Online Deduplication for Distributed Databases

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical & Computer Engineering

Lianghong Xu

B.S., Automation, Tsinghua University

Carnegie Mellon University
Pittsburgh, PA

September, 2016

For my parents and wife.

Keywords: deduplication, databases, delta compression

Abstract

The rate of data growth outpaces the decline of hardware costs, and there has been an ever-increasing demand in reducing the storage and network overhead for online database management systems (DBMSs). The most widely used approach for data reduction in DBMSs is block-level compression. Although this method is simple and effective, it fails to address redundancy across blocks and therefore leaves significant room for improvement for many applications.

This dissertation proposes a systematic approach, termed *similarity-based deduplication*, which reduces the amount of data stored on disk and transmitted over the network beyond the benefits provided by traditional compression schemes. To demonstrate the approach, we designed and implemented dbDedup, a lightweight record-level similarity-based deduplication engine for online DBMSs. The design of dbDedup exploits key observations we find in database workloads, including small item sizes, temporal locality, and the incremental nature of record updates. The proposed approach differs from traditional chunk-based deduplication approaches in that, instead of finding identical chunks anywhere else in the data corpus, similarity-based deduplication identifies a single similar data-item and performs differential compression to remove the redundant parts for greater savings.

To achieve high efficiency, dbDedup introduces novel encoding, caching and similarity selection techniques that significantly mitigate the deduplication overhead with minimal loss of compression ratio. For evaluation, we integrated dbDedup into the storage and replication components of a distributed NoSQL DBMS and analyzed its properties using four real datasets. Our results show that dbDedup achieves up to 37× reduction in the storage size and replication traffic of the database on its own and up to 61× reduction when paired with the DBMS's block-level compression. dbDedup provides both benefits with negligible effect on DBMS throughput or client latency (average and tail).

Acknowledgments

First and foremost, I would like to thank my PhD advisor Greg Ganger for his persistant guidance, support and encouragement throughout my graduate life. Greg has always been inspiring me with his rigorous attitude of science, commitment to good research, and optimistic personality. During the past several years, Greg has taught me a number of things, from technical to behavioral, but the most important lesson I learnt from him is to be confident. When I first entered Carnegie Mellon, I hardly had any experience in systems research, struggling with classes and losing research directions. Greg guided me through this stretch with firm support and his confidence in me greatly boosts my own. His encouragement made me decide to switch to a more challenging research topic in my fourth year that eventually leads to this dissertation. Without Greg, I would not have been where I am, and I feel so fortunate and grateful to have him as my advisor.

I would also like to give my special thanks to Andy Pavlo, who has been collaborating with me on my thesis project since the very early stage. Andy is energetic, diligent, and most of all, fun to work with. He regularly attended our weekly meetings and kept a close track of the project progress. He suggested expanding the scope of this research to distributed document-oriented databases and to online DBMSs in general, helped gather more datasets for evaluation, and set up connections with MongoDB when I needed help on implementation issues—not to mention that he actively helped with paper writing and slides polishing. Both Greg and Andy have been critical to make this dissertation come off.

This thesis work originates from the internship project I worked on at Microsoft Research Redmond back in 2013. The internship was a wonderful experience and I greatly appreciate the guidance and help from the researchers in the Cloud Computing and Storage group. Specifically, my mentor Sudipta Sengupta helped me to develop the research ideas, quickly get my hands dirty, and continued to collaborate on the project after the internship. Jin Li and Sanjeev Mehrotra gave detailed explanation on the data chunking portion of the prototype system and provided useful feedbacks when I encountered problems. I thank Andrei Marinescu for suggesting a motivating scenario from Microsoft Exchange cloud deployment and for providing the Exchange dataset. In addition, I would also like to thank all the friends I met during the internship who made the three months full of joy and excitement.

My research and study at CMU has been helped by a number of other faculty members, including Garth Gibson, Dave Andersen, Michael Kozuch, Onur Mutlu, Christos Faloutsos, Peter Steenkiste, Daniel P. Siewiorek, and David O'Hallaron. Thank you all for the insightful discussions and feedbacks at various seminars, PDL retreats and my Qual exam. Special thanks go to Garth, for kindly serving as my thesis committee, sharing his extensive experience in both academia and industry, and guiding me during my two teaching assistantships for the graduate storage systems class. I also want to thank a number of friends at PDL for the technical discussions or random chats: Ben Blum, Lei Cao, Jim Cipar, Henggang Cui, Bin Fan, Bin Fu, Aaron Harlap, Jesse Haber-Kucharsky, Rajat Kateja, Jin Kyu Kim, Elie Krevat, Likun Liu, Hyeotaek Lim, Yixin Luo, Junwoo Park, Kai Ren, Raja Sambasivan, Ilari Shafer, Alexey Tumanov, Matthew Wachs, Jinliang Wei, Lin Xiao, and Qing Zheng.

My life at CMU was made much easier with the help of an amazing group of administrative and technical staff in PDL. I would like to thank Karen Lindenfelser for various help on all kinds of miscellaneous stuff, and Joan Digney for helping polish my presentations and posters. I also want to thank Jason Boles, Chad Dougherty, Mitch Franzos, Michael Stroucken and Charlene Zang for providing reliable support when I need help with experiment setups.

I am especially grateful to my parents and my wife, for their unconditional support and unwavering confidence in me throughout my life. They provided me with enormous encouragement and help along this journey and I am greatly indebted to them for all their efforts.

Finally, I would like to thank the members and companies of the PDL Consortium (including Broadcom, Citadel, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung Information Systems America, Seagate Technology, Tintri, Two Sigma, Uber, Veritas and Western Digital) for their interest, insights, feedback, and support. This dissertation 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the prevalence of Web-based applications, more of today's data is stored in various forms of databases. Despite declining prices of hard drives, the hardware and maintanance cost for morden database management systems (DBMSs) keeps increasing due to exponential data explosion.

Database compression is one solution to this problem. For database storage, in addition to space saving, compression helps reduce the number of disk I/Os and improve performance, because queried data fits in fewer pages, leading to improved overall throughput for write-heavy workloads. For distributed databases replicated across geographical regions for high availability, there is also a strong need to reduce the amount of data transfer used to keep replicas in sync. Such replication requires 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.

The most widely used approach for data reduction in DBMSs is block-level compression [3, 18, 31, 40, 49, 52] on individual database pages or operation log batches. In this thesis, we argue that, although this method is simple and effective, it fails to address redundancy across blocks and therefore leaves significant room for improvement for many applications (e.g., due to app-level versioning in wikis or partial record copying in message boards). On the other hand, deduplication (dedup) has become popular in backup systems for eliminating duplicate content across an entire data corpus, often achieving much higher compression ratios. Unfortunately, traditional chunk-based dedup schemes are unsuitable for operational DBMSs, where many update queries modify a single record. The duplicate data in records is too fine-grained unless the system uses small chunk sizes. But, relatively large chunk sizes (e.g., 4–8 KB) are the norm to avoid huge in-memory indices and large numbers of disk reads.

1.1 Thesis Statement

Thesis Statement: For many database applications, greater reduction in storage usage and network bandwidth can be realized with similarity-based deduplication, which performs byte-level differential compression against a similar record already in the corpus.

This dissertation supports this thesis statement by describing an implementation of similarity-based dedup in a real database system (demonstrating its viability) and evaluating it with various real workloads (demonstrating its effectiveness). Specifically, we designed and implemented **dbDedup**, a lightweight scheme for online database systems that uses similarity-based dedup to compress individual records stored on disk and sent to remote replicas over network. Instead of indexing every chunk hash, dbDedup samples a small subset of chunk hashes for each new database record and then uses this sample to identify a similar record in the database. It then uses byte-level delta compression on the two records to reduce both online storage used and remote replication bandwidth. dbDedup provides higher compression ratios with lower memory overhead than chunk-based dedup and combines well with block-level compression.

We implemented dbDedup in the MongoDB DBMS [6] and evaluate its efficacy using four real-world datasets. Our results show that it achieves upto 37× reduction

(61× when combined with block-level compression) in storage size and replication traffic, significantly outperforming chunk-based dedup, while imposing negligible impact on the DBMS's runtime performance.

1.2 Contributions

This dissertation makes the following key contributions:

1. To our knowledge, it describes the first dedup system for operational DBMSs that reduces both database storage and replication bandwidth usage. It is also the first database dedup system that uses similarity-based dedup. It reveals the key observations of database workload characteristics including small item size, slight but distributed modifications, similar but uncorrelated records, as well as temporal locality and incremental nature of database updates. Based on these observations, it shows the limitation of block-level compression due to constrainted scope, demonstrates why tradidtional chunk-based dedup is a poor match for databases, and argues and proves that similarity-based dedup is a promising and effective approach.
2. It introduces several novel techniques that are critical to achieving acceptable dedup efficiency, enabling use for online database storage: It uses novel two-way encoding to efficiently transfer encoded new records (forward encoding) to remote replicas, while storing unencoded new records with encoded forms of selected source records (backward encoding). As a result, no decode is required for the common case of accessing the most recent record in an encoding chain. dbDedup uses a new technique called hop encoding to minimize the worst-case number of decode steps required to access a specific record in a long encoding chain. It uses a small yet effective source record cache to avoid most disk reads for similar records. During similarity selection, it uses a novel cache-aware selection techinque that greatly reduces the I/O overhead to fetch source records from the database by giving preference to candidate records that are present in the source record cache. In addition, it retains only the latest

copy of a record in the cache to reduce the memory overhead. To avoid performance overhead from updating source records, dbDedup introduces a lossy write-back delta cache tuned to maximize compression ratio while avoiding I/O contention. It also adaptively skips dedup effort for databases and records where little savings are expected. Finally, dbDedup uses an optimized delta compression algorithm to minimize the CPU and I/O overhead in delta compressing similar records.

3. It describes the design and implementation of similarity-based dedup in a distributed NoSQL DBMS (MongoDB), including a general-purpose end-to-end dedup workflow, full integration into the storage and replication components of DBMSs, as well as design choices and optimizations that are important for using deduplication in online DBMSs. It evaluates the system using four real-world datasets, quantifying the efficacy of dbDedup's approach.

Generality: This dissertation focuses on deduplication for databases because the applications are emblematic of the type of workloads (small objects with dispersed, fine-grained updates) for which dbDedup provides the most benefits. It is important to note, however, that our approach is applicable to other types of data stores (such as file systems and key-value stores) as well. The proposed deduplication framework and most of the techniques do not depend on implementation of a specific system backend. For instance, the notion of a "database record" used throughout this dissertation can be extended to any logical object, e.g., a file in the context of file systems, a document in document-oriented databases, a binary large object (BLOB), or even a chunk in the context of chunk-based deduplication systems.

The key design trade-off in dbDedup is to achieve higher compression ratio with delta compression at the cost of some I/O and computation overhead. The techniques proposed to alleviate this overhead, such as the encoding and caching mechanisms, are also generally applicable to other data stores, similar to the discussion above. 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 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, the chunk-based deduplication approach may suffice to provide a reasonably good compression ratio.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 motivates use of similarity-based dedup for database applications, categorizes dbDedup relative to other dedup systems, and discusses related work. Chapter 3 describes dbDedup's dedup workflow and mechanisms. Chapter 5 details dbDedup's implementation, including its integration into the storage and replication frameworks of a DBMS. We evaluate dbDedup in Chapter 6, conclude and discuss about future directions in Chapter 7.

Chapter 2

Background and Related Work

This chapter discusses why reducing storage usage and network bandwidth for DBMSs is desirable, motivates the potential value of deduplication in DBMSs, explains the two primary categories (exact match and similarity-based) of dedup approaches and why similarity-based is a better fit for dedup in DBMSs, puts dbDedup into context by categorizing previous dedup systems, and presents a survey of additional related work.

2.1 Need for Data Reduction in DBMSs

The rate of data growth outpaces the decline of hardware costs, and the demand has been increasing in reducing the storage and network overhead for online database management systems (DBMSs).

2.1.1 Database Storage Usage

There are several reasons for the strong need of reduction in database storage usage. While the price for commodity disks drops over years, data size grows at an even higher rate, not to mention that high-end disks are still not cheap. In addtion, modern DBMSs typically employ some levels of replication to ensure high data availability in face of failures. These systems also usually keep backups of data for long-term data recovery. All these factors add up to the total storage capacity.

Another important reason is to reduce the cost of data management, which has a strong correlation with the data size [12]. Reducing the size of the database helps shorten the time for administrative tasks, such as rebuilding indexes, backup, recovery, and bulk import/export.

The third reason is performance. When compressed, queried data could fit into fewer pages. In many cases, the benefits from the reduced I/O overhead largely offsets the computation work required for compression/decompression, leading to improved overall throughput for I/O intensive workloads.

2.1.2 Replication for Distributed DBMSs

Distributed DBMSs exploit replication to enhance data availability, just like other distributed storage systems. The design and implementation details of the replication mechanisms vary for different systems; 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).

Database replication involves propagating replication data from the primary to the secondary nodes in the form of 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.

2.1.3 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) [5] that provides continuous on-line backups using oplog replication and Google's B4 Software Defined Network system [42]. 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.

2.2 Why Dedup for Database Applications?

Deduplication is a specialized compression technique that identifies and eliminates duplicate content across a data corpus. 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). While dedup is widely used in file systems, it has not been fully explored in databases—most data reduction in DBMSs is based on block-level compression of individual database pages. 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 growing diversity of database management systems has changed the operational landscape. A record update typically involves reading the current version and writing back a highly similar record. Newly created records may also be similar to earlier records with only a small fraction of the content changed. Such redundancy creates great opportunity in data reduction for both database storage and replication bandwidth.

2.2.1 Compression Alone is Insufficient

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

Analytical DBMSs use more aggressive schemes (e.g., dictionary compression, run-length encoding) that significantly reduce the size of a database [20]. This is because these systems compress individual columns, and thus there is higher likelihood of duplicate data. And unlike in the above MySQL example, they also support query processing directly on compressed data.

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

We find that block-level compression alone is insufficient for data reduction in DBMSs. The size of a single database page is usually small (on the order of KBs) to amortize the disk I/O overhead on accessing records. Likewise, updates in replicated databases are sent in small batches (typically on the order of several MBs) to keep the secondary nodes reasonably up-to-date so that they can serve client read requests for applications that require bounded-staleness guarantees. At this small size, the database page or oplog batch mostly consists of updates to unrelated records, thus intra-batch compression yields only a marginal reduction.

To demonstrate this point, we loaded a Wikipedia dataset (see Section 6.1) into a modified version of MongoDB that compresses the oplog with *Snappy* [11]. We defer the discussion of our experimental setup until Chapter 6. We observe that compression only reduces the amount of data stored on disk as well as transferred from the primary to the replicas by around 2×.

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

2.2.2 Sources of Redundancy

For many applications, a major source of duplicate data is application-level versioning of records. While multi-version concurrency control (MVCC) DBMSs maintain historical versions to support concurrent transactions, they typically clean up older versions once they are no longer visible to any active transaction. As a result, few applications take advantage of versioning support provided by the DBMS to perform "time-travel queries". Instead, most applications implement versioning on their own when necessary. A common feature of these applications is that different revisions of one data item are written to the DBMS as completely unrelated records, leading to considerable redundancy that is not captured by simple page compression. Examples of such applications include websites powered by WordPress, which comprise 25% of the entire web [15], as well as collaborative wiki platforms such as Wikipedia [17] and Baidu Baike [1].

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

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

2.3 Similarity-based Dedup vs. Exact Dedup

Dedup approaches can be broadly divided into two categories. The first and most common ("exact dedup") looks for exact matches on the unit of deduplication (e.g., chunk) [29, 35, 36, 45, 76]. The second ("similarity-based dedup") looks for similar units (chunks or files) and applies delta compression to them [24, 59, 69]. For those database applications that do benefit from dedup, we find that similarity-based dedup outperforms chunk-based dedup in terms of compression ratio and memory usage, though it can involve extra I/O and computation overhead. This section briefly describes chunk-based dedup, why it does not work well for DBMSs, and why similarity-based dedup does.

2.3.1 Chunk-based Dedup

A traditional file dedup scheme based on exact matches of data chunks ("chunk-based dedup") [50, 55, 76] works as follows. An incoming file (corresponding to a new record in the context of DBMS) is first divided into chunks using Rabin-fingerprinting [56]; Rabin hashes are calculated for each sliding window on the data stream, and a chunk boundary is declared if the lower bits of the hash value match a pre-defined pattern. The average chunk size can be controlled by the number of bits used in the pattern. Generally, a match pattern of n bits leads to an average chunk size of 2^n B. For each chunk, the system calculates a unique identifier using a collision-resistant hash (e.g., SHA-1). It then checks a global index to see whether it has seen this hash before. If a match is found, then the chunk is declared a duplicate. Otherwise, the chunk is considered unique and is added to the index and the underlying data store.

While chunk-based dedup generally works well for backup storage workloads, it is rarely suitable for database workloads. There are two key aspects of databases that distinguish them from traditional backup or primary storage workloads. First, most duplication exists among predominantly small records. 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 [36, 48, 71], 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

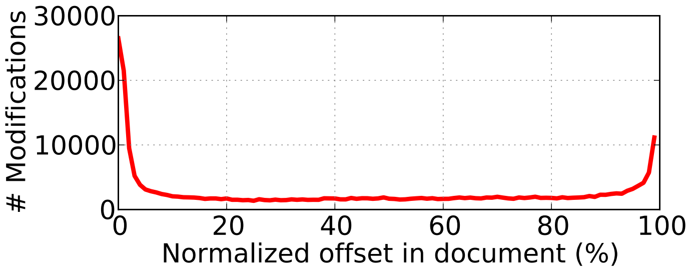


Figure 2.1: Distribution of record modifications for Wikipedia.

chunk size and building indexes for all the unique chunks imposes significant memory and storage overhead, which is infeasible for an inline deduplication system. dbDedup uses a small (configurable) chunk size of 256 B or less, and indexes only a subset of the chunks that mostly represent the record 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 databases are usually small (10s of bytes) but dispersed throughout the record. Fig. 2.1 illustrates this behavior by showing the distribution of modification offsets in the Wikipedia dataset. For the chunk-based approach, when the modifications are spread over the entire record, chunks with even slight modifications are declared as unique. Decreasing the chunk size alleviates this problem, but incurs higher indexing overhead.

While our approach employs delta compression based on the above observations of database workloads, it is important to note that whether or not to use it 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.

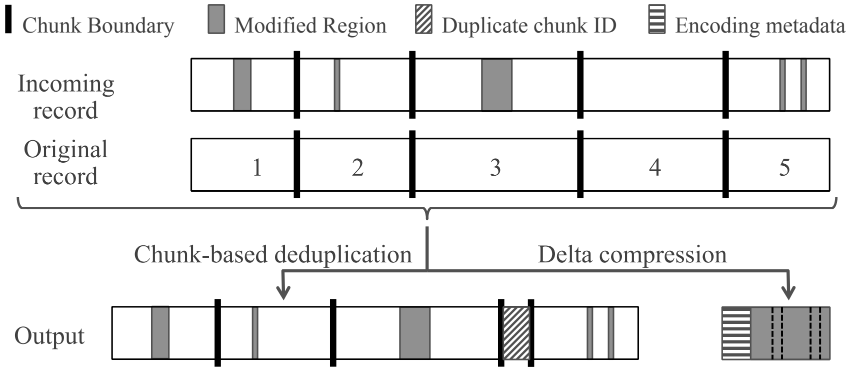


Figure 2.2: Comparison between chunk-based deduplication and similarity-based deduplication using delta compression for typical database workloads with small and dispersed modifications.

2.3.2 Better Compression with Similarity-based Dedup

dbDedup, in contrast, 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 record pairs to find the duplicate byte segments. Fig. 2.2 illustrates the effect of this behavior on the duplicated regions identified for the chunk-based and similarity-based deduplication approaches , given the update patterns. With byte-level delta compression, dbDedup is able to identify much more fine-grained duplicates and thus provide greater compression ratio than chunk-based dedup.

2.4 Categorizing Dedup Systems

Table 2.1 illustrates one view of how dbDedup relates to other systems using dedup, based on two axes: dedup approach (exact match vs. similarity-based) and dedup target (primary storage vs. secondary/backup data). To our knowledge, dbDedup is the first similarity-based dedup system for primary data storage, as well as being the first dedup system for on-line DBMSs addressing both primary storage and secondary data (the oplog).

Much prior work in data deduplication [22, 45, 59, 60, 76] was done in the context of backup data (as opposed to primary storage) where dedup does not need to keep up with primary data ingestion nor does it need to run on the primary (data-serving) node. Moreover, such backup workloads often run in appliances on premium hardware. dbDedup, being in the context of operational DBMSs, must run on primary data-serving nodes on commodity hardware and be frugal in its usage of CPU, memory, and I/O resources.

There has been recent interest in primary data dedup on the primary (data-serving) server but the solutions are mostly at the storage layer (and not at the data management layer, as in our work). In such systems, depending on the implementation, dedup can happen either inline with new data (Sun's ZFS [19], Linux SDFS [4], iDedup [62]) or in the background as post-processing on the stored data (Windows Server 2012 [36]), or provide both options (NetApp [21], Ocarina [9], Permabit [10]).

Systems in the lower middle column use a combination of exact and similarity-based dedup techniques at different granularities, but are in essence chunk-based dedup systems because they store hashes for every chunk. To the best of our knowledge, dbDedup is the first similarity-based dedup system for primary storage workloads that achieves data reduction on storage and network bandwidth requirement at the same time. This is because byte-level delta compression is traditionally considered expensive for on-line databases, due to the extra I/O and computation overhead relative to hash comparisons. As a result, previous systems either completely avoid it or use it when disk I/O is not a major concern. For example, SIDC [59] uses delta compression for network-level deduplication of replication streams; SDS [22] applies delta compression to large 16 MB chunks in backup streams retrieved by sequential disk reads. While dbDedup takes advantage of delta compression to achieve superior compression ratio, it uses a number of techniques to reduce the overhead involved, making it a practical dedup engine for on-line DBMSs.

	Exact Dedup		**Similarity-based Dedup**
	iDedup [62]		
	ZFS [19]		
	SDFS [4]		
Primary	Windows server 2012 [7]		***dbDedup***
	NetApp ASIS [21]		
	Ocarina [9]		
	Permabit [10]		
	DDFS [76]	Extreme binning [24]	
	Venti [55]	Sparse Indexing [45]	
	ChunkStash [32]	Silo [73]	SDS [22]
Secondary	DEDE [29]	SIDC [59]	
	HydraStor [34]	DeepStore [75]	

Table 2.1: Categorization of related work

2.5 Additional Related Work

Much of previous dedup work is discussed in Chapter 2. This section discusses some additional related work.

2.5.1 Deduplication

Chunk-based dedup systems differ in the granularity at which they detect duplicate data. Microsoft's Single Instance Storage (SIS) [7] and EMC's Centera [37] use file level duplication, LBFS [50] and Windows Server 2012 [36] use variable-sized data chunks obtained using Rabin fingerprinting [56], and Venti [55] uses individual fixed size disk blocks. Among content-dependent data chunking methods, Two-Threshold Two-Divisor (TTTD) [38], bimodal chunking algorithm [45], and regression chunking [36] produce variable-sized chunks.

dbDedup's target workload differs significantly from that of previous deduplication systems focused on backup [35, 45, 76] or primary storage [4, 9, 10, 19, 21, 36, 62]. For these workloads, more than 90% of duplication savings come from unmodified data chunks in large files on the order of MBs to GBs [48, 71], so typical chunk sizes of 4-8 KB work well. For user files, even whole-file deduplication may eliminate more than 50% of redundancy [36, 48]. dbDedup 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 6.2, this incurs significant indexing memory overhead. Instead, dbDedup finds a similar document and uses document-level delta compression to remove duplication with low memory and computation costs.

2.5.2 Database Compression

A number of database compression schemes haven been proposed during the past few decades. Most operational DBMSs that compress the database contents use page or block-level compression [3, 18, 31, 40, 49, 52]. Some use prefix compression, which looks for common sequences in the beginning of field values for a given column across all rows on each page. Just as with our dbDedup approach, such compression requires the DBMS to decompress tuples before they can be processed during query execution.

There are schemes in some OLAP systems that allow the DBMS to process data in its compressed format. For example, dictionary compression replaces recurring long domain values with short fixed-length integer codes. This approach is commonly used in column-oriented data stores [20, 39, 57, 78]. These systems typically focus on attributes with relatively small domain size and explore the skew in value frequencies to constrain the resulting dictionary to a manageable size [25]. The authors in [63] propose a delta encoding scheme where every value in a sorted column is represented by the delta from the previous value. Although this approach works well for numeric values, it is unsuitable for strings.

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

2.5.3 Delta Compression

There has been much previous work on delta compression, including several general-purpose algorithms based on the Lempel-Ziv approach [77], such as vcdiff [23], xDelta [46], and zdelta [68]. Specialized schemes can be used for specific data formats (e.g., XML) to improve compression quality [30, 44, 58, 72]. The delta compression algorithm used in dbDedup is adapted from xDelta, to which the relationship is discussed in Section 3.4.

Delta compression has been used to reduce network traffic for file transfer and synchronization protocols. Most systems assume that previous versions of the same file are explicitly identified by the application, and duplication only exists among prior versions of the same file [64, 69]. On exception is TAPER [41], which reduces network transfer for synchronizing file system replicas by sending delta-encoded files; it identifies similar files by computing the number of matching bits on the Bloom filters generated with the files' chunk hashes. dbDedup identifies a similar record from the data corpus without application guidance and therefore is a more generic approach than most of these previous systems.

The backward-encoding technique used in dbDedup is inspired by versioned storage systems such as RCS [66] and XDFS [46]. While these systems explicitly maintain versioning lineage for all the files, dbDedup builds the encoding chain purely based on similarity relationships between records, and thus does not require system-level support for versioning. [60] uses delta encoding for deduplicated backup storage. It uses forward encoding and only supports a longest encoding chain length of two. dbDedup uses hop encoding on top of backward encoding to reduce the worst-case source retrievals for read requests. In addition, it uses several novel caching mechanisms to further mitigate the I/O overhead involved in reading and updating encoded records.

2.5.4 Similarity Detection

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 [47]. Spring et al. [61] use this technique to identify repeated byte ranges on cached network packets to reduce the redundant network traffic. Broder [26, 27] 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. [43] adapt this method and combine it with compression and delta encoding to improve efficiency. Several other systems [24, 45, 53] take an alternative approach using a representative subset of chunk hashes (IDs) as the feature set. dbDedup uses a similar approach to extract features by sampling first-K chunk IDs, but only uses them to identify similar records rather than for chunk-level deduplication.

2.5.5 My Other Work

Before this dissertation work, my research was focused on elastic storage systems. These systems can be expanded or contracted to meet current demand, allowing servers to be turned off or used for other tasks. However, the usefulness of an elastic distributed storage system is limited by its agility: how quickly it can increase or decrease its number of servers. Due to the large amount of data they must migrate during elastic resizing, state-of-the-art designs usually have to make painful tradeoffs among performance, elasticity and agility.

We proposed a new elastic storage system, called SpringFS [74], that can quickly change its number of active servers, while retaining elasticity and performance goals. SpringFS uses a novel technique, termed bounded write offloading, that restricts the set of servers where writes to overloaded servers are redirected. This technique, combined with the read offloading and passive migration policies used in SpringFS, minimizes the work needed before deactivation or activation of servers. Analysis of real-world traces from Hadoop deployments at Facebook and various Cloudera customers and experiments with the SpringFS prototype confirm SpringFS's agility, show that it reduces the amount of data migrated for elastic resizing by up to two orders of magnitude, and show that it cuts the percentage of active servers required by 67-82%, outdoing state-of-the-art designs by 6-120%.

Chapter 3

Deduplication Workflow in dbDedup

dbDedup uses similarity-based dedup to achieve good compression ratio and low memory usage simultaneously. It differs from traditional chunk-based deduplication systems that break input data-item into chunks and find identical chunks stored anywhere else in the data corpus (e.g., the original database). Fig. 3.1 shows the overview of the dedup workflow used when preparing updated record data for local storage and remote replication. Fig. 3.2 illustrates the workflow with further details including data structures and actions involved in transforming each new record into a delta-encoded representation. During insert or update queries, new records are written to the local oplog, and dbDedup encodes them in the background, off the critical path. Four key steps are (1) extracting similarity features from a new record, (2) looking in the deduplication index to find a list of candidate similar records in the database corpus, (3) selecting one best record from the candidates, and (4) performing delta compression between the new and the similar record to compute encoded forms for local storage and replica synchronization.

3.1 Feature Extraction

As a first step in finding similar records in the database, dbDedup extracts similarity features from the new record using a content-dependent approach. dbDedup

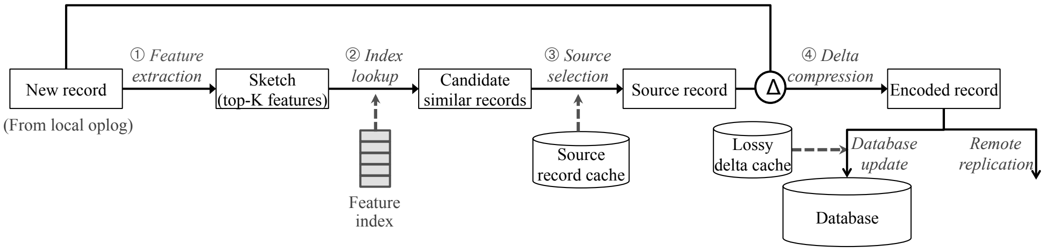


Figure 3.1: An overview of dbDedup Workflow – (1) Feature Extraction, (2) Index Lookup, (3) Source Selection, and (4) Delta Compression.

divides the new record into several variable-sized data chunks using the Rabin Fingerprinting algorithm [56] that is widely used in many chunk-based dedup systems. Unlike these systems that index a collision-resistant hash (e.g., SHA-1) for every unique chunk, dbDedup calculates a (weaker, but computationally cheaper) MurmurHash [8] for each chunk and only indexes a representative subset of the chunk hashes. dbDedup adapts a technique called *consistent sampling* [53] to select representative chunk hashes, which provides better similarity characterization than random sampling. It sorts the hash values in a consistent way (e.g., by magnitude from high to low), and chooses the first-K[1] hashes as the *similarity sketch* for the record. Each chunk hash in the sketch is called a *feature*—if two records have one or more common features, they are considered to be similar.

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

[1] For records with less than K chunks, the sketch size is less than K.

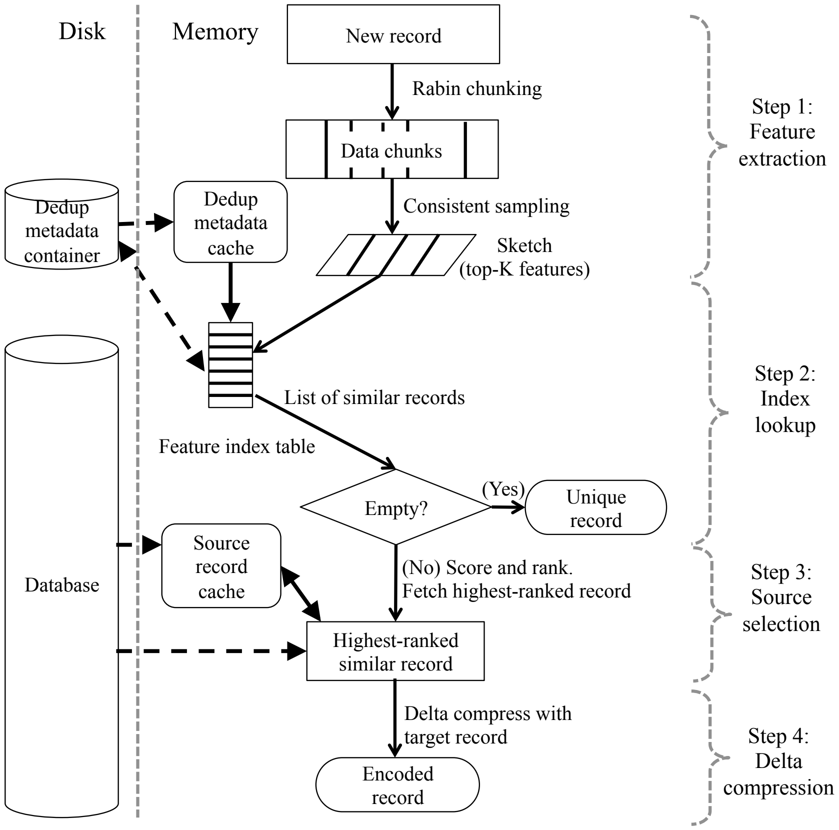


Figure 3.2: Detailed Workflow and Data Structures – A new record is converted to a delta-encoded form in four steps. On the left are the two disk-resident data stores involved in this process: the dedup metadata container (see Section 5.2) and the original database. The remainder of the structures shown are memory resident.

First-K featuers versus random sampling: The idea of sorting chunk hashes before selection is borrowed from handprint [53]. This technique significantly increases the probability of finding similarities between records as compared to a naïve random sampling algorithm. After sorting, shared chunk hashes from two records, if any, would be in the same order. As a result, if the sketches of two records A and B contain shared features, these features would corresponds to **the same chunks**. In contrast, with random selection, even if the selected chunk hashes belong to the intersection, they may be different chunks.

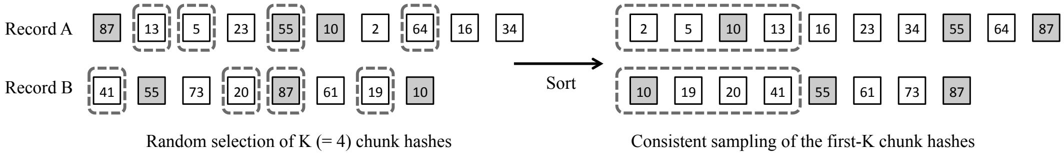


Figure 3.3: Consistent sampling versus random selection – Sorting before selection improves the probability of similarity detection. In both cases, one shared chunk hash is selected from each of the record. Consistent sampling ensures that the shared hashes correspond to the same chunk in the two records.

Fig. 3.3 illustrates the efficacy of the first-K approach in an intuitive way.[2] Given two records A and B with several shared chunks (shown as shaded). With random selection, even if some of the selected chunks are shared among the two records, they do not correspond to the same chunks and thus records A and B will not be identified as similar. On the other hand, after sorting, as long as the first-K chunk hashes contain at least one shared hash in both records, the shared ones would match to the same chunk.

3.2 Index Lookup

dbDedup's approach to finding similar records in the corpus (combining feature extraction and index lookup) is illustrated in Algorithm 1. For each extracted feature, dbDedup checks its internal feature index to find existing records that share that feature with the new record. After each search, dbDedup inserts the corresponding feature to the index for future lookup. Since dbDedup is an online dedup system, it is imperative that this index lookup process is fast and efficient. dbDedup achieves this by building an in-memory feature index (see Section 5.2) that uses a variant of Cuckoo hashing [32, 51] to map features to records. If a record has at least one feature in common with the new record, it is added to the list of candidate sources. The feature index stores at most K entries for each record in the corpus (one for each feature). As a result, the size of this index is smaller than the correspond-

[2] A more detailed quatitative analysis of the probability of similarity detection is described in [53].

Algorithm 1 Feature Extraction + Index Lookup

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

    $dataChunks \leftarrow$ RabinFingerprint($newRecord$)
    $chunkHashes \leftarrow$ 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($feature$)
        $simRecord \leftarrow$ IndexLookup($feature$)
        $candidates$.append($simRecord$)
        $i \leftarrow i + 1$
    end while
    for each $feature$ in $sketch$ do
        IndexInsert($feature, newRecord$)
    end for
    return $candidates$
end procedure
```

ing 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 [53]. For our experimental analysis, we found $K = 8$ is sufficient to identify similar records with reasonable memory overhead, and we use it as a default value for all experiments unless otherwise noted. dbDedup combines the lookup results for all first-K features and generates a list of existing similar records as input for the next step.

Algorithm 2 Source Selection

```
function SOURCESELECTION($candidates$)
    $scores \leftarrow empty$
    $maxScore \leftarrow 0$
    $bestMatch \leftarrow NULL$
    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 function
```

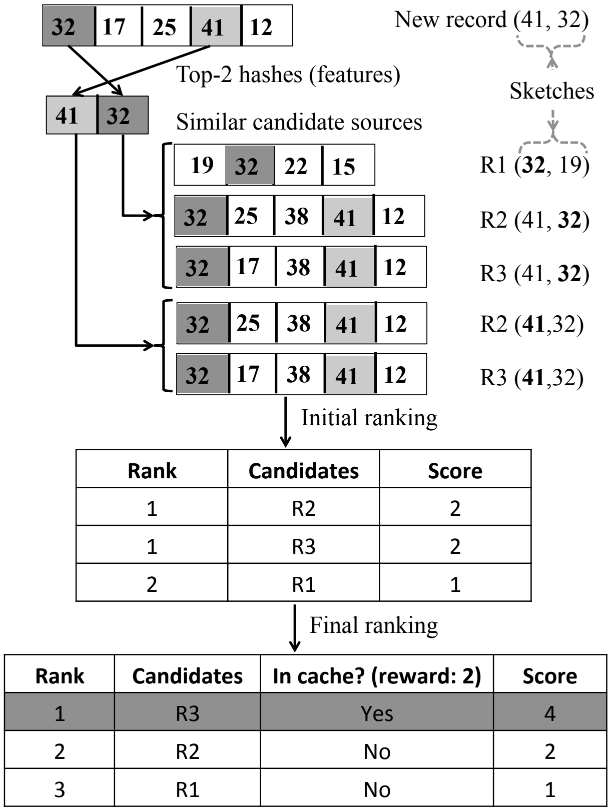
3.3 Source Selection

After dbDedup identifies a list of candidate source records, it next selects one of them to use. If no similar records are found, then the new record is declared unique and thus is not eligible for encoding. The index lookup results from the previous may contain multiple candidate similar records, yet dbDedup only chooses one of them to delta compress the new record in order to minimize the overhead involved. While most previous similarity selection algorithms make such decisions purely based on the similarity metrics of the inputs, dbDedup adds consideration of system performance, giving preference to candidate records that are present in the source record cache (see Section 4.2). We refer to this selection technique as *cache-aware selection*.

Algorithm 2 describes the mechanism dbDedup uses to choose the best source record out of a number of similar records. Fig. 3.4 provides an example of this selection process. dbDedup first assigns an initial score for each candidate similar record based on the number of features it has in common with the new record. Then, dbDedup increases that score by a reward if the candidate record already resides in the cache. The candidate with the highest score is then selected as the input for delta compression. While cache-aware selection may end up choosing a record that is sub-optimal in terms of similarity, we find it greatly reduces the I/O overhead to fetch source records from the database. We evaluate the effectiveness of cache-aware selection and its sensitivity to the reward score in Section 6.5.

3.4 Delta Compression

The last step in dbDedup workflow is to perform delta compression between the new record and the selected similar record. The system first checks its internal record cache for the source record; on miss, it retrieves the record from the database. dbDedup uses only one source record 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 record from the corpus is the dominating factor in this step, especially for the databases with small records. This is the same reasoning that underscores the benefits of dbDedup over chunk-based deduplication: our approach only requires one fetch per new record to reproduce the original record, versus one fetch per chunk. We describe the details of the encoding techniques in Section 4.1 and the compression algorithms in Section 5.3.



Rank	Candidates	Score
1	R2	2
1	R3	2
2	R1	1

Rank	Candidates	In cache? (reward: 2)	Score
1	R3	Yes	4
2	R2	No	2
3	R1	No	1

Figure 3.4: Example of Source Record Selection – The top two ($K = 2$) hashes of the new record are used as the features of its sketch (41, 32). The numbers in the records' chunks are the MurmurHash values. Records with each feature are identified and initially ranked by their numbers of matching features. The ranking increases if the candidate is in dbDedup's source record cache.

Chapter 4

Mitigating Deduplication Overhead

This chapter describes dbDedup's mechanisms to mitigate the dedup overhead, including its encoding and caching schemes to alleviate overhead from delta compression, and approaches to avoiding wasted effort on low-benefit dedup actions.

4.1 Encoding for Online Storage

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

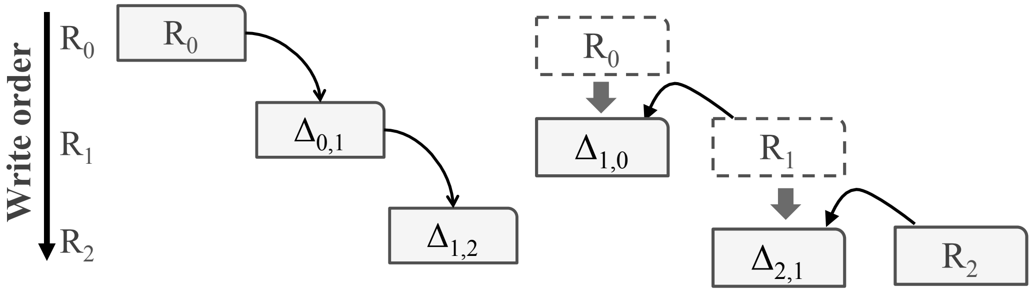
dbDedup greatly alleviates the painful tradeoff between compression gains and access speed in delta encoded storage with two new encoding schemes. It uses a *two-way encoding* technique that reduces both remote replication bandwidth and database storage, while optimizing for common case queries. In addition,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

4.1.1 Two-way Encoding

After a candidate record is selected from the data corpus, dbDedup generates the byte-level difference between the candidate and the new record in dual directions, using a technique that we call *two-way encoding*. For network transmission, dbDedup performs *forward encoding* (Fig. 4.1a), which uses the older (i.e., the selected candidate) record as the source and the new record as the target. After the encoding, the source remains in its original form, while the target is encoded as a reference to the source plus the delta from the source to the target. dbDedup sends the encoded data, instead of the original new record, to remote replicas. Using forward-encoding for network-level deduplication is a natural design choice, because it allows the replicas to easily decode the target record using the locally stored source record.

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

Instead, dbDedup uses *backward encoding* (Fig. 4.1b) for local storage to optimize for read queries to recent records. That is, for local storage, dbDedup performs delta compression in the reverse temporal order, using the new record as the source and the similar candidate record as the target. As a result, the most recent record in an encoding chain is always stored unencoded. Read queries to the latest version thus incur no decoding overhead at all. Although backward encoding is optimized for reads, it creates two potential issues. First, it amplifies the number of write operations, since an older record selected as a source needs to be updated to the encoded form. To mitigate the write amplification, dbDedup caches backward-



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

Figure 4.1: Illustration of two-way encoding – dbDedup uses forward encoding to reduce the network bandwidth for replica synchronization while using backward encoding to compress database storage.

encoded records to be written back to the database and delays the updates until system I/O is relatively idle, which we discuss in more detail in Section 4.2. A second issue arises when an older record is selected as the source. The existing data (a delta from its current base record) is replaced by the delta from the new record. Since backward encoding realizes space savings by updating delta sources, such *overlapped encoding* (Fig. 4.2) on the same source records can lead to some compression loss. Forward encoding, in contrast, naturally avoids this problem since no writeback is required. Fortunately, we find overlapped encoding is not common in real-world applications—most $(> 95\%)$ updates are incremental based on the latest version (see Section 6.2).

dbDedup performs delta encoding between new and candidate records in two directions, yet it only incurs the computation overhead of one encoding pass. It achieves this by first generating the forward-encoded data and then efficiently transforming it into the backward delta at memory speed. We call this process *re-encoding* and detail the algorithm in Section 5.3.

4.1.2 Hop Encoding

As discussed above, using backward encoding minimizes the decoding overhead for reading recent records, but it may still incur excessive source retrieval time for occa-

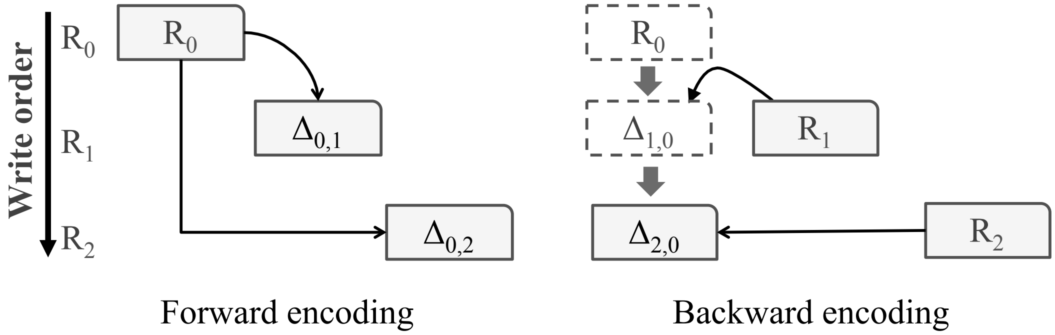


Figure 4.2: Overlapped encoding – Backward encoding may lead to compression loss when an older record is selected as the source. In this example, when R_0 is selected as the source for R_2, backward encoding leaves R_1 and R_2 both unencoded.

sional queries to older records (e.g., a specific version of a Wikipedia article). Prior work on delta encoded storage [28, 46] used a technique called *version jumping* to cope with this problem, by bounding the worst-case number of source retrievals at the cost of lower compression benefits. The idea is to divide the encoding chain into fixed-size clusters, where the last record in each cluster, termed *reference version*, is stored in its original form and the other records are stored as backward-encoded deltas. Doing so bounds the worst-case retrieval times to the cluster size but results in lower compression ratio, because the reference versions are not compressed. As the encoding cluster size decreases, the compression loss can increase significantly, since deltas are usually much smaller than base records.

dbDedup uses a novel technique that we call *hop encoding*, which preserves the compression ratio close to standard backward encoding, while achieving comparable worst-case retrieval times to the version jumping approach. As illustrated in Fig. 4.3, extra deltas are computed between particular records and others some distance back in the chain, in a fashion similar to skip lists [54]. We call these records *hop bases* and the minimum interval between them *hop distance*, noted as H. Hop encoding employs multiple levels of indirection to speed up the decoding process, with the interval on level L being H^L. Decoding a record involves first tracing back to the nearest hop base in logarithmic time and then following the encoding chain starting with it.

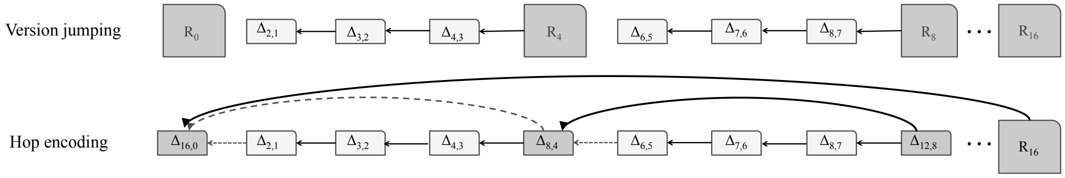


Figure 4.3: Hop encoding – A comparison of hop encoding and version jumping with an encoding chain of 17 records. Shaded records (R_0, R_4, etc.) are hop bases (reference versions), with a hop distance (cluster size) of 4. Hop encoding provides comparable decoding speed as version jumping while achieving a compression ratio close to standard backward encoding.

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

Table 4.1: Summary of the different encoding schemes – Hop encoding largely eliminates the painful tradeoff between space savings and decoding speed. N is the length of the encoding chain, and H denotes the hop distance (cluster size for version jumping). S_b and S_d refer to the size of a base record and a delta respectively, where $S_b \gg S_d$ in most cases. These sizes obviously vary for different records. Here we use the general notation for ease of reasoning.

Table 4.1 summarizes the trade-offs among three encoding techniques in terms of storage usage, worst-case number of retrievals, and the extra number of writebacks. For hop encoding, the number of worst-case source retrievals is close to that of version jumping (H). But because hop bases are stored in an encoded form, the compression ratio achieved is much higher than version jumping and comparable to standard backward encoding. All three encoding schemes incur some amount of write amplification, but the difference becomes negligible as hop distance increases. We present a more detailed comparison in Chapter 6.

4.2 Caching for Delta-encoded Storage

Delta encoded storage, due to its “chained” property, merits specialized caching mechanisms. Exploiting this property, dbDedup only caches a few key nodes in a given encoding chain, maximizing memory efficiency while eliminating most I/O overhead for accessing encoded records. It uses two specialized caches: a source record cache that reduces the number of database reads during encode and a lossy write-back delta cache that mitigates write amplification caused by backward encoding.

4.2.1 Source Record Cache

A key challenge in delta-encoded storage is the I/O overhead to read the base data from the disk as input for delta compression or decompression. Unlike chunk-based deduplication systems, dbDedup does not rely on having a deduplicated chunk store, either of its own or as the database implementation. Instead, it directly uses a source record from the database and fetches it whenever needed in delta compression and decompression. Querying the database to retrieve records, 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, dbDedup’s queries to retrieve source records will compete for resources with normal database queries and impact the performance of client applications.

dbDedup uses a small record cache to eliminate most of its database queries. The design of the record cache exploits the high degree of temporal locality in record updates of workloads that dedup well. For instance, updates to a Wikipedia article, forum posts to a specific topic, or email exchanges in the same thread usually occur within a short time frame. So, the probability of finding a recent similar record in the cache is high, even with a relatively small cache size. Another key observation is that the updates are usually incremental (based on the immediate previous update), meaning that two records tend to be more similar if they are closer in creation time.

Algorithm 3 Source Record Cache Replacement

```
procedure UpdateRecordCache($srcRecord$, $tgtRecord$)
    if $srcRecord$ in $srcRecordCache$ then
        $srcRecordCache$.remove($srcRecord$)
    end if
    if $srcRecordCache$.size() $\geq cacheSize$ then
        $srcRecordCache$.LRURemove(1)
    end if
    $srcRecordCache$.LRUAdd($tgtRecord$)
end procedure
```

Based on the observations above, the source record cache retains the latest record of an encoding chain in the cache. To accelerate backward encoding of hop bases, dbDedup additionally caches the latest hop bases in each hop level.[1] When a new record arrives, if dbDedup identifies a similar record in the cache (which is the normal case due to the cache-aware selection technique described in Section 3.3), it replaces the existing record with the new one. If the new record is a hop base, dbDedup replaces its adjacent bases accordingly. When no similar source is found, dbDedup simply adds the new record to the cache, and evicts the oldest record in a LRU manner if the cache becomes full.

Algorithm 3 describes the cache replacement process that occurs when dbDedup looks for a source record in its cache. Upon a hit, the record is directly fetched from the record cache, and its cache entry is replaced by the target record. Otherwise, dbDedup retrieves the source record using a database query and insert the target record into the cache. In either case, the source record is not added to the cache because it is older and expected to be no more similar to future records than the target record. When the size of the cache is reached, the oldest entry is evicted in a LRU manner.

dbDedup also uses a source record cache on each secondary node to reduce the number of database queries during delta decompression. Because the primary and secondary nodes process record updates in the same order, as specified in the oplog,

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

their cache replacement process and cache hit ratio are almost identical.

4.2.2 Lossy Write-back Delta Cache

As discussed in Section 4.1, backward encoding optimizes for read queries, but introduces some write amplification—record insertion triggers the source record to be delta compressed and updated on disk. The problem is exacerbated somewhat with hop encoding, where inserting a hop base causes writeback not only to the source record, but also to the adjacent bases on each hop level. For heavy insertion bursts, this could significantly increase the number of disk writes, leading to visible performance degradation.

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

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

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

4.3 Avoiding Unproductive Dedup Work

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

4.3.1 Automatic Deduplication Governor

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

4.3.2 Adaptive Size-based Filter

In our observation of several real-world database datasets (see Section 6.1), we find that most dedup savings come from a small fraction of the records that are larger in size. Fig. 4.4 shows the cumulative distribution function (CDF) of record size and the weighted CDF by contribution to space saving for the four workloads used in our experiments. For these datasets, the 60% largest records account for approximately

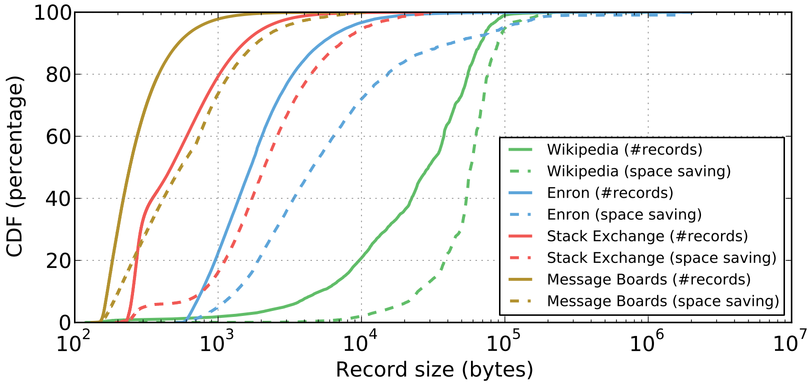


Figure 4.4: Size-based deduplication filter.

90–95% of data reduction. In other words, if we only deduplicate records larger than the 40%-tile record size, we can reduce dedup overhead by 40% while only losing 5–10% of the compression ratio.

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

Chapter 5

Implementation

This chapter describes dbDedup implementation details, including how it fits into DBMS storage and replication frameworks, its feature indexing mechanisms, and internals of its delta compression algorithms.

5.1 Integration into DBMSs

While implementation details vary across DBMSs, we illustrate the integration of dbDedup into a DBMS's storage and replication frameworks using a simple distributed setup consisting of one client, one primary node and one secondary node, as shown in Fig. 5.1. In this example, we use a setting with single-master, push-based, asynchronous replication that propagates updates in the form of oplogs. The integration of dbDedup into the storage and replication components is not necessarily coupled. For cases where only the replication bandwidth is the primary concern, we describe the integration of dbDedup simply into the replication component. We then describe how such integration is generally applicable to other replication settings with slight modifications.

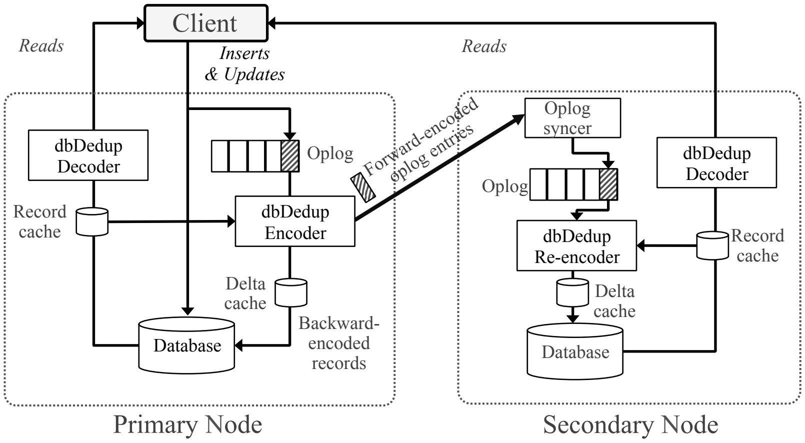


Figure 5.1: Integration of dbDedup into a DBMS. – An overview of how dbDedup fits into the storage and replication mechanisms of an example database system and the components that it interacts with. dbDedup deduplicates data to be stored and sent when a secondary requests new oplog entries. It checks each oplog entry before it is sent to the secondary and then again on the replica to reconstruct the original entries.

5.1.1 Integration into Storage and Replication Components

Below, we illustrate dbDedup's integration into the storage and replication components of an example DBMS by describing its behavior for primary operations.

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

With dbDedup, a new record is first stored in the local oplog. Later, when preparing to store the record or send it to a replica, it is processed by the dbDedup encoder following the deduplication steps described in Chapter 3. If dbDedup successfully selects a similar record from the existing data corpus, it retrieves the content of the similar record by first checking the source record cache. On cache misses, it directly reads the record from the underlying storage. It then applies bidirectional delta compression to the source and target records to generate the forward-encoded form of the new record and the backward-encoded form of the similar record. dbDedup inserts the new record to the primary database in its original form and caches the backward-encoded similar record in the lossy write-back cache until system I/O becomes idle. Then, dbDedup appends the forward-encoded record to the primary oplog that is transferred to the secondary during replica synchronization.

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

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

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

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

Delete: If the reference count for record to be deleted is zero, then the deletion proceeds as normal. Otherwise, dbDedup marks it as deleted but retains its content. Any client reads to a deleted record returns an empty result, but it can still serve as a decoding base for other records referencing it. When the reference count of a record drops to zero, dbDedup removes it from the database and decrement the reference count of its base record by one.

Read:

If the queried record is stored in its raw form, then it is directly sent to the client just like the normal case. If the record is encoded, then the dbDedup's decoder returns it back to its original form before it is returned to the client. During decoding, the decoder fetches the base record from the source record cache (or storage, on cache miss) and reconstructs the queried record using the stored delta. If the base record itself is encoded, the decoder repeats the step above iteratively until it finds a base record stored in its entirety.

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

5.1.2 Integration into Replication Component

When the network bandwidth for replication services rather than database storage usage is the major concern (e.g., for geo-replicated databases with sufficient storage

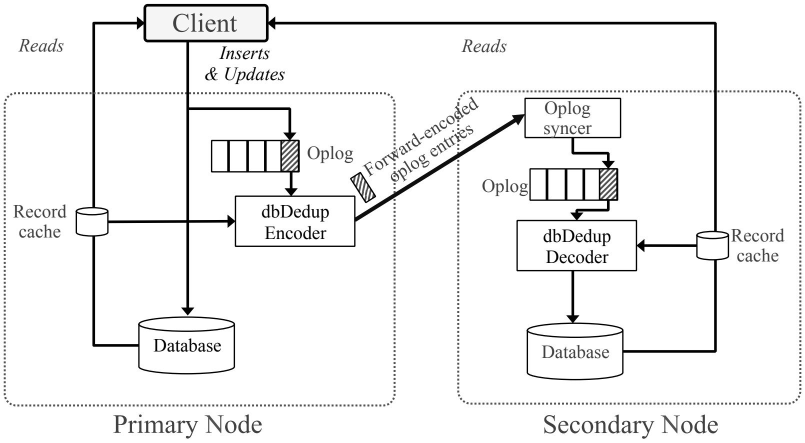


Figure 5.2: Integration of dbDedup into a DBMS's replication component. – An overview of how dbDedup fits into the replication mechanism of an example database system. Oplog entries containing insertions and updates are deduplicated before sent to the remote replicas. All records are stored in entirety on disk so there is no need for a write-back record cache buffering encoded data.

capacity), it is reasonable to only integrate dbDedup into the replication component of the DBMS to completely avoid the overhead of maintaining and accessing delta-encoded storage.

The simplified integration is shown in Fig. 5.2. With dbDedup, 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 Chapter 3, but only using forward-encoding instead of bidirectional delta compression in the last step. If the entry is marked for deduplication, then it is appended to the batch as a special message the dbDedup 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, dbDedup 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.

Because dbDedup is not integrated to the storage layer in this example, all database records are stored in their entirety, eliminating the need to use a write-back cache to store the backward-encoded data. On the other hand, the source record cache is still needed to reduce the number of database queries to retrieve source records, and to serve as an input to the cache-aware selection of the optimal source record.

Our approach differs from previous data reduction techniques for remote replication in several ways [59, 65, 67]. In these other systems, neither the senders nor receivers in the protocol are replicas, so the sender does not know what data is 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 records without explicit coordination for each record.

dbDedup's replication protocol is optimistic in that it assumes that the secondary will have the source record 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 record 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 record. This eliminates the need to reconstruct records when bandwidth savings are not being realized. In our evaluation in Chapter 6 with the Wikipedia dataset, we observe that only 0.05% of the oplog entries incur a second round trip during replication.

We next describe dbDedup's protocol for other replication mechanisms. The example above involves only one primary server that can receive writes, and thus only the primary maintains the deduplication index. 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. dbDedup supports both synchronous and asynchronous replication because it is orthogonal to the consistency setting. We show in Chapter 6 that dbDedup 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, dbDedup 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 dbDedup provides the most benefits.

5.2 Indexing Records by Features

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

To ensure fast deduplication, dbDedup'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 record's "dedup metadata" (including its sketch and database location) for each feature. In our implementation, the database location for each record is encoded with a 52 B database namespace ID and a 12 B record 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, dbDedup uses a two-level scheme. It stores the dedup metadata in a log-structured disk container and then uses a variant of Cuckoo hashing [51] 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, dbDedup only consumes 6 B per index entry.

For each feature in the target record'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 dbDedup retrieves the dedup metadata using the pointer stored in the matched bucket. If the record's dedup metadata contains the same feature in its sketch, it is added to the list of similar records. 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 records to the list of similar records 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 records 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 record will require first removing one of the other four matches from the index. We found that evicting the least-recently-used (LRU) record 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.

dbDedup uses a small dedup metadata cache to reduce the number of reads to the on-disk dedup metadata container [32, 59]. The container is divided into contiguous 64 KB pages, each containing 512 dedup metadata entries. Upon checksum

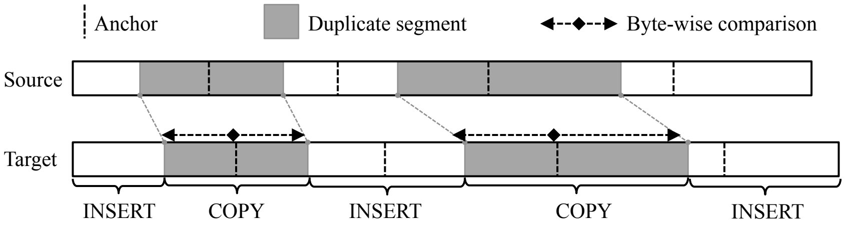


Figure 5.3: Illustration of delta compression in dbDedup.

matches, dbDedup 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 dbDedup fast and memory-efficient. We show in Chapter 6 that the indexing overhead is small and bounded in terms of CPU and memory usage, in contrast to traditional deduplication.

5.3 Delta Compression

To ensure lightweight dedup, it is important to make dbDedup's delta compression fast and efficient. The delta compression algorithm used in dbDedup is adapted from xDelta [46], a classic copy/insert encoding algorithm using a string matching technique to locate matching offsets in the source and target byte streams. The original xDelta algorithm mainly works in two steps. In the first step, xDelta divides the source stream into fixed-size (by default, 16-byte) blocks. It then calculates an Alder32 [33] checksum (the same fingerprint function used in gzip) for each byte block and builds a temporary in-memory index mapping the checksums to their corresponding offsets in the source. In the second step, xDelta scans the target object byte by byte from the beginning, using a sliding window of the same size as the byte blocks. For each target offset, it calculates a Alder32 checksum of the bytes in the sliding window and consults the source index populated in the first step. If it finds a match, xDelta extends the search process from the matched offsets, using bidirectional byte-wise comparison to determine the longest common sequence (LCS) between the source and target streams. It then skips the matched region to continue the iterative search. If it does not find a match, it moves the sliding window by one byte and restarts the matching. Along this process, xDelta encodes the matched regions in the target into COPY instructions and the unmatched regions into INSERT instructions.

The delta compression algorithm used in dbDedup, as shown in Algorithm 4 and Fig. 5.3, modifies xDelta based on the observation that a large fraction of computation time is spent in source index building and lookups. To mitigate this overhead, in the first encoding step, dbDedup samples a subset of the offset positions called *anchors*, whose checksums' lower bits match a pre-determined pattern. The interval between anchors indicates the sampling ratio and is controlled by the length of the bit pattern, similar to how variable-sized chunking algorithms determine the average chunk size. In the second step, dbDedup performs index lookups only for the anchors in the target, avoiding the need to consult the source index at every target offset. The anchor interval provides a tunable trade-off between compression ratio and encoding speed, and we evaluate its effects in Chapter 6. Of course, some details are omitted in the pseudo-code given above. For example, contiguous and overlapping COPY instructions are coalesced; short COPY instructions are converted into equivalent INSERT instructions when the encoding overhead exceeds space savings.

As we discussed in Section 4.1, after computing the forward-encoded data using the algorithm above, dbDedup uses delta re-encoding (Algorithm 5) to efficiently generate the backward-encoded source record. Instead of switching the source and target objects and performing delta compression again, dbDedup reuses the COPY instructions generated before and sorts them by their corresponding source offsets. It then fills the unmatched regions in the source with INSERT instructions. While it may result in slightly sub-optimal compression rate (e.g., due to overlapping COPY instructions that are merged), the re-encoding process is extremely fast (at memory speed), since there are no checksum calculations or index operations.

Delta decompression in dbDedup is straightforward (Algorithm 6). It simply iterates over the instructions generated by the compression algorithm and concatenates the matched and unmatched regions to reproduce the original target object.

Algorithm 4 Delta Compress

```
function DeltaCompress($src, tgt$)
    $i \leftarrow 0$                                         ▷ Initialization
    $j \leftarrow 0$
    $pos \leftarrow 0$
    $ws \leftarrow 16$
    $sIndex \leftarrow empty$
    $tInsts \leftarrow empty$
    while $i + ws <= src.length$ do                          ▷ Build index for src anchors
        $hash \leftarrow$ RabinHash($src, i, i + ws$)
        if IsAnchor($hash$) then
            $sIndex[hash] \leftarrow i$
        end if
        $i \leftarrow i + 1$
    end while
    while $j + ws <= tgt.length$ do                          ▷ Scan tgt for longest match
        $hash \leftarrow$ RabinHash($tgt, j, j + ws$)
        if IsAnchor($hash$) and $hash$ in $sIndex$ then
            $(soff, toff, l) \leftarrow$ ByteComp($src, tgt, sIndex[fp], j$)
            if $pos < toff$ then
                $insInst \leftarrow INST(INSERT, pos, toff - pos)$
                $memcpy(insInst.data, tgt, toff - pos)$
                $tInsts.append(insInst)$
            end if
            $cpInst \leftarrow INST(COPY, soff, l)$
            $tInsts.append(cpInst)$
            $pos \leftarrow toff + l$
            $j \leftarrow toff + l$
        else
            $j \leftarrow j + 1$
        end if
    end while
    return $tInsts$
end function
```

Algorithm 5 Delta Re-encode

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

Algorithm 6 Delta Decompress

```
function DELTADECOMPRESS($src, insts$)
    $pos \leftarrow 0$
    $tgt \leftarrow empty$
    for each $inst$ in $insts$ do
        if $inst.type = COPY$ then
            $memcpy(tgt[pos], src[inst.offset], inst.len)$
        else if $inst.type = INSERT$ then
            $memcpy(tgt[pos], inst.data, inst.len)$
        end if
        $pos \leftarrow pos + inst.len$
    end for
    return $tgt$
end function
```

Chapter 6

Evaluation

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

Unless otherwise noted, all experiments use a replicated MongoDB setup with one primary, one secondary, and one client node. Each node has four CPU cores, 8 GB RAM, and 100 GB of local HDD storage. We use MongoDB's WiredTiger [18] storage engine with the full journaling feature turned off to avoid interference.

6.1 Workloads

The four real-world datasets represent a diverse range of database applications: collaborative editing (Wikipedia), email (Enron), and on-line forums (Stack Exchange, Message Boards). Table 6.1 shows some key characteristics of these datasets. The average record size ranges from 1–16 KB, and most changes modify less than 100 B. We sort each dataset by creation timestamp to generate a write trace, and then generate a read trace using public statistics or known access patterns to mimic a real-world workload, as detailed below.

	Wikipeda	Enron	Stack Exchange	Message Boards
Record Size (bytes)	15875	2849	936	308
Change Size (bytes)	77	56	79	42
Change Distance (bytes)	3602	864	83	37
# of Changes per Record	4.3	3.1	5.8	3.9

Table 6.1: Average characteristics of four datasets.

Wikipedia: The full revision history of every article in the Wikipedia English corpus [16] from January 2001 to August 2014. We extracted a 20 GB subset via random sampling based on article IDs. Each revision contains the new version of the article and metadata about the user that made the edits (e.g., username, timestamp, comment). Most duplication comes from incremental revisions to pages. We insert the first 10,000 revisions to populate the initial database. We then issue read and write requests according to a public Wikipedia access trace [70], where the normalized read/write ratio is 99.9 to 0.1. 99.7% of read requests are to the latest version of a wiki page, and the remainder to a specific revision.

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

Stack Exchange: A public data dump from the Stack Exchange network [13] that contains the full history of user posts and associated information such as tags and votes. Most duplication comes from users revising their own posts and from copying answers from other discussion threads. We extracted a 10 GB subset (of

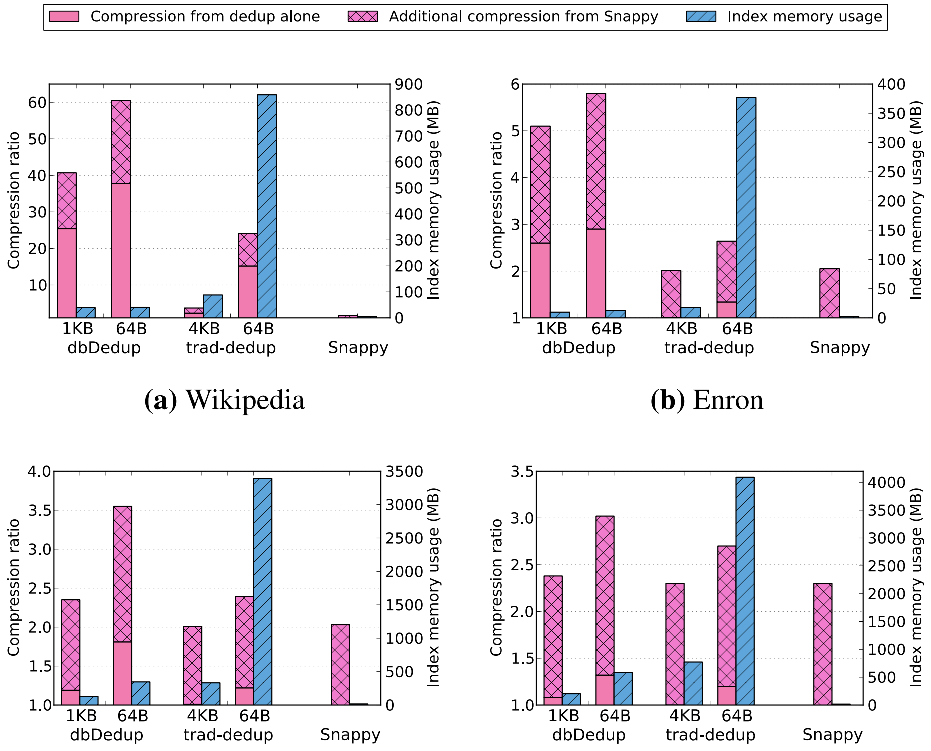
100 GB total) via random sampling. We insert the posts into the DBMS as new records in temporal order. For each post, we read it for the same number of times as its view count. The aggregate read/write ratio is 99.9 to 0.1.

Message Boards: A 10 GB forum dataset containing users' posts crawled from a number of public vBulletin-powered [14] message boards that cover a diverse range of threaded topics, such as sports, cars, and animals. Each post contains the forum name, thread ID, post ID, user ID, and the post body including quotes from other posts. This dataset also contains the view count per thread, which we use to generate synthetic read queries. Duplication mainly originates from users quoting others' comments. To mimic users' behavior in a discussion forum, for each post insertion, we issue a certain number of "thread reads" that request all the previous posts in the containing thread. The number of thread reads per insertion is derived by dividing the total view count of the thread by the number of posts it contains.

6.2 Compression Ratio and Index Memory

We first evaluate dbDedup's compression ratio and index memory usage and compare them to trad-dedup and Snappy [11], MongoDB's default block-level compressor. For each dataset, we load the records into the DBMS as fast as possible and measure the resulting storage sizes, the amount of data transferred over the network, and the index memory usage.

Fig. 6.1 shows the results for five configurations: (1) dbDedup with chunks of 1 KB or 64 bytes, (2) trad-dedup with chunks of 4 KB or 64 bytes, and (3) Snappy. The pink (left) bar shows storage compression ratio, indicating the contribution of dedup alone and compression after dedup. The compression ratio is defined as original data size divided by compressed data size, so a value of one means no compression achieved. The network transfer compression ratio is within 5% of that for storage, in all cases. The blue (right) bar shows index memory usage. The small source record cache (32 MB, used by both dbDedup and trad-dedup) and lossy write-back cache (8 MB, used by dbDedup only) are not shown.



(a) Wikipedia

(b) Enron

(c) Stack Exchange

(d) Message Boards

Figure 6.1: Compression Ratio and Index Memory – The compression ratio and index memory usage for dbDedup (1 KB chunks or 64 byte chunks), trad-dedup (4 KB and 64 byte), and Snappy. The upper portion of each dedup bar represents the added benefit of compressing after dedup.

The benefits are largest for Wikipedia (Fig. 6.1a). With a chunk size of 1 KB, dbDedup reduces data storage by 26× (41× combined with Snappy) using 36 MB index memory. Decreasing the chunk size to 64 B increases compression ratio to 37× (61×) using only 45 MB index memory. Decreasing chunk size for dbDedup does not increase index memory usage much, because dbDedup indexes at most K entries per record, regardless of chunk size. In contrast, while trad-dedup's compression ratio increases from 2.3× (3.7×) to 15× (24×) when using a chunk size of 64 B instead of 4 KB, its index memory grows from 80 MB to 780 MB, making it impractical for operational DBMSs. This is because trad-dedup indexes every unique chunk hash, leading to almost linear increase of index overhead as chunk size decreases, and also because it must use much larger index keys (20-byte SHA-1 hash vs. 2-byte checksum) since collisions would result in data corruption. Consuming 40% less index memory, dbDedup with 64 B chunk size achieves a compression ratio 16× higher than trad-dedup with its typical 4 KB chunk size. Snappy compresses the dataset by only 1.6×, because it can not eliminate the duplication caused by application-level versioning, but requires no index memory. It provides the same 1.6× compression when applied to the deduped data.

For the other datasets, the absolute benefits are smaller, but the primary observations are similar: dbDedup provides higher compression ratio with lower memory usage than trad-dedup, and Snappy's compression benefits (1.6–2.3×) complement deduplication. For the Enron dataset (Fig. 6.1b), dbDedup reduces storage by 3.0× (5.8×), which is consistent with results we obtained from experiments with data from a cloud deployment of Microsoft Exchange servers containing PBs of real user email data.[1] The two forum datasets (Figs. 6.1c and 6.1d) do not exhibit as much duplication as the Wikipedia or email datasets, because users do not quote or edit comments as frequently as Wikipedia revisions or email forwards/replies. Even so, we still observe that dbDedup reduces storage by 1.3–1.8× (3–3.5×). Because we were only able to crawl the latest posts in the Message Boards dataset, dbDedup's compression ratio is conservative, not including the benefits from delta compressing users' revisions to their own posts.[2]

In addition to storage usage, dbDedup simultaneously achieves significant compression on data transmission over the network with forward encoding. Fig. 6.2 shows the network-level compression as a normalized result to that on storage usage (1.0 on the Y-axis for each dataset). dbDedup achieves slightly lower compression on database storage than on data transferred over the network, mainly due to overlapped encodings (Section 4.1) and delta evictions from the write-back cache. Nevertheless, the difference is below 5% for all datasets, because overlapped encodings

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

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

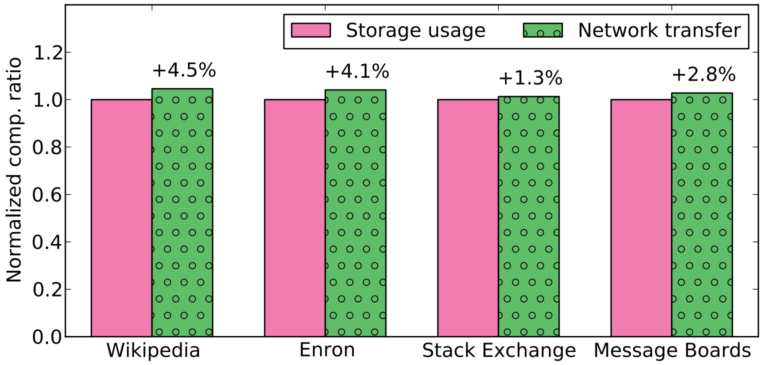


Figure 6.2: Storage and Network Bandwidth Savings – Relative compression ratios achieved by dbDedup (with 64-byte chunk size) for local storage and network transfer, for each of the datasets, normalized to the absolute storage compression ratios shown in Fig. 6.1 (for dbDedup with 64-byte chunks).

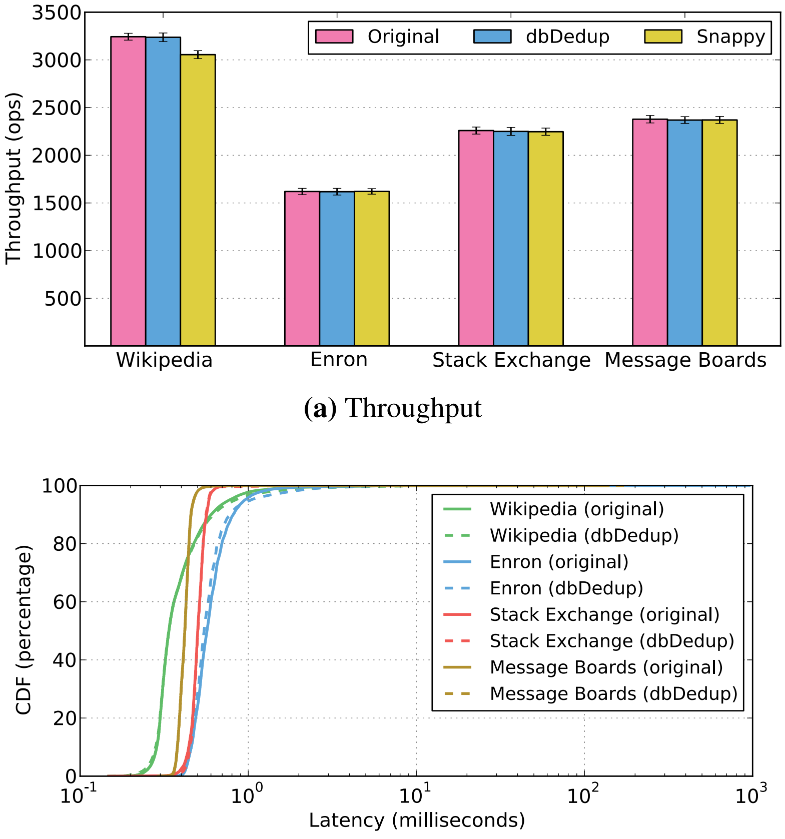
are uncommon and because the lossy write-back cache uses prioritized eviction.

6.3 Runtime Performance Impact

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

Throughput: Fig. 6.3a shows throughput for the four workloads. We see that dbDedup imposes negligible overhead on throughput. Snappy also degrades performance slightly for three of the workloads, since it is a fast and lightweight inline compressor. The exception is Wikipedia, for which using Snappy causes 5% throughput reduction, because some large Wikipedia records cannot fit in a single WiredTiger page and require extra I/Os.

Latency: Fig. 6.3b shows the CDF of client latency. For clarity, we only show the results for MongoDB with and without dbDedup enabled. Again, we observe that dbDedup has almost no effect on performance. The latency distribution curves



(a) Throughput

(b) Latency

Figure 6.3: Performance Impact – Runtime measurements of MongoDB's throughput and latency for the different workloads and configurations.

with dbDedup enabled closely track those for no compression/dedup. The difference in the 99.9%-tile latency is less than 1% for all workloads.

6.4 Dedup Time Breakdown

Fig. 6.4 shows the time required to load the Wikipedia dataset, using stacked bars to show the contribution of each step described in Chapter 3. The three bars show the benefits of adding each of dbDedup's two most significant speed optimizations:

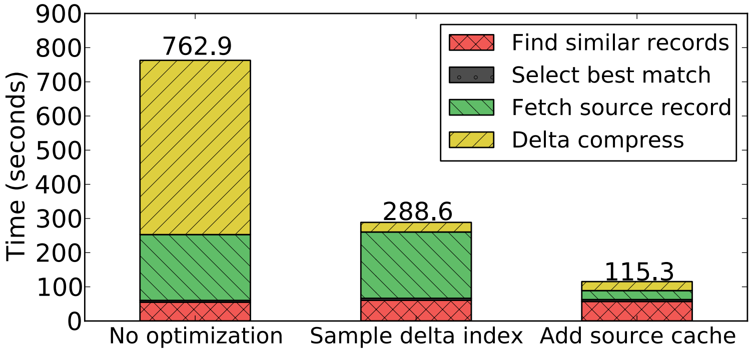


Figure 6.4: Deduplication Time Breakdown – Time breakdown of deduplication steps as individual refinements are applied.

sampling source index in delta computation and adding a source record cache. The default configuration uses both of the optimizations. With no optimizations, dbDedup spends most of the time fetching source records 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 record. dbDedup 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 records from the database the biggest contributor. By using a small source record cache of 32 MB, dbDedup reduces the source fetching time by $\sim$87%, which corresponds to the hit rate observed in Section 6.5.

6.4.1 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. dbDedup 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

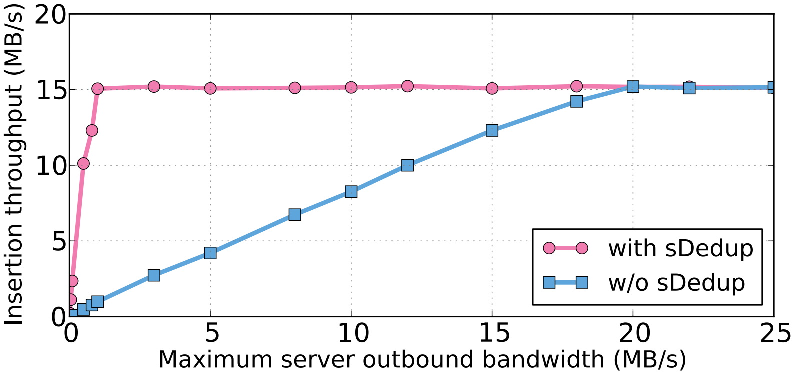


Figure 6.5: Insertion Throughput under Limited Bandwidth. – A evaluation of MongoDB's insertion throughput with and without dbDedup for various network bandwidth configurations.

server. The experiments load the Wikipedia snapshot into the DBMS as fast as possible and enforce replica synchronization every 1000 record insertions.

Fig. 6.5 shows the DBMS's insertion rate as a function of available network bandwidth. Without dbDedup, the required replication bandwidth is equal to the raw insertion rate, resulting in significant throttling when bandwidth is limited. With dbDedup, 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.

6.5 Effects of Caching

As described in Section 4.2, dbDedup uses two small caches to minimize I/O overheads involved in reading and updating source records: a source record cache (32 MB) and lossy write-back cache (8 MB). We now evaluate the effectiveness of these caches.

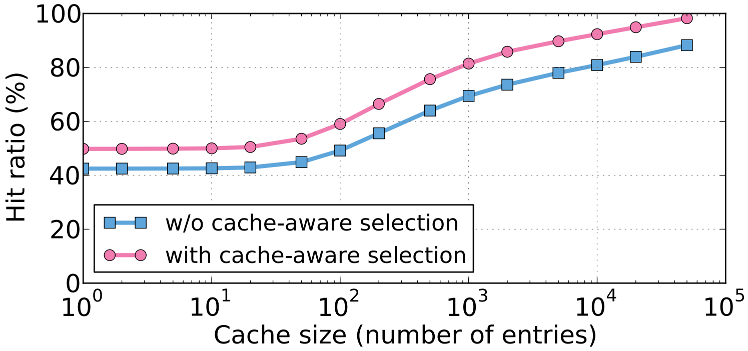


Figure 6.6: Source Record Cache Size – The efficacy of the source record cache and the cache-aware selection optimization.

6.5.1 Source Record Cache

dbDedup's source record cache reduces the number of database queries issued to fetch source records. To evaluate the efficacy of this cache, as a function of its size, We replay the Wikipedia workload starting with a cold cache, and report the steady-state hit rates with and without dbDedup's cache-aware selection technique (see Section 3.3).

Fig. 6.6 shows the hit rate of the record cache as a function of the cache size. Even without cache-aware selection, the source record cache is effective in removing many database queries due to temporal locality in the record 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 record 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.

Fig. 6.7 shows the effect of the source record cache (with a fixed cache size of 32 MB) on compression ratio (left Y-axis) and percent of source record retrievals requiring a DBMS read (cache miss ratio; right Y-axis), with a range of reward score values for the Wikipedia workload. Recall that dbDedup uses cache-aware selection

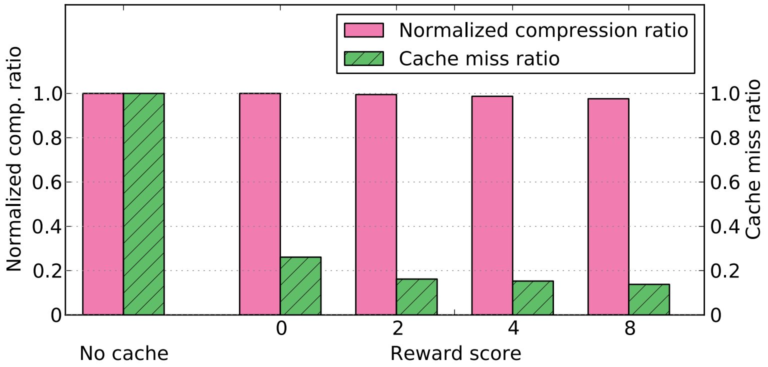


Figure 6.7: Reward Score – An evaluation of the normalized compression ratio and cache miss ratio as a function of the reward score for records residing in the source record cache.

of candidate similar records, assigning a reward score to candidates that are present in the cache (see Section 3.3).

When no cache is used (the left-most bars), every retrieval of a source record incurs a read query. Even without cache-aware selection (0 reward score), the small source record cache eliminates 75% of these queries. With a reward score of two (default), the cache-aware selection technique further cuts the miss ratio by 40% (to 16%), without reducing the compression ratio noticeably. Further increases to the reward score marginally reduce the cache miss ratio while reducing the compression ratio slightly, because less similar candidates are more likely to be selected as the source records.

6.5.2 Lossy Write-back Cache

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

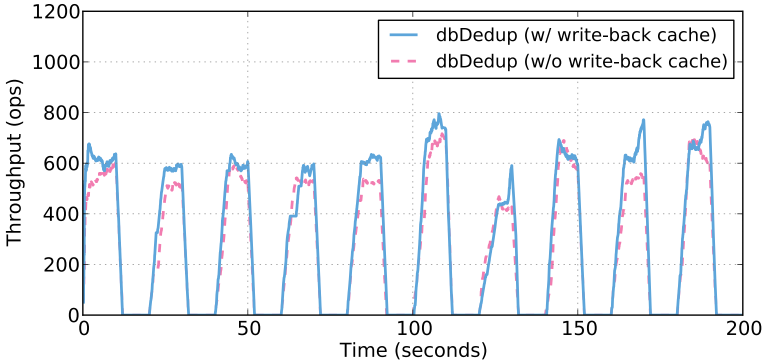


Figure 6.8: Write-back Cache – Runtime throughput of the DBMS with and without the write-back cache. Using the cache avoids DBMS slowdown during workload bursts.

such problems.

To emulate a bursty workload with I/O intensive and idle periods, we insert Wikipedia data at full speed for 10 seconds and sleep for 10 seconds, repeatedly. Fig. 6.8 shows MongoDB's insertion throughput over time, with and without the write-back cache. Without the cache, DBMS throughput visibly decreases during busy periods because of the extra database writes. In contrast, using the write-back cache avoids DBMS slowdown during workload bursts, as shown by the difference between the two lines at various points of time (e.g., at seconds 0, 130, 170, and 190).

6.6 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 records are inserted. To evaluate dbDedup'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

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

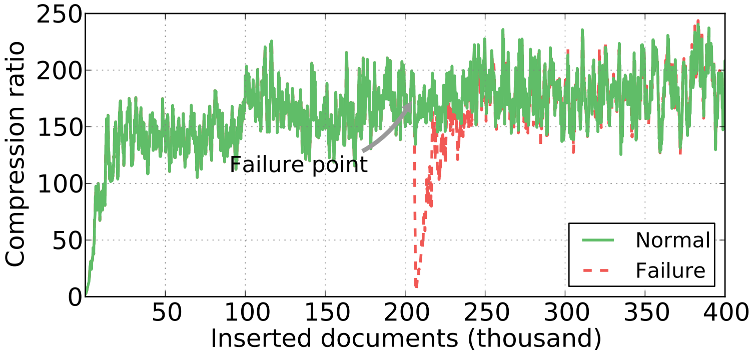


Figure 6.9: Failure Recovery – Measuring how quickly dbDedup recovers after the primary fails.

write workload. We load the dataset into a primary with two secondaries and stop (fail) the primary after 200k insertions.

Fig. 6.9 shows the compression ratios achieved by dbDedup in the normal and failure cases with a moving average of 2000 inserted records. The compression ratio decreases significantly at the failure point, because the records 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 record insertions). This is because most updates are to recent records, so that the effect of missing older records in the index fades rapidly.

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

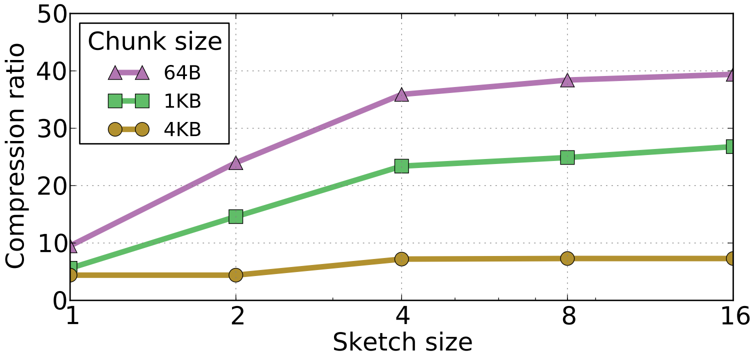


Figure 6.10: Sketch Size – The impact of the sketch size on the compression ratio for the Wikipedia dataset.

6.7 Tuning Parameters

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

6.7.1 Sketch Size

As described in Section 3.1, a sketch consists of the first-K features. Fig. 6.10 shows the compression ratio achieved by dbDedup as a function of the sketch size (K). For the smaller chunk sizes (≤ 1 KB) that provide the best compression ratios, K should be 4–8 to identify the best source records. $K > 8$ 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 record, and increasing the sketch size only helps slightly.

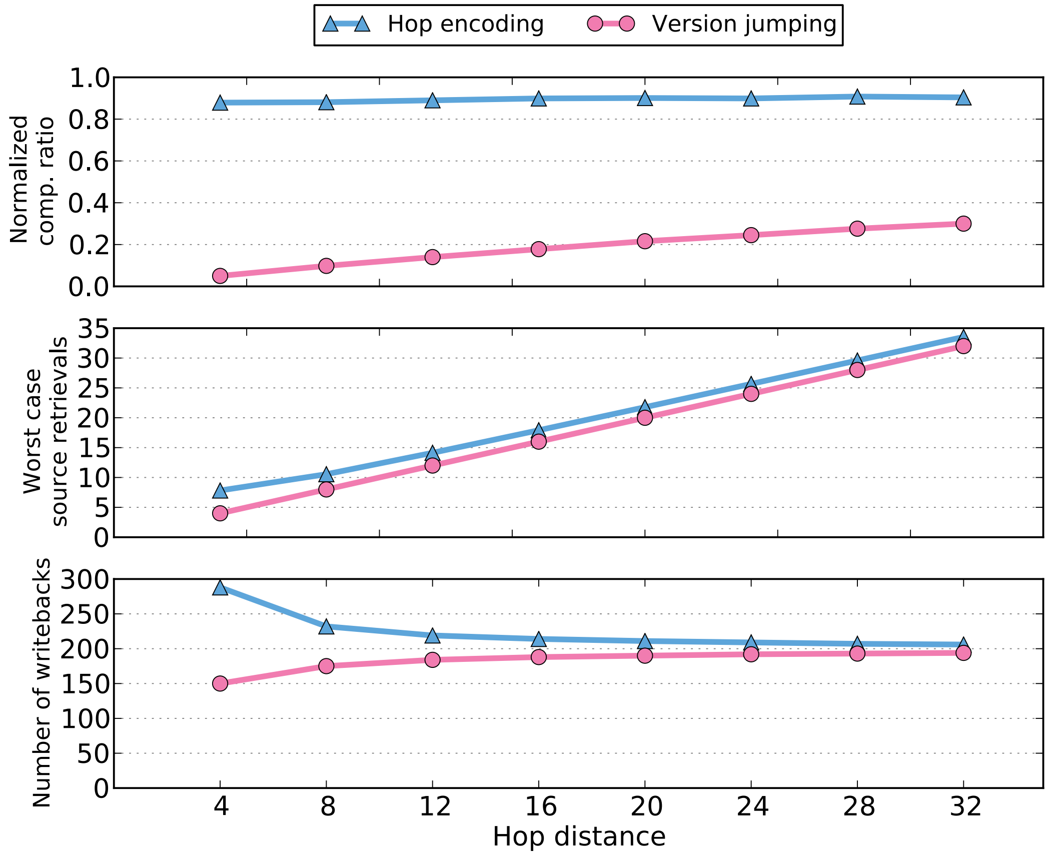


Figure 6.11: Hop Encoding vs. Version Jumping – For the Wikipedia workload and moderate hop distances, hop encoding provides much higher compression ratios with small increases in worst-case source retrievals and number of write-backs.

6.7.2 Hop Distance

dbDedup uses hop encoding to reduce the worst-case retrieval times while maintaining compression benefits. To evaluate its efficacy, we also implemented version jumping in MongoDB and compared the two encoding schemes.

Fig. 6.11 shows the results for three metrics as a function of hop distance: compression ratio (normalized to standard backward encoding), worst-case number of source retrievals (for an encoding chain length of 200), and number of write-backs. Version jumping results in significantly (60–90%) lower compression ratios, be-

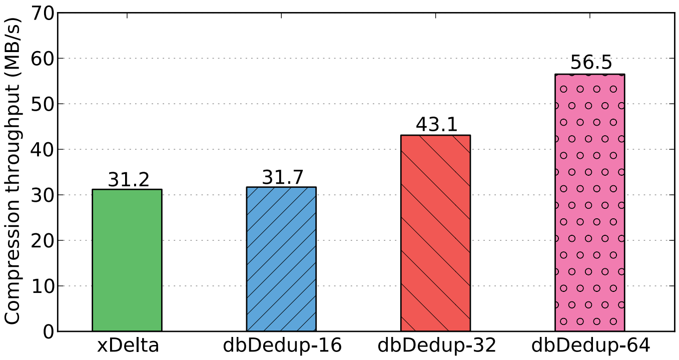


Figure 6.12: Anchor Interval – The impact of the anchor interval on the delta compression throughput and compression ratio for the Wikipedia dataset.

cause all reference versions are stored unencoded. Its compression ratio improves as the hop distance increases, because fewer records are stored in unencoded form. In contrast, because hop bases are stored as deltas, hop encoding provides compression ratios within 10% of full backward encoding. For hop encoding, the compression ratio remains relatively steady as hop distance increases, due to having fewer but less similar hop bases.

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

6.7.3 Anchor Interval

dbDedup outperforms the xDelta algorithm by reducing the computation overhead on source index insertion and lookups. It introduces a tunable anchor interval that

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

Table 6.2: Compression ratio with sharding – dbDedup provides consistent compression benefits in sharded environments.

controls the sampling rate of the offset points in the source byte stream.

Fig. 6.12 shows the compression ratio (left Y-axis) and throughput (right Y-axis) for various dbDedup anchor interval values, as well as for xDelta, for the Wikipedia workload. With an anchor interval of 16 (default window size in xDelta), dbDedup performs almost the same as xDelta. As anchor interval increases, dbDedup's delta compressing speed improves, because it reduces the number of source offset index insertions and lookups. The compression ratio does not significantly decrease, because dbDedup performs byte-level comparison bidirectionally from the matched points. With an anchor interval of 64, dbDedup outperforms xDelta by 80% in terms of compression throughput, while incurring only 7% loss in compression ratio. Increasing the anchor interval to 128 further improves the throughput by 10% but results in 15% loss in compression ratio. We use 64 as the default value, providing a balance between compression ratio and throughput.

6.8 Sharding

This section evaluates the performance of dbDedup 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 dbDedup. For experiments with sharding, we use the 20 GB Wikipedia dataset and shard records on article ID (like the Wikipedia service). To accommodate MongoDB's capacity balancing migrations, we modified dbDedup to remove the assumption that the source record can always be found on the primary node. When it cannot, because it was migrated, dbDedup simply deletes it from the dedup index and treats the target record as unique.

Table 6.2 shows the compression ratio as a function of the number of shards. The compression ratio is not significantly affected, because Wikipedia records with the same article ID go to the same server and most duplication comes from incremental updates to the same article. This indicates that dbDedup is robust and still works in sharded deployments.

Chapter 7

Conclusion and Future Directions

This chapter concludes the dissertation and discusses possible directions for future research.

7.1 Conclusion

This dissertation describes a systematic approach termed similarity-based deduplication to reducing the storage usage and network transfer for operational DBMSs. Our appraoch achieves higher compression ratios than block-level compression and chunk-based deduplication while being memory efficient by (1) partially indexing representative chunk hashes for new records; (2) performing byte-level delta compression on similar records; and (3) maximizing cache efficiency and utilization with source-aware cache replacement. Our approach imposes negaligible performance overhead on the runtime performance of the DBMS by performing the main dedup work off the critical path and using a single-pass encoding scheme for both storage and network layers. In addition, it uses a combination of several techniques in order to mitigate the deduplication overhead, including two-way encoding, hop encoding, cache-aware selection, lossy write-back caching, automatic dedup governing, and adaptive size-based filtering. Experimental results with four real-world workloads show that the proposed approach is able to achieve up to 37× reduction

(61× when combined with block-level compression) in storage size and replication traffic while imposing negligible overhead on DBMS performance.

This disseration makes the following contributions:

- The first dedup system for online DBMSs that achives data reduction in both database storage and network bandwidth for replication services, as well as the first database dedup system that employs similarity-based dedup.
- The characterization of real-world database workloads, based on which we illustrate the advantange of the proposed approach over the state-of-the-art data reduction schemes in face of these workloads.
- The presentation of a general-purpose end-to-end workflow of similarity-based dedup, and the proposal of various novel techniques in encoding, caching and similarity selection that are important to achieve high dedup efficiency.
- The implementation of dbDedup, a lightweight scheme for online database systems that uses similarity-based dedup to compress individual records stored on disk and sent to remote replicas over network, and the full integration of dbDedup into MongoDB's storage and replication components.
- The evaluation of dbDedup in terms of compression ratio, memory usage and runtime performance overhead (throughput and latency) using four real-world datasets, which shows that similarity-based dedup is a viable and efficient approach for online DBMSs.

7.2 Future Directions

This section describes several future research directions for applying or extending the proposed similarity-based dedup approach.

7.2.1 Client-side Deduplication

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 records could be deduplicated against the original records before they are sent to the servers. The deduplication approach in this senario would be similar to deduplicating replication streams between database servers. The only difference is that clients and servers are not replicas—clients only store and operate on a small subset of data in the servers. Because the servers store the original versions being used by clients as source records, the servers will be able to interpret and re-construct the deduplicated data using locally stored records. For applications in which most write requests are incremental updates on existing data, the potential for client bandwidth reduction is significant.

7.2.2 Similarity-informed Sharding

Currently, we assume that in a sharded setup, similar records would be assigned to the same database server so that the compression ratio with sharding would not significantly decrease as compared to that in a non-sharded setup. For the Wikipedia example, revisions of a particular article are distributed to the same servers because they share the same article ID. While we believe that this assumption holds for many applications, when it does not (e.g., when the Wikipedia articles are randomly sharded by system-wide unique IDs), the compression ratio may drop visibly as the number of shards increases, since there is no more guarranttee that similar records are sharded to the same server. In this scenario, a new sharding scheme is needed that preserves the dedup quality by taking into consideration of similarity properties of data to be sharded. It is interesting to explore the tradeoff between compression ratio, resource usage, query latency and load balancing of the new sharding method.

7.2.3 Field Name Compression in Document Databases

While we propose similarity-based dedup as a general-purpose approach that treats record data as raw byte streams for all DBMSs, we believe that greater savings may be realized using specialized schemes that exploits a priori knowledge of the input data format. An example of using simlarity-based dedup in combination with such knowledge is document databases, where field names are stored in each document to support flexible schema changes. This creates an extra source of duplication, in addition to those discussed in Section 2.2.2, that can be eliminated by slightly modifying the current dedup workflow of dbDedup. Since schema change is not a frequent operation, most documents belonging to the same collection would have a significant portion of duplicate field names. It is expected that a specialized dedup scheme that takes specific document layouts into consideration would provide higher compression ratio than the general approach.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

[12] Data Compression: Why Do we need it? https://blogs.msdn.microsoft.com/sqlserverstorageengine/2007/09/29/data-compression-why-do-we-need-it/.

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

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

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

[16] Wikimedia Downloads. https://dumps.wikimedia.org, .

[17] Wikipedia. https://www.wikipedia.org/, .

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

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

[20] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

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

[22] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. The design of a similarity based deduplication system. In *SYSTOR*, page 6, 2009.

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

[24] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS*, pages 1–9, 2009.

[25] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

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

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

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

[29] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX ATC*, 2009.

[30] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in xml documents. In *ICDE*, pages 41–52, 2002.

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

[32] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX Annual Technical Conference*, 2010.

[33] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[34] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, and Jerzy Szczepkowski. Hydrastor: A scalable secondary storage. In *FAST*, 2009.

[35] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, , and Michal Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST*, 2009.

[36] Ahmed El-Shimi, Ran Kalach, Ankit Kumar Adi, Oltean Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *USENIX Annual Technical Conference*, 2012.

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

[38] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. 2005.

[39] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[40] Balakrishna Iyer and David Wilhite. Data compression support in databases. 1994.

[41] Navendu Jain, Michael Dahlin, and Renu Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *FAST*, 2005.

[42] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14, 2013.

[43] Purushottam Kulkarni, Fred Douglis, Jason D LaVoie, and John M Tracey. Redundancy elimination within large collections of files. In *Usenix Annual Technical Conference*, 2004.

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

[45] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *FAST*, 2009.

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

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

[48] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *FAST*, 2011.

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

[50] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP*, 2001.

[51] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Al-*

gorithms, 51(2):122–144, 2004.

[52] Meikel Poess and Dmitry Potapov. Data compression in oracle. In *VLDB*, pages 937–947, 2003.

[53] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

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

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

[56] Michael O Rabin. *Fingerprinting by random polynomials*.

[57] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *VLDB*, 6(11):1080–1091, 2013.

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

[59] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. In *FAST*, 2012.

[60] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. *USENIX Hot Storage*, 2012.

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

[62] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *FAST*, 2012.

[63] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[64] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization. *Lossless Compression Handbook*, 2002.

[65] 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.

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

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

[68] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. zdelta: An efficient delta compression tool. *Technical Report TR-CIS-2002-02, Polytechnic University*, 2002.

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

[70] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[71] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *FAST*, 2012.

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

[73] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput.

In *USENIX Annual Technical Conference*, 2011.

[74] Lianghong Xu, James Cipar, Elie Krevat, Alexey Tumanov, Nitin Gupta, Michael A Kozuch, and Gregory R Ganger. Springfs: bridging agility and performance in elastic distributed storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 243–255, 2014.

[75] Lawrence L You, Kristal T Pollack, and Darrell DE Long. Deep store: An archival storage system architecture. In *ICDE*, pages 804–815, 2005.

[76] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.

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

[78] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, pages 59–59, 2006.