Efficient and flexible algorithms for monitoring distance-based outliers over data streams

Authors: Kontaki, Gounaris, Papadopoulos, Tsichlas, Manolopoulos

CS562 – Advanced Topics in Databases: Paper Presentation

Presenter: Giaourtas Michail
Introduction

- Outliers: Anomalies on a dataset
  - Noise effecting data processing or useful information
- Distance-based outliers
  - Most widely used
  - Object $x$ is an outlier if there are less than $k$ objects in a distance at most $R$ from $x$
  - The distance computation may differ depending on the application (Euclidean, Jaccard, quadratic etc.)
Introduction

- Detection over static data
  - Consumes more memory
  - Requires recomputation if a change is made in the data set
- Detection over data streams
  - Most widely used
  - Keeps n most recent objects (active objects) using a sliding window
    - Count-based
    - Time-based
Introduction

- This work:
  - Proves a linear space lower bound $\Omega(W)$
    - We need to store all objects
  - A basic algorithm for outlier detection over data streams with the ability of detecting for multiple values of $k$
  - An extended approach which also supports varying values for $R$
  - A micro-cluster approach for better performance
  - Algorithms can work with any distance calculation method
  - Performance evaluation results
Contents

- Related Work
- Basic Concepts
- Theoretical analysis
- Proposed Algorithms
  - Event-based approach
    - LUE & DUE
  - COD
  - ACOD
  - MCOD / AMCOD
- Evaluation
- Conclusions
Related Work – Static Data

- Most of the early techniques developed on static data
- Objects are modeled as a distribution (statistical-based approaches)
  - Not efficient for massive data and large dimensionalities
- Distance-based approach was introduced by Knorr for static data
- For each change in the data set the algorithm must be executed again
Related Work – Streaming Data

- Static data algorithms cannot process high rates of streaming data
- Outlier detection in streamed data became critical
  - Stock monitoring, spam detection, monitor sensor values etc.
- Keep track of the most recent data, not all of them
- Target is to reduce running time and storage requirements
Basic Concepts (1/3)

- Sliding Window
  - **Time-based**
    - For each window: $T_{end} = T_{start} + W$
    - Number of objects depends on arrival rate
    - Each object expires after $x = \left\lfloor \frac{W}{slide} \right\rfloor$ slides
  - **Count-based**
    - Number $n$ of objects in $W$ is fixed
- **Neighbors**: $p_i, p_j$ neighbors iff $Distance(p_i, p_j) \leq R$
- **Distance-based outlier**:
  - if $nn(p_i, R) < k \Rightarrow p_i$ is an outlier for the specific window slide
  - where $nn(p_i, R)$ number of neighbors
Basic Concepts (2/3)

- $D(R, k)$: Set of outliers
- $I(R, k)$: Set of inliers
- $P$: Set of objects in current Window
- A query is created per $R, k$ pair
- $O(n^2)$ space requirements to store all neighbors for each one of the $n$ objects in the current Window
Basic Concepts (3/3)

- Related algorithms managed to reduce computation and space cost
  - **Storm** algorithm stores maximum $k$ most recent preceding neighbors
    - Reduces computational cost to $O(n \log k)$
    - Reduces memory requirements to $O(kn)$
  - **Abstract-C** algorithm stores the number of neighbors of an object for all slides until it expires
    - Computational cost drops to $O(n)$
    - …but memory requirements reach $O(nW)$

- **This work:**
  - Manages lower computational cost
  - $O(n)$ space requirements which is proven to be optimal
Space lower bound proof

- Proof that space requirement for the distance-based outlier counting is $\Omega(n)$
- Using the problem of **Set-Disjointness** for calculating the communication complexity of a function $f: X \times Y \rightarrow Z$:
  - $[n] = \{1, 2, \ldots, n\} = \{T_1, T_2, \{i\}\}$,
  - where $T_1 \cap T_2 = \emptyset$, $i \notin T_1, T_2$, $T_1 \cup T_2 \cup \{i\} = [n]$,
  - Say $X \subseteq T_1 \cup \{i\}$ and $Y \subseteq T_2 \cup \{i\}$
  - The Set-Disjointness function is defined as
    - $f(x, y) = 1$ if $x \cdot y \neq 0$,
    - $f(x, y) = 0$ if $x \cdot y = 0$
  - One-way randomized communication complexity for the problem is $R^1_\delta(n) = \Omega(n)$
Space lower bound proof

- The problem is reduced to the counting of the number of outliers
- Finding the number of outliers is at most as difficult as discovering them

\[ if \ x_i = 0, \ \text{put a point within the disk with radius } R \]
\[ \text{Same for } y \]
\[ x = [0,1,0], \quad y = [1,1,0] \]
\[ if \ f(x, y) = 1 \text{ then there is an outlier for } k = 2 \]
Proposed Algorithms

- Event-based approach for detection of outliers
  - Two variations about handling the events: LUE & DUE
- One basic algorithm with 3 extra variations/extensions
The event-based approach

- The arrival and departure of objects can turn inliers to outliers or vice versa.
- The departure of an object is known: \( p.exp = T_{\text{now}} + \left\lceil \frac{W}{\text{Slide}} \right\rceil \)
  - An event queue can be created to check for changes resulted by departures.
- Using a Fibonacci heap we can efficiently support:
  - Findmin(): most recent event \( (O(1)) \)
  - Extractmin(): findmin() + delete \( (O(\log n)) \)
  - Increasetime(\( p, t \)): \( p.ev += t \) \( (O(1)) \)
  - Insert(\( p, t \)): \( (O(1)) \)
- Only used to check if an inlier becomes outlier.
The basic algorithm (specific R and k)

- Like **Storm** algorithm, only \( k \) (at most) preceding neighbors \( (P_p) \) are stored
- Let \( n_p^+ \) = number of succeeding neighbors
  - If \( n_p^+ \geq k \) => \( p \) is a safe inlier: not included in the event queue
  - Otherwise store \( k - n_p^+ \) most recent preceding neighbors
- Depending on the departures:
  \[
  p.ev = p.minexp = \min(p_i.exp \mid p_i \in P_p)
  \]
- Depending on the arrivals:
  - Lazy Update of Events (LUE)
  - Direct Update of Events (DUE)
Lazy Update of Events (LUE)

- For each new arrival
  - Update $n^+_p$ of all neighbor outliers
    - If a neighbor becomes an inlier it is inserted in the event queue
  - Update $n^+_p$ of all neighbor inliers
  - The new object is inserted to $\mathcal{D}(R, k)$ (if $n^-_p < k$) or to the event queue and $\mathcal{I}(R, k)$
LUE Run (1/3)

\[ k = 4 \]

\[ P_{p_8} = \{p_1\}, \quad n_{p_8}^+ = 2, \quad nn_{p_8} = 3 \]

\[ P_{p_{14}} = \{p_1, p_7, p_{10}, p_{12}\}, \quad n_{p_{14}}^+ = 0, \quad nn_{p_{14}} = 4 \]

- \( EvQ = \{p_{14}, p_1\} \)
- \( D(R, k) = \{\ldots, p_8, \ldots\} \)
- \( I(R, k) = \{\ldots, p_{14}, \ldots\} \)
Efficient and flexible algorithms for monitoring distance-based outliers over data streams

$k = 4$

$P_{p_8} = \{p_1\}, \quad n_{p_8}^+ = 3, \quad nn_{p_8} = 4$

$P_{p_{14}} = \{p_1, p_7, p_{10}, p_{12}\}, \quad n_{p_{14}}^+ = 0, \quad nn_{p_{14}} = 4$

- $EvQ = \{p_{14}(p_1), p_8(p_1)\}$ insert$(p_8, time)$
- $D(R, k) = \{..., , ..., \}$
- $I(R, k) = \{..., p_8, p_{14}, ..., \}$
LUE Run (3/3)

Efficient and flexible algorithms for monitoring distance-based outliers over data streams

\[ k = 4 \]

\[ P_{p_8} = \{\}, \quad n^+_p_{p_8} = 3, \quad nn_{p_8} = 3 \]

\[ P_{p_{14}} = \{p_7, p_{10}, p_{12}\}, \quad n^+_p_{p_{14}} = 0, \quad nn_{p_{14}} = 3 \]

- \( EvQ = \{\} \) \( extractmin() \)
- \( D(R, k) = \{\ldots, p_8, p_{14}, \ldots\} \)
- \( I(R, k) = \{\ldots, , \ldots\} \)
Direct Update of Events (DUE)

- For each new arrival, besides the actions of LUE
  - Update the event time and the preceding neighbours of all neighbor inliers and check if they become safe inliers
    - If a neighbor becomes an inlier it is inserted in the event queue
  - With updated event times, the extractmin() calls are reduced and replaced by increasetime() calls (which are faster)
**DUE Run (1/2)**

$k = 4$

$P_{p_8} = \{p_1\}, \ n_{p_8}^+ = 2, \ nn_{p_8} = 3$

$P_{p_{14}} = \{p_1, p_7, p_{10}, p_{12}\}, \ n_{p_{14}}^+ = 0, \ nn_{p_{14}} = 4$

- $EvQ = \{p_{14}(p_1)\}$
- $D(R, k) = \{..., p_8, ...\}$
- $I(R, k) = \{..., p_{14}, ...\}$
DUE Run (2/2)

\[ k = 4 \]

\[ P_{p_8} = \{p_1\}, \quad n_{p_8}^+ = 3, \quad nn_{p_8} = 4 \]

UPDATE:

\[ P_{p_{14}} = \{p_7, p_{10}, p_{12}\}, \quad n_{p_{14}}^+ = 2, \quad nn_{p_{14}} = 5 \]

UPDATE:

- \( EvQ = \{p_8(p_1), p_{14}(p_7)\} \) insert\((p_8, t)\), increase\(time(p_{14}, t)\)
- \( D(R, k) = \{\ldots, \ldots\} \)
- \( I(R, k) = \{\ldots, p_8, p_{14}, \ldots\} \)
- Also check for safe inliers
Outlier detection

- A new query per (R, k) pair
- Studying the cases
  - COD - R is fixed but k varies
  - K is fixed but R varies
  - ACOD - Both parameters vary
COD – Continuous Outlier Detection

- R is fixed but k varies
- Same neighbors: \( n_p^+ \) is the same for all queries
- Find \( k_{\text{max}} \) and store \( k_{\text{max}} - n_p^+ \) preceding neighbors
- For each departure check in \( I(R, q \cdot k_{\text{max}}) \) and decrease k until p is inlier
- For each arrival check in \( D(R, q \cdot k_{\text{max}}) \) and decrease k until p is outlier
COD Run (1/3)

\[ k = 2, 3, 4 \]

\[ P_{p_8.4} = \{p_1\}, \ n_{p_8.4}^+ = 2, \ nn_{p_8.4} = 3 \]

\[ P_{p_8.3} = \{p_1\}, \ n_{p_8.3}^+ = 2, \ nn_{p_8.3} = 3 \]

\[ P_{p_8.2} = \emptyset, \ n_{p_8.2}^+ = 2, \ nn_{p_8.3} = 2 \]

\[ P_{p_{14.4}} = \{p_1, p_7, p_{10}, p_{12}\}, \ n_{p_{14.4}}^+ = 0, \ nn_{p_{14.4}} = 4 \]

\[ P_{p_{14.3}} = \{p_7, p_{10}, p_{12}\}, \ n_{p_{14.3}}^+ = 0, \ nn_{p_{14.3}} = 3 \]

\[ P_{p_{14.2}} = \{p_{10}, p_{12}\}, \ n_{p_{14.2}}^+ = 0, \ nn_{p_{14.2}} = 2 \]

\[ EvQ = \{p_{14.4}(p_1), p_{8.3}(p_1)\} \]

\[ D(R, 4) = \{\ldots, p_8, \ldots\} \quad D(R, 3) = \{\ldots, \ldots\} \quad D(R, 2) = \{\ldots, \ldots\} \]

\[ J(R, 4) = \{\ldots, p_{14}, \ldots\} \quad J(R, 3) = \{\ldots, p_8, p_{14}, \ldots\} \quad J(R, 3) = \{\ldots, p_8, p_{14}, \ldots\} \]
COD Run (2/3)

\[ k = 2, 3, 4 \]

\[ P_{p_{8.4}} = \{p_1\}, \quad n^+_{p_{8.4}} = 3, \quad nn_{p_{8.4}} = 4 \]

\[ P_{p_{8.3}} = \emptyset, \quad n^+_{p_{8.3}} = 3, \quad nn_{p_{8.3}} = 3 \]

\[ P_{p_{8.2}} = \emptyset, \quad n^+_{p_{8.2}} = 3, \quad nn_{p_{8.3}} = 3 \]

\[ P_{p_{14.4}} = \{p_1, p_7, p_{10}, p_{12}\}, \quad n^+_{p_{14.4}} = 0, \quad nn_{p_{14.4}} = 4 \]

\[ P_{p_{14.3}} = \{p_7, p_{10}, p_{12}\}, \quad n^+_{p_{14.3}} = 0, \quad nn_{p_{14.3}} = 3 \]

\[ P_{p_{14.2}} = \{p_{10}, p_{12}\}, \quad n^+_{p_{14.2}} = 0, \quad nn_{p_{14.2}} = 2 \]

\[ EvQ = \{p_{14.4}(p_1), p_{8.4}(p_1)\} \]

\[ D(R, 4) = \{\ldots, \ldots\} \quad D(R, 3) = \{\ldots, \ldots\} \quad D(R, 2) = \{\ldots, \ldots\} \]

\[ J(R, 4) = \{\ldots, p_8, p_{14}, \ldots\} J(R, 3) = \{\ldots, p_8, p_{14}, \ldots\} J(R, 3) = \{\ldots, p_8, p_{14}, \ldots\} \]
COD Run (3/3)

\[ k = 2, 3, 4 \]

\[ P_{p_{8.4}} = \{ \}, \quad n^+_{p_{8.4}} = 3, \quad nn_{p_{8.4}} = 3 \]

\[ P_{p_{8.3}} = \{ \}, \quad n^+_{p_{8.3}} = 3, \quad nn_{p_{8.3}} = 3 \]

\[ P_{p_{8.2}} = \{ \}, \quad n^+_{p_{8.2}} = 3, \quad nn_{p_{8.3}} = 3 \]

\[ P_{p_{14.4}} = \{p_7, p_{10}, p_{12}\}, \quad n^+_{p_{14.4}} = 0, \quad nn_{p_{14.4}} = 3 \]

\[ P_{p_{14.3}} = \{p_7, p_{10}, p_{12}\}, \quad n^+_{p_{14.3}} = 0, \quad nn_{p_{14.3}} = 3 \]

\[ P_{p_{14.2}} = \{p_{10}, p_{12}\}, \quad n^+_{p_{14.2}} = 0, \quad nn_{p_{14.2}} = 2 \]

\[ EvQ = \{p_{14.3}(p_7)\} \quad p_8 \text{ becomes an outlier for } k = 4 \text{ but a safe inlier for } k < 4 \]

\[ D(R, 4) = \{ ..., p_8, p_{14}, ... \} \quad D(R, 3) = \{ ..., ..., ... \} \quad D(R, 2) = \{ ..., ..., ... \} \]

\[ J(R, 4) = \{ ..., ..., ... \} \quad J(R, 3) = \{ ..., p_8, p_{14}, ... \} \quad J(R, 3) = \{ ..., p_8, p_{14}, ... \} \]
ACOD - Advanced Continuous Outlier Detection

- Both parameters $R$, $k$ may vary
- Need to store succeeding neighbors $S_p$ as well
- Keep the nearest or the most recent preceding neighbors?
  - Keeping the most recent may affect a query with $q. R < R_{max}$
  - Keeping the closest ones may also lead to skipping a neighbour
- Problem is solved using $k$-I-skyband query
ACOD – Skyband Query

- 2-skyband query
- Each selected point has at most 2 other points at a lower x AND y position
- Set x as expiration time and y as distance
- \((k - 1)\)-skyband objects are stored as \(P_p\)
- \(P_p\) is calculated once and only the expired objects are discarded

\(P_p\) can be reduced to \((k - 1 - n'_p^+)\)-skyband where \(n'_p^+\) is the number of objects with distance < \(R_{min}\)
ACOD – Run

- $n'_{p_9} = 3 \quad R_{min} = 5 \quad R_{max} = 10 \quad k = 5$
- I-skyband query

- $P_{p_9} = \{p_1, p_2, p_3, p_6, p_7\}$
MCOD – Micro-cluster-based Continuous Outlier Detection

- Reduce computation complexity by reducing the number of neighbors that are taken into account
- Micro-clusters are created containing inliers
  - With radius R/2
  - And at least k+1 objects
  - Each object belongs to at most one micro-cluster
  - All objects in micro-clusters are definitely inliers
- Objects outside the micro-clusters are potential outliers ($PD$)
  - Only the objects in ($PD$) are considered in range queries
  - Event Queue only contains objects in ($PD$)
  - Efficient when objects appear in high density areas
MCOD – Micro-cluster-based Continuous Outlier Detection

Efficient and flexible algorithms for monitoring distance-based outliers over data streams
AMCOD – Advanced Micro-cluster-based Continuous Outlier Detection

- Supports multiple values of k and R
- Similarly to ACOD and using
  - $k_{max} - 1$ at least objects in each micro-cluster
  - $R_{min}/2$ as the maximum micro-cluster radius
Evaluation

- Algorithms implemented in C++ on single-core CPU using 1GB of RAM
- Two real-life and two synthetic data sets
  - FC (Forest Cover): 581,012 objects
  - ZIL (Zillow): 1,252,208 objects
  - IND: 5M objects following a uniform distribution
  - GAU: 5M objects with 60% uniform distribution and 40% following 4 different Gaussian distributions
- Comparison between COD, MCOD and Abstract-C (for varying k and fixed R)
- Comparison between COD, MCOD and ACOD, AMCOD
- Measuring CPU time, memory consumption and number of distance computations
- Each measurement corresponds to 1000 insertions/deletions
Evaluation – Default parameters

- $W = 200K = n$
- $k = 10$
- $R$ s.t. the number of outliers is $\sim 1\%$ of $n$
- $Slide = 1$
  - However, Abstract-C does not scale very well for $Slide = 1$
  - $Slide = 0.1\%$ of $W$ explicitly for Abstract-C
- LUE
COD vs MCOD vs Abstract-C for varying values of W

In IND the number of micro-clusters is very small

For small W the maintenance of micro-clusters is too expensive compared to the gain

Fig. 3. Running time vs. active objects (COD, MCOD and Abstract-C): (a) FC, (b) ZIL, (c) IND, (d) GAU.
COD vs MCOD vs Abstract-C for varying values of Slide

- COD and MCOD are not affected by the Slide size.

- Abstract-C gets better in big Slide sizes.

- Suitable for outlier snapshots and approximations but not for continuous monitoring.

**Fig. 4.** Running time vs. slide (COD, MCOD and Abstract-C): (a) FC, (b) ZIL, (c) IND, (d) GAU.
COD vs MCOD for varying values of outliers

In most of the cases MCOD is better than COD

With a large number of outliers the benefits of MCOD are reduced

Fig. 5. Running time vs. number of outliers (COD, MCOD and Abstract-C): (a) FC, (b) ZIL, (c) IND, (d) GAU.
ACOD vs AMCOD for varying values of $W$

AMCOD performs better especially in benchmarks with no uniform distribution, where micro-clusters are created more easily.

**Fig. 6.** Running time vs. active objects (ACOD and AMCOD): (a) FC, (b) ZIL, (c) IND, (d) GAU.
ACOD vs AMCOD for varying values of outliers

Also, ACOD and AMCOD perform worse than COD and MCOD since there is a single query per experiment and ACOD, AMCOD are meant to be used for multiple values of R.

Fig. 7. Running time vs. number of outliers (ACOD and AMCOD): (a) FC, (b) ZIL, (c) IND, (d) GAU.
COD vs MCOD vs ACOD vs AMCOD for varying values of $k$

$FC: R = 42 \quad k \in [5,10]$

$ZIL: R = 3600 \quad k \in [5,10]$  

$IND: R = 73.5 \quad k \in [5,14]$  

$GAU: R = 63 \quad k \in [5,14]$

Fig. 8. Running time vs. number of queries with different values of $k$: (a) FC, (b) ZIL, (c) IND, (d) GAU.
COD vs MCOD vs ACOD vs AMCOD for varying values of R and k

Fig. 9. Running time vs. number of queries with different values of R and k: (a) FC, (b) ZIL, (c) IND, (d) GAU.

Efficient and flexible algorithms for monitoring distance-based outliers over data streams
COD vs MCOD vs ACOD vs AMCOD

Event analysis for varying values of outliers

<table>
<thead>
<tr>
<th>Outliers (%W)</th>
<th>Algorithm</th>
<th>#events (K)</th>
<th>#events triggered (avg)</th>
<th>#events processed (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>COD</td>
<td>198.5</td>
<td>1.3</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>ACOD</td>
<td>144.4</td>
<td>9.6</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>MCOD</td>
<td>16.1</td>
<td>1.1</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>AMCOD</td>
<td>8.6</td>
<td>2.2</td>
<td>0.44</td>
</tr>
<tr>
<td>0.5</td>
<td>COD</td>
<td>196.2</td>
<td>1.9</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>ACOD</td>
<td>145.4</td>
<td>10.3</td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td>MCOD</td>
<td>46.6</td>
<td>1.9</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>AMCOD</td>
<td>28.3</td>
<td>5.1</td>
<td>1.39</td>
</tr>
<tr>
<td>1</td>
<td>COD</td>
<td>194.5</td>
<td>2.4</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>ACOD</td>
<td>146.4</td>
<td>10.6</td>
<td>2.04</td>
</tr>
<tr>
<td></td>
<td>MCOD</td>
<td>65.9</td>
<td>2.5</td>
<td>1.08</td>
</tr>
<tr>
<td></td>
<td>AMCOD</td>
<td>43.1</td>
<td>6.5</td>
<td>1.92</td>
</tr>
<tr>
<td>3</td>
<td>COD</td>
<td>189.5</td>
<td>3.3</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>ACOD</td>
<td>149.3</td>
<td>10.2</td>
<td>2.49</td>
</tr>
<tr>
<td></td>
<td>MCOD</td>
<td>113.3</td>
<td>3.5</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>AMCOD</td>
<td>84.8</td>
<td>8.3</td>
<td>2.35</td>
</tr>
</tbody>
</table>

Events in COD are reduced with the increment of the outliers

events processed < events triggered
ACOD vs AMCOD  
Micro-cluster effect for varying values of $W$

<table>
<thead>
<tr>
<th>$W$</th>
<th>ZIL</th>
<th>ACOD CPU RQ (s)</th>
<th>ACOD DC (K)</th>
<th>AMCOD CPU RQ (s)</th>
<th>AMCOD DC (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>1.04</td>
<td>0.08</td>
<td>61.0</td>
<td>0.17</td>
<td>269.2</td>
</tr>
<tr>
<td>200,000</td>
<td>2.52</td>
<td>0.22</td>
<td>425.8</td>
<td>0.27</td>
<td>548.5</td>
</tr>
<tr>
<td>300,000</td>
<td>6.98</td>
<td>0.27</td>
<td>577.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td>10.25</td>
<td>0.27</td>
<td>577.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500,000</td>
<td>16.06</td>
<td>0.27</td>
<td>577.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU RQ is the CPU time consumed for range queries.

DC is the total distance computations.
# COD vs MCOD vs ACOD vs AMCOD

Memory requirements for varying values of W

<table>
<thead>
<tr>
<th>W</th>
<th>FC</th>
<th>COD</th>
<th>ACOD</th>
<th>MCOD</th>
<th>AMCOD</th>
<th>ZIL</th>
<th>COD</th>
<th>ACOD</th>
<th>MCOD</th>
<th>AMCOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0.46</td>
<td>2.95</td>
<td>0.27</td>
<td>0.27</td>
<td>0.48</td>
<td>4.29</td>
<td>0.11</td>
<td>0.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td>9.58</td>
<td>100.45</td>
<td>4.94</td>
<td>5.49</td>
<td>9.60</td>
<td>111.85</td>
<td>2.74</td>
<td>3.11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300,000</td>
<td>14.11</td>
<td>133.27</td>
<td>11.04</td>
<td>13.12</td>
<td>14.47</td>
<td>194.28</td>
<td>4.01</td>
<td>4.61</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td>18.76</td>
<td>178.30</td>
<td>15.40</td>
<td>18.78</td>
<td>19.32</td>
<td>280.37</td>
<td>5.43</td>
<td>6.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500,000</td>
<td>23.52</td>
<td>232.72</td>
<td>20.23</td>
<td>25.15</td>
<td>24.23</td>
<td>377.51</td>
<td>6.56</td>
<td>7.66</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average number of neighbors maintained per object.

<table>
<thead>
<tr>
<th>W</th>
<th>FC</th>
<th>COD</th>
<th>ACOD</th>
<th>MCOD</th>
<th>AMCOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>9.2</td>
<td>32.2</td>
<td>1.7</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td>9.6</td>
<td>51.8</td>
<td>1.2</td>
<td>4.7</td>
<td></td>
</tr>
<tr>
<td>300,000</td>
<td>9.4</td>
<td>45.6</td>
<td>2.5</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td>9.4</td>
<td>45.8</td>
<td>2.8</td>
<td>10.5</td>
<td></td>
</tr>
<tr>
<td>500,000</td>
<td>9.4</td>
<td>47.8</td>
<td>2.8</td>
<td>11.2</td>
<td></td>
</tr>
</tbody>
</table>

Event handling variations (W = 200 K, outliers = 1%W).

<table>
<thead>
<tr>
<th>Events</th>
<th>IND</th>
<th>FC</th>
<th>ZIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#events (in K)</td>
<td>164.4</td>
<td>79.9</td>
<td>194.5</td>
</tr>
<tr>
<td>#events triggered</td>
<td>6.07</td>
<td>0.12</td>
<td>2.37</td>
</tr>
<tr>
<td>#events inserted</td>
<td>1.55</td>
<td>1.10</td>
<td>1.91</td>
</tr>
<tr>
<td>#increasetime ops</td>
<td>–</td>
<td>9.81</td>
<td>–</td>
</tr>
</tbody>
</table>

Efficient and flexible algorithms for monitoring distance-based outliers over data streams
Conclusions

- All algorithms outrun the Abstract-C algorithm
  - Except for the cases where Slide parameter is high
  - Thus, Abstract-C is suitable for snapshots or approximations of the outliers
- MCOD and AMCOD get better results than COD and ACOD
  - Except in highly distributed objects and high number of outliers
Any Questions?