Evaluating Continuous P-Fairness Queries On Data Streams

Subhodeep Ghosh
sg2646@njit.edu
NJIT
USA

Zhihui Du
zhihui.du@njit.edu
NJIT
USA

Manish Kumar
manishk@iitrpr.ac.in
IIT Ropar
India

Angela Bonifati
angela.bonifati@univ-lyon1.fr
Lyon 1 University
France

David A. Bader
bader@njit.edu
NJIT
USA

Senjuti Basu Roy
senjutib@njit.edu
NJIT
USA

Abstract

We introduce a fairness model for continuous queries over data streams, called **generalized P-fairness**. Generalized P-fairness allows to enforce fairness at a smaller granularity within blocks inside windows of the streaming data. It lies in between traditional group fairness, requiring per-window fairness, and P-fairness, requiring fairness in all prefixes of the window. It provides a more practical and flexible fairness model for streaming contexts. By adopting the sliding window model, we address two key problems: efficiently monitoring whether a stream satisfies generalized P-fairness, and reordering the current stream when it does not. To support real-time monitoring, we design sketch-based data structures for preprocessing the window that maintains attribute distributions with minimal overhead. We also present optimal, efficient algorithms for the reordering problem, backed by theoretical guarantees. We validate our methods through simulations on four real-world streaming scenarios. These results support our theoretical analysis, showing millisecond-level processing and throughput of 30, 000 queries per second on average, depending on the parameters. Our stream reordering algorithm improves generalized P-fairness by up to 95% at times but 50-60% on average, depending on the dataset.

ACM Reference Format:
Subhodeep Ghosh, Zhihui Du, Manish Kumar, Angela Bonifati, David A. Bader, and Senjuti Basu Roy. 2018. Evaluating Continuous P-Fairness Queries On Data Streams. In *Proceedings of Very Large Database Conference (VLDB'26).* ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Many real-world applications inherently generate data streams rather than static datasets—for instance, financial market tickers, performance metrics in network monitoring and traffic management, server logs, content recommendation engines, and user clickstreams in web analytics and personalization systems [2, 12]. Due to the continuous and evolving nature of these streams, querying them requires continuous queries—that is, queries that are executed persistently over time and return updated results as new data arrives.

In parallel, fairness [5] in algorithmic decision-making has emerged as a critical issue in data management, especially in high-stakes domains such as hiring, lending, and law enforcement [20, 28, 30]. A substantial body of research has focused on *group fairness*, which seeks to ensure that algorithmic outcomes do not systematically disadvantage particular demographic groups. However, most of this work assumes access to static datasets, where fairness constraints are applied once on a fixed input. Despite the widespread relevance of these streaming scenarios in real world applications, the study of fairness in streaming settings remains significantly underexplored.

A natural extension of group fairness to data streams is to check whether each sliding window satisfies fairness constraints like proportional representation [11, 20]. At the other extreme, P-fairness [32] demands that every prefix of the stream meets fairness criteria, ensuring persistent fairness over time. However, P-fairness is often too strict, while window-level fairness can be too lenient—highlighting the need for a more balanced fairness notion.

Motivating application: Online Content Recommendation Consider a news or video streaming platform that continuously recommends content (i.e., item such as articles or videos) to users in real time. The platform processes a data stream of items— such as articles or videos—from creators belonging to different ethnic groups. Its goal is to ensure fair exposure for all creator groups within each sliding window of recommendations.

Continuous Monitoring Queries. Each window is divided into several equal-sized subintervals (blocks). The platform first attempts to verify if every block in a window contains a proportional representation of items generated from each ethnicity. However, due to natural differences in popularity, certain ethnic groups can dominate the recommendations within a window, resulting in unfair visibility and opportunity.

Continuous Stream Reordering. If the monitoring queries indicate that fairness constraints are not satisfied, the platform pauses to make additional recommendation and reads a few additional items — these items come from landmark bits — each of which generates a new sliding window. It then permits changing the order of content within the current window as well as these landmark bits. By strategically rearranging the content, the platform aims to enforce fairness in as many blocks of each sliding window as possible, ensuring consistent and equitable exposure over time.

VLDB'26, Aug 31st – Sep 4th, 2026, Boston, MA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

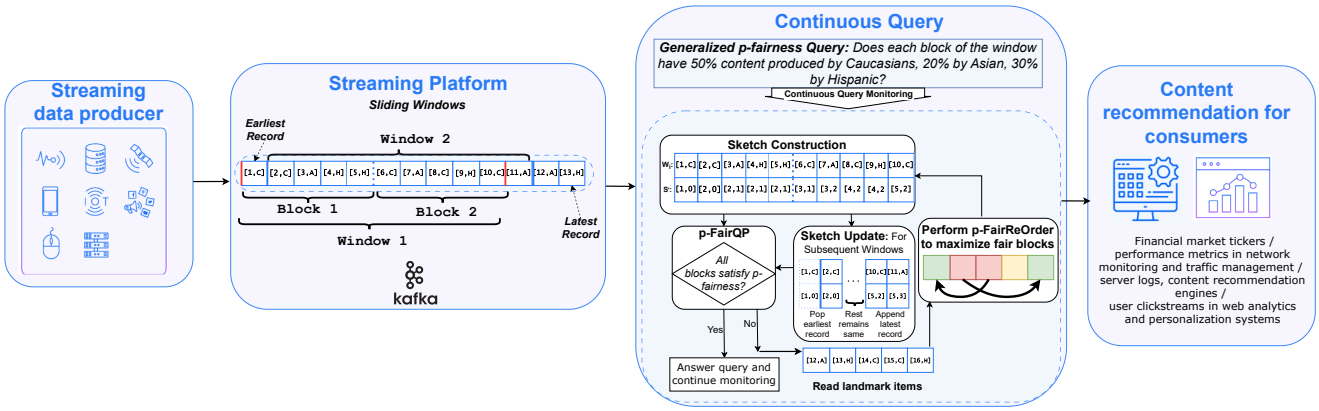


Figure 1: The proposed framework

B_1:	$[1, C]$	$[2, C]$	$[3, \mathcal{A}]$	$[4, \mathcal{H}]$	$[5, \mathcal{H}]$
B_2:	$[6, C]$	$[7, \mathcal{A}]$	$[8, C]$	$[9, \mathcal{H}]$	$[10, \mathcal{H}]$
B_3:	$[11, \mathcal{A}]$	$[12, \mathcal{A}]$	$[13, C]$	$[14, \mathcal{H}]$	$[15, \mathcal{H}]$

Figure 2: A sliding window of 15 items (contents) with 3 Blocks, which consists contents from three ethnic groups $[C(Caucasian), \mathcal{A}(Asian), \mathcal{H}(Hispanic)]$

Attribute	Value	Query 1	Query 2
Ethnicity	Caucasian (C)	0.3	0.5
	Asian ($\mathcal{A}$)	0.3	0.2
	Hispanic ($\mathcal{H}$)	0.4	0.3

Table 1: Proportionate fairness constraints for different queries

Contributions. We introduce **generalized P-fairness queries** for evaluating continuous group fairness over data streams. By decomposing each sliding window into blocks and enforcing fairness at the block level (Figure 1), these queries offer a more localized and flexible alternative to traditional window-level or prefix-based fairness notions.

Motivated by practical needs, we develop a computational framework addressing two core challenges: **Monitoring Continuous Generalized P-Fairness Queries** and the **Continuous Stream Reordering Problem**.

The monitoring component evaluates, for each sliding window, whether the fairness constraints are satisfied within every block (Refer to Continuous Query Module in Figure 1). If violations are detected, the framework reads a small number of additional items—referred to as landmark bits—each of which spawns a new sliding window. The framework then allows for reordering the content within the current window and these landmark bits. We formalize this content reordering task as the *Continuous Stream Reordering* problem, which aims to maximize the total number of unique fair blocks across both the current window and the newly generated windows.

To support continuous generalized P-fairness monitoring queries, we propose a **sketch-based** data structure designed specifically for streaming environments. This sketch efficiently encodes the distribution of protected attribute values within each block of a sliding window, enabling exact and efficient evaluation of fairness constraints. The sketch is highly space-efficient, requiring only linear space relative to the window size, which makes it ideal for high-throughput data streams. It also supports incremental updates: as new items arrive and old ones expire, the sketch can be updated efficiently without the need for full recomputation. By maintaining up-to-date distributional statistics, it allows for exact and efficient fairness evaluation across all blocks. We present analytical results that establish the correctness of the query answers and provide formal guarantees on the sketch's performance. Specifically, we analyze the query execution time, preprocessing overhead, update efficiency, and space complexity, demonstrating that the sketch offers accurate, low-latency monitoring while maintaining strong scalability in streaming settings.

We propose an **efficient and optimal** algorithm `p-FairReOrder` to the **Continuous Stream Reordering** problem, which incorporates both the current window and the landmark items to maximize the number of *unique fair blocks* (where two blocks are considered unique if they have at least one uncommon item). Each fair block must satisfy specified group-wise representation constraints, and the goal is to maximize the number of such blocks not only within the current window but also across future windows defined by landmark positions. The key idea of the algorithm is to construct a specific permutation of the content, called an *isomorphic stream* (see details in Sec. 4), which satisfies the fairness requirement consistently within its blocks. We present theoretical analyses.

We conduct extensive experiments using four large-scale real-world datasets by simulating streaming environment. The results align with our theoretical analysis, confirming the optimality of our solutions. The proposed framework processes queries in fractions of milliseconds, achieving an average throughput of 30,000 queries per second (depending on parameters), with minimal memory usage and high update efficiency. As no existing work addresses our

problem setting, we design appropriate baselines. Our framework outperforms them by several orders of magnitude.

2 Data Model, Proposed Framework, Formalism

We first present our data model, followed by our proposed framework, and then formally define the studied problems.

2.1 Data Model

Attribute denoting fairness. A protected attribute $\mathcal{A}$ has values ℓ possible values. Using Figure 1, *Ethnicity* is one of these attributes with three possible values, *C, A, H.*

Count based sliding window. A count-based sliding window is a type of window in stream processing where computations are performed over the most recent fixed number of data items, and the window moves forward by a specified number of items (called the slide). Let W represent a sliding window with $|W|$ items, and the default slide is 1 (unless otherwise specified). Basically, it maintains the last $|W|$ items of the data stream and updates the result every time a new item arrives.

Using Example 1, each sliding window W contains 15 items.

Data item. A data item t in a sliding window W is a pair = $< i, p >$, where i is its order in W, and p represents any specific value of the protected attribute $\mathcal{A}$ that is, $p \in \ell$.

Using Example 1, the third video of the window is created by a contributor of Asian (A) ethnicity.

Blocks in a window. A sliding window W is divided into a set of k equal sized subintervals, each called a block. A block B represents one of these subintervals, each with s elements or items. Wlog, we assume block size $s = \frac{|W|}{k}$.

We use B_q to represent the q-th block in a window W ($1 \leq q \leq k$). If the q-th block starts at item i, then $q = \lfloor \frac{(i+s-1)}{s} \rfloor$, ($1 \leq i \leq |W| - s + 1$).

Using Example 1, each sliding window contains $k = 3$ blocks, each with $15/3 = 5$ items. Blocks B_1, B_2, B_3 start at item 1, 6, and 11, respectively. Figure 2 shows $\{B_1, B_2, B_3\}$ in window W.

Prefix, Suffix of a Block. For a block B_q starting at index i

$$B_q = [t_i, t_{i+1}, \ldots, t_{i+s-1}], \quad \text{for } 1 \leq i \leq |W| - s + 1.$$

its prefix of length j is:

$$\text{Prefix}_j(B_q) = [t_i, t_{i+1}, \ldots, t_{i+j-1}], \quad \text{for } 1 \leq j \leq s.$$

Its suffix of length j is:

$$\text{Suffix}_j(B_q) = [t_{i+s-j}, t_{i+s-j+1}, \ldots, t_{i+s-1}], \quad \text{for } 1 \leq j \leq s.$$

As an example, the first block B_1's prefix of length 3 are items $[t_1, t_2, t_3]$ and the suffix of length 2 are items $[t_4, t_5]$.

Landmark Items. Given the current window W, the landmark items consist of $|\mathcal{X}|$ additional elements stored immediately after the window, starting from position $|W| + 1$. Figure 3 shows 5 such additional items.

Unique Blocks. A block consists of s consecutive items, starting at item i and ending at item $i + s - 1$. Without loss of generality, two blocks are considered unique if each contains at least one item not present in the other. As an example, the first block in the current window and the first block in the slided window (after the first slide) are unique because one contains items $[i_1, \ldots, i_5]$, the other contains $[i_2, \ldots, i_6]$ after the slide.

Disjoint Blocks. Blocks that do not overlap with one another are called *disjoint blocks*. If each of these blocks individually satisfies the given fairness criterion, they are referred to as *disjoint fair blocks*. As an example block B_1, B_2, B_3 are fully disjoint.

Proportionate or P-Fairness Constraints. For the attribute A with ℓ different values, for each $p \in \cup_{i=1}^{l} \mathcal{A}_i$, $f(p)$ denotes its required proportion constraint, such that $\Sigma_{p=1}^{\ell} f(p) = 1$.

Table 1 shows two different group fairness constraints of the *Ethnicity* attribute.

P-Fairness Query over Sliding Window. Given a sliding window W and $f(p)$ specifying the desired proportion for each value p of the protected attribute A, a P-Fair query returns *Yes* if, in every block within W, the number of items with attribute value p lies within the range $[\lfloor f(p) \times s \rfloor, \lceil f(p) \times s \rceil]$ for all p. Otherwise, it returns *No*. The floor and ceiling expression is introduced because $f(p) \times |s|$ may not be an integer, therefore, floor and ceiling allows it to round up to the two closest integer values.

Generalized Proportionate Fair Sliding Window. A sliding window W is Generalized P-fair, if the P-Fairness Query returns an answer *Yes*.

Using *Query constraint 1* listed in Table 1, this will require every block to have $[\lfloor .3 \times 5 \rfloor, \lceil .3 \times 5 \rceil]$ of data items of value C and value $\mathcal{A}$ ([1,2] data items) and $[\lfloor .4 \times 5 \rfloor, \lceil .4 \times 5 \rceil]$, i.e., 2 data items with protected attribute value $\mathcal{H}$. Based on this requirement, the window shown in Figure 2 is Generalized P-fair.

2.2 Proposed Framework

Given each sliding window and the specified P-fairness query, the framework (Refer to Figure 1) continuously monitors the windows. If the answer to Generalized P-fairness query is *No*, the framework first checks whether it contains a sufficient number of data items so that the items in the window could be reordered to satisfy the constraints. This is done by counting the number of instances corresponding to each protected attribute value p, and verifying whether this count meets the required proportion for the entire window. Specifically, if the following condition holds:

$$\forall p \in \cup_{i=1}^{l} \mathcal{A}_i, \quad \sum_{i=1}^{|W|} \mathbb{1}[\{t_i(A) = p\} \geq (k \cdot \lfloor f(p) \cdot s \rfloor)]$$

Basically, for every attribute value $p \in \cup_{i=1}^{l} \mathcal{A}_i$, it checks if the total number of times p appears in the window W (i.e., the number of items i such that $t_i(A) = p$) is at least $k \cdot \lfloor f(p) \cdot s \rfloor$, where $f(p)$ denotes the proportionality of p, s is block size, and k is the number of blocks. In that case, the items in the current window can potentially be re-ordered to satisfy the Generalized P-fairness constraint. If the condition holds, then internal reordering can be easily performed within the current window. However, a more challenging scenario arises when the condition does not hold—this is the case we address closely in this work.

In the latter case, the framework pauses and reads an additional $|\mathcal{X}|$ landmark items. Each landmark item corresponds to a slide,

collectively resulting in $|\mathcal{X}|$ additional sliding windows. The framework then considers both the items in the current window W and the landmark items $\mathcal{X}$, and performs reordering.

The ideal objective is to make both the current window and the $|\mathcal{X}|$ slided windows satisfy the Generalized P-fairness constraint. However, it is possible that even after incorporating the $|\mathcal{X}|$ landmark items, the combined set of data items may still lack sufficient instances of certain protected attribute values.

In such cases, the framework performs a special permutation within $W \cup \mathcal{X}$ with the goal of maximizing the total number of P-fair windows among the current window and the $|\mathcal{X}|$ slided windows.

After performing the permutation, the monitoring process resumes, and these two steps are repeated as needed until the entire stream has been processed.

$\mathcal{X}$: [16, C] [17, $\mathcal{A}$] [18, $\mathcal{A}$] [19, $\mathcal{A}$] [20, $\mathcal{H}$]

Figure 3: Five landmark items $\mathcal{X}$

Symbol	Explanation
W	Window with width $\|W\|$
B	Block in a window with width s
S	Sketch
k	Number of blocks in a window
$\mathcal{A}$	Protected attribute with cardinality ℓ
p	One value of $\mathcal{A}$
$\mathcal{X}$	Landmark items
$\mathcal{A}^i$	Protected attribute value of item i

Table 2: Key notations

2.3 Problem Definitions

Problem 1. **Continuous Monitoring of Generalized P-fairness** *For every sliding window W in a stream and a given P-fairness query, answer* Yes *if each block $B \in W$ is P-fair, else* No.

Given the window in Figure 2 and the *Query 1 constraints* in Table 1, window W is Generalized P-fair, hence the evaluation query answers *Yes*. However, the *Query 2 constraints* in Table 1, window W is *not* Generalized P-fair, hence the evaluation query answers *No*.

Problem 2. **Continuous Stream Reordering using Landmark Items** *Given a window W and additional $|\mathcal{X}|$ landmark items, for a given P-fairness constraint, identify a reordering for data items in $W \cup \mathcal{X}$, so that it maximizes the total number of unique P-fair blocks in W and the slided windows generated by landmark items.*

Given the *Query-2* constraints from Table 1, the minimum number of C instances required in window W of Figure 2 is calculated as $\lfloor 0.5 \times 5 \rfloor \times 3 = 6$. Since W contains only 5 instances of C, an additional 5 landmark items are read (as shown in Figure 3). These additional 5 items result in 5 more slided windows.

In total, this creates 16 unique blocks across the current window and the 5 slided windows. The objective is to perform reordering among the 20 available items (from W and the landmark items) such that the maximum possible number of these 16 blocks satisfy the P-fairness constraints.

3 Continuous Query Monitoring

This section presents our technical solution for continuously and efficiently monitoring Generalized P-Fairness queries over a sliding window W. The framework consists of three main steps, where the first and the third are part of preprocessing and the second step is for efficient query evaluation.

(1) **Sketch building:** It first computes the frequency distribution of the protected attribute $\mathcal{A}$ within the current window W, and stores this information in a lightweight data structure we refer to as the *Forward Sketch*.
(2) **Query Processing:** It then leverages the sketch to efficiently answer fairness monitoring queries.
(3) **Sketch update:** As the window slides forward, the sketch is updated to reflect the changes, enabling continuous monitoring.

3.1 Preprocessing

At a high level, sketches [9, 17, 33] offer compact summaries of data distributions. In our setting, the goal is to capture the distribution of protected attribute values within a sliding window.

*Definition 3.1 (**Forward Sketch**).* A *Forward Sketch* `FSketch` S of width $|W|$ is a data structure associated with a window W, where each entry at index i stores a vector of length $(\ell - 1)$ representing the cumulative frequencies of $(\ell - 1)$ protected attribute values from the beginning of the window up to position i. The j-th entry of the vector corresponds to the j-th value of the protected attribute $\mathcal{A}$. Since the protected attribute $\mathcal{A}$ has ℓ unique values, the frequency of the remaining protected attribute value at index i can be inferred by subtracting the sum of the stored frequencies from i, the total number of items considered up to that point.

Consider the sketch S in Figure 4 that shows blocks B_1 and B_2 of the running example. The first bit of the vector is assigned to store the cumulative frequency of `C` and the second one is that of for `A`. As an example, the 4-th entry of S contains $[2, 1]$, denoting a total frequency of 2 of `C` and 1 of `A` from the beginning till the 4-th position of the window. Since $\mathcal{A}$ has three distinct values, the frequency of `H` could be inferred by subtracting each i from the cumulative frequency of `C` and `A` at position i. As an example, the cumulative frequency of `H` from the beginning upto position i is therefore $4 - (2 + 1) = 1$.

3.1.1 Sketch Construction Algorithm. Each entry i of the sketch S stores a vector of length $(\ell - 1)$, representing the cumulative counts of the first $(\ell - 1)$ protected attribute values from the start of the window up to position i. For $i > 1$, the sketch at index i copies the counts from the previous index $(i - 1)$, and then increments the count corresponding to the j-th protected attribute value p if the item t_i has $\mathcal{A}^i = p$. If the encountered value is not stored explicitly in the vector, it does not do anything. This construction ensures that each sketch entry maintains a cumulative sum of the $\ell - 1$ protected attribute values up to that point in the window. From there, the value of the protected attribute that is not stored could always be calculated. Algorithm 1 has the pseudocode.

Theorem 3.2. *Algorithm* `Construct-Sketch` *(Algorithm 1) takes $O(|W| \times \ell)$ to run and takes $O(|W| \times \ell)$ space.*

Algorithm 1 Forward Sketch Construct - FSketch

Require: Window $W = \{t_1, t_2, \ldots, t_{|W|}\}$ of size $|W|$
Require: Set of protected attribute values $\mathcal{A} = \{p_1, p_2, \ldots, p_\ell\}$
Initialize sketch S as list of $|W|$ vectors, each of length $(\ell - 1)$, with all entries set to 0
for $i = 1$ to $|W|$ **do**
  if $i > 1$ **then**
    $S[i] \leftarrow S[i-1]$ ▷ Copy counts from previous entry
  end if
  Let $p \leftarrow \mathcal{A}^{t_i}$ ▷ Protected attribute value of item t_i
  for $j = 1$ to $(\ell - 1)$ **do**
    if $p = p_j$ **then**
      $S[i][j] \leftarrow S[i][j] + 1$
      break
    end if
  end for ▷ If $p = p_\ell$, no update is made
end for
return S

Proof. From Algorithm 1, it is evident that each entry in the sketch requires $O(\ell)$ time to compute and $O(\ell)$ space to store, as it maintains information for $\ell - 1$ protected attribute values. Since the outer loop iterates over the entire window of size $|W|$, the overall time and space complexity of the algorithm is $O(|W| \times \ell)$. □

3.1.2 Sketch Update. During the first time of creation, the sketch initializes by tracking $(\ell - 1)$ protected attribute values, starting from the first index of the first window. This ensures that the earliest data items are captured first. The update ensures that the sketch accurately represents the most recent window of data. When window W is slided over, *the sketch shifts all entries one position to the left.* The last entry in the sketch is then updated considering the last but one entry and with the protected attribute value of the newly added item. This way, the sketch remains synchronized with the current window and continues to provide an up-to-date, compact summary of the protected attributes over time.

Consider the sketch in Figure 4 again. When the window slides over by one item, the 11-th item comes in. All entries in the previous sketch (refer to Figure 5) is now moved one position to the left, and the last entry of the sketch is now updated.

Lemma 3.3. *The sketch takes $O(\ell)$ time to update.*

Proof. Each vector is maintained as a linked list, that takes only a constant time to change the first element of the window. A new vector is appended at the end, which takes $O(\ell)$ time, being the leading cost. □

S:	[1,0]	[2,0]	[2,1]	[2,1]	**[2,1]**	[3,1]	[3,2]	[4,2]	[4,2]	**[4,2]**

Figure 4: Sketch of blocks B_1 and B_2 in Figure 2

W:	[2,C]	[3,$\mathcal{A}$]	[4,$\mathcal{H}$]	[5,$\mathcal{H}$]	[6,C]	[7,$\mathcal{A}$]	[8,C]	[9,$\mathcal{A}$]	[10,C]	[11,$\mathcal{A}$]
S:	[2,0]	[2,1]	[2,1]	[2,1]	[3,1]	[3,2]	[4,2]	[4,2]	[5,2]	[5,3]

Figure 5: Sketch maintenance with one item slide

3.2 Query Processing

Next, we discuss the query processing Algorithm p-FairQP that answers *Yes* or *No* to a given P-fair query. The innovation of the algorithm is that it only needs to check at most the number of blocks k number of entries in the current sketch S, and returns the correct answer always. In some cases, it is able to terminate early, especially when it already has found violation of the fairness constraints. Specifically it only checks those entries of the current sketch S that represents the last entry of each block.

To evaluate the first block of size s, Algorithm p-FairQP first examines the s-th entry of the sketch S, which contains cumulative counts for the first $\ell - 1$ protected attribute values. The count for the final (implicit) attribute is computed by subtracting the sum of the stored counts from item number. The fairness constraint is verified by checking whether all of these $\ell - 1$ counts—and the inferred count of the ℓ-th attribute—satisfy the specified requirements.

For each subsequent block, the algorithm computes the difference between two sketch vectors : specifically, the last entry of the previous block and that of the current block. This difference yields the frequency of each protected attribute value within the current block. The process repeats for all blocks, continuing until either a block violates the fairness constraints—causing the algorithm to terminate early and return *No*—or all blocks pass the checks, in which case p-FairQP returns *Yes*. Algorithm 2 presents the pseudocode.

Consider Figure 4 and Query 1 from Table 1. The window W contains 10 elements, which p-FairQP partitions into two blocks, B_1, B_2, each of size $s = 5$. The algorithm starts by examining the sketch at the end of the first block, i.e., at index 5, where the sketch holds the cumulative vector $[2, 1]$. This indicates that within B_1, there are 2 instances of C and 1 instance of A, implying the remaining 2 items correspond to H. The block satisfies the fairness constraints for Query 1. Next, the algorithm proceeds to the end of the second block, at index 10. It computes the frequency vector for B_2 by subtracting the sketch at index 5 from that at index 10. The resulting vector reveals that B_2 contains 2 instances of C, 1 instance of A, and 2 instances of H. This also satisfies the specified fairness requirements. This process continues for all three blocks. Since all blocks meet the fairness criteria, the algorithm concludes that the answer to Query 1 is *Yes*.

On the other hand, consider Figure 4 and Query 2 from Table 1. Its first two blocks would be deemed fair, but the third block will not, returning *No* as an answer.

Lemma 3.4. *Worst case running time of* p-FairQP *is* $O(k \times \ell)$

Proof. At the worst case, p-FairQP needs to process all k blocks, each requiring $O(\ell)$ time. Therefore, it overall will take $O(k \times \ell)$ time. □

Algorithm 2 p-FairQP:Query Processing Algorithm

Require: Sketch S with $|W|$ entries indexed from 1 to $|W|$;
Require: Block size s, number of blocks k;
Require: Fairness proportions $f(p)$ for each $p \in \{1, \ldots, \ell\}$
Ensure: **Yes** if all blocks satisfy fairness constraints; **No** otherwise
For$b \leftarrow 1$ to k
$curr \leftarrow b \cdot s$
if $b = 1$ **then**
  $countVec \leftarrow S[curr]$
else
  $prev \leftarrow (b-1) \cdot s$
  $countVec \leftarrow S[curr] - S[prev]$
end if
$total \leftarrow s$
$lastCount \leftarrow total - \sum_{j=1}^{\ell-1} countVec[j]$
Construct $fullVec[1 \ldots \ell] \leftarrow (countVec[1 \ldots \ell - 1], lastCount)$
for $p \leftarrow 1$ to ℓ **do**
  $minReq \leftarrow \lfloor f(p) \cdot s \rfloor$
  $maxReq \leftarrow \lceil f(p) \cdot s \rceil$
  if $fullVec[p] < minReq$ **or** $fullVec[p] > maxReq$ **then**
    return No
  end if
end for
return Yes

Lemma 3.5. *Best case running time of* p-FairQP *is* $\Omega(\ell)$

Proof. At the best case, p-FairQP finds violation of the P-fairness constraints in the first block itself, requiring $\Omega(\ell)$ time to declare *No* as the query answer and terminate. □

Lemma 3.6. *Algorithm* p-FairQP *returns correct answer to any P-Fair Query.*

Proof. (sketch). Let W be a window of size $|W| = k \cdot s$, and let the fairness query specify desired proportions $f(p)$ for each protected attribute $p \in \{1, \ldots, \ell\}$. The goal of the query is to verify whether, in every block B_i of size s (for $1 \le i \le k$), the number of occurrences of each attribute p lies within the range $[\lfloor f(p) \cdot s \rfloor, \lceil f(p) \cdot s \rceil]$.

The sketch S stores cumulative counts of $\ell-1$ protected attributes at each index $1 \le i \le |W|$. For any block B_i, the algorithm computes the frequency of each attribute as follows:

- For the first block B_1, it uses the cumulative count stored at position s to compute the frequency of the first $\ell-1$ attributes in B_1.
- The frequency of the ℓ-th attribute is computed by subtracting the sum of the other $\ell - 1$ counts from the block size s.
- For each subsequent block B_i ($i > 1$), the algorithm computes the difference between the sketch values at positions $i \cdot s$ and $(i-1) \cdot s$, which yields the frequency of the first $\ell - 1$ attributes in B_i.
- Again, the frequency of the ℓ-th attribute is inferred as above.

In each block, the algorithm compares the frequency of every attribute value p to its allowed range. If a single attribute falls outside the allowed range in any block, the algorithm terminates early and correctly returns **No**. Otherwise, if all blocks satisfy the fairness constraints, it returns **Yes**. Since the sketch correctly captures cumulative counts and the algorithm faithfully reconstructs the per-block frequencies and verifies them against the fairness bounds, it always returns the correct answer.

□

4 Continuous Stream Reordering

Consider **Query 2** again, there are three protected attribute values: C, A, H. Two distinct fairness requirements are specified:
Case 1: The requirement is $[2, 1, 2]$, meaning each block must contain 2 elements with value C, 1 with A, and 2 with H.
Case 2: The requirement is $[3, 1, 1]$.

A block is considered fair if it satisfies either of these criteria. Since the original block B_3 does not meet either requirement, 5 additional landmark items are read (Figure 3). The updated input stream, including these landmark items, now consists of 6 elements with value C, 7 with A, and 7 with H. We propose an algorithm, p-FairReOrder, which takes these 20 elements and rearranges them to maximize the number of uniquely fair blocks.

Let $n = |W| + |\mathcal{X}|$ and $InStm = [ib_1, ib_2, \ldots, ib_n]$, be the stream under consideration with $|W|$ regular and $|\mathcal{X}|$ landmark items. The objective is to permute the elements into a new sequence $OutStm = [ob_1, ob_2, \ldots, ob_n]$ to maximize the number of unique fair blocks. Denote by $fb(Stm)$ the number of unique fair blocks in stream Stm. Then the problem can be formulated to maximize $OutStm = \arg\max_{Stm} fb(Stm)$.

4.1 Algorithm p-FairReOrder

Algorithm p-FairReOrder has an outer loop that takes each valid combination of protected attribute counts and run a subroutine (Subroutine MaxReOrder) to reorder and create the maximum number of unique fair blocks with that. For **Query 2**, there are two such combinations for $[C, A, H]$: $[2, 1, 2]$ and $[3, 1, 1]$, and it repeats the subroutine for each and outputs the one that maximizes the number of fair unique blocks at the end.

We begin by introducing key concepts used in the design of the Subroutine MaxReOrder.

Definition 4.1 (Isomorphic Blocks and Isomorphic Block Count). A set of disjoint blocks is said to be *isomorphic* if each block contains the same ordered sequence of protected attribute values. The total number of such blocks in a stream is called the *Isomorphic Block Count (IBC)*. Under a given p-fairness constraint, if one isomorphic block satisfies the fairness condition, then all blocks in the set are considered fair.

Assume the input stream consists of ℓ distinct attribute values. Let $v_j \ge 1$ denote the number of elements with value $p \in \cup_{i=1}^{\ell} \mathcal{A}_i$ required by the fairness constraint, such that $\sum_{j=1}^{\ell} v_j = s$. Let V_j be the total number of elements of value p in the input stream. Then, the number of *isomorphic blocks* (IBC) that satisfy the given fairness requirement can be computed as:

$$\text{IBC} = \min\left\{\text{IBC}_j \mid 1 \le j \le \ell\right\}, \quad \text{where IBC}_j = \left\lfloor \frac{V_j}{v_j} \right\rfloor.$$

For example, in **Query 2**, **Case 1:**

$$\left\lfloor \frac{6}{2} \right\rfloor = 3, \quad \left\lfloor \frac{7}{1} \right\rfloor = 7, \quad \left\lfloor \frac{7}{2} \right\rfloor = 3.$$

Thus, the total number of fair isomorphic blocks is:

$$\text{IBC} = \min(3, 7, 3) = 3.$$

For **Case 2:**, $\text{IBC} = \min(2, 7, 7) = 2$

Definition 4.2 (Isomorphic Stream and Extended Isomorphic Stream). A stream is called an *isomorphic stream* if it is entirely composed of isomorphic blocks—that is, each block follows the same fixed composition pattern. If such a stream is extended by a prefix of this block pattern, the resulting stream is called an *extended isomorphic stream*. This additional segment is referred to as the *extended prefix (EP)*, and its length is called the *extended prefix length (EPL)*.

Given a stream of length n and a fairness requirement specifying a block size s, if $IBC \cdot s = n$, the stream can be fully partitioned into IBC isomorphic blocks, or it can be reordered into an isomorphic stream.

If $IBC \cdot s < n$, the remaining $n - IBC \cdot s$ items cannot form a complete isomorphic block. These leftover elements may form an extended prefix EP. The length of the extended prefix is given by:

$$EPL = \sum_{i=1}^{\ell} EP_i, \quad \text{where } EP_i = \min\left(v_i,\ V_i - IBC \cdot v_i\right)$$

Here, ℓ is the number of distinct values (e.g., demographic groups) in the protected attribute; v_i is the required count of group i per fair block (from the fairness requirement); V_i is the total number of elements from group i in the entire stream; EP_i is the number of group i elements assigned to the extended prefix; IBC is the number of complete isomorphic blocks.
Note: $EPL < s$, since the extended prefix cannot complete a full block on its own.

Consider the fairness requirement of $[2, 1, 2]$ for C, A, and H, where $C = 6, A = 7, H = 7$.

The block size is $s = 5$, and since $IBC = 3$, we calculate: $EP_1 = \min(2,\ 6 - 3 \cdot 2) = 0, EP_2 = \min(1,\ 7 - 3 \cdot 1) = 1, EP_3 = \min(2,\ 7 - 3 \cdot 2) = 1$, which gives: $EPL = EP_1 + EP_2 + EP_3 = 0 + 1 + 1 = 2$. So the extended prefix is $EP = [A, H]$ (any permutation is valid). To complete the isomorphic block pattern, use the remaining counts $v_i - EP_i$:

$$\begin{aligned} C: &\quad 2 - 0 = 2 &&\Rightarrow [C, C] \\ A: &\quad 1 - 1 = 0 &&\Rightarrow [] \\ H: &\quad 2 - 1 = 1 &&\Rightarrow [H] \end{aligned}$$

One valid isomorphic block pattern is: $[A, H, C, C, H]$. We can now construct the extended isomorphic stream:

$$[A, H, C, C, H,\ A, H, C, C, H,\ A, H, C, C, H,\ A, H]$$

This consists of three full isomorphic blocks and a prefix of length 2, achieving optimal block-wise fairness. The 3 leftover As can be appended at the end to complete the stream.

To summarize, Subroutine `MaxReOrder`(see Subroutine 3) begins by computing IBC, the number of isomorphic blocks that can satisfy the given fairness requirement. If no block can satisfy the constraint, the algorithm simply returns the original stream.

If $IBC \cdot s = n$, where s is the block size and n is the total stream length, then the entire output stream can be constructed using exactly IBC isomorphic blocks, forming an *isomorphic stream*. In this case, the algorithm returns the built isomorphic stream directly.

If $IBC \cdot s < n$, the remaining elements—those that do not fit into the full isomorphic blocks—can be used to form an *extended prefix EP*. The extended stream is constructed by concatenating the IBC isomorphic blocks with this extended prefix. This stream contains the maximum number of fair blocks possible under the given constraints (see Lemma 4.3). Any leftover elements that are not part of the isomorphic blocks or the extended prefix are appended to complete the stream.

Algorithm 3 Subroutine `MaxReOrder`

Input: Input stream $InStm$ of length $|W| + \mathcal{X}$, P-fair constraint $[v_1, \ldots, v_\ell]$
Output: Reordered stream $OutStm$
Compute the number of fair isomorphic blocks IBC
if $IBC < 1$ **then**
  return $InStm$
end if
if $IBC \cdot s = n$ **then**
  Construct IBC fair isomorphic blocks based on the given P-fair constraint
  $OutStm \leftarrow$ isomorphic stream formed by the IBC fair isomorphic blocks
  return $OutStm$
end if
Build the extended prefix EP
Complete the Isomorphic block pattern using $v_i - EP_i$
Construct an extended stream by concatenating the IBC fair isomorphic blocks and EP
$OutStm \leftarrow$ extended stream + remaining elements
return $OutStm$

4.2 Optimality and Running Time Analysis

Lemma 4.3 (Optimality of Isomorphic and Extended Isomorphic Streams).

Proof. Let the stream have length n and block size s. The number of possible unique blocks of size s in the stream is $n - s + 1$. We say a stream is *optimal* if all these blocks are *fair*—that is, it contains the maximum possible number of unique fair blocks.
An *isomorphic stream* consists of $\text{IBC} = \frac{n}{s}$ disjoint, identical fair blocks of size s. Because each overlapping block (i.e., any block of s consecutive elements) is composed of suffixes and prefixes of these identical fair blocks, every such block also satisfies the fairness condition. Therefore, all $n - s + 1$ unique blocks are fair, and the stream achieves optimality.
An *extended isomorphic stream* includes a carefully constructed prefix EP appending the isomorphic blocks. This prefix is designed such that any overlapping block between EP and the last isomorphic block is fair. Since all blocks within the isomorphic part are also fair, the entire stream contains $n - s + 1$ fair blocks. Thus, the extended isomorphic stream is also optimal.

□

Theorem 4.4 (Algorithm p-FairReOrder is optimal).

Proof. If $IBC \cdot s = n$, the stream is an isomorphic stream, and the result follows from Lemma 4.3.

If $IBC \cdot s < n$, the extended stream includes $IBC \cdot s + EPL$ elements, which contribute $(IBC \cdot s + EPL - s + 1 = (IBC - 1) \cdot s + EPL + 1$ fair blocks. Based on the construction method, the remaining elements cannot complete another fair block or contribute to overlaps of any fair block. Thus, this is the maximum number of fair blocks possible. Subroutine MaxReOrder finds that optimality for each valid combination of protected attribute values and Algorithm p-FairReOrder checks this for all possible combinations and outputs the one that maximizes it. □

Running time. Subroutine MaxReOrder takes at most $O(n)$ ($n = |W| + |\mathcal{X}|$) time. At the worst case, MaxReOrder needs to be run 2^ℓ times. Therefore, Algorithm p-FairReOrder takes In $O(n \times 2^\ell)$ time. The space complexity is dominated by queue size, $O(n)$.

5 Experimental Evaluations

5.1 Experimental Setup

5.1.1 Datasets. We have comprehensively tested continuous p-fairness queries on four publicly available datasets, as shown in Table 3 along with the number of records and the monitored $\mathcal{A}$.
The Hospital Admissions data [6] is a curated list of patient data from patients admitted to a tertiary care medical college and hospital in India over two years, from April 2017 to March 2019, during which there were over 15,000 admissions. The records are sorted by date of admission. The "AGE" attribute was binned into five different bins based on quantiles.
MovieLens 32M [14] describes movie ratings and tags from MovieLens, a movie recommendation service. It contains over 32 million ratings across 87,585 movies from over 200,000 users. The data is divided into four individual datasets, with the movies dataset describing the movie title and genres, which was joined over movieID with the ratings dataset containing the ratings for the movies. The genres were further preprocessed to form "moods" describing the theme of the movie, whether it has a dark, light, or neutral setting. The two moods $\mathcal{A}$ exclude the neutral mood.
Stock Market Dataset [25] contains historical daily prices for all tickers trading on NASDAQ up to April 2020. Specifically, Apple Inc.'s stocks were chosen to be monitored over time. The dataset was preprocessed to form the attribute percentage change in daily opening and closing prices, and then those percentages were binned into five bins of equal size rather than equal intervals. The "Volume" attribute is also binned into equal-sized bins of two and three categories. Twitter Sentiment [10] contains 16 million web-scraped tweets from April 17, 2009, to May 27, 2009, containing user ID and body of the tweet. The tweets were run through a sentiment analysis tool [23] that categorized the tweets into five different sentiments based on their polarity.
The datasets were preprocessed to drop rows with missing values for date, and the resulting dataset had records as mentioned in table 3.

From here on, we use the names *hospital*, *movies*, *tweets*, and *stocks* respectively for all the above-mentioned datasets.

Datasets used	$\mathcal{A}$	Records
Hospital Admissions	Gender, Outcome, Age	10,101
Twitter Sentiments	Sentiments	16,000,000
MovieLens 32M	Ratings, Moods (2,3)	33,832,161
Stock Market	% Change bins, Volume (2,3)	9,908

Table 3: Datasets

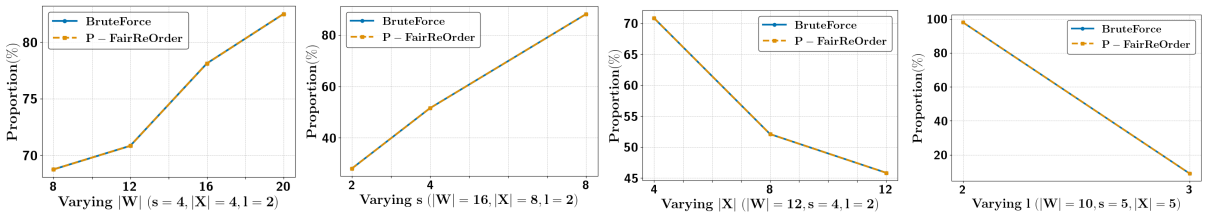
5.1.2 Simulating Streaming Environment. The streaming environment is simulated using Confluent Kafka Python (CKP), which is the official Python client library for Apache Kafka, developed and maintained by Confluent. Since the entire framework is built upon Python, CKP provides a robust and high-performance way to interact with Kafka topics, allowing production and consumption from them. A single-broker Kafka cluster and its Zookeeper peer are booted up inside a Docker container, providing a reproducible mini-cluster. The broker is exposed on an external listener so that it listens to a producer run on the host machine. The producer auto-creates a topic where it pushes the records fetched from the datasets in JSON format as events. It is modulated to send messages to the topic at a controlled rate to mimic a real-life streaming environment. The consumer built on CKP polls the broker, appends the latest message to an in-memory buffer, and keeps only the most recent $|\mathcal{W}|$ messages in the buffer, thus creating a sliding $\mathcal{W}$. The $\mathcal{W}$ is then sent for pre-processing and query processing. The CKP setup is lightweight, and the Docker container guarantees that the producer, broker, and consumer all share one virtual network.

5.1.3 Hardware and Software. All the experiments were ran on HP-Omen 16-n0xxx running Microsoft Windows 11 Home (v 10.0.26100) and Python (v 3.9.13) with an 8-core AMD Ryzen 7 6800H processor and 16 GB RAM. The Kafka cluster (confluentinc/cp-kafka:7.8.0) and its ZooKeeper peer (confluentinc/cp-zookeeper:7.8.0) are run on Docker (v 4.41.2). The confluent platform 7.8.0 boots up Apache Kafka (v 3.7.0) inside the container, simulating a streaming environment.

5.1.4 Baselines. We compare our framework against two baselines. One involves the pre-processing step, where our *Forward Sketch* method is compared against the *Baseline Sketch* or *BSketch*.
In *BSketch*, for every new window, the sketch is reconstructed from scratch; no information is retained from the previous iteration. Thus, there is no Sketch update, only Sketch Construction. The Sketch is also constructed from the back to the start of a given window as opposed to Forward Sketch.

Next we compare *P-FairReOrder* against *BruteForce* reordering. BruteForce reordering is an extremely naive and computationally and memory-intensive approach. It tries all possible combinations of the input stream and selects the one that produces the maximum number of *unique blocks*, resulting in a time complexity of the order $O(|W|!/p_1! \times p_2! \ldots \times p_l!)$. For this reason, even moderately sized input streams are infeasible for BruteForce to process.

5.1.5 Measures. In the following experiments, we perform qualitative and quantitative analysis of our proposed framework.
For qualitative analysis, we first compare the proportion of fair blocks produced by using BruteForce against P-FairReOrder. The



(a) Varying Window size **(b) Varying Block size** **(c) Varying Landmark items** **(d) Varying Cardinality**

Figure 6: Qualitative Analysis: Performance against BruteForce

proportion of the total number of fair blocks is measured before and after P-FairReOrder, given a particular P-Fairness constraint and fixed $|X|$. We varied $|X|$ separately to observe its effect on the proportion of fair blocks after using P-FairReOrder.

In quantitative analysis, we study the memory consumption, running time, and tail latency for the pre-processing and query processing steps. The memory consumption is the average memory consumed during any iteration, while the running time is the total time it takes to process a fixed number of windows. We also measure the proportion of total running time spent on different parts of the pre-processing and query processing steps, respectively. Tail latency focuses on the small number of cases where the running time is much higher than the average case. We also measure the throughput for query processing. Throughput is defined as the number of queries answered in one second.

In the ablation study, we look at the average running time of different parts of the pre-processing and query processing steps. Particularly, the pre-processing step consists of the Sketch Construction and Sketch Updation, and the query processing step consists of P-FairQP and P-FairReOrder for reorderable streams.

Finally, we compare the average running time and memory consumption of the baselines against our methods.

5.1.6 Parameters. The following parameter setting was maintained across all the experiments:

- Varying $|W|$ with fixed s, $|X|$ and l
- Varying s with fixed $|W|$, $|X|$ and l
- Varying $|X|$ with fixed $|W|$, s and l
- Varying l with fixed $|W|$, s and $|W|$

Except for subsections 6.4 and 6.5, we study all the datasets across all the experiments. Another exception is the *tweets* dataset not being used for varying l since it only has one $\mathcal{A}$.

5.2 Fairness Analyses

In this section, we compare our proposed `P-FairReOrder` with `BruteForce` to empirically evaluate our theoretical claim. We observe from figure 6 that `P-FairReOrder` returns the same number of unique P-fair blocks as that of `BruteForce` corraborating its optimality.

We vary window size $|W|$, block size s, and cardinality ℓ of different datasets and present the percentage of unique blocks that become fair after `P-FairReOrder` is applied on the original stream. In Figure 7 we present all the results. A general observation is, `P-FairReOrder` significantly improves the generalized P-Fairness of the data stream.

In general, with increasing window size $|W|$, as well as block size s, fairness substantially improves, which is intuitive, because a larger window or block size increases the likelihood of more number of blocks satisfying P-fairness for the same query. On the other hand, with increasing cardinality, the group fairness requirement becomes more complex, leading to less percentage of P-fair blocks.

In Figure 8, we vary $|\mathcal{X}|$ and measure the percentage of unique fair blocks after `P-FairReOrder` is applied. The results indicate that with increasing $|\mathcal{X}|$, the percentage of unique fair blocks increases - this is intuitive, since with increasing landmark items, it becomes more likely to satisfy query constraints, thereby increasing the overall percentage.

5.3 Query Processing Analyses

We first present the effectiveness of `p-FairQP` and `p-FairReOrder` varying pertinent parameters. In Section 5.3.4, we separately compare implemented baselines with our solutions, exhibiting that the baseline is $2-3$ order of magnitude slower than ours.

5.3.1 Throughput. In these experiments, we measure the throughput (per second) of the query processing algorithm `p-FairQP` by varying $|W|$, s, and ℓ, considering both scenarios: one where reordering involving landmark items is required using `p-FairReOrder`, and one where it is not. Figure 12 and 16 have these results and demonstrate that our algorithms exhibit very high throughout (as high as 40,000 at cases). As expected, `p-FairQP` incurs higher processing time when reordering is required, resulting in lower throughput, and vice versa. We also observe an inverse relationship between $|W|$ and throughput. This is indeed true as the window size increases $|W|$, this leads to processing higher number of blocks, leading to more time.

On the other hand, we observe the directly proportional relationship between s and throughput. Growing s decreases the number of blocks for a fixed $|W|$, thus decreasing the time taken for query processing and increasing the throughput.

The impact of ℓ is less pronounced. While higher cardinality may increase query time, evenly distributed constraints often lead to more fair blocks, reducing reordering effort. Thus, overall query time stays stable with moderate ℓ, and throughput remains nearly constant without `p-FairReOrder`.

5.3.2 Tail Latency. We report the 90th percentile tail latency of p-FairQP over 5,000 windows, both with and without p-FairReOrder, as shown in Figure 11 and Figure 15.
As shown in Figure 11a and Figure 15a, tail latency increases with larger $|W|$. For instance, the time to process a single query at times can exceed 10 ms when $W = 2000$. However, it is important to note that this represents the slowest 10% of cases and does not reflect the typical query processing time.
Figure 11b and Figure 15b illustrates the expected decrease in tail latency with increasing s, consistent with earlier observations.
Similarly, Figure 11c and Figure 15d shows the effect of increasing ℓ. Due to variations in the distribution of values across different $\mathcal{A}$, the tail latency does not exhibit a clear trend in this case.
Tail latency is significantly higher when p-FairReOrder is needed inside p-FairQP, as reordering introduces substantial overhead compared to query answering. In contrast, the absence of reordering results in minimal delays, even for the slowest 10% of queries.

5.3.3 Query Processing Time & Memory Consumption. Total query processing time for 5000 windows, as observed in Figure 9 and Figure 13 typically grows with increasing window size while decreases with increasing block size. The effect of landmarks is of interest here Figure 13d. As landmark increases, a larger number of windows are processed at once - reducing the frequency of calling p-FairReOrder. Also as was seen in Figure 8, increasing landmark increases the fairness. Fair blocks are much simpler and faster to build than unfair blocks in p-FairReOrder. Thus the total time to perform p-FairReOrder decreaes with increasing landmark. Memory consumption of p-FairQP is negligible as shown in Figure 14 and consistent to the sketch memory consumption presented later.

5.3.4 Query Processing Time compared to Baseline. We compare the runtime of p-FairReOrder with that of BruteForce across varying values of $|W|$, s, and $|\mathcal{X}|$, as shown in Figure 27. Due to the extreme inefficiency of the BruteForce algorithm, the streaming environment automatically terminated the process (connection closed due to polling for over 5 minutes) when attempting experiments with window and landmark sizes above 24 items or cardinality beyond 3. Figure 27a highlights the inefficiency of BruteForce. While its runtime is comparable to that of p-FairReOrder—remaining in the sub-millisecond range—for $|W| \leq 16$, it spikes sharply to over 10 seconds when $|W| = 20$, clearly demonstrating its exponential time complexity. BruteForce combinatorially search all possible reordering. As a result, it is far more sensitive to changes in s than p-FairReOrder, since larger s values lead to fewer blocks and a higher likelihood of achieving fairness sooner. This sensitivity is reflected in Figure 27b. The effect of increasing $|\mathcal{X}|$ mirrors that of increasing $|W|$ for both algorithms, as it effectively lengthens the input stream. However, the runtime growth in this case is more gradual. Overall, p-FairReOrder is more than 3 order of magnitude faster than its baseline counterpart.

BSketchQP is also compared with p-FairQP and p-FairQP always outperforms as shown in Figures 25b, 25d, 25f, 25h.

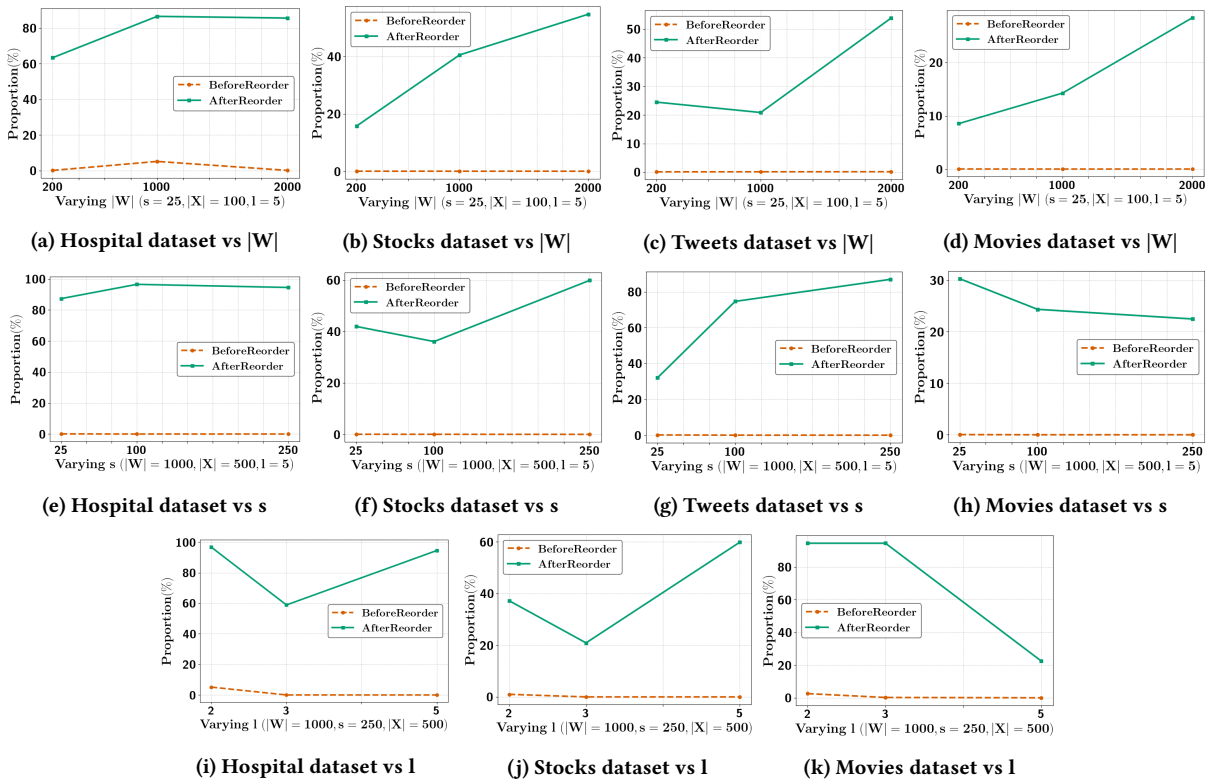
5.4 Preprocessing Analyses

We present the preprocessing analyses of our solutions, and later in the section we present the comparison with appropriate baselines.

5.4.1 Sketch Construction & Update. We report the preprocessing time for 5,000 windows on the stocks and movies datasets in Figure 17 and Figure 20, showing the time breakdown for sketch construction and updates, with and without p-FairReOrder resprectively(please note if p-FairReOrder is used the sketch needs to be constructed from scratch). Without p-FairReOrder, sketch construction time is negligible, as it is built once and updated in the remaining windows.
As shown in Figure 17a and 20a, preprocessing time increases with $|W|$ due to higher sketch construction cost, while update time remains unchanged. Varying s has minimal effect (Figure 17b, 20b), as sketch operations are block-independent. Larger ℓ slightly increases time, but the impact is minimal, demonstrating robustness to cardinality changes.
Interestingly, preprocessing can be slower without p-FairReOrder at times. With fixed $|X| = 500$, reordering is invoked infrequently, reducing sketch overhead. Moreover, batch memory access with landmark items speeds up updates, whereas loading fresh records per window without them introduces additional overhead. This phenomenon can be observed properly in Figure 20c, where as landmarks increase, total preprocessing time decreases.
5.4.2 Memory Requirement. Figure 18 and 21 shows average peak memory usage during preprocessing over 5,000 windows, reflecting the maximum memory used—including background processes—under varying parameters.
As expected, memory consumption of FSketch increases with $|W|$ (Figure 18a, 21a) due to larger sketches. It decreases with increasing s (Figure 18b, 21b), as larger blocks reduce the number of configurations needed during p-FairReOrder, lowering memory usage. Similarly, memory usage increases with $|\mathcal{X}|$, not due to sketch size, but because reordering longer streams in p-FairReOrder incurs greater background memory overhead. These experiments corroborate our theoretical analyses, exhibiting that propsoed FSketch is lean and bounded by window size.
5.4.3 Comparison with Baseline. We compare the preprocessing times of *BSketch* and *FSketch* in Figures 25a, 25c, 25e, 25g on the stocks dataset under varying parameters. *FSketch* consistently outperforms *BSketch* by an orders of magnitude, with the gap widening as window size increases (Figure 25a)—a result of Backward Sketch rebuilding the sketch from scratch for each window, unlike Forward Sketch, which supports efficient updates. As shown in Figures 25c, 25e and 25g, block size, landmarks, and cardinality have little impact, with minor variations attributed to platform-level fluctuations rather than the methods themselves. The small gap between memory consumption is shown Figure 26. As expected, with increasing window size and landmarks, the memory consumption increases, while with increasing block size, it decreases. With increasing cardinality, it stays stable.

5.5 Ablation Study

We perform an ablation study on the tweets dataset to measure the average runtime of key components—Sketch Construction, Sketch Update, p-FairReOrder, and p-FairQP—under varying $|W|$, s, and $|\mathcal{X}|$. As shown in Figure 24 and 23, p-FairReOrder is the most time-intensive, followed by Sketch Construction, while Sketch Update and p-FairQP remain consistently fast across all settings. Sketch construction scales with $|W|$ and is stable with respect to s and

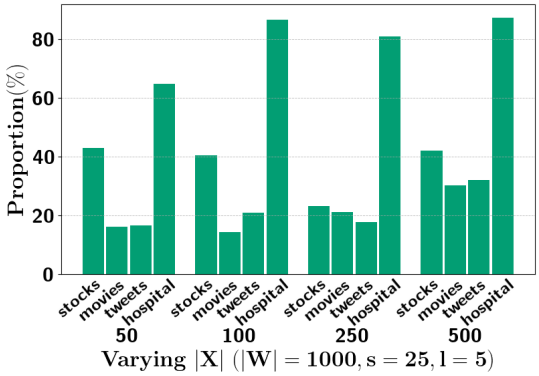


(a) Hospital dataset vs |W| (b) Stocks dataset vs |W| (c) Tweets dataset vs |W| (d) Movies dataset vs |W|

(e) Hospital dataset vs s (f) Stocks dataset vs s (g) Tweets dataset vs s (h) Movies dataset vs s

(i) Hospital dataset vs l (j) Stocks dataset vs l (k) Movies dataset vs l

Figure 7: Fairness analysis



(a) Varying Landmark items

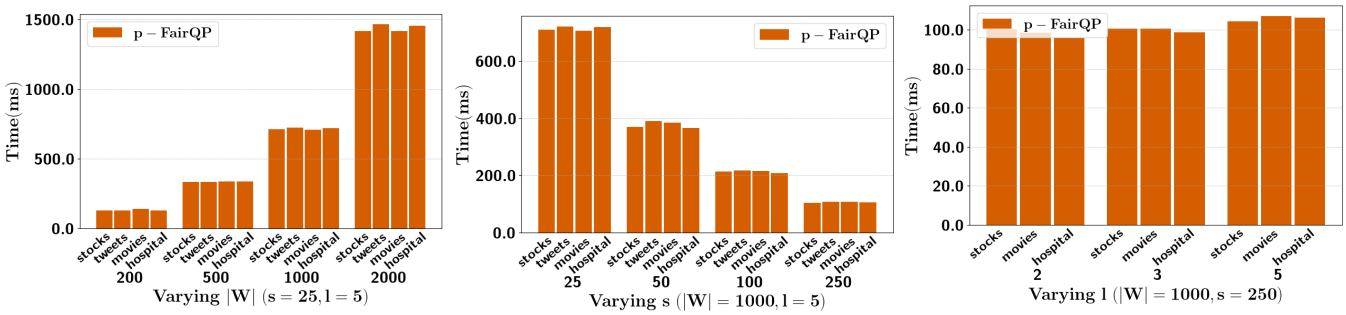
Figure 8: Effect of landmark on fairness

$|\mathcal{X}|$. `p-FairReOrder` scales linearly with $|W|$ and $|\mathcal{X}|$, and slightly decreases with larger s.

6 Related Work

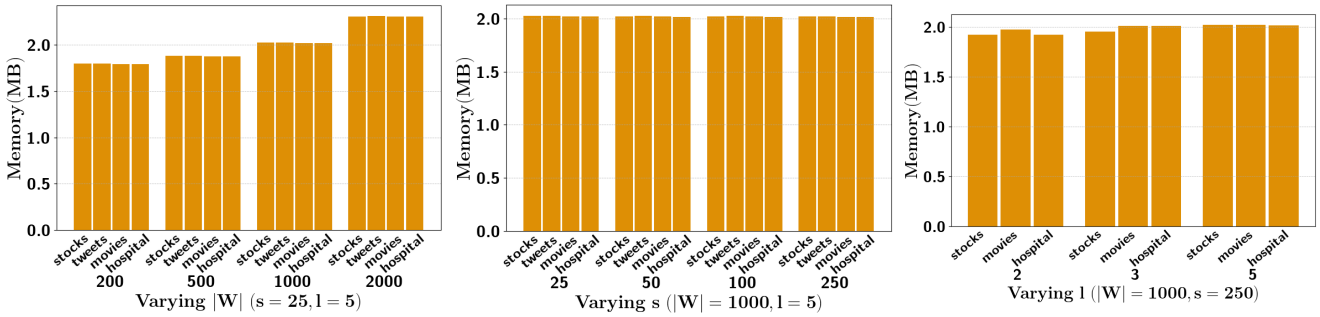
To the best of our knowledge, no related work studies the problems we propose.

Group Fairness. The study of *group fairness* originated in the context of classification, aiming to ensure that individuals from different demographic group - defined by protected attributes such as race, gender, or age - are treated equitably. Fairness at the group level is typically assessed using statistical criteria such as *demographic parity* [19] and *statistical parity* [15], which compare decision outcomes or error rates across groups.



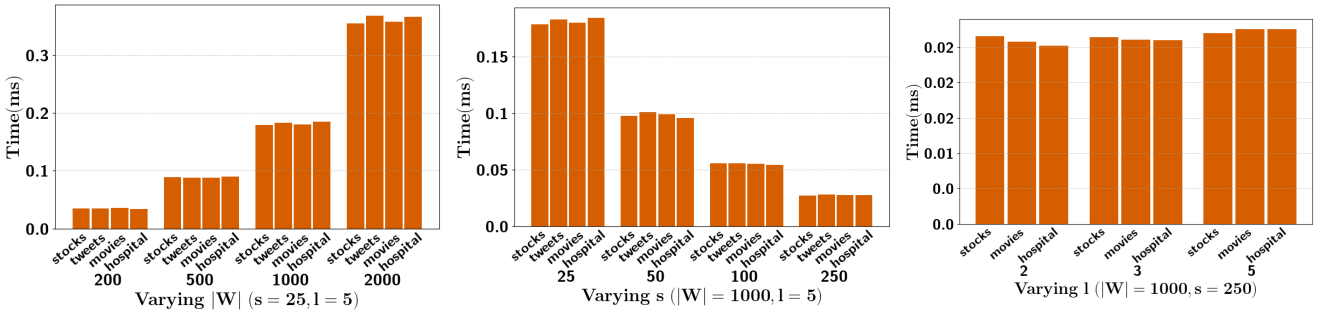
(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 9: Total Query processing running time for 5000 read-only windows



(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 10: Average Query processing Memory Consumption over 5000 read-only windows

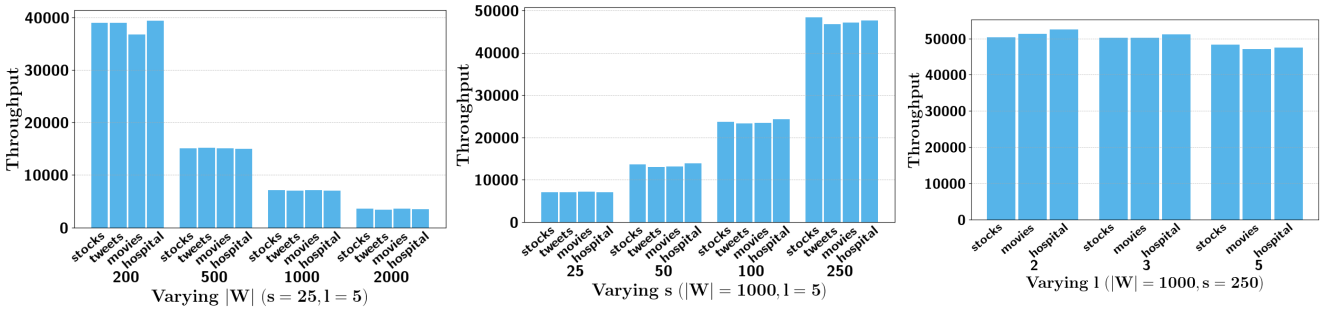


(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 11: 90 percentile query processing tail latency over 5000 read-only windows

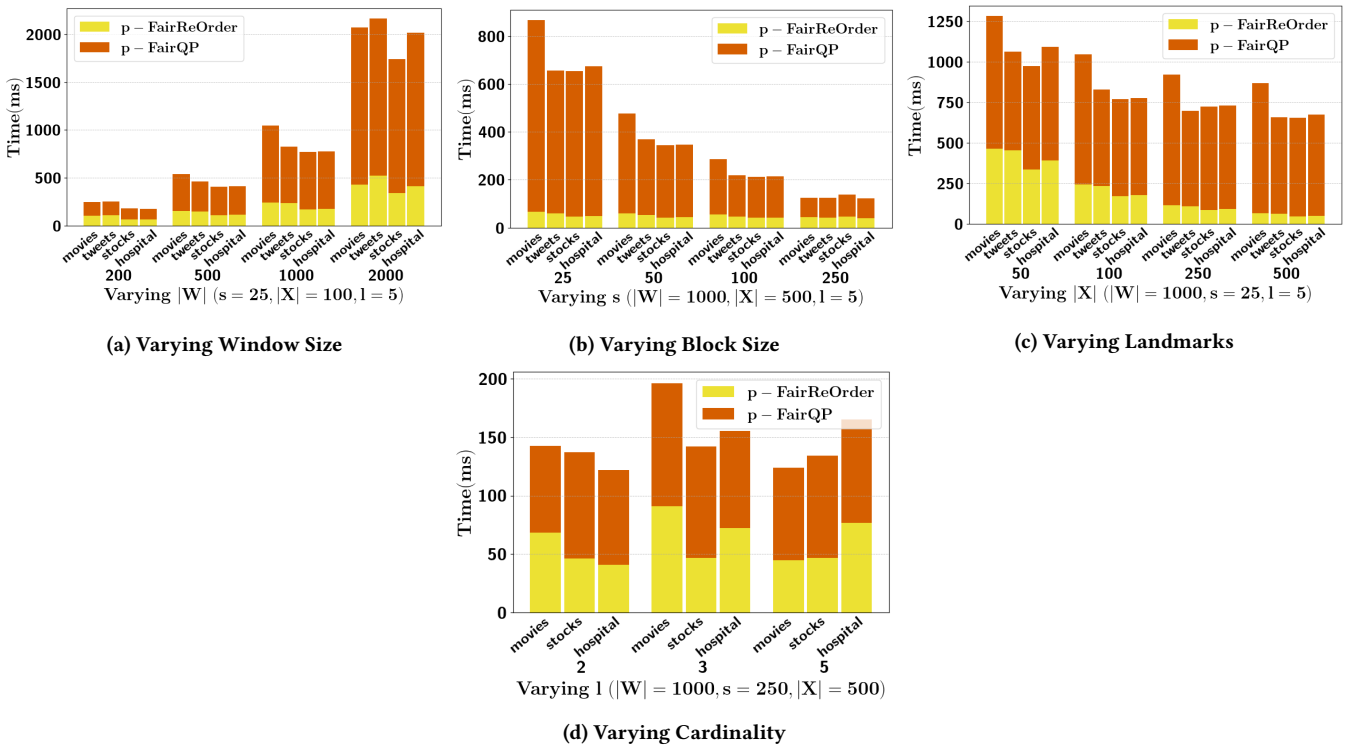
In *data management*, group fairness has been widely studied in tasks such as ranking and recommendation [16, 20, 28, 30, 32], typically focusing on a single multi-valued protected attribute. In these settings, statistical parity has been adapted into fairness constraints such as *top-k parity* [20], which ensures proportional representation within the overall top-k results. More stringent notions like *P-fairness* [32] have also been introduced to enforce fairness across every prefix of the ranked list, thereby ensuring balanced exposure throughout. In contrast, enforcing group fairness under *multiple* protected attributes is recognized as a computationally challenging problem [7, 18], due to the combinatorial complexity of satisfying fairness constraints across multiple intersecting subgroups.

We generalize the notion of P-fairness *to every block (instead of every prefix), and initiate its study in the streaming settings.*



(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 12: Read-only Throughput - Number of queries processed in one second



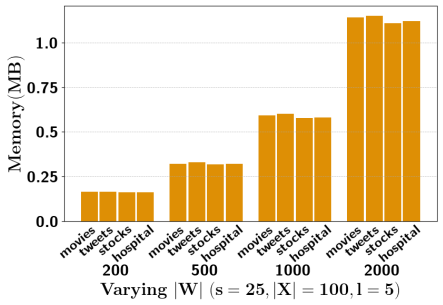
(a) Varying Window Size (b) Varying Block Size (c) Varying Landmarks

(d) Varying Cardinality

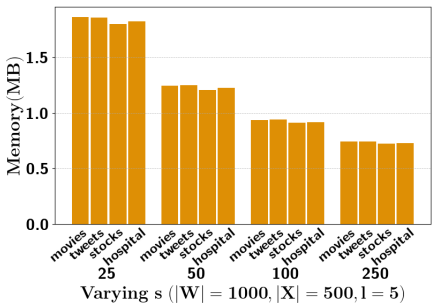
Figure 13: Total Query processing running time for 5000 windows

Continuous queries on data streams. Continuous Query Language (CQL) [1] extends SQL to support queries over nested and graph-structured data by introducing constraints such as path, value, type, and quantification, along with key and counting constraints for consistency. Golab and Özsu [13] propose memory-efficient algorithms for sliding window multi-joins in data streams. C-SPARQL [3] extends SPARQL with windowing to enable continuous queries over RDF streams. Lim et al. [22] present a duality model using spatial joins for efficient continuous query evaluation. Munagala et al. [24] reduce computation costs via adaptive filter sharing across queries.

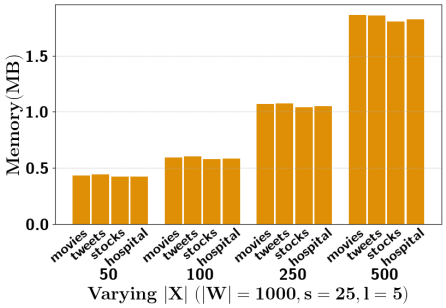
Maintaining data quality in streaming systems remains challenging due to their dynamic nature. Denial constraints [8], which define forbidden patterns, are commonly used for ensuring consistency. Recent work [26, 27] supports real-time constraint validation through window-based checks and schema adaptation, with frameworks like Apache Flink and Stream DaQ offering automated enforcement. Related works[21, 29] also study repair techniques to satisfy constraints by adding dummy values.



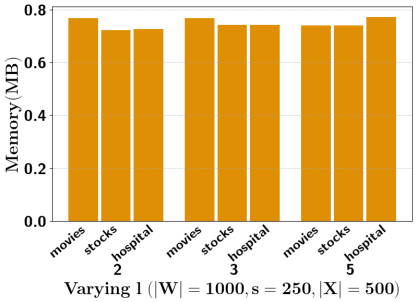
(a) Varying Window Size



(b) Varying Block Size

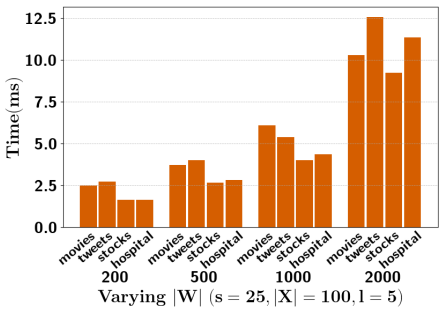


(c) Varying Landmark items

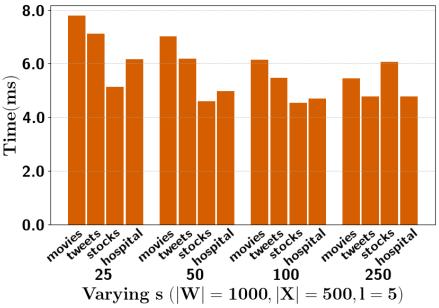


(d) Varying Cardinality

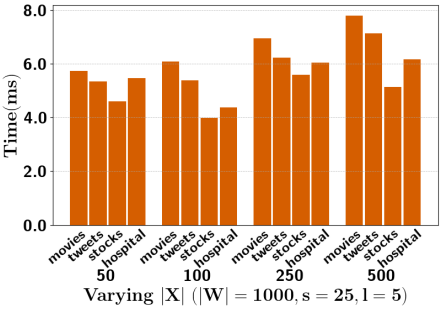
Figure 14: Average Query processing Memory Consumption over 5000 windows



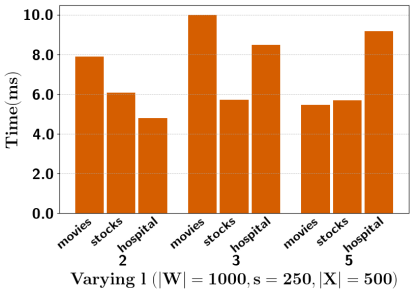
(a) Varying Window Size



(b) Varying Block Size

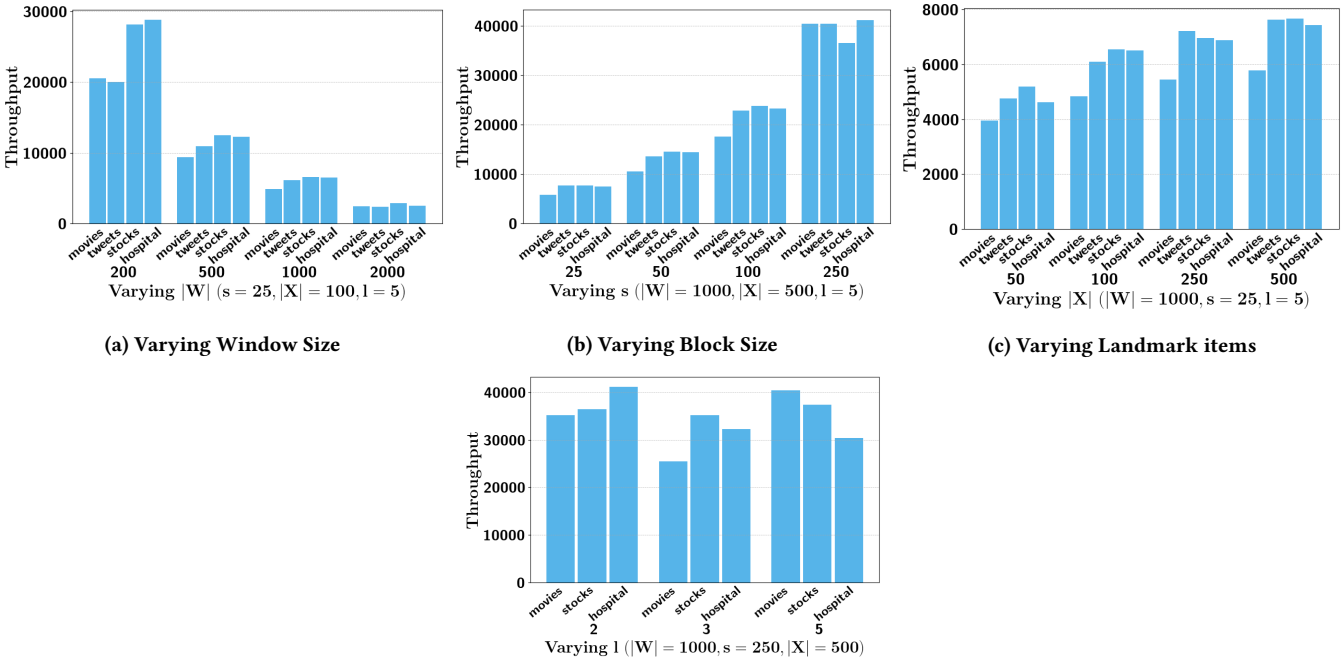


(c) Varying Landmark items



(d) Varying Cardinality

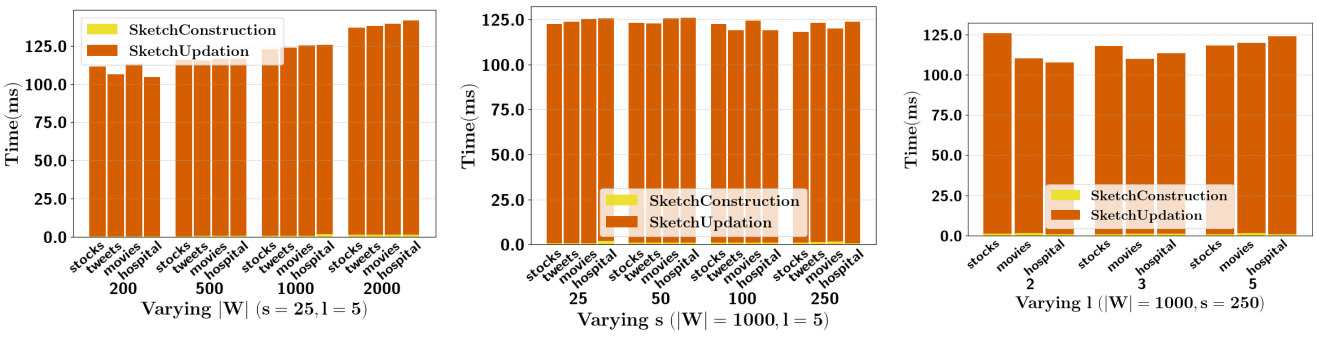
Figure 15: 90 percentile query processing tail latency over 5000 windows



(a) Varying Window Size (b) Varying Block Size (c) Varying Landmark items

(d) Varying Cardinality

Figure 16: Throughput - Number of queries processed in one second



(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 17: Total Preprocessing running time for 5000 read-only windows

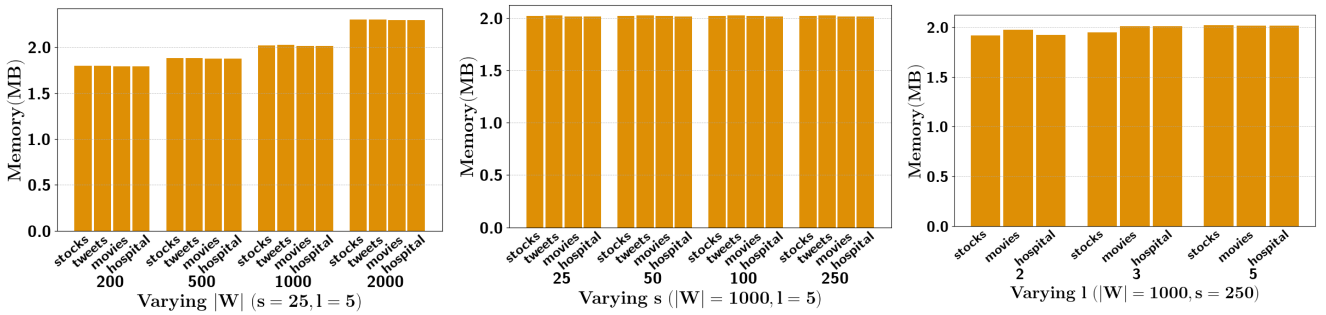
In contrast, we focus on enforcing fairness constraints within each block of the window, allowing only data rearrangement without modifying or imputing values. As a result, existing approaches on data quality and repair are not directly applicable to our setting.

Stream fairness. Research on data stream have recently started paying more attention to fairness. Baumeister et al. [4] checks fairness measures like demographic parity and equalized odds by estimating probabilities from live data streams. Wang et al. [31] address fairness in evolving data streams with *Fair Sampling over Stream (FS^2)*, a method that balances class distribution while considering fairness and concept drift. They also propose *Fairness Bonded Utility (FBU)*, a unified metric to evaluate the trade-off between fairness and accuracy.

In contrast, we propose continuous fairness queries over data streams ensuring fairness within each block of the sliding window.

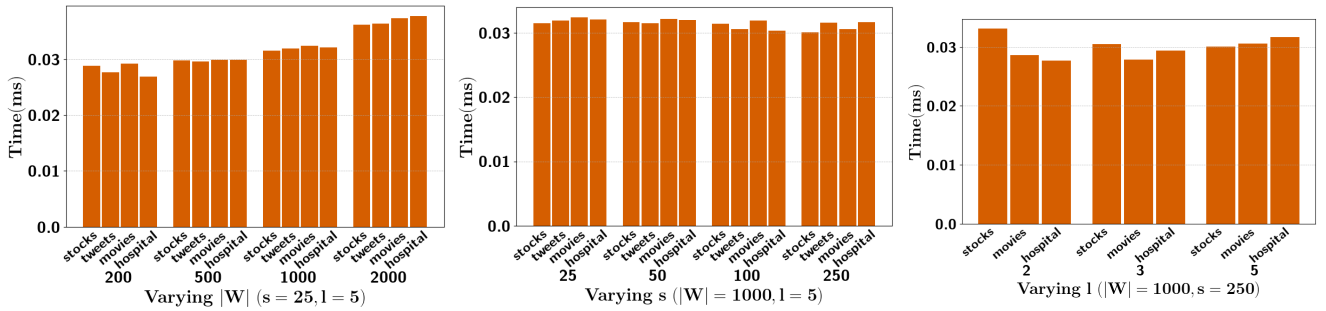
7 Conclusion

This work advances the state of fairness-aware data stream processing by introducing **continuous generalized P-Fairness** queries, which enforces group fairness at a finer granularity through block-level evaluation within sliding windows. This localized approach



(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 18: Average Preprocessing Memory Consumption over 5000 read-only windows



(a) Varying Window Size (b) Varying Block Size (c) Varying Cardinality

Figure 19: 90 percentile preprocessing tail latency over 5000 windows

captures fairness violations more precisely and flexibly than traditional window-level methods. We propose a comprehensive computational framework that includes an efficient sketch-based monitoring structure and an optimal stream reordering algorithm, both designed to support continuous fairness evaluation and reordering in real time. Together, these contributions enable scalable, low-latency, and principled enforcement of group fairness in high-throughput streaming systems, offering strong theoretical guarantees and practical utility for dynamic, fairness-critical applications.

We are currently investigating how to support P-fairness queries defined over multiple protected attributes, where fairness constraints must be satisfied independently for each attribute.

References

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2 (2006), 121–142.

[2] Shivnath Babu and Jennifer Widom. 2001. Continuous queries over data streams. *ACM Sigmod Record* 30, 3 (2001), 109–120.

[3] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2009. C-SPARQL: SPARQL for continuous querying. In *WWW*. ACM, 1061–1062.

[4] Jan Baumeister, Bernd Finkbeiner, Frederik Scheerer, Julian Siber, and Tobias Wagenpfeil. 2025. Stream-Based Monitoring of Algorithmic Fairness. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 15696)*. Springer, 60–81.

[5] Reuben Binns. 2020. On the apparent conflict between individual and group fairness. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 514–524. https://doi.org/10.1145/3351095.3372864

[6] Sandeep Chandra Bollepalli, Ashish Kumar Sahani, Naved Aslam, Bishav Mohan, Kanchan Kulkarni, Abhishek Goyal, Bhupinder Singh, Gurbhej Singh, Ankit Mittal, Rohit Tandon, Shibba Takkar Chhabra, Gurpreet S. Wander, and Antonis A. Armoundas. 2022. An Optimized Machine Learning Model Accurately Predicts In-Hospital Outcomes at Admission to a Cardiac Unit. *Diagnostics* 12, 2 (2022). https://doi.org/10.3390/diagnostics12020241

[7] Felix S Campbell, Alon Silberstein, Julia Stoyanovich, and Yuval Moskovitch. 2024. Query Refinement for Diverse Top-k Selection. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[8] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *Proc. VLDB Endow.* 6, 13 (2013), 1498–1509.

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

[10] FERNO2. 2020. training.1600000.processed.noemoticon.csv. https://www.kaggle.com/datasets/ferno2/training1600000processednoemoticoncsv.

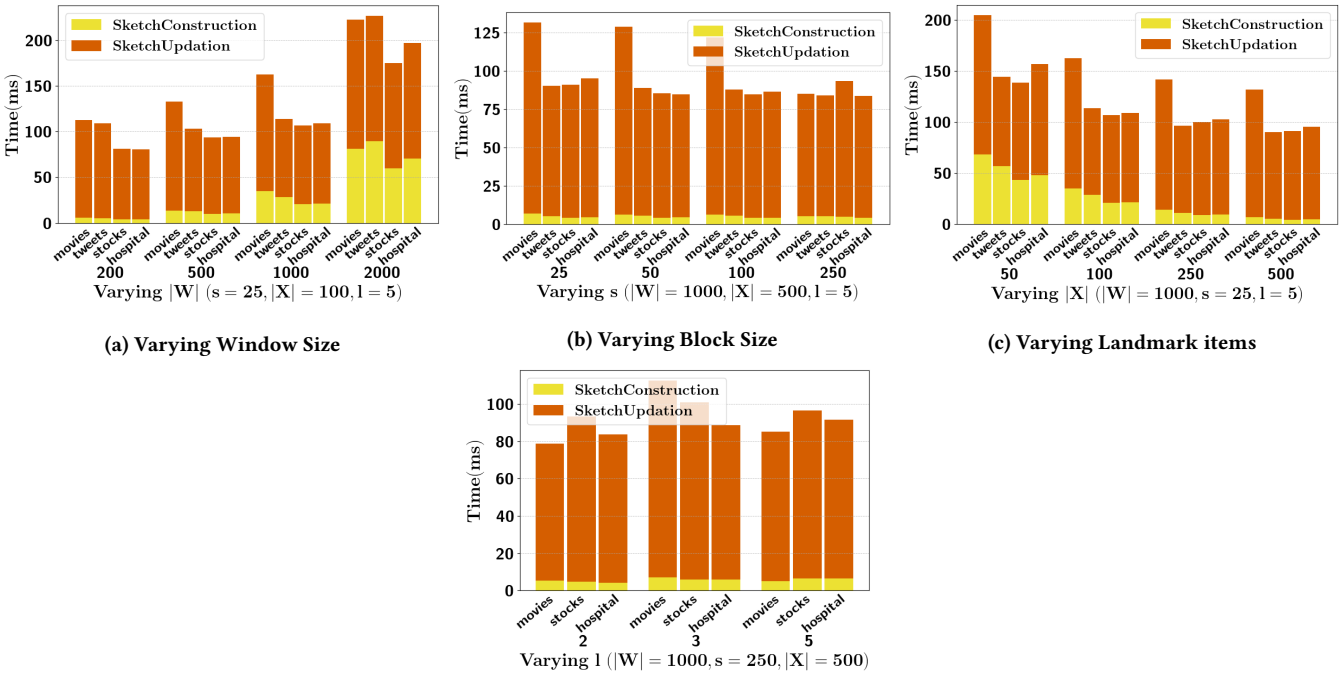
[11] David García-Soriano and Francesco Bonchi. 2021. Maxmin-fair ranking: individual fairness under group-fairness constraints. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore). Association for Computing Machinery, New York, NY, USA, 436–446. https://doi.org/10.1145/3447548.3467349

[12] Lukasz Golab and M Tamer Özsu. 2003. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings 2003 VLDB Conference*. Elsevier, 500–511.

[13] Lukasz Golab and M. Tamer Özsu. 2003. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *VLDB*. Morgan Kaufmann, 500–511.

[14] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (Dec. 2015), 19 pages. https://doi.org/10.1145/2827872

[15] Corinna Hertweck, Christoph Heitz, and Michele Loi. 2021. On the moral justification of statistical parity. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. 747–757.



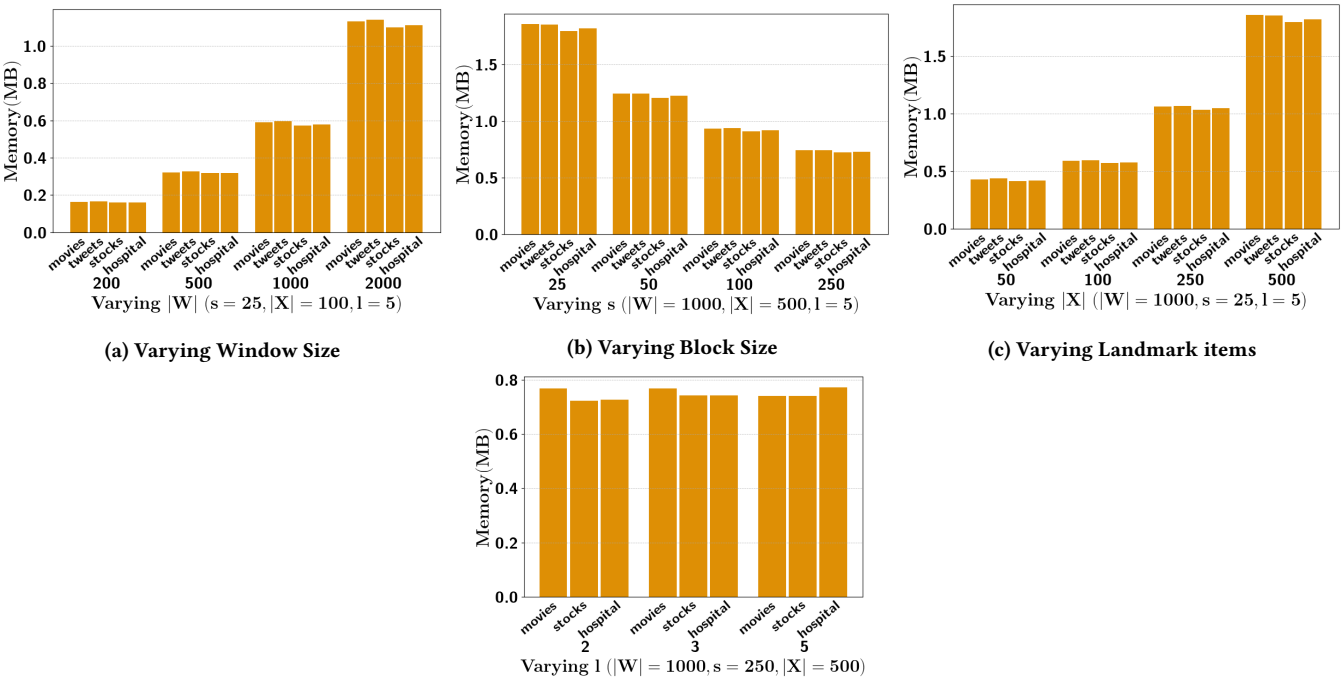
(a) Varying Window Size

(b) Varying Block Size

(c) Varying Landmark items

(d) Varying Cardinality

Figure 20: Total Preprocessing running time for 5000 windows



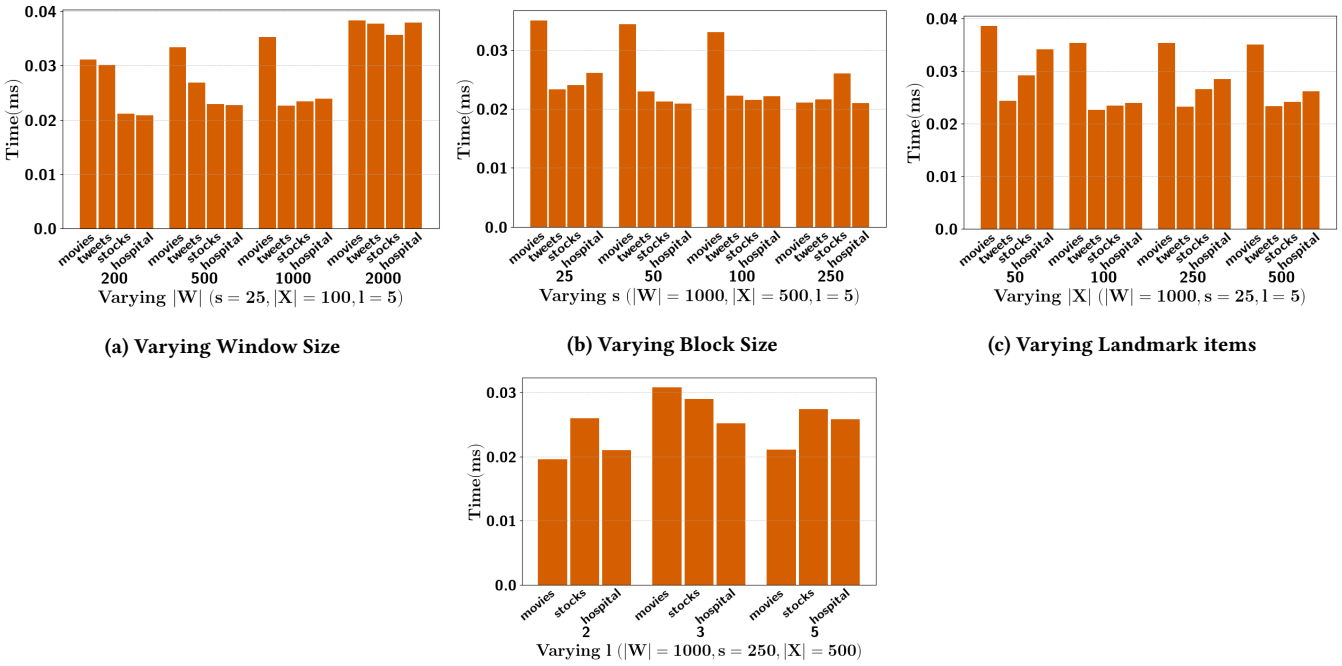
(a) Varying Window Size

(b) Varying Block Size

(c) Varying Landmark items

(d) Varying Cardinality

Figure 21: Average Preprocessing Memory Consumption over 5000 windows



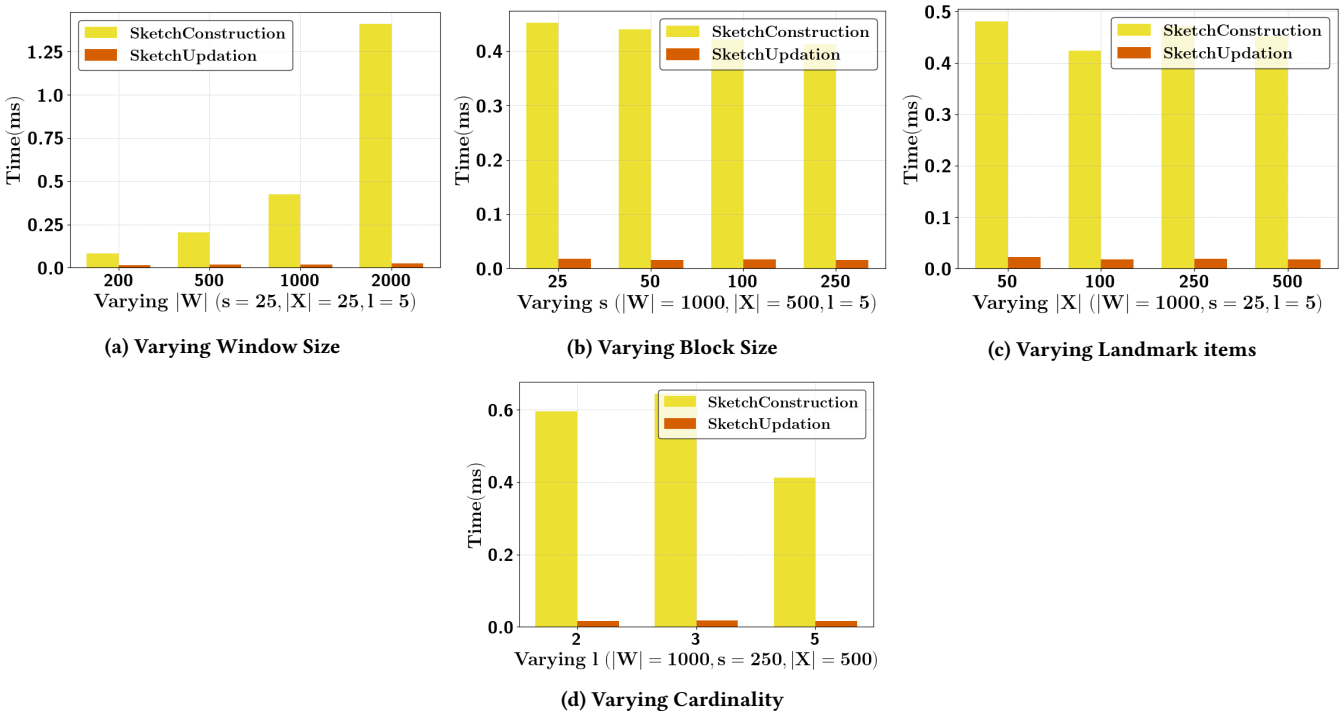
(a) Varying Window Size

(b) Varying Block Size

(c) Varying Landmark items

(d) Varying Cardinality

Figure 22: 90 percentile preprocessing tail latency over 5000 windows



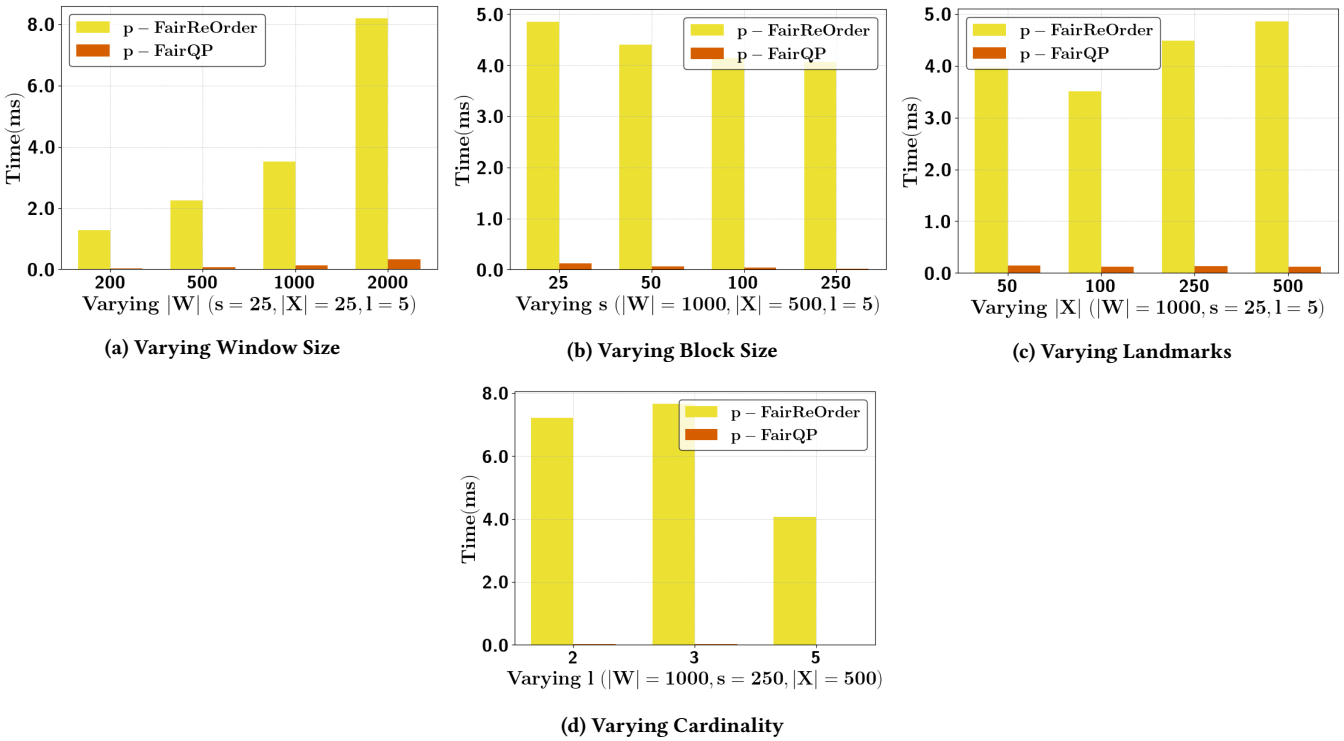
(a) Varying Window Size

(b) Varying Block Size

(c) Varying Landmark items

(d) Varying Cardinality

Figure 23: Average Sketch Construction vs Sketch Updation running time



(a) Varying Window Size **(b) Varying Block Size** **(c) Varying Landmarks**

(d) Varying Cardinality

Figure 24: Average p-FairReOrder vs p-FairQP running time

[16] Nyi Nyi Htun et al. 2021. Perception of fairness in group music recommender systems. In *IUI*. 302–306.

[17] Piotr Indyk. 2006. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM* 53, 3 (2006), 307–323.

[18] Md Mouinul Islam et al. 2022. Satisfying complex top-k fairness constraints by preference substitutions. *PVLDB* (2022).

[19] Zhimeng Jiang, Xiaotian Han, Chao Fan, Fan Yang, Ali Mostafavi, and Xia Hu. 2022. Generalized demographic parity for group fairness. In *International Conference on Learning Representations*.

[20] Caitlin Kuhlman and Elke Rundensteiner. 2020. Rank aggregation algorithms for fair consensus. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[21] Samuele Langhi, Angela Bonifati, and Riccardo Tommasini. 2025. Evaluating Continuous! eries with Inconsistency Annotations. (2025).

[22] Hyo-Sang Lim, Jae-Gil Lee, Min-Jae Lee, Kyu-Young Whang, and Il-Yeol Song. 2006. Continuous query processing in data streams using duality of data and queries. In *SIGMOD Conference*. ACM, 313–324.

[23] Steven Loria. 2018. textblob Documentation. *Release 0.15* 2 (2018).

[24] Kamesh Munagala, Utkarsh Srivastava, and Jennifer Widom. 2007. Optimization of continuous queries with shared expensive filters. In *PODS*. ACM, 215–224.

[25] Oleh Onyshchak. 2020. Stock Market Dataset. https://doi.org/10.34740/KAGGLE/DSV/1054465

[26] Vasileios Papastergios and Anastasios Gounaris. 2025. Stream DaQ: Stream-First Data Quality Monitoring. *CoRR* abs/2506.06147 (2025).

[27] Eduardo H. M. Pena, Fábio Porto, and Felix Naumann. 2024. Discovering Denial Constraints in Dynamic Datasets. In *ICDE*. IEEE, 3546–3558.

[28] Evaggelia Pitoura et al. 2022. Fairness in rankings and recommendations: an overview. *The VLDB Journal* (2022), 1–28.

[29] Shaoxu Song, Aoqian Zhang, Jianmin Wang, and Philip S. Yu. 2015. SCREEN: Stream Data Cleaning under Speed Constraints. In *SIGMOD Conference*. ACM, 827–841.

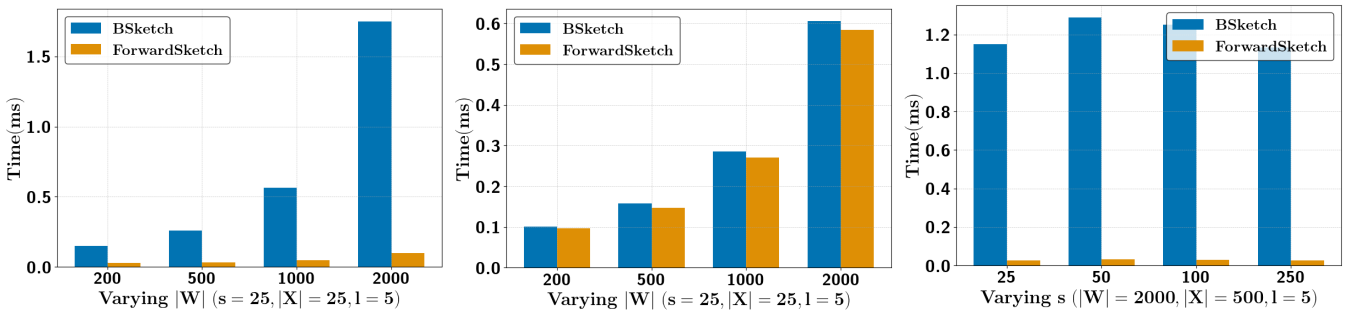
[30] Julia Stoyanovich, Bill Howe, and Hosagrahar Visvesvaraya Jagadish. 2020. Responsible data management. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[31] Zichong Wang, Nripsuta Saxena, Tongjia Yu, Sneha Karki, Tyler Zetty, Israat Haque, Shan Zhou, Dukka Kc, Ian Stockwell, Xuyu Wang, Albert Bifet, and Wenbin Zhang. 2023. Preventing Discriminatory Decision-making in Evolving Data Streams. In *FAccT*. ACM, 149–159.

[32] Dong Wei et al. 2022. Rank aggregation with proportionate fairness. In *SIGMOD*.

[33] Li Yan, Nerissa Xu, Guozhong Li, Sourav S. Bhowmick, Byron Choi, and Jianliang Xu. 2022. SENSOR: Data-driven Construction of Sketch-based Visual Query Interfaces for Time Series Data. *Proc. VLDB Endow.* 15, 12 (2022), 3650–3653.

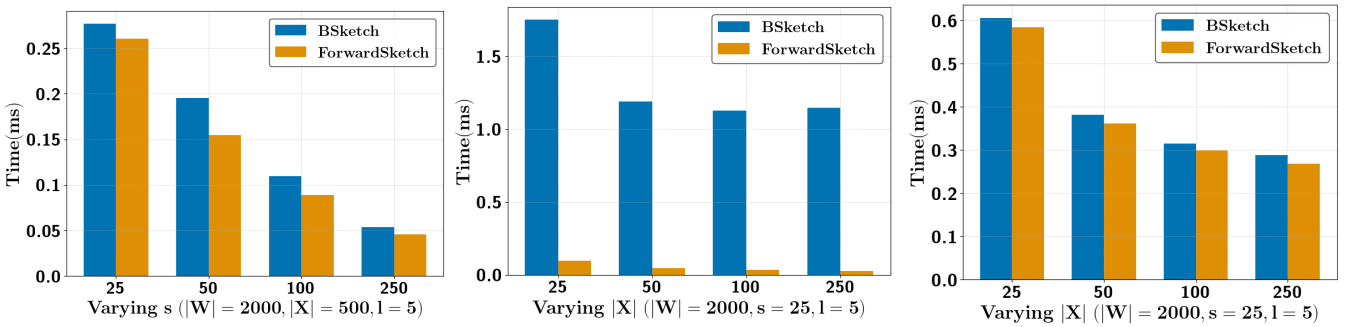
Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009



(a) Average preprocessing time for varying window sizes

(b) Average query processing time for varying window sizes

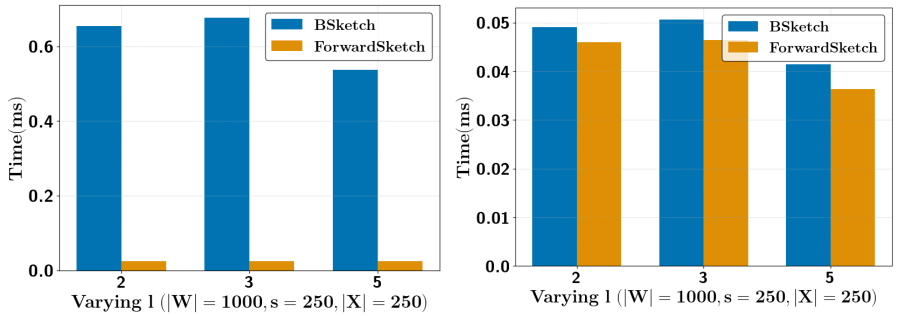
(c) Average preprocessing time for varying block sizes



(d) Average query processing time for varying block sizes

(e) Average preprocessing time for varying landmark items

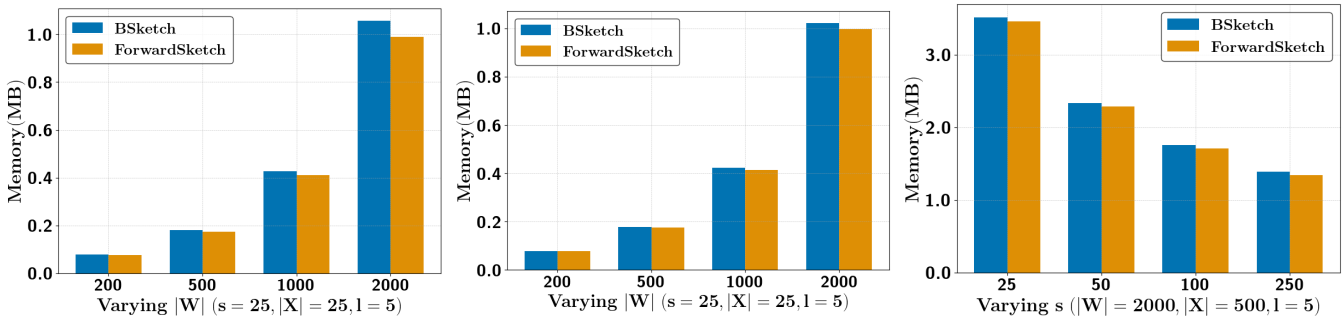
(f) Average query processing time for varying landmark items



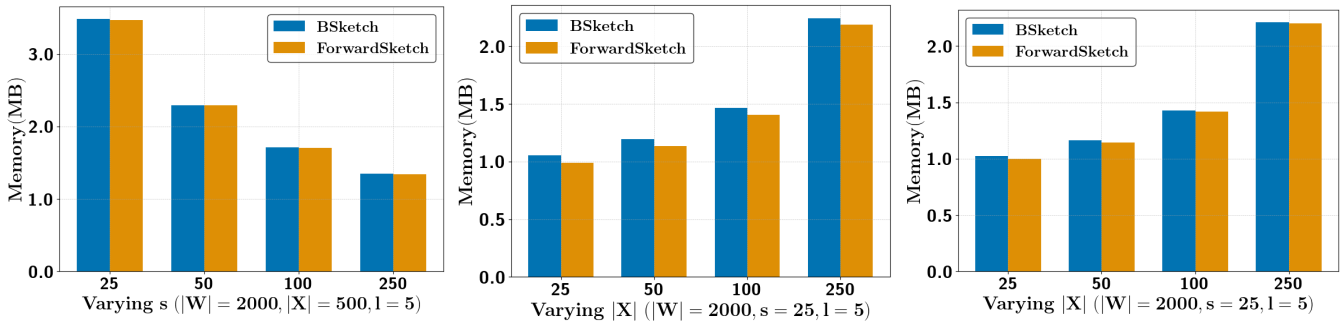
(g) Average preprocessing time for varying cardinality

(h) Average query processing time for varying cardinality

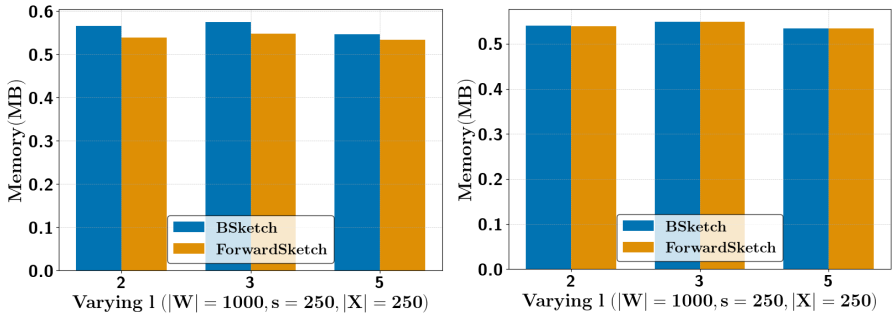
Figure 25: BSketch vs ForwardSketch Running Time across different parameters



(a) Average preprocessing peak consumption for varying window sizes (b) Average query processing peak consumption for varying window sizes (c) Average preprocessing peak consumption for varying block sizes

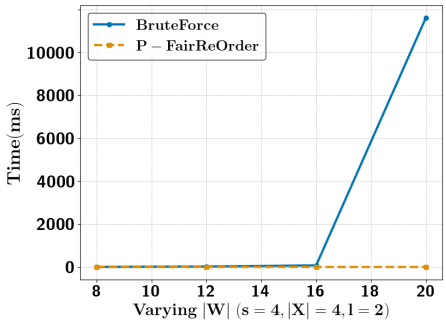


(d) Average query processing peak consumption for varying block sizes (e) Average preprocessing peak consumption for varying landmark items (f) Average query processing peak consumption for varying landmark items

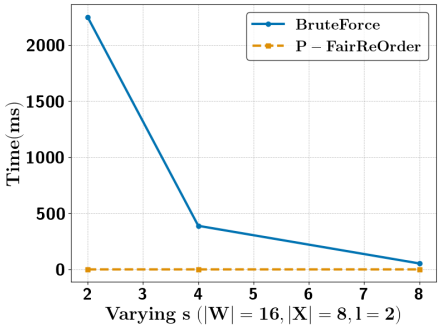


(g) Average preprocessing peak consumption for varying cardinality (h) Average query processing peak consumption for varying cardinality

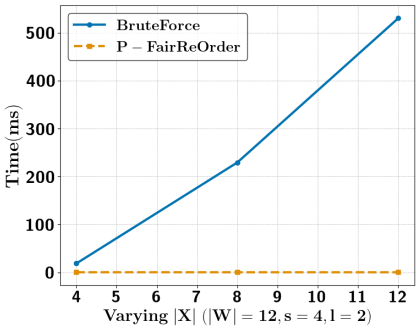
Figure 26: BSketch vs ForwardSketch Memory Consumption across different parameters



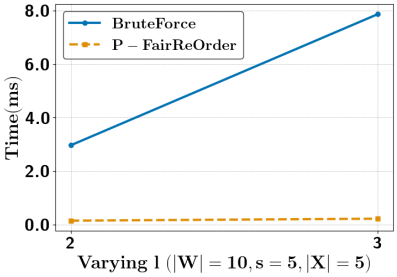
(a) Varying Window sizes



(b) Varying Block sizes

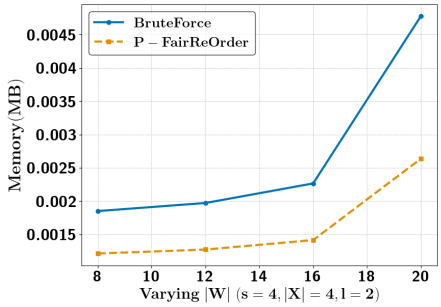


(c) Varying Landmark

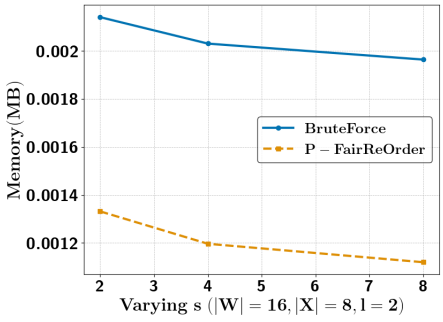


(d) Varying Cardinality

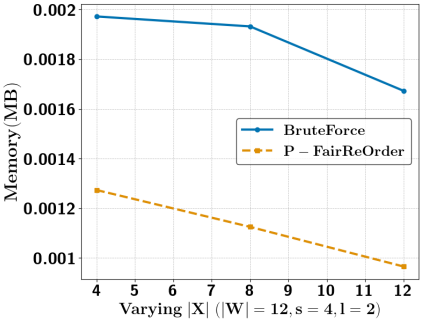
Figure 27: BruteForce vs p-FairReOrder Running Time across different parameters



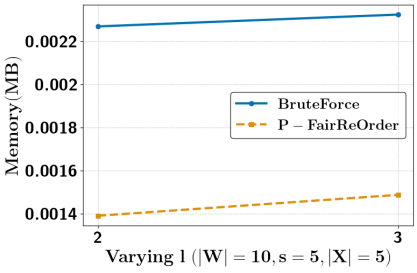
(a) Varying Window sizes



(b) Varying Block sizes



(c) Varying Landmark



(d) Varying Cardinality

Figure 28: BruteForce vs p-FairReOrder Memory Consumption across different parameters