Reducing Replication Bandwidth for Distributed Document Databases

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

CMU-PDL-14-108
December 2014

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

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 also thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Huawei, Intel, Microsoft, NEC Labs, 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 MongoDB Incorporated. Experiments were enabled by generous hardware donations from Intel, NetApp, and APC.

Keywords: Deduplication, Geo-replication, Document-oriented Databases

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 an important consideration for document DBMSs in order to guarantee availability. Replication can be between failure-independent nodes in the same data center and/or in geographically diverse data centers. A replicated DBMS maintains synchronization across multiple nodes by sending operation logs (oplogs) across the network, and the network bandwidth required can become a bottleneck. As such, there is a strong need to reduce the bandwidth required to maintain secondary database replicas, especially for geo-replication scenarios where wide-area network (WAN) bandwidth is expensive and capacities grow slowly across infrastructure upgrades over time.

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 of documents in oplog entries by delta encoding against similar documents selected from the entire database. It exploits key workload characteristics of document-oriented workloads, including small document sizes, temporal locality, and incremental nature of document edits. Our experimental evaluation of sDedup using MongoDB with three real-world datasets shows that it is able to achieve up to 38× reduction in oplog bytes sent over the network, in addition to the standard 3× reduction from compression, significantly outperforming traditional chunk-based deduplication techniques while incurring negligible performance overhead.*

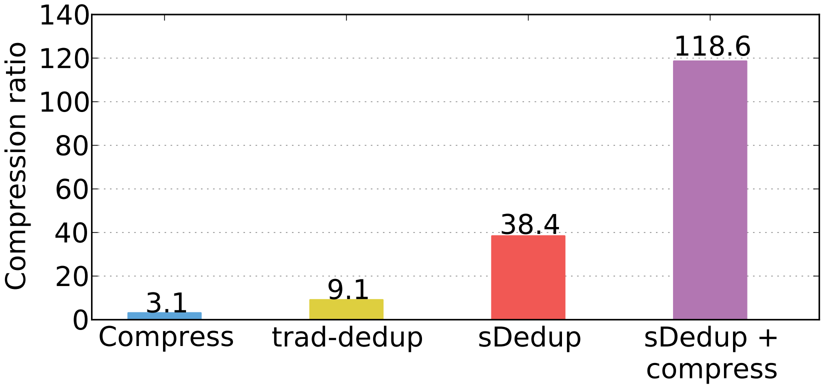


Figure 1: Compression ratios for Wikipedia. The four bars represent compression ratios achieved for a snapshot of the Wikipedia dataset (see Section 5 for details) for four approaches (1) standard compression on each oplog batch (4 MB average size), (2) traditional chunk-based dedup (256 B chunks), (3)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

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 schema-less manner, instead using a hierarchy of properties. Document DBMSs support rich relational and hierarchical queries over such documents.

Because these DBMSs are typically used with front-end (i.e., 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 (i.e., nodes on different racks, different clusters) or across data centers in geographically separated regions.

For popular applications with many concurrent users, such replication can require significant amount of network bandwidth. This network bandwidth, however, becomes increasingly scarce and more 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, with WAN bandwidth being the most expensive and growing relatively slowly across infrastructure upgrades over time.

One approach to solving this problem is to use some form of compression on the oplog to reduce the amount redundant data that is sent from the primary DBMS nodes to the replicas. 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, much higher ratios can be realized with *deduplication* techniques that exploit redundancy with data beyond the current oplog batch. For Wikipedia data, as shown in Fig. 1, traditional deduplication approaches can achieve up to 9× and our proposed similarity-based deduplication scheme up to 38×. Moreover, these compression ratios can be combined with the 3× from compression, yielding almost 120× for our proposed approach.

Most traditional data deduplication systems target chunk-based backup streams for large-scale file systems [16, 33, 40, 18, 34, 24]. As such, they rely upon several properties of such workloads. First and foremost is that backup files are large and changes affect an extremely small percentage of the locations within the backup file. This allows use of large chunks, which is crucial to avoiding the need for massive dedup indices; the trad-dedup bar in Fig. 1 ignores this issue and shows data for a 256 B chunk size. With a more reasonable 4 KB chunk size, trad-dedup achieves 2.3×. Second, they assume that good *chunk locality* exists across backup streams, such that chunks tend to appear in roughly the same order in each backup stream (from a given file system)A. This allows simple prefetching of dedup metadata to be effective during the deduplication and reconstruction processes.

In our experience, the workloads for document database applications do not exhibit these characteristics. Instead of containing large blocks of data that correspond to the same entity (e.g., backup stream), documents are small; the size of the average document is less than 100 KB [3]. The replication streams of these databases also do not exhibit much chunk locality. They instead have *temporal locality* where there are frequent updates to specific documents within a short interval of time, each reflected in the oplog as a new document that is similar to the version it replaced. 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 approaches for deduplication a poor match for document DBMSs.

We present the design and implementation of a deduplication system, called *sDedup* , that exploits the characteristics of document database workloads. sDedup employs a number of techniques to achieve excellent compression rates, with only a small amount of memory and I/O overhead. The key idea of sDedup is to perform deduplication on a per-document basis rather than on a per-chunk basis. It uses consistent sampling and a compact Cuckoo hash table to significantly reduce the size of the meta-data needed to keep track of documents with inconsequential loss in deduplication quality. sDedup applies delta compression using a variant of the *xDelta* algorithm [25] to encode similar documents and improve compression rates over to chunk-based deduplication. This similarity/delta compression-based deduplication approach, known as *similarity-based deduplication*, does not require exact duplicates at any chunking level in order to eliminate redundancy.

We integrated sDedup into the replication infrastructure of the MongoDB DBMS [2] and evaluated the system using real-world datasets. For this analysis, we also implemented in MongoDB a simple compression scheme and chunk-based deduplication approach. Our results show that sDedup achieves much higher compression ratios, as illustrated in Fig. 1. sDedup's approach also uses much less memory (e.g., 75% less for the example in Fig. 1) than chunk-based dedup. Furthermore, using sDedup has a negligible impact on the runtime performance of the DBMS.

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 MongoDB. Section 5 evaluates sDedup on real-world datasets and explores sensitivity to important configuration parameters. We conclude with a survey of related work in Section 6 and a discussion of our future research directions in Section 7.

2 Background and Motivation

This section discusses why reducing network bandwidth usage for DBMS replication is desirable and motivates the need for an approach using similarity-based deduplication.

2.1 Network Bandwidth for Replication

Database replication requires the secondary nodes (replicas) to be synchronized with state changes (user updates) in the primary. 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 in real-time with the user update (strong consistency) or can lag behind by some given amount (bounded staleness) or be delayed indefinitely (eventual consistency). Synchronization involves the primary node sending a description of changes it has seen since the last such synchronization to the secondary nodes. 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 master to the replica. The replica node then replays the log to update the state of its copy of the database. Network bandwidth needed for such synchronization is directly proportional to the volume and rate of updates happening at the primary. When network bandwidth is not sufficient, it can become the bottleneck for replication performance and even end-to-end client performance in the stricter consistency models (e.g., strong, bounded staleness).

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. However, network bandwidth gets increasingly scarce and expensive going up the network hierarchy, with WAN bandwidth across regional data centers being most expensive, scarce, and slow-growing over time. Reducing the cross-replica network bandwidth usage of replicated services allows more services to use more diverse replicas at comparable performance without needing to upgrade the network. Given the nature of WAN bandwidth, geo-replication is likely to benefit the most from reduction in cross-replica network bandwidth usage.

2.2 The Need for Similarity Deduplication

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

The emergence of hierarchical data center infrastructure, the need to provide increased levels of reliability and availability on commodity hardware in the cloud, and the popularity of document-oriented, semi-structured 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, which are mostly represented in the form of text data (e.g., wiki pages, emails, blogs/forums, service logs, sensor data streams). Even small updates to text data cannot be easily expressed as incremental operations. As a result, usually a document update 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 for data reduction for replication and local storage. We focus on reducing network bandwidth for replication in this paper.

Past work in the literature has explored different ways of removing redundant data for various target applications, 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: Replicated databases are synchronized by propagating updates from the primary node's oplog to several secondary nodes. These updates 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 stricter consistency guarantees, the size of the oplog batch is usually on the order of megabytes. 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 performed a simple experiment using the Wikipedia dataset with a modified

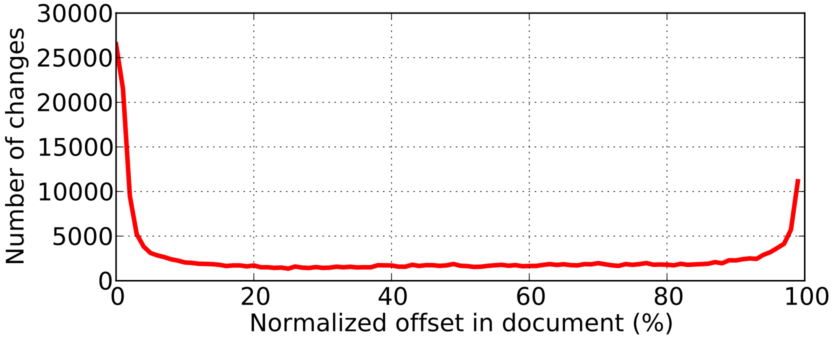


Figure 2: Distribution of document modifications for Wikipedia. The offset of a particular document is the number of changes that occur to other documents in the database before that document is updated again.

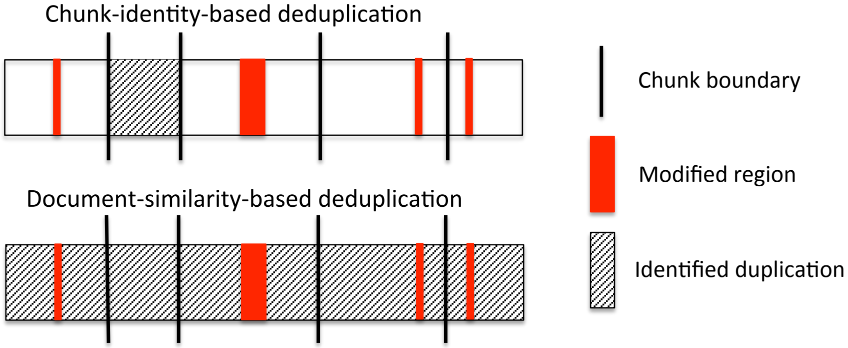

Chunk-identity-based deduplication
Document-similarity-based deduplication
Chunk boundary
Modified region
Identified duplication

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

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 compression only reduced 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: Data deduplication is a specialized data compression technique that eliminates duplicate copies of repeating 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 term in the dictionary is large (kilobytes), whereas with simple compression the terms are usually short strings (bytes).

A typical deduplication scheme in a replicated file system works as follows. An incoming file (corresponding to a document in the context of document-oriented databases) is first divided into chunks in a content-dependent manner using Rabin-fingerprinting [31]; 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 matching pattern. Generally, a match pattern of n bits leads to an average chunk size of 2^n bytes. For each chunk, a collision-resistant strong signature hash (e.g., SHA-1) is calculated as its identity. The chunk id is then looked up in a global index table. If a match is found, the chunk is declared as duplicated. 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 files of sizes ranging from tens of megabytes to hundreds of gigabytes [26, 39, 18], 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 fairly small (kilobytes). Using a large chunk size may lead to a significant reduction in deduplication quality. On the other hand, using a small chunk size and building indexes for all the unique chunks imposes significant memory and storage overhead, which is infeasible for an in-line deduplication system. As we will discuss in the next section, our sDedup approach uses a small (configurable) chunk size of 1 KB or less, but indexes only a subset of the chunks that mostly represent the document for purposes of detecting similarity. This means that the deduplication system is able to achieve more efficient memory usage with small chunk sizes, while still providing a high compression ratio.

The second key observation is that document database updates 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 identifiable duplication for each of the document-similarity-based and chunk-identity-based deduplication approaches. For the chunk-based approach, when modifications to documents are spread over the entire document, many chunks with only slight modifications would be declared as unique. Decreasing the chunk size alleviates this problem, but incurs much higher indexing overhead. sDedup is able to identify all duplicate regions with the same chunk size. It utilizes a fast and memory-efficient similarity match index to identify similar documents with at least one common chunk, and uses a byte-level delta compression algorithm on similar document pairs to find the longest segments of duplicate bytes.

3 Dedup workflow in sDedup

This section describes the deduplication workflow of sDedup, our similarity-based deduplication system. sDedup takes as input a data-item to be encoded, which we call the *target* document, and performs similarity-based deduplication on it. sDedup differs from traditional chunk-based deduplication systems [16, 33, 40, 18, 34, 24], which break the input data-item into chunks and find identical chunks stored anywhere else in the data corpus (e.g., the document database, in this paper's context). sDedup instead identifies a single similar data-item and performs differential compression to remove the redundant parts.

sDedup's workflow has three primary steps: (1) finding all documents in the corpus that are similar to the target (input) 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 selected source document. Fig. 4 illustrates data structures and actions involved in transforming each target document into a delta encoded representation. The remainder of this section describes these three steps in further detail. We then describe in Section 4 how sDedup is implemented and integrated into the replication mechanisms of MongoDB.

3.1 Finding Candidate Source Documents

For a given target document to encode, sDedup first identifies similar documents in the corpus as candidates to use as the source document for delta compression. sDedup computes a *sketch* of the target document composed of a subset of its chunk hashes. The target document is first divided into chunks using the same Rabin fingerprinting algorithm used in traditional deduplication. For each chunk, sDedup computes its unique 64-bit hash using MurmurHash [4]. The sketch consists of the top-*K* hash values sorted in a consistent

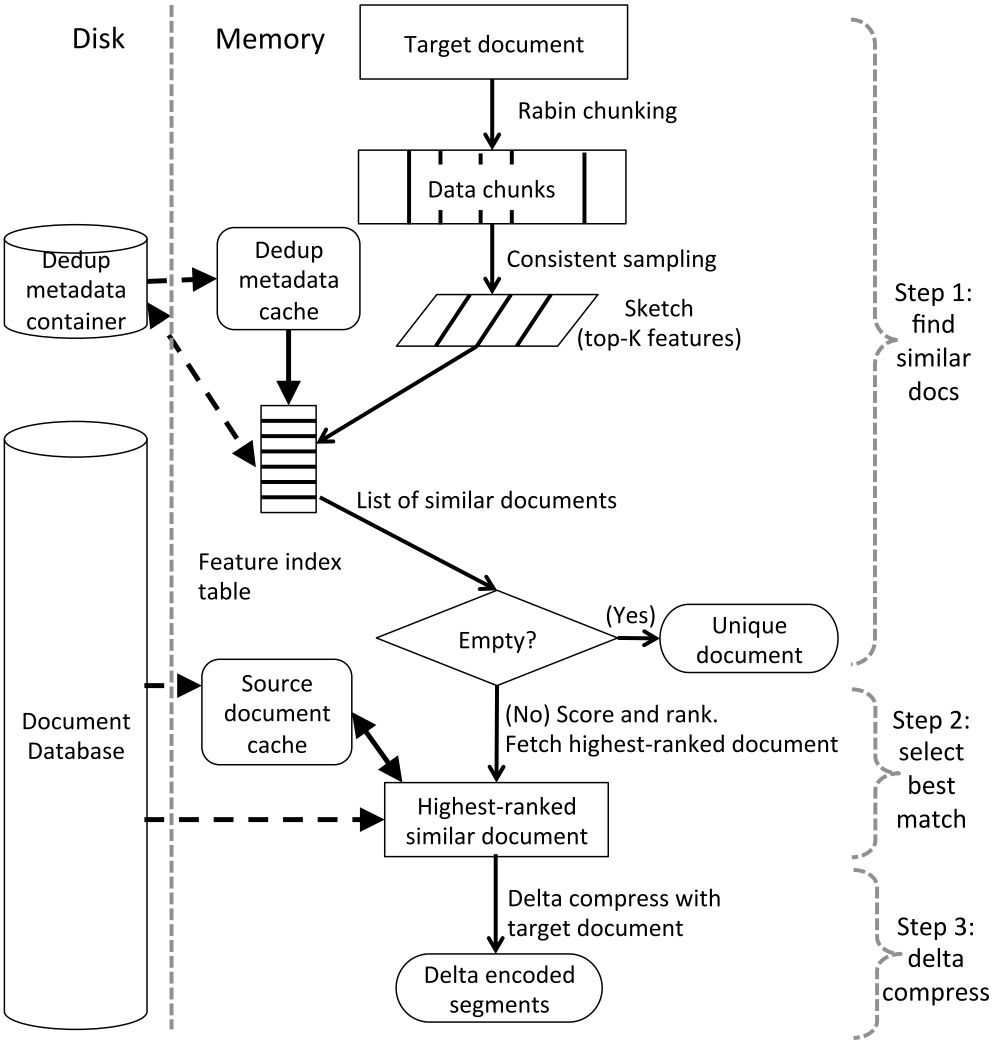


Figure 4: sDedup Workflow and Data Structures – A target (input) document is converted to a delta-encoded form in three steps, flowing from top to bottom in the picture. On the left are the two disk-resident data stores involved in this process: the original document database and dedup metadata storage. The remainder of the structures shown are memory resident.

manner, such as by magnitude. This sorting approach has been shown to be an effective way to characterize a data-item's content in small bounded space [29]. We refer to each one of the top-K hashes as a *feature*. Together, all the top-K features compose of the similarity sketch for the document.

Next sDedup checks to see whether each feature (hash) of the sketch exists in its internal feature index table. This index stores K entries for each document in the corpus: one for each feature in the sketch of that document.[1] sDedup maintains this index on the primary node and asynchronously updates it as new documents in the database are added, removed, and modified. Each index entry has the ID of a document with the corresponding feature. Each document for which there is at least one identical feature is considered "similar," and its ID and sketch are added to the list of candidate sources. The full document ID and sketch

[1] When the total number of chunks in a document is less than K, all the chunk hashes are included in the sketch as features. In this case, the number of index entries for this document is less than K.

are kept in separate sDedup metadata storage for persistence and to minimize the feature index table size.

Because sDedup's feature index only has K entries per document, it is much smaller than the corresponding index for traditional deduplication systems, which must have an entry for every chunk in the system. The value of K is a configurable parameter that trades off resource usage for similarity metric quality. In general, a larger K gives better similarity coverage, but requires more index lookups and more index memory. Thus, in practice a small value of K is good enough to identify moderately similar pairs with a high probability [29]. For our experimental analysis in this paper, we found $K = 8$ is sufficient to identify similar documents with reasonable memory overhead. We also explicitly evaluate the impact of this parameter on sDedup's performance in Section 5.4.1.

3.2 Selecting the Best Source Document

After sDedup identifies a list of candidate source documents, it next selects one of them to use. Each candidate is assigned a score, and the one with the highest score is selected. To break ties, the newest document is chosen. This decision is based on the empirical observation that newer documents are usually better choices. Fig. 5 provides an example of this ranking process. If no similar documents were found, then the target document is declared unique and thus is not eligible for encoding.

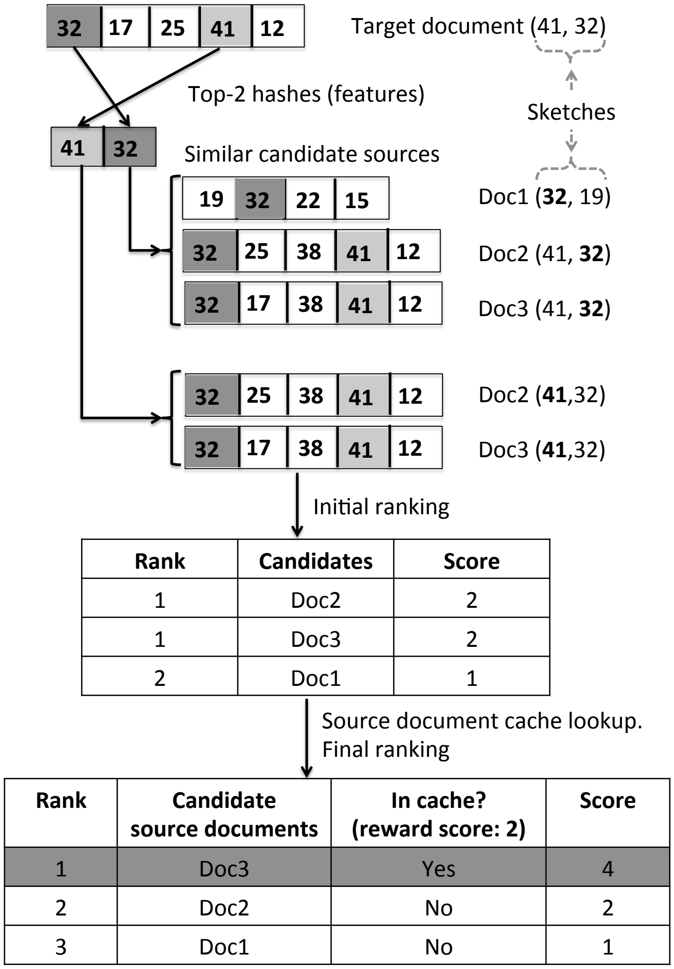
The base score for each candidate source document is the number of features from its sketch ($M \leq K$) that are also in the target document sketch. This base score is adjusted upward by a reward (two, by default) if the candidate is present in sDedup's source document cache (cf. Section 4.3). This means that the system does not need to retrieve the document from corpus database for delta compression in the next step. Although this reward can result in a less similar document being used, we found that it increases efficiency significantly. We evaluate the effectiveness of the document cache and the cache-aware selection reward in Section 5.4.2.

3.3 Delta Compression

In the final step, sDedup's uses delta compression on the target document with the selected source document. The system first checks its internal document cache for the source document; if it is not there then it retrieves it from the corpus (i.e., database). sDedup uses only one source document for this compression. This is because we found that using more than one is not only unnecessary (i.e., it does not produce a better compression ratio), but it also greatly increases the overhead of the deduplication scheme. 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. Thus, using two or more source documents would increase the most significant overhead for little benefit because it would increase the number of look-ups in the corpus. This is the same reasoning that underscores the benefits of sDedup over chunk-based traditional deduplication: our approach only requires one fetch per target document to reproduce the original target document, versus one fetch per chunk.

Efficient algorithms for computing delta encodings on similar files have been widely studied [21, 35, 25, 22, 33]. sDedup employs a delta compression algorithm based on xDelta [25], but reduces the computation overhead with minimal loss of compression ratio. 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 table that maps the hash values to the offsets inside the source document. In order to reduce the computation overhead involved in building this index, sDedup uses fixed sampling to index only offsets at fixed intervals. Because the matching granularity is at the byte level (as opposed to the entire document), the reduction in compression ratio is negligible when the sampling interval is much smaller than the document size.

After sDedup builds its source index, it then calculates the hash value for each offset of the target document using the same sliding-window approach. It uses these two indexes to quickly perform the comparison between hash values. When no match is found, the sliding window moves forward by one byte



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

Rank	Candidate source documents	In cache? (reward score: 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 at the top are used as the two features of its sketch (41, 32); the numbers in the chunks of each document represent the corresponding MurmurHash values. Documents with each feature are identified, via the feature index table, and initially ranked by their numbers of matching features. The ranking is adjusted upward (by two, in this example) if the candidate source document is in the source document cache, because it is more efficient.

and calculates the hash value for the next offset. When a match is found, sDedup compares the source and current documents from the matching points byte-by-byte in both forward and backward directions. The byte matching process continues until it finds the longest match on both ends, which determines the boundaries between unique and duplicate byte segments. The next index look-up skips the offset positions covered by the duplicate segments and starts from the beginning of the new unique 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, segment offset in the source document or the unique bytes, and length. Delta decompression is straightforward and fast. It simply iterates over the ordered segment descriptors and concatenates the duplicate and unique segments to reproduce the original target document.

4 Implementation

We next describe the implementation details of sDedup. Our current version of sDedup is fully integrated into MongoDB (v2.7), the leading open-source document-oriented system. This section describes how we the grafted it into MongoDB's replication framework, as well as the internals of sDedup's indexing mechanisms and the source document cache.

4.1 MongoDB Replication

MongoDB supports primary-secondary replication through a protocol that propagates updates from the primary node to the secondary replicas. The DBMS supports more than one secondary node per primary node. The secondary nodes independently pull updates from the primary without any coordination; this reduces the complexity of the replication system since the primary does not have to track the state of its secondary nodes.

MongoDB maintains an *oplog* at the primary nodes for recovery and replication. Each client write request is applied to the corresponding database and appended to that node's oplog. Each oplog entry contains a timestamp (assigned by the host node) and the JSON-based query that describes the client operation (i.e., insert, update, or delete). For insertion and update operations, the oplog entry includes the physical content of the document as opposed to the logical operation [20].

Fig. 6 illustrates MongoDB's replication protocol with a single secondary replica. The secondary node can request updates from the primary whenever it chooses, by specifying the latest timestamp that it has received. The primary node responds by sending a batch of oplog entries that are later than the specified timestamp. Normally these entries are sent in their native form. When the secondary node receives the response, it reconstructs each entry and appends it to its local oplog, so that its oplog replayer can apply the entries to the local copy of the database.

With sDedup, before an oplog entry is queued up in a batch that is sent to the secondary, it is first passed to the deduplication sub-system 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 response, 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 used to reduce the bandwidth required for replication, but does not to reduce the storage overhead of the actual database.

sDedup's replication protocol is optimistic in that the primary node 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, the secondary sends a supplemental request to the primary to fetch the un-encoded oplog entry. This allows the secondary to retrieve the original oplog entry, rather than the source document; this eliminates the need to reconstruct documents in cases where 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.

This approach differs from previous systems that use data reduction techniques for remote replication in several key aspects [36, 37, 33]. In these other systems, neither the senders nor receivers in the protocol are replicas, so the sender does not know which content may be present at the receiver. As a result, they require several network round-trips to achieve an agreement between a sender and a receiver on which part of the data to be transferred can be reduced. For database replication traffic, however, the secondary's contents are known to the primary—they are identical to the primary's contents, except for unsynchronized updates. As a result, each side can use local indexes and source documents without explicit coordination for each document.

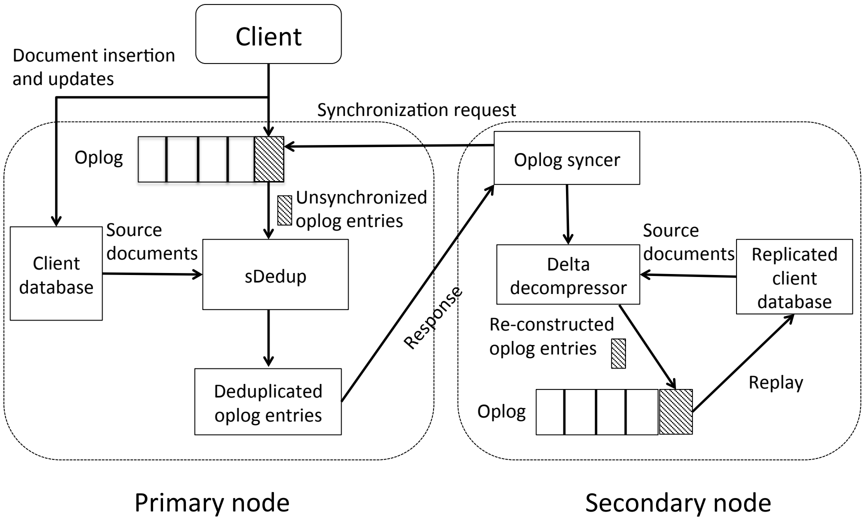


Figure 6: Integration of sDedup into MongoDB – An overview of how sDedup fits into MongoDB's replication mechanism and the components that it interacts with. sDedup deduplicates data to be sent when a secondary requests currently unseen oplog entries. It checks each oplog entry before it is sent to the secondary and then again on the replica to reconstruct the original entries.

4.2 Indexing Documents by Sketch Features

An important aspect of sDedup's design is how it finds similar documents in the corpus. As explained in Section 3.1, this relies on the system's ability to efficiently look up documents by the features of their sketches. 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, sDedup maintains a special index that is separate from the other indexes in the database.

In order for deduplication to be fast enough that it can be used with the replication protocol while the system is running, these feature look-ups must be primarily in-memory operations. Thus, the size of the index is an important consideration since it is using up memory that the DBMS could otherwise use for database indexes and other caches. A naïve approach for this index is to store entries (*K* per document) that contains the document's sketch and its database location. Depending on the database system with which sDedup is being used with, the entry size can range from tens of bytes to hundreds of bytes. In our MongoDB implementation, the database location for each document is 64 bytes: the first 52 bytes contain MongoDB database namespace ID and remaining 12 bytes are the MongoDB object ID. Combined with the 64-byte sketch, the total size of each entry is 128 bytes.

To reduce the memory overhead of this feature index, sDedup uses a two-level scheme. The 128-byte value (which we refer to as the dedup metadata) is stored in a log-structured disk container, and a variant of Cuckoo hashing [28] is used 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-byte compact signature key of the feature value and a 4-byte pointer into the dedup metadata container.

For each feature in the target document's sketch, the lookup and insertion process works as follows; both are performed during the process of finding similar documents. First, a hash value of the feature is calculated starting with the first (out of 16) hashing function. The candidate slot 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 signature key for the feature. Then, the signature key is compared against that of each occupied bucket in the slot. If a match is found, a read to the dedup metadata container is issued 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 hashing function is used to obtain the next candidate slot and the new signature key. The lookup process repeats and adds all matched documents to the list of similar documents, until it finds an empty bucket, which means that there are no more matches. An entry for the target (new) document is added to the empty bucket. If no empty bucket is found after iterating with all sixteen hashing functions, we randomly pick a victim bucket to make room for the new document, and re-insert the victim into the hash table as if it was a new document.

The size and load on the Cuckoo hash table can be further reduced by setting an upper bound on the number of similar documents listed in the index for any given feature. So, for a setting of four, the lookup process for a given feature can stop once it finds a fourth match, since there are no more. And, in this case, insertion of an entry for the target document will require first selecting one of the other four matches to remove from the index; based on update characteristics, we find 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 Cuckoo hash lookup process, all of the matching entries would be removed and reinserted as though they are new entries.

sDedup uses a small dedup metadata cache to reduce the number of reads to the on-disk dedup metadata container. The metadata container is divided into contiguous 64 KB pages, each containing 512 document metadata entries. Upon signature key matches, sDedup fetches an entire page of document metadata entries into the cache and adds it to a LRU list of cache papers. The default configuration uses 128 cache entries of 64 KB, totaling 8 MB. This simple dedup metadata cache has worked well in our experiments, eliminating most disk accesses to the metadata container, 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 approaches.

4.3 Source Document Cache

Unlike traditional deduplication applications, sDedup does not rely on having a deduplicated store, either of its own or as the document database implementation. In-line applications of deduplication to reducing bandwidth keep their own store of chunks for use in deduplication, but we considered that to be too much overhead. Instead, sDedup uses a source document from the document database and fetches it for use in delta compression and decompression.

Having in-line deduplication rely on regular database queries can be a significant performance issue for both deduplication and real clients. The latency of a database query, even with indexing, is usually much 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 specialized document cache to eliminate most of its database queries. sDedup's source document cache provides a high hit ratio by exploiting the common update behavior of document databases combined with the cache-aware similarity selection algorithm described in Section 3.1. The design of the source document cache is based on two insights regarding the document update pattern of our target workloads. First, good temporal locality exists among similar documents. For example, multiple updates to a certain Wikipedia page tend to happen within a short time window; similarly, email replies in the same thread usually have timestamps close to each other. Second, a newer version of a document usually exhibits higher similarity to future updates than an older version. For example, when updating a Wikipedia page, edits

	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 1: Average characteristics of three document datasets.

are usually based on the immediate previous version instead of an older version; in email communications, replies often prepend the most recent previous reply. In these cases, it suffices to only retain the latest version of the document in the cache. Since versions are not identified explicitly, sDedup assumes that the target document is a new version of the selected source document.

The cache replacement policy works as follows: when a document is identified as a source for delta compression, it is looked up in the source document cache. Upon a hit, the document is fetched from the cache, and its cache entry is replaced by the target document. Upon a miss, the source document is retrieved using a database query, and the target document is inserted into the cache. The source document is not kept in the cache, because it is older and expected to be less similar (or 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.

Note that sDedup also uses a source document cache on each secondary node for the same purpose of reducing the number of database queries to fetch source documents 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.

5 Evaluation

We now evaluate the performance and efficacy of sDedup using three real-world datasets. For this evaluation, we implemented both our sDedup and traditional deduplication (trad-dedup) approaches in the replication component of MongoDB. Experimental results show that sDedup significantly outperforms the traditional deduplication approach in terms of compression ratio and memory usage, while providing comparable processing throughput.

The experiments were all performed using a non-sharded installation of MongoDB comprised of one primary node, one secondary node, and a client node. Each node is a separate KVM virtual machine (VM) instance that run on separate physical machine with four CPU cores, 8 GB RAM, and 100 GB of local HDD storage. We disabled MongoDB's journaling feature to avoid overhead interference in our evaluation.

5.1 Data Sets

We use three datasets in our evaluation, namely Wikipedia, Stack Exchange, and Microsoft Exchange. These datasets are made available to us and represent a wide range of applications in document databases including collaborative text editing, on-line forums, and Email messaging. Table 1 shows some of the key characteristics of these datasets. The average document sizes of these datasets are all very small, ranging from around 1–16 KB. We also observe a significant number of changes per document, where each change is typically less than 100 bytes. Below we give more detailed description about each of the three datasets.

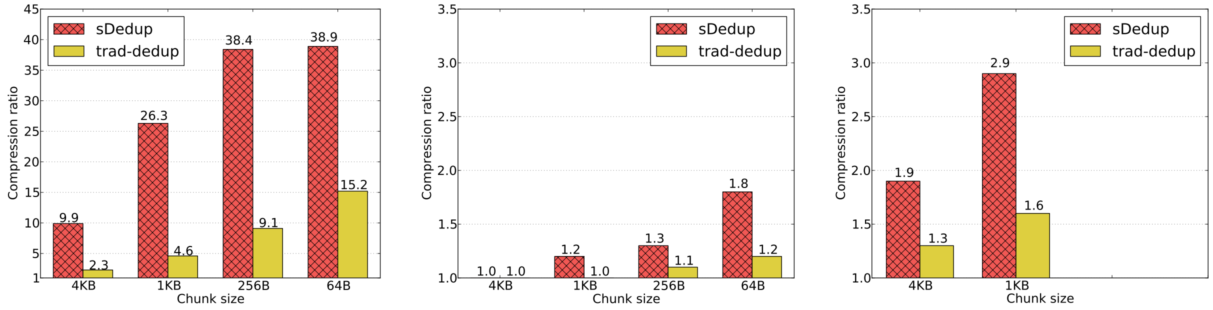
Wikipedia: The first dataset is the dump of the entire Wikipedia English corpus [9]. This data includes all of the revision history to every article from January 2001 to August 2014. We extracted a 20 GB subset of the articles based on random sampling of the article names. Each revision contains the new version of the article and the meta-data about the user that made the change (e.g., username, timestamp, revision comment).

Most duplication in this dataset comes from incremental revisions to the same Wikipedia pages. In our experiments, each revision is inserted into the DBMS as a new document. No read queries are issued to the database, because they do not change the oplog and thus have no impact in our evaluation.

Stack Exchange: The second dataset we used is a sample of a public data dump from the Stack Exchange network [8]. The dataset contains the history of user-posted questions, comments, answers, and other information such as tags, votes and suggested edits. Duplication in this dataset mainly comes from users revising their own posts according to other users' comments and from copying answers from other discussion threads. We randomly sample 10 GB data by post ID out of a total of 100 GB data for the evaluation. Each entity in the dataset is inserted into the DBMS as a new document and there are no queries to read them back.

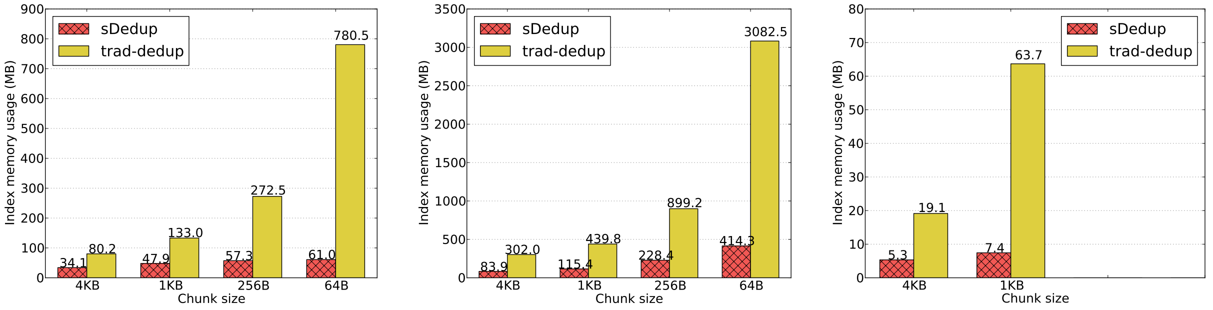
Microsoft Exchange: The last dataset is from a 4.3 GB sample of an anonymized database of email blobs from a hosted Microsoft Exchange cloud deployment. Each email blob contains the text message, thread ID, and other metadata, such as sender and receiver IDs. Duplication is mainly caused by message forwarding and replies within the same thread containing the content of all previous messages. We were not granted direct access to the critical user email data, and were only allowed a limited number of experiments for the evaluation of the compression ratio. We report all the available results for this dataset that we have obtained so far.

5.2 Compression Ratio



(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.

This subsection evaluates the compression ratio achieved by sDedup and the trad-dedup approach, using the three datasets described above.[2] In our first experiment, we loaded the three datasets into the DBMS as fast as it could handle and we measure the amount of compression that the different deduplication approaches are able to achieve. The compression ratio is calculated as the difference between the amount of data transferred from the primary node to the secondary node when the DBMS does not use oplog deduplication. For this experiment, we do not use additional compression (i.e., *gzip*) after the replication system constructs the oplog messages on the primary node, as this would hide the nuances of the deduplication approaches. As shown in Fig. 1, compressing the messages further would reduce the size by another 3× regardless of the deduplication scheme that the system uses.

To demonstrate the impact of chunk size on the compression ratio, we used a variety of settings ranging from 4 KB to 64 bytes. Fig. 7a shows the results for the Wikipedia dataset. The y-axis starts from one, which corresponds to the baseline case when no compression is obtained. When the chunk size is 4 KB, which is the typical setting in file deduplication systems, trad-dedup is only able to achieve a compression ratio of 2.3×. In comparison, sDedup is able to achieve a much better compression ratio of 9.9×, by virtue of its ability to find similar documents as long as one chunk is unmodified and to identify fine-grained, byte-level duplicate regions with delta compression. When chunk size is decreased to 1 KB, the compression ratio of both sDedup and trad-dedup improves accordingly. The improvement of sDedup is more significant, however, because finding similar documents in sDedup provides more benefits than just identifying more duplicate chunks in trad-dedup. When the chunk size is further decreased to 256 bytes, sDedup is still better; it has a compression ratio of 38.4× of the original data size as compared to just 9.1× with trad-dedup. Decreasing the chunk size to 64 bytes improves the compression ratio for trad-dedup, whereas the benefits in sDedup are not as significant because it approaches the upper bounds of deduplication on the Wikipedia dataset. That is, further decreasing the chunk size beyond 256 bytes in sDedup does not provide much additional gain. Note that sDedup limits the maximum number of features for the document sketch, and thus using an smaller chunk size does not have much impact on indexing memory usage or computation overhead.[3] In contrast, as shown in Section 5.3, using smaller chunk sizes in trad-dedup could cause the amount of memory used for indexes and the number of look-ups to increase significantly.

Next, the results in Fig. 7b show the compression ratios for the two approaches on the Stack Exchange dataset. The documents in this data set are on average smaller than the Wikipedia dataset (cf. Table 1). In addition, users' own posts are not revised as frequently as Wikipedia pages which are edited by many users. These affect the absolute compression ratios of the two approaches, but the relative advantage of sDedup over trad-dedup still holds for all chunk sizes. It is interesting to mention that the Stack Exchange network encourages the users to remove duplicate content by downvoting, flagging, or closing posts similar to existing answers. Even so, sDedup is able to remove around 50% of duplicate data.

Lastly, we measured the compression ratios of the two approaches using the Microsoft Exchange dataset (we only have available results with 1 KB and 4 KB chunk sizes). Like with the Stack Exchange data, the email messages in this dataset exhibit less duplication than the Wikipedia dataset. This is because the number of email message exchanges per thread is smaller compared to the number of revisions per Wikipedia page. The results in Fig. 7c clearly show, however, that sDedup still provides a higher compression ratio than trad-dedup at all chunk sizes.[4] When the chunk size is 1 KB, sDedup is able to reduce the data transferred by

[2] We also implemented and evaluated a naïve deduplication approach that only identifies whole document duplicates. However, because completely identical documents are extremely rare, we observe almost no compression (i.e., compression ratio of nearly one) for all datasets using this approach.

[3] Decreasing the chunk size requires the system to sort more chunk hashes when generating document sketches, but this overhead is negligible unless the chunk size is very small. In fact, we observe no slowdown in sDedup when decreasing the chunk size from 4 KB to 256 bytes. When further decreasing the chunk size to 64 bytes, we only observe a slowdown of 2% due to increased sorting overhead.

[4] Our experiments using the Microsoft Exchange datasets were completed using an older version of sDedup. We have added many optimizations since then, but are unable to re-run our experiments due to legal issues. We believe that

65%, approximately 2× better than trad-dedup.

5.3 Indexing Memory Overhead

Memory efficiency is the key factor in making in-line deduplication practical. sDedup achieves this goal with two techniques. First, it only samples the top-K hashes as the similarity sketch, and builds indexes for features, instead of all chunks. Second, it uses a compact 2-byte key signature in the cuckoo hash table, and only consumes 6 bytes (including a 4-byte pointer to the on-disk metadata container) per index entry. The indexing memory usage for each document with M ($\leq K$) features is thus 6×M. In comparison, trad-dedup builds indexes for all unique chunks and uses the full 20-byte SHA1-hash as the signature key. As a result, it consumes 24 bytes of index memory for each unique chunk. As described in Section 4, by default, we use a metadata cache containing 128 pages of metadata (8 MB), and a 2000-entry source document cache (around 32 MB, assuming average document size of 16 KB) to strike a good balance between memory usage and I/O overhead. These (approximately) fixed-sized caches are used by both sDedup and trad-dedup to reduce the number of disk accesses, and are excluded in the evaluation below to highlight the analysis of index memory usage.

For these experiments, we used the Wikipedia and Stack Exchange data sets and again vary the chunk sizes of each deduplication approach. The results for the 20 GB Wikipedia dataset is shown in Fig. 8a. sDedup uses less memory than trad-dedup at all chunk sizes. The deduplication index memory usage for trad-dedup, which is proportional to the number of unique chunks, increases significantly as the chunk size decreases. In contrast, the magnitude of increase in index memory usage for sDedup is relatively small, because it sets an upper bound for the maximum number of features per sketch. This property gives sDedup much more freedom in the trade-off space between compression ratio and index memory usage when choosing suitable chunk size, whereas trad-dedup is concerned about increased index overhead when decreasing chunk sizes. When average chunk size is 64 bytes, trad-dedup consumes 780 MB memory for building internal deduplication indexes, which is more than 12× higher than sDedup. Such excessive memory usage explains why it is unrealistic to use a very small chunk size in the traditional chunk-based deduplication approaches. When chunk size is 256 bytes, combined with the results in Fig. 7a, we see sDedup is able to achieve 4× higher compression ratio than trad-dedup while only using 20% index memory. From a different angle, given the same memory constraints (e.g., 80 MB in this case), sDedup is able to achieve 38.9× compression ratio using a 64-byte chunk size, whereas trad-dedup can only obtain 2.3× compression ratio with a chunk size of 4 KB.

Fig. 8b shows the comparison of index memory usage using the 10 GB Stack Exchange dataset. Still, we have the same observation that sDedup uses much less index memory than trad-dedup at all chunk sizes. However, the memory usage for both approaches are higher than that with the Wikipedia dataset. This is mainly because the document size in the Stack Exchange dataset is relatively small, and the duplicate data is less than that in the Wikipedia dataset. Last, we show the results with the 4.3 GB Microsoft Exchange dataset. Similarly, sDedup significantly outperforms trad-dedup in terms of index memory usage for both 4 KB and 1 KB chunk sizes. Like in the Wikipedia dataset, decreasing chunk size in sDedup does not increase the index memory much, due to its upper bound on the number of sampled features. When decreasing chunk size from 4 KB to 1 KB, the index memory consumption of sDedup only increases by 40%, whereas that of trad-dedup increases by 234%.

5.4 Tuning Parameters

In addition to chunk size, sDedup provides a number of other tunable parameters that enable trade-offs among compression ratio, index memory usage, and deduplication throughput. Below we focus on two of

the results for sDedup would be better with the latest version.

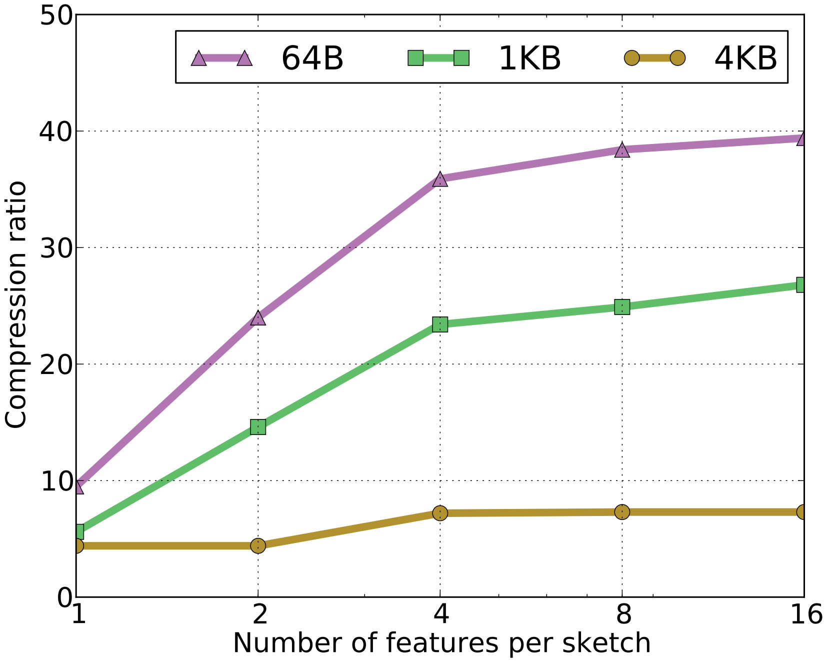


Figure 9: Sketch Size – The impact of the sketch size on the compression ratio for the Wikipedia dataset.

these parameters, namely the number of features per sketch (the K in top-K hashes when calculating the sketch), and the size of the source document cache. We evaluate the impact of varying these parameters on the deduplication performance, and explain how we choose the “sweetspot” in the trade-off space.

5.4.1 Sketch Size

As described in Section 3.1, the sketch size (K) quantifies the similarity coverage for each document. Fig. 9 shows the compression ratio by sDedup as a function of the sketch size. The experiments use the Wikipedia dataset with different chunk sizes to demonstrate the correlation among sketch size, compression ratio and chunk size. As we can see, for smaller chunk sizes such as 1 KB and 64 bytes, increasing the sketch size significantly improves the compression ratio. However, for larger chunk sizes such as 4 KB, the benefits of using a larger sketch size are not as significant. This is because with a relatively large chunk size, a non-trivial number of documents only consist of one chunk, for which increasing the sketch size does not improve the similarity coverage. For all chunk sizes, however, increasing sketch size beyond eight does not bring much additional gain in terms of compression ratio, but increases the indexing memory overhead. Therefore, we choose $K = 8$ as a reasonable trade-off for all experiments.

5.4.2 Source Document Cache Size

sDedup’s source document cache reduces the number of database queries that it uses to fetch source documents from the DBMS on the primary node. The miss ratio of this cache is proportional to the number of database queries, and thus it is important to select the right cache size that strikes a good balance between I/O overhead and memory usage. To more accurately evaluate the effectiveness of this cache we use a snapshot of the Wikipedia dataset that contains the revisions for all articles on a randomly selected day in September, 2009. This snapshot is approximately 3 GB and sorted by revision timestamp. We then replay the entire trace as document insertions into MongoDB with a cold cache, and report the final results when the cache is working

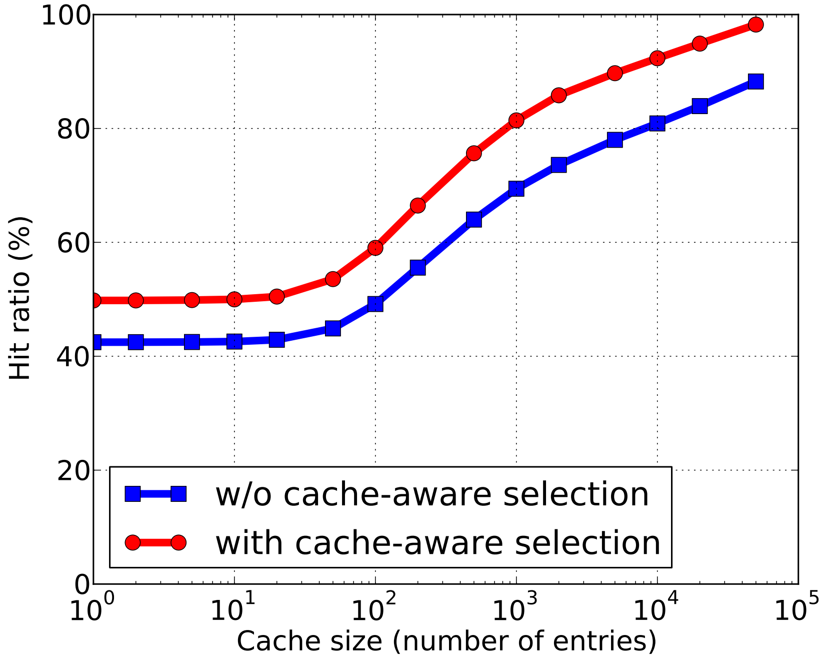


Figure 10: Source Document Cache Size – The effectiveness of the cache size for the source document cache with and without the cache-aware selection optimization.

in a steady state.

As described in Section 3.2, sDedup uses a cache-aware selection phase to rank candidates of similar documents. To evaluate the improvement by this mechanism, we replayed the same trace in two different settings of sDedup, one with cache-aware selection enabled and the other without.

The graph in Fig. 10 shows the hit ratio of the document cache as a function of varying cache sizes. Even without the cache-aware selection optimization, the results show that the document cache is effective in removing many database queries due to decent temporal locality in the document update pattern. The cache hit ratio is around 75% even with a relatively small cache size of 2000 entries ($\sim$ 32 MB, assuming average document size of $\sim$16 KB). With our cache-aware selection, the document cache hit ratio increases by $\sim$10% for all cache sizes. When the cache size is set to 2000 documents, adding cache-aware selection leads to a hit ratio of 87%, reducing the number of database queries (i.e., cache misses) further by 50% as compared to the case when the optimization is not applied. We use a cache size of 2000 entries for all experiments, because it consumes reasonable amount of memory while providing significant improvement in terms of deduplication throughput.

5.5 Processing Throughput

We next evaluate the internal deduplication throughput of sDedup, with a focus on the performance improvement as individual optimizations are added. We also show that using sDedup in MongoDB imposes negligible overhead in terms of insertion throughput, which makes it a practical approach for inline deduplication.

5.5.1 Deduplication Throughput

We calculate the internal processing throughput of sDedup by measuring the amount of time that it spends in each deduplication step on the primary node, as described in Section 3: (1) finding similar documents, (2)

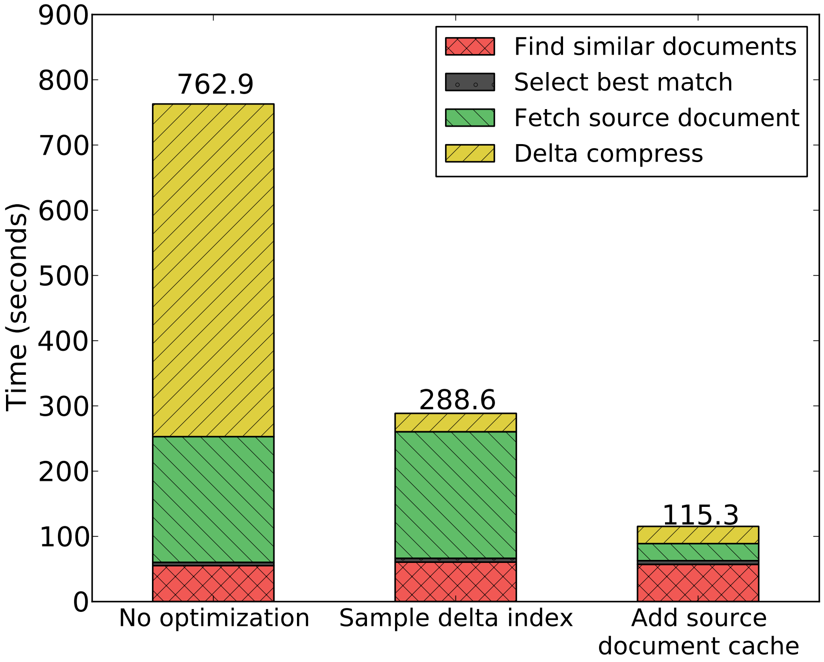


Figure 11: Deduplication Time Breakdown – Time breakdown of deduplication steps as individual refinements are applied.

selecting best match, (3) fetching source document, and (4) delta compression. We illustrate the reduction in deduplication time as optimizations are incrementally applied. Specifically we focus on the two most important optimizations in sDedup that significantly improve the deduplication speed: sampling source index in delta computation and adding a source document cache. We again use the 3 GB Wikipedia snapshot to execute three trials for this experiment and report the average results. By default, we use a chunk size of 256 bytes, a sketch size of at most $K = 8$ features and a source document cache size of 2000 entries.

Fig. 11 shows the effectiveness of individual optimizations by breaking down the time spent on each of the steps. Without these two optimizations, sDedup spends most of its time fetching source document from the DBMS and then computing the delta compression. As described in Section 3.3, this compression step is time consuming because the system needs to build indexes for each offset in the source document. As a result, the number of hash table insertions is approximately the same as the number of bytes in the source document. sDedup reduces the computation overhead in this process by only indexing a small subset of offsets on fixed positions, while only sacrificing negligible loss in terms of compression ratio. When sDedup uses a sampling ratio of $\frac{1}{32}$, the time spent on delta compression reduces by 95%, which makes fetching source document from the database the most time-consuming process. sDedup uses a document cache to reduce the I/O overhead involved in querying the database for source documents. As shown in the figure, adding a small source document cache of 2000 entries effectively reduces the source fetching time by approximately 87%, which corresponds to the results we obtain in Section 5.4.2.

5.5.2 Impact on Insertion Throughput

It is important for sDedup to be lightweight for it to be a feasible solution as an in-line deduplication engine. Ideally, integrating the deduplication system into MongoDB should have no negative impact on the original system performance. In this experiment, we evaluate the impact of the deduplication system on the DBMS's insertion throughput by executing the same workload with and without sDedup enabled. We use the 3 GB

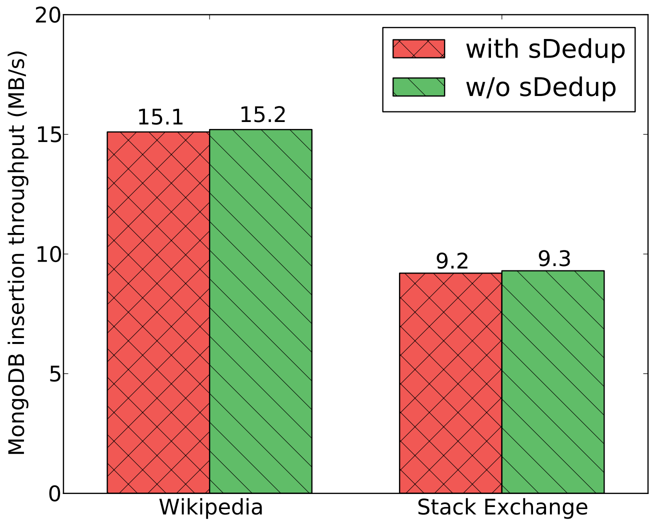
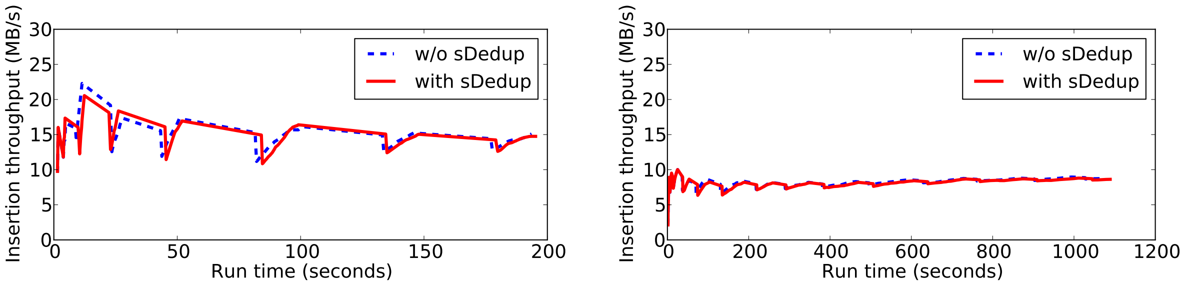


Figure 12: Impact on Insertion Throughput – The aggregate insertion throughput of MongoDB with and without sDedup enabled.



(a) Wikipedia

(b) Stack Exchange

Figure 13: Realtime MongoDB insertion throughput. – A comparison of MongoDB realtime insertion throughput with and without sDedup enabled. Enabling the deduplication engine in MongoDB imposes negligible overhead.

Wikipedia dataset and 10 GB Stack Exchange dataset for our evaluation. We insert the documents into MongoDB as fast as possible and measure the aggregate and real-time insertion throughput for each dataset. We only focus on the insertion throughput, because sDedup is only involved in document writes, but not read queries.

Fig. 12 shows the aggregate performance of MongoDB insertion throughput for both workloads. These results demonstrate that sDedup imposes almost negligible impact on the insertion throughput of the original system. We attribute this to its resource-efficient design and implementation, as well as its ability to reduce the work needed to transfer data over the network. The graphs in Fig. 13 show the sustained insertion throughput for the system using a three second moving average. Since MongoDB uses mmap for its memory management, the small dips in the DBMS's performance for both system configurations are due to the OS flushing dirty pages to disk. Again, these results show that that there is no discernible difference in the performance when sDedup is enabled.

6 Related Work

We are not aware of any previous work that uses deduplication techniques for reducing network bandwidth for replication in DBMSs. The common practice in such systems is to use compression. We have shown in this paper that deduplication can achieve significantly higher data transfer reduction over compression in document database scenarios. Much of the related work that we survey below comes from the storage systems area and is concerned with deduplication techniques.

Deduplication systems differ in the granularity at which they detect duplicate data. Microsoft Storage Server [10] and EMC's Centera [19] use file level duplication, LBFS [27] and Windows Server 2012 [18] use variable-sized data chunks obtained using Rabin fingerprinting [31], and Venti [30] uses individual fixed size disk blocks. *Winnowing* [32] has been used as a fingerprinting technique for identifying copying within large sets of documents.

There are two high level approaches for detecting duplicate data. The first approach looks for exact match on the unit of deduplication (e.g., chunk). Such systems [40, 24, 15, 17, 18] build an index of chunk hashes (using a strong hash like SHA-1) and consult it for detecting duplicate chunks. The second approach looks for similar units (chunks or files) and deduplicates them. If deduplication is at the chunk level, then the candidate chunk is delta encoded against a similar chunk [33]. If it is at the file level, then the candidate file is deduplicated by eliminating exact match chunks from a similar file [12] or by delta encoding against a similar file [38]. The technique of similarity based deduplication at document level in sDedup is comparable to delta encoding against a similar file. The work in [29] uses a consistent sampling scheme to find similar files, but their objective is to identify multiple sources for downloading a file in a peer-to-peer setting and not for deduplication.

There is also an inline (or, real-time) aspect in our work in that deduplication for replication traffic needs to keep up with updates at primary node. Much of prior work in data deduplication [40, 24] was done in the context of backup workloads (as opposed to primary workloads) where deduplication 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 the appliance model on premium hardware. Our work, being in the context of replicated document DBMSs for the cloud, is required to run on primary 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 deduplication on the data serving/storage server. In such systems, depending on the implementation, deduplication can happen either inline with new data (Sun's ZFS [11], Linux SDFS [1], iDedup [34]) or in the background as post-processing on the stored data (Windows Server 2012 [18], or provide both options (NetApp [5], Ocarina [6], Permabit [7]).

According to [39], in backup storage workloads, more than 90% of data is contained in files larger than 5GB, because backup software combine individual files into tar-like archives across backup periods. This number is around 1MB for primary storage workloads [26, 39], which mainly consist of files created and shared by end-users. Typical chunk sizes of 4-8 KB works well in removing redundancy in these relatively large files. Even whole-file deduplication can work reasonably well for primary storage workloads – evaluations in [26, 18] show that more than 50% of files are wholly duplicate. In contrast, document databases consist of predominantly small documents with small changes and almost no whole document duplicates. sDedup is designed based on this fundamental difference from traditional target workloads, and employs techniques that may otherwise be considered inefficient. For example, because document size is very small, it is reasonable to apply delta compression on document-level, whereas delta compression between very large files is inefficient.

There has been prior work on reducing network bytes for file transfer or synchronization. Most of this work assumes that previous versions of the same file are well-identified and deduplication (by eliminating identical chunks or delta encoding) happens only against prior versions of the same file [38, 35]. When deduplication can happen across different files, a mechanism is needed to identify similar files for deduplicating against. The basic technique of identifying features in objects so that similar objects have identical features was pioneered by Broder [13, 14] in the context of web pages. TAPER [23] aims to reduce network transfer for keeping two file system replicas synchronized by sending delta encoded files over the wire. It identifies similar files for delta compression by computing the number of matching bits on a Bloom filter based metric per file; this Bloom filter contains the hashes of all the chunks in the respective file.

7 Future Work

In addition to reducing replication bandwidth between replicas and primary nodes, the similarity-based deduplication approach could also be applied to reducing the amount of data transferred between clients and database servers. Specifically, clients' writes that are actually modified versions of existing documents could be deduplicated against the original documents before they are sent to the servers. Because the servers store the original versions being used by clients as source documents, the servers will be able to re-construct the encoded data using locally stored documents. For applications in which most write requests are incremental updates on existing data, such as with Wikipedia, the potential for client bandwidth reduction is significant.

8 Conclusion

sDedup is a similarity-based deduplication system that addresses the network bandwidth problem for replicated document databases. sDedup 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 significantly outperforms traditional chunk-based deduplication approaches in terms of compression ratio and indexing memory usage, while imposing negligible performance overhead.

References

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

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

[3] MongoDB Monitoring Service (November 2012). 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] 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.

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

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

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

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

[17] 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.

[18] 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.

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

[20] J. M. Hellerstein and M. Stonebraker. Readings in database systems. chapter Transaction Management, pages 238–243. 4th edition, 2005.

[21] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1998.

[22] S. Ihm, K. Park, and V. S. Pai. Wide-area network acceleration for the developing world. In *USENIX Annual Technical Conference*, 2010.

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

[24] 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.

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

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

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

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

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

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

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

[32] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

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

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

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

[36] 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.

[37] N. Tolia, M. Satyanarayanan, and A. Wolbach. Improving mobile database access over wide-area networks without degrading consistency. In *Mobisys*, 2007.

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

[39] 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.

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