Deduplication algorithms and models for efficient data storage

Laura Conde-Canencia, Belaid Hamoum

Deduplication algorithms and models for efficient data storage

Laura Conde-Canencia
Lab-STICC, CNRS UMR 6285
Université Bretagne-Sud
Lorient, France
laura.conde-canencia@univ-ubs.fr

Belaid Hamoum
Lab-STICC, CNRS UMR 6285
Université Bretagne-Sud
Lorient, France
belaid.hamoum@univ-ubs.fr

***Abstract*—This paper is dedicated to data deduplication algorithms and models that lead to efficient solutions to reduce the amount of data both transmitted over the network and stored in data systems. To be specific, we consider the case where replicas of an original file are generated by edit errors and adopt a theoretical approach to explore data files. Our study can apply to primary, backup or archival storage. We introduce a new variable-length block-level deduplication algorithm that outperforms prior work and reduces the computational complexity by focusing on *pivots*. We provide a theoretical comparative analysis of the algorithm computational costs and experimental results to evaluate its performance. The proposed deduplication solution enhances prior approaches in terms of cost and achieves the same rates as brute force or naive methods.**

***Index Terms*—Data deduplication, inline data processing, edit channel, insertions/deletions/substitutions, deduplication ratio, brute force methods.**

I. Introduction

Data deduplication is an emerging technology that improves storage utilization and offers an efficient way of handling data replication. In other words, data deduplication optimizes free space on a volume by examining the data on the volume and looking for duplicated portions on it. Currently, deduplication is a key feature in many cloud and enterprise server settings such as, for example, Microsoft OneDrive and Windows Server. Such systems run data deduplication in the background for optimizing the Hyper-V VDI environment, backup storage and file servers.

The principle of data deduplication is that redundant data blocks are removed and replaced with pointers to a unique data copy leading to reduced storage costs (i.e., power, cooling, floor space requirements, ...). The higher cost reductions are achieved with the higher *deduplication ratios* because, obviously, the less disk capacity is needed for storage.

Compared to classical data compression [1] [2], which is based on small amounts of local redundancy (i.e., in the order of 10 to 258 bytes), data deduplication considers larger amounts of global redundancy. To be specific, the amounts of local redundancy in data compression are in the order of 10 to a few hundreds of bytes. In data deduplication, ranges of up to hundreds of kilobytes have been reported with source data of a few hundreds of gigabytes [3].

Prior art in deduplication is mostly authored by the computer science community ([4] [5] among others) and contributions mainly concern hash algorithms [6]. However, only two works [7] [8] have considered the problem of deduplication from an information theory point of view. This paper focuses on significantly improving the deduplication performance of the work in [8] while keeping the low-cost characteristics.

The remainder of the paper is organized as follows: Section II briefly describes the types of data deduplication as well as the context and contribution of this paper. Section III presents the problem statement, notation and definitions. Section IV describes the innovative data deduplication algorithm which is based on pivot matching and sliding windows techniques. A theoretical analysis on the cost of the new algorithm compared to brute force methods and prior work is presented in Section V. Experimental results and data deduplication ratios are presented and discussed in Section VI. Finally, Section VII concludes the paper.

II. Types of data deduplication

Data deduplication can operate at three different levels [6]: file, fixed-length block or variable-length block. A simple example of *file level deduplication* would be the following: consider a system that retains 100 e-mails, each with the same 2 MB attachment; by applying file deduplication, the 200 MB needed to store the attachments would be reduced to 2 MB. *Block-level deduplication* looks within a file and saves unique replicas of each block. Files can be broken into blocks of the same size (fixed-length block deduplication) or into blocks of various sizes depending on their contents (variable-length block deduplication). This last alternative allows the deduplication effort to achieve better deduplication ratios [9]. In this paper we consider variable-length block-level deduplication.

Considering the kind of data processing, there exists two main methods to deduplicate redundant data: *inline* and *post-processing* deduplication. *Inline deduplication* analyzes data as it enters the backup system. Redundancies are removed as the data is written to backup storage. *Post-processing deduplication* is an asynchronous backup process that removes redundant data after it is written to storage. Duplicate data is removed and replaced with a pointer to the first iteration of the block. Compared to post-processing deduplication, inline deduplication requires less backup storage, but can cause bottlenecks. The approach in this paper is mainly adapted to

inline deduplication and specifically focuses on reducing costs and avoiding bottlenecks.

III. Preliminaries and notation

In this work we consider servers storing data files and how to reduce the storage needs by deduplicating a file totally or partially. We specifically study the problem of deduplicating file Y, i.e. not storing it in the system, but only providing a point to file X (or a part of it). For this we consider both files to be sufficiently similar under *edit errors*, and assume that they were originated from the same source.

A. Definitions related to the files

File X contains symbols that are drawn from a non-binary alphabet according to an arbitrary distribution. The number of bits to represent a symbol of the non-binary alphabet is denoted by q and we consider n to be large enough. Also, file X is partitioned into substrings as:

$$X = S_1, P_1, S_2, P_2, \ldots, S_{k-1}, P_{k-1}, S_k, P_k$$

where S_i is a *segment* substring, P_i is a *pivot* substring and k is the number of both pivots and segments.

A *block* is a substring that corresponds to the concatenation of one or several consecutive S_i, P_i pairs. A *chunk* is a block whose content is identical in X and Y and can thus be deduplicated.

The length of the *segment* substrings is L_S, the length of the *pivot* substrings is L_P and we assume the file length to be divisible by $L_B = L_S + L_P$. We consider $L_S >> L_P$ and L_P to be small enough so that the probability that a pivot contains an *edit* is relatively low, but long enough to include enough symbols to ensure a comparison that delivers enough information.

The substring $X(i,j)$ corresponds to symbols $X(i), X(i+1), \ldots, X(j)$, with $1 \leq i \leq j \leq n$ and its length is $l = j-i+1$. Also, $n = \sum_{t=1}^{b} l_t$ where b is the number of blocks in a file, l_t is the length of block t and $t = 1, 2, \ldots, b$. Note that l_t can differ from one block to another (i.e., variable-length blocks).

B. Edit channel model

In our model, we consider that the edit events (i.e., edit errors) are applied to the data string sequentially, as follows: let an r-length *edit pattern* $E = (E_1, E_2, \ldots, E_r)$ be defined so that the output file Y of an edit channel is obtained from file X as in Figure 1. For $1 \leq t \leq r$, where $n \leq r < \infty$ and n is the length of X,

- If $E_t = 0$, X_j is stored and the process moves on to symbol X_{j+1}. This occurs with probability $1 - \beta_D - \beta_I$.
- If $E_t = -1$, X_j is deleted and the process moves on to symbol X_{j+1}. *Idem* with probability β_D.
- If $E_t = 1$, a new symbol taken from distribution $\mu(x)$ is inserted. *Idem* with probability β_I.

The number of *net edits* is defined as $r_{ne} = \sum_{t=1}^{r} E_t$ and corresponds to the number of insertions minus the number of deletions. As in [10] [8], we consider the probability of an edit $\beta = \beta_I + \beta_D$ to be arbitrarily small, n to be large enough and the same edit channel model as in [10] with $\beta_I = \beta_D$.

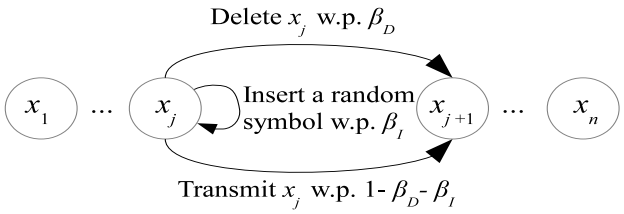

Fig. 1: Edit channel model: the states are the n symbols of file X. A file symbol may experience a symbol insertion with probability β_I and the edit process remains in the same state. The following file symbol or next state is reached by either deleting the current file symbol (with probability β_D) or transmitting it (with probability $1-\beta_I-\beta_D$). A substitution occurs when an insertion immediately follows a deletion. "w.p." stands for "with probability".

IV. Deduplication algorithm based on pivots with sliding windows

This Section describes the contribution of this paper which is the new deduplication algorithm with the following characteristics: variable-length, block-level and pivot-based. Note that in [8] a pivot-based solution was already introduced but it showed limited deduplication ratios due to major drawbacks.

In this Section we first present some necessary definitions to describe the new deduplication algorithm. We then introduce the sliding-window pivot-based algorithm and its core component.

A. Definitions

Let pivot P be $(p_1, p_2, \ldots, p_{L_P})$. The *shift* operator $T(P,u)$ shifts the symbols in P $|u|$ positions to the right if $u > 0$ or to the left if $u < 0$. For example:

- $T(P,2) = (\Delta, \Delta, p_1, p_2, \ldots, p_{L_P-2})$,
- $T(P,-3) = (p_4, p_5, \ldots, p_{L_P}, \Delta, \Delta, \Delta)$

where Δ is the null value and $p_i \odot \Delta = 0$, $\forall p_i$.

Let $P_x = (p_{x,1}, p_{x,2}, \ldots, p_{x,L_P})$ be a pivot in X and $P_y = (p_{y,1}, p_{y,2}, \ldots, p_{y,L_P})$ the corresponding pivot in Y.

The *pivot-matcher* operator is defined as:

$$P_x \odot P_y = (p_{x,1} \odot p_{y,1}, p_{x,2} \odot p_{y,2}, \ldots, p_{x,L_p} \odot p_{y,L_p}).$$

where $\odot$ is the *symbol-matcher* operator, defined as:

$$p_{x,i} \odot p_{y,j} = \begin{cases} 1 & \text{if } p_{x,i} = p_{y,j} \\ 0 & \text{if } p_{x,i} \neq p_{y,j} \end{cases}$$

P_y is a *good match* of pivot P_x if $P_x \odot P_y = (1, 1, \ldots, 1)$ and in this case $r_{ne} = 0$.

B. Principle of the sliding-window pivot-based algorithm

As already described, the algorithm partitions the files into alternating components, called pivots and segments, with the length of pivots being much smaller than the length of segments. The idea is to significantly reduce the computational costs related to symbol comparisons and improve naive or brute force methods by only focusing on the pivots.

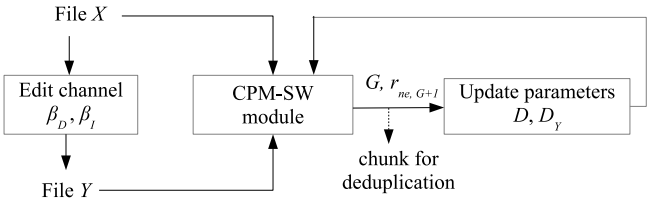

Fig. 2: Principle of the sliding-window pivot-based algorithm.

The core element of the proposed protocol is the Consecutive-Pivot-Matching Sliding-Window (CPM-SW) module which determines the size of the chunks. The algorithm uses the CPM-SW module as many times as necessary until all the pivots in X are compared to the corresponding subsequences in Y. The system knows at the end of the algorithm which chunks can be deduplicated and which blocks contain edit errors and cannot be deduplicated.

The key parameters in the algorithm are:

- D_x: first position of the pivot in file X, from which the comparison starts.
- D_y: first position of the pivot in file Y.
- G: number of consecutive matched pivots, which is provided for each chunk by the CPM'SW module.

The general principle of the algorithm is:

1) Consider substrings of files X and Y to start comparison at the first pivot.
2) Use the core module (CPM-SW in Figure 2) to determine the size of the chunk for deduplication and the number of net edits in the last segment (i.e., $r_{ne,G+1}$).
3) Update substrings of files X and Y to continue the comparison (with the CPM-SW Module) until the end of the files is reached.

Fig. 2 and Algorithm 1 provide further details to this description.

Data: Files X and Y, parameters L_S and L_P.
Result: Chunks for deduplication.
Initialization: $D_x = D_y = L_S + 1$;
while $D_x < n$ **do**
  Execute the CPM-SW module. Inputs are:
  $X(D_x, n)$ and $Y(D_y, n_y)$;
  Outputs are: G and $r_{ne,G+1}$;
  Chunk for deduplication is:
  $Y(D_y - L_S, D_y - L_S + G(L_S + L_P))$;
  Update parameters:
  $D_{x,new} = D_x + (G+1) * (L_S + L_P)$ and
  $D_{y_{new}} = D_{x,new} + r_{ne,G+1}$
end

Algorithm 1: sliding-window pivot-based algorithm for variable-length deduplication of file Y, which is an edited version of file X. Parameters L_S, L_P, n and n_y were introduced in Section III.

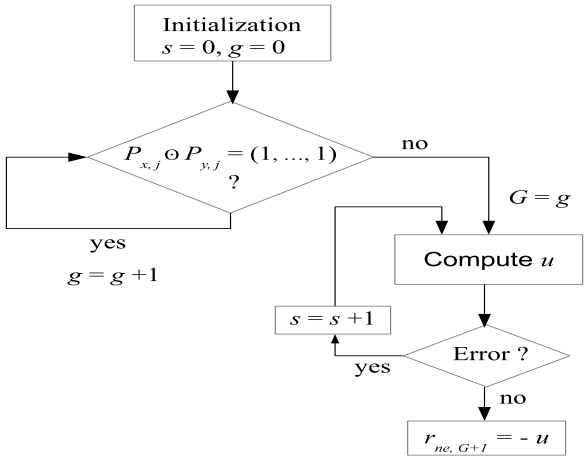

Fig. 3: Principle of the CPM-SW module.

C. Description of the CPM-SW module

This module is the core component of our deduplication algorithm and its goal is to determine G and $r_{ne,G+1}$, even when there are edit errors in pivots.

Fig. 3 describes the steps of the CPM-SW module. In this figure:

$$P_{x,j} = X(D_x + gL_B + s, D_x + gL_B + L_P + s - 1)$$
$$P_{y,j} = Y(D_y + gL_B + s, D_y + gL_B + L_P + s - 1)$$

where $j = 1, \ldots, k$; $g = 0, \ldots, k$; $L_B = L_P + L_S$; $s = 0, \ldots, n_s$ and n_s denotes the maximum number of slides,

The 'Compute u' block calculates $r_{ne,G+1}$ knowing that:

- if $P_{x,G+1} \odot T(P_{y,G+1}, u) = (\underbrace{1, \ldots, 1}_{L_P - u}, \underbrace{0, \ldots, 0}_{u})$, then $r_{ne,G+1} = -u > 0$
- if $P_{x,G+1} \odot T(P_{y,G+1}, u) = (\underbrace{0, \ldots, 0}_{u}, \underbrace{1, \ldots, 1}_{L_P - u})$, then $r_{ne,G+1} = -u < 0$.

Note that if there is an edit error in the pivot, any of the patterns $(\underbrace{1, \ldots, 1}_{L_P - u}, \underbrace{0, \ldots, 0}_{u})$ or $(\underbrace{0, \ldots, 0}_{u}, \underbrace{1, \ldots, 1}_{L_P - u})$ appears, $r_{ne,G+1}$ cannot be determined and it is not possible to update D_y and continue the deduplication algorithm. This was one of the main drawbacks of the work in [8]. To overcome this issue, we propose to use a *sliding window* technique, where the *window* corresponds to the pivot. To be specific, if there is an edit in the pivot, the CPM-SW module *slides* $P_{x,j}$ and $P_{y,j}$ to the right until there are no edits that affect $P_{x,j}$ or the number of slides reaches n_s.

Example Let us consider the hexadecimal alphabet (i.e., $q = 4$) and $L_P = 5$. Let the beginning of file X be:

$$\underbrace{1, ..., A, B, 3,}_{L_S} \underbrace{4, C, D, 5, E,}_{L_P} \underbrace{6, F, ..., 8, 9}_{L_S} \underbrace{B, 2, C, 3, 1}_{L_P} C, B, A, ...$$

and the corresponding edit pattern:

$$E = \underbrace{0, \ldots, 0, -1, -1}_{L_S} \underbrace{0, \ldots, 0}_{L_P + L_S} \underbrace{0, 0, 0, -1, 0}_{L_P}$$

so that the beginning of file Y is:

$$1, ..., A, 4, C, \underbrace{D, 5, E, 6, F,}_{L_P} \underbrace{..., 8, 9,}_{L_S} \underbrace{B, 2, C, 1, C,}_{L_P} B, A, ...$$

First execution of the CPM-SW module performs the pivot matching with $P_{x,1} = (4, C, D, 5, E)$ and $P_{y,1} = (D, 5, E, 6, F)$. Following the scheme in Fig. 3: $P_{x,1} \odot P_{y,1} \neq (1, ..., 1)$, $G = 0$, the 'Compute u' block is activated and because $P_{x,1} \odot T(P_{y,1}, 2) = (0, 0, 1, 1, 1)$, $u = 2$, 'no error' and $r_{ne,1} = -2$.

Second execution of the CPM-SW module performs the pivot matching with $P_{x,2} = (B, 2, C, 3, 1)$ and $P_{y,2} = (B, 2, C, 1, C)$. Again $P_{x,1} \odot P_{y,1} \neq (1, ..., 1)$, $G = 0$. Because there is an edit in the pivot, $\forall u \in (1, ..., L_P)$, u cannot be computed, thus 'error' and the sliding window technique is activated:

- for $s = 1$, $P_{x,2} = (2, C, 3, 1, C)$ and $P_{y,2} = (2, C, 1, C, B)$, then 'error' and $s = 2$;
- for $s = 2$, $P_{x,2} = (C, 3, 1, C, B)$ and $P_{y,2} = (C, 1, C, B, A)$, then 'error' and $s = 3$;
- for $s = 3$, $P_{x,2} = (3, 1, C, B, A)$ and $P_{y,2} = (1, C, B, A, 4)$, then $u = 1$, 'no error', $r_{ne,2} = -1$, and the SW-PD can continue its search for chunks.

V. Theoretical analysis of the algorithm costs

In order to evaluate the benefits of our sliding-window pivot-based algorithm compared to prior work in [8] and to naive or brute force methods, we derive expressions for the average number of compared symbols in a n-length file for all these techniques. The *cost function* depends on the edit probabilities $\beta = \beta_I + \beta_D$. The brute force methods compare data symbol by symbol from the beginning of the file. In our study we consider file- and fixed-block-level deduplication [6].

The file-level brute force method compares the files sequentially from the beginning and stops as soon as an edit error is detected. The average number of compared symbols is thus:

$$N_F = \sum_{i=1}^{n-1} i\beta(1-\beta)^{i-1} + n[\beta(1-\beta)^{n-1} + (1-\beta)^n]$$

For block-level brute force methods, the principle is the same but once and edit error is detected, the comparison continues at the beginning of the following block and so on, until the last block in the file. In this case, the average number of compared symbols is:

$$N_B = k \sum_{i=1}^{L_B-1} i\beta(1-\beta)^{i-1} + L_B[\beta(1-\beta)^{L_B-1} + (1-\beta)^{L_B}]$$

where k is the number of blocks in the n-length file and L_B is the length of a block. When an edit is detected, the comparison continues at the beginning of the following block.

For the algorithm in [8], the average number of compared symbols is:

$$N_{PD} = L_P.k.(1-\beta)^{L_S} + (L_P)^2.k.[1-(1-\beta)^{L_S}].$$

For the sliding-window pivot-based (SW-PD) algorithm, the average number of compared symbols is:

$$\begin{aligned} N_{SW-PD} =& L_P.k.(1-\beta)^{L_B} + \\ &+ {L_P}^2.k.n_s.(1-\beta)^{L_S}[1-(1-\beta)^{L_P}] + \\ &+ {L_P}^2.k.n_s.[1-(1-\beta)^{L_S}].[1-(1-\beta)^{L_P}] \\ &+ {L_P}^2.k.[1-(1-\beta)^{L_S}](1-\beta)^{L_P} \end{aligned}$$

where n_s is the maximum number of slides and the first term represents the case of without edits in the whole block; the second term represents cases with no edits in the segment and one or more edits in the pivot; the third term, *idem* represents cases with one or more edits in both the segment and the pivot; finally, the forth term, one or more edits in segment and no edits in pivot. Note that $(L_P)^2$ symbols are compared at each execution of the 'Compute u' block.

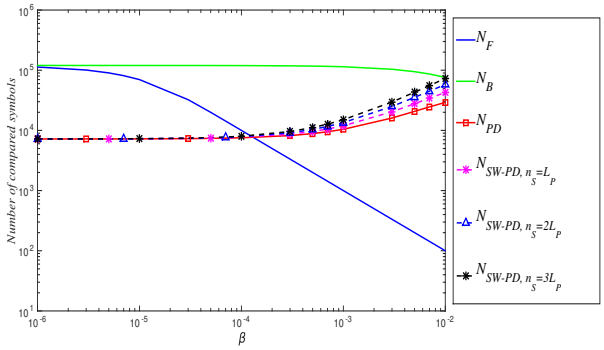

Fig. 4: Number of compared symbols as a function of β; $n = 120000$, $L_S = 94$, $L_P = 6$ (in symbols), $k = 1200$. PD stands for the work in [8].

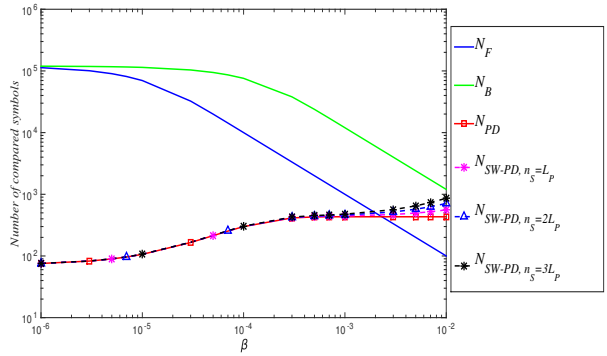

Fig. 5: Number of compared symbols as a function of β; $n = 120000$, $L_S = 9994$, $L_P = 6$ (in symbols), $k = 12$. PD stands for the work in [8].

Figures 4 and 5 compare the costs of the four different algorithms for two different values of k and files of size $n = 1.2 \times 10^5$ symbols. For $\beta < 10^{-4}$ the work in [8] and the new SW-PD reduce costs by 10 (for $k = 1200$) and 1000 (for $k = 12$). Costs for the file-level brute force method drop for higher β, however deduplication ratios become extremely

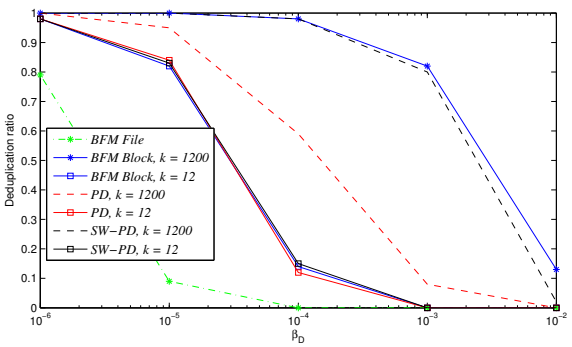

Fig. 6: Data deduplication ratio as a function of β for the four different algorithms. For the PD and SW-PD, $L_P = 6$ and $L_S = 94$ (i.e., $k = 1200$) or $L_S = 9994$ (i.e., $k = 12$).

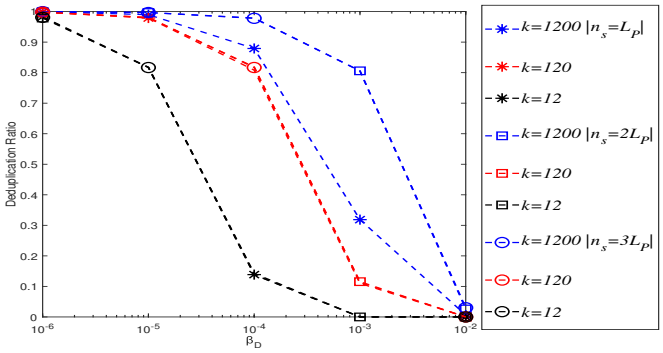

Fig. 7: Effect of n_s on the performance of the SW-PD algorithm. $L_P = 6$, $k = 12, 120, 1200$ and $n_s = L_P, 2L_P, 3L_P$.

poor as simulations will show in Section VI. Also, costs for block-level brute force methods and the new SW-PD tend to equalize when β increases. We thus conclude that the benefits in terms of cost of the new SW-PD are undeniable up to β values around 10^{-3}.

Concerning the cost of the new SW-PD algorithm and the influence of the maximum number of slides (n_s), we observe that it is only for β values over 10^{-3} that the cost slightly increases for $n_s = L_P$ compared to [8], and that higher values of n_s do not increase cost significantly.

VI. Experimental results

We consider the data deduplication ratio to compare the efficiency of our new SW-PD to prior work. This ratio is calculated as the amount of data after deduplication divided by the total capacity of data to back up (i.e., the data that was examined for duplicates). The more redundant data, or the lower edit probabilities, the higher deduplication ratios can be expected. In a more general context than ours, an environment that contains only primarily Windows servers with similar files will lead to much higher deduplication ratios than a multiple-platform environment with different operating systems.

We evaluate performance of the new SW-PD algorithm, the work in [8], and the brute force methods in terms of data deduplication rates through simulation. Our simulations generate 1000 pairs of files X and Y, with $n = 120000$ symbols that are independent identically distributed according to an arbitrary distribution. The size of the alphabet is $2^q = 64$ (i.e., 720 million bits are simulated to obtain each value in a curve). File Y is an edited version of X under the edit channel model described in Section III with $\beta_D = \beta_I$.

Fig. 6 shows the evolution of the data deduplication ratio as a function of β_D for the four considered algorithms. The most interesting result in these curves is the outstanding performance of the new SW-PD technique: for $k = 1200$ and for all $\beta_D < 10^{-3}$, the SW-PD attains the same deduplication rates as the block-level brute force method while reducing costs by a factor of 10 (see Fig. 4). Note that the state-of-the-art [8] presents much lower deduplication rates (curve drops from $\beta_D > 10^{-5}$). For $k = 12$, the block-level BFM, [8] and the new SW-PD show roughly similar deduplication ratios. However (see Fig. 5) both [8] and SW-PD present costs that are 1000 times smaller than the cost of the brute force methods (for $\beta_D < 10^{-4}$).

Figure 7 explores the impact of the maximum number of slides (parameter n_s) on the performance of the SW-PD algorithm. For $k = 1200$, $n_s = 2L_P$ and $n_s = 3L_P$ present significantly better deduplication ratios; which makes sense as $L_S/L_P < 10$ and the *relatively* high number of pivots makes the use of sliding windows more likely and with better performance for $n_s > L_P$. For lower values of k, increasing n_s does not lead to higher deduplication rates. As the cost of the SW-PD is almost similar for $n_s = L_P$ and $2L_P$, we would consider $n_s = 2L_P$ for $k > 100$.

Acknowledgement

The authors would like to thank Professors Lara Dolecek and Tyson Condie (University of California Los Angeles) for discussions on the topic of the paper.

VII. Conclusion

This paper was dedicated to variable-length block-level data deduplication in the context of edit errors. We particularly considered the challenge of introducing an original deduplication algorithm based on pivots that resulted in an efficient low-cost solution. This solution proposes a sliding window technique to outperform the state of the art. The theoretical expressions on the cost of the algorithm show reduction factors from 10 up to 1000 (compared to brute force methods and depending on the values of the k parameter). Simulation results show deduplication ratios that achieve the same values as the equivalent brute force methods.

We can conclude that the new sliding-window pivot deduplication algorithm constitutes a robust data deduplication technique for edit probabilities below 10^{-3} and real time data reduction applications (i.e., inline deduplication), where the rapidity of the protocol is a priority.

Future work will consider the analysis of order topics in data deduplication: structured vs. non-structured data, edit distance between files, size of the alphabet... among others.

REFERENCES

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[2] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, Apr 1984.

[3] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication—large scale study and system design," in *Annual Technical Conference*, Boston, MA, 2012, pp. 285–296.

[4] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep store: an archival storage system architecture," in *21st International Conference on Data Engineering (ICDE'05)*, April 2005, pp. 804–815.

[5] D. Bhagwat and K. E. *et al*, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *IEEE Int. Symp. on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, Sept 2009.

[6] A. Venish and K. S. Sankar, "Study of chunking algorithm in data deduplication," *Proc. of Int. Conf. on Soft Computing Systems, Advances in Intelligent Systems and Computing, Springer India*, 2016.

[7] U. Niesen, "An information-theoretic analysis of deduplication," in *IEEE Int. Symp. on Information Theory (ISIT)*, June 2017, pp. 1738–1742.

[8] L. Conde-Canencia, T. Condie, and L. Dolecek, "Data deduplication with edit errors," in *2018 IEEE Global Communications Conference (GLOBECOM)*, Dec 2018, pp. 1–6.

[9] "Quantum white book: Effectiveness of variable-block vs fixed-block deduplication on data reduction: A technical analysis," www.quantum.com, Tech. Rep., 2015.

[10] F. Sala, C. Schoeny, N. Bitouze, and L. Dolecek, "Synchronizing files from a large number of insertions and deletions," *IEEE Transactions on Communications*, vol. 64, no. 6, pp. 2258–2273, June 2016.