Probabilistic Deduplication for Cluster-Based Storage Systems

Davide Frey
INRIA
Rennes, France
davide.frey@inria.fr

Anne-Marie Kermarrec
INRIA
Rennes, France
anne-marie.kermarrec@inria.fr

Konstantinos Kloudas
INRIA
Rennes, France
konstantinos.kloudas@inria.fr

ABSTRACT

The need to backup huge quantities of data has led to the development of a number of distributed deduplication techniques that aim to reproduce the operation of centralized, single-node backup systems in a cluster-based environment. At one extreme, stateful solutions rely on indexing mechanisms to maximize deduplication. However the cost of these strategies in terms of computation and memory resources makes them unsuitable for large-scale storage systems. At the other extreme, stateless strategies store data blocks based only on their content, without taking into account previous placement decisions, thus reducing the cost but also the effectiveness of deduplication.

In this work, we propose, PRODUCK, a stateful, yet lightweight cluster-based backup system that provides deduplication rates close to those of a single-node system at a very low computational cost and with minimal memory overhead. In doing so, we provide two main contributions: a lightweight probabilistic node-assignment mechanism and a new bucket-based load-balancing strategy. The former allows PRODUCK to quickly identify the servers that can provide the highest deduplication rates for a given data block. The latter efficiently spreads the load equally among the nodes. Our experiments compare PRODUCK against state-of-the-art alternatives over a publicly available dataset consisting of 16 full *Wikipedia* backups, as well as over a private one consisting of images of the environments available for deployment on the Grid5000 experimental platform. Our results show that, on average, PRODUCK provides (i) up to 18% better deduplication compared to a stateless *minhash*-based technique, and (ii) an 18-fold reduction in computational cost with respect to a stateful Bloom-filter-based solution.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; H.3.4 [**Information Storage And Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Cloud Computing, Distributed Storage Systems, Deduplication, Set Intersection

1. INTRODUCTION

Cloud providers, social networks, data-management companies and on-line backup services are witnessing a tremendous increase in the amount of data they receive every day. This phenomenon implies huge storage needs that increase exponentially [10] as more users join and already existing ones become more engaged to the offered services. For such companies, the data they store constitutes part of their business and potential data disruption could lead to decreased revenues due to

SOCC'12, October 14-17, 2012, San Jose, CA USA

loss of credibility or service deterioration. To minimize this danger and also for legal requirements [6], data centers need to periodically backup full copies of their data for periods of up to several years and have them available for retrieval upon request.

Until recently, most backup software combined files into huge tarballs and stored them on tape in an effort to minimize costs [6]. However, the decrease in the cost of magnetic disks combined with *deduplication* techniques that can make more efficient use of storage space have made disk-based backup solutions more popular. Based on the observation that consecutive backups of the same files are highly likely to have parts in common, deduplication identifies identical chunks of data in a large backup and replaces them with references to a previously stored copy of the chunk [15], thereby reducing the required storage space. Using deduplication, single-node commercial backup systems are now capable of storing petabytes of data to disk [11]. Yet, the backup needs of modern data centers are already surpassing this limit [10], and the amount of data to backup is bound to increase even further as more and more companies outsource much of their infrastructure to the cloud.

An appealing approach to address these increasing requirements is the design of cluster-based backup platforms. Integrating deduplication into cluster-based solutions, however, is a challenging task [6, 7, 4]. In a single-node system, the main difficulty is to maximize deduplication while maintaining high throughput by identifying the best chunk size (*i.e.* the granularity at which the deduplication can be identified) and managing to keep the index of already stored chunks in memory. A cluster, however, introduces a new challenge: assigning each chunk to a cluster node while *(i)* maximizing deduplication and *(ii)* balancing the storage load on the available nodes. The tension between these two conflicting requirements has led researchers to explore two main approaches to assign data chunks to nodes. So-called *stateless* solutions [6, 4] assign data to nodes based only on the observation of the contents of the chunk of data being stored. This, in many cases, provides a somewhat natural form of load balancing, due to its randomized nature, but it yields suboptimal deduplication performance as assignment decisions do not take into account the locations of already stored chunks. At the other extreme, *stateful* approaches maintain information about currently stored data in order to assign identical data blocks to the same node. This yields much better deduplication performance, but with two main drawbacks. First, stateful techniques require more complex load-balancing strategies to avoid storing everything on the same node. Second, existing solutions require significantly more computing and memory resources to index all the stored information. This leads deployed systems, such as [6] to operate according to a *stateless* model, at least until a *practical* stateful system becomes available.

In this paper, we attempt to satisfy this need by proposing PRODUCK, a lightweight cluster-based backup system that aims to make stateful deduplication usable in practice. PRODUCK achieves deduplication rates that are close to those of a single-node system by combining state-of-the-art techniques with novel contributions. First, similar to [6], PRODUCK addresses the tradeoff between small and large chunk sizes with a two-level chunking algorithm. It routes and stores data based on relatively large superchunks, composed of a number of smaller chunks that constitute the information units for deduplication. Second, it incorporates two novel contributions that enable it (i) to achieve stateful superchunk assignments with minimal CPU and memory requirements, and (ii) to balance the load on cluster nodes without hampering deduplication performance.

The first of these contributions is a probabilistic similarity metric that makes it possible to identify the cluster node that currently stores the highest number of chunks out of those that appear in a given superchunk. This metric is based on a probabilistic technique for computing set intersection that originates from the field of information retrieval [16], and which, to our knowledge, has never been used in the context of storage systems. The second novel contribution of PRODUCK is a deduplication-friendly bucket-based load-balancing strategy. Specifically, PRODUCK uses fixed-size buckets to measure the deviation of a node's disk usage from the average. This facilitates the aggregation of similar superchunks, i.e. superchunks with many common chunks, on the same nodes even in the initial phases of a backup process, ultimately yielding better deduplication.

PRODUCK may be deployed as a stand-alone system, but it can also be used as a middleware platform to integrate existing high-throughput single-node deduplication solutions. This allows users to easily integrate new techniques, leverage existing research results and encourages PRODUCK's adoption in a wide range of backup services. In this paper, we focus on the evaluation of PRODUCK as a stand-alone system. We thus leave a study of its integration with existing platforms as future work.

We carried out our evaluation over two datasets. The first consists of more than 520GB of consecutive snapshots of the English version of *Wikipedia* and can be freely downloaded from [1]. The second comprises more than 140GB of binary data consisting of images of the environments available for deployment on the Grid5000 experimental platform hosted by INRIA. The results are very promising. PRODUCK improves deduplication by up to 16% and up to 21% for the two datasets when compared to a *stateless* strategy presented by the authors of [6] and used in their commercial product. At the same time, it provides, on average, an 18-fold reduction in the time complexity of chunk assignment when compared to a *stateful* solution presented in the same work. In addition, PRODUCK achieves these results by only keeping an index of 64KB per storage node

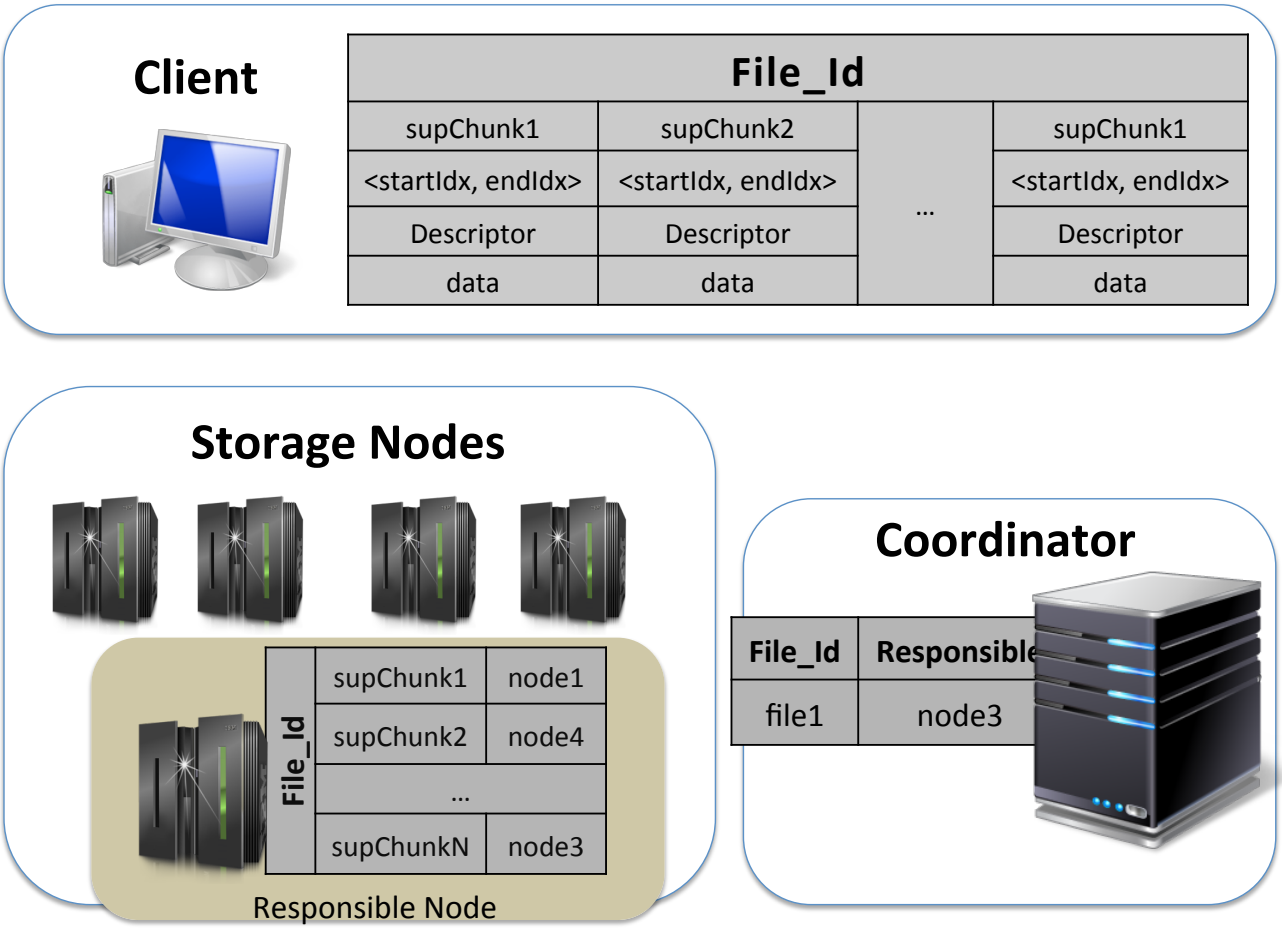


Figure 1: System Architecture

in memory and while keeping the system load balanced. The load of the most loaded node does not differ by more than 2% from the average load in the system. Diving into the specifics of our system, we also present and evaluate our solutions to multiple issues we faced when designing our system as these can provide useful insights to other researchers in the field.

The rest of the paper is organized as follows. Section 2 describes our system architecture, the chunking algorithm, and the way PRODUCK assigns superchunks to nodes. Section 3 presents our message-exchange protocol. Section 4 presents our experimental setup: our datasets, the architectures we compare against and the metrics we consider. Section 5 presents and discusses our experimental results. Section 6 introduces related work, while Section 7 concludes this paper and outlines our directions for future work.

2. PRODUCK PRINCIPLES

We start by analyzing the main features in the design of PRODUCK. We present its architecture, we then describe its chunking mechanism, and finally, we introduce PRODUCK's two main contributions: its chunk-assignment protocol, and its load-balancing strategy.

2.1 Architecture

PRODUCK's architecture, depicted in Figure 1, consists of three main entities: a CLIENT application, a COORDINATOR node, and a set of STORAGENODES.

The CLIENT constitutes PRODUCK's frontend, and thus provides the entry point through which PRODUCK's users can interact with the backup system. The CLIENT offers facilities to manage stored data, such as listing the contents of a backup.

In the following, however, we concentrate on its two fundamental tasks: storing and retrieving a file. When storing a file, the Client is responsible for translating it into a set of chunks that it will then deliver to the StorageNodes. When retrieving a file, the Client obtains the chunks from the StorageNodes and is responsible for recomposing them into the original file. We describe the details of the chunking process in Section 2.2.

The Coordinator node is responsible for managing the requests of Clients, determining which StorageNodes should store each chunk of a given file, and putting Clients in contact with the StorageNodes. To achieve this, the Coordinator implements our novel chunk assignment and load balancing strategies, as described, respectively in Section 2.3 and 2.4. In the current version of Produck, there is only one Coordinator for the entire system. Federating multiple Coordinator nodes for higher throughput or fault-tolerance can be achieved through standard techniques and is outside the scope of this paper.

StorageNodes perform a key role in our backup system by providing storage space according to the assignment decisions made by the Coordinator. As shown in Figure 1, however, StorageNodes also carry out an additional task, that of acting as *responsible nodes*, i.e. directory services for Clients that need to read the content of a given file from multiple StorageNodes. The Coordinator randomly selects a *responsible node* for each file among the StorageNodes. This offloads the Coordinator from the management of read requests and of file metadata. We describe the details of the interactions between Clients, the Coordinator, and StorageNodes in Section 3.

2.2 Chunking

As outlined in Section 1, a deduplication system operates by splitting files into chunks and identifying identical chunks within one and across multiple backup operations. A key performance tradeoff is associated with the size of a chunk. As in a single-node system, smaller chunks make duplicate detection more efficient as they search at a finer granularity, but result in a greater overhead for storing the associated indexing information. In addition, smaller chunks also limit the achievable throughput. Disks are more efficient when accessing continuous data, while the use of very small chunks results in less efficient random-access patterns.

Similar to [6], Produck addresses this tradeoff by using a two-level chunking algorithm. Specifically, it uses relatively small chunks as deduplication units. However, it groups contiguous chunks into large superchunks when storing data onto storage nodes. This makes it possible to optimize data transfer by dealing with large amounts of contiguous data, while retaining the deduplication advantage associated with small data chunks. The chunking algorithm is executed at the Client as soon as the user requests the backup of a file. In the following, we describe its details by explaining how it creates chunks and superchunks.

Content-based chunking.

The most intuitive way to split a file into chunks is to use fixed-size chunks starting from the beginning of the file. However, this approach suffers from a fundamental drawback when new data is added at the beginning or in the middle of a file. The contents of the chunks that follow the location where the data is written are shifted and the chunks are considered new even if they are essentially the same. This is a major issue in a deduplicated backup system that aims to identify duplicate chunks across multiple versions of the same files or across different files.

To address this issue, Produck uses, like most deduplication systems, content-based chunking (CBC). Instead of using fixed chunk sizes, Produck determines chunk boundaries on the basis of the data contained in the chunks. Specifically, it defines a minimum and a maximum chunk size, $l_{\min}$ and $l_{\max}$, and defines chunk boundaries as sequences of w bytes that (i) cause the chunk to be within these limits, and (ii) satisfy a condition on their hash values. Let W be a window of w bytes, let f be its fingerprint consisting of its Rabin hash value, and let D and r be two integer values. Then, W is selected as a chunk boundary if its fingerprint satisfies Equation (1).

$$f \mod D = r \tag{1}$$

If a chunk has no window that satisfies the equation while maintaining the size within the limits, then it is truncated at the maximum size. After a chunk boundary is declared, a SHA1 hash value of the whole chunk is computed and used as the chunk's fingerprint, as we will see below. The values of $l_{\min}$, and $l_{\max}$, in combination with D, r, and w control the average chunk size, which in our case is 1KB.

From a practical perspective, the chunk-creation process is run by the Produck Client. When it needs to store a new file, the Client iterates through its contents as follows. First, it initializes a new chunk and inserts the first $l_{\min}$ bytes of the file in it. Then it computes the fingerprint f of the last w bytes it added. If the fingerprint satisfies Equation (1), then it closes the chunk and it iterates the process by initializing a new chunk with the next $l_{\min}$ bytes from the file. If, instead, the equation is not satisfied, i.e. $f \mod D \neq r$, the Client continues filling the current chunk by adding bytes one at a

time, recomputing the fingerprint of its last w bytes after each byte, and reevaluating the equation. It continues doing so until either the equation is satisfied, or the chunk has reached the maximum prescribed size, $l_{\max}$, at which point it stores the chunk and repeats the process by creating a new one.

Using this content-based process, the CLIENT can create chunks that are independent of the content that may be added or removed earlier in the file. This makes it easier for PRODUCK to identify commonalities between files, as localized changes affect a limited number of chunks.

Super-chunk formation.

The CLIENT is also responsible for grouping chunks into superchunks. Similar to chunks, superchunks are built using a content-based approach. However, instead of verifying the fingerprint of a window of w bytes, the CLIENT identifies the boundary of a superchunk by checking the fingerprint, computed as the SHA1 hash, of the last of its chunks. Specifically, when the CLIENT closes a new chunk, c, as part of the process described above, it first adds c to the current superchunk. Then it computes c's fingerprint, f, by hashing c's content. If f satisfies Equation (1), the CLIENT closes the superchunk and starts a new one, otherwise it continues adding chunks to the current one, until the maximum allowed size is reached or a content-based boundary is declared. In our case, we use an average superchunk size of 15MB or $15 * 1024$ chunks. We examine the impact of varying the superchunk size in Section 5.2.

2.3 Chunk Assignment

The key operation in a cluster-based storage system is the assignment of data chunks (superchunks in the case of PRODUCK) to STORAGENODES so as to maximize deduplication. In PRODUCK, this assignment is carried out by the COORDINATOR through a novel chunk-assignment protocol that exploits a stateful approach to aggregate, on the same nodes, superchunks that share a large number of chunks.

Our protocol addresses the computational and memory requirements of stateful deduplication by leveraging a key observation. To choose the STORAGENODE offering the highest deduplication rate for a given superchunk, the protocol does not need to know exactly which chunks are stored on each STORAGENODE, but only the overlap between the chunks on each STORAGENODE and those in the superchunk to be stored. Our novel protocol computes this overlap by relying on PCSA [9, 8], a probabilistic method for computing the cardinality of a multiset, i.e., the number of distinct items (chunks in our case) it contains. To the best of our knowledge, PRODUCK is the first application of PCSA in the context of backup systems.

Probabilistic counting.

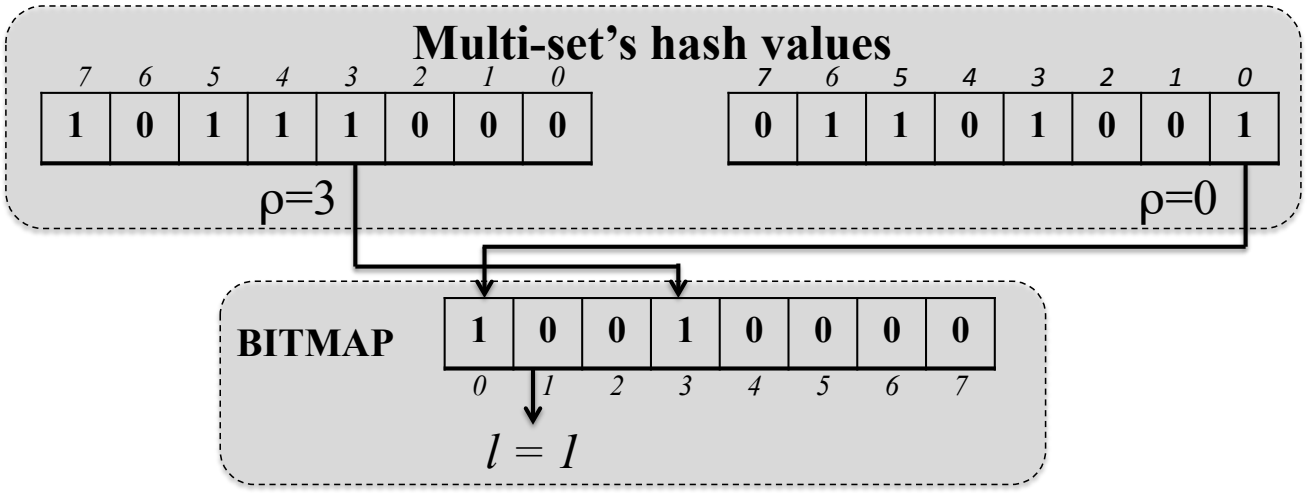


Figure 2: PCSA multi-set cardinality estimation.

Let S be a multiset of chunks, for example a superchunk or a set of superchunks stored by a STORAGENODE. Let $h : S \to [0, 2^L)$ be a hash function that outputs values that are uniformly spread over its target set. Also, let $bit(y, k)$ denote the k-th bit in y's binary representation, with bit 0 being the least significant. PCSA defines a function $\rho : [0, 2^L) \to [0, L)$ identifying the position of the least significant 1-bit in y, with position 0 corresponding to the least significant bit.

$$\rho(y) = \begin{cases} \min\limits_{k \geq 0} bit(y,k) \neq 0 & \text{for } y > 0 \\ L & \text{for } y = 0 \end{cases} \tag{2}$$

Given a multiset S, PCSA associates it with a BITMAP, a vector of L bits initialized as 0's. Then, it iterates through all the elements in the multiset, $d \in S$ and computes $p = \rho(h(d))$, i.e. the position of the least significant 1-bit in the hash value of d, and records this position by setting the corresponding bit in the BITMAP, BITMAP[p], to 1. Since the hash function distributes its values uniformly in $[0, 2^L)$, BITMAP[0] will be set to 1 approximately half of the times, BITMAP[1] will be set 1/4 of the times, BITMAP[2] will be set 1/8 of the times, and so on. This leads to the probability of setting the bit at position k shown in Equation (3).

$$P(p(h(d)) = k) = 2^{-k-1} \tag{3}$$

Because of this, PCSA uses the position, l, of the leftmost zero in the BITMAP vector counting from the left as an estimation of $\log_2(\phi|S|)$, where $\phi = 0.77351$ [9]. This makes it possible to estimate the cardinality of S as $2^l/\phi$. Figure 2 exemplifies the process. The least significant 1-bit for the left item is at position 3 and causes BITMAP[3] to be set. The one for the right item is instead at position 0 and it causes BITMAP[0] to be set. Given these values, the position of the leftmost zero in the BITMAP vector is then $l = 1$, which leads to a cardinality estimation of $2.58 = 2^1/0.77351$.

Clearly the probabilistic nature of PCSA implies the presence of an estimation error. To reduce it, the authors of [9] propose the use of m BITMAP vectors, instead of a single one. Specifically, let $r = (h(d) \mod m)$; the improvement consists in using the value of $h(d)/m$ to update the rth vector as described above. Averaging the positions of the leftmost 0s (counting from the left) from all the m BITMAP vectors provides an estimate of $\log_2(\phi|S|)$ with less than a 2% error when using $m = 8192$. We therefore use $m = 8192$ in our implementation. Even with this improvement, it is important to observe that the estimation error tends to be larger for very small multisets. This fact must be taken into account when selecting the size of PRODUCK's superchunks, as discussed in Section 5.2.

Probabilistic multiset intersection.

Although PCSA was designed to estimate the cardinality of a multiset, the cardinality of the union of two multisets, A and B, can be estimated by simply applying the bitwise OR operator on the two corresponding BITMAP vectors, BITMAP[A] and BITMAP[B].

$$\text{BITMAP}[A \cup B] = \text{BITMAP}[A] \mid \text{BITMAP}[B] \tag{4}$$

Leveraging this observation, the authors of [16] propose to evaluate the cardinality of the intersection of two multisets by expressing it as follows.

$$|A \cap B| = |A| + |B| - |A \cup B| \tag{5}$$

This makes it possible to approximate $|A \cap B|$ simply by estimating the three cardinalities on the right side of Equation (5) using PCSA.

Maintaining chunk information.

In the context of PRODUCK, we have two types of multisets. The first are the superchunks created by CLIENTS as described in Section 2.2. These are multisets because each superchunk can contain repeated chunks of data. The second consists instead of the sets of chunks stored by each STORAGENODE. Again, these are multisets because each STORAGENODE stores multiple superchunks, which in turn may contain multiple identical chunks (although stored once).

The CLIENT is responsible for creating BITMAP vectors for the superchunks it creates. Specifically, when a CLIENT wishes to store a file on the backup system, it first splits it into chunks and superchunks as described in Section 2.2. For each created superchunk, it computes an array of 8192 BITMAP vectors, as described above, and sends it to the COORDINATOR as a request to store the corresponding superchunk. The total size of these vectors is 64KB. The value of 8192 for the number of vectors was chosen as it provides a good tradeoff between the estimation error and the time complexity of PCSA, as we will show in Section 5. In addition, this allows the COORDINATOR to efficiently assign superchunks to nodes by only storing 64KB per storage node, thus leading to minimal space requirements.

The COORDINATOR instead creates BITMAP vectors for the second type of multisets as a result of its assignment decisions. Specifically, the COORDINATOR maintains a storage BITMAP vector, BITMAP^S, for each storage node. Each such vector is initialized to a sequence of 0's and is updated by combining it with the BITMAP vectors of stored superchunks according to Equation (4). In the case of data deletion, the BITMAP vectors describing node content also have to be updated to reflect the new state. In our experiments, recomputing the BITMAP vectors of all the superchunk in the *Wikipedia* dataset (520GB) takes less than 7 minutes. We expect PRODUCK to perform these updates twice a day similar to what is done in [11].

Choosing the best STORAGENODE .

When a CLIENT wishes to store a superchunk, c, on the backup system, it computes a corresponding BITMAP vector and sends it to the COORDINATOR, along with the associated superchunk identifier. The COORDINATOR then uses the above technique to identify the STORAGENODE that is most suited to storing the received superchunk. Once it has determined the STORAGENODE that will store superchunk c, the COORDINATOR informs the CLIENT and updates the STORAGENODE's BITMAP vector, by combining it with the one corresponding to c using Equation (4).

In order to select the best STORAGENODE for each superchunk c, the COORDINATOR estimates, for each STORAGENODE n, the cardinality of the intersection between the chunks stored by n and the content of c. This is easily achieved by applying Equations (4) and (5) to the corresponding BITMAP vectors and yields a measure of the *overlap* between the received superchunk and each STORAGENODE.

In addition to the overlap between a node's content and a superchunk, and to minimize the effect of estimation errors, we leverage data locality properties observed in backup workloads. Specifically, instead of selecting the STORAGENODE with the highest overlap value, the COORDINATOR ranks the top k STORAGENODES according to their overlaps. If the node that stored the previous superchunk in the file is among these k and his difference from the first node is less than 10%, then the COORDINATOR selects this node for the current superchunk, otherwise it selects the top node. We tested several values of k, and $k = 3$ provided a good compromise between locality and overlap. Finally, if no STORAGENODE is found to have an overlap with a given superchunk or all nodes with an overlap are overloaded, the COORDINATOR selects a STORAGENODE at random among the non-overloaded ones.

2.4 Load Balancing

PRODUCK achieves load balancing through a novel bucket-based mechanism that is specifically designed to operate with our similarity-based chunk-assignment strategy. A naïve approach to load balancing could, for example, constraint the selection of the most suitable STORAGENODE to the nodes that have a storage load that does not exceed the average load of the system by more than a fixed percentage, e.g. 5%. This mechanism, which is used for example in [6], however, is too aggressive when used with our chunk assignment strategy as shown in Section 5.3.

Our novel bucket-based load-balancing strategy, instead, operates by splitting the storage space of each node into fixed-size *buckets*. At initialization time, the COORDINATOR grants each STORAGENODE permission to use one of its buckets. When a STORAGENODE fills the latest bucket it was granted, the COORDINATOR grants it a new bucket only if doing so would not exceed the number of buckets allocated to the least loaded node by more than one. The impact of the maximum number of *buckets* that the most loaded node is allowed to differ from the least loaded one is further studied in Section 5. In the same section, we show that this strategy gives better results in terms of deduplication while guaranteeing an equal distribution of storage load among the nodes in the system.

3. PRODUCK OPERATION

To provide a clearer picture of the behavior of PRODUCK, we now detail the operations carried out when storing and when retrieving a file.

3.1 Backing Up a File

The storage of a file starts with a request that a user or an application issues to the PRODUCK CLIENT. The CLIENT first splits the file into chunks and superchunks as described above. Then, it starts the actual backup operation by interacting with the COORDINATOR and STORAGENODES. The details of such an interaction are depicted in Figure 3.

When the CLIENT has created the superchunks, it sends a PUTFILEREQ message to the COORDINATOR. The PUTFILEREQ contains the file's identifier (fileId), the number of superchunks in the file, their checksums, and the size of the file in bytes. The fileId is the SHA1 hash of the file content and is used in the retrieval process as a file-integrity check. Upon reception of the PUTFILEREQ, the COORDINATOR chooses a STORAGENODE uniformly at random and delegates the responsibility for the file to it. This guarantees that all nodes will be responsible for approximately the same number of files. After selecting a

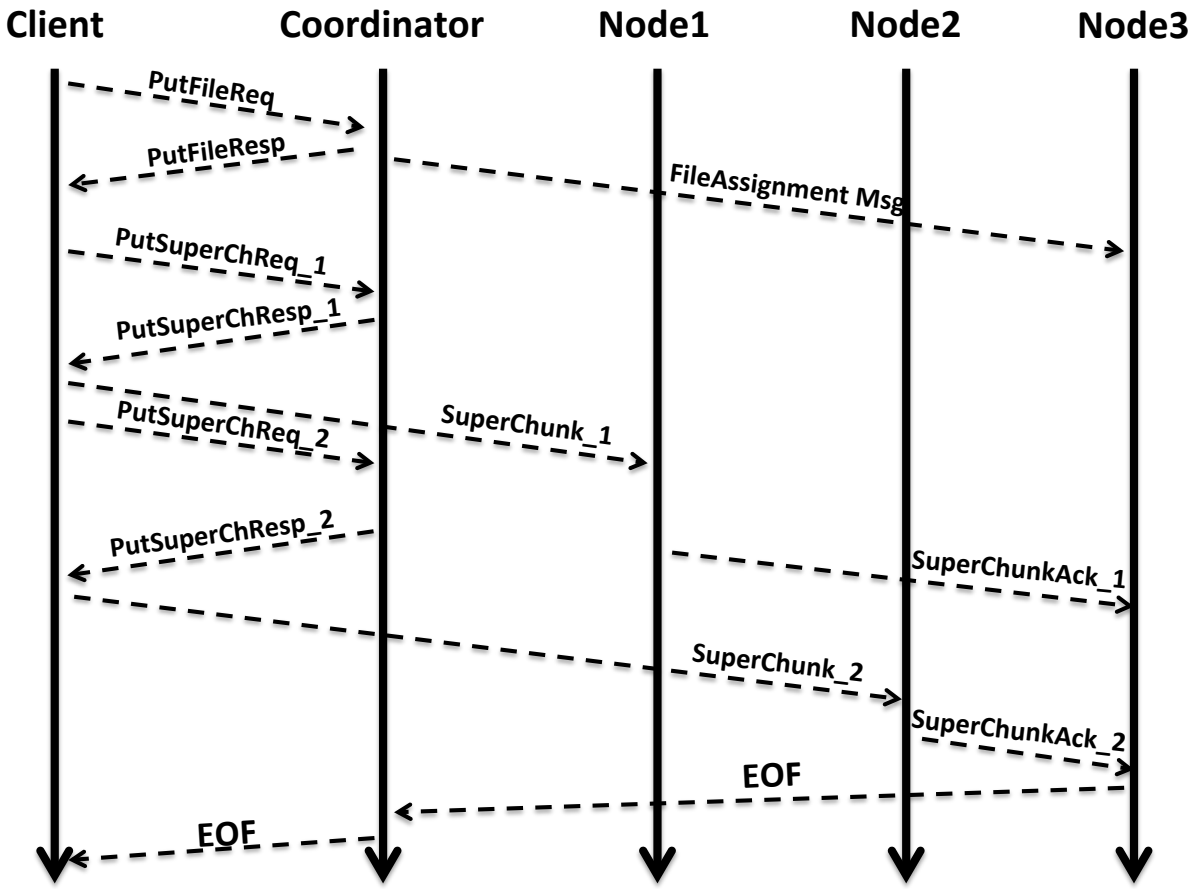


Figure 3: Message exchanges for storing a file.

responsible StorageNode, the Coordinator replies to the Client with the identifier of this node. In addition, it informs the responsible node about its selection and sends it all the information received by the Client in a FileAssignmentMsg.

The responsible node keeps track of which StorageNodes store each of the superchunks in the file. This allows the Coordinator to maintain state only on a per-file, instead of on a per-superchunk granularity. Furthermore, it contributes to responsiveness and scalability as this information is small enough to fit in the Coordinator's memory.

After the PutFileReq, the Client sends a PutSuperChReq request to the Coordinator for each of the superchunks in the file. This message contains the identifier of the superchunk and its bitmap. This is sufficient for the Coordinator to pick the best candidate among the StorageNodes as described in Section 2.3. After picking the right StorageNode, the Coordinator replies to the Client with a PutSuperChResp informing it of which node will store the superchunk. The client reacts by sending the actual data to the selected StorageNode in a SuperChunk message. When the transmission of the superchunk is over, the StorageNode that received it sends a SuperChunkAck message to the node responsible for the file. The SuperChunkAck contains the checksum of the data in the superchunk and the corresponding identifier. This enables the responsible node to record which StorageNode stored which superchunk, and allows it to check the integrity of the data received by the StorageNode. If something went wrong, the responsible node requests the Client to resend the data to the StorageNode.

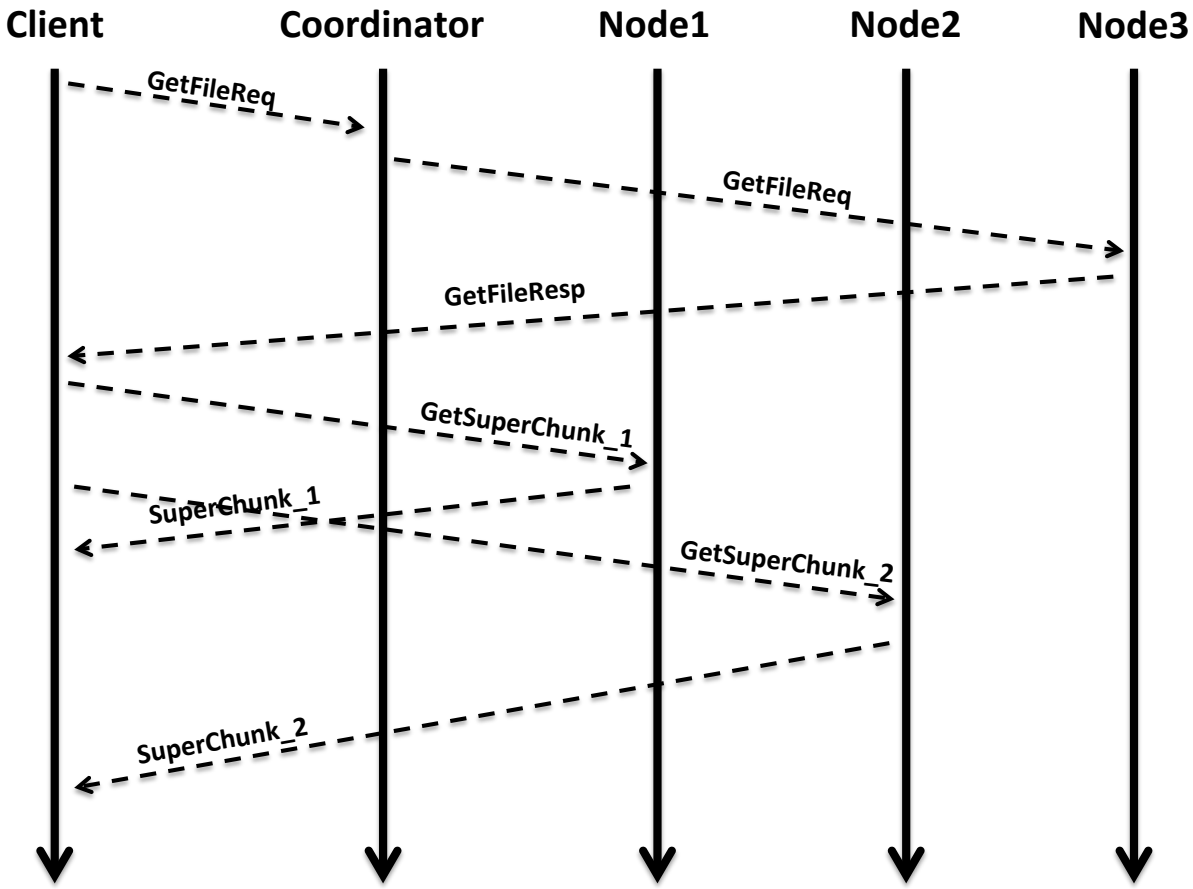


Figure 4: Message exchanges for retrieving a file.

The above process is repeated for all the superchunks in the file. When the responsible node has received acknowledgements for all the superchunks, it sends an EOF message to the Coordinator, which then forwards it to the Client.

3.2 Recovering a File

The process of reading a file is also initiated by a user request to the client. As depicted in Figure 4, the client reacts to this request by contacting the Coordinator with a GetFileReq message, specifying the identifier of the file to retrieve. The Coordinator looks up the responsible node for the file and forwards the GetFileReq message to it. The responsible node replies to the client by providing the identifier, checksum, and StorageNode for each superchunk in the file in a GetFileResp message. The client downloads each superchunk from the corresponding StorageNode and uses the checksums received from the responsible node to verify their integrity.

4. EXPERIMENTAL METHODOLOGY

In this section we present our experimental set-up: the datasets we used, the competitors we compared Produck against, and the metrics we evaluated. Since we focus on evaluating deduplication efficiency and load balancing with respect to storage, we evaluate Produck through simulations to avoid any networking side-effects. Yet, we built our simulator so that

the transition to a real implementation only consists in changing its communication primitives from in-memory transactions to network messages.

4.1 Datasets

We evaluate PRODUCK and its competitors using two real-world workloads. The first is publicly available for download from [1] and consists of 16 full snapshots of the English version of *Wikipedia*. The oldest dates back to March 2011, while the newest was taken in May 2012. This dataset contains full versions of all articles in *Wikipedia* and accounts for more than 520GB of data. The second dataset contains images of the environments available for deployment on the servers of the Grid5000 experimental platform [2] and accounts for 142GB. These workloads contain both text and binary data, thus covering many common cases. Table 1 presents more details on these datasets. In particular, the Deduplication Factor is the ratio of the original size of each dataset divided by its size after being deduplicated based on our *chunking* mechanism.

Dataset	**Size (GB)**	**Deduplication Factor**	**Data Format**
Wikipedia	522	1.96	HTML
Images	142	4.27	OS images

Table 1: Dataset Description

4.2 Competitors

Before evaluating the specific aspects of PRODUCK, we compare it against two state-of-the-art cluster-based deduplication storage systems presented in [6]: BLOOMFILTER, a *stateful* strategy, and MINHASH, a *stateless* one. The latter is used in a commercial product [6], thus representing the current state of commercial cluster-based deduplication systems. As in PRODUCK, both these strategies split files into chunks and superchunks using content-based chunking with the hash values of chunks serving as their signatures. The difference between them lies in the way each strategy selects the node that should store each superchunk.

- **BloomFilter:** in BLOOMFILTER, the COORDINATOR maintains a bloom filter for each storage node as an index of the node's contents. It then uses these bloom filters to assign each superchunk to the node that already stores most of the chunks it contains. To compute the overlap between a node's contents and the chunks in a new superchunk, the COORDINATOR checks the hashes of the chunks in the superchunk against the node's bloom filter. To improve performance during the bootstrap phases, the COORDINATOR ignores overlap values that are below a dynamically computed threshold. Moreover, to balance the load in the system, it considers a node eligible to store a superchunk only if its load does not exceed the average load of the system by more than 5%. This strategy is *stateful* since the decision of storing a new superchunk takes into account the location of existing ones.

- **MinHash:** MINHASH selects the minimum hash (*minhash*) of the chunks in a superchunk as the superchunk's signature. Instead of assigning superchunks to nodes, this strategy allocates superchunks to *bins*, which it then assigns to nodes. Typically $\#bins \gg \#nodes$. When assigning a superchunk to a *bin*, MINHASH applies the modulo-$\#bins$ operator to the superchunk's *minhash* value. To keep the system load balanced when a node becomes overloaded, i.e. when its load exceeds the average by more than 5%, MINHASH reassigns *bins* to nodes (*reshuffling*). In all cases, MINHASH assigns a superchunk to a bin by taking into account only the superchunk's content. This makes MINHASH a *stateless* strategy. Here, we have to note that for *reshuffling*, MINHASH has to keep in memory the state of each *bin*, *i.e.* the fingerprints of all the unique chunks plus those of the duplicates placed in non-optimal *bins*.

In all our experiments, we set the superchunk size for the two strategies mentioned above to 1MB. This value is quoted in [6] as the one giving the best results. We confirmed this by running MINHASH on our datasets, as we will show in Section 5. In addition, we set the number of *bins* for MINHASH to 100.

4.3 Evaluation Metrics

We evaluate PRODUCK along the same metrics as the ones used in [6], namely *Total Deduplication* (TD) representing deduplication efficiency, *Data Skew* (DS) capturing load imbalance and *Effective Deduplication* (ED) accounting for both the deduplication factor and load balance. In addition, we also measure the *Assignment Time* (AT) to capture the computational cost incurred by each of the assignment strategies: PRODUCK, BLOOMFILTER and MINHASH. We now detail each of these metrics.

- **Total Deduplication (TD):** TD is computed as the ratio between the original size of the dataset and its size after being deduplicated. The result is then normalized by the deduplication achieved on a single-node cluster. This metric measures how efficiently our system detects the duplicates present in the workload. Specifically, the normalization helps compare our performance to the optimal case of a single-node system with no load-balancing constraints.
- **Data Skew (DS):** DS is the ratio of the occupied storage space on the most loaded node to the average load in the system. DS captures the efficiency of a system in spreading the load equally among the available nodes. A high value of DS means that a node is overloaded and can result in duplicate data being sent to sub-optimal (in terms of deduplication) nodes.
- **Effective Deduplication (ED):** ED is defined as Total Deduplication (TD) divided by Data Skew (DS) and normalized by the TD achieved by a single node system. This metric encompasses both deduplication effectiveness and storage (im)balance. The intuition behind it, is that the performance of the whole cluster degrades if one node is overloaded. The reason for the normalization is the same as in the case of TD.
- **Assignment Time (AT):** Finally, the Assignment time, measured in seconds, measures the time it takes for each strategy to assign all the dataset to the available nodes. This metric provides an estimation of the computational cost of each strategy and can be viewed as an upper bound on the system's throughput as data cannot be stored before being assigned to a node.

5. EXPERIMENTAL RESULTS

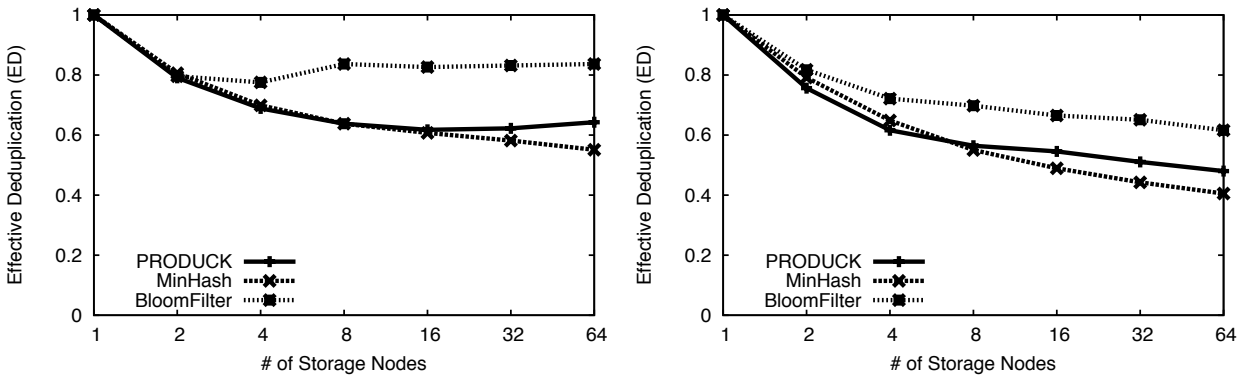
In this section, we provide a thorough evaluation of PRODUCK. In Section 5.1 we present the results of the comparison of PRODUCK against the two alternative cluster-based deduplication strategies presented in Section 4.2. We then dive into the specifics of PRODUCK in Section 5.2 and study how different values of its configuration parameters affect its performance. Finally, Section 5.3 shows the contribution of our *bucket*-based load-balancing mechanism to the overall performance of the system and presents how load balancing is preserved as more data is added to the system.

5.1 PRODUCK against Competitors

We first compare PRODUCK against the BLOOMFILTER (*stateful*) and the MINHASH (*stateless*) approaches across different cluster sizes. For all approaches, we use a chunk size of 1KB. However, for the sake of fairness, we set the other parameters to values that have been identified as optimal for each approach. The optimal superchunk size for MINHASH and BLOOMFILTER was identified by their authors [6] as 1MB. This is confirmed by our own experiments. With 64 nodes, MINHASH yields ED values of 0.49 and 0.26 for *Wikipedia*, and 0.27 and 0.13 for *Images* for superchunk sizes of 10MB and 100MB respectively, against values of 0.55 and 0.40 with a superchunk size of 1MB. For PRODUCK, we set the superchunk size to 15MB, the size of each *bucket* used for load balancing to 5 superchunks and the maximum allowed difference between the nodes to 1 *bukcet*. The reasoning behind these default values are presented in Section 5.2.

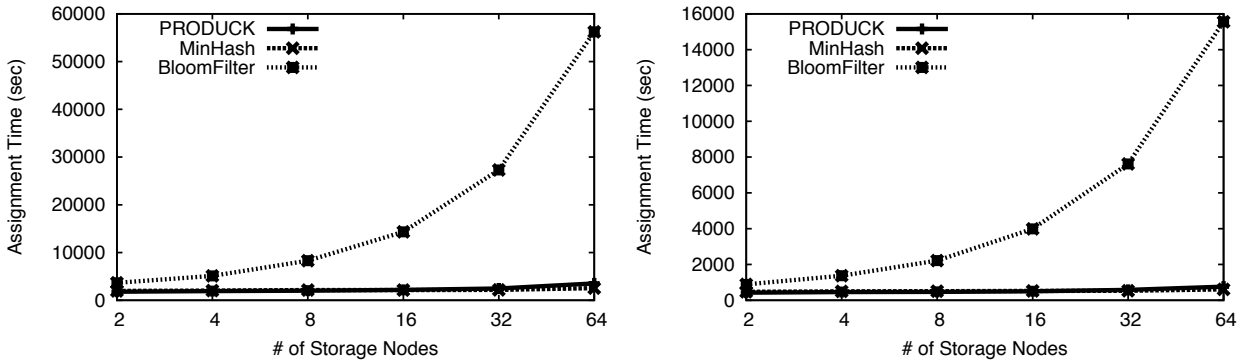
We are firstly interested in studying how each strategy manages the tension between the level of deduplication it achieves and the requirement for a load-balanced system. For this purpose, we choose to present in Figure 5 the Effective Deduplication (ED) of each system, as this metric combines deduplication effectiveness and load balancing. We observe from Figure 5 that PRODUCK outperforms MINHASH across both datasets for big clusters and that the difference increases as the cluster-size grows. In fact, focusing on 32- and 64-node clusters, which constitute the most difficult cases, as the more the nodes the more pronounced the deduplication/load balancing tension, we can see that PRODUCK improves ED by 7% and 16% for the *Wikipedia* dataset and 16% and 21% for the *Images* one when compared to MINHASH. In addition, for the *Wikipedia* dataset, which has the largest size after deduplication, we see that PRODUCK scales better than MINHASH as the cluster grows while for *Images* the two curves present similar behavior.

Although PRODUCK is outperformed by the BLOOMFILTER strategy, its ED is at most only 23% and 22% lower, respectively in the *Wikpedia* and *Images* datasets. The better ED of BLOOMFILTER is expected since this strategy sacrifices memory and computational resources to achieve almost accurate knowledge of a node's content at all times. This allows BLOOMFILTER



(a) *Wikipedia* (b) *Images*

Figure 5: Effective Deduplication (ED) for all datasets for Produck, BloomFilter and MinHash as a function of the cluster size.



(a) *Wikipedia* (b) *Images*

Figure 6: Assignement Time (AT) in seconds for all datasets for Produck, BloomFilter and MinHash as a function of the cluster size.

to leverage deduplication while ensuring that the load remains balanced. Yet, this also causes its cost to be significantly higher than that of PRODUCK as shown in Figure 6. The plot presents the AT metric, i.e. the time needed by each strategy to assign all the data in a given dataset to the available storage nodes. This metric, apart from being an illustration of computational cost, also serves as an upper bound on the throughput of each storage system. Figure 6 clearly indicates that the computational cost of BLOOMFILTER is much higher than those of both PRODUCK and MINHASH, and that the difference increases as the cluster size grows. For a 32-node cluster, BLOOMFILTER is more than 11 times slower than PRODUCK for *Wikipedia* and more than 13 times for *Images* while for a 64-node one, these values become 16 and 21 respectively. More importantly, the memory overhead induced by the maintenance of the Bloom filters remains far from negligible. To achieve a 1% rate of false positives, a Bloom filter requires 9.6 bits per element. Given a chunk size of 1KB, for a storage node with a capacity of 140TB as the ones mentioned in the commercial version of [6], the Bloom filter can reach the size of 168GB. And this is just for 1 storage node. The amount of memory required by the COORDINATOR should be multiplied by the number of storage nodes in the system. These drawbacks were also observed by the authors of [6] who addressed this issue by sampling the chunks in a superchunk at a frequency of 1 over 8. We applied the same strategy and present the corresponding results for ED and AT in the last two columns of Table 2, along with those for all the other metrics and strategies.

Table 2 shows the results of all the metrics, including Data Skew (DS) and Total Deduplication (TD) for all the considered approaches. From the table, it is clear that PRODUCK consistently improves TD with respect to the MINHASH approach

nodes	Produck				MinHash				BloomFilter					
	ED	DS	TD	AT	ED	DS	TD	AT	ED	DS	TD	AT	ED (1/8)	AT (1/8)
Wikipedia														
1	1.00	1.00	1.00	-	1.00	1.00	1.00	-	1.00	1.00	1.00	-	1.00	-
2	0.79	1.00	0.79	1795	0.80	1.00	0.80	1959	0.80	1.00	0.80	3655	0.79	2177
4	0.69	1.00	0.69	1916	0.70	1.00	0.70	2035	0.78	1.00	0.79	5108	0.66	2484
8	0.64	1.00	0.64	2015	0.63	1.01	0.64	2135	0.84	1.01	0.86	8294	0.59	2962
16	0.62	1.00	0.62	2170	0.60	1.01	0.61	2146	0.83	1.02	0.87	14344	0.56	3819
32	0.62	1.00	0.62	2466	0.58	1.03	0.59	2171	0.83	1.05	0.87	27292	0.64	5304
64	0.64	1.01	0.65	3541	0.55	1.05	0.58	2548	0.84	1.05	0.88	56237	0.79	9302
Images														
1	1.00	1.00	1.00	-	1.00	1.00	1.00	-	1.00	1.00	1.00	-	1.00	-
2	0.76	1.00	0.76	426	0.79	1.00	0.79	472	0.82	1.00	0.82	889	0.74	529
4	0.62	1.00	0.62	457	0.65	1.00	0.65	500	0.72	1.05	0.76	1370	0.56	642
8	0.56	1.01	0.57	470	0.55	1.02	0.56	506	0.70	1.05	0.73	2216	0.44	799
16	0.55	1.01	0.55	503	0.49	1.03	0.50	521	0.67	1.05	0.70	3989	0.47	1049
32	0.51	1.02	0.52	582	0.44	1.05	0.46	531	0.65	1.05	0.68	7616	0.54	1530
64	0.48	1.01	0.48	762	0.40	1.08	0.44	610	0.62	1.05	0.65	15555	0.57	2763

Table 2: Effective Deduplication (ED), Data Skiew (DS) , Total Deduplication (TD) and Assignment Time (AT) as functions of the cluster size. The last two columns of the table are for the case where sampling (1 over 8) is applied to the superchunks to reduce the number of comparisons and the size of the bloom filter.

for large clusters, while ensuring that the load remains equally spread among nodes. In fact, DS does not surpass 1.02, which corresponds to the most loaded node being only 2% more loaded than the average. More importantly, the fact that BloomFilter provides a better deduplication factor comes at the price of a high computational and memory cost. After applying sampling, we see that its ED drops significantly in both datasets and across most cluster sizes, becoming lower than that of Produck in most of the cases. At the same time, its AT remains 3 to 4 times larger than that of Produck. In addition, although sampling reduces the sizes of the Bloom filters by 8 times, the total amount of memory required for storing the Bloom filters on the Coordinator remains large: 64×21GB for a system of 64 storage nodes of 140TB each.

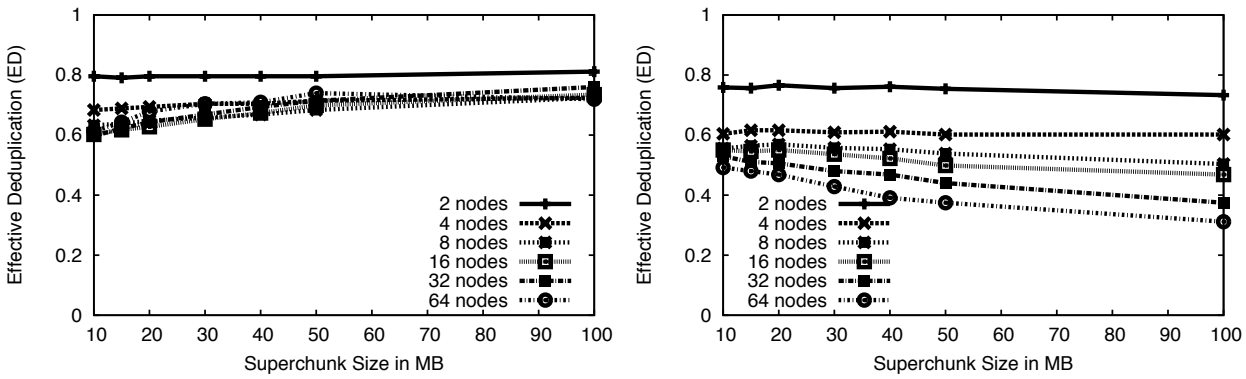
5.2 PRODUCK Sensitivity Analysis

In this section, we study the impact of various configuration parameters on the performance of Produck. This contributes to (i) illustrating the tradeoffs in cluster-based deduplication, and (ii) revealing the reasoning behind the default values we selected for our system. The parameters we are interested in are (i) the size of the superchunk, (ii) the size of the *buckets* used for load balancing, and (iii) the maximum allowed difference in the number of used storage *buckets* between the most and the least loaded node.

The results of this analysis are presented in Figures 7, 8, and 9 across different cluster sizes. This provides an idea of the evolution of the system as the cluster grows. In the plots, the size of the superchunk is measured in groups of 1024 consecutive chunks, which correspond to MBs as the average chunk size is of 1KB. The default superchunk size is 15MB. The bucket size is instead expressed in number of superchunks, with a default value of 5. Finally, the maximum allowed *bucket* difference is expressed in number of *buckets*, and has a default value of 1. In each experiment, we vary one of the above three parameters while keeping the other two constant at their respective default values.

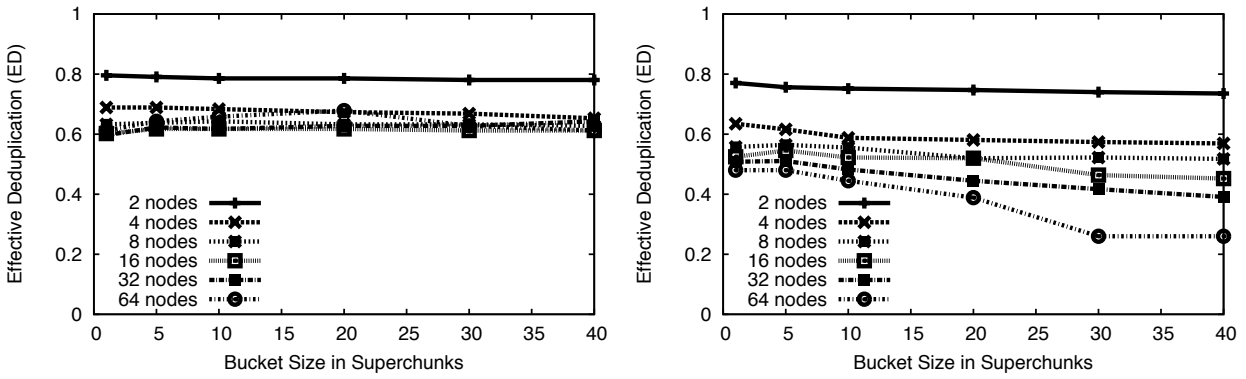
Superchunk size.

Figure 7 shows how the values of ED are affected by the size of superchunks for various cluster sizes. In each of the datasets, performance is best for a specific superchunk size. In the case of *Wikipedia* the best ED is achieved with a size of around 50MB, while for *Images* the maximum is around 10MB. The difference between the optimal values of the two datasets reflects their different properties. As a general rule, smaller superchunks are better for workloads characterized by larger deduplication factors. This is because smaller superchunks improve the ability to detect redundancy and thus provide a greater advantage for workloads characterized by the presence of a large number of duplicates. Nonetheless, even in the case of *Images*, too



(a) *Wikipedia* (b) *Images*

Figure 7: Effective Deduplication (ED) for both datasets for different superchunk and cluster sizes.



(a) *Wikipedia* (b) *Images*

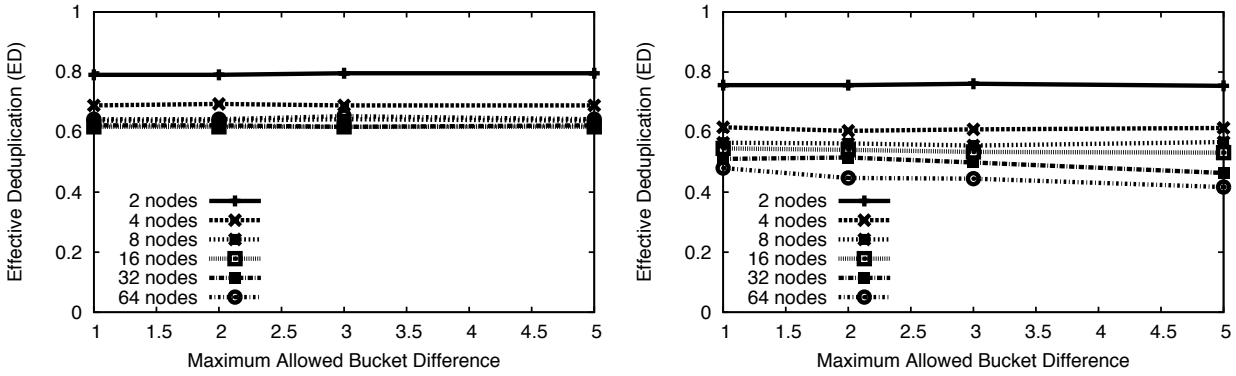
Figure 8: Effective Deduplication (ED) for both datasets for different *bucket* and cluster sizes.

small superchunk sizes prove to be disadvantageous. The reason lies in the estimation error inherent in PCSA. As observed in Section 2.3, the estimation error is larger when the multisets (superchunks) being considered are small. The choice of the best superchunks size therefore requires a balance between these two aspects. In this paper, we opted for a default value of 15MB. This value not only provides good results on the considered datasets, but it is also close to the optimal value for *Images*, which, according to the study in [23], is a good example of real workloads. We are investigating the possibility of dynamically determining the best superchunk size for the workload being considered.

Figure 7 also shows another interesting fact. In the case of *Images*, ED tends to drop as the cluster size grows. This negative impact of the cluster size on ED is expected. The bigger the cluster the more the load-balancing mechanism will spread the data among the cluster nodes. Yet, this is at odds with the effort of the deduplication mechanism that wants data to be concentrated on as few nodes as possible so that more duplicates end up on the same server. The figure also shows that the impact of the cluster size is much less significant for the *Wikipedia* dataset. Again, this is because the larger number of duplicates in *Images* make it more important to keep data clustered.

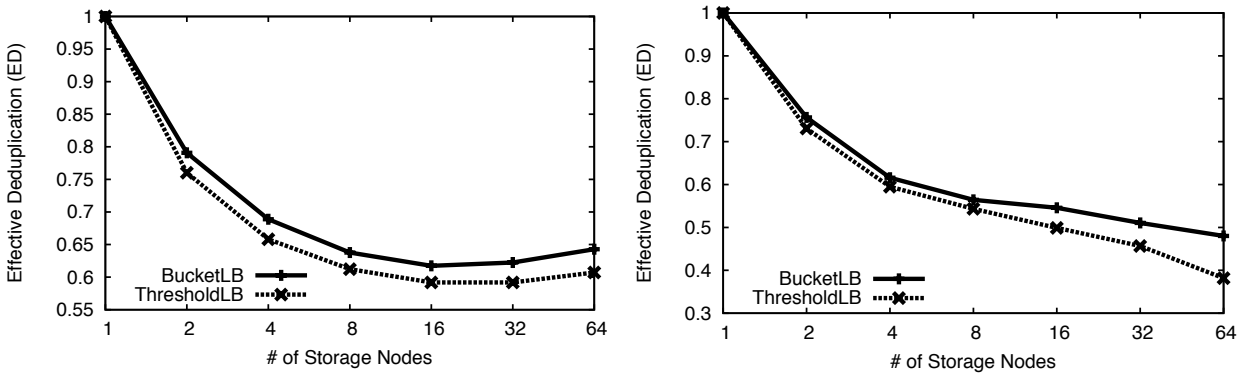
Bucket size.

Figures 8 analyze the values of ED for increasing *bucket* sizes. A first conclusion we can draw from the graphs is that ED either stabilizes or drops for *bucket* sizes greater than 10 superchunks. In the case of the *Wikipedia* dataset, where duplicates are scarce, all *bucket* sizes perform almost the same and, once again, the cluster size does not have a big impact on ED beyond



(a) *Wikipedia* (b) *Images*

Figure 9: Effective Deduplication (ED) for both datasets for different values of maximum allowed *bucket* difference between the most and the least loaded nodes and across different cluster sizes.



(a) *Wikipedia ED* (b) *Images ED*

Figure 10: Effective Deduplication (ED) for both datasets for two load balancing strategies (i) our *bucket*-based one, here called *BucketLB*, and (ii) the one from [6], namely *ThresholdLB*, where the storage load of a node is not allowed to deviate by more than 5% from the average load in the system.

a size of 2 nodes (ED varies from 0.69 to 0.61). For the *Images* dataset, where deduplication has more potential for space reduction, smaller *bucket* sizes have better performance. More precisely, a *bucket* size of 5 seems to be the optimal value for most cluster sizes. This may seem counter intuitive as bigger *bucket* sizes mean fewer interventions of the load-balancing mechanism, and thus more deduplication. However, a closer look at the results of the experiments reveals that, in most cases, although the achieved deduplication drops slightly with increasing bucket sizes, the drop in ED is mostly due to an increase in the load imbalance (DS) among the nodes. This, in turn, leads to more non-optimal node assignments because of the effort to spread the load equally, ultimately causing a small drop in deduplication.

Based on this analysis, we set PRODUCK's default *bucket* size to 5 superchunks. This keeps the system load balanced even for small datasets like *Images* on big clusters (for *Images* on a 64 node cluster, the value of DS is 1.01) while giving good deduplication results. For larger datasets, equal load distribution is also guaranteed as the maximum deviation of a node from the average load in the system is bounded. We further examine the efficiency of PRODUCK's load-balancing strategy in Section 5.3.

Maximum allowed bucket difference.

nodes	Wikipedia				Images			
	Bucket		Threshold		Bucket		Threshold	
	TD	DS	TD	DS	TD	DS	TD	DS
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.79	1.00	0.8	1.05	0.76	1.00	0.77	1.05
4	0.69	1.00	0.69	1.05	0.62	1.00	0.62	1.05
8	0.64	1.00	0.64	1.05	0.56	1.01	0.56	1.05
16	0.62	1.00	0.62	1.05	0.55	1.01	0.52	1.05
32	0.62	1.00	0.62	1.05	0.51	1.02	0.48	1.06
64	0.65	1.01	0.64	1.05	0.48	1.01	0.41	1.06

Table 3: Total Deduplication (TD), and Data Skiew (DS) for both datasets and for both load balancing strategies, *BucketLB* and *ThresholdLB*, as a function of the cluster size.

Finally, Figure 9 presents the performance of PRODUCK for different levels of allowed load imbalance among STORAGENODES. The plot shows that increasing the maximum allowed load imbalance does not have a significant impact for the *Wikipedia* dataset. The same also holds for *Images*, although here, the impact is a bit more pronounced. This is rather expected as in *Images* there are more duplicates. For very small cluster sizes, the maximum allowed *bucket* difference does not significantly affect the achieved ED. When the cluster grows, a maximum allowed difference of 1 *bucket* gives the best results.

5.3 Load Balancing and Evolution with Time

The whole design of PRODUCK was driven by the tension between the requirement for efficient deduplication and the one for a load-balanced system. We now study the impact of our novel load-balancing strategy on the performance of PRODUCK. To this end, we compare our *bucket*-based mechanism (*BucketLB*) to the one used by the BLOOMFILTER strategy in [6], here termed *ThresholdLB*. In *ThresholdLB*, the storage load of a node is not allowed to deviate by more than 5% from the average load in the system. We ran PRODUCK using both load balancing strategies for different cluster sizes. Figure 10 presents the Effective Deduplication (ED) and Table 3 presents the Total Deduplication (TD) and Data Skew (DS) achieved in both datasets. In terms of ED, *BucketLB* outperforms *ThresholdLB* across all cluster sizes and datasets. In fact, on a 64-node cluster, *BucketLB* achieves 4.2% and 10.3% better ED, respectively, on *Wikipedia* and *Images*. From Table 3, we can see that for small cluster sizes, both strategies achieve the same total deduplication, but *ThresholdLB* pays a higher price in terms of load imbalance, as its DS is constantly higher than that of *BucketLB*. As the cluster grows and the intervention of the load-balancing mechanism is more pronounced, we see that *BucketLB* also enables PRODUCK to achieve better deduplication especially in workloads with more duplicates. Ultimately, this allows PRODUCK to achieve the same or better deduplication while keeping the system more load balanced. In addition, its benefits are more pronounced as the cluster grows. This leads to the conclusion that *BucketLB* provides PRODUCK with better scalability, a highly desired property as deduplication clusters are expected to grow constantly due to the fast pace at which digital data is produced [10].

We also examine how our load-balancing mechanism behaves as more data is added to the system. We focus on the *Wikipedia* dataset, which is the largest one after deduplication, and we store the snapshots in the order they were taken while recording system statistics after each snapshot is stored. We do so in order to make our results as close to a real deployment as possible. The same conclusions hold for the other dataset. The results for Data Skew in 32- and 64-node clusters are presented in Figure 11. We focus on big clusters for two reasons. First, backup clusters are expected to grow constantly, as mentioned earlier. Second, the effect of the load-balancing mechanism on deduplication is more pronounced on large clusters as data is more scattered across nodes. Figure 11 shows that our *bucket*-based load-balancing policy manages to guarantee an equally distributed storage load even from the first few GBs stored in the system. In a 64-node cluster, after only the first snapshot has been stored (29GBs), the most loaded node does not store more than 10% more data than the average load in the system (DS=1.1). In addition, the load imbalance drops fast and after 6 snapshots, DS does not surpass the value of 1.03. The drop in the load imbalance is expected as nodes are allowed to deviate by one bucket at most and buckets are of fixed size. As more data is added to the system, and the average load increases, the size of a bucket becomes smaller as a percentage of the increasing average load, thus causing a continuous decrease in DS. The good performance of PRODUCK in terms of load balancing allows it to avoid periodic rebalancing operations as is instead done in MINHASH [6].

Finally, although we considered nodes equipped with disks of equal size, this does not limit the applicability of PRODUCK and its load-balancing strategy. In the case of nodes with different disk sizes, each node can be assigned a weight equal to the

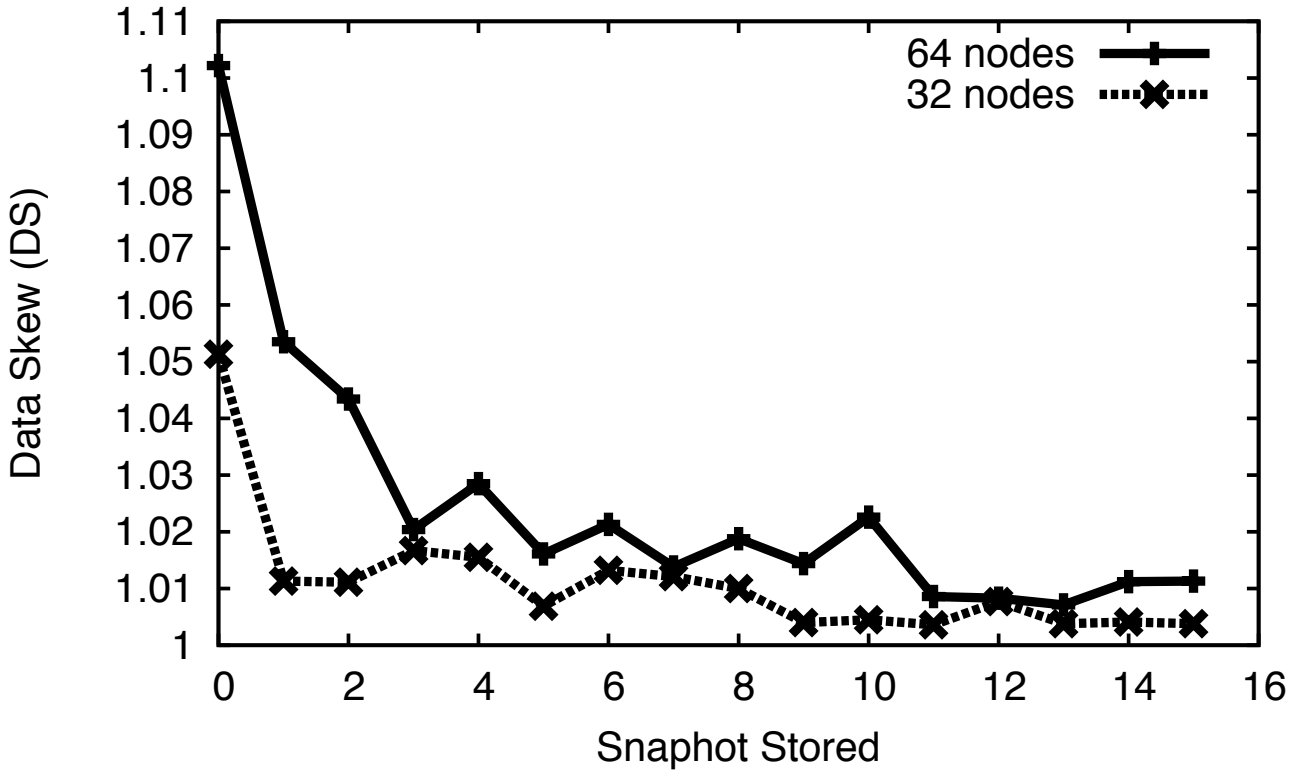


Figure 11: Evolution of the Data Skew (DS) for the *Wikipedia* dataset as data is stored at their chronological order.

ratio of its disk size to the maximum disk size in the system. This allows our load-balancing mechanism to load each node proportionally to its capacity.

6. RELATED WORK

Deduplication is widely used in many fields, ranging from file systems [17, 22], faster data tranfers [19] and copy-detection mechanisms [5]. As a consequence, multiple algorithms have been proposed. These contain whole-file duplicate detection [3], fixed size chunking [20] and content-based chunking [17]. In addition, many studies [13, 15, 18] compare these strategies and the main finding is that content-based chunking outperforms its counterparts.

A parameter that has an important impact on the deduplication efficiency of the system is the chunk size it uses. As a rule of thumb, smaller chunk sizes result in better duplicate detection while bigger ones incur less overhead due to metadata maintenance and achieve better throughput. The authors of [21] propose a compact, multi-resolution fingerprinting methodology that tries to estimate similarity between data items at different chunk sizes. HYDRAstor [7] and HydraFS [22], a filesystem based on HYDRAstor, manage to provide high throughput by using bigger chunks (64KB). Chunks are routed by partitioning the fingerprint space evenly across storage nodes. In [12], the authors focus instead on deduplicating streams and try to balance the tradeoff between throughput and deduplication. Specifically, they identify, with high probability, where in the stream new data is likely to be added and they use smaller chunks around these points.

Among deduplicated storage systems, in Sparse Indexing [14], the authors focus on decreasing the index of the chunks stored in the system by using low-rate *sampling* techniques. Data is split into segments in a content-based manner and each segment is deduplicated against a few others. This is shown by the authors to result in small deduplication losses. This, and other existing single-node deduplication solutions such as [11], can be combined with PRODUCK as it is independent of the

architecture of its storage nodes. This was a design decision since the beginning as it allows PRODUCK to leverage the findings of existing research.

Moving on to cluster-based deduplication solutions, the authors of [4] propose the use of MINHASH as in [6] but on a per-file granularity to route files to the appropriate storage node. The authors of [6] show this method to be suboptimal when it comes to backups of big sets of files and propose the segmentation of input data into superchunks, with chunks of the same superchunk being sent to the same node. The authors of [6] also propose BLOOMFILTER, a stateful solution which uses Bloom filters to estimate the overlap between superchunks and storage nodes. However, the large memory cost of maintaining Bloom filters makes this strategy impractical for real backup systems. We compared PRODUCK against both MINHASH and BLOOMFILTER in Section 5. As we explained in Section 4, BLOOMFILTER also uses an overlap threshold that could in principle be used to mitigate the estimation error of PCSA. We experimented with this technique in previous versions of PRODUCK. Nonetheless, its impact was only measurable when using relatively small numbers of BITMAP vectors (e.g. $m = 1024$ instead of $m = 8196$).

7. CONCLUSION

In this work, we presented PRODUCK, a *stateful* yet lightweight deduplication solution for cluster-based storage systems. We compared our solution to state-of-the-art *stateful* and *stateless* techniques over two real-world workloads of close to 700GB in total and we showed that PRODUCK achieves an 18% average improvement in deduplication with respect to existing *stateless* solutions while reducing the superchunk-assignment time by up to 18 times, when compared to *stateful* ones. In addition, this is achieved with minimal computational and memory costs. Specifically, the memory overhead is limited to an indexing structure of 64KB per storage node. Our findings demonstrate that *stateful* solutions can actually be used in practice.

Apart from the above conclusions, our study has two main contributions. The first is the introduction and evaluation of a compact indexing data structure in the context of data *deduplication*. This structure allows the COORDINATOR to accurately detect the overlap between the contents stored by a node and those of a superchunk, with minimal memory requirements ($64KB$) and minimal computational cost. The second contribution is a novel, *bucket*-based load-balancing mechanism. This mechanism manages to maintain the system load balanced while not sacrificing duplicate detection and without requiring any additional rearrangement of data to nodes, which could result in huge amounts of traffic.

In the future, we are planning to investigate the possibility of dynamically determining the optimal size of superchunks based on the data to be stored. In our current system, superchunk size is a system-wide parameter set at deployment time. The chosen size may not be optimal for some workloads as shown in Section 5. Dynamically finding the optimal superchunk size for each workload would further enhance the performance of PRODUCK. Finally, we are working on a real implementation of PRODUCK which will enable us to study its behavior in real settings.

Acknowledgments.

We are grateful to the anonymous reviewers for their constructive suggestions and comments, which helped us improve the final version of this paper. The work we presented was supported by the ERC Starting Grant GOSSPLE number 204742. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

8. REFERENCES

[1] http://dumps.wikimedia.org/enwiki/.

[2] https://www.grid5000.fr/.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI*, 2002.

[4] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *MASCOTS*, 2009.

[5] S. Brin, J. Davis, and H. Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *SIGMOD*, 1995.

[6] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in Scalable Data Routing for Deduplication Clusters. In *FAST*, 2011.

[7] 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.

[8] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, 2003.

[9] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31, 1985.

[10] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth Through 2011. Technical report, An IDC White Paper - sponsored by EMC, March 2008.

[11] Fanglu Guo and Petros Efstathopoulos. Building a High-performance Deduplication Systems. In *USENIX ATC*, 2011.

[12] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *FAST*, 2010.

[13] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX ATC*, 2004.

[14] 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.

[15] Dutch T. Meyer and William J. Bolosky. A Study of Practical Deduplication. In *FAST*, 2011.

[16] Sebastian Michel, Matthias Bender, Nikos Ntarmos, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Discovering and Exploiting Keyword and Attribute-Value Co-occurrences to Improve P2P Routing Indices. In *CIKM*, 2006.

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

[18] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *USENIX ATC*, 2004.

[19] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting Similarity for Multi-Source Downloads Using File Handprints. In *NSDI*, 2007.

[20] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *USENIX ATC*, 2008.

[21] Kanat Tangwongsan, Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Efficient Similarity Estimation for Systems Exploiting Data Redundancy. In *INFOCOM*, 2010.

[22] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST*, 2010.

[23] 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.