Reducing Replication Bandwidth for Distributed Document Databases

Lianghong Xu[⋆] Andrew Pavlo[⋆] Sudipta Sengupta[†] Jin Li[†] Gregory R. Ganger[⋆]

Carnegie Mellon University[⋆], Microsoft Research[†]

Abstract

With the rise of large-scale, Web-based applications, users are increasingly adopting a new class of document-oriented database management systems (DBMSs) that allow for rapid prototyping while also achieving scalable performance. Like for other distributed storage systems, replication is important for document DBMSs in order to guarantee availability. The network bandwidth required to keep replicas synchronized is expensive and is often a performance bottleneck. As such, there is a strong need to reduce the replication bandwidth, especially for geo-replication scenarios where wide-area network (WAN) bandwidth is limited.

This paper presents a deduplication system called *sDedup* that reduces the amount of data transferred over the network for replicated document DBMSs. sDedup uses *similarity-based deduplication* to remove redundancy in replication data by delta encoding against similar documents selected from the entire database. It exploits key characteristics of document-oriented workloads, including small item sizes, temporal locality, and the incremental nature of document edits. Our experimental evaluation of sDedup with three real-world datasets shows that it is able to achieve up to 38× reduction in data sent over the network, significantly outperforming traditional chunk-based deduplication techniques while incurring negligible performance overhead.

1. Introduction

Document-oriented databases are becoming more popular due to the prevalence of semi-structured data. The document model allows entities to be represented in a schemaless manner using a hierarchy of properties. Because these DBMSs are typically used with user-facing applications, it is important that they are always on-line and available. To ensure this availability, these systems replicate data across nodes with some level of diversity. For example, the DBMS could be configured to maintain replicas within the data center (e.g., nodes on different racks, different clusters) or across data centers in geographically separated regions.

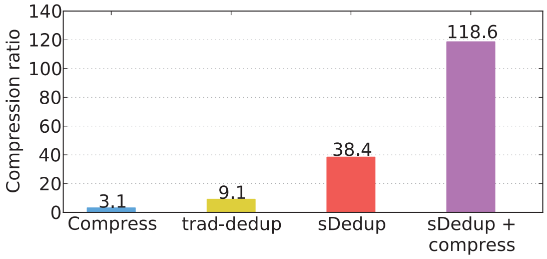


Figure 1: Compression ratios for Wikipedia – The four bars represent compression ratios achieved for the Wikipedia dataset (see Section 5) for four approaches: (1) standard compression on each oplog batch (4 MB average size), (2) traditional chunk-based dedup (256 B chunks), (3) our system that uses similarity-based dedup, and (4) similarity-based dedup combined with compression.

Such replication can require significant network bandwidth, which becomes increasingly scarce and expensive the farther away the replicas are located from their primary DBMS nodes. It not only imposes additional cost on maintaining replicas, but can also become the bottleneck for the DBMS's performance if the application cannot tolerate significant divergence across replicas. This problem is especially onerous in geo-replication scenarios, where WAN bandwidth is expensive and capacity grows relatively slowly across infrastructure upgrades over time.

One approach to solving this problem is to compress the operation log (oplog) that is sent from the primary DBMS nodes to the replicas for synchronization. For text-based document data, simply running a standard compression library (e.g., *gzip*) on each oplog batch before transmission will provide approximately a 3× compression ratio. But higher ratios are possible with *deduplication* techniques that exploit redundancy with data beyond a single oplog batch. For a workload based on Wikipedia, as shown in Fig. 1, an existing deduplication approach achieves compression up to 9× while our proposed similarity-based deduplication scheme is able to compress at 38×. Moreover, these ratios can be combined with the 3× from compression, yielding ∼120× reduction for our proposed approach.

Most deduplication systems [20, 22, 29, 39, 41, 47] target backup streams for large-scale file systems and rely upon several properties of these workloads. Foremost is that backup files are large and changes affect an extremely small portion of the data. This argues for using large chunks to

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA
© 2015 ACM. ISBN 978-1-4503-3651-2/15/08. . . $15.00
DOI: http://dx.doi.org/10.1145/2806777.2806840

	#Primaries	Direction	Consistency	Update Form
CouchDB	single/multi	push/pull	async/sync	doc revisions
MongoDB	single	pull	async/sync	oplogs
RavenDB	single/multi	push	async	doc revisions
RethinkDB	single	push	async/sync	change feeds

Table 1: Key replication features of four document DBMSs.

avoid the need for massive dedup indices; the trad-dedup bar in Fig. 1 ignores this issue and shows the result for a 256 B chunk size. With a typical 4 KB chunk size, trad-dedup achieves a 2.3× compression ratio. Second, these systems assume that good *chunk locality* exists across backup streams, such that chunks tend to appear in roughly the same order in each backup cycle. This allows for efficient prefetching of dedup metadata during the deduplication and reconstruction processes.

In our experience, the workloads for document database applications do not exhibit these characteristics. Instead of large blocks of data corresponding to the same entity (e.g., backup stream), documents are small, with an average size less than 100 KB [3]. The replication streams do not exhibit much chunk locality. They instead have *temporal locality* where frequent updates to specific documents happen within a short interval of time. The scope of these modifications is small relative to the original size of the document but often distributed throughout the document. As we will show in this paper, these differences make traditional deduplication approaches a poor match for document DBMSs.

We present the design and implementation of a deduplication system, called *sDedup*, that exploits the characteristics of document databases. Unlike many traditional deduplication systems that employ dedicated servers, sDedup is a lightweight module that can be integrated into the replication framework of an existing DBMS with minimal software complexity. sDedup achieves an excellent compression ratio while imposing negligible impact on performance and memory overhead. It uses consistent sampling of chunk hashes combined with a fast and compact Cuckoo hash table to reduce the metadata needed to keep track of documents in the corpus. sDedup uses a source document cache and a variant of the *xDelta* algorithm [30] to minimize the CPU and I/O overhead in delta compressing similar documents. This approach, called *similarity-based deduplication*, does not require exact duplicates to eliminate redundancies.

This paper makes three contributions: Foremost, to the best of our knowledge, we are the first to use similarity-based deduplication techniques to reduce the replication bandwidth in a distributed DBMS. Second, we present the general-purpose end-to-end workflow of sDedup, as well as design choices and optimizations that are important for using inline deduplication in document databases. Third, we integrate sDedup into the replication framework of the MongoDB DBMS [2]; our evaluation shows that it achieves higher compression rates and less memory overhead than the chunk-based deduplication approach, while having almost no impact on the DBMS's runtime performance.

The rest of this paper is organized as follows. Section 2 provides an overview of why existing approaches to reducing replication bandwidth are insufficient for document DBMSs. Section 3 describes the sDedup deduplication workflow. Section 4 details sDedup's implementation and integration into a document DBMS. Section 5 evaluates sDedup on real-world datasets and explores sensitivity to key configuration parameters. We present a survey of related work in Section 6 and conclude in Section 7.

2. Background and Motivation

This section discusses replication mechanisms of document databases in general, why reducing network bandwidth usage for DBMS replication is desirable, and motivates the need for an approach using similarity-based deduplication.

2.1 Replication in Document Databases

Distributed document DBMSs exploit replication to enhance data availability. The design and implementation details of the replication mechanisms vary for different systems; we summarize the key replication features for the leading document DBMSs in Table 1. Depending on the application's desired trade-off between the complexity of conflict resolution and the peak write throughput, there can be a single or multiple primary nodes that receive user updates. Replica synchronization can be initiated by either the primary (push) or the secondary nodes (pull). How often this synchronization should occur depends on the application's "freshness" requirement of the reads that the secondary nodes serve, and/or how up-to-date the replica should be upon failover, and is specified by the consistency model. Application of updates to secondary nodes can happen either in real-time with the user update (sync), or be delayed by some amount (async).

Document database replication involves propagating replication data from the primary to the secondary nodes in the form of document updates. A common way of doing this is by sending over the database's write-ahead log, also sometimes referred to as its operation log (oplog), from the primary to the secondary. The secondary node then replays the log to update the state of its copy of the database. We show in Section 4.1 that sDedup can be integrated to a DBMS's replication framework with minimal software complexity.

2.2 Network Bandwidth for Replication

The network bandwidth needed for replica synchronization is directly proportional to the volume and rate of updates happening at the primary. When the network bandwidth is not sufficient, it can become the bottleneck for replication performance and even end-to-end client performance for write-heavy workloads.

The data center hierarchy provides increasingly diverse levels of uncorrelated failures, from different racks and clusters within a data center to different data centers. Placing replicas at different locations is desirable for increasing the availability of cloud services. But network bandwidth is more restricted going up the network hierarchy, with WAN bandwidth across regional data centers being the most costly, limited, and slow-growing over time. Reducing the cross-replica network bandwidth usage allows services to use more diverse replicas at comparable performance without needing to upgrade the network.

All major cloud service providers have to deal with WAN bandwidth bottlenecks. Some real-world examples include the MongoDB Management Service (MMS) [3] that provides continuous on-line backups using oplog replication and Google's B4 Software Defined Network system [26]. More generally, there are many applications that replicate email, message board, and social networking application data sets. All of these systems have massive bandwidth requirements that would significantly benefit from lower network bandwidth usage.

2.3 The Need for Similarity-based Deduplication

There has not been much previous work on network-level deduplication in the database community. There are three reasons for this: first, database objects are small compared to files or backup streams. Thus, deduplication may not provide a good compression ratio without maintaining excessively large indexes. Second, for relational DBMSs, especially for those using column-based data stores, simple compression algorithms are good enough to provide a satisfactory compression ratio. Third, the limitation of network bandwidth had not been a critical issue before the advent of replicated services in the cloud (especially geo-replication).

We contend that the emergence of hierarchical data center infrastructures, the need to provide increased levels of reliability on commodity hardware in the cloud, and the popularity of document-oriented databases has changed the operational landscape. More of the data that is generated with today's applications fits naturally or can be converted to *documents*, the central concept in a document-oriented database. Document-oriented databases allow greater flexibility to organize and manipulate these datasets, mostly represented in the form of text data (e.g., wiki pages, emails, blogs/forums, tweets, service logs). Even small updates to text data cannot be easily expressed as incremental operations. As a result, a document update typically involves reading the current version and writing back a highly similar document. Newly created documents may also be similar to earlier documents with only a small fraction of the content changed. Such redundancy creates great opportunity in data reduction for replication.

Past work has explored different ways of removing redundant data for various applications. These techniques are generally categorized into compression and deduplication. We explain below why similarity-based deduplication is the most promising approach for document databases.

Compression alone is insufficient: Updates in replicated databases are sent in batches to amortize the cost of transmission over the network. In order to keep the secondary nodes reasonably up-to-date so that they can serve client read requests for applications that require bounded-staleness guarantees, the size of the oplog batch is usually on the order of MBs. At this small size, the oplog batch mostly consists of updates to unrelated documents, thus intra-batch compression yields only a marginal reduction.

To demonstrate this point, we loaded a Wikipedia dataset into a modified version of MongoDB that compresses the oplog with *gzip*. We defer the discussion of our experimental setup until Section 5. The graph in Fig. 1 shows that compression only reduces the amount of data transferred from the primary to the replicas by 3×. Further reduction is possible using a technique from the file system community known as deduplication, but this technique has different drawbacks in the document database context.

Limitation of chunk-identity based deduplication: Deduplication is a specialized compression technique that eliminates duplicate copies of data. It has some distinct advantages over simple compression techniques, but suffers from high maintenance costs. For example, the "dictionary" in traditional deduplication schemes can get large and thus require specialized indexing methods to organize and access it. Each indexed item in the dictionary is a relatively large byte block (KBs), whereas for simple compression it is usually a short string (bytes).

A traditional file deduplication scheme [33, 36, 47] works as follows. An incoming file (corresponding to a document in the context of document DBMSs) is first divided into chunks using Rabin-fingerprinting [37]; 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, a collision-resistant hash (e.g., SHA-1) is calculated as its identity, which is then looked up in a global index table. 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 underlying data store).

There are two key aspects of document databases that distinguish them from traditional backup or primary storage workloads. First, most duplication exists among predominantly small documents. These smaller data items have a great impact on the choice of chunk size in a deduplication system. For primary or backup storage workloads, where most deduplication benefits come from large files ranging from 10s of MBs to 100s of GBs [22, 32, 45], using a chunk size of 8–64 KB usually strikes a good balance between deduplication quality and the size of chunk metadata indexes. This does not work well for database applications, where object sizes are mostly small (KBs). Using a large chunk size may lead to a significant reduction in deduplication quality. On the other hand, using a small chunk size and

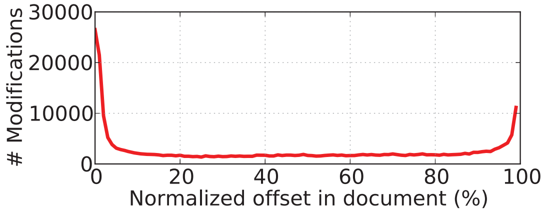
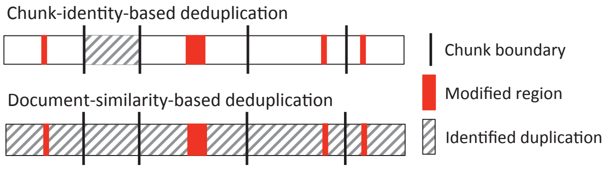


Figure 2: Distribution of document modifications for Wikipedia.



Chunk-identity-based deduplication
Document-similarity-based deduplication
Chunk boundary
Modified region
Identified duplication

Figure 3: Comparison between chunk-identity-based and document-similarity-based deduplication approaches.

building indexes for all the unique chunks imposes significant memory and storage overhead, which is infeasible for an inline deduplication system. sDedup uses a small (configurable) chunk size of 256 B or less, and indexes only a subset of the chunks that mostly represent the document for purposes of detecting similarity. As a result, it is able to achieve more efficient memory usage with small chunk sizes, while still providing a high compression ratio.

The second observation is that updates to document databases are usually small (10s of bytes) but dispersed throughout the document. Fig. 2 illustrates this behavior by showing the distribution of modification offsets in the Wikipedia dataset. Fig. 3 illustrates the effect of this behavior on the duplicated regions identified for the similarity-based and chunk-based deduplication approaches. For the chunk-based approach, when the modifications are spread over the entire document, chunks with even slight modifications are declared as unique. Decreasing the chunk size alleviates this problem, but incurs higher indexing overhead. In contrast, sDedup is able to identify all duplicate regions with the same chunk size. It utilizes a fast and memory-efficient similarity index to identify similar documents, and uses a byte-by-byte delta compression scheme on similar document pairs to find the duplicate byte segments.

We focus on textual data because it is emblematic of document DBMS workloads. It is important to note, however, that our approach is applicable to non-textual data as well. Specifically, given a target object, the same dedup index in sDedup can be used to identify similar objects in the corpus. Nevertheless, whether or not to perform delta compression largely depends on the characteristics of the target workloads. While delta compression is a good choice for relatively small semi-structured data (like text) with dispersed modifications, it might not be best for large BLOBs with sparse changes due to the greater I/O and computation overheads that could be involved. In this scenario, as discussed above, the chunk-based deduplication approach may suffice to provide a reasonably good compression ratio.

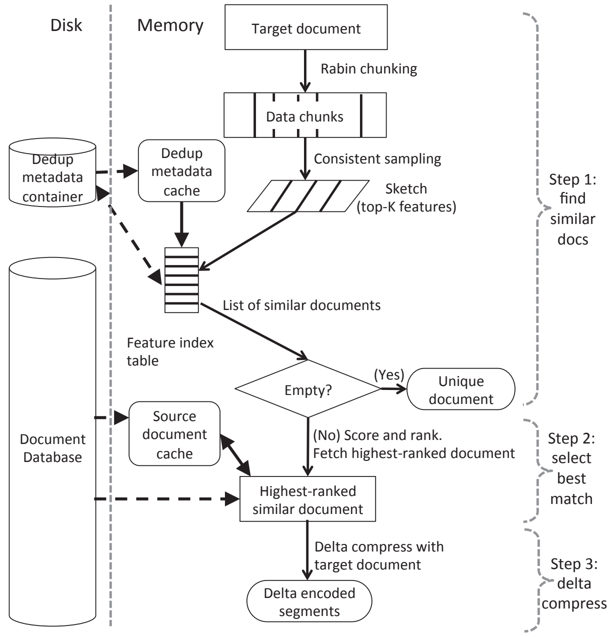


Figure 4: sDedup Workflow and Data Structures – A target (input) document is converted to a delta-encoded form in three steps. On the left are the two disk-resident data stores involved in this process: the dedup metadata container (see Section 4.2) and the original document database.

3. Dedup workflow in sDedup

We now describe the workflow of sDedup, our similarity-based deduplication system. It differs from chunk-based deduplication systems that break the input data-item into chunks and find identical chunks stored anywhere else in the data corpus (e.g., the original database). sDedup's workflow has three steps: (1) finding documents in the corpus that are similar to the target document, (2) selecting one of the similar documents to use as the deduplication source, and (3) performing differential compression of the target document against the source document. Fig. 4 illustrates the data structures and actions involved in transforming each target document into a delta-encoded representation. The remainder of this section describes this process in further detail.

3.1 Finding Candidate Source Documents

sDedup's approach to finding similar documents in the corpus is illustrated in Algorithm 1. The target document is first divided into variable-sized data chunks using the Rabin fingerprinting algorithm [37] that is commonly used in traditional deduplication approaches. For each chunk, sDedup computes its unique 64-bit hash using MurmurHash [4].[1] It then computes a *sketch* of the target document composed of

[1] We use MurmurHash instead of the stronger 160-bit cryptographic SHA-1 hash used in traditional deduplication because we only use these hashes to identify similar documents rather than for chunk-level deduplication. This reduces the computation overhead at the cost of a higher hash collision rate, but it does not impair correctness since we perform delta compression in the final step.

Algorithm 1 Finding Similar Documents

```
procedure FindSimilarDocs($tgtDoc$)
    $i \leftarrow 0$
    $sketch \leftarrow empty$
    $candidates \leftarrow empty$

    $dataChunks \leftarrow$ RabinFingerprint($tgtDoc$)
    $chunkHashes \leftarrow$ MurmurHash($dataChunks$)
    $uniqueHashes \leftarrow$ Unique($chunkHashes$)
    $sortedHashes \leftarrow$ Sort($uniqueHashes$)
    $sketchSize \leftarrow$ Min($K, sortedHashes$.size())
    while $i < sketchSize$ do
        $feature \leftarrow sortedHashes[i]$
        $sketch$.append($feature$)
        $simDocs \leftarrow$ IndexLookup($feature$)
        $candidates$.append($simDocs$)
        $i \leftarrow i + 1$
    end while
    for each $feature$ in $sketch$ do
        IndexInsert($feature, tgtDoc$)
    end for
    return $candidates$
end procedure
```

Algorithm 2 Selecting the Best Match

```
procedure SelectBestMatch($candidates$)
    $scores \leftarrow empty$
    $maxScore \leftarrow 0$
    $bestMatch \leftarrow NULL$
    for each $cand$ in $candidates$ do
        if $scores[cand]$ exists then
            $scores[cand] \leftarrow scores[cand] + 1$
        else
            $scores[cand] \leftarrow 1$
        end if
    end for
    for each $cand$ in $scores$.keys() do
        if $cand$ in srcDocCache then
            $scores[cand] \leftarrow scores[cand] + reward$
        end if
        if $scores[cand] > maxScore$ then
            $maxScore \leftarrow scores[cand]$
            $bestMatch \leftarrow cand$
        end if
    end for
    return $bestMatch$
end procedure
```

a subset of its chunk hashes. The sketch consists of the top-K hash values (which we call *features*) sorted in a consistent manner, such as by magnitude.[2] This consistent sampling approach has been shown to be an effective way to characterize a data-item's content in a small bounded space [35].

Next sDedup checks to see whether each feature exists in its internal feature index (see Section 4.2). If a document has at least one feature in common with the target document, it is considered "similar" and added to the list of candidate sources. The feature index stores at most K entries for each document in the corpus (one for each feature). As a result, the size of this index is smaller than the corresponding index for traditional deduplication systems, which must have an entry for every unique chunk in the system. The value of K is a configurable parameter that trades off resource usage for similarity metric quality. Generally, a larger K yields better similarity coverage, but leads to more index lookups and memory usage. In practice, a small value of K is good enough to identify moderately similar pairs with a high probability [35]. For our experimental analysis, we found $K = 8$ is sufficient to identify similar documents with reasonable memory overhead. We explicitly evaluate the impact of this parameter on sDedup's performance in Section 5.5.

3.2 Selecting the Best Source Document

After sDedup identifies a list of candidate source documents, it next selects one of them to use. If no similar documents are found, then the target document is declared unique and thus is not eligible for encoding. Algorithm 2 describes the mechanism sDedup uses to choose the best source document out of a number of similar documents. Fig. 5 provides an example of this selection process.

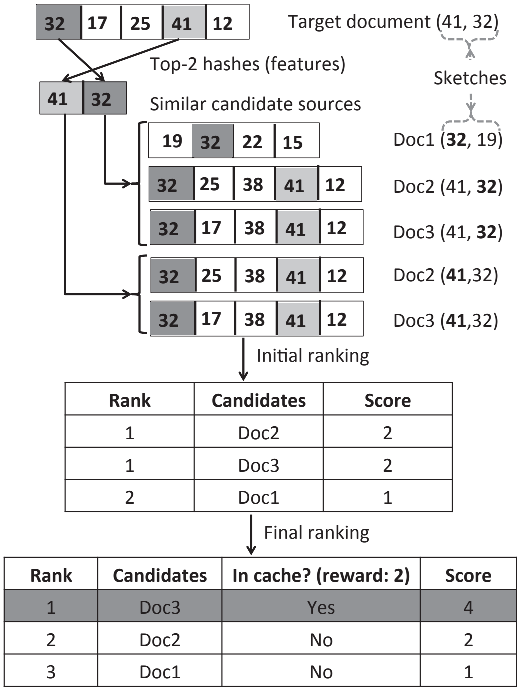
The system first assigns each candidate an initial score, which is the number of similarity features that the candidate has in common with the target document. It then ranks all the source candidates by this score from high to low. To break ties, newer documents get higher ranks. This decision is based on the empirical observation that newer documents are usually better choices.

While most previous similarity selection methods [12, 15, 25, 29, 39] rank similar objects merely by similarity properties, sDedup takes into consideration the end-to-end system constraints and gives preference to documents residing in the source document cache (see Section 4.3). After the initial ranking, the base score is adjusted upward by a reward (two, by default) if the candidate is present in the cache. This indicates that sDedup does not need to retrieve the document from corpus database for delta compression. Although this reward may result in a less similar document being selected, it improves the hit ratio of the source document cache and thus reduces the I/O overhead to retrieve the source document. We call this technique "cache-aware selection" and evaluate its effectiveness in Section 5.5.

3.3 Delta Compression

In the final step, sDedup delta compresses the target document against the selected source document. The system first checks its internal document cache for the source document; on miss, it retrieves the document from the database. sDedup uses only one source document for delta compression. We found that using more than one is not only unnecessary (i.e., it does not produce a better compression), but also greatly increases the overhead. In particular, we found that fetching the source document from the corpus is the dominating factor in this step, especially for the databases with small documents. This is the same reasoning that underscores the benefits of sDedup over chunk-based deduplication: our approach only requires one fetch per target document to reproduce the original target document, versus one fetch per chunk.

[2] For documents with less than K chunks, the sketch size is less than K.



Rank	Candidates	Score
1	Doc2	2
1	Doc3	2
2	Doc1	1

Rank	Candidates	In cache? (reward: 2)	Score
1	Doc3	Yes	4
2	Doc2	No	2
3	Doc1	No	1

Figure 5: Example of Source Document Selection – The top two ($K = 2$) hashes of the target document are used as the features of its sketch (41, 32). The numbers in the documents' chunks are the MurmurHash values. Documents with each feature are identified and initially ranked by their numbers of matching features. The ranking increases if the candidate is in sDedup's cache.

The delta compression algorithm used in sDedup is similar to rsync [44] and based on xDelta [30], but reduces the computation overhead with minimal compression loss. sDedup first calculates hash values for a sliding window on the source document in a manner similar to Rabin fingerprinting. It then builds a temporary index that maps the hash values to the offsets within the source document. To reduce the overhead of building this index, sDedup uses fixed sampling to index only offsets at fixed intervals. Because the matching granularity is at the byte level, the compression loss is negligible when the sampling interval is much smaller than the document size.

After sDedup builds its source index, it calculates the hash for each offset of the target document using the same sliding-window approach and looks up the hash in the source index. When no match is found, the sliding window moves forward by one byte and calculates the hash for the next offset. Otherwise, sDedup compares the source and target documents from the matching points byte-by-byte in both forward and backward directions. This process continues until it finds the longest match on both ends, which determines the boundaries between unique and duplicate segments. The next index lookup skips the offsets covered by the duplicate segments and starts from the beginning of the new segment.

The encoded output is a concatenated byte stream of all unique segments and an ordered list of segment descriptors, each specifying the segment type and offset in the source document or the unique bytes. The sDedup instance on the secondary node decompresses the message by iterating over the segment descriptors and concatenating the duplicate and unique segments to reproduce the original document.

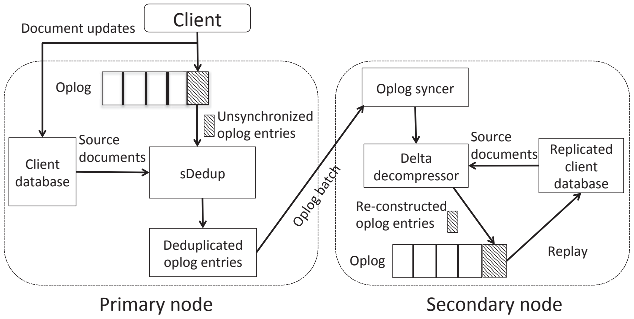


Figure 6: Integration of sDedup into a document DBMS.

4. Implementation

We next describe the implementation details of sDedup, including how it fits into the replication frameworks of document DBMSs, as well as the internals of its indexing mechanisms and the source document cache.

4.1 Integration into Document DBMSs

sDedup is a lightweight module that can be integrated into the replication frameworks of existing DBMSs. While the implementation details vary for different systems, we illustrate the integration using a typical setting with single-master, push-based, asynchronous replication that propagates updates in the form of oplogs, as shown in Fig. 6. We then describe how such integration is generally applicable to other replication settings with slight modifications.

An oplog is maintained at the primary and secondary nodes for replication and recovery. Each client write request is applied to the primary node's local database and appended to its oplog. Each oplog entry includes a timestamp and a payload that contains the inserted/modified documents. The primary pushes updates to the secondaries periodically or when the size of unsynchronized oplog entries exceeds a given threshold. The updates consist of a batch of oplog entries with timestamps later than the last synchronization checkpoint. Normally these entries are sent in their native form. When a secondary receives the updates, it appends the oplog entries to its local oplog, so that its oplog replayer can apply them to the local copy of the database.

With sDedup, before an oplog entry is queued up in a batch to be sent, it is first passed to the deduplication subsystem and goes through the steps described in Section 3. If the entry is marked for deduplication, then it is appended to the batch as a special message the sDedup receiver on the secondary knows how to interpret. When the secondary node receives the encoded data, it reconstructs each entry into the original oplog entry and appends it to its local oplog. At this point the secondary oplog replayer applies the entry to its database just as if it was a normal operation. Thus, sDedup is not involved in the critical write path of the primary and is only used to reduce the replication bandwidth instead of the storage overhead of the actual database.

sDedup's replication protocol is optimistic in that it assumes that the secondary will have the source document for each oplog entry available locally. When this assumption holds, no extra round trips are involved. In the rare cases when it does not (e.g., a source document on the primary gets updated before the corresponding oplog entry is deduplicated), the secondary sends a supplemental request to the primary to fetch the original unencoded oplog entry, rather than the source document. This eliminates the need to reconstruct documents when bandwidth savings are not being realized. In our evaluation in Section 5 with the Wikipedia dataset, we observe that only 0.05% of the oplog entries incur a second round trip during replication.

We next describe sDedup's protocol for other replication mechanisms. When there are multiple primary servers, each of them maintains a separate deduplication index. The index is updated when a primary either sends or receives updates to/from the other replicas. Eventually all the primaries will have the same entries in their deduplication indexes through synchronization. When secondaries independently initiate synchronization requests (pull), the primary does not add an oplog entry's features to its index until all secondaries have requested that entry. Because the number of unsynchronized oplog entries is normally small, the memory overhead of keeping track of the secondaries' synchronization progress is negligible. sDedup supports both synchronous and asynchronous replication because it is orthogonal to the consistency setting. We show in Section 5 that sDedup has little impact on performance with eventual consistency or bounded-staleness. For applications needing strict consistency where each write requires an acknowledgement from all replicas, sDedup currently imposes a minor degradation (5–15%) on throughput. In practice, however, we believe that strict consistency is rarely used in geo-replication scenarios where sDedup provides the most benefits.

4.2 Indexing Documents by Features

An important aspect of sDedup's design is how it finds similar documents in the corpus. Specifically, given a feature of the target document, sDedup needs to find the previous documents that contain that feature in their sketches. To do this efficiently, sDedup maintains a special index that is separate from the other indexes in the database.

To ensure fast deduplication, sDedup's feature lookups must be primarily in-memory operations. Thus, the size of the index is an important consideration since it consumes memory that could otherwise be used for database indexes and caches. A naïve indexing approach is to store an entry that contains the document's "dedup metadata" (including its sketch and database location) for each feature. In our implementation, the database location for each document is encoded with a 52 B database namespace ID and a 12 B document ID. Combined with the 64 B sketch, the total size of each dedup metadata entry is 128 B.

To reduce the memory overhead of this feature index, sDedup uses a two-level scheme. It stores the dedup metadata in a log-structured disk container and then uses a variant of Cuckoo hashing [34] to map features to pointers into the disk container. Cuckoo hashing allows multiple candidate slots for each key, using a number of different hashing functions. This increases the hash table's load factor while bounding lookup time to a constant. We use 16 random hashing functions and eight buckets per slot. Each bucket contains a 2 B compact checksum of the feature value and a 4 B pointer to the dedup metadata container. As a result, sDedup only consumes 6 B per index entry.

For each feature in the target document's sketch, the lookup and insertion process works as follows. First, the system calculates a hash of the feature starting with the first (out of 16) Cuckoo hashing function. The candidate slot in the Cuckoo hash table is obtained by applying a modulo operation to the lower-order bits of the hash value; the higher-order 16 bits of the hash value is used as the checksum for the feature. Then, the checksum is compared against that of each occupied bucket in the slot. If a match is found, then sDedup retrieves the dedup metadata using the pointer stored in the matched bucket. If the document's dedup metadata contains the same feature in its sketch, it is added to the list of similar documents. The lookup then continues with the next bucket. If no match is found and all the buckets in the slot are occupied, the next Cuckoo hashing function is used to obtain the next candidate slot. The lookup process repeats and adds all matched documents to the list of similar documents until it finds an empty bucket, which indicates that there are no more matches. At this point, an entry for the feature is inserted into the empty bucket. If no empty bucket is found after iterating with all 16 hashing functions, we randomly pick a victim bucket to make room for the new feature, and re-insert the victim into the hash table as if it was new.

The size and load on the Cuckoo hash table can be further reduced by specifying an upper bound on the number of similar documents stored for any given feature. For instance, with a setting of four, lookup for a given feature stops once it finds a fourth match. In this case, insertion of an entry for the target document will require first removing one of the other four matches from the index. We found that evicting the least-recently-used (LRU) document for the given feature is the best choice. Because the LRU entry could be early in the lookup process, all of the matching entries would be removed and reinserted as though they were new entries.

sDedup uses a small dedup metadata cache to reduce the number of reads to the on-disk dedup metadata container [20, 39]. The container is divided into contiguous 64 KB pages, each containing 512 dedup metadata entries. Upon checksum matches, sDedup fetches an entire page of dedup metadata into the cache and adds it to a LRU list of cache pages. The default configuration uses 128 cache pages (8 MB total). This cache eliminates most disk accesses to the metadata container for our experiments, but more sophisticated caching schemes and smaller pages could be beneficial for other workloads.

The combination of the compact Cuckoo hash table and the dedup metadata cache makes feature lookups in sDedup fast and memory-efficient. We show in Section 5 that the indexing overhead is small and bounded in terms of CPU and memory usage, in contrast to traditional deduplication.

4.3 Source Document Cache

Unlike chunk-based deduplication systems, sDedup does not rely on having a deduplicated chunk store, either of its own or as the document database implementation. Instead, it directly uses a source document from the database and fetches it whenever needed in delta compression and decompression. Querying the database to retrieve documents, however, is problematic for both deduplication and real clients. The latency of a database query, even with indexing, could be higher than that of a direct disk read, such as is used in some traditional dedup systems. Worse, sDedup's queries to retrieve source documents will compete for resources with normal database queries and impact the performance of client applications.

sDedup uses a small document cache to eliminate most of its database queries. This cache achieves a high hit ratio by exploiting the common update pattern of document database workloads. First, good temporal locality exists among similar documents. For example, updates to a Wikipedia article or email in the same thread tend to group within a short time interval. Second, a newer version of a document usually exhibits higher similarity to future updates than an older version, because most document updates happen to the immediate previous version instead of an older version. Based on these observations, in many cases, it suffices to only retain the latest version of the document in the cache.

Cache replacement occurs when sDedup looks for a source document in its cache. Upon a hit, the document is directly fetched from the document cache, and its cache entry is replaced by the target document. Otherwise, sDedup retrieves the source document using a database query and insert the target document into the cache. In either case, the source document is not added to the cache because it is older and expected to be no more similar to future documents than the target document. When the size of the cache is reached, the oldest entry is evicted in a LRU manner.

sDedup also uses a source document cache on each secondary node to reduce the number of database queries during delta decompression. Because the primary and secondary nodes process document updates in the same order, as specified in the oplog, their cache replacement process and cache hit ratio are almost identical.

	Wikipeda	Microsoft Exchange	Stack Exchange
Document Size (bytes)	15875	9816	936
Change Size (bytes)	77	92	79
Change Distance (bytes)	3602	1860	83
# of Changes per Doc	4.3	5.3	5.8

Table 2: Average characteristics of three document datasets.

5. Evaluation

This section evaluates sDedup using three real-world datasets. For this evaluation, we implemented both sDedup and traditional deduplication (trad-dedup, see Section 2.3) in the replication component of MongoDB v2.7. The results show that sDedup significantly outperforms traditional deduplication in terms of compression ratio and memory usage, while providing comparable processing throughput.

Unless otherwise noted, all experiments use a non-sharded MongoDB installation 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 disabled MongoDB's full journaling feature to avoid interference.

5.1 Data Sets

We use three datasets representing different document database applications: collaborative text editing (Wikipedia), on-line forums (Stack Exchange), and email (Microsoft Exchange). Table 2 shows some key characteristics of these datasets. The average document size ranges from 1–16 KB, and most changes modify less than 100 B.

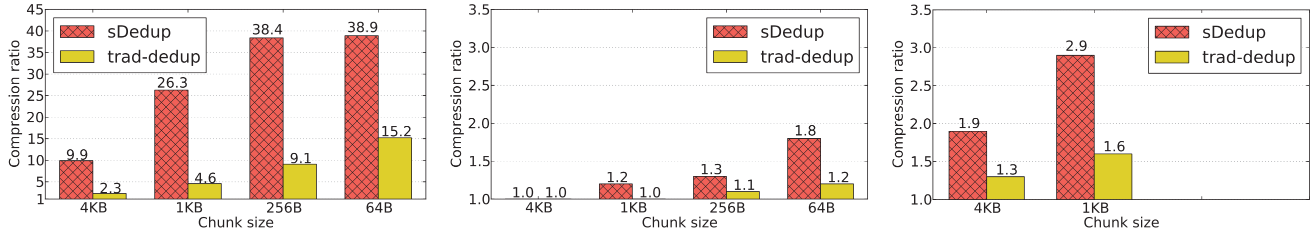
Wikipedia: The full revision history of every article in the Wikipedia English corpus [9] from January 2001 to August 2014. We extracted a 20 GB subset via random sampling. Each revision contains the new version of the article and metadata about the user that made the change (e.g., username, timestamp, comment). Most duplication comes from incremental revisions to pages, and each revision is inserted into the DBMS as a new document.

Stack Exchange: A public data dump from the Stack Exchange network [8] 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. Each post, revision, etc. is inserted into the DBMS as a new document.

Microsoft Exchange: A 4.3 GB sample of email blobs from a cloud deployment. Each blob contains the text message, thread ID, and metadata such as sender and receiver IDs. Duplication mainly exists in message forwarding and replies that contain content of previous messages. We were not granted direct access to the user email data, allowing only limited experimentation.

5.2 Compression Ratio

This section evaluates the compression ratios achieved by sDedup and trad-dedup. Each dataset is loaded into a DBMS

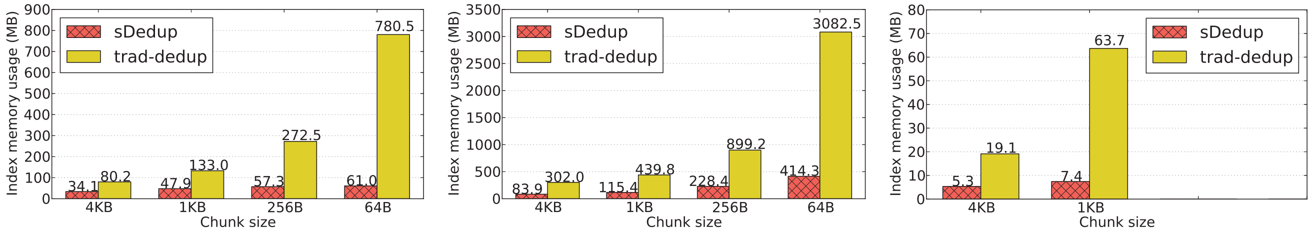


(a) Wikipedia

(b) Stack Exchange

(c) Microsoft Exchange

Figure 7: Compression Ratio – An evaluation of the compression achieve for the different datasets with varying chunk sizes.



(a) Wikipedia

(b) Stack Exchange

(c) Microsoft Exchange

Figure 8: Indexing Memory Overhead – A comparison of the amount of memory used to track the internal deduplication indexes.

instance as fast as possible, and the replication bandwidth is measured. The compression ratio is computed as the difference between the amount of data transferred from the primary to the secondary when the DBMS does and does not use oplog deduplication, without additional compression (e.g., *gzip*). As shown in Fig. 1, using *gzip* reduces the data size by another 3× for each approach.

Fig. 7a shows the results for the Wikipedia dataset, for each of the four chunk sizes (from 4 KB to 64 B). The Y-axis starts from one, which corresponds to the baseline of no deduplication. With a typical chunk size setting of 4 KB, trad-dedup only achieves a compression ratio of 2.3×. sDedup achieves a compression ratio of 9.9×, because it identifies byte-level duplicate regions between similar documents via delta compression. When the chunk size is decreased to 1 KB, both compression ratios improve. sDedup improves more, however, because finding more similar documents enables greater deduplication than just identifying more duplicate chunks. When the chunk size is further decreased to 256 B, sDedup is still better, achieving a compression ratio of 38.4× as compared to 9.1× with trad-dedup. The improvement for sDedup is smaller, because it approaches the upper bound of the potential data reduction for the Wikipedia dataset. Further decreasing the chunk size provides little additional gain for sDedup but is beneficial for trad-dedup. Recall, however, that trad-dedup index memory grows rapidly with smaller chunk sizes, while sDedup's does not; Section 5.3 quantifies this distinction.

Fig. 7b shows the compression ratios for the Stack Exchange dataset. The documents in this dataset are smaller than in Wikipedia (see Table 2), and posts are not revised as frequently, affecting the absolute compression ratios of the two approaches. But, the relative advantage of sDedup over trad-dedup still holds for all chunk sizes.

Fig. 7c shows the compression ratios for the Microsoft Exchange dataset (we could only obtain results for 1 KB and 4 KB chunk sizes). This dataset exhibits less duplication than Wikipedia, because the number of email exchanges per thread is smaller than the number of revisions per article. sDedup still provides a higher compression ratio than trad-dedup at all chunk sizes. When the chunk size is 1 KB, sDedup reduces the data transferred by 65%.

5.3 Indexing Memory Overhead

Memory efficiency is a key factor in making inline deduplication practical. sDedup achieves this goal by consistent sampling of chunk hashes and use of a compact Cuckoo hash table. The index memory usage for each document is at most 48 B ($K = 8$), regardless of its size or number of chunks. In comparison, trad-dedup indexes every unique chunk, using the 20 B SHA1-hash as the checksum. Thus, it consumes 24 B of index memory for each chunk. In addition, as described in Section 4, both approaches use small caches for dedup metadata (∼8 MB) and source documents (∼32 MB) to reduce I/O overhead.

Fig. 8 shows the index memory usage corresponding to the experiments in Fig. 7. sDedup consistently uses less memory; the difference is largest for the small chunk sizes that provide the best compression ratios. Using a small chunk size does not explode sDedup's memory usage, because it uses only the top-K index entries per document. Conversely, trad-dedup's memory usage grows rapidly as chunk size decreases, because the number of unique chunks increases proportionally. For Wikipedia (Fig. 8a), with an average chunk size of 64 B, trad-dedup consumes 780 MB memory for deduplication indexes, which is more than 12×

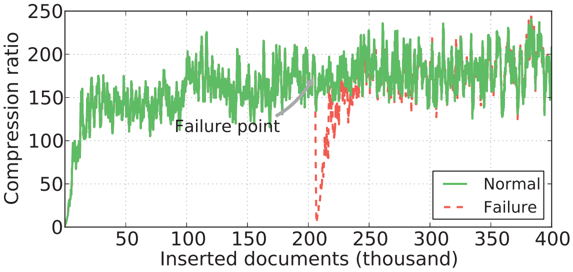


Figure 9: Failure Recovery – Measuring how quickly sDedup recovers after the primary fails.

higher than sDedup. This shows that using small chunk sizes is impractical for trad-dedup. When chunk size is 256 B, sDedup achieves 4× higher compression ratio than trad-dedup while using only $1/5$ the index memory. Fig. 8b and Fig. 8c show similar results for the Stack Exchange and Microsoft Exchange datasets.

5.4 Failure Recovery

When a primary node fails, a secondary node is elected to become the new primary. Because the dedup index is maintained on the original primary, the new primary needs to build its own index from scratch as new documents are inserted. To evaluate sDedup's performance in presence of a primary node failure,[3] we use a 80 GB Wikipedia dataset sorted by revision timestamp to emulate the real-world write workload. We load the dataset into a primary with two secondaries and stop (fail) the primary after 200k insertions.

Fig. 9 shows the compression ratios achieved by sDedup in the normal and failure cases with a moving average of 2000 inserted documents. The compression ratio decreases significantly at the failure point, because the documents that would originally be selected as similar candidates can no longer be identified due to loss of the in-memory deduplication index. The compression ratio up returns to normal reasonably quickly (after ∼50k new document insertions). This is because most updates are to recent documents, so that the effect of missing older documents in the index fades rapidly.

When the primary node restarts due to a normal administrative operation, sDedup can rebuild its in-memory deduplication index (on the original primary) to minimize the loss of compression ratio. sDedup achieves this by first loading the log-structured dedup metadata using a sequential disk read, and then replaying the feature insertions for each document in the oplog. The rebuild process finishes quickly (less than three seconds for 200k documents), after which sDedup behaves as if no restart occurred.

5.5 Tuning Parameters

sDedup has two primary tunable parameters, in addition to the chunk size explored above, that affect performance/memory trade-offs: sketch size and source document cache size. This section quantifies the effects of these parameters and explains how we select default values.

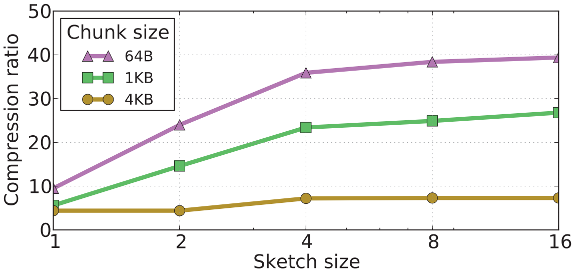


Figure 10: Sketch Size – The impact of the sketch size on the compression ratio for the Wikipedia dataset.

Sketch size: As described in Section 3.1, a sketch consists of the top-K features. Fig. 10 shows the compression ratio achieved by sDedup as a function of the sketch size (K). For the smaller chunk sizes ($\leq$ 1 KB) that provide the best compression ratios, K should be 4–8 to identify the best source documents. $K > 8$ provides minimal additional benefit, while increasing index memory size, and $K = 8$ is the default configuration used in all other experiments. Larger chunk sizes, such as 4 KB, do not work well because there are too few chunks per document, and increasing the sketch size only helps slightly.

Source document cache size: sDedup's source document cache reduces the number of database queries issued to fetch source documents. To evaluate the efficacy of this cache, as a function of its size, we use a snapshot of the Wikipedia dataset that contains the revisions for all articles on a randomly selected day in September 2009, which is ∼3 GB. We replay the revisions in timestamp order as document insertions into MongoDB, starting with a cold cache, and report the steady-state hit rates with and without sDedup's cache-aware selection technique (see Section 3.2).

Fig. 11 shows the hit rate of the document cache as a function of the cache size. Even without cache-aware selection, the source document cache is effective in removing many database queries due to temporal locality in the document updates. Enabling cache-aware selection provides an additional ∼10% hits (e.g., 50% hit rate instead of 40%) for all cache sizes shown. For example, with a relatively small cache size of 2000 entries (∼32 MB, assuming average document size of 16 KB) the hit ratio is ∼75% without and ∼87% with cache-aware selection. So, the number of cache misses is cut in half. We use a cache size of 2000 entries for all other experiments, providing a reasonable balance between performance and memory usage.

5.6 Processing Throughput

This section evaluates sDedup's throughput and impact on the overall DBMS insertion throughput, showing that it does not hurt performance when bandwidth is plentiful and significantly improves performance when it is not.

[3] Failure on a secondary node has no effect on the compression ratio, because only the primary maintains the deduplication index.

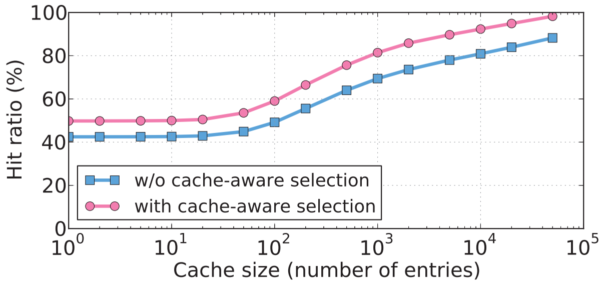


Figure 11: Source Document Cache Size – The efficacy of the source document cache and the cache-aware selection optimization.

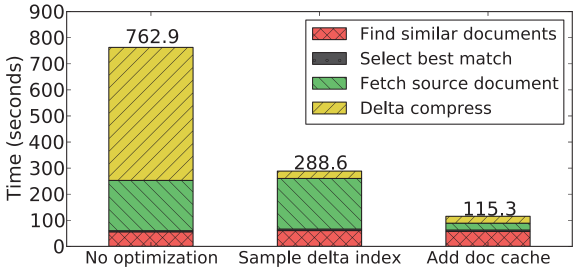


Figure 12: Deduplication Time Breakdown – Time breakdown of deduplication steps as individual refinements are applied.

Deduplication throughput: Fig. 12 shows the time required to insert the same 3 GB Wikipedia snapshot used in Section 5.5, using stacked bars to show the contribution of each step described in Section 3. The three bars show the benefits of adding each of sDedup's two most significant speed optimizations: sampling source index in delta computation and adding a source document cache. The default configuration uses both of the optimizations.

With no optimizations, sDedup spends most of the time fetching source documents from the DBMS and performing delta compression. The unoptimized delta compression step is slow because it builds an index for each offset in the source document. sDedup addresses this issue by sampling only a small subset of the offsets, at a negligible cost in compression ratio. With a sampling ratio of $\frac{1}{32}$, the time spent on delta compression is reduced by 95%, which makes fetching source documents from the database the biggest contributor. By using a small source document cache of 2000 entries, sDedup reduces the source fetching time by ~87%, which corresponds to the hit rate observed in Section 5.5.

Impact on insertion throughput: Ideally, using sDedup should have no negative impact on a DBMS's performance, even when it is not needed because network bandwidth between primary and secondary is plentiful. Fig. 13 shows MongoDB's aggregate and real-time insertion throughput for the 3 GB Wikipedia snapshot and the Stack Exchange dataset, inserted as fast as possible with and without sDedup. The results show that sDedup does not greatly reduce write throughput, because of its resource-efficient design and implementation. We focus on write throughput, because sDedup is not involved for read queries.

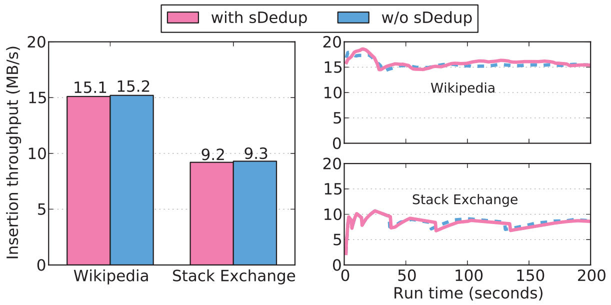


Figure 13: Impact on Insertion Throughput – The aggregate and real-time insertion throughput with and without sDedup.

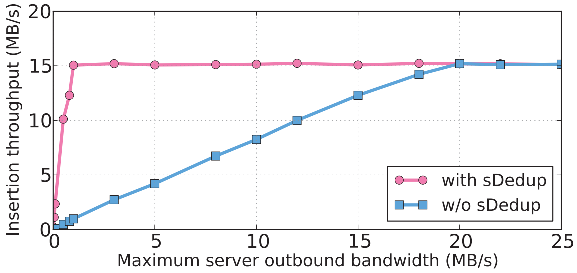


Figure 14: Insertion Throughput under Limited Bandwidth.

Performance with limited bandwidth: When network bandwidth is restricted, such as for WAN links, remote replication can throttle insertion throughput and reduce end-user performance. sDedup improves the robustness of a DBMS's performance in the presence of limited network bandwidth.

To emulate an environment with limited bandwidth, we use a Linux traffic control tool (*tc*) to configure the maximum outbound network bandwidth on the primary server. The experiments load the Wikipedia snapshot into the DBMS as fast as possible and enforce replica synchronization every 1000 document insertions.

Fig. 14 shows the DBMS's insertion rate as a function of available network bandwidth. Without sDedup, the required replication bandwidth is equal to the raw insertion rate, resulting in significant throttling when bandwidth is limited. With sDedup, on the other hand, the DBMS is able to deliver full write performance even with limited network bandwidth, because less data is transferred to the secondary.

5.7 Sharding

This section evaluates the performance of sDedup in a sharded cluster, in which data is divided among multiple primary servers that each has a corresponding secondary. Each primary/secondary pair runs an independent instance of sDedup. For experiments with sharding, we use the 20 GB Wikipedia dataset and shard documents on article ID (like the Wikipedia service). To accommodate MongoDB's capacity balancing migrations, we modified sDedup to remove the assumption that the source document can always be found on the primary node. When it cannot, because it was migrated, sDedup simply deletes it from the dedup index and treats the target document as unique.

Number of shards	1	3	5	9
Compression ratio	38.4	38.2	38.1	37.9

Table 3: Compression ratio with sharding.

Table 3 shows the compression ratio as a function of the number of shards. The compression ratio is not significantly affected, because Wikipedia documents with the same article ID go to the same server and most duplication comes from incremental updates to the same article. This indicates that sDedup is robust and still works in sharded deployments.

6. Related Work

We are not aware of any previous work that uses deduplication to reduce replication bandwidth for DBMSs. The common practice is to use compression. We have shown that deduplication reduces data more than compression in document databases, and the two can be combined for greater reduction. This section surveys related work in storage deduplication, similarity search, and delta compression.

There are two high-level approaches for detecting duplicate data. The first looks for exact matches on the unit of deduplication. Deduplication granularity can be a whole file [10, 23] or a data chunk [18, 21, 22, 29, 33, 36, 47]. Deduplication techniques based on variable-sized data chunks usually provide the best compression ratio due to their resistance to boundary-shifting [24], but at the cost of slightly higher computation overhead in chunking. Such systems build an index of chunk hashes (using SHA-1) and consult it for detecting duplicate chunks. The second approach looks for similar units (chunks or files) and deduplicates them. There are two methods to deduplicate similar objects. The first [14, 29, 35] divides similar objects into smaller sub-chunks and eliminates exact matches using sub-chunk hashes. Some previous work [12, 39] uses the second method to delta compress objects and stores the encoded data. In these systems, if the source data chunk is stored in the encoded form, it might require one or multiple decoding stages to reconstruct the original content before it can be used for delta compression. sDedup does not store encoded data on persistent storage and thus has no such overhead.

sDedup's target workload differs significantly from that of previous deduplication systems focused on backup [21, 29, 47] or primary storage [1, 5–7, 11, 22, 41]. For these workloads, more than 90% of duplication savings come from unmodified data chunks in large files on the order of MBs to GBs [32, 45], so typical chunk sizes of 4-8 KB work well. For user files, even whole-file deduplication may eliminate more than 50% of redundancy [22, 32]. sDedup is optimized for small documents with dispersed changes, for which chunk-based deduplication does not yield satisfactory compression ratios unless using small chunk sizes. However, as shown in Section 5.3, this incurs significant indexing memory overhead. Instead, sDedup finds a similar document and uses document-level delta compression to remove duplication with low memory and computation costs.

There has been much previous work in finding similar objects in large repositories. The basic technique of identifying similar files by maximizing Jaccard coefficient of two sets of polynomial-based fingerprints is pioneered by Manber [31]. Spring et al. [40] use this technique to identify repeated byte ranges on cached network packets to reduce the redundant network traffic. Broder [16, 17] extends this approach to group multiple fingerprints into super-fingerprints. A super-fingerprint match indicates high probability of similarity between objects, so that the algorithm may scale to very large files. Kulkarni et al. [27] adapt this method and combine it with compression and delta encoding to improve efficiency. Several other systems [14, 29, 35] take an alternative approach using a representative subset of chunk hashes (IDs) as the feature set. sDedup uses a similar approach to extract features by sampling top-*K* chunk IDs, but only uses them to identify similar documents rather than for chunk-level deduplication. We defer it as an interesting future work to explore using the super-fingerprints-based approach for finding similar documents in sDedup.

Delta compression has been used to reduce network traffic for file transfer or synchronization. Most of this work assumes that previous versions of the same file are explicitly identified by the application, and duplication happens only among prior versions of the same file [42, 44]. When the underlying DBMS is not told about versioning, or duplication exists across different documents, sDedup is still able to identify a similar document from the data corpus and therefore is a more generic approach. TAPER [25] 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 chunk hashes in the respective files.

Delta compression has been widely studied. Notable examples are general-purpose algorithms based on the Lempel-Ziv [48], such as vcdiff [13], xDelta [30], and zdelta [43]. Specialized schemes can be used for specific data formats (e.g., XML) to improve compression quality [19, 28, 38, 46]. The delta compression algorithm used in sDedup is based on the xDelta algorithm for identifying byte-level duplication between two binary objects.

7. Conclusion

sDedup is a similarity-based deduplication system that addresses the network bandwidth problem for replicated document databases. It exploits key characteristics of document database workloads to achieve excellent compression ratios while being resource efficient. Experimental results with three real-world datasets show that sDedup is able to achieve up to two orders of magnitude reduction in data sent over the network, significantly outperforming traditional chunk-based deduplication approaches in terms of compression ratio and indexing memory usage while imposing negligible performance overhead.

Acknowledgements: We thank Sanjeev Mehrotra for helping with the data chunking portion of the system and for useful discussions. We thank Andrei Marinescu for suggesting a motivating scenario from Microsoft Exchange cloud deployment and for providing the Exchange dataset. We thank our shepherd, Fred Douglis, and the anonymous reviewers for helpful feedback on the paper. We also thank the members and companies of the PDL Consortium (including Actifio, Avago, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Seagate, Symantec, 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.

References

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

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

[3] MongoDB Monitoring Service. https://mms.mongodb.com.

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

[5] NetApp Deduplication and Compression. www.netapp.com/us/products/platform-os/dedupe.html.

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

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

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

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

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

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

[12] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 6. ACM, 2009.

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

[14] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–9. IEEE, 2009.

[15] D. Bhagwat, K. Eshghi, and P. Mehra. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–112. ACM, 2007.

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

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

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

[19] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 41–52. IEEE, 2002.

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

[21] 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.

[22] 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.

[23] EMC Corporation. EMC Centera: Content Addresses Storage System, Data Sheet, April 2002.

[24] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. 2005.

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

[26] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[27] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Usenix Annual Technical Conference*, 2004.

[28] E. Leonardi and S. S. Bhowmick. Xanadue: a system for detecting changes to xml data in tree-unaware relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1137–1140. ACM, 2007.

[29] 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.

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

[31] U. Manber et al. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[32] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *FAST*, 2011.

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

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

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

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

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

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

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

[40] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. *ACM SIGCOMM Computer Communication Review*, 30(4):87–95, 2000.

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

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

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

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

[45] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *FAST*, 2012.

[46] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 519–530. IEEE, 2003.

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

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