Applied Intelligence
https://doi.org/10.1007/s10489-021-02614-w

# AFQN: approximate $Q_n$ estimation in data streams

**Italo Epicoco[1,2] · Catiuscia Melle[1] · Massimo Cafaro[1] · Marco Pulimeno[1]**

Accepted: 13 June 2021


**Abstract**

We present AFQN (Approximate Fast $Q_n$), a novel algorithm for approximate computation of the $Q_n$ scale estimator in a streaming setting, in the sliding window model. It is well-known that computing the $Q_n$ estimator exactly may be too costly for some applications, and the problem is a fortiori exacerbated in the streaming setting, in which the time available to process incoming data stream items is short. In this paper we show how to efficiently and accurately approximate the $Q_n$ estimator. As an application, we show the use of AFQN for fast detection of outliers in data streams. In particular, the outliers are detected in the sliding window model, with a simple check based on the $Q_n$ scale estimator. Extensive experimental results on synthetic and real datasets confirm the validity of our approach by showing up to three times faster updates per second. Our contributions are the following ones: (i) to the best of our knowledge, we present the first approximation algorithm for online computation of the $Q_n$ scale estimator in a streaming setting and in the sliding window model; (ii) we show how to take advantage of our UDDSKETCH algorithm for quantile estimation in order to quickly compute the $Q_n$ scale estimator; (iii) as an example of a possible application of the $Q_n$ scale estimator, we discuss how to detect outliers in an input data stream.

**Keywords** Data streams · $Q_n$ estimator · Sliding window model · Outliers

## 1 Introduction

In this paper we deal with the problem of computing approximately the $Q_n$ scale estimator when the input is a data stream. $Q_n$ is a *robust* measure of dispersion [17]; it is a rank-based estimator proposed by Rousseeuw and Croux, a statistic based on absolute pairwise differences which does not require location estimation.

It is worth recalling here that statistical robustness comes at a cost, since computing the $Q_n$ estimator is computationally expensive as we shall see. The challenge is therefore to design a streaming algorithm working in the sliding window model [6, 15], able to quickly provide an approximation for the $Q_n$ estimator for each item in the input data stream. Moreover, the quality of such an approximation must be high, with regard to the accuracy. We shall show how to approximately compute the $Q_n$ estimator by using our UDDSKETCH algorithm [7] for quantile estimation.

As an application, we deal with the problem of analyzing an input data stream to detect anomalies, also known as *outliers*. Formally, we will denote by $\sigma$ a data stream consisting of a sequence of $n$ items drawn from a universe $\mathcal{U}$. In general, depending on the specific application, items may be duplicated and may correspond to abstract or real entities, such as IP addresses, graph edges, points, geographical coordinates, numbers etc. A detailed survey discussing data streams and fundamental streaming algorithms is [15].

Given the input stream, we process its items using the sliding window model [6, 15]. In this model, the freshness of recent items is captured either by a *time window*, i.e., a temporal interval of fixed size in which only the most recent items are taken into account or by an *item window*, i.e. a window containing a predefined number of recent items; detection of outliers is strictly related to those items falling

✉ Massimo Cafaro
massimo.cafaro@unisalento.it

Italo Epicoco
italo.epicoco@unisalento.it; italo.epicoco@cmcc.it

Catiuscia Melle
catiuscia.melle@unisalento.it

Marco Pulimeno
marco.pulimeno@unisalento.it

[1] University of Salento, Via per Monteroni 73100 Lecce, Italy

[2] Euro-Mediterranean Centre on Climate Change Foundation, Via Augusto Imperatore 16, Lecce, Italy

Published online: 04 August 2021

in the window. The items in the stream become stale over time, since the window periodically slides forward.

We are provided with a potentially infinite input stream whose items are numbers, and our task is to determine the outliers. Distinguishing inliers and outliers is a notoriously difficult problem even in the simplest case of a static input dataset. An outlier is traditionally defined as an observation which markedly deviates from the other members of a dataset, and searching for outliers requires finding those observations which appear to be inconsistent with the rest of the data [11]. Therefore, outliers are often thought as anomalies; they may arise either because of human or instrument errors, fraudulent behaviour, changes or system's faults, natural deviations in populations, etc.

Detecting an outlier may indicate a system abnormal running condition such as an engine defect, an anomalous object in an image, an intrusion with malicious intension inside a system, a fault in a production line, etc. An outlier detection system accomplishes the task of monitoring data in order to reveal anomalous instances. A comprehensive list of outlier detection use-cases is given in [11].

Owing to the underlying nature of the input stream, outliers' detection becomes particularly challenging and relevant. A data stream is usually characterized by the high rate of items' arrivals and by its length, which may be unbounded. As an immediate consequence, processing the items of a stream to compute a function of the input is quite hard, given that an algorithm is only allowed a single pass over the data stream items. In particular, each item is only seen just once, and must be quickly processed and discarded. Typically, processing an item must be done in constant time. Another problem is strictly related to the potentially unbounded length of the stream, which implies that the data items can not be stored, making infeasible processing the input at a later time. Indeed, in many cases the task requires near real-time processing of the stream.

Determining the outlierness of an item centered in the current window requires computing a score based on the $Q_n$ estimator; we shall show that our AFQN algorithm is both fast and accurate and is therefore a valid alternative to the exact but computationally expensive $Q_n$ estimator.

To recap, our contributions are the following ones:

- to the best of our knowledge, we present the first approximation algorithm for online computation of the $Q_n$ scale estimator in a streaming setting and in the sliding window model;
- we show how to take advantage of our UDDSKETCH algorithm for quantile estimation in order to quickly compute the $Q_n$ scale estimator;
- as an example of a possible application of the $Q_n$ scale estimator, we discuss how to detect outliers in an input data stream.

This paper is organized as follows. In Section 2, we introduce the $Q_n$ estimator and discuss related work. We describe our AFQN algorithm in Section 3. As an application, we present in Section 4 a robust statistical approach to outlier detection. Section 5 provides extensive experimental results. Our conclusions are drawn in Section 6.

## 2 Related work

The $Q_n$ estimator is a robust statistical method for univariate data. Given a set $\{x_1, x_2, \ldots, x_n\}$, the $Q_n$ statistic was initially defined by its authors [17] as

$$Q_n = 2.2219 \left\{ \left| x_i - x_j \right| ; i < j \right\}_{(k)} \tag{1}$$

where $k \approx \binom{n}{2}/4$ and the notation $\{\cdot\}_{(k)}$ denotes computing the $k$th order statistics on the set. However, the authors slightly modified the definition in (1) by taking into account that

$$\binom{h-1}{2} + 1 \leq k \leq \binom{h}{2}, \tag{2}$$

where $h = \lfloor n/2 \rfloor + 1$. The final definition is

$$Q_n = d_n \, 2.2219 \left\{ \left| x_i - x_j \right| ; i < j \right\}_{(k)}, \tag{3}$$

where $k = \binom{h}{2}$ and $d_n$ is a correction factor which depends on $n$. From a statistical perspective, the breakdown point of $Q_n$ is 50%, i.e., the estimator is robust enough to counter the negative effects of almost 50% large outliers without becoming extremely biased. In addition, its Gaussian efficiency is about 82%, i.e., it is an efficient estimator since it needs fewer observations than a less efficient one to achieve a given performance. To better understand how efficient the $Q_n$ estimator is, it is worth recalling here that the MAD (Median Absolute Deviation about the median of the data) estimator [9] provides an efficiency of only about 36%.

Consider a static dataset consisting of $n$ items. Computing the $Q_n$ estimator naively requires determining the set of the absolute pairwise differences, whose size is quadratic in $n$. Then, the differences must be sorted in order to determine the $k$th order statistic. Therefore, the naive approach requires in the worst case $O(n^2 \lg n)$ time. A slightly better algorithm requires $O(n^2)$ time in the worst case, by using the SELECT algorithm [1] which is linear in the input size in the worst case; again, the input size refers to the size of the set of the absolute pairwise differences. Since the SELECT algorithm is only of theoretical interest, selecting

the $k$th order statistic is usually done with the QUICKSELECT algorithm [8, 10] which is extremely fast. However, QUICKSELECT is linear in the input size only on average (expected computational complexity).

Croux and Rousseeuw [5] proposed the first offline algorithm with worst case complexity $O(n \lg n)$, taking advantage of an algorithm designed by Johnson and Mizoguchi [12] to determine the $k$th order statistic in a matrix which is required to have nonincreasing rows and columns, of the form

$$U = X + Y = \left\{x_i + y_j; 1 \leq i, j \leq n\right\}, \tag{4}$$

where the vectors $X$ and $Y$ are sorted using $O(n \lg n)$ time in the worst case. The key idea is to use the matrix $U$ of order $n$ without actually computing all of its entries, since this would require $O(n^2)$ time. This is coupled with a pruning strategy which allows discarding those numbers that can not be the $k$th order statistic.

In order to reduce the problem of computing the $Q_n$ estimator to the problem of selecting an order statistic using the matrix $U$, Croux and Rousseeuw noted that

$$\begin{aligned} &\left\{\left|x_i - x_j\right|; i < j\right\}_{(k)} \\ &= \left\{x_{(i)} - x_{(n-j+1)}; 1 \leq i, j \leq n\right\}_{(k^*)}, \end{aligned} \tag{5}$$

where $k^* = k + n + \binom{n}{2}$.

In the previous equation, $x_{(1)} \leq \ldots \leq x_{(n)}$ are the sorted observations (we recall that $x_1, \ldots, x_n$ are the unsorted observations). It follows that, defining the vectors $X = \{x_{(1)}, \ldots, x_{(n)}\}$ and $Y = \{-x_{(n)}, \ldots, -x_{(1)}\}$, Croux and Rousseeuw can apply the Johnson and Mizoguchi algorithm to the matrix obtained considering those observations whose indexes are such that $1 \leq i, j \leq n$:

$$U = X + Y = \left(x_{(i)} - x_{(n-j+1)}\right), \ 1 \leq i, j \leq n. \tag{6}$$

A minor difference is that the Johnson and Mizoguchi algorithm requires that the vectors $X$ and $Y$ be in nonincreasing order; Croux and Rousseeuw use, instead, the nondecreasing order.

The first algorithm working in a streaming setting has been proposed by [16]. The algorithm works in the sliding window model; let $s$ be the window size. The window is initially empty, and incoming observations from the stream are added to the window until the window reaches its size. Once the window is full, a new incoming observation triggers a window's *update*: the new observation is added to the window, and the oldest observation is removed from it.

In order to compute the $Q_n$ estimator during a window's update, the algorithm reuses the same consideration of Croux and Rousseeuw: given $X = \{x_1, \ldots, x_s\}$, $k' = \binom{\lfloor s/2 \rfloor + 1}{2}$ and $k = k' + s + \binom{s}{2}$, it holds that

$$\left\{\left|x_i - x_j\right|, i < j\right\}_{(k')} = \left\{x_{(i)} - x_{(s-j+1)}, 1 \leq i, j \leq s\right\}_{(k)}. \tag{7}$$

It follows that in this case the algorithm must compute the $k$th order statistic of $U = X + (-X)$. The algorithm maintains a buffer $\mathcal{B}$ of size $b = O(s)$, storing matrix entries $u_{(k-\lfloor (b-1)/2 \rfloor)}, \ldots, u_{(k+\lfloor b/2 \rfloor)}$, centered on the $k$th order statistic. The buffer is initially populated using a variant of the Croux and Rousseeuw algorithm. The implementation relies on the use of AVL trees, respectively for $X$, $-X$ and the buffer $\mathcal{B}$. Insertion and deletion of an item in the trees triggers a computation of the new position of the $k$th order statistic in $\mathcal{B}$. If the $k$th order statistic is no longer in $\mathcal{B}$, then $\mathcal{B}$ is recomputed using the offline algorithm of Croux and Rousseeuw. In the worst case, [16] requires $O(s \lg s)$ time. The authors prove that, "for a constant signal with stationary noise, the expected amortized time per update is $O(\lg s)$". However, it is worth remarking here that, in order to achieve this expected amortized time, the authors assume that the rank of each data point in the set of all data points is equiprobable, which of course is not always the case.

In [2] we presented FQN (Fast $Q_n$), a novel streaming algorithm working in the sliding window model for computing the $Q_n$ estimator with worst case $O(s)$ running time, where $s$ is the window's size. FQN outperforms [16] with regard to speed and does not assume anything related to the underlying distribution of the input stream. Our algorithm maintains two different permutations of the current window: the former is the permutation related to the actual order of arrival of the observations from the stream, the latter is instead the sorted permutation. Maintaining the sorted permutation can be done in $O(s)$ by mimicking the way InsertionSort [4] inserts an item. Then, in order to compute the $k$th order statistic of the absolute pairwise differences, we cleverly reuse, adapting it to our context, an algorithm by Mirzaian and Arjomandi [14]. This algorithm determines the $k$th order statistic in $O(s)$ worst case time.

## 3 The AFQN algorithm

Our AFQN algorithm dynamically maintains and processes consecutive windows arising from the input stream. Let $W$ be our sliding window and $x_i$ the $i$th item in the input stream. We let the size of $W$ be $s = 2w + 1$, where $w$ is the semi-window size.

Besides the window, AFQN exploits a sketch data structure provided by our recent UDDSKETCH algorithm [7] for quantile estimation. The sketch is used to insert and remove as needed the differences required to compute the $Q_n$ estimator. Formally, we recall that given a set $\{x_1, x_2, \ldots, x_n\}$, we need a clever way to compute the $k$th order statistic of the set $\left\{\left|x_i - x_j\right|; i < j\right\}$ where $k = \binom{\lfloor n/2 \rfloor + 1}{2}$. In particular, the size of the set of differences is $(n \times (n-1))/2 = O(n^2)$.

When the algorithm starts, the window $W$ is empty. We insert the items in $W$ one at a time; after inserting $s$ items, the window is full. Each time we insert an item $x_i$, we also insert its corresponding $i-1$ differences with the previous items into the sketch. Therefore, for each item we insert at most $O(s)$ differences, so that when the window is full for the first time the sketch contains $(s \times (s-1))/2 = O(s^2)$ differences.

Once the first window has been processed, each time a new item arrives from the stream the window and the sketch are updated as follows. The algorithm processes the stream in windows $W =< x_{i-2w}, \ldots, x_i >$ of size $s$ (implemented as a circular buffer). The window slides one item ahead when a new item $x_{i+1}$ arrives from the input stream, we remove from $W$ the least recent (in the temporal sequence of item arrivals) item $x_{i-2w}$ and we also delete the $s-1$ differences between $x_{i-2w}$ and the other $s-1$ items in $W$ from the sketch. We then proceed inserting the new item $x_{i+1}$ into $W$ along with the $s-1$ differences between $x_{i+1}$ and the other $s-1$ items in $W$ into the sketch.

As explained, handling a new item's arrival only requires $O(s)$ sketch insertions and deletions, so that we avoid computing all of the $O(s^2)$ differences each time a new item arrives from the input stream. In order to compute the $k$th order statistic related to the current window $W$, we simply query the sketch determining the quantile corresponding to $k = \binom{\lfloor s/2 \rfloor + 1}{2}$. Figure 1 depicts how the algorithm works, whilst Algorithm 5 refers to the pseudocode for AFQN.

We now discuss in details our UDDSKETCH algorithm.

UDDSKETCH is based on the same data structure used by the DDSKETCH algorithm [13]. This data structure supports the following operations: inserting an item, deleting an item, collapsing the data structure if required and querying it for quantile estimation. However, DDSKETCH guarantees good accuracy only for selected input distributions, whilst our algorithm provides better accuracy for almost all of the possible input distributions. We achieved this result by engineering a new collapsing procedure which is able to uniformly distribute the error committed. The accuracy is defined as follows.

**Definition 1** $q$-quantile. Denoting with $S$ a multi-set of size $n$ defined over $\mathbb{R}$ and with $R(x)$ the rank of an item $x$, i.e., the number of items in $S$ which are smaller than or equal to $x$, then the lower (respectively upper) $q$-quantile item $x_q \in S$ is the item $x$ whose rank $R(x)$ in $S$ is $\lfloor 1 + q(n-1) \rfloor$ (respectively $\lceil 1 + q(n-1) \rceil$) for $0 \leq q \leq 1$.

As an example, $x_0$ and $x_1$ are respectively the minimum and maximum item in $S$, whilst $x_{0.5}$ corresponds to the median. Relative accuracy is defined as follows.

**Definition 2** Relative accuracy. An item $\tilde{x}_q$ is an $\alpha$-accurate $q$-quantile if, for a given $q$-quantile item $x_q \in S$, $|\tilde{x}_q - x_q| \leq \alpha x_q$. A sketch data structure is an $\alpha$-accurate $(q_0, q_1)$-sketch if it can output $\alpha$-accurate $q$-quantiles for $q_0 \leq q \leq q_1$.

The sketch data structure is a set of buckets. In the sequel, we shall assume without loss of generality that the input consists of items $x \in \mathbb{R}_{\geq 0}$ (owing to the fact that the absolute differences are greater than or equal to zero; nevertheless, the algorithm can also handle negative values: this requires using another sketch in which an item $x \in \mathbb{R}_{<0}$ is handled by inserting $-x$). The algorithm must be initialized using two input parameters, $\alpha$ and $m$. The former is the user's defined accuracy, and the latter is the maximum number of buckets that can be used. If inserting an item requires adding a new bucket and the number of buckets exceeds $m$, then a collapsing procedure is executed in order

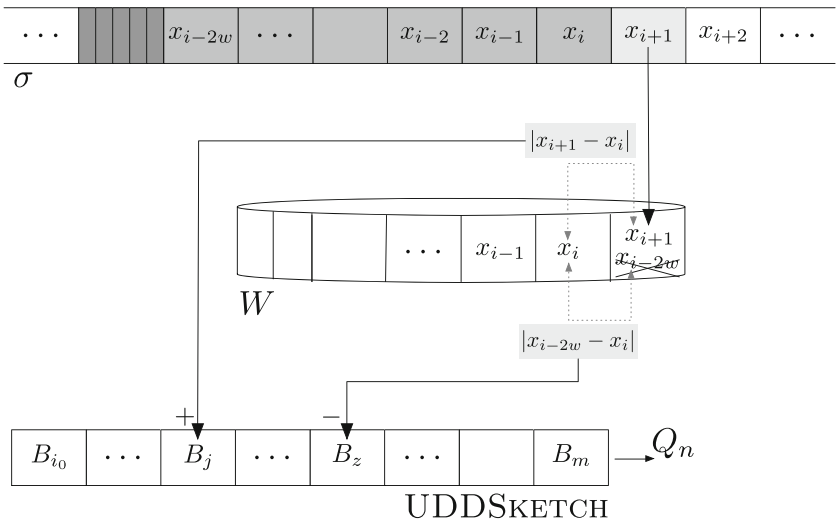


**Fig. 1** Algorithm's details

to satisfy the constraint on $m$; collapsing the sketch reduces the number of buckets to at most $m$.

Bucket boundaries are defined with regard to the quantity $\gamma = \frac{1+\alpha}{1-\alpha}$. Let the $i$th bucket be $B_i$, with index $i = \lceil \lg_\gamma x \rceil$. The input items $x$ such that $\gamma^{i-1} < x \leq \gamma^i$ fall in the bucket $B_i$, which is just a counter variable initialized to zero. To insert an item $x$ into the sketch, if the corresponding bucket already exists the algorithm simply increments that bucket's counter by one, otherwise the new bucket is added to the sketch, and then its counter is set to one. Symmetrically, an item is deleted decrementing by one the corresponding bucket's counter; if the counter's value becomes zero, its bucket is removed from the sketch. At the beginning, the sketch is empty, and the buckets are dynamically added or removed as required. We point out here that the bucket indexes are dynamic as well, since they depend on both the input items and the $\gamma$ value.

The insertion procedure may of course cause the sketch to grow without bounds. To prevent this, the algorithm executes a collapsing procedure when the number of buckets exceeds the maximum number of $m$ buckets. The pseudocode for UDDSKETCH insertion of an item $x$ into the sketch $\mathcal{S}$ is shown as Algorithm 1, whilst Algorithm 2 refers to the pseudocode for deleting an item.

**Algorithm 1** UDDSketch-Insert($x, \mathcal{S}$)

**Require:** $x \in \mathbb{R}_{\geq 0}$; sketch $\mathcal{S} = \{B_i\}_i$
  $i \leftarrow \lceil \lg_\gamma x \rceil$
  **if** $B_i \in \mathcal{S}$ **then**
    $B_i \leftarrow B_i + 1$
  **else**
    $B_i \leftarrow 1$
    $\mathcal{S} \leftarrow \mathcal{S} \cup B_i$
  **end if**
  **while** $|\mathcal{S}| > m$ **do**
  UNIFORMCOLLAPSE($\mathcal{S}$)
  **end while**

**Algorithm 2** UDDSketch-Delete($x, \mathcal{S}$)

**Require:** $x \in \mathbb{R}_{\geq 0}$; sketch $\mathcal{S} = \{B_i\}_i$
  $i \leftarrow \lceil \lg_\gamma x \rceil$
  $B_i \leftarrow B_i - 1$
  **if** $B_i == 0$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \setminus B_i$
  **end if**

In the original DDSKETCH algorithm the collapse is applied to the first two buckets whose counters' values are greater than zero (or, alternatively, it can be done on the last two buckets). Denoting respectively by $B_y$ and $B_z$, with $y < z$, the first two buckets, the collapsing procedure adds the count stored by $B_y$ to $B_z$, then $B_y$ is removed from the sketch.

Our UDDSKETCH algorithm uses a carefully designed uniform collapsing procedure. Instead of collapsing only the first two buckets with counts greater than zero, we collapse them all, in pairs. Consider a pair of indices $(i, i+1)$, with $i$ being an odd index and $B_i \neq 0$ or $B_{i+1} \neq 0$. When processing this pair of buckets we create and add to the sketch a new bucket whose index is $j = \lceil \frac{i}{2} \rceil$, and set its counter's value to the sum of the $B_i$ and $B_{i+1}$ counters' values. This new bucket replaces the two collapsed buckets which are then evicted from the sketch. The pseudocode of our uniform collapse procedure is shown as Algorithm 3.

**Algorithm 3** UDDSketch UniformCollapse($\mathcal{S}$).

**Require:** sketch $\mathcal{S} = \{B_i\}_i$
  **for all** $\{i : B_i > 0\}$ **do**
    $j \leftarrow \lceil \frac{i}{2} \rceil$
    $B'_j \leftarrow B'_j + B_i$
  **end for**
  **return** $\mathcal{S} \leftarrow \{B'_i\}_i$

Quantile estimation can be done by using the Query procedure, whose pseudocode is shown as Algorithm 4. The input is a quantile $0 \leq q \leq 1$. To estimate the quantile value, the procedure determines the index of the first bucket with count greater than zero, and then it sums the next counters' values greater than zero until the sum is less than or equal to $q(n-1)$. Letting $i$ be the index of the last bucket considered, the final estimation is obtained as $\tilde{x}_q = 2\gamma^i/(\gamma+1)$. A theoretical bound on the accuracy and the actual accuracy achieved experimentally by UDDSKETCH on several distributions is available in [7].

Since the computational complexity of inserting an item into the sketch is in the worst-case $O(1)$, it follows that inserting the $s$ differences requires at most $O(s)$. Therefore, for a theoretical perspective, AFQN shares the same complexity of FQN; however, as we shall show in Section 5, in practice, AFQN is faster.

**Algorithm 4** Query($q, \mathcal{S}$).

**Require:** $q$, quantile such that $0 \leq q \leq 1$; sketch $\mathcal{S} = \{B_i\}_i$
  $i_0 \leftarrow \min(\{j : B_j > 0\})$
  $count \leftarrow B_{i_0}$
  $i \leftarrow i_0$
  **while** $count \leq q(n-1)$ **do**
    $i \leftarrow \min(\{j : B_j > 0 \wedge j > i\})$
    $count \leftarrow count + B_i$;
  **end while**
  **return** $2\gamma^i/(\gamma+1)$

**Algorithm 5** QN($\mathcal{S}$, $W$, $x_{i+1}$)

**Require:** $\mathcal{S}$, the sketch; $W =< x_{i-2w}, \ldots, x_i >$ the temporal window of size $s = 2w + 1$; $x_{i+1}$ the incoming item.
  **for each** item $x_j \in W$, $j \neq i - 2w$
    UDDSKETCH-DELETE($|x_{i-2w} - x_j|$)
    UDDSKETCH-INSERT($|x_{i+1} - x_j|$)
  **end for**
  $k \leftarrow \binom{\lfloor s/2 \rfloor + 1}{2}$
  $q \leftarrow \frac{2(k-1)}{s(s-1)-2}$
  $q_n \leftarrow$ QUERY$(q, \mathcal{S})$
  **return** $q_n$

## 4 A robust statistical approach to outlier detection

An outlier detector working in streaming can be designed using a temporal window which slides forward one item at a time. Denoting with $W$ the current window and by $w$ the semi-window size, the algorithm processes the stream in windows $W =< \sigma_{i-2w}, \ldots, \sigma_i >$ of size $s = 2w + 1$. Once the first window has been processed, the new one is obtained by sliding the window one item ahead when the next item arrives from the stream. Letting $i - 2w$ be the index of the first item in a window $W$ (i.e., the oldest one), then the item under test in $W$ is the one located at the index $i - w$.

Traditionally, outlier detection has been based on the $z$-scores of the observations given by $z_{score} = \frac{x-\mu}{\sigma}$ where $x$ denotes the observation under test, and $\mu$ and $\sigma$ denote respectively the mean and the standard deviation of the observations. A different outlierness test has been proposed in [18], using robust estimators such as the median and the MAD (Median Absolute Deviation from the median of the observations): $(x - median(W))/\text{MAD}$.

As an example application for outlier detection, we use a slightly different $z$-score, in which we substitute the $Q_n$ estimator in place of MAD, obtaining the following outlierness test: $|x - median(W)|/Q_n$. The reasons for preferring the $Q_n$ estimator to MAD are its greater Gaussian efficiency (82% versus 36%) and its ability to deal with skewed distributions [17].

Denoting the item under test with $x = \sigma_{i-w}$, in order to determine whether $x$ is an outlier we proceed as follows. We begin computing the quantile $q$ according to Eq. 3 by considering only the values in the current window. Next, we determine $med$, the value of $W$ corresponding to the median order statistic and compute $q_n$, the $Q_n$ dispersion for the window $W$.

**Table 1** Metrics utilized

| Metric | | Definition |
|---|---|---|
| Recall | = | $\frac{\text{TP}}{\text{TP+FN}}$ |
| Precision | = | $\frac{\text{TP}}{\text{TP+FP}}$ |
| $F_1$ score | = | $\frac{2*(\text{Recall}*\text{Precision})}{(\text{Recall}+\text{Precision})}$ |
| $J(A, B)$ | = | $\frac{\lvert A \cap B \rvert}{\lvert A \cup B \rvert}$ |

TP denotes true positives, FP false positives, TN true negatives, FN false negatives. $A$ and $B$ are two countable finite sets

**Algorithm 6** Outlier Detection Using the $Q_n$ estimator

**Require:** $\sigma$, the input stream; $t$, multiplier of the $Q_n$ dispersion
  $Outliers \leftarrow \emptyset$
  $k \leftarrow \binom{\lfloor s/2 \rfloor + 1}{2}$
  $q \leftarrow \frac{2(k-1)}{s(s-1)-2}$
  **for each** window $W$ of size $s = 2w + 1$ in $\sigma$
    $x \leftarrow \sigma_{i-w}$
    $med \leftarrow$ MEDIAN$(W)$
    $q_n \leftarrow$ QUERY$(q, \mathcal{S})$
  **if** $|x - med| > t \cdot q_n$ **then**
    $Outliers \leftarrow Outliers \cup \{x\}$
  **end if**
  **end for**
  **return** $Outliers$

Let $t$ be a scalar integer acting as a multiplier of the $Q_n$ dispersion. In practice, $t$ is used to control the degree of outlierness of an item. To determine if an item $x$ is an

**Table 2** Synthetic data: experiments carried out

| Distribution | Parameters |
|---|---|
| beta | $\alpha = 2, \beta = 1/4$ |
| chi-squared | $\nu = 3$ |
| exponential | $\lambda = 1/2$ |
| gamma | $\alpha = 1, \beta = 2$ |
| half-normal | $\theta = 1/2$ |
| inverse gaussian | $\mu = 2, \lambda = 1$ |
| log-normal | $\mu = 1, \sigma = 3$ |
| normal | $\mu = 1, \sigma = 3$ |
| pareto | $k = 3, \alpha = 0.75$ |
| poisson | $\mu = 3$ |
| uniform | $min = 0, max = 100000$ |
| zipf | $n = 100000000, \rho = 1.2$ |

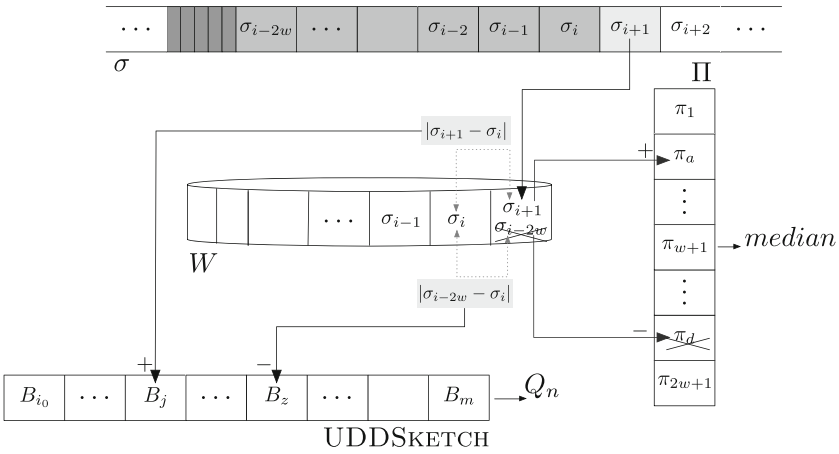


**Fig. 2** Implementation's details

outlier, we check the following condition (corresponding to the devised outlierness test): $|x - med| > t \cdot q_n$; if the condition is true, then $x$ is an outlier, otherwise $x$ is an *inlier* (i.e., a normal observation). In practice, the condition $|x - med| > t \cdot q_n$ identifies as outliers those points that are not within $t$ times the $Q_n$ dispersion from the sample median; regarding $t$, a commonly used value is $t = 3$ [3, 19]. The pseudocode is shown as Algorithm 6.

## 5 Experimental results

In this Section, we present and discuss experimental results, thoroughly comparing our algorithms FQN and AFQN. The former exactly determines the $Q_n$ values for its input data stream, whilst the latter provides an approximation for the $Q_n$ values. However, we do not directly compare the two algorithms with regard to the obtained $Q_n$ values, owing to the fact that the accuracy of UDDSKETCH has been thoroughly analyzed, both theoretically and experimentally in [7]. Rather, we shall compare the results obtained by the two algorithms when the computed $Q_n$ values are used for outlier detection as discussed in Section 4.

In particular, we shall compare and contrast the algorithms with regard to their speed, measured as the number of *updates per second*, and with regard to their accuracy related to the detection of the outliers in the input data stream. For this purpose, the following metrics shall be reported: Recall, Precision, $F_1$ score and Jaccard similarity between the sets of outliers determined by the algorithms. We shall assume that the set of outliers determined by FQN is the *ground truth*, i.e., the reported outliers are the *actual* outliers.

Table 1 reports the metrics used. Recall is related to the number of false negatives, and ranges between 0 and 1. It is 1 when there are no false negatives, i.e. all of the true outliers have been correctly reported in output, and it is 0 when no true outlier is reported. Similarly, precision is related to false positives. Its value ranges between 0 and 1, it is 1 when there are no false positives, i.e., no false outlier is reported in output and it is 0 when all of the items reported are false outliers. The $F_1$ measure takes into account both recall and precision, being their harmonic mean. $F_1$ is 1 when both precision and recall are 1, and is 0 if either the precision or the recall is 0. Finally, the Jaccard similarity $J(A, B)$ of two sets $A$ and $B$ is defined as the ratio $\frac{|A \cap B|}{|A \cup B|}$; when the sets $A$ and $B$ are both empty, the Jaccard similarity is defined as $J(A, B) = 1$.

The AFQN algorithm has been implemented in C++. The source code has been compiled using the Apple clang compiler v11.0.3 on macOS Big Sur version 11.1 with the following flags: -Os -std=c++14. The tests have been carried out on an Apple MacBook Pro laptop equipped with 32 GB of RAM and a 2.6 GHz exa-core Intel Core I7 processor with 12 MB of cache level 3. The source code is freely available for inspection and for reproducibility of results[1]. The tests have been performed on synthetic datasets, according to the distributions shown in Table 2; moreover, we also perform our experiments on a real dataset, kindly provided by Yahoo as part of its Webscope program[2]: Yahoo! Synthetic and real time-series with labeled anomalies, version 1.0. In particular, we have added the results obtained running the experiments on the dataset real_17 available in the A1Benchmark within the whole dataset, considering this dataset both with and without the provided human curated annotations which explicitly report the anomalies. This dataset contains 1424 items, 227 of which have been annotated as outliers.

It is worth noting here that, once again, computing the median in Algorithm 6 could be done approximately by using another instance of an UDDSKETCH data structure, since the median is just another quantile. However, we recall that our purpose is to compare, *ceteris paribus*, the results obtained by the two algorithms when the computed

[1] https://github.com/cafaro/AFQN

[2] http://research.yahoo.com

$Q_n$ values are used for outlier detection; since in FQN we compute the median exactly, had we approximately computed the median in AFQN we would have unfairly biased the obtained results.

The median of the window $W$ can be computed exactly in the worst case in $O(s)$ time by using the SELECT algorithm [1] but, as already observed in Section 2, the QUICKSELECT algorithm [8, 10] is much faster and is therefore the algorithm used in practice, despite being linear in the input size only on average.

In the streaming setting, the median can be computed exactly in $O(1)$; the key idea is to maintain the window in sorted order, so that the median can be directly accessed in constant time. This requires handling two different permutations of the window: one is $W$, which represents the actual order in which incoming items arrive from the stream,

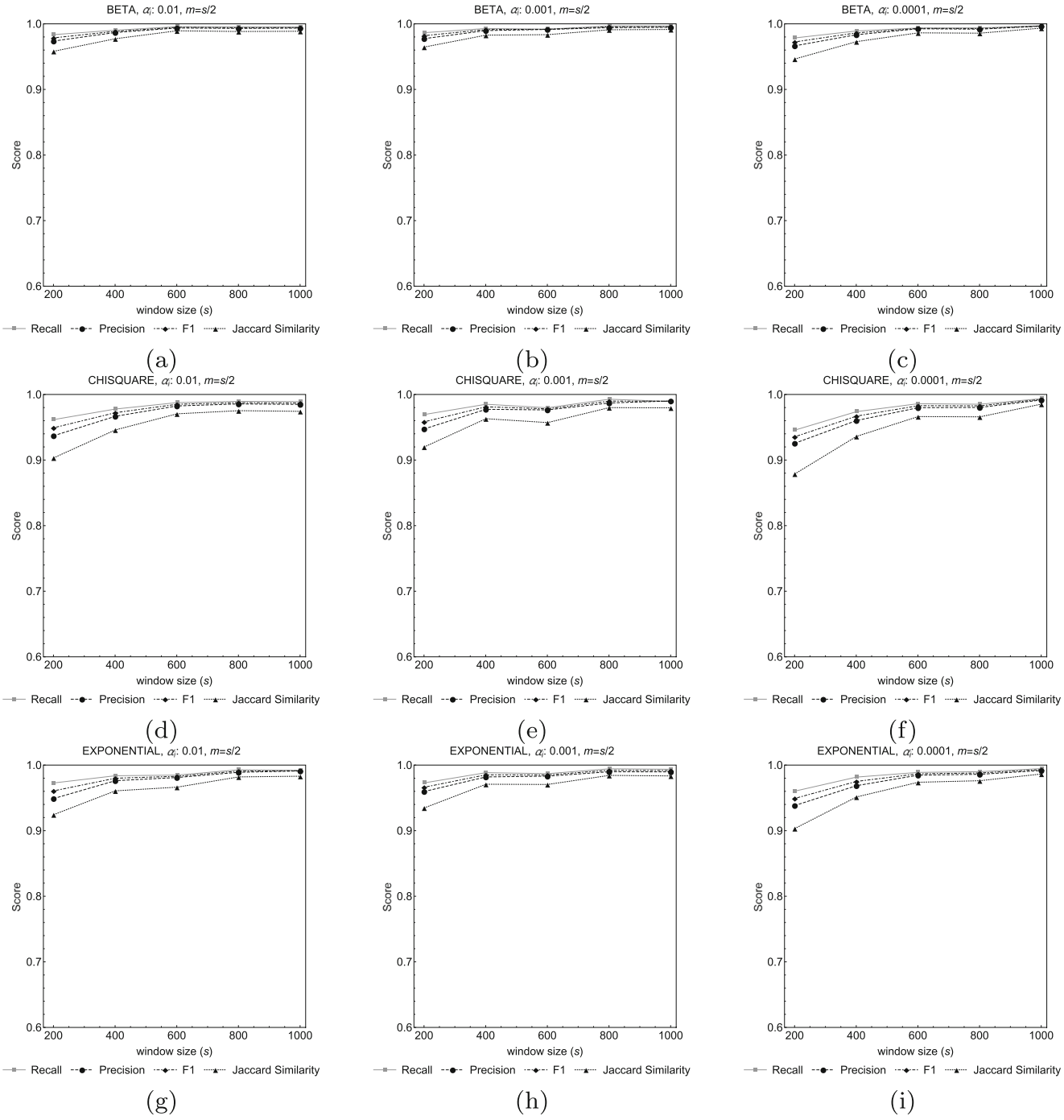


**Fig. 3** Precision, Recall, F1 score and Jaccard Similarity varying the UDDSketch initial accuracy parameter: beta, chisquare and exponential distributions

and the other is $\Pi$, which represents the sorted permutation of the items in the window. Therefore, a tradeoff is in place: we use additional space ($2s$ total space instead of $s$, but note that the amount of space used is still $O(s)$) for the data structure representing $\Pi$, but in return we are able to compute the median exactly in $O(1)$.

In our implementation $\Pi$ is an array, storing the items $\pi_1, \cdots, \pi_s$. To maintain $\Pi$ in sorted order, items are inserted in $\Pi$ as in InsertionSort [4]. Note that we do not sort $\Pi$: upon a new item's arrival, we remove the least recent (in the temporal sequence of item arrivals) item; since the previous window was already sorted, removing the

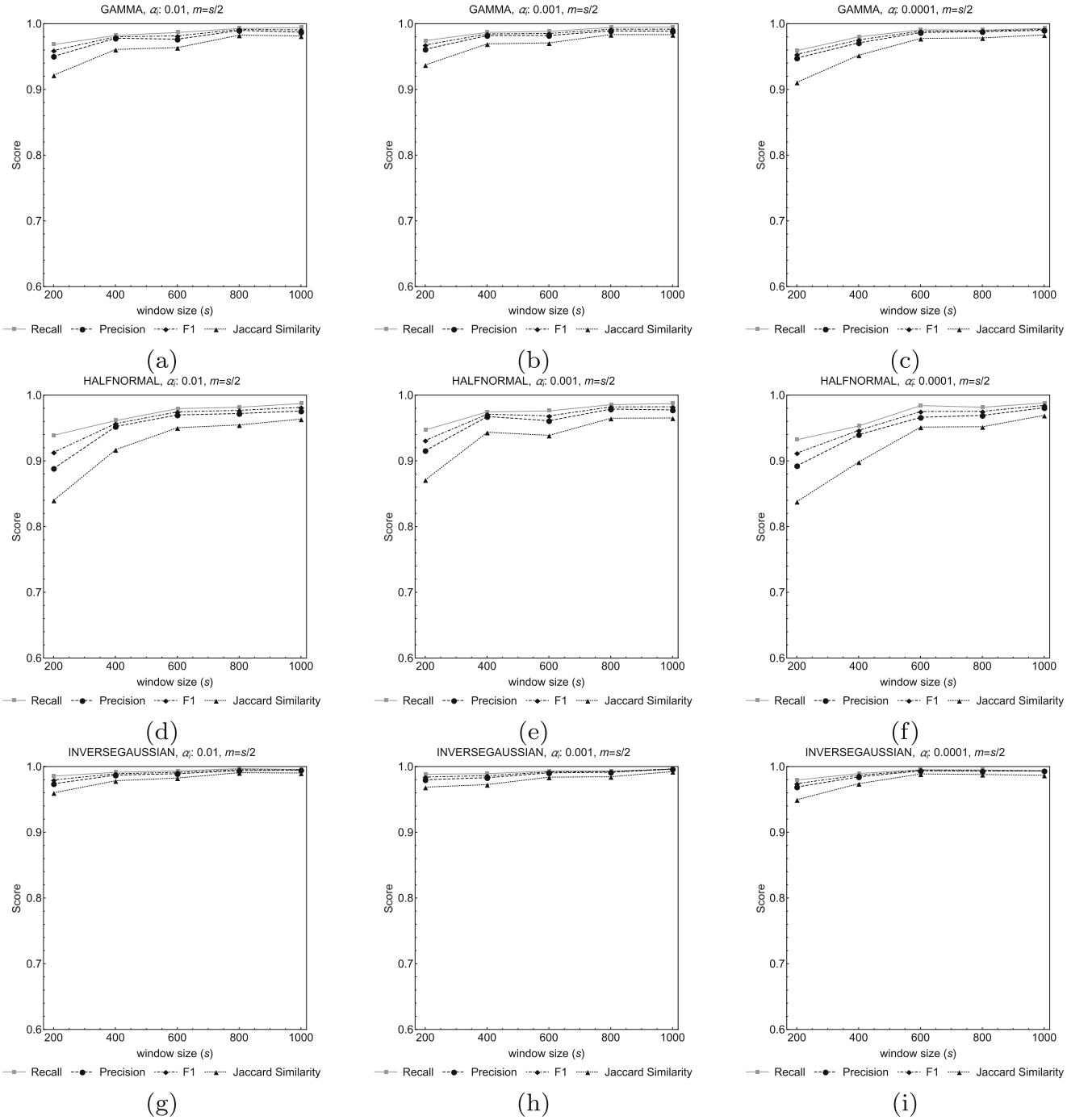


**Fig. 4** Precision, Recall, F1 score and Jaccard Similarity varying the UDDSketch initial accuracy parameter: gamma, halfnormal and inversegaussian distributions

least recent item leaves the window sorted. The new item is inserted using the InsertionSort insertion procedure, which requires $O(s)$ worst case time. Indeed, we can simply use a backward linear scan of $\Pi$ from right to left until we find the index, call it $j$, where the new item must be inserted. Then, we insert it by sliding all of the items $\pi_{j+1}, \cdots, \pi_s$ one position to the right. Alternatively, a binary search [4] can be used to determine the index $j$ in time $O(\lg s)$, but the total time required for the insertion in $\Pi$ is still $O(s)$, owing to the need to shift all of the items $\pi_{j+1}, \cdots, \pi_s$ one position to the right. When the window's size $s$ increases, this implementation is, in practice, experimentally faster

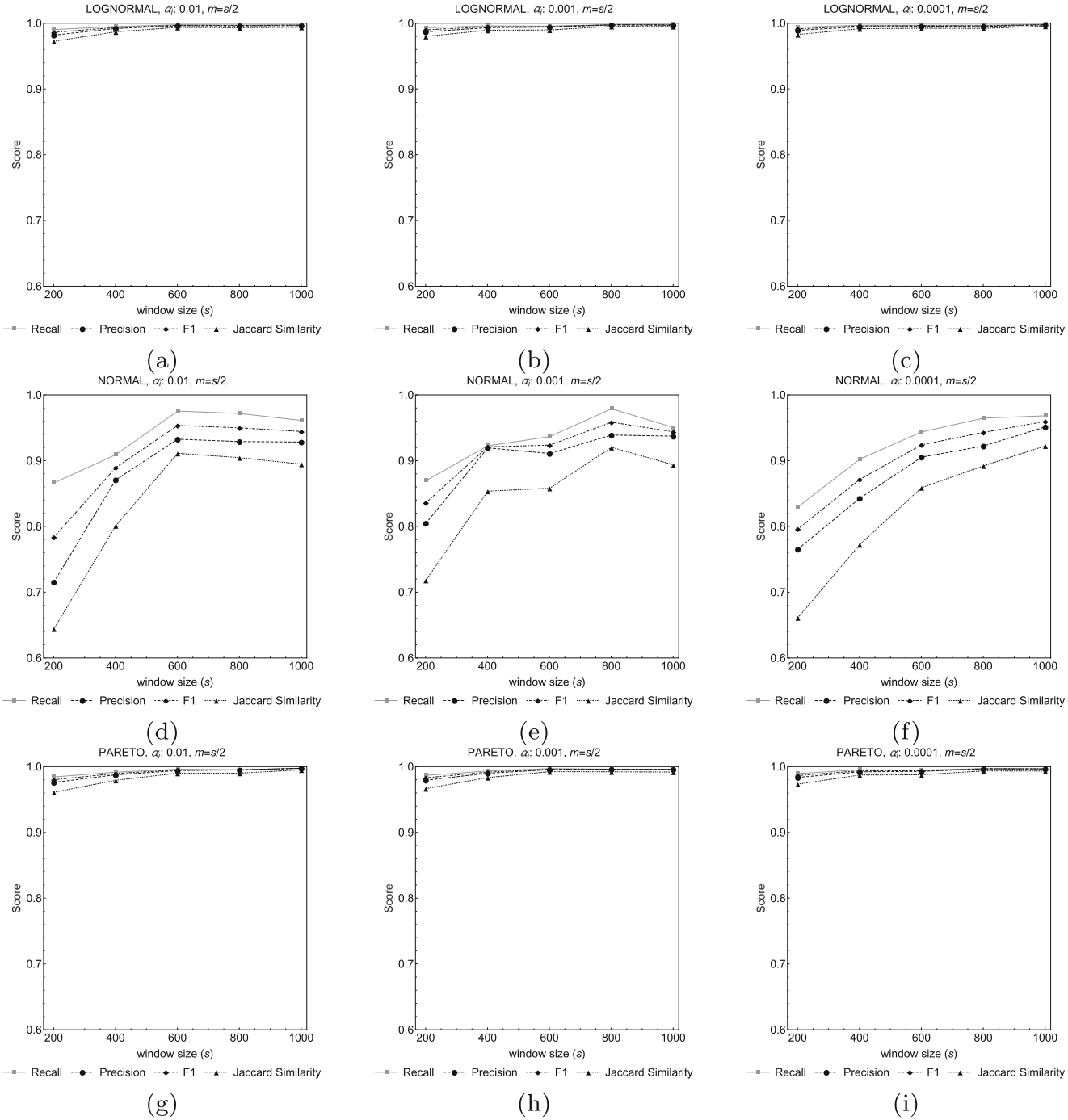


**Fig. 5** Precision, Recall, F1 score and Jaccard Similarity varying the UDDSketch initial accuracy parameter: lognormal, normal and pareto distributions

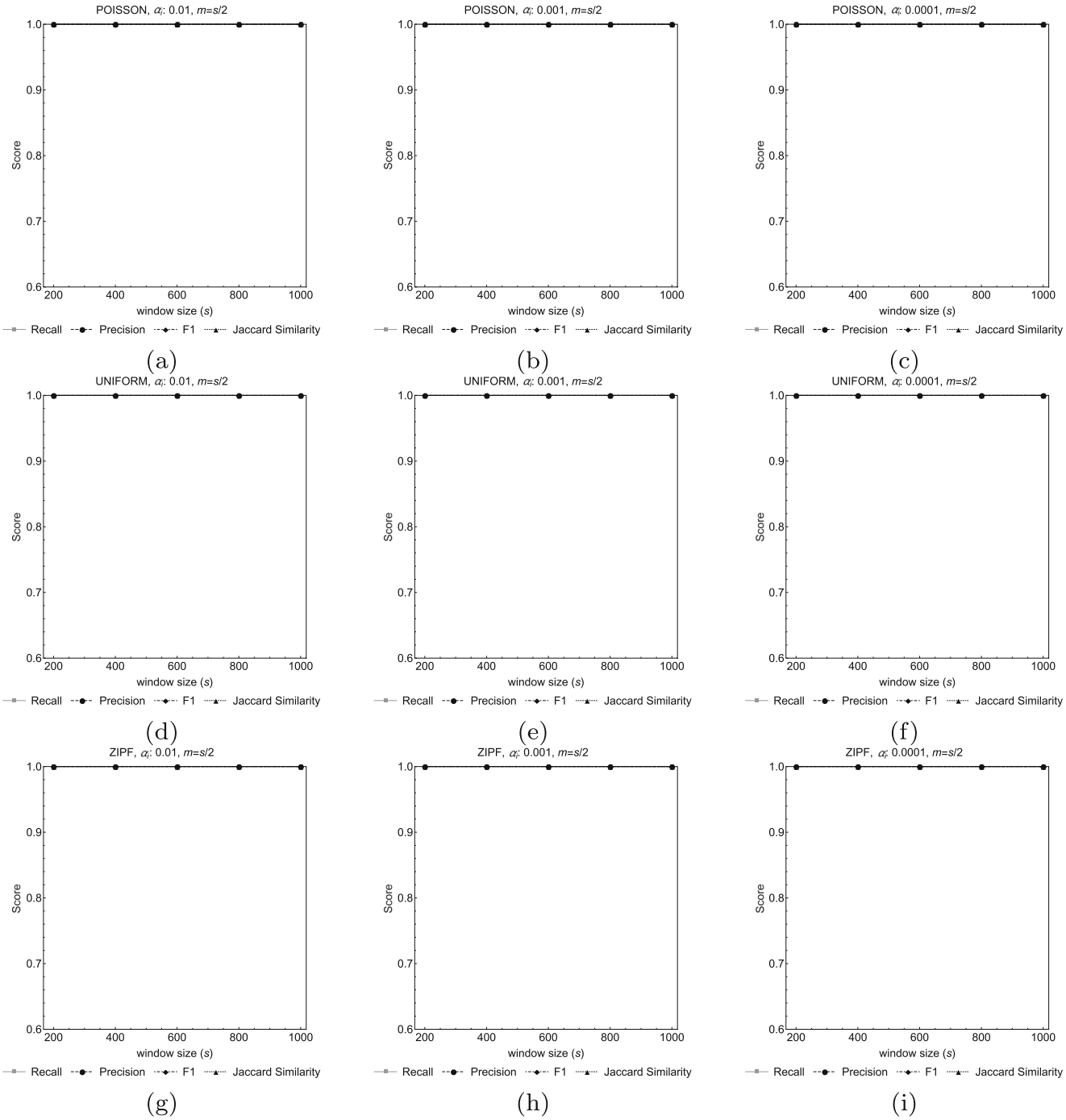


**Fig. 6** Precision, Recall, F1 score and Jaccard Similarity varying the UDDSketch initial accuracy parameter: poisson, uniform and zipf distributions

than using the QUICKSELECT algorithm, even though the computational complexity is the same, $O(s)$. Figure 2 depicts the implementation.

For each distribution, the algorithms have been executed three times and their results have been averaged. Figures 3, 4, 5 and 6 depict the precision, recall, F1 score and Jaccard similarity achieved by our AFQN algorithm varying its initial accuracy parameter and fixing the number $m$ of buckets to $s/2$, where $s$ is the window's size. As shown, AFQN provides excellent accuracy in practice for all of the distributions under test, even using a very tiny amount of buckets, equal to one half of the window's size. Only for the halfnormal and normal distributions the use of 100 or 200 buckets is not enough to achieve accuracy values greater than 0.9

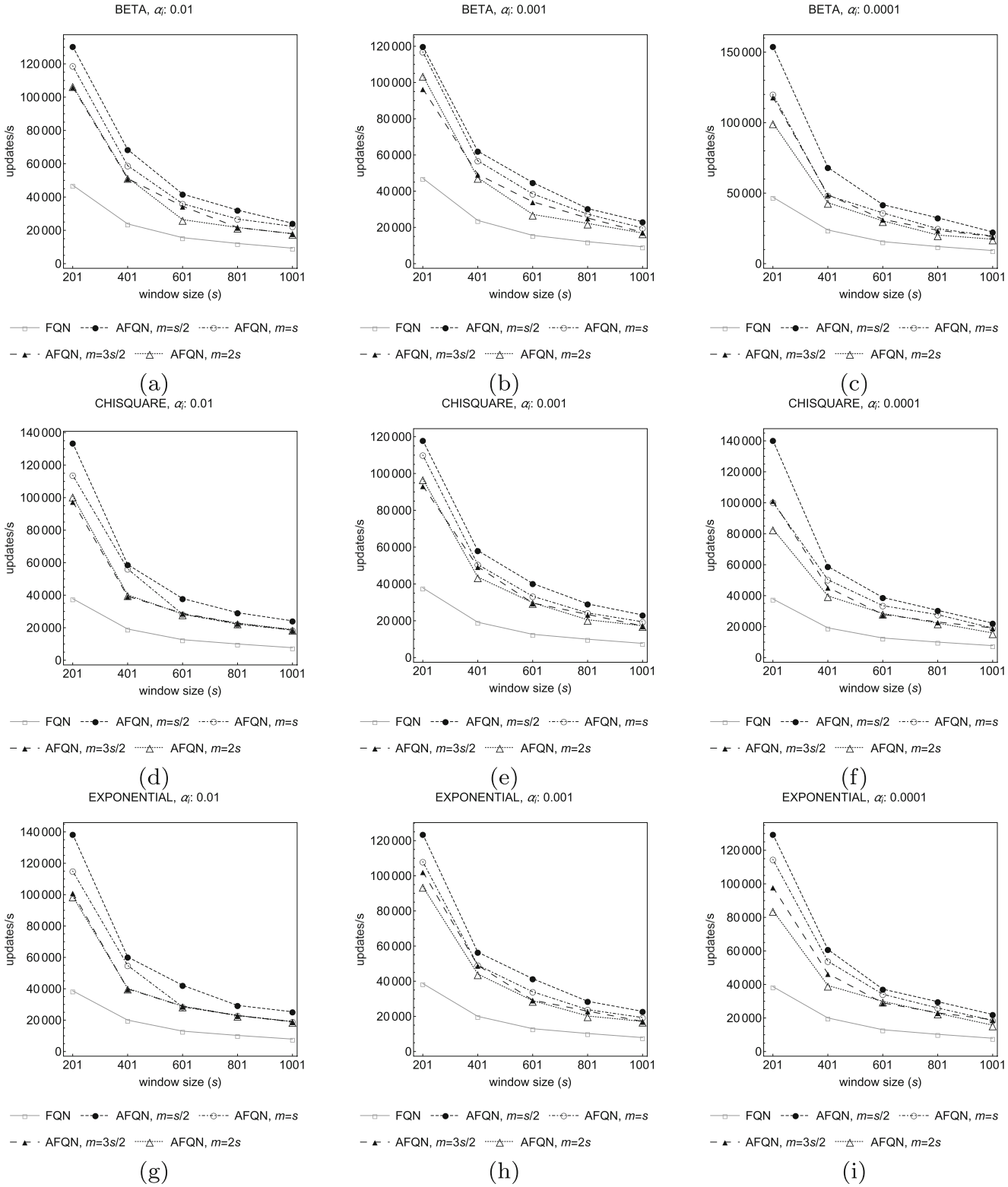


**Fig. 7** Updates per second varying the UDDSketch initial accuracy parameter and the sketch size: beta, chisquare and exponential distributions

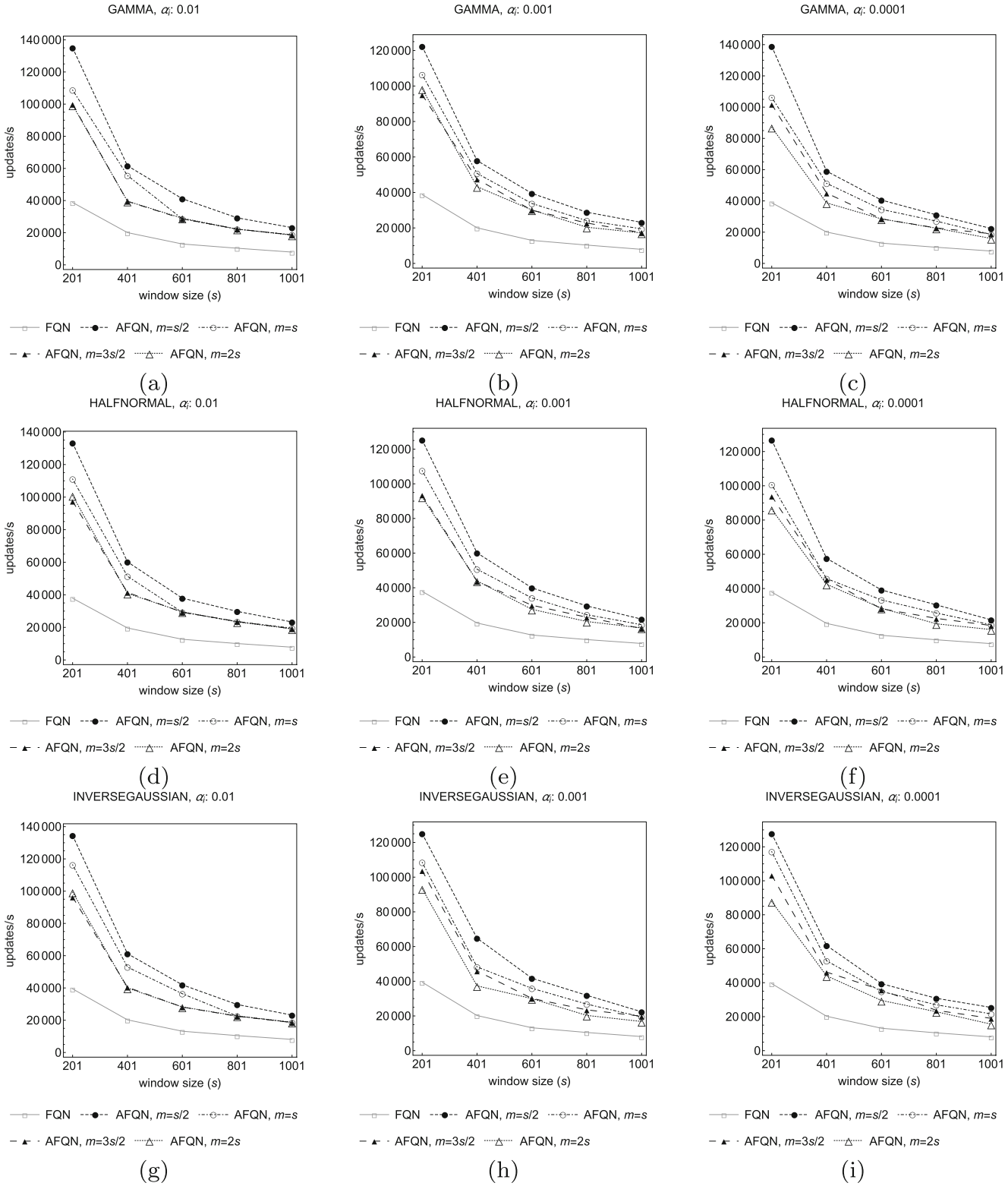


**Fig. 8** Updates per second varying the UDDSketch initial accuracy parameter and the sketch size: gamma, halfnormal and inversegaussian distributions

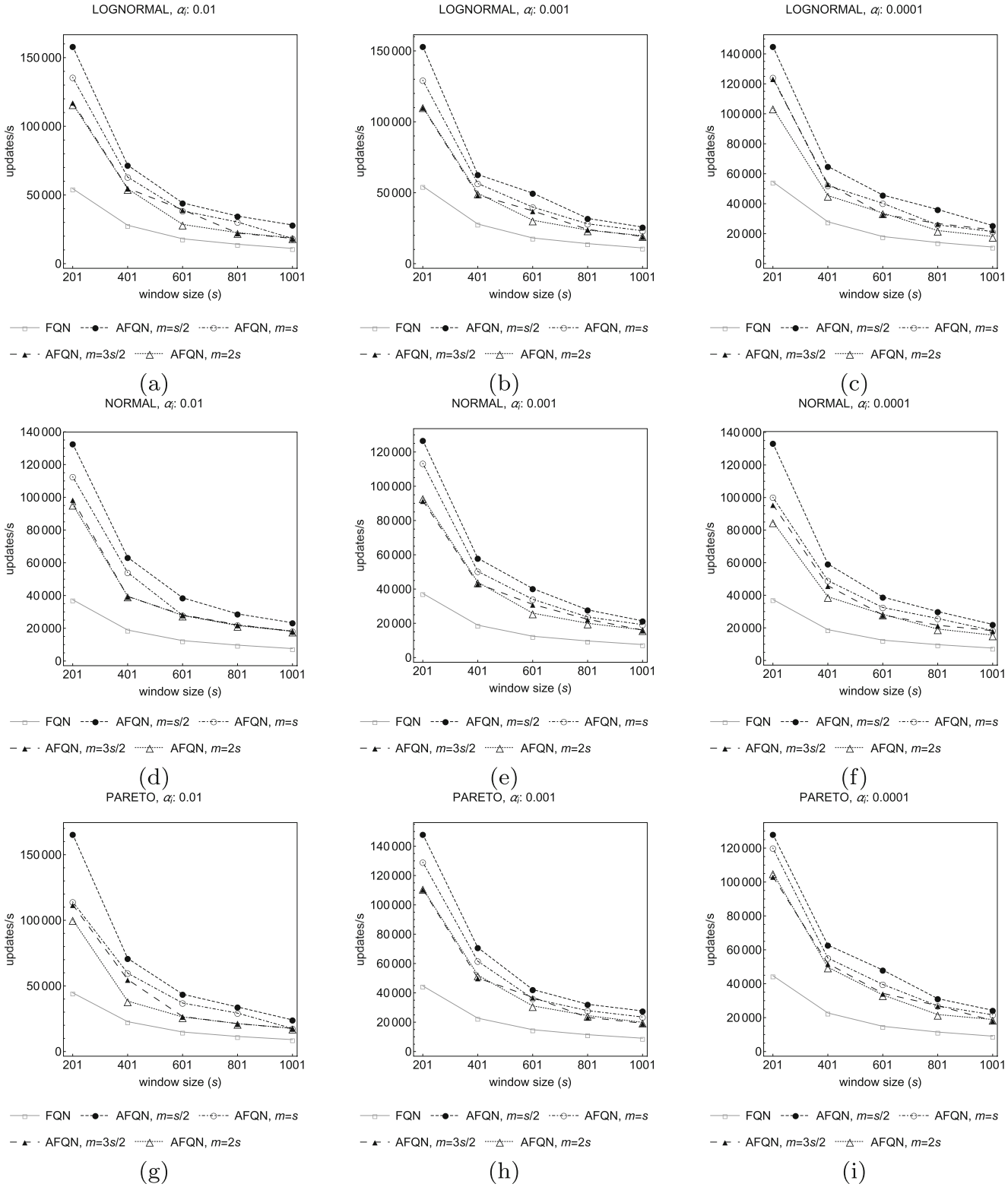


**Fig. 9** Updates per second varying the UDDSketch initial accuracy parameter and the sketch size: lognormal, normal and pareto distributions

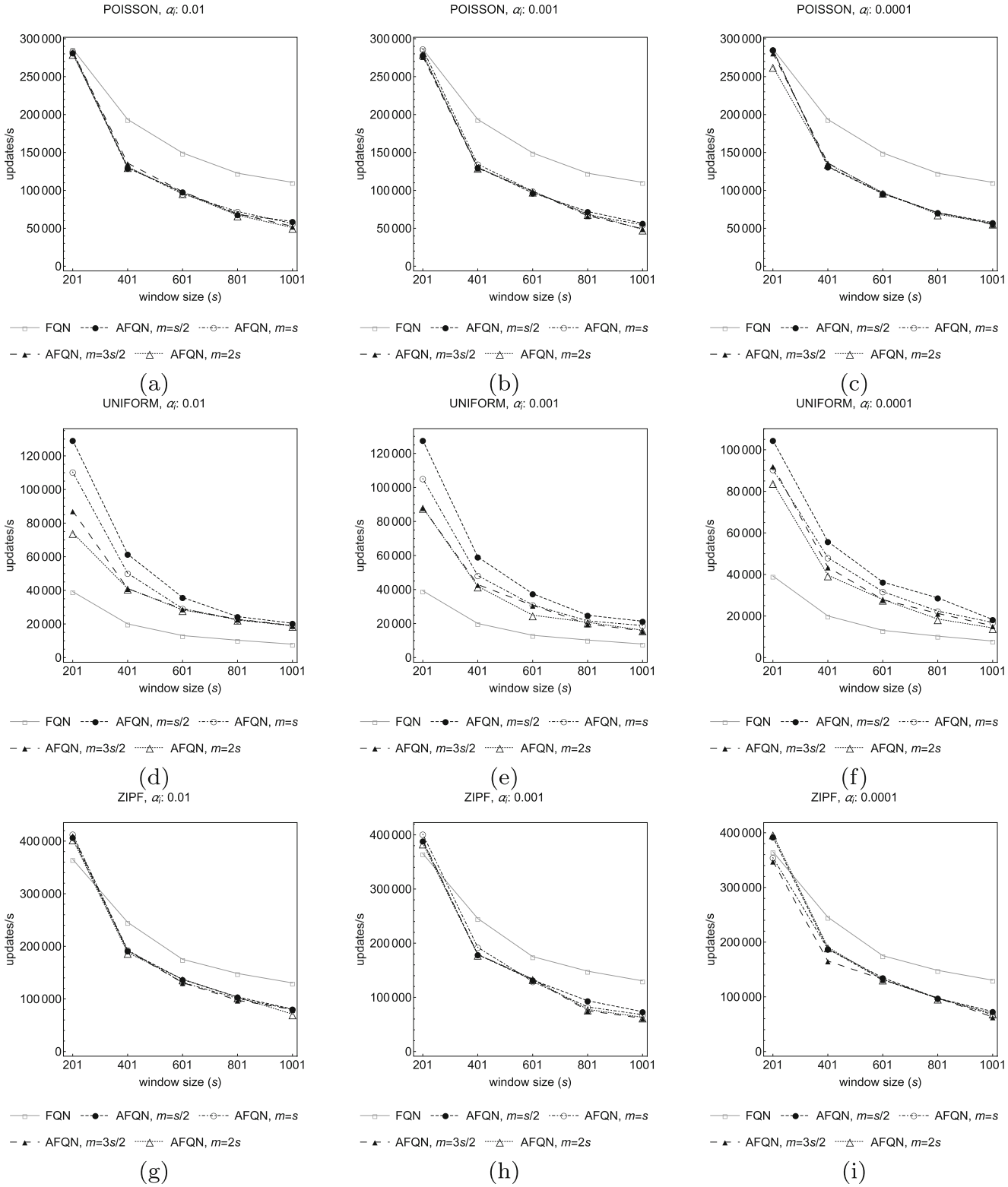


**Fig. 10** Updates per second varying the UDDSketch initial accuracy parameter and the sketch size: poisson, uniform and zipf distributions

(we recall here that 1.0 is the maximum score for each of the accuracy metrics under consideration). However, even for these two distributions the accuracy results are anyway extremely high.

The accuracy values obtained must be also discussed from the perspective of the actual running time required to achieve them. Figures 7, 8, 9 and 10 depict the running time of both FQN and AFQN with regard to the number of updates per second (higher values are better). In particular, we vary both the initial accuracy parameter and the number $m$ of buckets ($s/2$, $s$, $3s/2$ and $2s$).

Regarding AFQN we show the updates per second when varying its initial accuracy parameter. As shown, AFQN is always faster than FQN with the notable exception of the poisson and zipf distributions. Even though the computational complexity of both FQN and AFQN is the same, i.e., linear in the window's size, the constant hidden by the asymptotic notation $O(s)$ (in which $s$ denotes the window's size) is different, leading to these experimental results, in which AFQN is shown to be empirically faster in general than FQN. Regarding the specific behaviour of FQN related to the poisson and zipf distributions, the reason why FQN beats AFQN with regard to the number of updates/s is that these are *discrete* distributions with a few distinct items. In this case, FQN has an advantage owing to how its selection algorithms works [2]. In particular, AFQN is up to three times faster than FQN when using a very small number of buckets (equal to 100) and up to two times faster when using a greater number of buckets (in the range 200-500 buckets). Therefore, the experiments confirm the validity of our approach: AFQN proves to be extremely fast whilst providing, simultaneously, excellent accuracy in almost all of the cases of practical interest.

Regarding the real dataset, Figs. 11 and 12 depict the experimental results. In particular, the former picture refers to the dataset without the provided human curated annotations which explicitly report the anomalies; therefore, we assume that the set of outliers determined by FQN is the *ground truth*, i.e., the reported outliers are the *actual* outliers. The latter picture refers instead to the dataset when considering the annotations as ground truth. It is worth

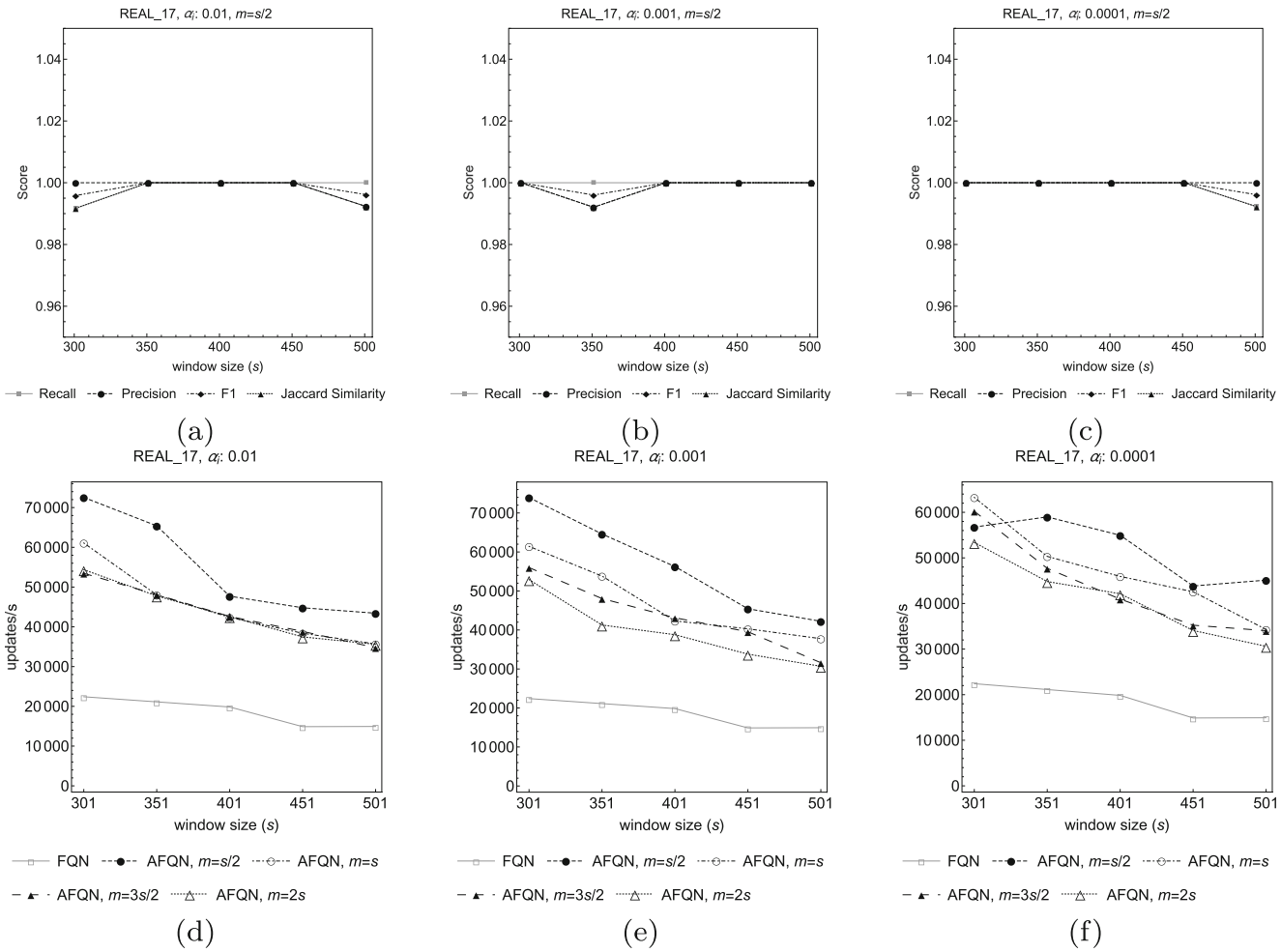


**Fig. 11** Precision, Recall, F1 score, Jaccard Similarity and updates/s varying the UDDSketch initial accuracy parameter: Yahoo A1Benchmark real_17 dataset without annotations

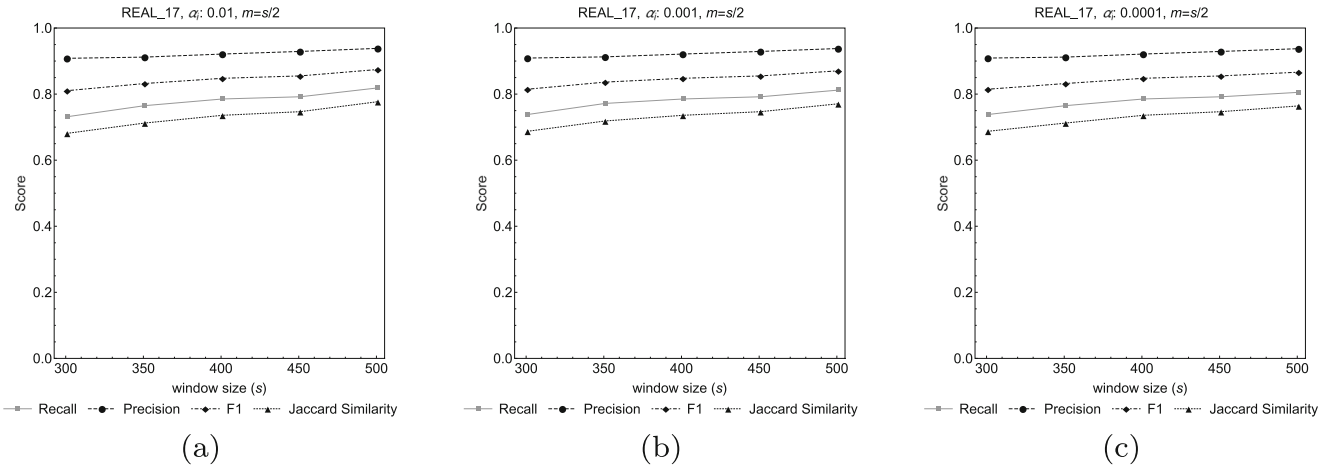


**Fig. 12** Precision, Recall, F1 score and Jaccard Similarity varying the UDDSketch initial accuracy parameter: Yahoo A1Benchmark real_17 dataset with annotations

noting here that, owing to its length, we choose to process the dataset using an appropriate windows' size. Therefore, in our experiments we vary the window' size from 301 to 501 using a step size equal to 50.

As shown in Fig. 11, the behaviour of AFQN does not change when processing a real dataset. Finally, Fig. 12 shows that the F1 score when considering the annotations is consistently beyond the value 0.8, and therefore the $Q_n$ scale estimator provides very good outlier recognition.

## 6 Conclusions

In this paper we have introduced AFQN (Approximate Fast $Q_n$), a novel algorithm for approximate computation of the $Q_n$ scale estimator in a streaming setting. The need for approximate estimation of the $Q_n$ estimator arises owing to the fact that exact computation may be too costly for some applications, and the problem is a fortiori exacerbated in the streaming setting, in which the time available to process incoming data stream items is short. We designed AFQN to approximate the $Q_n$ estimator quickly and with high accuracy. As an application, we have also shown the use of AFQN for fast detection of outliers in data streams. The incoming items are processed in the sliding window model, with a simple check based on the $Q_n$ scale estimator. Extensive experimental results on synthetic and real datasets have confirmed the validity of our approach, since AFQN is actually fast and can provide almost exact results.

**Funding** Open access funding provided by Università del Salento within the CRUI-CARE Agreement.

## Declarations

**Conflict of interest** The authors declare that they have no conflict of interest.



## References

1. Blum M, Floyd RW, Pratt VR, Rivest RL, Tarjan RE (1973) Time bounds for selection. J. Comput. Syst. Sci. 7(4):448–461
2. Cafaro M, Melle C, Pulimeno M, Epicoco I (2021) Fast online computation of the qn estimator with applications to the detection of outliers in data streams. Expert Syst Appl 164:113831. https://doi.org/10.1016/j.eswa.2020.113831. http://www.sciencedirect.com/science/article/pii/S0957417420306424
3. Chandola V, Banerjee A, Kumar V (2009) Anomaly detection: A survey (tr). ACM Comput Surv 41(3):1–58. https://doi.org/10.1145/1541880.1541882. http://portal.acm.org/citation.cfm?doid=1541880.1541882
4. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press
5. Croux C, Rousseeuw PJ (1992) Time-efficient algorithms for two highly robust estimators of scale. In: Dodge Y, Whittaker J (eds) Computational statistics. Physica-Verlag HD, Heidelberg, pp 411–428
6. Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows: (extended abstract). In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02. Society for Industrial and Applied Mathematics, Philadelphia, pp 635–644
7. Epicoco I, Melle C, Cafaro M, Pulimeno M, Morleo G (2020) Uddsketch: Accurate tracking of quantiles in data streams. IEEE Access 8:147604–147617. https://doi.org/10.1109/ACCESS.2020.3015599

8. Floyd RW, Rivest RL (1975) Expected time bounds for selection. Commun ACM 18(3):165–172
9. Hampel FR (1974) The influence curve and its role in robust estimation. J Am Stat Assoc 69(346):383–393. https://doi.org/10.1080/01621459.1974.10482962
10. Hoare CAR (1961) Algorithm 65: find. Commun ACM 4(7):321–322
11. Hodge V, Austin J (2004) A survey of outlier detection methodologies. Artif Intell Rev 22(2):85–126. https://doi.org/10.1023/B:AIRE.0000045502.10941.a9
12. Johnson D, Mizoguchi T (1978) Selecting the k-th element in $x + y$ and $x\_1 + x\_2 + \cdots + x\_m$. SIAM J Comput 7(2):147–153. https://doi.org/10.1137/0207013
13. Masson C, Rim JE, Lee HK (2019) Ddsketch: a fast and fully-mergeable quantile sketch with relative-error guarantees. Proc VLDB Endow 12(12):2195–2205. https://doi.org/10.14778/3352063.3352135
14. Mirzaian A, Arjomandi E (1985) Selection in x + y and matrices with sorted rows and columns. Inf Process Lett 20(1):13–17. https://doi.org/10.1016/0020-0190(85)90123-1. http://www.sciencedirect.com/science/article/pii/0020019085901231
15. Muthukrishnan S (2005) Data streams: Algorithms and applications. Found Trends® Theor Comput Sci 1(2):117–236. https://doi.org/10.1561/0400000002
16. Nunkesser R, Schettlinger K, Fried R (2008) Applying the qn estimator online. In: Preisach C, Burkhardt H, Schmidt-Thieme L, Decker R (eds) Data analysis, machine learning and applications. Springer, Berlin, pp 277–284
17. Rousseeuw PJ, Croux C (1993) Alternatives to the median absolute deviation. J Am Stat Assoc 88(424):1273–1283. https://doi.org/10.1080/01621459.1993.10476408
18. Rousseeuw PJ, Hubert M (2011) Robust statistics for outlier detection. WIREs Data Min Knowl Discov 1(1):73–79. https://doi.org/10.1002/widm.2
19. Shewhart WA (1931) Economic control of quality of manufactured product. Macmillan And Co Ltd, London




**Italo Epicoco** is an Assistant Professor at the University of Salento. He received a Ph.D. in Computational Engineering at the University of Lecce, Italy. He is an affiliate researcher of the Euro-Mediterranean Center on Climate Change - CMCC. His research interests include High Performance, Distributed, Grid and Cloud Computing with particular emphasis on parallel data mining. During his past research activities he addressed issues related to the optimization of numerical methods for solving PDEs applied to Earth System Models and to fluid dynamics models on High-End parallel architectures including heterogeneous architectures made of accelerators (NVIDIA GPU and Intel MIC). Relevant activities also included optimized management of a huge amount of data produced by the climate models. He published more than 40 papers in refereed books, journals and conference proceedings.

**Catiuscia Melle** graduated magna cum laude in Computer Engineering receiving her M.Sc. degree from the University of Salento, Italy. Currently, she is a Ph.D. Student at the University of Salento where she works in the field of Data Mining. Her past research interests include networking and content distribution applications.

**Massimo Cafaro** is Associate Professor at the Department of Engineering for Innovation of the University of Salento, Italy. His research covers Parallel and Distributed Computing, Cloud and Grid Computing, Data Mining and Big Data. He received a Ph.D. in Computer Science from the University of Bari, Italy. He is a Senior Member of IEEE and of IEEE Computer Society, Senior Member of the ACM, Vice Chair of Regional Centers and Coordinator of the Technical Area on Data Intensive Computing for the IEEE Technical Committee on Scalable Computing. He serves as an Associate Editor for IEEE Access. He is the author of more than 100 refereed papers on parallel, distributed and cloud/grid computing. He holds a patent on distributed database technologies.

**Marco Pulimeno** is a Postdoc researcher at the University of Salento, Italy. He received a Ph.D. in Mathematics and Computer Science from the University of Salento, Italy. His research interests include High Performance Computing, Distributed Computing, and, in particular, parallel data mining. He published on the topic of frequent items in several refereed journals and conference proceedings.