio is within 5% of that for storage, in all cases. The blue (right) bar shows index memory usage. The small source record cache (32 MB, used by both dDedup and trad-dedup) and lossy write-back cache (8 MB, used by dDedup only) are not shown.

The benefits are largest for Wikipedia (Fig. 9a). With a chunk size of 1 KB, dDedup reduces data storage by 26× (41× combined with Snappy) using 36 MB index memory. Decreasing the chunk size to 64 B increases compression ratio to 37× (61×) using only 45 MB index memory. Decreasing chunk size for dDedup does not increase index memory usage much, because dDedup indexes at most K entries per record, regardless of chunk size. In contrast, while trad-dedup's compression ratio increases from 2.3× (3.7×) to 15× (24×) when using a chunk size of 64 B instead of 4 KB, its index memory grows from 80 MB to 780 MB, making it impractical for operational DBMSs. This is because trad-dedup indexes every unique chunk hash, leading to almost linear increase of index overhead as chunk size decreases, and also because it must use much larger index keys (20-byte SHA-1 hash vs. 2-byte checksum) since collisions would result in data corruption. Consuming 40% less index memory, dDedup with 64 B chunk size achieves a compression ratio 16× higher than trad-dedup with its typical 4 KB chunk size. Snappy compresses the dataset by only 1.6×, because it can not eliminate the duplication caused by application-level versioning, but requires no index memory. It provides the same 1.6× compression when applied to the deduped data.

For the other datasets, the absolute benefits are smaller, but the primary observations are similar: dDedup provides higher compression ratio with lower memory usage than trad-dedup, and Snappy's compression benefits (1.6–2.3×) complement deduplication. For the Enron dataset (Fig. 9b), dDedup reduces storage by 3.0× (5.8×), which is consistent with results we obtained from experiments with data from a cloud deployment of Microsoft Exchange servers containing PBs of real user email data.[4] The two forum datasets (Figs. 9c and 9d) do not exhibit as much duplication as the Wikipedia or email datasets, because users do not quote or edit comments as frequently as Wikipedia revisions or email forwards/replies. Even so, we still observe that dDedup reduces storage by 1.3–1.8× (3–3.5×). Because we were only able to crawl the latest posts in the Message Boards dataset, dDedup's compression ratio is conservative, not including the benefits from delta compressing users' revisions to their own posts.[5]

5.3 Runtime Performance Impact

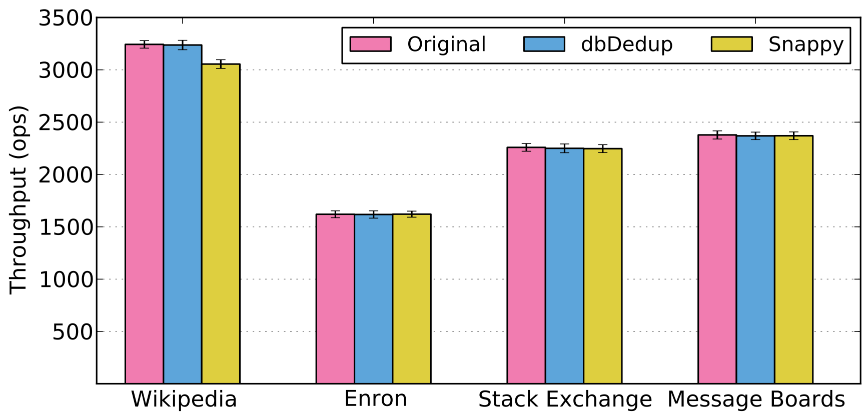
This subsection shows that dDedup has negligible impact on DBMS performance by comparing three MongoDB configurations: no compression, with dDedup, and with Snappy.

Throughput: Fig. 10a shows throughput for the four workloads. We see that dDedup imposes negligible overhead on throughput. Snappy also degrades performance slightly for three of the workloads, since it is a fast and lightweight inline compressor. The exception is Wikipedia, for which using Snappy causes 5% throughput reduction, because some large Wikipedia records cannot fit in a single WiredTiger page and require extra I/Os.

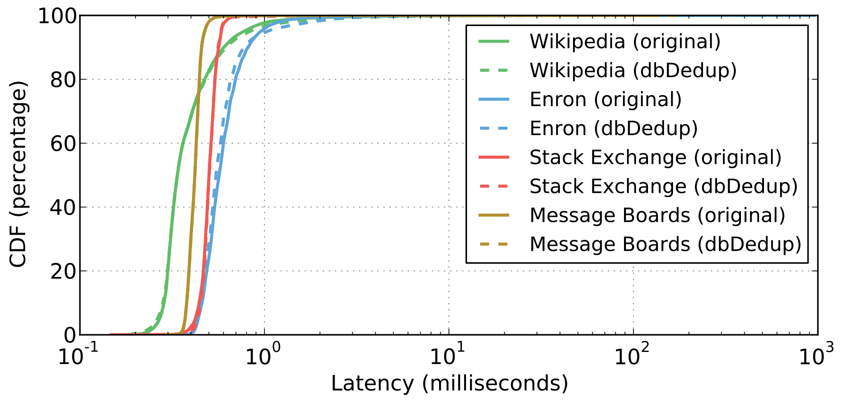
Latency: Fig. 10b shows the CDF of client latency. For clarity, we only show the results for MongoDB with and without dDedup enabled. Again, we observe that dDedup has almost no effect on performance. The latency distribution curves with dDedup enabled closely track those for no compression/dedup. The difference in the 99.9%-tile latency is less than 1% for all workloads.

[4]Sadly, we cannot reveal details due to confidentiality restrictions.

[5]We find that 15% of posts are edited at least once, and most edited posts are larger than the average post size.



(a) Throughput



(b) Latency

Figure 10: Performance Impact – Runtime measurements of MongoDB's throughput and latency for the different workloads and configurations.

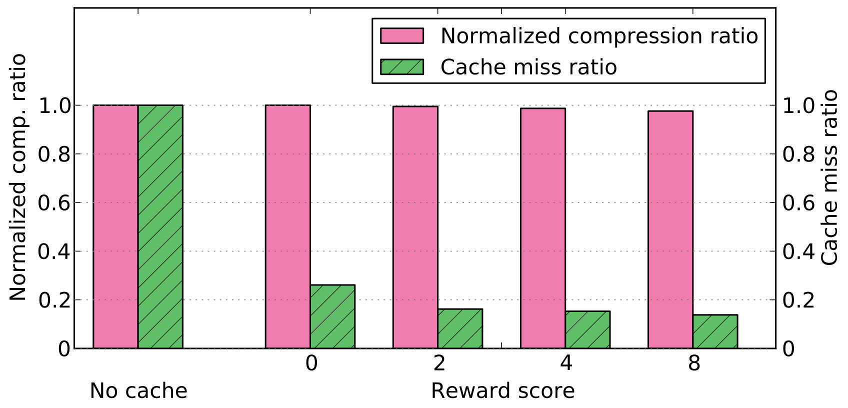
5.4 Effects of Caching

As described in Section 3.3, dDedup uses two small caches to minimize I/O overheads involved in reading and updating source records: a source record cache (32 MB) and lossy write-back cache (8 MB). We now evaluate the effectiveness of these caches.

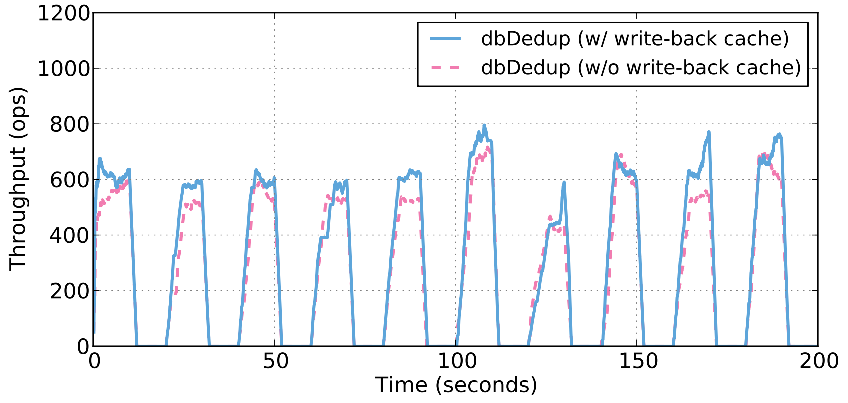
Source Record Cache: Fig. 11a shows the effect of the source record cache on compression ratio (left Y-axis) and percent of source record retrievals requiring a DBMS read (cache miss ratio; right Y-axis), with a range of reward score values for the Wikipedia workload. Recall that dDedup uses cache-aware selection of candidate similar records, assigning a reward score to candidates that are present in the cache (see Section 3.1.3).

When no cache is used (the left-most bars), every retrieval of a source record incurs a read query. Even without cache-aware selection (0 reward score), the small source record cache eliminates 74% of these queries. With a reward score of two (default), the cache-aware selection technique further cuts the miss ratio by 40% (to 16%), without reducing the compression ratio noticeably. Further increases to the reward score marginally reduce the cache miss ratio while reducing the compression ratio slightly, because less similar candidates are more likely to be selected as the source records.

Lossy Write-back Cache: dDedup uses backward encoding to avoid decode when reading the latest



(a) Source Record Cache



(b) Lossy Write-back Cache

Figure 11: Effects of Caching – Runtime measurements of dDedup's caches in MongoDB for the Wikipedia workload.

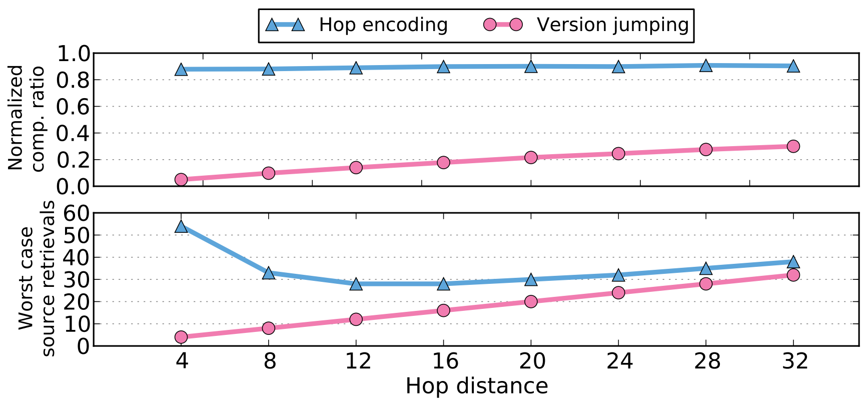
"versions" of an update sequence. Thus, deduplicating a new record involves both writing the full new record and replacing the source record with delta-encoded data. The extra write (the replacing) may lead to significant performance problems for I/O intensive workloads during write bursts. dDedup's lossy write-back cache mitigates such problems.

To emulate a bursty workload with I/O intensive and idle periods, we insert Wikipedia data at full speed for 10 seconds and sleep for 10 seconds, repeatedly. Fig. 11b shows MongoDB's insertion throughput over time, with and without the write-back cache. Without the cache, DBMS throughput visibly decreases during busy periods because of the extra database writes. In contrast, using the write-back cache avoids DBMS slowdown during workload bursts, as shown by the difference between the two lines at various points of time (e.g., at seconds 0, 130, 170, and 190).

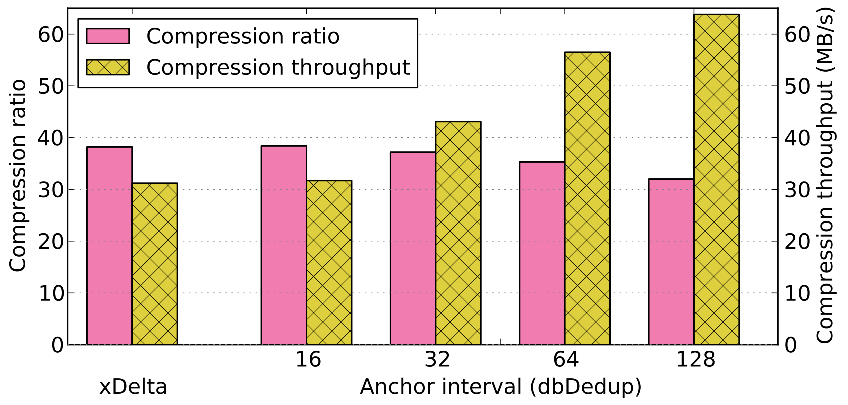
5.5 Tuning Parameters

dDedup has two primary tunable parameters, beyond those explored above, that affect compression/performance trade-offs: hop distance and anchor interval. This subsection quantifies the effects of these parameters and explains the default values.

Hop Distance: dDedup uses hop encoding for its backward encoding to reduce the worst-case retrieval times when reading records. For comparison, we also implemented version jumping used in previous delta



(a) Hop Distance



(b) Anchor Interval

Figure 12: Tuning Parameters – Runtime measurements of dDedup's tuning parameters in MongoDB for the Wikipedia workload. The hop distance results in Fig. 11a include the DBMS's compression ratio (upper graph) and worst-case source retrievals (lower graph).

encoding systems (see Section 3.2.2).

Fig. 12a shows the normalized compression ratio and worst-case number of source record retrievals (for an encoding chain length of 200), as a function of hop distances (called "cluster sizes" in version jumping), for the Wikipedia workload. With a small hop distance (e.g., 4), the compression ratio achieved with version jumping drops over 90%, because all reference versions are stored unencoded. In contrast, using hop encoding causes a drop of only 10% because adjacent hop bases are encoded between each other. As expected, the number of source retrievals for hop encoding is much larger than for version jumping, at this small hop distance, because accessing a record at the beginning of the encoding chain requires a number of hops inversely proportional to the hop distance.

For version jumping, the compression ratio and the worst-case retrieval count both increase with hop distance; fewer records are stored uncompressed, and more retrievals are needed to traverse an encoding cluster. For hop encoding, the compression ratio remains relatively steady as hop distance increases, due to having fewer but less similar hop bases. The worst-case retrieval times quickly approach those of version jumping, however, as the number of hops to traverse the backward encoding chain decreases. Empirically, we find that a hop distance of 16 (default) provides a good trade-off between compression ratio and decoding overhead.

Anchor Interval: dDedup outperforms the xDelta algorithm by reducing the computation overhead on source index insertion and lookups. It introduces a tunable anchor interval that controls the sampling rate of the offset points in the source byte stream.

Fig. 12b shows the compression ratio (left Y-axis) and throughput (right Y-axis) for various dDedup anchor interval values, as well as for xDelta, for the Wikipedia workload. With an anchor interval of 16 (default window size in xDelta), dDedup performs almost the same as xDelta. As anchor interval increases, dDedup's delta compressing speed improves, because it reduces the number of source offset index insertions and lookups. The compression ratio does not significantly decrease, because dDedup performs byte-level comparison bidirectionally from the matched points. With an anchor interval of 64, dDedup outperforms xDelta by 80% in terms of compression throughput, while incurring only 7% loss in compression ratio. Increasing the anchor interval to 128 further improves the throughput by 10% but results in 15% loss in compression ratio. We use 64 as the default value, providing a balance between compression ratio and throughput.

6 Additional Related Work

Most previous dedup work is discussed in Section 2. This section discusses some additional related work.

Database Compression: A number of database compression schemes haven been proposed during the past few decades. Most operational DBMSs that compress the database contents use page or block-level compression [29, 36, 44, 41, 3, 16]. Some use prefix compression, which looks for common sequences in the beginning of field values for a given column across all rows on each page. Just as with our dDedup approach, such compression requires the DBMS to decompress tuples before they can be processed during query execution.

There are schemes in some OLAP systems that allow the DBMS to process data in its compressed format. For example, dictionary compression replaces recurring long domain values with short fixed-length integer codes. This approach is commonly used in column-oriented data stores [18, 35, 66, 48]. These systems typically focus on attributes with relatively small domain size and explore the skew in value frequencies to constrain the resulting dictionary to a manageable size [23]. The authors in [53] propose a delta encoding scheme where every value in a sorted column is represented by the delta from the previous value. Although this approach works well for numeric values, it is unsuitable for strings.

None of these techniques detect and eliminate redundant data with a granularity smaller than a single field, thus losing potential compression benefits for many applications that inherently contain such redundancy. dDedup, in contrast, is able to remove much more fine-grained duplicates with byte-level delta compression. Unlike other inline compression schemes, dDedup is not in the critical write path for queries, and hence, it has minimal impact on the DBMS's runtime performance. In addition to this, because dDedup compresses data at record level, it only performs the dedup steps once, and uses the encoded result for both database storage and network transfer. In contrast, the same record would be compressed twice (in database page and oplog batch), for page compression schemes to achieve data reduction at both layers.

Delta Compression: There has been much previous work on delta compression, including several general-purpose algorithms based on the Lempel-Ziv approach [65], such as vcdiff [21], xDelta [40], and zdelta [57]. Specialized schemes can be used for specific data formats (e.g., XML) to improve compression quality [28, 60, 38, 49]. The delta compression algorithm used in dDedup is adapted from xDelta, to which the relationship is discussed in Section 4.2.

Delta compression has been used to reduce network traffic for file transfer and synchronization protocols. Most systems assume that previous versions of the same file are explicitly identified by the application, and duplication only exists among prior versions of the same file [58, 54]. On exception is TAPER [37], which reduces network transfer for synchronizing file system replicas by sending delta-encoded files; it identifies similar files by computing the number of matching bits on the Bloom filters generated with the files' chunk hashes. dDedup identifies a similar record from the data corpus without application guidance and therefore is a more generic approach than most of these previous systems.

The backward-encoding technique used in dDedup is inspired by versioned storage systems such as RCS [56] and XDFS [40]. While these systems explicitly maintain versioning lineage for all the files, dDedup builds the encoding chain purely based on similarity relationships between records, and thus does not require system-level support for versioning. [51] uses delta encoding for deduplicated backup storage. It uses forward encoding and only supports a longest encoding chain length of two. dDedup uses hop encoding on top of backward encoding to reduce the worst-case source retrievals for read requests. In addition, it uses several novel caching mechanisms to further mitigate the I/O overhead involved in reading and updating encoded records.

Similarity Detection: Prior work has provided various approaches to computing sketches (similarity metrics) for identifying similar items. The basic technique of identifying features in objects so that similar objects have identical features was pioneered by Broder [24, 25] in the context of web pages. Several papers [55, 45, 20, 50, 62] propose methods for computing sketches for similarity detection that are robust to small edits in the data. The feature extraction approach used in dDedup is similar to that in DOT [45] and sDedup [62].

7 Conclusion

dDedup is a lightweight similarity-based deduplication engine for operational DBMSs that reduces both storage usage and the amount of data transferred for remote replication. Combining partial indexing and byte-level delta compression, dDedup achieves higher compression ratios than block-level compression and chunk-based deduplication while being memory efficient. It uses novel encoding and caching mechanisms to avoid significant I/O overhead involved in accessing delta-encoded records. Experimental results with four real-world workloads show that dDedup is able to achieve up to 37× reduction (61× when combined with block-level compression) in storage size and replication traffic while imposing negligible overhead on DBMS performance.

References

[1] Baidu Baike. http://baike.baidu.com/.

[2] Enron Email Dataset. https://www.cs.cmu.edu/~./enron/.

[3] InnoDB Compression. http://dev.mysql.com/doc/refman/5.6/en/innodb-compression-internals.html.

[4] Linux SDFS. www.opendedup.org.

[5] MongoDB. http://www.mongodb.org.

[6] MurmurHash. https://sites.google.com/site/murmurhash.

[7] Ocarina Networks. www.ocarinanetworks.com.

[8] Permabit Data Optimization. www.permabit.com.

[9] Snappy. http://google.github.io/snappy/.

[10] Stack Exchange Data Archive. https://archive.org/details/stackexchange.

[11] vBulletin. https://www.vbulletin.com.

[12] W3Techs. http://www.w3techs.com.

[13] Wikimedia Downloads. https://dumps.wikimedia.org.

[14] Wikipedia. https://www.wikipedia.org/.

[15] Windows Storage Server. technet.microsoft.com/en-us/library/gg232683(WS.10).aspx.

[16] WiredTiger. http://www.wiredtiger.com/.

[17] ZFS Deduplication. blogs.oracle.com/bonwick/entry/zfs_dedup.

[18] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[19] C. Alvarez. NetApp deduplication for FAS and V-Series deployment and implementation guide. 2010.

[20] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *SYSTOR*, page 6, 2009.

[21] J. Bentley and D. McIlroy. Data compression using long common strings. In *Data Compression Conference, 1999. Proceedings. DCC'99*, pages 287–295, 1999.

[22] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS*, pages 1–9, 2009.

[23] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[24] A. Broder. On the resemblance and containment of documents. Compression and Complexity of Sequences, 1997.

[25] A. Broder. Identifying and filtering near-duplicate documents. 11th Annual Symposium on Combinatorial Pattern Matching, 2000.

[26] R. C. Burns and D. D. Long. Efficient distributed backup with delta compression. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 27–36, 1997.

[27] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX ATC*, 2009.

[28] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *ICDE*, pages 41–52, 2002.

[29] G. V. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[30] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX Annual Technical Conference*, 2010.

[31] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[32] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, and J. Szczepkowski. Hydrastor: A scalable secondary storage. In *FAST*, 2009.

[33] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, , and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST*, 2009.

[34] A. El-Shimi, R. Kalach, A. K. Adi, O. J. Li, and S. Sengupta. Primary data deduplication-large scale study and system design. In *USENIX Annual Technical Conference*, 2012.

[35] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[36] B. Iyer and D. Wilhite. Data compression support in databases. 1994.

[37] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *FAST*, 2005.

[38] E. Leonardi and S. S. Bhowmick. Xanadue: a system for detecting changes to xml data in tree-unaware relational databases. In *SIGMOD*, pages 1137–1140, 2007.

[39] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *FAST*, 2009.

[40] J. P. MacDonald. File system support for delta compression. Master's thesis, University of California, Berkeley, 2000.

[41] S. Mishra. Data compression: Strategy, capacity planning and best practices. *SQL Server Technical Article*, 2009.

[42] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, 2001.

[43] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[44] M. Poess and D. Potapov. Data compression in oracle. In *VLDB*, pages 937–947, 2003.

[45] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

[46] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, 2002.

[47] M. O. Rabin. *Fingerprinting by random polynomials.*

[48] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *VLDB*, 6(11):1080–1091, 2013.

[49] S. Sakr. Xml compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322, 2009.

[50] P. Shilane, M. Huang, G. Wallace, and W. Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. In *FAST*, 2012.

[51] P. Shilane, G. Wallace, M. Huang, and W. Hsu. Delta compressed and deduplicated storage using stream-informed locality. *USENIX Hot Storage*, 2012.

[52] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *FAST*, 2012.

[53] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[54] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. *Lossless Compression Handbook*, 2002.

[55] D. Teodosiu, Y. Gurevich, M. Manasse, and J. Porkka. Optimizing file replication over limited bandwidth networks using remote differential compression. *Tech. Rep. MSR-TR-2006-157, Microsoft Research*, 2006.

[56] W. F. Tichy. Rcs–a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[57] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool. *Technical Report TR-CIS-2002-02, Polytechnic University*, 2002.

[58] A. Tridgell. Efficient algorithms for sorting and synchronization. In *PhD thesis, Australian National University* , 2000.

[59] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[60] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *ICDE*, pages 519–530, 2003.

[61] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *USENIX Annual Technical Conference*, 2011.

[62] L. Xu, A. Pavlo, S. Sengupa, J. Li, and G. R. Ganger. Reducing replication bandwidth for distributed document databases. In *SoCC*, pages 222–235, 2015.

[63] L. L. You, K. T. Pollack, and D. D. Long. Deep store: An archival storage system architecture. In *ICDE*, pages 804–815, 2005.

[64] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.

[65] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[66] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, pages 59–59, 2006.