Online Deduplication for Databases

Lianghong Xu
Carnegie Mellon University
lianghon@andrew.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Sudipta Sengupta
Microsoft Research
sudipta@microsoft.com

Gregory R. Ganger
Carnegie Mellon University
ganger@ece.cmu.edu

ABSTRACT

dbDedup 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, dbDedup uses byte-level delta encoding of individual records within the database to achieve greater savings. dbDedup'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 dbDedup in a distributed NoSQL DBMS and analyzed its properties using four real datasets. Our results show that dbDedup 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. dbDedup provides both benefits with negligible effect on DBMS throughput or client latency (average and tail).

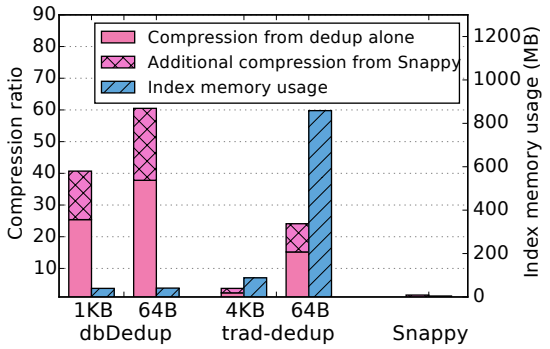


Figure 1: Compression ratio and index memory usage for Wikipedia data stored in five MongoDB configurations: with dbDedup (1 KB chunk size and 64 B), with traditional dedup (4 KB and 64 B), and with Snappy (block-level compression). dbDedup provides higher compression ratio and lower index memory overhead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

1. INTRODUCTION

The rate of data growth is exceeding 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 operational DBMSs is block-level compression [30, 37, 46, 43, 3, 16]. Such DBMSs are used to support user-facing applications that execute simple queries to retrieve a small number of records at a time (as opposed to performing complex queries that scan large segments of the database). Although block-level compression 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 application-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 applications execute update queries that modify single records. The amount of duplicate data in an individual record is likely insignificant. But large chunk sizes (e.g., 4–8 KB) are the norm to avoid huge in-memory indexes and large numbers of disk reads.

This paper presents **dbDedup**, a lightweight scheme for on-line database systems that uses *similarity-based deduplication* [65] to compress individual records. Instead of indexing every chunk hash, dbDedup 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. dbDedup 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 introduce and combine several techniques to achieve this efficiency. Foremost is that we present a novel *two-way encoding* to efficiently transfer encoded new records (forward encoding) to remote replicas, while storing new records with encoded forms of selected source records (backward encoding). As a result, no de-

SIGMOD'17, May 14-19, 2017, Raleigh, NC, USA

DOI: http://dx.doi.org/10.1145/3035918.3035938

code 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, we also introduce a *lossy write-back delta cache* tuned to maximize compression ratio while avoiding I/O contention. Our approach also uses a new technique, called *hop encoding*, that minimizes the worst-case number of decode steps required to access a specific record in a long encoding chain. Finally, we describe how to adaptively disable deduplication for databases and records where little savings are expected.

To evaluate our approach, we implemented dbDedup in the MongoDB DBMS [5] and measured its efficacy using four real-world datasets. Our results show that it achieves up to 37× reduction (61× when combined with block-level compression) in storage size and replication traffic. dbDedup outperforms chunk-based dedup while imposing negligible impact on the DBMS's performance.

This paper makes the following contributions.

1. To the best of our knowledge, we present 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.
2. We introduce novel techniques that are critical to achieving acceptable dedup efficiency, enabling practical use for online database storage.
3. We evaluate a full implementation of the system in a distributed NoSQL DBMS, using four real-world datasets.

The rest of this paper is organized as follows. Section 2 motivates use of similarity-based dedup for database applications and categorizes our approach relative to other dedup systems. Section 3 describes dbDedup's dedup workflow and mechanisms. Section 4 details dbDedup's implementation, including its integration into the storage and replication frameworks of a DBMS. We then evaluate our approach using several real-world data sets in Section 5. Lastly, we conclude in Section 6 with a discussion of the 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 dbDedup into context by categorizing previous dedup systems.

2.1 Why Dedup for Database Applications?

The most common way that operational DBMSs reduce the storage size of data is through block-level compression on individual database pages. For example, MySQL's InnoDB can compress pages when they are evicted from memory and written to disk [3]. When these pages are brought back into memory, the system can keep the pages compressed as long as no query tries to read its contents. Since the scope of the compression algorithm is only a single page, the amount of reduction that the system can achieve is low.

Analytical DBMSs use more aggressive schemes (e.g., dictionary compression, run-length encoding) that significantly reduce the size of a database [18]. This is because these systems compress individual columns, and thus there is higher likelihood of duplicate data. And unlike in the above MySQL example, they also support query processing directly on compressed data.

This type of compression is not practical in an operational DBMS. These systems are designed for highly concurrent workloads that execute queries that retrieve a small number of records at a time. If the DBMS had to compress each attribute every time a new record was inserted, then they would be too slow to support on-line, Web-based applications.

We observe, however, that many database applications could benefit from dedup due to similarities between non-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. Although dedup is widely used in file systems, it has not been fully explored in operational databases. 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.

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, dbDedup 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) [67, 40, 27, 34, 35]. The second ("similarity-based dedup") looks for similar units (chunks or files) and applies delta compression to them [61, 53, 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 dbDedup'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") [44, 49, 67] works as follows. An incoming file (corresponding to a new record in the context of

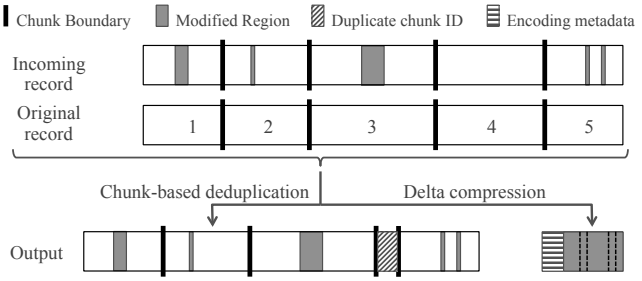


Figure 2: Comparison between chunk-based deduplication and similarity-based deduplication using delta compression for typical database workloads with small and dispersed modifications.

	Exact Dedup	Similarity-based Dedup	
Primary	iDedup [55] ZFS [17] SDFS [4] Windows server 2012 [15] NetApp ASIS [19] Ocarina [7] Permabit [8]	***dbDedup***	
Secondary	DDFS [67] Venti [49] ChunkStash [31] DEDE [27] HydraStor [33]	Extreme binning [22] Sparse Indexing [40] Silo [64] SIDC [53] DeepStore [66]	 SDS [20] sDedup [65]

Table 1: Categorization of related work

DBMS) is first divided into chunks using Rabin-fingerprinting [50]; 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, dbDedup'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, dbDedup's byte-level delta compression is able to identify much more fine-grained duplicates and thus provide greater compression ratio than chunk-based dedup.

2.3 Categorizing Dedup Systems

Table 1 illustrates one view of how dbDedup 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, dbDedup 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 [67, 40, 20, 53, 54] 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. dbDedup, 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 [55]) or in the background as post-processing on the stored data (Windows Server 2012 [35]), 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, dbDedup 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 [53] and sDedup [65] 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 dbDedup 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. dbDedup DESIGN

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

3.1 Deduplication Workflow

dbDedup 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 dbDedup 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, dbDedup extracts similarity features from the new record using a content-dependent approach. dbDedup divides the new record into several variable-sized data chunks using the Rabin Fingerprinting algorithm [50] 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, dbDedup calculates a (weaker, but computationally cheaper) MurmurHash [6] for each chunk and

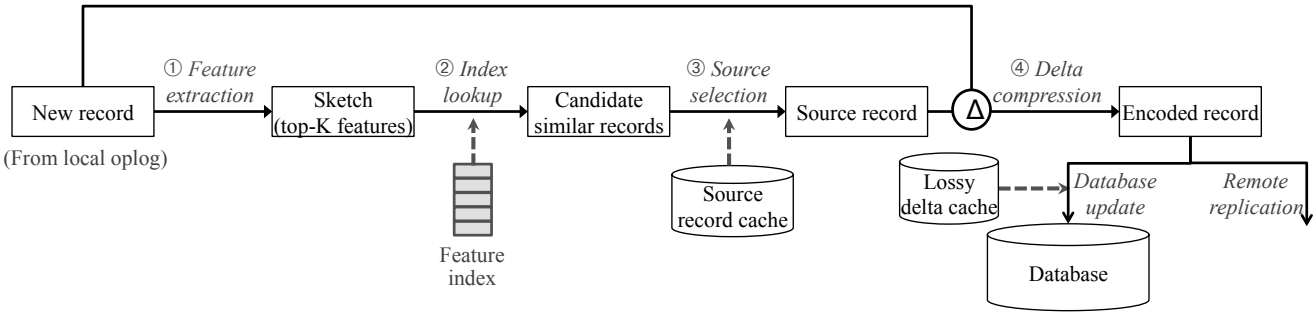


Figure 3: dbDedup Workflow – (1) Feature Extraction, (2) Index Lookup, (3) Source Selection, and (4) Delta Compression.

only indexes a representative subset of the chunk hashes. dbDedup adapts a technique called *consistent sampling* [47] 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, dbDedup bounds the memory overhead of its dedup index to be at most K index entries per record. This important property allows dbDedup to use small chunk sizes for better similarity detection while not consuming excessive RAM like in chunk-based dedup. Moreover, because dbDedup 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 dbDedup performs delta compression in the final step.

3.1.2 Index Lookup

For each extracted feature, dbDedup finds existing records that share that feature with the new record. Since dbDedup is an online dedup system, it is imperative that this index lookup process is fast and efficient. dbDedup achieves this by building an in-memory feature index that uses a variant of Cuckoo hashing [45, 31] 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, dbDedup 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. dbDedup then inserts the feature and a reference to the new record to the empty bucket for future lookup. Finally, dbDedup 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, dbDedup 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.

[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.

3.1.3 Source Selection

The index lookup results may contain multiple candidate similar records, yet dbDedup 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, dbDedup 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, dbDedup 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, dbDedup 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 dbDedup 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.

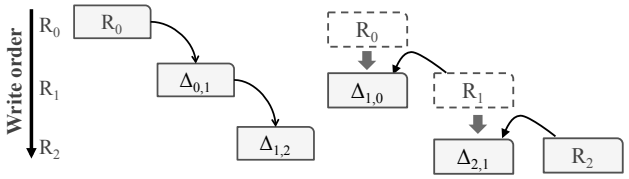
3.2 Encoding for Online Storage

Efficient access of delta-encoded storage is a long-standing challenge due to the I/O and computation overhead involved in the encoding and decoding steps. In particular, reconstructing encoded data may require reading all the deltas along a long encoding chain until reaching an unencoded (raw) data-item. To provide reasonable performance guarantees, most online systems use delta encoding only to reduce network transmission (leaving storage unencoded) or use it to a very limited extent in the storage components (e.g., by constraining the maximum length of the encoding chain to a small value). But, doing so significantly under-exploits the potential space savings that could be achieved.

dbDedup greatly alleviates the painful tradeoff between compression gains and access speed in delta encoded storage with two new encoding schemes. It uses a *two-way encoding* technique that reduces both remote replication bandwidth and database storage, while optimizing for common case queries. In addition, it uses *hop encoding* to reduce worst-case source retrievals for reading encoded records, while largely preserving the compression benefits.

3.2.1 Two-way Encoding

After a candidate record is selected from the data corpus, dbDedup generates the byte-level difference between the candidate and



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

Figure 4: Illustration of two-way encoding – dbDedup uses forward encoding to reduce the network bandwidth for replica synchronization while using backward encoding to compress database storage.

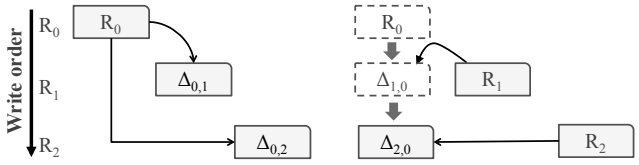
the new record in dual directions, using a technique that we call *two-way encoding*. For network transmission, dbDedup performs *forward encoding* (Fig. 4a), which uses the older (i.e., the selected candidate) record as the source and the new record as the target. After the encoding, the source remains in its original form, while the target is encoded as a reference to the source plus the delta from the source to the target. dbDedup 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.

dbDedup 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 application-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, dbDedup uses *backward encoding* (Fig. 4b) for local storage to optimize for read queries to recent records. That is, for local storage, dbDedup 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. Read queries to the latest version thus incur no decoding overhead at all. Although backward encoding is optimized for reads, it creates two potential issues. First, it amplifies the number of write operations, since an older record selected as a source needs to be updated to the encoded form. To mitigate the write amplification, dbDedup caches backward-encoded records to be written back to the database and delays the updates until system I/O is relatively idle, which we discuss in more detail in Section 3.3. A second issue arises when an older record is selected as the source. The existing data (a delta from its current base record) is replaced by the delta from the new record. Since backward encoding realizes space savings by updating delta sources, such *overlapped encoding* (Fig. 5) on the same source records can lead to some compression loss. Forward encoding, in contrast, naturally avoids this problem since no writeback is required. Fortunately, we find overlapped encoding is not common in real-world applications—most ($> 95\%$) updates are incremental based on the latest version (see Section 5.2).

dbDedup performs delta encoding between new and candidate records in two directions, yet it only incurs the computation overhead of one encoding pass. It achieves this by first generating the forward-encoded data and then efficiently transforming it into the backward delta at memory speed. We call this process *re-encoding* and detail the algorithm in Section 4.2.

3.2.2 Hop Encoding



Forward encoding Backward encoding

Figure 5: Overlapped encoding – Backward encoding may lead to compression loss when an older record is selected as the source. In this example, when R_0 is selected as the source for R_2, backward encoding leaves R_1 and R_2 both unencoded.

	Storage usage	#Worst-case retrievals	#Writebacks
Backward encoding	$S_b + (N-1) \cdot S_d$	N	N
Version jumping	$\frac{N}{H} \cdot S_b + (N - \frac{N}{H}) \cdot S_d$	H	$N - \frac{N}{H}$
Hop encoding	$S_b + (N-1) \cdot S_d$	$H + \log_H N$	$N + N \cdot \frac{H}{(H-1)^2}$

Table 2: Summary of the different encoding schemes – Hop encoding largely eliminates the painful tradeoff between space savings and decoding speed. N is the length of the encoding chain, and H denotes the hop distance (cluster size for version jumping). S_b and S_d refer to the size of a base record and a delta respectively, where $S_b \gg S_d$ in most cases. These sizes obviously vary for different records. Here we use the general notation for ease of reasoning.

As discussed above, using backward encoding minimizes the decoding overhead for reading recent records, but it may still incur excessive source retrieval time for occasional queries to older records (e.g., a specific version of a Wikipedia article). Prior work on delta encoded storage [26, 42] used a technique called *version jumping* to cope with this problem, by bounding the worst-case number of source retrievals at the cost of lower compression benefits. 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 results in lower compression ratio, because the reference versions are not compressed. As the encoding cluster size decreases, the compression loss can increase significantly, since deltas are usually much smaller than base records.

dbDedup uses a novel technique that we call *hop encoding*, which preserves the compression ratio close to standard backward encoding, while achieving comparable worst-case retrieval times to the version jumping approach. As illustrated in Fig. 6, extra deltas are computed between particular records and others some distance back in the chain, in a fashion similar to skip lists [48]. We call these records *hop bases* and the minimum interval between them *hop distance*, noted as H. Hop encoding employs multiple levels of indirection to speed up the decoding process, with the interval on level L being H^L. Decoding a record involves first tracing back to the nearest hop base in logarithmic time and then following the encoding chain starting with it.

Table 2 summarizes the trade-offs among three encoding techniques in terms of storage usage, worst-case number of retrievals, and the extra number of write-backs. For hop encoding, the number of worst-case source retrievals is close to that of version jumping (H). But because hop bases are stored in an encoded form, the compression ratio achieved is much higher than version jumping and comparable to standard backward encoding. All three encoding schemes incur some amount of write amplification, but the difference becomes negligible as hop distance increases. We present a more detailed comparison in Section 5.

3.3 Caching for Delta-encoded Storage

Delta encoded storage, due to its "chained" property, merits spe-

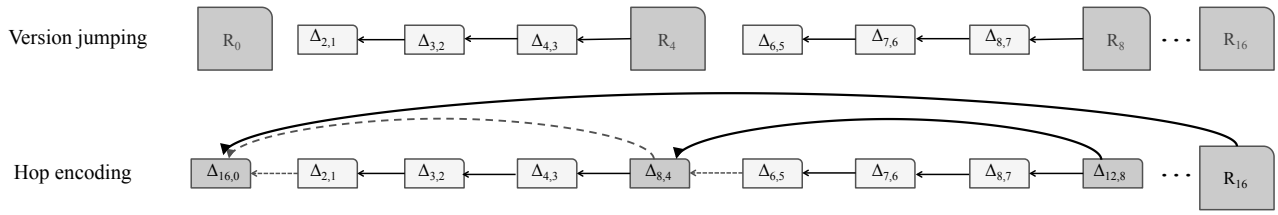


Figure 6: Hop encoding – A comparison of hop encoding and version jumping with an encoding chain of 17 records. Shaded records (R_0, R_4, etc.) are hop bases (reference versions), with a hop distance (cluster size) of 4. Hop encoding provides comparable decoding speed as version jumping while achieving a compression ratio close to standard backward encoding.

cialized caching mechanisms. Exploiting this property, dbDedup only caches a few key nodes in a given encoding chain, maximizing memory efficiency while eliminating most I/O overhead for accessing encoded records. It 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 dbDedup, like in other delta-encoded systems, is the I/O overhead to retrieve the base data from the disk as input for delta compression. Specifically, reading a selected similar record may involve an extra disk access, contending with client query processing and other database activities.

dbDedup uses a small yet effective record cache to avoid most disk reads for source 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 the observations above, the source record cache retains the latest record of an encoding chain in the cache. To accelerate backward encoding of hop bases, dbDedup additionally caches the latest hop bases in each hop level.[2] When a new record arrives, if dbDedup identifies a similar record in the cache (which is the normal case due to the cache-aware selection technique described in Section 3.1), it replaces the existing record with the new one. If the new record is a hop base, dbDedup replaces its adjacent bases accordingly. When no similar source is found, dbDedup simply adds the new record to the cache, and evicts the oldest record in a LRU manner if the cache becomes full.

3.3.2 Lossy Write-back Delta Cache

As discussed in Section 3.2, backward encoding optimizes for read queries, but introduces some write amplification—record insertion triggers the source record to be delta compressed and updated on disk. The problem is exacerbated somewhat with hop encoding, where inserting a hop base causes writeback not only to the source record, but also to the adjacent bases on each hop level. For heavy insertion bursts, this could significantly increase the number of disk writes, leading to visible performance degradation.

dbDedup uses a *lossy write-back cache* to address this problem. The key observation is that write-backs are not strictly required for backward-encoded storage. Failure or delay in applying such write-back operations does not impair data consistency or integrity—updated records remain intact and the only consequence is potential compression loss. This unique "lossy" property provides natural fault tolerance and allows dbDedup great flexibility in scheduling when and in which order writebacks are applied.

On record insertion, dbDedup writes the new record to the database as normal, and stores the delta of the source record in the cache. It delays the actual write-back operation until the system I/O becomes relatively idle. The idleness metric can vary, but we use the I/O queue length as an indication in our current implementation.

To preserve maximum compression with constrained memory, dbDedup sorts deltas in the cache by the absolute amount of space saving they contribute and prioritizes the order of writebacks accordingly. When I/O becomes idle, more valuable deltas are written out first. When the cache becomes full before the system gets idle enough, the entry with the least compression gain is discarded without impacting correctness. By prioritizing the update and eviction orders, dbDedup more effectively reaps the compression benefits from cached deltas.

3.4 Avoiding Unproductive Dedup Work

dbDedup 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, dbDedup 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, dbDedup 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×) 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. dbDedup 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. 7 shows the cumulative distribution 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 ac-

[2] In our experience, the number of hop levels is usually small (≤ 3), so the cache only needs to store very few records for each encoding chain.

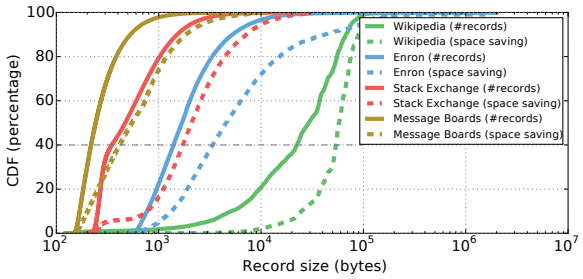


Figure 7: Size-based deduplication filter.

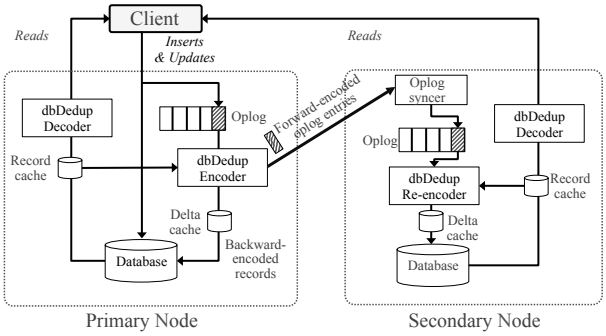


Figure 8: Integration of dbDedup into a DBMS.

count 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.

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

4.1 DBMS Integration

While implementation details vary across DBMSs, we illustrate the integration of dbDedup using a simple distributed setup consisting of one client, one primary node and one secondary node, as shown in Fig. 8. For simplicity, we assume that only the primary node serves write requests[3] and that it pushes updates asynchronously to the secondary node in the form of oplog batches. We now describe dbDedup's behavior for primary DBMS operations.

Insert: The primary node 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 reaches a threshold, the primary sends 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.

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

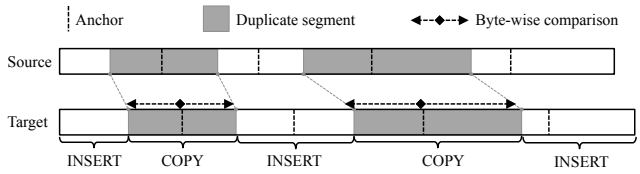


Figure 9: Illustration of delta compression in dbDedup.

With dbDedup, the primary node first stores the new record in its local oplog. Later, when preparing to store the record or send it to a replica, it is processed by the dbDedup encoder following the deduplication steps described in Section 3.1. If dbDedup 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 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. dbDedup 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, dbDedup appends the forward-encoded record to the primary oplog that is transferred to the secondary during replica synchronization.

On the secondary node, the DBMS's oplog syncer receives and propagates the encoded oplog entries to the dbDedup re-encoder. The re-encoder first decodes the new record by reading the base similar record from its local database[4] (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, dbDedup 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.

dbDedup maintains a *reference count* for each stored record that tracks the number of records referencing it as a decode base. Because dbDedup 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, dbDedup first checks the reference count of the queried record. If the count is zero, meaning no other records refer to it for decoding, dbDedup directly applies the update as normal. Otherwise, dbDedup 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 reaches zero, dbDedup compacts all the updates to the record and replaces it with the new data.

dbDedup uses a write-back cache to delay the update of a delta-encoded source record. To prevent it from overwriting normal client updates, dbDedup always checks the cache for each update. If it finds a record with the same ID (to be written back later), it invalidates the entry and proceeds normally with the client update.

Delete: If the reference count for record to be deleted is zero, then the deletion proceeds as normal. Otherwise, dbDedup marks

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

Algorithm 1 Delta Compress

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

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 the reference count of a record drops to zero, dbDedup removes it from the database and decrement the reference count of its base record by one.

Read: If the queried record is stored in its raw form, then it is directly sent to the client just like the normal case. If the record is encoded, then the dbDedup's decoder returns it back to its original form before it is 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.

Garbage Collection: Each record's reference count ensures that an encoding chain will not be corrupted on updates or deletions. To facilitate garbage collection, dbDedup checks for deleted objects on reads. Specifically, along a decoding path, if a record is seen as deleted, dbDedup creates a delta between its two neighboring records, and decrements its reference count by one. When no other records depend on it for decoding, the record can be safely deleted from the database.

4.2 Delta Compression

To ensure lightweight dedup, it is important to make dbDedup's delta compression fast and efficient. The delta compression algorithm used in dbDedup is adapted from xDelta [42], 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 [32] 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.

Algorithm 2 Delta Re-encode

```
1: function DELTAREENCODE(src, tgt, tInsts)
2:     sPos ← 0
3:     tPos ← 0
4:     copySegs ← empty
5:     sInsts ← empty
6:     for each inst in tInsts do
7:         if inst.type = COPY then
8:             copySegs.append(inst.sOff, tPos, inst.len)
9:         end if
10:        tPos ← tPos + inst.len
11:    end for
12:    copySegs.sortBy(sOff)
13:    for each seg in copySegs do
14:        if sPos < seg.sOff then
15:            insInst ← INST(INSERT, sPos, sOff − sPos)
16:            memcpy(insInst.data, src, sOff − sPos)
17:            sInsts.append(insInst)
18:        end if
19:        cpInst ← INST(COPY, seg.tOff, seg.len)
20:        sInsts.append(cpInst)
21:        sPos ← seg.sOff + seg.len
22:    end for
23:    return sInsts
24: end function
```

As shown in Algorithm 1 and Fig. 9, dbDedup's delta compression algorithm is a modified version of xDelta based on the observation that a large fraction of its time is spent in source index building and lookups. In the first encoding step, dbDedup 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. In the second step, dbDedup performs index lookups only for the anchors in the target, avoiding the need to consult the source index at every target offset. The anchor interval provides a tunable trade-off between compression ratio and encoding speed, and we evaluate its effects in Section 5. We omit some optimizations in the pseudo-code given above due to space constraints. 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, dbDedup 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, dbDedup 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 dbDedup 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.

5. EVALUATION

This section evaluates dbDedup using four real-world datasets. For this evaluation, we implemented both dbDedup and traditional chunk-based dedup (trad-dedup) in MongoDB (v3.1). The results show that dbDedup 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 meta-data 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 [62], where the normalized read/write ratio is 99.9 to 0.1. 99.7% of read requests are 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. We extracted a 10 GB subset via random sampling. Most of the duplication in this data set comes from users revising their own posts and from copying answers from other discussion threads. 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

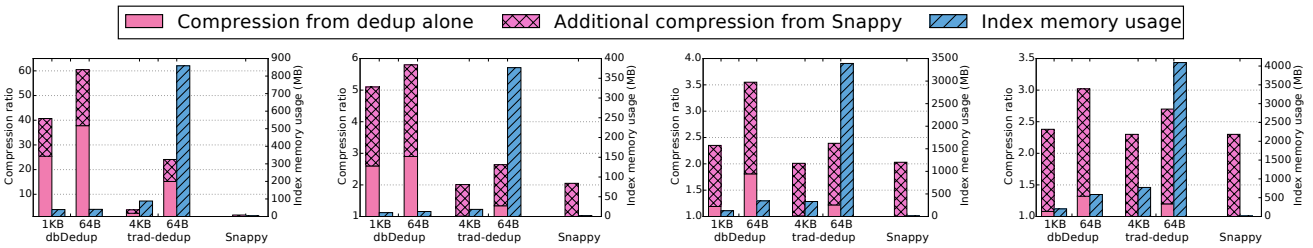
We first evaluate dbDedup's compression ratio and index memory usage and compare them to trad-dedup and Snappy [9], MongoDB's default block-level compressor. For each dataset, we load the records into the DBMS as fast as possible and measure the resulting storage sizes, the amount of data transferred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

Fig. 10 shows the results for five configurations: (1) dbDedup 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 blue (right) bar shows index memory usage. The small source record cache (32 MB, used by both dbDedup and trad-dedup) and lossy write-back cache (8 MB, used by dbDedup only) are not shown.

The benefits are largest for Wikipedia (Fig. 10a). With a chunk size of 1 KB, dbDedup 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 dbDedup does not increase index memory usage much, because dbDedup 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, dbDedup 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: dbDedup 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. 10b), dbDedup 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.[5] The two forum datasets (Figs. 10c and 10d) do not exhibit as much duplica-

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



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

Figure 10: Compression Ratio and Index Memory – The compression ratio and index memory usage for dbDedup (1 KB or 64 byte chunks), trad-dedup (4 KB and 64 byte), and Snappy. The upper portion of each bar represents the added benefit of compressing after dedup.

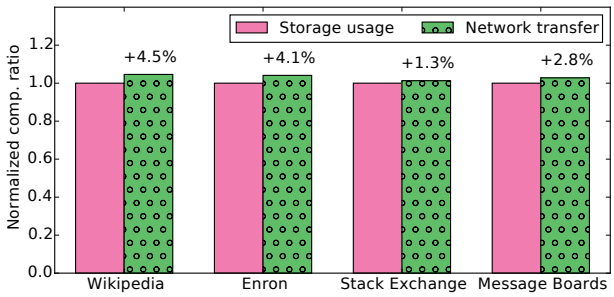


Figure 11: Storage and Network Bandwidth Savings – Relative compression ratios achieved by dbDedup (with 64-byte chunk size) for local storage and network transfer, for each of the datasets, normalized to the absolute storage compression ratios shown in Fig. 10 (for dbDedup with 64-byte chunks).

tion 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 dbDedup 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, dbDedup's compression ratio is conservative, not including the benefits from delta compressing users' revisions to their own posts.[6]

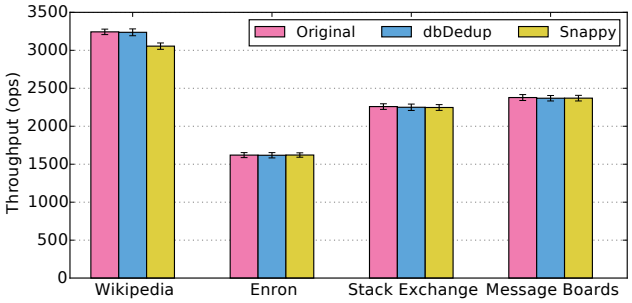
In addition to storage usage, dbDedup simultaneously achieves significant compression on data transmission over the network with forward encoding. Fig. 11 shows the network-level compression as a normalized result to that on storage usage (1.0 on the Y-axis for each dataset). dbDedup achieves slightly lower compression on database storage than on data transferred over the network, mainly due to overlapped encodings (Section 3.2) and delta evictions from the write-back cache. Nevertheless, the difference is below 5% for all datasets, because overlapped encodings are uncommon and because the lossy write-back cache uses prioritized eviction.

5.3 Runtime Performance Impact

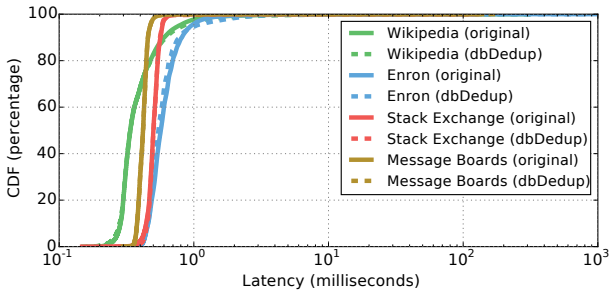
This experiment is designed to measure dbDedup's impact on the DBMS's performance. We compare three MongoDB deployment configurations: (1) No compression ("Original"), (2) dbDedup, and (3) Snappy. For each setting, we run the experiments three times for all the workloads and report the average.

Throughput: Fig. 12a shows the throughput for the four workloads. We see that dbDedup 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

[6]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 12: Performance Impact – Runtime measurements of MongoDB's throughput and latency for the different workloads and configurations.

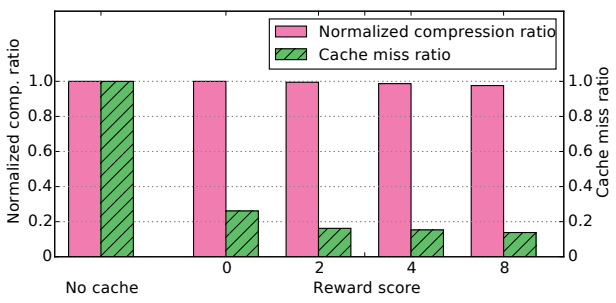
cannot fit in a single WiredTiger page and require extra I/Os.

Latency: Fig. 12b shows the CDF of client latency. For clarity, we only show the results for MongoDB with and without dbDedup enabled. Again, we observe that dbDedup has almost no effect on performance. The latency distribution curves with dbDedup enabled closely track those for no compression/dedup. The difference in the 99.9%-tile latency is less than 1% for all workloads.

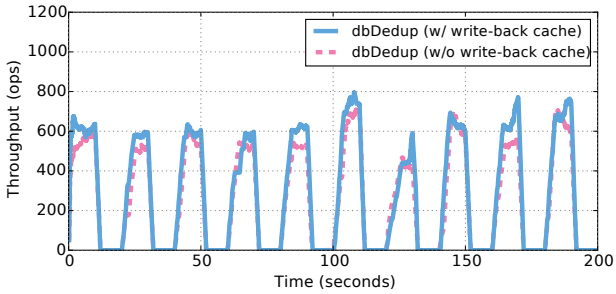
5.4 Effects of Caching

dbDedup uses two specialized 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. 13a 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 dbDedup 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).



(a) Source Record Cache



(b) Lossy Write-back Cache

Figure 13: Effects of Caching – Runtime measurements of dbDedup's caching mechanisms for the Wikipedia workload.

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: dbDedup uses backward encoding to avoid decode when reading the latest "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. dbDedup'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. 13b 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).

dbDedup 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.

5.5 Hop Encoding

dbDedup uses hop encoding to reduce the worst-case retrieval

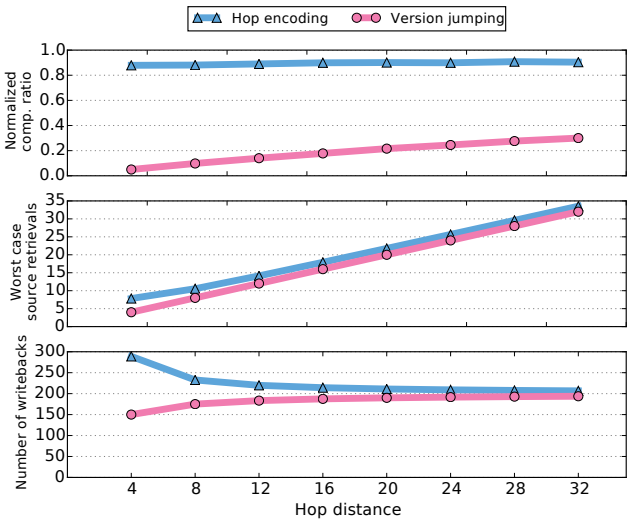


Figure 14: Hop Encoding vs. Version Jumping – For the Wikipedia workload and moderate hop distances, hop encoding provides much higher compression ratios with small increases in worst-case source retrievals and number of write-backs.

times while maintaining compression benefits. To evaluate its efficacy, we also implemented version jumping in MongoDB and compared the two encoding schemes.

Fig. 14 shows the results for three metrics as a function of hop distance: compression ratio (normalized to standard backward encoding), worst-case number of source retrievals (for an encoding chain length of 200), and number of write-backs. Version jumping results in significantly (60–90%) lower compression ratios, because all reference versions are stored unencoded. Its compression ratio improves as the hop distance increases, because fewer records are stored in unencoded form. In contrast, because hop bases are stored as deltas, hop encoding provides compression ratios within 10% of full backward encoding. For hop encoding, the compression ratio remains relatively steady as hop distance increases, due to having fewer but less similar hop bases.

The number of worst-case source retrievals for hop encoding is close to that for version jumping. With multiple hop levels, tracing back to the nearest hop base only takes logarithmic time. As the hop distance increases, the decoding time is dominated by traversing backward deltas between adjacent hop bases. The bottom graph shows the number of extra writebacks needed in each scheme. While hop encoding incurs more writebacks for small hop distances, both schemes quickly approach the length of the encoding chain as hop distance increases. Empirically, we find that a hop distance of 16 (default) provides a good trade-off between compression ratio and decoding overhead.

5.6 Optimization of Delta Compression

dbDedup 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. 15 shows the compression ratio (left Y-axis) and throughput (right Y-axis) for various anchor interval values, as a comparison with xDelta, for the Wikipedia workload. With an anchor interval of 16 (the default window size in xDelta), dbDedup performs almost the same as xDelta. dbDedup's delta compressing speed improves as anchor interval increases, because it reduces the number of insertions and lookups in the source offset index . The com-

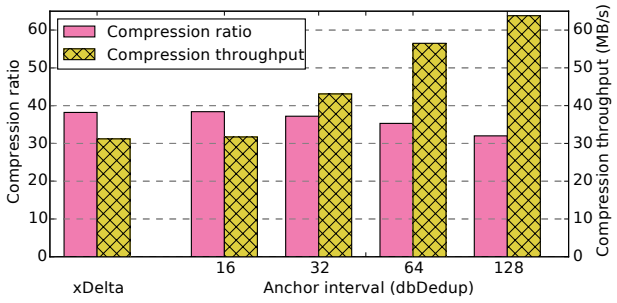


Figure 15: Optimization of Delta Compression – Comparison of the optimized variant of dbDedup versus xDelta using the Wikipedia workload.

pression ratio does not significantly decrease, because dbDedup performs byte-level comparison bidirectionally from the matched points. With an anchor interval of 64, dbDedup 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 reasonable 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 have been proposed during the past few decades. Most operational DBMSs that compress the database contents use page or block-level compression [30, 37, 46, 43, 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 dbDedup approach, such compression requires the DBMS to decompress tuples before query processed.

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, 36, 69, 51]. 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 [56] 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. dbDedup, in contrast, is able to remove much more fine-grained duplicates with byte-level delta compression. Unlike other inline compression schemes, dbDedup 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 dbDedup 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 Encoding: There has been a large body of previous work on delta encoding techniques, including several general-purpose algorithms based on the Lempel-Ziv approach [68], such as vcdiff [21], xDelta [42], and zdelta [60]. Specialized schemes can be used for specific data formats (e.g., XML) to improve compression quality [28, 63, 39, 52]. The delta compression algorithm used in dbDedup 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 [61, 57]. On exception is TAPER [38], 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. dbDedup 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 dbDedup is inspired by versioned storage systems such as RCS [59] and XDFS [42]. Similar techniques have been used in version control systems such as Git [41] and SVN [29] to enable traveling back over the commit history. Unlike these systems which explicitly maintain versioning lineage for all the files, dbDedup establishes the encoding chain entirely based on the similarity relationships between records, and thus does not require system-level support for versioning. [54] uses delta encoding for deduplicated backup storage. It uses forward encoding and only allows encoding chains with a maximum length of two items. Similar to other delta encoded storage systems that uses version jumping, it has to sacrifice compression gains in order to bound the worst-case retrievals for base data.

To our knowledge, dbDedup is the first system to explore distinct forms of encoding for network and storage-level compression and to provide efficient transformations between the two. With the novel hop encoding scheme, dbDedup significantly alleviates the painful trade-offs between compression ratio and retrieval overhead for delta encoded storage. In addition, it introduces new caching mechanisms specialized for delta encoded storage, significantly reducing the I/O overhead involved while maximizing memory efficiency. All the techniques combined make online access of encoded storage practical.

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 [58, 47, 20, 53, 65] propose methods for computing sketches for similarity detection that are robust to small edits in the data. The feature extraction approach used in dbDedup is similar to that in DOT [47] and sDedup [65].

7. CONCLUSION

dbDedup 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, dbDedup 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 dbDedup 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.

Acknowledgements: We would like to thank Keith Bostic for his guidance on the internals of WiredTiger. We thank the anonymous reviewers for helpful feedback on the paper. We also thank the members and companies of the PDL Consortium (including Broadcom, Citadel, Dell EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Huawei, Intel, Microsoft Research, NetApp, Oracle, Samsung, Seagate, Tintri, Two Sigma, Uber, Veritas and 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 by MongoDB Incorporated. Experiments were enabled by generous hardware donations from Intel and NetApp.

8. 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] B. Collins-Sussman, B. Fitzpatrick, and M. Pilato. Version control with subversion. 2004.

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

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

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

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

[34] 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.

[35] 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.

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

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

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

[39] 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.

[40] 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.

[41] J. Loeliger. Version control with git: Powerful tools and techniques for collaborative software development. 2009.

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

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

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

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

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

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

[48] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449. Springer, 1989.

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

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

[51] 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.

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

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

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

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

[56] 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.

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

[58] 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.

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

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

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

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

[63] 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.

[64] 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.

[65] 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.

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

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

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

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