Exploring Probabilistic Data Structures for Privacy-Preserving Pedestrian Dynamic Analysis

HRISTO BIZHEV, University of Twente, Netherlands

Monitoring pedestrian dynamics is critical for urban planning, resource allocation, and public safety. Traditional methods of data collection pose significant privacy concerns, necessitating the use of privacy-preserving techniques. This paper analyzes the functionality of five prominent probabilistic data structures for pedestrian counting and crowd monitoring and evaluates the performance of two selected ones, Bloom Filters and HyperLogLog. We explore their effectiveness in estimating set cardinality, union, and intersection sizes across varying parameters. Experimental results indicate that both data structures have similar accuracy, with relative errors around 0.43% for set cardinality and union estimations. However, Bloom Filters demonstrate significantly better performance in terms of execution time and memory usage, being five times faster and more than 40 times more space-efficient than HyperLogLog. Despite HyperLogLog's slightly better accuracy in intersection estimations, Bloom Filters' overall efficiency makes them more suitable for real-time applications.

Additional Key Words and Phrases: probabilistic data structures, privacy-preserving, crowd monitoring, pedestrian dynamics

1 INTRODUCTION

Monitoring pedestrian dynamics and movement patterns is crucial for various applications, including urban planning [1], resource allocation and public safety [2]. City planners require accurate pedestrian data about how numerous the crowds are and how they move to optimize infrastructure, transportation systems, and public spaces. Emergency services need real-time information on crowd densities to respond effectively in case of incidents or disasters. Businesses and event organizers could also use the data for crowd analysis and control [3, 4].

Automatically measuring pedestrians' dynamics allows for the collection of more accurate data in a more convenient manner than manual methods. In these methods, scanners are placed in public areas to gather unique identifiers of people like MAC addresses or public transport card information for each person to automate the process. Collecting these identifiers allows interested parties to approximate the number of people around the scanners and the flow of movement between them. Dealing with data of crowds has always been a sensitive matter regardless of the monitoring method as the people involved are worried about their privacy. Gathering unique identifiers of people in different places over a period of time can endanger their privacy since they can be identified again [5]. These actions are forbidden according to the European General Data Protection Regulation (GDPR) [6]. As an instance of automatically analyzing pedestrian dynamics, the widespread adoption of mobile devices has introduced a new approach. This method utilizes signals emitted by these devices, such as Wi-Fi probe requests. Although the method enables automatic pedestrian tracking, resulting in more efficient and accurate data collection compared to manual methods, it also highlights significant concerns regarding the privacy of the collected data.

Traditional methods, which use storing and processing raw data containing sensitive information, are no longer viable due to data protection regulations. The proposed solution uses probabilistic data structures as they only retain hashed parts of the data or fingerprints representing it while using fewer resources than standard databases. Of the five analyzed, two have the required functions of set and intersection cardinality calculation for analyzing pedestrian dynamics - Bloom Filter and HyperLogLog.

In this paper, we discuss the potential solutions based on the conducted experiment and previous research with a focus on functionality and performance.

2 PROBLEM STATEMENT

In this research field, most of the works focus on implementations in different fields [7, 8] and the impact on privacy[9]. While these are indeed important topics, there is a lack of analysis on the performance of the different probabilistic data structures. They are important resource-preserving instruments; understanding their nuances is key to choosing and implementing one for your system. This paper will analyse the performance of different data structures and explore how different parameters impact the relation between pedestrian count accuracy and privacy preservation.

The problem statement leads to the following research questions:

• RQ1: How effective are Bloom Filters, HyperLogLog, and other probabilistic data structures in accurately counting pedestrians?

• RQ2: How do parameters such as filter size, hash functions, and error rates impact the performance of probabilistic data structures in pedestrian counting?

3 BACKGROUND

3.1 Pedestrian dynamics

Pedestrian dynamics encompasses the study of how people move and interact in various environments, providing valuable insights into crowd behaviour and movement patterns. Pedestrian dynamics studies how people move and interact in different environments. It gives valuable insight into crowd behaviour and movement patterns. Two key metrics in this field are footfall and crowd flow, terminology formally defined in [10]. Footfall represents the count of pedestrians passing through a particular area, in our case those detected by sensors, over a specific period. Crowd

TScIT 41, July 5th, 2024, Enschede, The Netherlands
© 2024 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in , https://doi.org/10.1145/nnnnnnn.nnnnnnn.

flow involves people's movement patterns through different areas, which can be visualized by identifying the groups of people detected at 2 or more different locations. If we are to assume the data is gathered in sets, then to realize the needed statistical data we shall need a system that has the functions of calculating set cardinality and finding the intersections of 2 or more sets while being fast and efficient enough to accomplish those real-time analytics.

3.2 Probabilistic data structures

Probabilistic data structures are one of the most prominent solutions for handling large-scale data efficiently, often at the cost of some accuracy [11]. They are beneficial in scenarios where approximate results are acceptable and offer significant memory and computation time savings. The pedestrian count does not require to be exact to be useful therefore they are a logical choice to explore as their low space and time complexity makes them more suited for real-time analysis than deterministic data structures. The research conducted on available probabilistic data structures identified five key structures, as shown in Table 1: Bloom Filter, HyperLogLog, Count-min Sketch, Cuckoo Filter, and Quotient Filter. These structures were selected based on their ability to efficiently manage large-scale data with minimal memory and computational requirements, making them ideal for real-time analysis applications. Each of these data structures supports different functions that are crucial for various applications. Bloom Filters and HyperLogLog, in particular, fit the criteria because they support both cardinality estimation and intersection operations. These functions are essential for applications in pedestrian dynamics analysis where calculating the number of distinct elements (footfall) and identifying intersections (crowd flow) are critical tasks, therefore the focus of this paper will be on Bloom Filters and HyperLogLog.

3.2.1 Bloom Filter. Bloom Filters [12] is a space-efficient probabilistic data structure whose main functionality is to check if an element is part of a set. It is implemented using an array of bits and multiple independent hash functions. When an element is added to the Bloom Filter, it is hashed by each hash function, and the corresponding bits in the array are set to 1. Querying for an element requires that it is passed through all the hash functions and that the received bits be inspected. If any of them are 0's, then the element is guaranteed to not be in the filter. When all the bits are set to 1, the element is estimated to be in the set. Its probabilistic nature comes from the fact that it produces false positives as the query returns a positive result, it might be the case that the corresponding bits have been set to 1 by the addition of another element. The probability of any of these bits to be 1 is $(1-(1-\frac{1}{m})^{kn})$ for $\mathbf{m}$ bits, $\mathbf{k}$ hash functions and $\mathbf{n}$ inserted elements, which can be approximated to $(1-e^{-\frac{kn}{m}})$. Therefore when taking into account the number of hash functions, the probability of all selected bits being 1 and the algorithm returning an incorrect positive claim of membership is approximately $(1-e^{-\frac{kn}{m}})^k$. The authors of [8] show that for a finite Bloom filter, the false positive probability of the data structure does

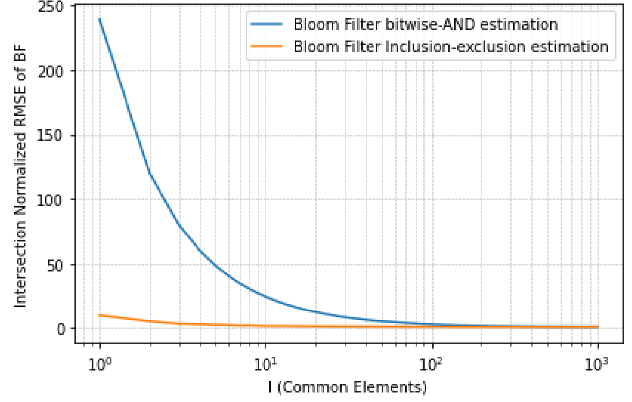

Fig. 1. The normalized RMSE of intersection estimations of Bloom Filter

not exceed $(1-e^{-\frac{k*(n+0.5)}{m-1}})^k$. As the optimal amount of hash functions can be derived from the false positive probability to be $\frac{m}{n}\ln 2$[13], the parameters of the algorithm can be adjusted to the expected number of entries and the needed error rate.

The count of the elements in BF can be approximated with the following formula: count(BF) $\approx -\frac{m}{k}\ln(1-\frac{|BF|}{m})$ where |BF| is the number of bits set to 1.

Operations on different Bloom Filters can be done by using bitwise OR and AND operations to create the union and intersection of the sets respectively. The union operation keeps the same False positive rate of the used algorithms, the count can be calculated using the same formula - $count(BF_1 \cup BF_2) \approx -\frac{m}{k}\ln(1-\frac{|BF_1 \cup BF_2|}{m})$ [7]. However, for intersections, this method will result in more false positives compared to creating another Bloom Filter (BF) with the common elements of the sets. This is because the AND operation might indicate matching bits that were set by different values, which are not actually in the intersection. To avoid this the intersection cardinality can be calculated using the inclusion-exclusion principle: $count(BF_1 \cap BF_2) = count(BF_1)+count(BF_2)-count(BF_1 \cup BF_2)$. Experimenting with both methods [Figure 1] shows that using the bitwise-AND function results in large error for relatively small intersection sizes. More about the metrics and the settings of the experiment are explained in Section 4.

In Bloom filters, the time complexity for adding and querying an element is O(k), and the time complexity for estimating the count is O(m). The space complexity is also O(m), showing that both scale linearly with the size of the filter.

Bloom Filters are widely used in various domains due to their efficiency in space and time. Bloom Filters help manage cache and reduce latency in web servers by quickly checking if a web object is cached [14] and are also used in systems like Apache Cassandra [15] and Google Bigtable [16] to minimize disk look-ups for non-existent rows, enhancing read performance. A lot of variants have been created to compensate for the specific limitations of the initially proposed algorithm or to improve its performance. Some examples of them are:

Data Structure	Membership Testing	Union	Intersection	Cardinality	Frequency	Deletion
Bloom Filter	Yes	Yes	**Yes**	**Yes**	No	No
HyperLogLog	No	**Yes**	No	**Yes**	No	No
Count-min Sketch	No	No	No	No	Yes	No
Cuckoo Filter	Yes	Yes	No	No	No	Yes
Quotient Filter	Yes	Yes	No	No	No	Yes

Table 1. Functions supported by the presented data structures

- Counting Bloom Filter: Supports deletion of elements by maintaining a count of the number of times a bit is set to 1 [14].
- Scalable Bloom Filter: Adapts dynamically to the number of stored elements while ensuring a maximum false positive probability, making it suitable for environments with constantly varying dataset sizes [17].
- Bloomier Filter: An extension that supports associating values with keys. It can be used to implement associative arrays with probabilistic space efficiency [18].

A thorough survey of most variations and their fields of deployment has been performed by the authors of [19]. Table 4 on page 31 of their paper summarizes the extensive comparison of the functionalities and complexity of the different implementations.

3.2.2 HyperLogLog. HyperLogLog [20] is a probabilistic data structure used for counting the number of distinct elements (cardinality estimation) in a multiset. It provides a trade-off between accuracy and memory usage, making it suitable for large-scale data processing tasks.

HyperLogLog works by applying uniform hashing on each element of the set and using the properties of hash functions to estimate the cardinality. It divides the hash space into multiple registers and keeps track of the maximum number of leading zeros observed in each register. If the longest leading sequence of zeros is n bits long, then a good estimation of the cardinality of the multiset is 2^n.

When an element is added to HyperLogLog, it is hashed, and the hash value is divided into two parts: the index part and the value part. The index part determines which register to update, while the value part is used to count the number of leading zeros plus one. This value updates the register if it is greater than the current value stored.

The probability of observing a certain number of leading zeros follows a geometric distribution. By averaging the observations across all registers and applying a harmonic mean, HyperLogLog produces an estimate of the cardinality. The formula used to estimate the cardinality is as follows: count(HLL) $\approx \alpha_m m^2 Z = \alpha_m m^2 (\sum_{j=1}^{m} 2^{-M[j]})^{-1}$, where α_m is a bias correction constant, $\mathbf{m}$ is the number of registers and $\mathbf{Z}$ is the harmonic mean of the values in the registers. To improve accuracy, HyperLogLog applies corrections for small and large cardinalities:

- Small Range Correction: If the estimate E $\leq \frac{5}{2}m$, the algorithm uses a linear counting approach [21].
- Large Range Correction: The algorithm applies a logarithmic correction for very large estimates to adjust for the maximum possible hash value.

HyperLogLog also has a merge function that allows the combination of multiple HyperLogLog structures to produce a single estimate representing the union of all the combined sets. To merge two HyperLogLog structures, you take the maximum value of each corresponding register and set it to the merged HyperLogLog: $HLL_{merged}[i] = max(HLL_1[i], HLL_2[i])$. This operation ensures that the resulting HyperLogLog structure accurately reflects the maximum number of leading zeros observed for each register across all merged structures.

As HyperLogLog does not directly support intersection operations, the cardinality of the intersection of two sets can be estimated using the principle of inclusion-exclusion and the estimates from the merged HyperLogLog structures in a similar fashion as for Bloom Filter: $count(HLL_1 \cap HLL_2) = count(HLL_1) + count(HLL_2) - count(HLL_1 \cup HLL_2)$.

The theoretical space complexity of HyperLogLog is O(log(log(n)). The time complexity for adding an element and getting the count are both constant at O(1) and O(m) respectively. Meanwhile, the merge operation has O(x*m) complexity for merging x amount of HLLs.

The most notable variation of the algorithm is HLL+ [22]. It achieves a reduction in memory usage and adjusts the bias for smaller cardinalities by replacing Linear counting. By using a 64-bit hash function instead of the 32-bit one as in the original papers, the hash collisions for large cardinalities are reduced, removing the need for large-range correction.

3.2.3 Non-counting data structures.

- Count-Min Sketch [23] is a probabilistic data structure that provides frequency estimation for data streams. It uses multiple hash functions to map elements to a series of counters, allowing it to estimate the frequency of elements with sub-linear memory usage. While it does not support direct cardinality or membership testing, it excels in scenarios where tracking the frequency of elements is essential, such as natural language processing [24]. Its functions could also be utilized to highlight stationary objects that are constantly detected by the sensors, e.g. printers and other IoT, but bear no value to the statistical data to be extracted [25].
- Cuckoo Filter [26] is an extension of the Bloom Filter that supports element deletion in addition to insertion and query operations. It uses cuckoo hashing

to manage collisions and stores fingerprints of the inserted elements, allowing it to handle dynamic data sets efficiently. An advantage it holds over its predecessor is that has better space efficiency and uses less time to perform membership queries. Cuckoo Filter is particularly useful in applications requiring frequent updates and deletions, such as database management systems and network security [27].
- Quotient Filter [28] is another alternative to Bloom Filters that supports membership queries and deletions. It uses a compact representation of a hash table where each entry contains a quotient and a remainder, making it space-efficient. Quotient Filter is beneficial for applications requiring dynamic membership testing with low memory overhead, such as file systems and distributed databases [29].

4 METHODOLOGY

4.1 Experiment Design

We implemented the base version of Bloom Filter and the 64-bit version of HyperLogLog [1] in Python and initialized them by the desired error rate for the experiment p = 1%. To gather the data, each setting will be iterated 100 times to ensure statistical reliability, for sets of n = 1000 numbers. The rest of the parameters of the algorithms are:

- To minimize the memory usage of the Bloom Filter, the size of the bit array is set to contain 2000 elements, therefore m = $\frac{-n*\ln p}{(\ln 2)^2} \approx 9585$ bits.
- As shown in [13] the false positive rate is minimized when the density of set bits in the array is 0.5, therefore the optimal number of hash functions used for Bloom Filter is k = $\frac{m}{n} * \ln 2 \approx 6$.
- The optimal amount of registers for HyperLogLog is m = $(\frac{1.04}{p})^2 \approx 10816$.
- MurMurHash3 is a non-cryptographic hash function that was used for both algorithms as it is highly uniform, which is a requirement for HLL, and a fast function, making it suitable for the situation.

4.2 Scenarios

To evaluate the performance and accuracy of Bloom Filter and HyperLogLog, a series of experiments are conducted focusing on the two primary crowd monitoring scenarios: footfall and crowd flow. The experiments were designed to measure the ability of these data structures to perform set and intersection cardinality counting under varying intersection sizes. The datasets used for the experiment were sets consisting of randomly generated numbers up to 10^{10} to simulate the different encrypted information given to the data structures. Two sets of equal length are generated for each iteration, with varying degrees of intersection to mimic different levels of crowd overlap covering all values for the sets - from 1 to 1000. The analyzed metrics for accuracy will be the relative error and Root-mean-square error (RMSE) of the cardinality, union and intersection estimations. The former is calculated by dividing the deviation from the actual value by the actual value, giving us a normalised error measure, and allowing comparisons across different scales and units. The former is the quadratic mean of the differences between the estimated and actual values and represents the standard deviation of the estimation errors.

[1] https://github.com/dituncho/probabilistic_data_structures

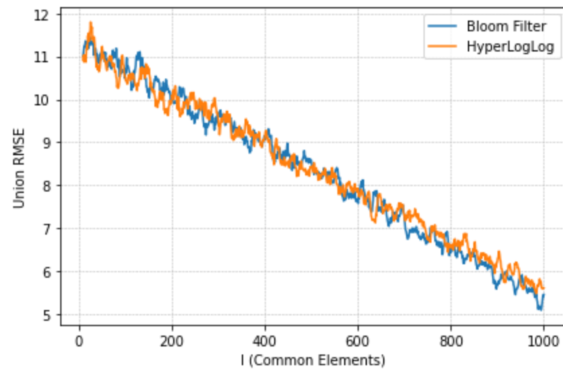

Fig. 2. The RMSE of union estimations of Bloom Filter and HyperLogLog for different intersection sizes

5 EXPERIMENT RESULTS

5.1 Set Cardinality

Computing the arithmetic mean of the relative error of set cardinality estimation of all iterations returns approximately 0.43% for both algorithms. Performing paired t-test on the differences gives us a p-value of 0.42 which shows for $\alpha = 0.05$ that there is no significant difference between the relative errors. The standard deviation of the estimation data is 0.00025, confirming that the iteration parameter has no bearing on the cardinality estimation functions.

5.2 Union and Intersection Cardinality

The results plotted in Figures 2 and 3 show that the deviation of the union estimations has a linear relation with the decreasing size of the union as the estimations have a mean relative error of approximately 0.44%.

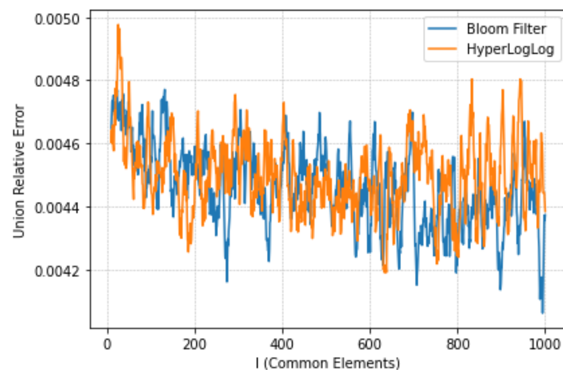

Fig. 3. The mean relative error of union estimations of Bloom Filter and HyperLogLog for different intersection sizes

To illustrate better the proportion of the intersection estimations, a normalization of RMSE will be used - Normalized

RMSE = $\frac{CorrectCount+RMSE}{CorrectCount}$. The metric is similar to Relative error but provides a more intuitive understanding of the error in terms of the actual count. The range of its values is from 1 to infinity, with 1 meaning the RMSE = 0 and the estimation is equal to the actual value and any bigger number representing the growing proportional difference between the observed and the correct values. In Figure 4 it can be seen that for intersection sizes below 10, the estimation can differentiate up to 8-9 times from the actual count. With intersections of bigger sizes, the mean normalized RMSE is 1.016 for both algorithms. Performing a paired t-test on these values returns p-value = $5 * 10^{-5}$ and a test statistic of 3.5, therefore it can confidently be said that the mean normalized RMSE of HLL is smaller.

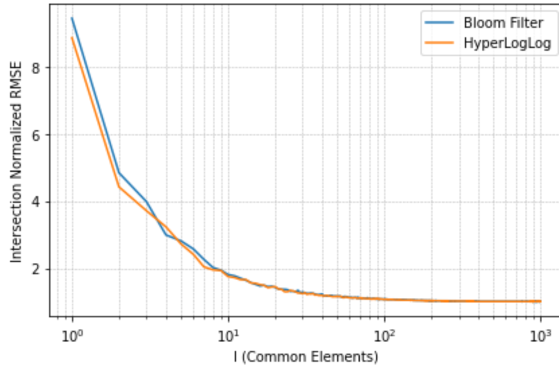

Fig. 4. Normalized RMSE of the intersection estimations of Bloom Filter and HyperLogLog for different intersection sizes

5.3 Performance

The tests were run on a Windows device with AMD Ryzen 7 5800H and 16 GB RAM. The accuracy of the data structure may be similar but their performance is not. In all cases, Bloom Filter was 5 times faster than HyperLogLog in adding all elements to the array, creating the union between the sets and estimating the cardinalities of the sets, union and intersection. The theoretical space for the implemented Bloom Filter and HyperLogLog would be 9585 bits and 10816 registers * 64 bits = 692224 bits. This would make Bloom Filter 72 times more space-efficient than HyperLogLog. However, as per the calculations of the library Pympler, the objects in Python used 23744 and 1055736 bits respectively. In comparison that would make the two data structures have a difference of 44 times instead.

6 DISCUSSION AND FUTURE WORK

This research provides a focused performance analysis on a limited pool of probabilistic data structures due to the given time frame. Here will be discussed part of the research that could be extended.

6.1 Privacy

An important part of crowd monitoring is the privacy and animosity of the people involved. While HyperLogLog provides the advantage of slightly lower deviation in estimations and does not support membership queries, which can enhance privacy, Bloom Filter's efficiency in speed and memory usage makes it a more suitable choice for real-time pedestrian monitoring applications. Moreover, there are already implementations of Bloom Filter in the field of crowd monitoring with a focus on privacy and encryption [10]. Obstacles like MAC randomization [9] and statistical techniques which take it into account are important to consider when applying the system in practice and outdoor environments.

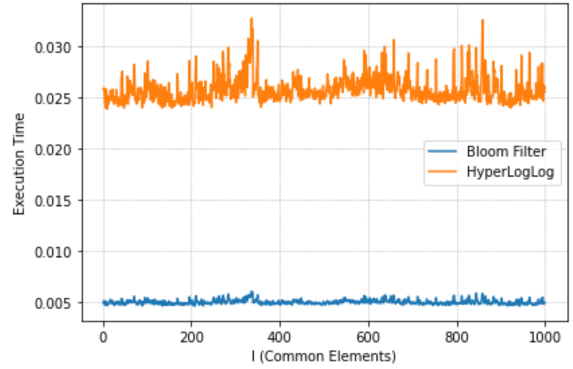

Fig. 5. Execution time of Bloom Filter and HyperLogLog

6.2 Scope

We gave an overview of the prominent probabilistic data structures but not all counting ones were analyzed. Also, the implementations are of the naive versions of the algorithms as mentioned in Section 4. If more thorough research is to be done, the rest will need to be taken into account as well. There is related work in the sphere which solely focuses on reviewing the relevant structures [11]. There is additionally a lot of effort into creating different variations and optimising them for specific use cases, e.g. for Bloom Filter [19], which offer additional flexibility and efficiency improvements.

7 CONCLUSION

This paper explored the performance and accuracy of Bloom Filters and HyperLogLog for privacy-preserving pedestrian dynamic analysis. The experiments demonstrated that both data structures provide similarly accurate cardinality and intersection estimations, with relative errors around 0.43% and 0.44% for set and union cardinalities, respectively. However, a paired t-test showed a statistically significant lower mean normalized RMSE for HyperLogLog in intersection cardinality estimations, indicating its slightly better accuracy.

Despite the comparable accuracy, Bloom Filter exhibited significantly better performance in terms of execution time and memory usage. Bloom Filter was consistently five times faster than HyperLogLog across all operations, including element addition, union creation, and cardinality estimation. Additionally, we found Bloom Filter to be 44 times more space-efficient based on the actual memory usage of the Python objects.

The parameters used for the algorithms are calculated by using mathematical formulas for the optimal values, provided by the cited research on them. The dynamic parameter of the conducted experiment was the intersection size. As it increased, the accuracy of the intersection cardinality exponentially increased as well, plateauing after passing 10% of the maximum tested intersection size.

8 USE OF AI TOOLS

During the preparation of this work, the author used ChatGPT and Grammarly to rephrase certain parts of the text for improved readability. After using these tools and services, the author reviewed and edited the content as needed and assumed full responsibility for the content of the work.

REFERENCES

[1] M. Southworth, “Designing the walkable city,” *Journal of Urban Planning and Development*, vol. 131, no. 4, pp. 246–257, 2005.
[2] C. Martella, J. Li, C. Conrado, and A. Vermeeren, “On current crowd management practices and the need for increased situation awareness, prediction, and intervention,” *Safety Science*, vol. 91, pp. 381–393, 2017.
[3] B. Bonne, A. Barzan, P. Quax, and W. Lamotte, “WiFiPi: Involuntary tracking of visitors at mass events,” in *2013 IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, (Madrid), pp. 1–6, IEEE, June 2013.
[4] A. Basalamah, “Crowd Mobility Analysis using WiFi Sniffers,” *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 12, 2016.
[5] M. Cunche, “I know your MAC address: targeted tracking of individual using Wi-Fi,” *Journal of Computer Virology and Hacking Techniques*, vol. 10, pp. 219–227, Nov. 2014.
[6] O. Radley-Gardner, H. Beale, and R. Zimmermann, eds., *Fundamental Texts On European Private Law*. Hart Publishing, 2016.
[7] S. J. Swamidass and P. Baldi, “Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval,” *Journal of Chemical Information and Modeling*, vol. 47, pp. 952–964, May 2007. Publisher: American Chemical Society.
[8] A. Goel and P. Gupta, “Small Subset Queries and Bloom Filters Using Ternary Associative Memories, with Applications,”
[9] M. Van Steen, V.-D. Stanciu, N. Shafaeipour, C. Chilipirea, C. Dobre, A. Peter, and M. Wang, “Challenges in Automated Measurement of Pedestrian Dynamics,” in *Distributed Applications and Interoperable Systems* (D. Eyers and S. Voulgaris, eds.), vol. 13272, pp. 187–199, Cham: Springer International Publishing, 2022. Series Title: Lecture Notes in Computer Science.
[10] V.-D. Stanciu, M. V. Steen, C. Dobre, and A. Peter, “Privacy-Preserving Crowd-Monitoring Using Bloom Filters and Homomorphic Encryption,” in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, (Online United Kingdom), pp. 37–42, ACM, Apr. 2021.
[11] A. Singh, S. Garg, R. Kaur, S. Batra, N. Kumar, and A. Y. Zomaya, “Probabilistic data structures for big data analytics: A comprehensive review,” *Knowledge-Based Systems*, vol. 188, p. 104987, Jan. 2020.
[12] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, pp. 422–426, July 1970.
[13] O. Papapetrou, W. Siberski, and W. Nejdl, “Cardinality estimation and dynamic length adaptation for Bloom filters,” *Distributed and Parallel Databases*, vol. 28, pp. 119–156, Dec. 2010.
[14] Li Fan, Pei Cao, J. Almeida, and A. Broder, “Summary cache: a scalable wide-area Web cache sharing protocol,” *IEEE/ACM Transactions on Networking*, vol. 8, pp. 281–293, June 2000.
[15] “Bloom Filters | Apache Cassandra Documentation.”
[16] G. Dinnyes and N. Suttle, “Cloud Bigtable improves single-row read throughput by up to 50 percent.”
[17] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, “Scalable Bloom Filters,” *Information Processing Letters*, vol. 101, pp. 255–261, Mar. 2007.
[18] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables,”
[19] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, “Optimizing Bloom Filter: Challenges, Solutions, and Comparisons,” Jan. 2019. arXiv:1804.04777 [cs].
[20] P. Flajolet, E. Fusy, and O. Gandouet, “HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm,”
[21] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, “A linear-time probabilistic counting algorithm for database applications,” *ACM Transactions on Database Systems*, vol. 15, pp. 208–229, June 1990.
[22] S. Heule, M. Nunkesser, and A. Hall, “HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm,” in *Proceedings of the 16th International Conference on Extending Database Technology*, (Genoa Italy), pp. 683–692, ACM, Mar. 2013.
[23] G. Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications,” *Journal of Algorithms*, vol. 55, pp. 58–75, Apr. 2005.
[24] P. Talukdar and W. Cohen, “Scaling Graph-based Semi Supervised Learning to Large Number of Labels Using Count-Min Sketch,” in *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pp. 940–947, PMLR, Apr. 2014. ISSN: 1938-7228.
[25] G. Cormode and S. Muthukrishnan, “What’s Hot and What’s Not: Tracking Most Frequent Items Dynamically,” *ACM Transactions on Database Systems*, vol. 30, pp. 249–278, Mar. 2005. Publisher: Association for Computing Machinery.
[26] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, “Cuckoo Filter: Practically Better Than Bloom,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, (Sydney Australia), pp. 75–88, ACM, Dec. 2014.
[27] Y. Zhao, W. Dai, S. Wang, L. Xi, S. Wang, and F. Zhang, “A Review of Cuckoo Filters for Privacy Protection and Their Applications,” *Electronics*, vol. 12, p. 2809, Jan. 2023. Number: 13 Publisher: Multidisciplinary Digital Publishing Institute.
[28] J. Clerry, “Compact Hash Tables Using Bidirectional Linear Probing,” *IEEE Transactions on Computers*, vol. C-33, pp. 828–834, Sept. 1984. Conference Name: IEEE Transactions on Computers.
[29] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, “Don’t thrash: how to cache your hash on flash,” *Proceedings of the VLDB Endowment*, vol. 5, pp. 1627–1637, July 2012.