Optimal Approximate Matrix Multiplication over Sliding Windows

Haoming Xian
The Chinese University of Hong Kong
Hong Kong SAR, China
hmxian@se.cuhk.edu.hk

Qintian Guo*
The Hong Kong University of Science and Technology
Hong Kong SAR, China
qtguo@ust.hk

Jun Zhang*
Bitlink Capital Limited
Hong Kong SAR, China
zj@bitlink.capital

Sibo Wang
The Chinese University of Hong Kong
Hong Kong SAR, China
swang@se.cuhk.edu.hk

ABSTRACT

Matrix multiplication is a core operation in numerous applications, yet its exact computation becomes prohibitively expensive as data scales, especially in streaming environments where timeliness is critical. In many real-world scenarios, data arrives continuously, making it essential to focus on recent information via sliding windows. While existing approaches offer approximate solutions, they often suffer from suboptimal space complexities when extended to the sliding-window setting.

In this work, we introduce SO-COD, a novel algorithm for approximate matrix multiplication (AMM) in the sliding-window streaming setting, where only the most recent data is retained for computation. Inspired by frequency estimation over sliding windows, our method tracks significant contributions, referred to as "snapshots", from incoming data and efficiently updates them as the window advances. Given matrices $\boldsymbol{X} \in \mathbb{R}^{d_x \times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{d_y \times n}$ for computing $\boldsymbol{X}\boldsymbol{Y}^T$, we analyze two data settings. In the *normalized* setting, where each column of the input matrices has a unit L_2 norm, SO-COD achieves an optimal space complexity of $O\left(\frac{d_x+d_y}{\epsilon}\right)$. In the *unnormalized* setting, where the square of column norms vary within a bounded range $[1, R]$, we show that the space requirement is $O\left(\frac{d_x+d_y}{\epsilon}\log R\right)$, which matches the theoretical lower bound for an ϵ-approximation guarantee. Time complexity analysis further shows that our SO-COD has comparable update time and superior query time compared to existing methods. Extensive experiments on synthetic and real-world datasets demonstrate that SO-COD effectively balances space cost and approximation error, making it a promising solution for large-scale streaming matrix multiplication.

PVLDB Reference Format:
Haoming Xian, Qintian Guo, Jun Zhang, and Sibo Wang. Optimal Approximate Matrix Multiplication over Sliding Windows. PVLDB, 19(3): 455 - 467, 2025.
doi:10.14778/3778092.3778105

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CUHK-DBGroup/SOCOD.

*Qintian Guo and Jun Zhang are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778105

1 INTRODUCTION

Matrix multiplication is a core operation across machine learning [42], data analysis [12], and graph mining [41]. However, as data volumes grow, exact matrix multiplication becomes increasingly expensive, especially in large-scale scenarios demanding real-time or near-real-time processing. To address these challenges, approximate matrix multiplication (AMM) has emerged as an attractive alternative, offering substantial reductions in computational and storage overhead while preserving high accuracy.

In real-world applications, data often arrives in a streaming fashion [6, 10, 16], as exemplified by social media analytics [39] (where posts or tweets are continuously generated), and user behavior analysis [20] (where search queries or web content are continuously updated). This continuous flow of data has spurred interest in extending the approximate matrix multiplication algorithms to streaming models [35, 45], and a plethora of works [8, 9, 13, 29, 35, 37] have been proposed, gaining prominence due to their robust approximation guarantees and strong empirical performance. However, a key limitation of traditional streaming methods is that they treat all incoming data uniformly, without prioritizing more recent information. In many real-world applications, the most valuable insights stem from the most recent data, necessitating a focus on a sliding window of the most current columns rather than retaining the entire historical dataset. For instance, in social media analysis, matrix $\boldsymbol{X}$ could represent a stream of recent tweets, while matrix $\boldsymbol{Y}$ could represent user interests. The multiplication $\boldsymbol{X}\boldsymbol{Y}^T$ captures the relevance of each tweet to user preferences. Similarly, in user behavior analysis, matrix $\boldsymbol{X}$ might represent recent search queries, and matrix $\boldsymbol{Y}$ could represent the relevance of advertisements or content. By using a sliding window, the approach prioritizes recent data, ensuring that the most current user activities influence the recommendations. Thus, matrix multiplication over a sliding window in a streaming setting is a natural choice, as it aligns with the data evolving nature and emphasizes the timeliness of insights, which is crucial for real-world applications.

To address this challenge, a line of research has focused on developing more efficient sketches that use less space while maintaining the same approximation guarantees. For example, by integrating techniques for sliding-window sampling with sampling-based matrix multiplication (see [4, 13, 14]), we can discard outdated samples and draw new ones from the most recent rows or columns of the input matrix. However, the space cost of the sampling-based method remains relatively high, scaling with $O\left(\frac{1}{\epsilon^2}\log N\right)$, where N is the number of rows and columns involved in the multiplication, and ϵ is the relative error guarantee for approximate matrix multiplication

Table 1: Space and time complexity of approximate matrix multiplication algorithms in the sliding-window setting.

Algorithm	Space Complexity		Query Time Complexity		Update Time Complexity	
	Normalized model	Unnormalized model	Normalized model	Unnormalized model	Normalized model	Unnormalized model
Sampling [14]	$O(\frac{d_x+d_y}{\epsilon^2}\log N)$	$O\left(\frac{d_x+d_y}{\epsilon^2}\log NR\right)$	$O\left(\frac{d_x+d_y}{\epsilon^2}\right)$	$O\left(\frac{d_x+d_y}{\epsilon^2}\right)$	$O\left(\frac{d_x+d_y}{\epsilon^2}\right)$	$O\left(\frac{d_x+d_y}{\epsilon^2}\right)$
DI-COD [44]	$O(\frac{d_x+d_y}{\epsilon}\log^2\frac{1}{\epsilon})$	$O\left(\frac{(d_x+d_y)R}{\epsilon}\log^2\frac{R}{\epsilon}\right)$	$O\left(\frac{d_x+d_y}{\epsilon}\log(\frac{1}{\epsilon})\right)$	$O\left(\frac{d_x+d_y}{\epsilon}R\log(\frac{R}{\epsilon})\right)$	$O\left(\frac{d_x+d_y}{\epsilon}\log(\frac{1}{\epsilon})\right)$	$O\left(\frac{d_x+d_y}{\epsilon}\log(\frac{R}{\epsilon})\right)$
EH-COD [44]	$O(\frac{d_x+d_y}{\epsilon^2}\log\epsilon N)$	$O\left(\frac{d_x+d_y}{\epsilon^2}\log\epsilon NR\right)$	$O\left(\frac{d_x+d_y}{\epsilon^3}\log(\epsilon N)\right)$	$O\left(\frac{d_x+d_y}{\epsilon^3}\log(\epsilon NR)\right)$	$O\left(\frac{d_x+d_y}{\epsilon}\log(\epsilon N)\right)$	$O\left(\frac{d_x+d_y}{\epsilon}\log(\epsilon NR)\right)$
SO-COD (ours)	$O(\frac{d_x+d_y}{\epsilon})$	$O\left(\frac{d_x+d_y}{\epsilon}\log R\right)$	$O\left(\frac{d_x+d_y}{\epsilon}\right)$	$O\left(\frac{d_x+d_y}{\epsilon}+\log\log R\right)$	$O\left(\frac{d_x+d_y}{\epsilon}+\frac{1}{\epsilon^3}\right)$	$O\left(\left(\frac{d_x+d_y}{\epsilon}+\frac{1}{\epsilon^3}\right)\log R\right)$

(refer to Def. 1). Yao et al. [44] present two algorithms, EH-COD and DI-COD, which reduce the dependence on N. However, they still incur a space cost that involves either a quadratic dependence on $\frac{1}{\epsilon}$ or an extra factor of $\log^2\left(\frac{1}{\epsilon}\right)$. Thus, although progress has been made, the space complexity of existing sketching methods remains suboptimal, indicating room for further improvement.

Motivated by these limitations, we introduce *SO-COD* (Space Optimal Co-Occuring Directions), a novel algorithm specifically designed for approximate matrix multiplication in the sliding window setting. By maintaining a compact yet representative summary of the most recent data, SO-COD not only achieves optimal space complexity—matching the best-known theoretical bounds from the streaming literature—but also effectively meets the demands of time-sensitive applications.

At a high level, our approach draws inspiration from the classic sliding window frequency estimation problem [2, 25], where one estimates the frequency of individual elements over the most recent N items by recording snapshot events when an element's count reaches a threshold and expiring outdated snapshots as the window slides. In our work, we extend this snapshot concept to the significantly more challenging task of approximate matrix multiplication (AMM) for the matrices $\boldsymbol{X} \in \mathbb{R}^{d_x\times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{d_y\times n}$. Rather than tracking individual elements, we register the key directions that contribute substantially to the product $\boldsymbol{X}\boldsymbol{Y}^T$ as snapshots, and we expire these snapshots once they fall outside the sliding window. This extension is non-trivial since matrix multiplication involves complex interactions that far exceed the simplicity of frequency estimation. It is crucial to emphasize that, although Yin et al. [46] also utilize a λ-snapshot framework, their work is confined to approximating covariance matrices (i.e., $\boldsymbol{A}\boldsymbol{A}^T$), which represents only a special case of the AMM problem. In contrast, our method is designed to address the full generality of AMM, overcoming challenges that their approach cannot.

We consider two settings. First, when every column of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is normalized (i.e., each column has an L_2 norm of 1), our method requires space $O\left(\frac{d_x+d_y}{\epsilon}\right)$. We then extend our solution to the unnormalized case, where the squared L_2 norm of each column lies within $[1, R]$, and demonstrate that the space cost increases to $O\left(\frac{d_x+d_y}{\epsilon}\log R\right)$. In both cases, we prove that these bounds are tight by matching known space lower bounds for an ϵ-approximation guarantee (see Def. 1). Table 1 summarizes the space complexity of SO-COD compared to existing solutions.

Then we further analyze the amortized time complexity for each algorithm, as presented in Table 1. We observe that the update time complexity of our solution is comparable to that of the baseline methods. In particular, while our update time includes an additive factor of $\frac{1}{\epsilon^3}$, it does not involve any multiplicative logarithmic terms such as $\log(\epsilon NR)$ or $\log\left(\frac{R}{\epsilon}\right)$. In terms of query time complexity, however, our solution outperforms the competitors. Specifically, compared to EH-COD and DI-COD, our query time is lower by a factor of $O(1/\epsilon^2)$ and $O(R\cdot\log(R/\epsilon))$, respectively.

We evaluate the performance of SO-COD with extensive experiments on both synthetic and real-world datasets, showing its accuracy and space efficiency in comparison to existing methods. The results confirm that SO-COD effectively balances the space cost and approximation errors. Moreover, in terms of time efficiency, our SO-COD offers comparable update time and significantly faster query time, while still maintaining optimal space usage. These results make our SO-COD a practical choice for large-scale streaming matrix multiplication over sliding windows.

2 PRELIMINARIES

2.1 Notations and Basic Concepts

In this paper, we use bold uppercase letters to denote matrices (e.g., $\boldsymbol{A}$), bold lowercase letters to denote vectors (e.g., $\boldsymbol{x}$), and lowercase letters to denote scalars (e.g., w).

Given a real matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$, its condensed singular value decomposition (SVD) is given by $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, where $\boldsymbol{U} \in \mathbb{R}^{n\times r}$ and $\boldsymbol{V} \in \mathbb{R}^{m\times r}$ are matrices with orthonormal columns, and $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$ is a diagonal matrix containing the singular values of $\boldsymbol{A}$ arranged in non-increasing order, i.e., $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$, with r denoting the rank of $\boldsymbol{A}$. We define several matrix norms as follows: the Frobenius norm of $\boldsymbol{A}$ is given by $\|\boldsymbol{A}\|_F = \sqrt{\sum_{i,j} A_{i,j}^2} = \sqrt{\sum_{i=1}^r \sigma_i^2}$, the spectral norm is $\|\boldsymbol{A}\|_2 = \sigma_1$, and the nuclear norm is $\|\boldsymbol{A}\|_* = \sum_{i=1}^r \sigma_i$. For a matrix $\boldsymbol{A}$, we denote its i-th column by $\boldsymbol{a}_i$; hence, if $\boldsymbol{A} \in \mathbb{R}^{n\times l}$ and $\boldsymbol{B} \in \mathbb{R}^{m\times l}$, then their product can be expressed as: $\boldsymbol{A}\boldsymbol{B}^T = \sum_{i=1}^l \boldsymbol{a}_i\boldsymbol{b}_i^T$. Finally, $\boldsymbol{I}_n$ denotes the $n\times n$ identity matrix and $\boldsymbol{0}^{n\times m}$ denotes the $n\times m$ zero matrix.

2.2 Problem Definition

The Approximate Matrix Multiplication (AMM) problem has been extensively studied in the literature [13, 15, 35, 45]. Given two matrices $\boldsymbol{X} \in \mathbb{R}^{d_x\times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{d_y\times n}$, the AMM problem aims to obtain two smaller matrices $\boldsymbol{A} \in \mathbb{R}^{d_x\times l}$ and $\boldsymbol{B} \in \mathbb{R}^{d_y\times l}$, with $l \ll n$, so that $\left\|\boldsymbol{X}\boldsymbol{Y}^\top - \boldsymbol{A}\boldsymbol{B}^\top\right\|_2$ is sufficiently small. Next, we formally define the problem of AMM over a Sliding Window.

Definition 1 (AMM over a Sliding Window). *Let $\{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t\geq 1}$ be a sequence of data items (a data stream), where for each time t we have $\boldsymbol{x}_t \in \mathbb{R}^{d_x}$ and $\boldsymbol{y}_t \in \mathbb{R}^{d_y}$. For a fixed window size N and for any*

Algorithm 1: Co-occurring Directions (COD)

Input: $\boldsymbol{X} \in \mathbb{R}^{d_x \times n}$, $\boldsymbol{Y} \in \mathbb{R}^{d_y \times n}$, sketch size l
Output: $\boldsymbol{A} \in \mathbb{R}^{d_x \times l}$ and $\boldsymbol{B} \in \mathbb{R}^{d_y \times l}$

1 Initialize $\boldsymbol{A} \leftarrow \boldsymbol{0}^{d_x \times l}$ and $\boldsymbol{B} \leftarrow \boldsymbol{0}^{d_y \times l}$
2 **for** $i = 1, \ldots, n$ **do**
3 Insert $\boldsymbol{x}_i$ into an empty column of $\boldsymbol{A}$
4 Insert $\boldsymbol{y}_i$ into an empty column of $\boldsymbol{B}$
5 **if** $\boldsymbol{A}$ *or* $\boldsymbol{B}$ *is full* **then**
6 $(\boldsymbol{Q}_x, \boldsymbol{R}_x) \leftarrow \mathrm{QR}(\boldsymbol{A})$
7 $(\boldsymbol{Q}_y, \boldsymbol{R}_y) \leftarrow \mathrm{QR}(\boldsymbol{B})$
8 $[\boldsymbol{U}, \Sigma, \boldsymbol{V}] \leftarrow \mathrm{SVD}(\boldsymbol{R}_x \boldsymbol{R}_y^\top)$
9 $\delta \leftarrow \sigma_{l/2}(\Sigma)$
10 $\hat{\Sigma} \leftarrow \max(\Sigma - \delta \boldsymbol{I}_l, \boldsymbol{0})$
11 Update $\boldsymbol{A} \leftarrow \boldsymbol{Q}_x \boldsymbol{U} \sqrt{\hat{\Sigma}}$
12 Update $\boldsymbol{B} \leftarrow \boldsymbol{Q}_y \boldsymbol{V} \sqrt{\hat{\Sigma}}$
13 **return** $\boldsymbol{A}, \boldsymbol{B}$

time $T \geq N$, define the sliding window matrices

$$\boldsymbol{X}_W = \begin{bmatrix} \boldsymbol{x}_{T-N+1} & \boldsymbol{x}_{T-N+2} & \cdots & \boldsymbol{x}_T \end{bmatrix} \in \mathbb{R}^{d_x \times N},$$

$$\boldsymbol{Y}_W = \begin{bmatrix} \boldsymbol{y}_{T-N+1} & \boldsymbol{y}_{T-N+2} & \cdots & \boldsymbol{y}_T \end{bmatrix} \in \mathbb{R}^{d_y \times N}.$$

A streaming algorithm (or matrix sketch) κ is said to yield an ϵ-approximation for AMM over the sliding window if, at every time $T \geq N$, it outputs matrices $\boldsymbol{A}_W \in \mathbb{R}^{d_x \times m}$ and $\boldsymbol{B}_W \in \mathbb{R}^{d_y \times m}$ (with $m \leq N$, typically $m \ll N$) satisfying

$$\left\| \boldsymbol{X}_W \boldsymbol{Y}_W^\top - \boldsymbol{A}_W \boldsymbol{B}_W^\top \right\|_2 \leq \epsilon \, \|\boldsymbol{X}_W\|_F \, \|\boldsymbol{Y}_W\|_F.$$

That said, the product $\boldsymbol{A}_W \boldsymbol{B}_W^\top$ produced by the sketch κ approximates the true product $\boldsymbol{X}_W \boldsymbol{Y}_W^\top$ with a spectral norm error that is at most an ϵ-fraction of the product of the Frobenius norms of $\boldsymbol{X}_W$ and $\boldsymbol{Y}_W$.

An important application of AMM over sliding windows is *Canonical Correlation Analysis (CCA)* [27]. CCA is a fundamental statistical technique for uncovering linear relationships between two data matrices and has been widely applied in areas such as speech processing [3], natural language processing [38], weather prediction [7], and stock analysis [31]. In the sliding-window setting, CCA computation requires efficient and accurate tracking of the cross-covariance term $X_W Y_W^\top$ over time [45] to enable real-time analysis under memory constraints. Recomputing this term exactly is prohibitively expensive, whereas our sketch provides an approximation with theoretical guarantees. Thus, the solution proposed in this paper is well-suited for sliding window–based CCA.

In [40], it is shown that one may assume without loss of generality that the squared norms of the columns of $\boldsymbol{X}$ and $\boldsymbol{Y}$ are normalized to lie in the intervals $[1, R_x]$ and $[1, R_y]$, respectively, which is a mild assumption. We denote $R = \max(R_x, R_y)$.

Next, we present two key techniques that underpin our solution: the Co-occurring Directions (COD) algorithm [35] and the λ-snapshot method [25]. While each technique was originally developed for a different problem, we show later that their careful and nontrivial integration yields a solution with optimal space to gain ϵ-approximation guarantee.

2.3 Co-Occurring Directions

Co-occurring Directions (COD) [35] is a deterministic streaming algorithm for approximate matrix multiplication. Given matrices $\boldsymbol{X} \in \mathbb{R}^{d_x \times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{d_y \times n}$ whose columns arrive sequentially, COD maintains sketch matrices $\boldsymbol{A} \in \mathbb{R}^{d_x \times l}$ and $\boldsymbol{B} \in \mathbb{R}^{d_y \times l}$ (with $l \ll n$) so that $\boldsymbol{X}\boldsymbol{Y}^\top \approx \boldsymbol{A}\boldsymbol{B}^\top$. Algorithm 1 shows the pseudo-code of COD algorithm. In essence, each incoming column pair $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is inserted into an available slot in the corresponding sketch (Lines 3-4). When a sketch becomes full, a compression step is performed (Lines 5-12): the sketches are first orthogonalized via QR decomposition (Lines 6-7); then, an SVD is applied to the product of the resulting factors (Line 8); finally, the singular values are shrunk by a threshold δ (typically chosen as the $\ell/2$-th singular value) to update the sketches (Lines 9-12). This process effectively discards less significant directions while preserving the dominant correlations, thereby controlling the approximation error.

Theorem 1 ([35]). *The output of Co-occurring Directions (Algorithm 1) provides correlation sketch matrices $(\boldsymbol{A} \in \mathbb{R}^{d_x \times l}, \boldsymbol{B} \in \mathbb{R}^{d_y \times l})$ for $(\boldsymbol{X} \in \mathbb{R}^{d_x \times n}, \boldsymbol{Y} \in \mathbb{R}^{d_y \times n})$, where $l \leq \min(d_x, d_y)$, satisfying:*

$$\left\| \boldsymbol{X}\boldsymbol{Y}^\top - \boldsymbol{A}\boldsymbol{B}^\top \right\|_2 \leq \frac{2\|\boldsymbol{X}\|_F \|\boldsymbol{Y}\|_F}{l}.$$

Algorithm 1 runs in $O(n(d_x + d_y)l)$ time using $O((d_x + d_y)l)$ space.

2.4 λ-Snapshot Method

We explain the key idea of the λ-snapshot method [24, 25] by beginning with a bit stream $f = \{b_1, b_2, b_3, \ldots\}$ where $b_i \in \{0, 1\}$, and the goal is to approximate the number of 1-bits in a sliding window. The λ-snapshot method achieves this by "sampling" every λ-th 1-bit. That is, if we index the 1-bits in order of occurrence, the λ-th, 2λ-th, 3λ-th, etc., are stored. The stream is conceptually divided into blocks of λ consecutive positions (called λ-blocks), and a block is deemed *significant* if it intersects the current sliding window and contains at least one sampled 1-bit. The algorithm maintains a λ-counter that tracks: *(i)* a queue Q holding the indices of significant λ-blocks; *(ii)* a counter ℓ recording the number of 1-bits seen since the last sample; and *(iii)* auxiliary variables for the current block index and the offset within that block. When a new bit arrives, the offset is incremented and any block that no longer falls within the sliding window is removed from Q. If the offset reaches λ, the block index is incremented. Upon encountering a 1-bit, ℓ is incremented; when ℓ reaches λ, that 1-bit is sampled (i.e., ℓ is reset to 0) and the current block index is *registered* and appended to Q. The estimated count of 1-bits in the current window W_p is given by $v(S) = |Q|\lambda + \ell$. If the true count is m, then it is guaranteed that: $m \leq v(S) \leq m + 2\lambda$, so that the error is bounded by 2λ. As the window slides, blocks falling entirely out of range are removed from Q, ensuring that the estimate $v(S) = |Q|\lambda + \ell$ remains valid with an error of at most 2λ. It is easy to verify that the number of λ-blocks is no larger than $\frac{N}{\lambda}$. By setting $\lambda = \frac{\epsilon N}{2}$, the λ-snapshot method can achieve error of at most ϵN using $O(\frac{1}{\epsilon})$ space.

An example of how the λ-snapshot method works to count the number of 1-bits in a sliding window is provided as follows.

Example 1. *Consider a bit stream $\{1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0\}$ with 11 elements and let $\lambda = 3$. The 1-bits occur at positions 1, 3, 4, 6, 7, 8,*

Algorithm 2: SO-COD: Initialize($d_x, d_y, \epsilon, \theta$)

Input: d_x, d_y: dimensions of the input spaces for $\boldsymbol{X}$ and $\boldsymbol{Y}$; ϵ: the approximation ratio; θ: the register threshold for snapshot;

1 $l \leftarrow \min\{\lceil 1/\epsilon \rceil, d_x, d_y\}$
2 $\hat{\boldsymbol{A}} \leftarrow \boldsymbol{0}^{d_x \times l}, \quad \hat{\boldsymbol{B}} \leftarrow \boldsymbol{0}^{d_y \times l}$.
3 $S \leftarrow$ an empty queue.
4 $\hat{\boldsymbol{A}}' \leftarrow \boldsymbol{0}^{d_x \times l}, \quad \hat{\boldsymbol{B}}' \leftarrow \boldsymbol{0}^{d_y \times l}$.
5 $S' \leftarrow$ an empty queue.

and 10; hence, the 3rd and 6th 1-bits (positions 4 and 8) are sampled. We partition the stream into λ-blocks of length 3:

Block 1: $[1, 3]$*, Block 2:* $[4, 6]$*, Block 3:* $[7, 9]$*, Block 4:* $[10, 12]$.

Note that Block 4 is still incomplete. At time $T = 11$, assume a sliding window of size $N = 5$: $W_{11} = [7, 11]$. Within W_{11}, the 1-bits occur at positions 7, 8, and 10. Block 2 lies entirely outside W_{11} and is dropped, and although Block 4 overlaps W_{11}, it contains no sampled bit. Thus, only Block 3 (spanning 7–9 and containing the sampled 1-bit at position 8) remains registered in the queue Q. Moreover, the 1-bit at position 10, arriving after the last sampled bit, yields a leftover count $\ell = 1$. The λ-snapshot estimate is hence $v(S) = |Q| \cdot \lambda + \ell = 1 \cdot 3 + 1 = 4$. Since the true count of 1-bits in W_{11} is 3, the error is $4 - 3 = 1$.

To support frequency estimation over sliding window, a naive approach is to apply the λ-snapshot method to every distinct element in the stream. For any given element e, the stream can be represented as a bit stream, where each new bit is set to 1 if the stream element equals e and 0 otherwise. However, maintaining a separate λ-snapshot structure for each element would lead to unbounded space usage. Lee et al. [25] show that by maintaining only $O(1/\epsilon)$ such λ-snapshot structures and combine with the well known frequency estimation algorithm MG-sketch [33], an ϵ-approximation for the frequency of each element can be achieved while using just $O(1/\epsilon)$ space. Although there are $O(1/\epsilon)$ λ-snapshot structures, they collectively track a stream containing at most N ones, ensuring that the overall space cost remains bounded. For each element e, let $f(e)$ be its true frequency and $\hat{f}(e)$ be the estimated frequency derived using the λ-snapshot method, then $f(e) - \hat{f}(e) \leq \epsilon N$.

3 OUR SOLUTION

In this section, we introduce our method, SO-COD (Space Optimal Co-Occurring Directions). As we mentioned earlier, our algorithm is inspired by the λ-snapshot method for ϵ-approximation frequency estimation over sliding windows [25]. The key idea is simple yet powerful: as new data arrives, we continuously monitor the dominant (i.e., most significant) co-occurring direction. When the product of the norms of the corresponding vectors exceeds a preset threshold θ, we "register" a snapshot capturing that direction and remove it from the sketch. The manner in which SO-COD registers snapshots bears a strong resemblance to the block generation logic of the λ-snapshot. In the λ-snapshot method, a λ-block is generated when the number of 1-bits reaches the threshold λ. By analogy, we register a snapshot whenever a dominant co-occurring direction is detected—that is, when the product of the norms of two vectors exceeds θ. The snapshot vectors in our algorithm play the same

Algorithm 3: SO-COD: Update($\boldsymbol{x}_i, \boldsymbol{y}_i$)

Input: $\boldsymbol{x}_i$: the column vector of $\boldsymbol{X}$ arriving at timestamp i; $\boldsymbol{y}_i$: the column vector of $\boldsymbol{Y}$ arriving at timestamp i

1 **if** $i \pmod N = 1$ **then**
2 $\quad (\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}) \leftarrow (\hat{\boldsymbol{A}}', \hat{\boldsymbol{B}}')$.
3 $\quad (\hat{\boldsymbol{A}}', \hat{\boldsymbol{B}}') \leftarrow (\boldsymbol{0}^{d_x \times l}, \boldsymbol{0}^{d_y \times l})$.
4 $\quad S \leftarrow S'$ and $S' \leftarrow$ an empty queue.
5 **while** $S[0].t + N \leq i$ **do**
6 $\quad S$.POPLEFT() `// Remove expired snapshots`
7 $\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}} \leftarrow \text{COD}(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}, \boldsymbol{x}_i, \boldsymbol{y}_i)$.
8 **while** $\|\hat{\boldsymbol{a}}_1\|_2 \cdot \|\hat{\boldsymbol{b}}_1\|_2 \geq \theta$ **do**
9 $\quad$ Append a snapshot to S: record $(\boldsymbol{v} = \hat{\boldsymbol{a}}_1, \boldsymbol{u} = \hat{\boldsymbol{b}}_1, s = S[-1].t, t = i)$.
10 $\quad$ Remove the first row from both $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{B}}$.
11 $\quad \hat{\boldsymbol{a}}_2, \hat{\boldsymbol{b}}_2$ becomes the new $\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{b}}_1$
12 Repeat Lines 7-11 for the auxiliary sketch

core role as the λ-block in the λ-snapshot method: both capture salient information from the stream, preserving only substantial updates while retaining accuracy and space efficiency.

In this section, we focus on the *normalized sliding window* case, where every incoming vector is unit-norm ($\|\boldsymbol{x}_i\| = \|\boldsymbol{y}_i\| = 1$). In this case, we have $\|\boldsymbol{A}_W\|_F^2 = \|\boldsymbol{B}_W\|_F^2 = N$ (with N being the window length). In Section 4, we extend the method to handle the general case ($R \geq 1$), referred to as the unnormalized sliding window.

3.1 Algorithm Description

Sketch Structure. To handle the sliding window efficiently, our proposed SO-COD maintains two separate COD sketches:

- A *primary sketch* ($\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}$) along with its snapshot queue S, which summarizes data within the current sliding window.
- An *auxiliary sketch* ($\hat{\boldsymbol{A}}', \hat{\boldsymbol{B}}'$) with its snapshot queue S', which accumulates the most recent updates.

Each element in the queue S (or S') contains four components: (u, v), two snapshot vectors representing corresponding significant directions; t, the timestamp when this snapshot is registered; and s, the timestamp of the last registered snapshot.

Algorithm 2 provides the pseudo-code for initializing the two sketches. First, it calculates the capacity of the COD sketches $l = \min\{\lceil 1/\epsilon \rceil, d_x, d_y\}$ at Line 1. For normalized model, given the approximation ratio ϵ and the window size N, we set the register threshold $\theta = \epsilon N$. When it comes to the unnormalized model, the setting of θ is different, where more details are introduced in Section 4. After that, it sets both the primary and auxiliary sketches to zero matrices and initializes the queue as empty (Lines 2-5).

Update Algorithm. When a new pair ($\boldsymbol{x}_i, \boldsymbol{y}_i$) arrives, we update both sketches using the COD routine. The update procedure, detailed in Algorithm 3, involves three main steps:

(1) Window refresh (Lines 1-4): Every N timestamps, the auxiliary sketch and its snapshots replaces that of the primary sketch to mitigate the influence of outdated data outside the sliding window that has accumulated in the sketch. Without this swap, the COD sketch would incorporate all data from the beginning of the stream

and thus be affected by the entire history, rather than by at most twice the window size. See the theoretical analysis for details.

(2) Snapshot expiration (Lines 5-6): It removes any snapshots from both the primary queue S and the auxiliary queue S' that have fallen outside the window. Actually, the auxiliary sketch never sees more than N pairs. Whenever it accumulates N pairs of vectors, it replaces the primary sketch, after which a new, empty auxiliary sketch is initialized. Consequently, the auxiliary sketch never produces snapshots outside the window, and expiration needs to be handled only on the primary sketch.

(3) Registering dominant directions (Lines 7-11): After updating a sketch, we check if the product of the norms of its first (i.e., most significant) column vectors exceeds θ. When $\|\hat{\boldsymbol{a}}_1\|_2 \cdot \|\hat{\boldsymbol{b}}_1\|_2 \geq \theta$, a snapshot capturing these vectors (and the current timestamp information) is appended to the appropriate queue, and the corresponding column is removed from the sketch. This is the core mechanism of our method: only when a co-occurring direction is sufficiently significant do we register it.

(4) Updating the auxiliary sketch (Line 12): After updating the pair $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ to the primary sketch, we repeat Lines 7–11 on the auxiliary sketch. This ensures that, when the swap between $\boldsymbol{S}$ and $\boldsymbol{S'}$ occurs, the auxiliary sketch has processed exactly N vector pairs.

Complexity Analysis. Next, we analyze the running cost for SO-COD. For each incoming pair $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, we need to feed them to the COD algorithm (Line 7 of Alg. 3). It needs to always perform Lines 6-8 in Alg. 1 every time to obtain the singular values of $\boldsymbol{R}_X \boldsymbol{R}_Y^T$. Performing QR decomposition on $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{B}}$ takes $O(d_x l^2)$ and $O(d_y l^2)$ time, respectively. And then computing $\boldsymbol{R}_X \boldsymbol{R}_Y^T$ and performing SVD on matrix $\boldsymbol{R}_X \boldsymbol{R}_Y^T$ cost $O(l^3)$ time. Thereby, the computational bottleneck of the algorithm is the QR decomposition performed within each COD update on the $d_x \times l$ and $d_y \times l$ matrices. Hence, each update runs in $O((d_x + d_y) l^2)$ time.

For space cost, by setting the sketch size $l = min(\lceil \frac{1}{\epsilon} \rceil, d_x, d_y)$, the memory cost of the two COD sketches is bounded by $O(\frac{d_x + d_y}{\epsilon})$. Additionally, setting the register threshold $\theta = \epsilon N$ results in the number of snapshots being $O(\frac{1}{\epsilon})$. In total, the space cost for the whole DS-COD sketch is $O(\frac{d_x + d_y}{\epsilon})$. For the choice of l and θ, we will include more discussions in our later theoretical analysis.

3.2 Fast Update Algorithm

A major computational bottleneck in the basic SO-COD update (see Alg. 3) is the repeated full matrix decomposition, which costs $O((d_x + d_y) l^2)$ per update. In fact, the original COD method defers a full decomposition until the sketch accumulates l columns, resulting in an amortized cost of $O((d_x + d_y) l)$ per column. However, in our sliding window scenario, we must process every column to ensure that no significant direction is missed.

We observe that the COD analysis only requires the maintained matrices $\boldsymbol{Q}_X$ and $\boldsymbol{Q}_Y$ to be orthonormal; the specific upper triangular structure of $\boldsymbol{R}_X$ and $\boldsymbol{R}_Y$ is not essential. Motivated by this observation, we aim to incrementally maintain the decomposition by ensuring that $\boldsymbol{Q}_X$ and $\boldsymbol{Q}_Y$ remain orthonormal without recomputing a full decomposition from scratch each time. This leads to our algorithm *FastUpdate* (Alg. 5). It successfully avoids unnecessary full decompositions, by updating the existing orthonormal basis incrementally, in a manner akin to the Gram–Schmidt process.

Algorithm 4: IncDec: Incremental Decomposition

Input: $\boldsymbol{Q} \in \mathbb{R}^{d \times l}$, $\boldsymbol{R} \in \mathbb{R}^{l \times l}$, and $\boldsymbol{x} \in \mathbb{R}^d$
Output: $\boldsymbol{Q'} \in \mathbb{R}^{d \times (l+1)}$, $\boldsymbol{R'} \in \mathbb{R}^{(l+1) \times (l+1)}$

1 $\boldsymbol{x'} \leftarrow \boldsymbol{x} - \sum_{i=1}^{l} \langle \boldsymbol{q}_i, \boldsymbol{x} \rangle \boldsymbol{q}_i$.
// It is easy to verify that vector $\boldsymbol{x'}/\|\boldsymbol{x'}\|$ is a unit vector orthogonal to every column vector of $\boldsymbol{Q}$
2 $\boldsymbol{v} \leftarrow \left(\langle \boldsymbol{q}_1, \boldsymbol{x} \rangle, \langle \boldsymbol{q}_2, \boldsymbol{x} \rangle, \cdots, \langle \boldsymbol{q}_l, \boldsymbol{x} \rangle \right)^T$.
3 $\boldsymbol{Q'} \leftarrow \begin{bmatrix} \boldsymbol{Q} & \boldsymbol{x'}/\|\boldsymbol{x'}\|_2 \end{bmatrix}$.
4 $\boldsymbol{R'} \leftarrow \begin{bmatrix} \boldsymbol{R} & \boldsymbol{v} \\ \boldsymbol{0} & \|\boldsymbol{x'}\|_2 \end{bmatrix}$.
5 **return** $\boldsymbol{Q'}, \boldsymbol{R'}$.

To design this strategy, we additionally maintain four matrices: $\boldsymbol{Q}_X$, $\boldsymbol{R}_X$ and $\boldsymbol{Q}_Y$, $\boldsymbol{R}_Y$, which capture the decomposition of the primary sketches $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{B}}$, respectively.

Incremental Decomposition. When a new vector $\boldsymbol{x}$ (or $\boldsymbol{y}$) arrives, we update the current decomposition using the procedure, described in Alg. 4. Briefly, we compute the residual: $\boldsymbol{x'} = \boldsymbol{x} - \sum_{i=1}^{l} \langle \boldsymbol{q}_i, \boldsymbol{x} \rangle \boldsymbol{q}_i$, where $\{\boldsymbol{q}_i\}_{i=1}^{l}$ are the columns of the current basis $\boldsymbol{Q}$ (see Alg. 4, Line 1). Normalizing $\boldsymbol{x'}$ produces a new unit vector orthogonal to the existing basis, and the inner products $\langle \boldsymbol{q}_i, \boldsymbol{x} \rangle$ form a vector $\boldsymbol{v}$ that updates the upper triangular matrix $\boldsymbol{R}$ (Lines 2–3 of Alg. 4). This incremental update requires only $O(dl)$ time per update, which is dramatically lower than a full decomposition.

FastUpdate Algorithm. With the incremental decomposition procedure in place, we now present the complete details of our FastUpdate Algorithm in Alg. 5. The first two steps, window refresh and snapshot expiration, are identical to those in Alg. 3. For the update step, we first invoke Alg. 4 to incrementally update and obtain the decompositions $\boldsymbol{Q}_X$, $\boldsymbol{R}_X$, $\boldsymbol{Q}_Y$, $\boldsymbol{R}_Y$ (Lines 9–10 in Alg. 5).

Once these updated decompositions are available, we compute the product $\boldsymbol{R}_X \boldsymbol{R}_Y^T$ and perform a singular value decomposition (SVD) on it (which takes $O(l^3)$ time, as shown in Line 11). The SVD yields singular values $\{\sigma_i\}$ and the associated singular vectors. We then check whether any singular value σ_i meets or exceeds the snapshot threshold θ. For each $\sigma_i \geq \theta$, a snapshot is generated immediately by computing: $\hat{\boldsymbol{a}}_i = \boldsymbol{Q}_X \boldsymbol{u}_i \sqrt{\sigma_i}$ and $\hat{\boldsymbol{b}}_i = \boldsymbol{Q}_Y \boldsymbol{v}_i \sqrt{\sigma_i}$, where $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ are the singular vectors corresponding to σ_i, and it holds that $\|\hat{\boldsymbol{a}}_i\|_2 \, \|\hat{\boldsymbol{b}}_i\|_2 = \sigma_i$. This snapshot computation requires $O((d_x + d_y) l)$ time. Note that our snapshot registration process differs from that in Alg. 1. Here, we must update $\hat{\boldsymbol{A}}$, $\hat{\boldsymbol{B}}$, $\boldsymbol{R}_X$, and $\boldsymbol{R}_Y$ so that the invariant $\hat{\boldsymbol{A}} = \boldsymbol{Q}_X \boldsymbol{R}_X$ and $\hat{\boldsymbol{B}} = \boldsymbol{Q}_Y \boldsymbol{R}_Y$ remains preserved after the update. Although this update strategy is different, we will show that it still produces the correct result. Finally, as detailed in Lines 15–19 of Alg. 5, the snapshot vectors are removed from $\hat{\boldsymbol{A}}$, $\hat{\boldsymbol{B}}$, $\boldsymbol{R}_X$, and $\boldsymbol{R}_Y$ using our incremental decomposition results; the correctness is supported by Lemma 1. Finally, we use the same update strategy for the auxiliary sketch (Alg. 5 Line 20).

By replacing full decompositions with an efficient incremental update strategy, our FastUpdate algorithm achieves an amortized

Algorithm 5: SO-COD:FastUpdate($\boldsymbol{x}_i, \boldsymbol{y}_i$)

Input: $\boldsymbol{x}_i$: the column vector of $\boldsymbol{X}$ arriving at timestamp i;
$\boldsymbol{y}_i$: the column vector of $\boldsymbol{Y}$ arriving at timestamp i

1 **while** $S[0].t + N \leq i$ **do**
2 S.POPLEFT()
3 $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}) \leftarrow (\left[\hat{\boldsymbol{A}}, \boldsymbol{x}_i\right], \left[\hat{\boldsymbol{B}}, \boldsymbol{y}_i\right])$
4 **if** *columns of* $\hat{\boldsymbol{A}} \geq l$ **then**
5 $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}) \leftarrow COD(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}})$
6 $(\boldsymbol{Q}_X, \boldsymbol{R}_X) \leftarrow QR(\hat{\boldsymbol{A}})$
7 $(\boldsymbol{Q}_Y, \boldsymbol{R}_Y) \leftarrow QR(\hat{\boldsymbol{B}})$
8 **else**
9 $(\boldsymbol{Q}_X, \boldsymbol{R}_X) \leftarrow IncDec(\boldsymbol{Q}_X, \boldsymbol{R}_X, \boldsymbol{x}_i)$
10 $(\boldsymbol{Q}_Y, \boldsymbol{R}_Y) \leftarrow IncDec(\boldsymbol{Q}_Y, \boldsymbol{R}_Y, \boldsymbol{y}_i)$
11 $(\boldsymbol{U}, \Sigma, \boldsymbol{V}) \leftarrow SVD(\boldsymbol{R}_X \boldsymbol{R}_Y^T)$
12 **for** $i = 1, \cdots, l$ **do**
13 **if** $\sigma_i \geq \theta$ **then**
14 S append snapshot ($\boldsymbol{v} = \boldsymbol{Q}_X \boldsymbol{u}_i \sqrt{\sigma_i}$, $\boldsymbol{u} = \boldsymbol{Q}_Y \boldsymbol{v}_i \sqrt{\sigma_i}$, $s = S[-1].t$, $t = i$)
15 $\hat{\boldsymbol{A}}' \leftarrow \hat{\boldsymbol{A}} - \boldsymbol{Q}_X \boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$
16 $\hat{\boldsymbol{B}}' \leftarrow \hat{\boldsymbol{B}} - \boldsymbol{Q}_Y \boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{R}_Y$
17 $\boldsymbol{R}'_X \leftarrow \boldsymbol{R}_X - \boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$
18 $\boldsymbol{R}'_Y \leftarrow \boldsymbol{R}_Y - \boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{R}_Y$
19 $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}, \boldsymbol{R}_X, \boldsymbol{R}_Y) \leftarrow (\hat{\boldsymbol{A}}', \hat{\boldsymbol{B}}', \boldsymbol{R}'_X, \boldsymbol{R}'_Y)$
20 Repeat the above process to the auxiliary sketch

update cost of $O((d_x + d_y)l + l^3)$ per column. This novel improvement retains key properties required for COD while substantially reducing computational overhead, making our approach particularly well-suited for high-dimensional streaming data.

Lemma 1. *if* $\hat{\boldsymbol{A}} = \boldsymbol{Q}_X \boldsymbol{R}_X, \hat{\boldsymbol{B}} = \boldsymbol{Q}_Y \boldsymbol{R}_Y, (\boldsymbol{U}, \Sigma, \boldsymbol{V}) = SVD(\boldsymbol{R}_X \boldsymbol{R}_Y^T)$,$\hat{a}_i = \boldsymbol{Q}_X \boldsymbol{u}_i \sqrt{\sigma_i}$,$\hat{b}_i = \boldsymbol{Q}_Y \boldsymbol{v}_i \sqrt{\sigma_i}$, $\hat{\boldsymbol{A}}' = \hat{\boldsymbol{A}} - \boldsymbol{Q}_X \boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$, $\hat{\boldsymbol{B}}' = \hat{\boldsymbol{B}} - \boldsymbol{Q}_Y \boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{R}_Y$, $\boldsymbol{R}'_X = \boldsymbol{R}_X - \boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$ *and* $\boldsymbol{R}'_Y = \boldsymbol{R}_Y - \boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{R}_Y$, *then* $(\boldsymbol{Q}_X, \boldsymbol{R}'_X)$ *is an orthogonal decomposition of* $\hat{\boldsymbol{A}}'$, $(\boldsymbol{Q}_Y, \boldsymbol{R}'_Y)$ *is the orthogonal decomposition of* $\hat{\boldsymbol{B}}'$ *and* $(\hat{\boldsymbol{A}}', \hat{\boldsymbol{B}}')$ *is the same as removing* $(\hat{a}_i, \hat{b}_i)$ *from* $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}})$, *that is,* $\hat{\boldsymbol{A}}' \hat{\boldsymbol{B}}'^T = \hat{\boldsymbol{A}} \hat{\boldsymbol{B}}^T - \hat{a}_i \hat{b}_i^T$.

Query algorithm. The query procedure for SO-COD operate through a direct aggregation of the COD sketch matrices and snapshots from the primary sketch. Specifically, the current sketches $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}})$ are horizontally concatenated with matrices $(\hat{\boldsymbol{C}}, \hat{\boldsymbol{D}})$ stacked by the non-expiring snapshots, constructing an augmented representation that preserves historical data integrity within the window while incorporating current COD sketches.

Complexity Analysis. To compute $\boldsymbol{Q}_X \boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$, we should first compute $\boldsymbol{Q}_X \boldsymbol{u}_i$ in $O(d_x l)$ time, $\boldsymbol{u}_i^T \boldsymbol{R}_X$ in $O(l^2)$ time and then multiply $\boldsymbol{Q}_X \boldsymbol{u}_i$ by $\boldsymbol{u}_i^T \boldsymbol{R}_X$ in $O(d_x l)$ time to obtain $\boldsymbol{Q}_X \boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$. Similarly, we can compute $\boldsymbol{Q}_Y \boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{R}_Y$ in $O(d_y l)$ time. For $\boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$, we can first calculate $\boldsymbol{u}_i^T \boldsymbol{R}_X$ in $O(l^2)$ time and then multiply $\boldsymbol{u}_i$ by $\boldsymbol{u}_i^T \boldsymbol{R}_X$ to attain $\boldsymbol{u}_i \boldsymbol{u}_i^T \boldsymbol{R}_X$ in $O(l^2)$ time. We can also compute $\boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{R}_Y$ in $O(l^2)$ time. As a result, extracting a snapshot from $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}})$ takes $O((d_x + d_y)l)$ time. However, it is noteworthy that after executing

Algorithm 6: SO-COD:Query()

Output: $\boldsymbol{A}_W$ and $\boldsymbol{B}_W$
1 $\boldsymbol{A}_W = \left[\hat{\boldsymbol{A}}, \boldsymbol{C}\right]$, where $\boldsymbol{C}$ is stacked $s_i.\boldsymbol{u}$ for all $s_i \in S$
2 $\boldsymbol{B}_W = \left[\hat{\boldsymbol{B}}, \boldsymbol{D}\right]$, where $\boldsymbol{D}$ is stacked $s_i.\boldsymbol{v}$ for all $s_i \in S$
3 **return** $\boldsymbol{A}_W, \boldsymbol{B}_W$

Lines 17 and 18, $\boldsymbol{R}_X$ and $\boldsymbol{R}_Y$ are not necessarily upper triangular matrices. What can be guaranteed is that $(\boldsymbol{Q}_X, \boldsymbol{R}'_X)$ and $(\boldsymbol{Q}_Y, \boldsymbol{R}'_Y)$ are the orthogonal column decomposition of $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{B}}$ respectively. However, as we mentioned earlier, a closer examination of the COD algorithm's proof in [35] reveals that the upper triangular structure of $\boldsymbol{R}_X$ and $\boldsymbol{R}_Y$ is nonessential to the algorithm's validity. The algorithm's correctness hinges solely on $\boldsymbol{Q}_X$ and $\boldsymbol{Q}_Y$ being column orthogonal. Notably, our proposed Algorithm 4 is designed to handle decompositions with non-triangular $\boldsymbol{R}_X$ and $\boldsymbol{R}_Y$. It remains valid for the pairs $(\boldsymbol{Q}_X, \boldsymbol{R}'_X)$ and $(\boldsymbol{Q}_Y, \boldsymbol{R}'_Y)$ as long as $\boldsymbol{Q}_X$ and $\boldsymbol{Q}_Y$ are column orthogonal. Consequently, decompositions $(\boldsymbol{Q}_X, \boldsymbol{R}'_X)$ and $(\boldsymbol{Q}_Y, \boldsymbol{R}'_Y)$ remain valid for the COD algorithm and Algorithm 4, regardless of the upper triangular structure of $\boldsymbol{R}_X$ and $\boldsymbol{R}_Y$.

Suppose that there exist m singular values surpassing the threshold θ in Line 13 from the beginning to current timestamp t, aligned as $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq \sigma_m \geq \theta$, extracting these m snapshots takes $O(m(d_x + d_y)l)$ time in total. Since $m\theta \leq \sum_{i=1}^{m} \sigma_i \leq \left\|\boldsymbol{A}_{1,t} \boldsymbol{B}_{1,t}^T\right\|_* \leq \sum_{i=1}^{t} \left\|\boldsymbol{x}_i \boldsymbol{y}_i^T\right\|_* \leq t$, we have $m \leq t/\theta$. As a result, it takes $O(\frac{t(d_x+d_y)l}{\theta})$ over t timestamps, and the amortized time per column is $O(\frac{(d_x+d_y)l}{\theta})$. With $\theta = \epsilon N = N/l$, this amortized time is $O((d_x + d_y)l^2/N)$. Assuming $N = \Omega(l)$, this amortization for a single update remains $O((d_x + d_y)l + l^3)$, which is a rational assumption. Thus, compared to Alg. 3 costing $O((d_x + d_y)l^2)$ for each update, Alg. 5 only takes amortized $O((d_x + d_y)l + l^3)$ time for each update, which is faster.

Finally, we present the following theorem about the space and time complexity of our SO-COD, as well as its error guarantee.

Theorem 2. *Let* $\{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t \geq 1}$ *be a stream of data with* $\|\boldsymbol{x}_t\| = \|\boldsymbol{y}_t\| = 1$ *for all* t, *and let* $\boldsymbol{X}_W$ *and* $\boldsymbol{Y}_W$ *denote the sliding window matrices as defined in Definition 1. Given a window size* N *and relative error* ϵ, *the SO-COD algorithm outputs matrices* $\boldsymbol{A}_W \in \mathbb{R}^{d_x \times O(\frac{1}{\epsilon})}$ *and* $\boldsymbol{B}_W \in \mathbb{R}^{d_y \times O(\frac{1}{\epsilon})}$ *such that, if the register threshold is set to* $\theta = \epsilon N$ *and* $l = \min\left(\left\lceil \frac{1}{\epsilon} \right\rceil, d_x, d_y\right)$, *then*

$$\left\|\boldsymbol{X}_W \boldsymbol{Y}_W^\top - \boldsymbol{A}_W \boldsymbol{B}_W^\top\right\|_2 \leq 8\epsilon N = 8\epsilon \|\boldsymbol{X}_W\|_F \|\boldsymbol{Y}_W\|_F.$$

Furthermore, the SO-COD sketch uses $O\left(\frac{d_x+d_y}{\epsilon}\right)$ *space and supports each update in* $O((d_x + d_y)l + l^3)$ *time with Alg. 5.*

4 GENERAL UNNORMALIZED MODEL

In this section, we extend SO-COD to handle unnormalized data, where the squared norms of the input vectors satisfy $\|\boldsymbol{x}_i\|_2^2, \|\boldsymbol{y}_i\|_2^2 \in [1, R]$, where R represents the maximum value of squared norms. Following the approach in [25] for mutable sliding window sizes, we construct a multi-layer extension of SO-COD, which we denote by ML-SO-COD (Multi-Layer SO-COD). In this framework, the

Algorithm 7: ML-SO-COD:Initialize(d_x, d_y, ϵ, N, R)

Input: d_x, d_y: The dimension of $\boldsymbol{X}$ and $\boldsymbol{Y}$ respectively; N: the length of sliding window; R: upper bound of squared norms; ϵ: the approximation ratio

1 $L \leftarrow \lceil \log_2 R \rceil$;
2 $M \leftarrow$ an empty list
3 **for** $i = 0, \cdots, L-1$ **do**
4 $\quad M.append(\text{SO-COD.}Initialize(d_x, d_y, \epsilon, 2^i \epsilon N))$

data stream is processed through $L = \lceil \log_2 R \rceil$ layers. Each layer i (for $i = 0, 1, \ldots, L-1$) maintains an independent SO-COD sketch with a distinct register threshold θ_i defined by $\theta_i = 2^i \epsilon N$. That is, in layer i, a snapshot is registered when $\|\hat{\boldsymbol{a}}_i\|_2 \|\hat{\boldsymbol{b}}_i\|_2 \geq 2^i \epsilon N$. By assigning θ values to scale exponentially across levels, we create a dynamic coverage mechanism that spans a wide range of Frobenius norm magnitudes. This exponential scaling ensures that when the Frobenius norm of matrices within sliding windows fluctuates, there will always be at least one level that can still maintain precise approximate matrix multiplication. For each level, we aim to avoid excessive snapshot storage, and we explicitly restrict the number of snapshots stored per level to no more than $\frac{6}{\epsilon}$. We show that even under this space limit, ML-SO-COD still retains the ϵ-approximation guarantee. More details are provided in the theoretical analysis [1]. By constraining the number of snapshots per layer to $O(1/\epsilon)$, the overall space complexity of ML-SO-COD is $O((d_x + d_y)/\epsilon \cdot \log R)$.

Multi-Layer Sketch Structure. Alg. 7 shows the initialization of an ML-SO-COD sketch, thereby defining the multi-layer structure. In Line 1, we compute the number of layers $L = \lceil \log_2 R \rceil$. For each layer $i \in \{0, 1, \ldots, L-1\}$, an independent SO-COD sketch is initialized with the register threshold $\theta_i = 2^i \epsilon N$ and added to the list M. Consequently, in layer i, the SO-COD sketch will generate a snapshot when $\|\hat{\boldsymbol{a}}_i\|_2 \|\hat{\boldsymbol{b}}_i\|_2 \geq 2^i \epsilon N$. To minimize redundancy and retain only the most significant information, we restrict the number of snapshots per layer to $O(1/\epsilon)$. As a result, the total space required by ML-SO-COD is $O((d_x + d_y)/\epsilon \cdot \log R)$.

Update Algorithm. As shown in Alg. 8, each incoming column pair $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is processed across all layers. To ensure that the number of snapshots in each layer remains within the bound of $O(1/\epsilon)$, we explicitly set this bound to $\frac{6}{\epsilon}$. It first checks if the number of snapshots exceeds $\frac{6}{\epsilon}$ (and prunes expired snapshots promptly, as indicated in Lines 2–3). Then, for each layer j, if the product of $\|\boldsymbol{x}_i\|_2$ and $\|\boldsymbol{y}_i\|_2$ exceed the threshold $2^j \epsilon N$, we generate a snapshot $(\boldsymbol{u} = \boldsymbol{x}_i, \boldsymbol{v} = \boldsymbol{y}_i)$ directly, preserving all information of $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ without introducing approximation error for $\boldsymbol{x}_i \boldsymbol{y}_i^T$ to reduce additional calculation (Lines 4–5). Otherwise, the update procedure applies the column pair $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ to update the corresponding SO-COD sketch $M[j]$ (Line 7) via the SO-COD.*FastUpdate* procedure. Due to the direct update mechanism, we can better restrict the times of extracting snapshots from sketch. Specifically, suppose that in layer j, there exist m singular values surpassing the threshold θ in Alg.5 (Line 13) from the beginning to current timestamp t, aligned as $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq \sigma_m \geq \theta$, extracting these m snapshots takes $O(m(d_x + d_y)l)$ time in total. Since $m\theta \leq \sum_{i=1}^{t} \|\boldsymbol{x}_i \boldsymbol{y}_i^T\|_* \cdot \mathbb{I}(\|\boldsymbol{x}_i\|_2 \|\boldsymbol{y}_i\|_2 < \theta) < t\theta$, where

Algorithm 8: ML-SO-COD:Update($\boldsymbol{x}_i, \boldsymbol{y}_i$)

Input: $\boldsymbol{x}_i$: the column vector of $\boldsymbol{X}$ arriving at timestamp i; $\boldsymbol{y}_i$: the column vector of $\boldsymbol{Y}$ arriving at timestamp i

1 **for** $j = 0, \cdots, L-1$ **do**
2 $\quad$ **while** $len(M[j].S) > \frac{6}{\epsilon}$ *or* $M[j].S[0].t \leq i - N$ **do**
3 $\quad\quad M[j].S.POPLEFT()$
4 $\quad$ **if** $\|\boldsymbol{x}_i\|_2 \|\boldsymbol{y}_i\|_2 \geq 2^j \epsilon N$ **then**
5 $\quad\quad M[j].S$ append snapshot $(\boldsymbol{u} = \boldsymbol{x}_i, \boldsymbol{v} = \boldsymbol{y}_i, s = M[j].S[-1].t, t = i)$
6 $\quad$ **else**
7 $\quad\quad M[j].FastUpdate(\boldsymbol{x}_i, \boldsymbol{y}_i)$
8 $\quad$ Repeat the above process to the auxiliary sketch

$\mathbb{I}(\|\boldsymbol{x}_i\|_2 \|\boldsymbol{y}_i\|_2 < \theta)$ equals 1 if $\|\boldsymbol{x}_i\|_2 \|\boldsymbol{y}_i\|_2 < \theta$ and otherwise 0. Hence, we have $m \leq t$, which implies amortized time for extracting the snapshots per layer is $O((d_x + d_y)l)$. As the update is executed in each of the L layers, the overall time cost per update is $O(((d_x + d_y)l + l^3) \log R)$. ML-SO-COD also maintains a window refresh for each level, but with a different trigger. A swap is executed at level j when the sum of products of input vector norms $\sum \|\boldsymbol{x}_i\|_2 \|\boldsymbol{y}_i\|_2$ processed in auxiliary sketch, exceeds $2^j N$. For the sake of simplicity, this mechanism is not detailed in Alg. 8.

Query Algorithm. Alg. 9 describes the procedure for forming the sketch corresponding to the sliding window $[t - N + 1, t]$. Because of the per-layer constraint on the number of snapshots, a given layer might not contain enough snapshots to cover the entire window and thereby produce a valid sketch. To address this, we select the lowest layer for which the snapshots fully cover the window while minimizing the approximation error. More precisely, a layer is deemed valid if the last expired snapshot before its earliest non-expired snapshot occurs at time s satisfying $s \leq t - N$. A naive approach would scan all $O(\log R)$ layers, resulting in a time complexity of $O(\log R)$; however, since the snapshot density decreases monotonically with increasing layer index (due to the larger register thresholds), a binary search can be employed to reduce the query time complexity to $O(\log \log R)$.

Complexity Analysis. The following theorem demonstrates the error guarantee, space and time complexity for ML-SO-COD.

Theorem 3. *Let $\{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t \geq 1}$ be a stream of data so that for all t it holds that $\|\boldsymbol{x}_t\|_2^2$, $\|\boldsymbol{y}_t\|_2^2 \in [1, R]$. Let $\boldsymbol{X}_W = [\boldsymbol{x}_{t-N+1}, \ldots, \boldsymbol{x}_t]$ and $\boldsymbol{Y}_W = [\boldsymbol{y}_{t-N+1}, \ldots, \boldsymbol{y}_t]$ denote the sliding window matrices as defined in Def. 1. Given window size N and relative error parameter ϵ, the ML-SO-COD algorithm outputs matrices $\boldsymbol{A}_W \in \mathbb{R}^{d_x \times O(\frac{1}{\epsilon})}$ and $\boldsymbol{B}_W \in \mathbb{R}^{d_y \times O(\frac{1}{\epsilon})}$ such that if the sketch size is set to $l = \min\left(\left\lceil \frac{1}{\epsilon} \right\rceil, d_x, d_y\right)$, then $\left\|\boldsymbol{X}_W \boldsymbol{Y}_W^\top - \boldsymbol{A}_W \boldsymbol{B}_W^\top\right\|_2 \leq 4\epsilon \|\boldsymbol{X}_W\|_F \|\boldsymbol{Y}_W\|_F$. Furthermore, the ML-SO-COD sketch uses $O\left(\frac{d_x + d_y}{\epsilon} \log R\right)$ space and supports each update in $O\left((d_x + d_y)l + l^3) \log R\right)$ time.*

Space Optimality. We derive a lower bound on the space complexity of any deterministic algorithm that solves the AMM problem over a sliding window. As we will see, the space complexity of our

Algorithm 9: ML-SO-COD:Query()

Output: $\boldsymbol{A}_W$ and $\boldsymbol{B}_W$

Find $i = \min_j 1 \le M[j].S[0].s \le t - N$

$\boldsymbol{A}_W = \left[M[i].\hat{\boldsymbol{A}}, \boldsymbol{C}\right]$, where $\boldsymbol{C}$ is stacked $s_j.\boldsymbol{u}, \forall\, s_j \in M[i].S$

$\boldsymbol{B}_W = \left[M[i].\hat{\boldsymbol{B}}, \boldsymbol{D}\right]$, where $\boldsymbol{D}$ is stacked $s_j.\boldsymbol{v}, \forall\, s_j \in M[i].S$

return $\boldsymbol{A}_W, \boldsymbol{B}_W$

ML-SO-COD (see Theorem 3) exactly matches this lower bound, thereby confirming its optimality with respect to space efficiency.

Prior work [18] established a lower bound for matrix sketching, and [28] extended this result to AMM, as stated as follows.

Lemma 2 (Lower Bound for AMM [28]). *Consider any AMM algorithm with input matrices $\boldsymbol{X} \in \mathbb{R}^{d_x \times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{d_y \times n}$, and outputs $\boldsymbol{A} \in \mathbb{R}^{d_x \times m}$ and $\boldsymbol{B} \in \mathbb{R}^{d_y \times m}$ that satisfy $\|\boldsymbol{X}\boldsymbol{Y}^T - \boldsymbol{A}\boldsymbol{B}^T\|_2 \le \frac{1}{m}(\|\boldsymbol{X}\|_F \|\boldsymbol{Y}\|_F)$. Assuming a constant number of bits per word, any such algorithm requires at least $\Omega((d_x + d_y)m)$ bits of space.*

We also have the following lemma.

Lemma 3 ([28]). *For any $\delta > 0, d_x \le d_y$, there exists a set of matrix pairs $\hat{\mathcal{Z}}_l = \{(\hat{\boldsymbol{X}}_1, \hat{\boldsymbol{Y}}_1), \ldots, (\hat{\boldsymbol{X}}_M, \hat{\boldsymbol{Y}}_M)\}$, where $M = 2^{\Omega(l(d_y - l)\log(1/\delta))}$ and each $\hat{\boldsymbol{X}}_i \in \mathbb{R}^{d_x \times l}$ and $\hat{\boldsymbol{Y}}_i \in \mathbb{R}^{d_y \times l}$ satisfies $\hat{\boldsymbol{X}}_i^T \hat{\boldsymbol{X}}_i = I_l$ and $\hat{\boldsymbol{Y}}_i^T \hat{\boldsymbol{Y}}_i = I_l$. Besides, for any $i \ne j$, $\|\hat{\boldsymbol{X}}_i \hat{\boldsymbol{Y}}_i^T - \hat{\boldsymbol{X}}_j \hat{\boldsymbol{Y}}_j^T\| > \sqrt{2\delta}$.*

Combining Lemmas 2 and 3, we obtain the following lower bound for the AMM problem over a sliding window.

Theorem 4 (Lower Bound for AMM over Sliding Window). *Given $\boldsymbol{X}_W \in \mathbb{R}^{(d_x+1)\times N}, \boldsymbol{Y}_W \in \mathbb{R}^{(d_y+1)\times N}$ and $1 \le \|\boldsymbol{x}\|_2^2, \|\boldsymbol{y}\|_2^2 \le R + 1$ for all $x \in \boldsymbol{X}_W$ and $y \in \boldsymbol{Y}_W$, a deterministic algorithm that returns $(\boldsymbol{A}_W, \boldsymbol{B}_W)$ such that $\left\|\boldsymbol{X}_W \boldsymbol{Y}_W^T - \boldsymbol{A}_W \boldsymbol{B}_W^T\right\|_2 \le \frac{\epsilon}{3}\|\boldsymbol{X}_W\|_F \|\boldsymbol{Y}_W\|_F$ where $\epsilon = 1/l$ and $N \ge \frac{1}{2\epsilon}\log\frac{R}{\epsilon}$, must use $\Omega(\frac{d_x+d_y}{\epsilon}\log R)$ space.*

5 RELATED WORK

Streaming AMM. In the literature, a plethora of research works have been studying the AMM problem in the streaming setting, which can be primarily categorized into two types: randomized algorithms, such as sampling [13], random projections [9, 29, 37], and hashing [8], and deterministic algorithms, including FD-AMM [45] and Co-occurring Directions (COD) [28, 35]. By sampling an appropriate number l of columns from X and Y, Drineas et al. [13] show that an approximate matrix product with a Frobenius norm error guarantee can be achieved. Sarlos [37] demonstrates that a similar guarantee can also be obtained using a random projection technique, provided that the projection matrix satisfies the requirements of a Johnson–Lindenstrauss (JL) transform. On the other hand, the deterministic methods typically utilize SVD to construct approximate matrix sketches, often resulting in lower approximation errors. FD-AMM [45] employs the Frequent Directions (FD) algorithm in a deterministic manner, producing two sketching matrices with approximation guarantees. The COD algorithm [35] is designed to retain only the dominant singular values of the correlation matrix, preserving the most important correlations.

Sliding Window Model. When recent data matters most, the *sliding-window* model is standard for streaming queries—sum [11], distinct elements [21], frequency counts and top-k [22, 30, 36], and quantiles [2]. Datar et al. [11] introduced *Exponential Histograms (EH)*: a logarithmic set of timestamped buckets with exponentially increasing sizes that estimate the number of 1s in the last N positions using $O\left(\frac{1}{\epsilon}\log^2 N\right)$ space; they also extended EH to sums of bounded positive integers ($\le R$) with $O\left(\frac{1}{\epsilon}\log N\,(\log N + \log R)\right)$ space. Arasu et al. [2] generalized sliding-window frequency estimation to multi-category streams via block/layer "aging", and adapted *Misra–Gries* as a black box to achieve additive-ϵ accuracy in $O\left(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon}\right)$ space. Subsequent work simplified and tightened space: Lee et al. [25] used a snapshot mechanism for frequent items in $O\left(\frac{1}{\epsilon}\right)$ space, and Ben-Basat et al. extended *Space-Saving* [32] to the sliding-window model with the same $O\left(\frac{1}{\epsilon}\right)$ bound. For distributed settings, Papapetrou et al. [36] designed an efficient distributed sliding-window framework.

Badeau et al. [5] compute the SVD over a sliding window by retaining all rows of the matrix, instead of leveraging sketching algorithms. Wei et al. [40] studied the problem of approximate matrix covariance in sliding windows, which was later improved in terms of space efficiency by [46]. Zhang et al. [47] extended this line of work to distributed sliding-window settings. To our knowledge, the only prior work on approximate matrix multiplication (AMM) in the sliding-window model is by Yao et al. [44], who combine COD with the dyadic-interval (DI) [2] and exponential-histogram (EH) [11] frameworks to obtain DI-COD and EH-COD. While DI and EH frameworks can be extended to sliding-window algorithms (e.g. using Misra-Gries algorithm [33] for frequency estimation), they introduce a multiplicative space overhead [25]. Therefore, DI-COD and EH-COD inherit this overhead, resulting in suboptimal space complexity. In [44], Yao et al. also include a sampling-based baseline that combines priority sampling with a general sliding-window framework [4]; this method requires $O\left(\frac{1}{\epsilon^2}\log R\right)$ space. In this paper, we embed the λ-snapshot method into COD in a non-trivial way, yielding an algorithm for AMM over sliding windows with optimal space complexity, improving over all prior results.

Table 2: Tested Datasets.

Dataset.	Abbr.	d_x	d_y	n	N
Uniform Random	*UR*	2000	1000	10000	4000
Random Noisy	*RN*	2000	1000	10000	4000
Multimodal Data	*MD*	2048	3000	123287	10000
HPMax	*HP*	16384	16384	32768	10000
TWeibo	*TW*	1657	1657	2320895	50000

6 EXPERIMENTS

In this section, we evaluate our proposed algorithms against three different baseline algorithms on both synthetic and real-world datasets. We have made the source code publicly available [1].

6.1 Baseline Algorithms

Sampling method. For the AMM problem, the algorithm samples a small proportion of the matrices. Specifically, each pair of columns (x_i, y_i) is assigned to a priority $\rho = u^{1/(\|\boldsymbol{x}_i\|_2 \|\boldsymbol{y}_i\|_2)}$, where u is

uniformly sampled from the interval $(0, 1)$ [14]. This priority-based sampling strategy ensures that columns with larger norms are more likely to be selected, thereby preserving the most significant contributions to the matrix product. To achieve an ϵ-approximation guarantee, it requires $O(\frac{1}{\epsilon^2})$ independent samples selected based on the highest priorities with replacement [13, 14]. To extend the priority sampling on the sliding window, we use the technique from [4], leading to a space complexity of $O(\frac{d_x+d_y}{\epsilon^2}\log N)$ for the normalized model and $O(\frac{d_x+d_y}{\epsilon^2}\log NR)$ for unnormalized model.

DI-COD. DI-COD applied the Dyadic Interval approach [2] to Co-Occurring Directions, maintaining a hierarchical structure with $L = \log\frac{R}{\epsilon}$ parallel levels, each of which contains a dynamic number of blocks. For i-th level, the window is segmented into at most 2^{L-i+1} block, and each block maintains a COD sketch. The space cost for DI-COD is $O\left(\frac{(d_x+d_y)R}{\epsilon}\log^2\frac{R}{\epsilon}\right)$.

EH-COD. Exponential Histogram Co-occurring Directions (EH-COD) combines the Exponential Histograms technique [11] and incorporates the COD algorithm for efficiently approximating matrix multiplication within the sliding window model. The space cost for EH-COD is $O\left(\frac{d_x+d_y}{\epsilon^2}\log\epsilon NR\right)$.

6.2 Experiments Setup

Datasets. Experiments are conducted on both synthetic and real-world datasets widely used in matrix multiplication [17, 23, 35, 44, 45]. All datasets are *unnormalized*. The details are listed below:

- **Uniform Random [44, 45].** We generate two random matrices: one of size 2000×10000 and another of size 1000×10000. The entries of both matrices are drawn uniformly at random from the interval $[0, 1)$. The window size for this dataset is $N = 4000$.
- **Random Noisy [17, 35].** We generate the input matrix $\boldsymbol{X}^T = \boldsymbol{SDU} + \boldsymbol{F}/\zeta \in \mathbb{R}^{n\times d_x}$. Here, the term $\boldsymbol{SDU}$ represents an m-dimensional signal, while $\boldsymbol{F}/\zeta$ is a Gaussian noise matrix, with scalar parameter ζ controlling the noise-to-signal ratio. Specifically, $\boldsymbol{S} \in \mathbb{R}^{n\times m}$ is a random matrix where each entry is drawn from a standard normal distribution. $\boldsymbol{D} \in \mathbb{R}^{m\times m}$ is a diagonal matrix with entries $\boldsymbol{D}_{i,i} = 1-(i-1)/m$, and $\boldsymbol{U} \in \mathbb{R}^{m\times d_x}$ is a random rotation which represents the row space of the signal and satisfies that $\boldsymbol{U}^T\boldsymbol{U} = I_m$. $\boldsymbol{F}$ is again a Gaussian matrix with each entry generated i.i.d. from a normal distribution $N(0, 1)$. Matrix $\boldsymbol{Y}$ is generated in the same manner as $\boldsymbol{X}$. We set $n = 10000$,$d_x = 2000$, $d_y = 1000$, $m = 400$, $\zeta = 100$, and the window size $N = 4000$.
- **Multimodal Data [35].** We study the empirical performance of the algorithms in approximating correlation between images and captions. Following [35], we consider Microsoft COCO dataset [26]. For visual features we use the residual CNN Resnet101 [19] to generate a feature vector of dimension $d_x = 2048$ for each picture. For text we use the Hierarchical Kernel Sentence Embedding [34], resulting in a feature vector of dimensions $d_y = 3000$. We construct the matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ with sizes 2048×123287 and 3000×123287, respectively, where each column represents a feature vector. The window size is set to $N = 10000$.
- **HPMaX [23]:** We also include the dataset HPMaX, which is used to test the performance of heterogenous parallel algorithms for matrix multiplication. In this dataset, both of $\boldsymbol{X}$ and $\boldsymbol{Y}$ have size of 16384×32768. The window size N is 10000.
- **TWeibo [43, 48]:** TWeibo is a large directed attributed graph dataset comprising 2.3 million vertices and 1657 attributes. Using the PANE [43] algorithm, we generated two approximate node-attribute affinity matrices, namely the forward affinity matrix $\boldsymbol{F}$ and the backward affinity matrix $\boldsymbol{B}$ respectively. Both matrices are of size 1657×2320895. The window size is set to $N = 50000$.

Evaluation Metrics. Recall that our SO-COD achieves the optimal space complexity while providing an ϵ-approximation guarantee. Therefore, we design the experiments to explicitly demonstrate the trade-off between space consumption and empirical accuracy across different datasets. Specifically, we tune the parameters of each algorithm and report both the maximum sketch size and the empirical relative correlation error.

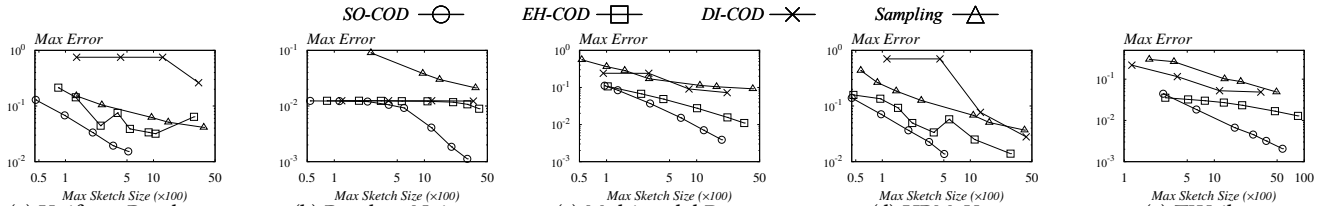
- **Maximum sketch size**. This metric is measured by the *maximum* number of column vectors maintained by a matrix sketching algorithm. The maximum sketch size metric represents the peak space cost of a matrix sketching algorithm.
- **Relative correlation error**. This metric is used to assess the approximation quality of the output matrices. It is defined as $\left\|\boldsymbol{X}_W\boldsymbol{Y}_W^T - \boldsymbol{A}_W\boldsymbol{B}_W^T\right\|_2 / \|\boldsymbol{X}_W\|_F \|\boldsymbol{Y}_W\|_F$, where $\boldsymbol{X}_W$ and $\boldsymbol{X}_W$ denotes the exact matrices covered by the current window, and $\boldsymbol{A}_W$ and $\boldsymbol{B}_W$ denotes sketch matrices for $\boldsymbol{X}_W\boldsymbol{Y}_W^T$.

For the selection of queries, we uniformly choose quantile points at fractions of $i/50$ from each dataset. We then report the maximum sketch size throughout the process, as well as the maximum and average errors of these selected queries.

6.3 Experimental Results

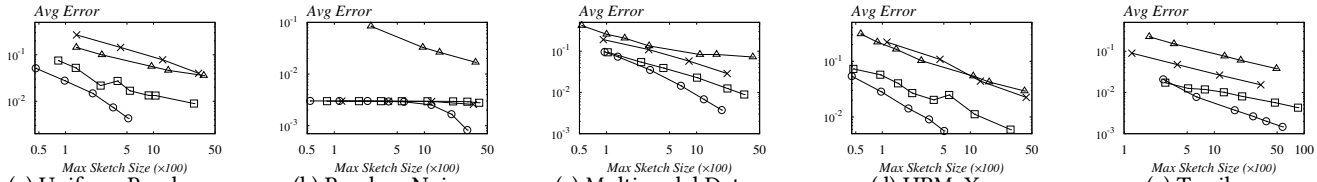
Sketching Quality. We first adjust the error parameter ϵ for each algorithm to analyze the trade-off between space efficiency and empirical accuracy. Generally, when the error parameter ϵ decreases, the maximum sketch size increases. As shown in Figures 1–2, we report the maximum sketch size, as well as the maximum and average relative correlation errors. Both the x-axis and y-axis are displayed on a logarithmic scale to encompass the wide range of values.

First, we observe that the curve representing our solution SO-COD consistently resides in the lower-left corner compared to other baselines, in terms of both maximum and average errors. This implies that for a given space cost (i.e., maximum sketch size), our SO-COD consistently produces matrices with much lower correlation errors. Therefore, our solution demonstrates a superior space-error trade-off, aligning with its optimal space complexity as discussed in Section 4. Second, on certain datasets (e.g., Multimodal Data and HPMax), the second-best algorithm, EH-COD, produces results comparable to our solution when the maximum sketch size is small (i.e., the error parameter ϵ is large). However, the gap between the two curves widens as the maximum sketch size increases (i.e., the error parameter ϵ decreases). This is also aligned with the theoretical result that the suboptimal space complexity $O(\frac{d_x+d_y}{\epsilon^2}\log\epsilon NR)$ of EH-COD is outperformed by our optimal complexity $O(\frac{d_x+d_y}{\epsilon}\log R)$. Finally, we observe that the EH-COD baseline performs better than the DI-COD baseline in almost all cases, which aligns with the observations in [44].



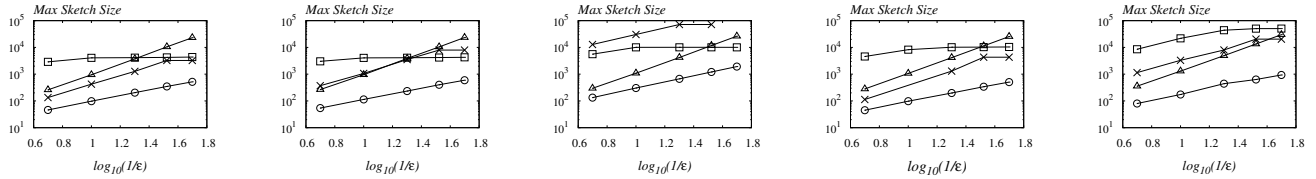
(a) Uniform Random (b) Random Noisy (c) Multimodal Data (d) HPMaX (e) TWeibo

Figure 1: Maximum Sketch Size vs. Maximum Error.



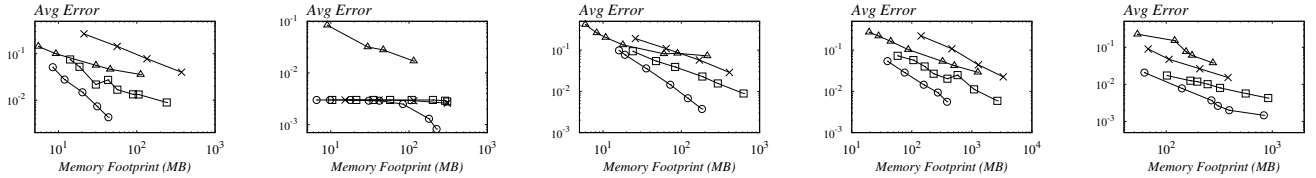
(a) Uniform Random (b) Random Noisy (c) Multimodal Data (d) HPMaX (e) Tweibo

Figure 2: Maximum Sketch Size vs. Average Error.



(a) Uniform Random (b) Random Noisy (c) Multimodal Data (d) HPMaX (e) TWeibo

Figure 3: $\log_{10}(1/\epsilon)$ vs. Maximum Sketch Size ($\log_{10}(\frac{1}{0.25}) \approx 0.6$, and $\log_{10}(\frac{1}{0.016}) \approx 1.8$).



(a) Uniform Random (b) Random Noisy (c) Multimodal Data (d) HPMaX (e) Tweibo

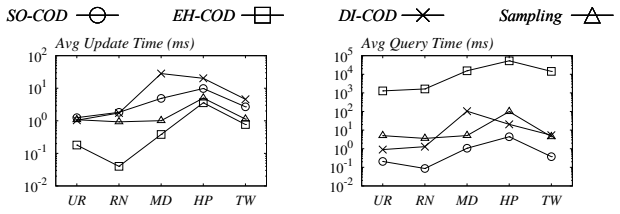
Figure 4: Memory Footprint vs. Average Error.

Then, we examine the impact of the error parameter on the space cost of each algorithm. We vary the parameter ϵ and report the maximum value of sketch size. The results are shown in Figure 3. The curve of our solution SO-COD consistently remains the lowest. This indicates that, for a given error parameter, SO-COD requires the least space, thereby confirming the conclusion of space optimality. One may note that as $\log_{10}(1/\epsilon)$ increases, the space growth of the EH-COD algorithm gradually slows down. This occurs because, as ϵ decreases, the storage capacity of EH-COD increases, and the entire sketch becomes sufficient to store the entire window without significant COD compression operations. Consequently, the maximum sketch size approaches the window size.

Memory Footprint. As the algorithms require auxiliary memory (e.g., for performing SVD), we measure and report the total memory usage of each algorithm to assess their practical space efficiency. As shown in Figure 4, we illustrate the trade-off between average error and memory footprint. The curve for our solution, SO-COD, consistently lies in the bottom-left region of the plot compared to other algorithms, indicating both lower error and smaller memory usage. Moreover, the trend is consistent with the pattern observed in Figure 2. These results confirm that SO-COD delivers superior performance not only in theory but also in practice.

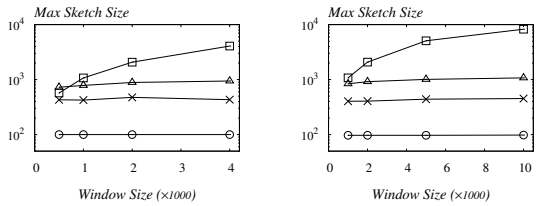
Average Update and Query Time. We then report the average update time and query time to evaluate the efficiency of SO-COD compared to its competitors, with the error parameter fixed at $\epsilon = 0.1$. For measuring query time, we make queries at the selected quantile points (i.e., $\frac{i}{50} \times N$) and report the average query time. The experimental results are presented in Figure 5.

As shown in Figure 5(a), in terms of update time, our SO-COD is comparable to both the Sampling method and DI-COD, while the EH-COD method achieves the fastest update speed among all four algorithms. However, this efficiency in updates comes at the cost of significantly slower query performance. Specifically, the query time of EH-COD is at least two orders of magnitude higher than that of the other algorithms. On the Multimodal Data and HPMax datasets, EH-COD requires approximately 1.5×10^4 and 5.3×10^4 milliseconds, respectively, for query execution. This trade-off arises because EH-COD involves a time-consuming COD merging operation for query, leading to higher query time complexity while others achieve better query time complexity without merging. In contrast, our SO-COD



(a) Update Time (b) Query Time

Figure 5: Average update time and query time with $\epsilon = 0.1$.



(a) Uniform Random (b) HPMax

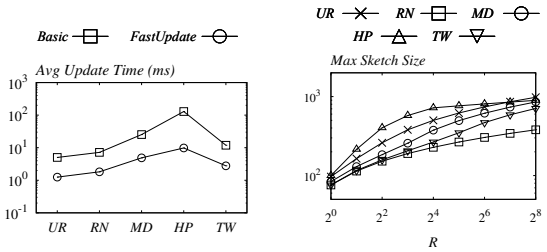
Figure 6: Maximum sketch size with window size varying

achieves the fastest query performance, delivering at least a 5× speedup compared to the second-fastest algorithm.

The experimental results on time efficiency align well with the theoretical time complexity analysis (see Table 1 for more details). Regarding amortized update time, SO-COD is comparable to the baseline methods. Specifically, our SO-COD introduces an additive term of $\frac{1}{\epsilon^3}$ in the update time, whereas EH-COD and DI-COD incur multiplicative logarithmic factors of $\log(\epsilon NR)$ and $\log(R/\epsilon)$, respectively. In terms of query time complexity, SO-COD demonstrates clear advantages over both EH-COD and DI-COD. Compared to EH-COD (resp. DI-COD), our method reduces the query time by a factor of $O\left(1/\epsilon^2\right)$ (resp. $O\left(R \cdot \log\left(R/\epsilon\right)\right)$).

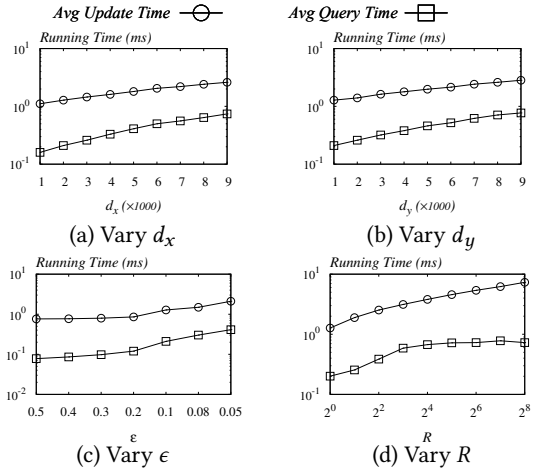
Varying Window Size. To examine the impact of the window size setting on each algorithm, we vary the window size and then report the maximum sketch size of each algorithm. The error parameter is fixed at $\epsilon = 0.1$ for each algorithm. For the Uniform Random dataset, we vary the window size from $\{500, 1000, 2000, 4000\}$; for the HPMax dataset, we vary the window size from $\{1000, 2000, 5000, 10000\}$. The results are presented in Figure 6. From the figures, it is observed that our algorithm requires the smallest sketch size. We also noticed that all the algorithms, except for EH-COD, demonstrate high robustness to the setting of window size. The reason for the unsatisfactory performance of EH-COD is that, there exists a hidden large constant within its space complexity. Consequently, its empirical sketch size is close to the window size, i.e., almost storing the whole window. Therefore, when the window size varies, its sketching performance is significantly affected. Interestingly, although the sampling method theoretically involves the multiplicative term $\log(N)$, the result shows a low sensitivity to the window size and a slightly increasing trend as the window size increases. This suggests that the theoretical space complexity may be overly conservative, indicating potential for further optimization.

Impact of FastUpdate Optimization. We evaluated the performance of our FastUpdate algorithm (see Algorithm 5) against the naive update approach (see Algorithm 3) to demonstrate the effectiveness of our proposed optimization. Both update strategies were tested across all datasets, and the average update times are



(a) FastUpdate Optimization (b) Max Sketch Size

Figure 7: (a) Impact of FastUpdate; (b) Maximum sketch size of SO-COD with varying R.



(a) Vary d_x (b) Vary d_y

(c) Vary ϵ (d) Vary R

Figure 8: Impact of parameters on update and query performance: Uniform Random dataset.

reported in Figure 7(a). Our results show that FastUpdate significantly outperforms the naive version, achieving speedups of up to 13×. This performance gain stems from the fact that the naive algorithm performs a full QR decomposition for each newly arriving column, whereas FastUpdate employs an incremental update strategy, avoiding the costly full matrix decompositions.

Impact of Value R. We further investigate the impact of the parameter R on the sketch size. Recall that R denotes the upper bound on the squared norms of all column vectors. To evaluate this, we rescale the norms of the column vectors such that R varies from 1 to 256, and we report the maximum sketch size, as shown in Figure 7(b). As we can observed, the sketch size of our algorithm increases only slightly with larger values of R. For example, on the HPMax dataset, the sketch size ranges from 100 to 980 as R increases from 1 to 256. This demonstrates a low sensitivity to R, aligning with the theoretical space complexity's $\log(R)$ multiplicative factor.

Parameters Sensitivity for Update and Query Performance. Next, we analyze the impact of parameters on the update and query performance of SO-COD. We use the synthetic Uniform Random dataset, as it allows us to conveniently vary the matrix size and R for clearer analysis. For analysis of d_x, we fix $d_y = 1000$ (the default size), $\epsilon = 0.1$, $R = 1$, and vary d_x from 1000 to 9000; for analysis of d_y, we fix $d_x = 2000$ (the default size), $\epsilon = 0.1$, $R = 1$, and vary d_y from 1000 to 9000; for the analysis of ϵ, we fix $d_x = 2000$, $d_y = 1000$, $R = 1$,

and vary ϵ from 0.5 to 0.05; for the analysis of R, we set $d_x = 2000$, $d_y = 1000$, $\epsilon = 0.1$, and vary R from 1 to 256.

As shown in Figures 8(a) and (b), both the query time and update time exhibit consistent growth with the increase of d_x and d_y, confirming their linear dependence on d_x and d_y.

When varying ϵ, both update and query complexities include the term $\frac{d_x+d_y}{\epsilon}$. Hence, as ϵ decreases, the running time increases roughly inversely with ϵ. This explains the monotonic growth observed in Figure 8(c), where smaller ϵ values result in higher costs for both update and query operations. The curves show consistent upward trends, matching the theoretical $1/\epsilon$ dependence.

When varying R, the dependence differs between query and update performance. For queries, the cost first increases with R when $R \leq 8$. Once $R \geq 8$, the query cost becomes stable, consistent with the theoretical $\log \log R$ dependence. The initial increase (when $R \leq 8$) arises because the number of snapshots is less than $6/\epsilon$ (see Algorithm 8), so a larger R increases the number of snapshots, thereby degrading query performance. When R is sufficiently large, the number of snapshots reaches the upper bound $6/\epsilon$, at which point the performance stabilizes. For updates, since the update complexity of SO-COD depends on $\log R$, the cost exhibits more noticeable growth as R increases. This explains the steeper slope of the update curve compared to the query curve when varying R.

7 CONCLUSION

In this paper, we propose SO-COD, a novel algorithm for approximate matrix multiplication over sliding windows. SO-COD achieves the optimal space complexity in both the normalized and unnormalized settings. Experiments on synthetic and real-world datasets demonstrate its space efficiency and superior estimation accuracy. Future work will focus on reducing the time complexity to achieve linear dependence on the number of non-zero entries.

ACKNOWLEDGMENTS

This work was supported by the Hong Kong RGC GRF grant (No. 14217322) and the 1+1+1 CUHK-CUHK(SZ)-GDSTC Joint Collaboration Fund (No. 2025A0505000045).

A PROOFS

We provide the proof of Theorem 2. All omitted proofs are deferred to our technical report [1].

Proof of Theorem 2. We denote the matrices of snapshot vectors generated between timestamps a and b as $\boldsymbol{C}_{a,b}$ and $\boldsymbol{D}_{a,b}$. Specifically, $\boldsymbol{C}_{a,b}$ if formed by stacking $s_j.u$ for all $s_j.t \in [a,b]$ and $\boldsymbol{D}_{a,b}$ is formed by stacking $s_j.v$ for all $s_j.t \in [a,b]$. Similarly, let $\boldsymbol{X}_{a,b}$ and $\boldsymbol{Y}_{a,b}$ represent the matrices stacked by all $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$, respectively, where $i \in [a,b]$. We denote P as the moment right before the primary sketch begins to receive updates. By the refresh mechanism, we have $P = (k-1)N$, where $k = \max(1, \lfloor (T-1)/N \rfloor)$. Define $(\hat{\boldsymbol{A}}_T, \hat{\boldsymbol{B}}_T)$ as the primary COD sketch at moment T. Next, we have

$$
\begin{aligned}
&\left\| \boldsymbol{X}_W \boldsymbol{Y}_W^T - \boldsymbol{A}_W \boldsymbol{B}_W^T \right\|_2 \\
&= \left\| \boldsymbol{X}_{T-N+1,T} \boldsymbol{Y}_{T-N+1,T}^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T - \boldsymbol{C}_{T-N+1,T} \boldsymbol{D}_{T-N+1,T}^T \right\|_2 \\
&= || \boldsymbol{X}_{P,T} \boldsymbol{Y}_{P,T}^T - \boldsymbol{X}_{P,T-N} \boldsymbol{Y}_{P,T-N}^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T - \boldsymbol{C}_{P,T} \boldsymbol{D}_{P,T}^T \\
&+ \boldsymbol{C}_{P,T-N} \boldsymbol{D}_{P,T-N}^T ||_2 \leq \left\| \boldsymbol{X}_{P,T} \boldsymbol{Y}_{P,T}^T - \boldsymbol{C}_{P,T} \boldsymbol{D}_{P,T}^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T \right\| + \\
&\left\| \boldsymbol{X}_{P,T-N} \boldsymbol{Y}_{P,T-N}^T - \boldsymbol{C}_{P,T-N} \boldsymbol{D}_{P,T-N}^T \right\|_2 \\
&= \left\| [\boldsymbol{X}_{P,T}, -\boldsymbol{C}_{P,T}] [\boldsymbol{Y}_{P,T}, \boldsymbol{D}_{P,T}]^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T \right\| + \\
&\left\| \boldsymbol{X}_{P,T-N} \boldsymbol{Y}_{P,T-N}^T - \boldsymbol{C}_{P,T-N} \boldsymbol{D}_{P,T-N}^T - \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T + \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\|_2 \\
&\leq \left\| [\boldsymbol{X}_{P,T}, -\boldsymbol{C}_{P,T}] [\boldsymbol{Y}_{P,T}, \boldsymbol{D}_{P,T}]^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T \right\| + \left\| \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\|_2 + \\
&\left\| [\boldsymbol{X}_{P,T-N}, -\boldsymbol{C}_{P,T-N}] [\boldsymbol{Y}_{P,T-N}, \boldsymbol{D}_{P,T-N}]^T - \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\| \quad (1)
\end{aligned}
$$

where $(\hat{\boldsymbol{A}}_{T-N}, \hat{\boldsymbol{B}}_{T-N})$ and $(\hat{\boldsymbol{A}}_T, \hat{\boldsymbol{B}}_T)$ can be considered as the COD sketches of matrices $([\boldsymbol{X}_{P,T-N}, -\boldsymbol{C}_{P,T-N}], [\boldsymbol{Y}_{P,T-N}, \boldsymbol{D}_{P,T-N}])$ and $([\boldsymbol{X}_{P,T}, -\boldsymbol{C}_{P,T}], [\boldsymbol{Y}_{P,T}, \boldsymbol{D}_{P,T}])$, respectively. Extracting a snapshot with vectors (v,u) from $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}})$ can be seen as a COD sketch update with pair of columns $(-v,u)$. It is also known that:

$$
\left\| \boldsymbol{C}_{P,T} \right\|_F \left\| \boldsymbol{D}_{P,T} \right\|_F \leq \left\| \boldsymbol{X}_{P,T} \boldsymbol{Y}_{P,T}^T \right\|_* \leq \left\| \boldsymbol{X}_{P,T} \right\|_F \left\| \boldsymbol{Y}_{P,T} \right\|_F = T - P.
$$

Similarly, we also have

$$
\left\| \boldsymbol{C}_{P,T-N} \right\|_F \left\| \boldsymbol{D}_{P,T-N} \right\|_F \leq T - N - P.
$$

The two previous terms are products involving the Frobenius norm. We associate these terms with P, which represents the timestamp when the primary sketch is ready to incorporate new vectors. Due to the refresh mechanism, we bound this by $O(N)$, for $T-P$ is no more than $2N$ and $T-N-P$ is no more than N.

By the approximation error guarantee of COD [35], we have:

$$
\begin{aligned}
&\left\| [\boldsymbol{X}_{P,T}, -\boldsymbol{C}_{P,T}] [\boldsymbol{Y}_{P,T}, \boldsymbol{D}_{P,T}]^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T \right\|_2 \\
&\leq \epsilon \left\| [\boldsymbol{X}_{P,T}, -\boldsymbol{C}_{P,T}] \right\|_F \left\| [\boldsymbol{Y}_{P,T}, -\boldsymbol{D}_{P,T}] \right\|_F = 2\epsilon(T-P) \quad (2) \\
&\left\| [\boldsymbol{X}_{P,T-N}, -\boldsymbol{C}_{P,T-N}] [\boldsymbol{Y}_{P,T-N}, \boldsymbol{D}_{P,T-N}]^T - \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}^T \right\|_2 \\
&\leq \epsilon \left\| [\boldsymbol{X}_{P,T-N}, -\boldsymbol{C}_{P,T-N}] \right\|_F \left\| [\boldsymbol{Y}_{P,T-N}, -\boldsymbol{D}_{P,T-N}] \right\|_F \\
&= 2\epsilon(T-N-P) \quad (3) \\
&\left\| \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\|_2 \leq \epsilon N. \quad (4)
\end{aligned}
$$

Combining Inequalities 1-4, the approximation error is bounded by

$$
\begin{aligned}
&\left\| \boldsymbol{X}_W \boldsymbol{Y}_W^T - \boldsymbol{A}_W \boldsymbol{B}_W^T \right\|_2 \leq \left\| [\boldsymbol{X}_{P,T}, -\boldsymbol{C}_{P,T}] [\boldsymbol{Y}_{P,T}, \boldsymbol{D}_{P,T}]^T - \hat{\boldsymbol{A}}_T \hat{\boldsymbol{B}}_T^T \right\| \\
&+ \left\| [\boldsymbol{X}_{P,T-N}, -\boldsymbol{C}_{P,T-N}] [\boldsymbol{Y}_{P,T-N}, \boldsymbol{D}_{P,T-N}]^T - \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\| \\
&+ \left\| \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\|_2 \leq 2\epsilon(T-P) + 2\epsilon(T-N-P) + 2\epsilon N \\
&\leq \epsilon(4T - 4P) \leq 8\epsilon N.
\end{aligned}
$$

Next, we prove the number of snapshots is bounded by $O(\frac{1}{\epsilon})$. Suppose that at time T the queue has k snapshots of vector pairs $(\boldsymbol{u}_1, \boldsymbol{v}_1), (\boldsymbol{u}_2, \boldsymbol{v}_2), \cdots, (\boldsymbol{u}_k, \boldsymbol{v}_k)$, and the corresponding singular values are $\sigma_1, \sigma_2, \cdots, \sigma_k$. Then we know

$$
\begin{aligned}
k\epsilon N &\leq \sum_{i=1}^{k} \left\| \boldsymbol{u}_i \right\|_2 \left\| \boldsymbol{v}_i \right\|_2 = \sum_{i=1}^{k} \sigma_i \leq \left\| \hat{\boldsymbol{A}}_{T-N} \hat{\boldsymbol{B}}_{T-N}{}^T \right\|_* + \left\| \boldsymbol{X}_W \boldsymbol{Y}_W^T \right\|_* \\
&\leq l\epsilon N + N = 2N.
\end{aligned}
$$

Hence, we have $k\epsilon N \leq 2N$, which implies $k \leq \frac{2}{\epsilon}$. Thus, the space cost of SO-COD for normalized model is dominated by COD sketch and snapshot storage, yielding a total space cost of $O((d_x + d_y)/\epsilon)$.

REFERENCES

[1] Source code and technical report. https://github.com/CUHK-DBGroup/SOCOD.

[2] Arvind Arasu and Gurmeet Singh Manku. 2004. Approximate Counts and Quantiles over Sliding Windows. In *PODS*. 286–296.

[3] Raman Arora and Karen Livescu. 2013. Multi-view CCA-based acoustic features for phonetic recognition across speakers and domains. In *ICASSP*. 7135–7139.

[4] Brian Babcock, Mayur Datar, and Rajeev Motwani. 2002. Sampling from a moving window over streaming data. In *SODA*. 633–634.

[5] Roland Badeau, Gaël Richard, and Bertrand David. 2004. Sliding window adaptive SVD algorithms. *IEEE Trans. Signal Process.* 52, 1 (2004), 1–10.

[6] Ran Ben-Basat, Roy Friedman, and Rana Shahout. 2018. Stream Frequency Over Interval Queries. *Proc. VLDB Endow.* 12, 4 (2018), 433–445.

[7] A Busuioc, R Tomozeiu, and C Cacciamani. 2008. Statistical downscaling model based on canonical correlation analysis for winter extreme precipitation events in the Emilia-Romagna region. *International Journal of Climatology* 28, 4 (2008), 449–464.

[8] Kenneth L. Clarkson and David P. Woodruff. 2013. Low rank approximation and regression in input sparsity time. In *STOC*. 81–90.

[9] Michael B. Cohen, Jelani Nelson, and David P. Woodruff. 2016. Optimal Approximate Matrix Product in Terms of Stable Rank. In *ICALP*. 11:1–11:14.

[10] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proc. VLDB Endow.* 1, 2 (2008), 1530–1541.

[11] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *SIAM J. Comput.* 31, 6, 1794–1813.

[12] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation using Matrix Multiplication. In *SIGMOD*. 1213–1223.

[13] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. 2006. Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication. *SIAM J. Comput.* 36, 1, 132–157.

[14] Pavlos S. Efraimidis and Paul G. Spirakis. 2006. Weighted random sampling with a reservoir. *Inf. Process. Lett.* 97, 5, 181–185.

[15] Deena P. Francis and Kumudha Raimond. 2022. A practical streaming approximate matrix multiplication algorithm. *J. King Saud Univ. Comput. Inf. Sci.* 34, 1, 1455–1465.

[16] Mohamed Medhat Gaber, Arkady B. Zaslavsky, and Shonali Krishnaswamy. 2005. Mining data streams: a review. *SIGMOD Rec.* 34, 2 (2005), 18–26.

[17] Mina Ghashami, Amey Desai, and Jeff M. Phillips. [n.d.]. Improved Practical Matrix Sketching with Guarantees. In *ESA*. 467–479.

[18] Mina Ghashami, Edo Liberty, Jeff M. Phillips, and David P. Woodruff. 2016. Frequent Directions: Simple and Deterministic Matrix Sketching. *SIAM J. Comput.* 45, 5, 1762–1792.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[20] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proc. ACM Manag. Data* 1, 1 (2023), 25:1–25:26.

[21] Yishan Jiao. 2006. Maintaining stream statistics over multiscale sliding windows. *ACM Trans. Database Syst.* 31, 4 (2006), 1305–1334.

[22] Cheqing Jin, Ke Yi, Lei Chen, Jeffrey Xu Yu, and Xuemin Lin. 2008. Sliding-window top-k queries on uncertain streams. *Proc. VLDB Endow.* 1, 1 (2008), 301–312.

[23] Homin Kang, Hyuck-Chan Kwon, and Duksu Kim. 2020. HPMaX: heterogeneous parallel matrix multiplication using CPUs and GPUs. *Computing* 102, 12, 2607–2631.

[24] Lap-Kei Lee and H. F. Ting. 2006. Maintaining significant stream statistics over sliding windows. In *SODA*. 724–732.

[25] Lap-Kei Lee and H. F. Ting. 2006. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *PODS*. 290–297.

[26] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. , 740–755 pages.

[27] Yichao Lu and Dean P. Foster. 2014. large scale canonical correlation analysis with iterative least squares. In *NeurIPS*. 91–99.

[28] Luo Luo, Cheng Chen, Guangzeng Xie, and Haishan Ye. 2021. Revisiting Co-Occurring Directions: Sharper Analysis and Efficient Algorithm for Sparse Matrices. In *AAAI*. 8793–8800.

[29] Avner Magen and Anastasios Zouzias. 2011. Low Rank Matrix-valued Chernoff Bounds and Approximate Matrix Multiplication. In *SODA*. 1422–1436.

[30] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate Frequency Counts over Data Streams. In *VLDB*. 346–357.

[31] Peter Mazuruse. 2014. Canonical correlation analysis: Macroeconomic variables versus stock returns. *Journal of Financial Economic Policy* 6, 2 (2014), 179–196.

[32] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*, Thomas Eiter and Leonid Libkin (Eds.), Vol. 3363. 398–412.

[33] Jayadev Misra and David Gries. 1982. Finding Repeated Elements. *Sci. Comput. Program.* 2, 2, 143–152.

[34] Youssef Mroueh, Etienne Marcheret, and Vaibhava Goel. 2015. Asymmetrically Weighted CCA And Hierarchical Kernel Sentence Embedding For Image & Text Retrieval. *CORR* abs/1511.06267 (2015).

[35] Youssef Mroueh, Etienne Marcheret, and Vaibhava Goel. 2017. Co-Occurring Directions Sketching for Approximate Matrix Multiply. In *AISTATS*. 567–575.

[36] Odysseas Papapetrou, Minos N. Garofalakis, and Antonios Deligiannakis. 2015. Sketching distributed sliding-window data streams. *VLDB J.* 24, 3 (2015), 345–368.

[37] Tamás Sarlós. 2006. Improved Approximation Algorithms for Large Matrices via Random Projections. In *FOCS*. 143–152.

[38] Alexei Vinokourov, John Shawe-Taylor, and Nello Cristianini. 2002. Inferring a Semantic Representation of Text via Cross-Language Correlation Analysis. In *NeurIPS*. 1473–1480.

[39] Yanhao Wang, Qi Fan, Yuchen Li, and Kian-Lee Tan. 2017. Real-Time Influence Maximization on Dynamic Social Streams. *Proc. VLDB Endow.* 10, 7 (2017), 805–816.

[40] Zhewei Wei, Xuancheng Liu, Feifei Li, Shuo Shang, Xiaoyong Du, and Ji-Rong Wen. 2016. Matrix Sketching Over Sliding Windows. In *SIGMOD*. 1465–1480.

[41] Siyue Wu, Dingming Wu, Junyi Quan, Tsz Nam Chan, and Kezhong Lu. 2024. Efficient and Accurate PageRank Approximation on Large Graphs. *Proc. ACM Manag. Data* 2, 4 (2024), 196:1–196:26.

[42] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S. Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *Proc. VLDB Endow.* 13, 5 (2020), 670–683.

[43] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S Bhowmick. 2021. Scaling Attributed Network Embedding to Massive Graphs. *Proc. VLDB Endow.* 14, 1 (2021), 37–49.

[44] Ziqi Yao, Lianzhi Li, Mingsong Chen, Xian Wei, and Cheng Chen. 2024. Approximate Matrix Multiplication over Sliding Windows. In *SIGKDD*. 3896–3906.

[45] Qiaomin Ye, Luo Luo, and Zhihua Zhang. 2016. Frequent Direction Algorithms for Approximate Matrix Multiplication with Applications in CCA. In *IJCAI*. 2301–2307.

[46] Hanyan Yin, Dongxie Wen, Jiajun Li, Zhewei Wei, Xiao Zhang, Zengfeng Huang, and Feifei Li. 2024. Optimal Matrix Sketching over Sliding Windows. *Proc. VLDB Endow.* 17, 9, 2149–2161.

[47] Haida Zhang, Zengfeng Huang, Zhewei Wei, Wenjie Zhang, and Xuemin Lin. 2017. Tracking Matrix Approximation over Distributed Sliding Windows. In *ICDE*. 833–844.

[48] Xingyi Zhang, Kun Xie, Sibo Wang, and Zengfeng Huang. 2021. Learning based proximity matrix factorization for node embedding. In *SIGKDD*. 2243–2253.