Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

PDF Download
3744569.pdf
13 March 2026
Total Citations: 0
Total Downloads: 624

Latest updates: https://dl.acm.org/doi/10.1145/3744569

Published: 07 August 2025
Online AM: 12 June 2025
Accepted: 30 May 2025
Revised: 31 March 2025
Received: 11 April 2024

Citation in BibTeX format

RESEARCH-ARTICLE

Hierarchical Spatial Decompositions under Local Differential Privacy

ECE ALPTEKIN, Koç University, Istanbul, Turkey

BERKAY KEMAL BALIOGLU, Koç University, Istanbul, Turkey

M. EMRE GURSOY, Koç University, Istanbul, Turkey

Open Access Support provided by:

Koç University

Hierarchical Spatial Decompositions under Local Differential Privacy

ECE ALPTEKIN, BERKAY KEMAL BALIOGLU, and M. EMRE GURSOY, Department of Computer Engineering, Koç University, Istanbul, Turkey

The popularity of smartphones, GPS-enabled devices, social networks, and connected vehicles all contribute to the increasing volume of spatial data. Spatial decompositions assist in handling big spatial data, and they have been commonly used in the Differential Privacy (DP) literature for range query answering, spatial indexing, count-of-counts histograms, data summarization, and visualization. However, their applications under the emerging Local DP (LDP) notion are scarce. In this article, we study the problem of building hierarchical spatial decompositions under LDP, focusing on two methods: quadtrees and kd-trees. We develop two solutions for quadtrees: a baseline solution that is inspired by the centralized DP literature, and a proposed solution that utilizes a single data collection step from users, propagates density estimates to remaining nodes, and performs structural corrections to the quadtree. Since kd-trees rely on node medians which are data-dependent, we observe that it is not feasible to build kd-trees using a single data collection step. We therefore propose an iterative solution that constructs kd-trees in top-down fashion by utilizing a novel algorithm for estimating node medians at each tree depth. We experimentally evaluate our quadtree and kd-tree algorithms using four real-world spatial datasets, multiple utility metrics, varying privacy budgets, and tree parameters. Results demonstrate that our algorithms enable the building of accurate spatial decompositions that provide high utility in practice. Notably, our quadtrees and kd-trees achieve substantially lower errors in answering spatial density queries (up to 10-fold improvement) when compared with a state-of-the-art method.

CCS Concepts: • **Information systems** → **Data structures**; **Spatial-temporal systems**; *Data mining*; • **Security and privacy** → **Data anonymization and sanitization**;

Additional Key Words and Phrases: Local differential privacy, location-based services, spatial data mining, spatial-temporal systems

Associate Editor: Jundong Li

ACM Reference format:
Ece Alptekin, Berkay Kemal Balioglu, and M. Emre Gursoy. 2025. Hierarchical Spatial Decompositions under Local Differential Privacy. *ACM Trans. Knowl. Discov. Data.* 19, 7, Article 128 (August 2025), 37 pages.
https://doi.org/10.1145/3744569

We sincerely thank the Scientific and Technological Research Council of Turkiye (TUBITAK) for their support under project number 121E303.

Authors' Contact Information: Ece Alptekin, Department of Computer Engineering, Koç University, Istanbul, Turkey; e-mail: ealptekin21@ku.edu.tr; Berkay Kemal Balioglu, Department of Computer Engineering, Koç University, Istanbul, Turkey; e-mail: bbalioglu23@ku.edu.tr; M. Emre Gursoy (corresponding author), Department of Computer Engineering, Koç University, Istanbul, Turkey; e-mail: emregursoy@ku.edu.tr.



This work is licensed under Creative Commons Attribution International 4.0.

© 2025 Copyright held by the owner/author(s).
ACM 1556-472X/2025/8-ART128
https://doi.org/10.1145/3744569

1 Introduction

The rising popularity of smartphones, GPS-enabled mobile devices, social networks, connected vehicles, and the Internet of Vehicles continues to increase the usage of **Location-Based Services (LBSs)** and applications. As a consequence, large volumes of spatial data are nowadays available for collection, storage, and analysis. However, ensuring the privacy of spatial data is crucial because it contains sensitive information about an individual's movements, habits, and preferences, potentially revealing personal details such as home and work addresses, frequented places, and social relationships.

Local Differential Privacy (LDP) has recently emerged as a state-of-the-art privacy standard in academia and the industry [8, 11, 15, 21, 48]. With the rising adoption of LDP and the need to protect users' location privacy, there is increasing interest in applying LDP to spatial data. Indeed, LDP has been applied to spatial data aggregation [6], LBS usage [23, 39], spatial query answering [38], trajectory collection [47], and sharing [10, 12], as well as indoor positioning [26, 31]. On the other hand, *spatial decompositions*, which are a common approach to dealing with big spatial data, have not received much attention under LDP. Spatial decompositions partition (decompose) the geographical space into smaller subspaces through the likes of tree-based and grid-based data structures. They have been used in the centralized DP literature for range query answering [9, 36, 45], spatial indexing and modeling [32, 50], count-of-counts histograms [27], data summarization and visualization [3, 33], as well as synthetic data generation [18, 20, 22]; however, their applications in LDP are relatively scarce.

Motivated by the above, in this article, we study the problem of building hierarchical spatial decompositions for spatial data under LDP. In particular, we focus on two types of popular decompositions: quadtrees and kd-trees. For quadtrees, we first propose a *baseline solution* inspired by a method from the centralized DP literature [50] and adapt it to LDP by making modifications. However, we observe that this baseline solution yields low utility in practice. This is because the baseline solution divides the total ε privacy budget into several pieces, each used for LDP estimation in one depth of the quadtree. Since the privacy budget used per estimation is smaller, higher noise is added to satisfy LDP, which causes noisy node densities and erroneous quadtree structures since estimation results also affect whether an internal tree node should split (have children). To address these weaknesses, we then develop a new, *proposed solution* which performs a single LDP estimation. This single-step estimation is performed at the hypothetical leaves of the quadtree and results in low noise, thereby enabling high-quality estimates to be obtained at the leaves. Then, leaves' densities are propagated upwards (in bottom-up fashion) to populate the densities of remaining nodes. Finally, structural corrections and consistency are achieved for the quadtree in a top-down manner.

For kd-trees, we observe that it is not possible to perform a single step of estimation and then propagate the densities since kd-trees rely on node medians which are data-dependent. Therefore, we propose a solution that iteratively constructs a kd-tree in top-down fashion, which utilizes a novel algorithm for estimating the medians of nodes at each depth of the kd-tree. Our algorithm bucketizes the geographical regions covered by a kd-tree node, obtains LDP density estimates for each bucket, and then finds which bucket contains the median value based on the estimates. To determine the densities of children nodes, we propose two strategies: (i) our initial strategy uses the LDP estimates obtained in the previous step, and (ii) our improved strategy evenly divides the parent's density. We experimentally observe that the improved strategy, albeit simpler, provides higher accuracy than the initial strategy. This is because our median estimation algorithm typically achieves high accuracy in determining the median (higher than bucket-wise density estimation), therefore evenly dividing based on the LDP median is effective.

We experimentally evaluate our quadtree and kd-tree algorithms using four real-world spatial datasets, three utility metrics (**Average Query Error (AQE)**, **Tree Edit Distance (TED)**, **Node**

Density Difference (NDD)), and typical privacy budget values such as $\varepsilon = 0.1, 0.5, 1, 2$. For quadtrees, results show that the proposed solution consistently outperforms the baseline solution in a variety of settings and quadtree parameters. For kd-trees, results show that the improved strategy yields lower error compared to the initial strategy in the majority of the experiments. Furthermore, upon experimenting with our novel LDP median estimation method, we observe that it can estimate LDP medians with generally $\leq$2% error (compared to the true medians) when $\varepsilon \geq 0.5$. Finally, we compare our quadtree and kd-tree algorithms against a state-of-the-art work from the literature for **Answering Spatial Density Queries under LDP (ASDQ-LDP)** [38]. Comparison results show that our algorithms achieve substantially lower errors under various datasets and ε values, offering up to 10-fold improvement in some settings.

Contributions. In short, the contributions of this article can be summarized as follows:

(1) We propose two solutions for building quadtrees under LDP: A *baseline solution* inspired from the centralized DP literature, and a *proposed solution* which relies on a single step of LDP data collection to address the iterative noise accumulation in the baseline solution.
(2) We propose a solution that iteratively constructs a kd-tree under LDP in top-down fashion, which internally utilizes a novel algorithm for estimating nodes' medians at each depth. Furthermore, two strategies are proposed and evaluated for determining the densities of children nodes in kd-trees.
(3) We experimentally evaluate all algorithms and solutions using four real-world spatial datasets, multiple ε values, multiple metrics, and parameters. Results demonstrate that our algorithms are effective, and they result in accurate spatial decompositions that can be used in practice.
(4) Finally, we experimentally compare our quadtrees and kd-trees against ASDQ-LDP [38], a state-of-the-art work for answering spatial density queries under LDP. Results show that our quadtrees and kd-trees achieve substantially lower errors under various settings.

Technical Challenges. While our article studies the problem of building quadtrees and kd-trees under LDP, our contributions are far from trivial. Our work solves several algorithmic challenges (e.g., improving utility by proposing new algorithms instead of just adding LDP to existing ones), theoretical challenges (e.g., novel privacy proofs and complexity analyses), and empirical challenges (e.g., extensive evaluation of proposed methods, comparisons with each other and prior work):

— Our solutions go beyond adding LDP to existing non-private algorithms or adapting DP algorithms to satisfy LDP. (We note that even such adaptations would have intellectual value, as they necessitate clever combinations of LDP blocks with existing algorithms.) For example, as our results show, adapting the DP quadtree solution to LDP yields high utility loss. This motivates our proposal of a new solution ("proposed" solution) for quadtrees. Similarly, new algorithms are necessary for building kd-trees under LDP. For example, a fundamental building block for kd-tree construction is finding medians under LDP. We propose an algorithm to solve this problem, which can also be adapted and re-used in contexts other than kd-trees.
— Proposing LDP solutions brings novel theoretical challenges in terms of new privacy proofs and complexity analyses. We prove that our solutions satisfy ε-LDP and perform their computational complexity analyses using the big-O notation. A good example of a technical challenge here is Theorem 1. In this theorem, it is necessary to show that our specific use of the **Optimized Unary Encoding (OUE)** protocol satisfies LDP. The existing LDP proof for OUE assumes that users' bitvectors are unary. However, because of the way in which our solution uses OUE, this assumption does not hold in our case. Thus, a new and advanced proof had to be developed to address this challenge.
— In terms of empirical challenges, extensive experiments are conducted to analyze how our solutions behave under different privacy budgets ε and tree parameters h^* and θ. Impacts of

data sparsity and skewness are analyzed. Our LDP median estimation algorithm is individually evaluated using its internal parameter. Through these experiments, we are able to provide recommendations regarding when to use quadtrees versus kd-trees. Furthermore, we show that our quadtrees and kd-trees provide higher utility than a state-of-the-art work, ASDQ-LDP [38], in the task of answering spatial density queries under LDP. This is another way in which our work advances the state-of-the-art.

Differences from Prior Conference Publication. This article builds on top of our prior conference publication [1]. In [1], we proposed solutions for building quadtrees under LDP and experimentally evaluated them. However, [1] did not contain any algorithms or experiments regarding kd-trees. Furthermore, [1] did not contain experimental comparisons against ASDQ-LDP [38]. Therefore, in terms of the four contributions listed above, this article contains more than 50% novelty compared to [1]. Contribution 1 and half of contribution 3 (experiments related to quadtrees) overlap with [1], whereas contributions 2 and 4, as well as the remaining half of contribution 3 (experiments related to kd-trees and our LDP median algorithm) are novel in this article. Section 6 is also completely new in this article.

Article Organization. The rest of this article is organized as follows. In Section 2, we establish preliminaries regarding the data model, LDP, quadtrees, and kd-trees. In Section 3, we provide our algorithms for building quadtrees under LDP. In Section 4, we provide our algorithms for building kd-trees under LDP. Experimental evaluations are reported and experiment results are discussed in Section 5. Further discussions and analyses are given in Section 6. Finally, we survey related work in Section 7 and conclude in Section 8.

2 Preliminaries

2.1 Data Model and Background

Let Ω denote the two-dimensional geographical space subject to decomposition, and let $\mathcal{U} = \{u_1, u_2, u_3, \ldots\}$ denote the set of users. The number of users is represented by $n = |\mathcal{U}|$. The true location of each user $u_i \in \mathcal{U}$ is denoted by l_i, which is a tuple consisting of latitude and longitude coordinates. By definition, $l_i \in \Omega$ for all u_i. We denote by $l_i[x]$ the longitude of l_i (x-axis) and by $l_i[y]$ the latitude of l_i (y-axis). It is assumed that Ω itself is not sensitive but each user's location l_i is sensitive. For example, if users are from the city of Istanbul, then Ω corresponds to the geographic boundaries of Istanbul, which is publicly known. However, each l_i corresponds to specific GPS coordinates of user u_i, which is privacy-sensitive.

A spatial decomposition decomposes the geographical space Ω into smaller subspaces. There are mainly three types of spatial decompositions:

- Grid-based decompositions utilize uniform or data-adaptive grid structures to divide Ω into cells.
- Hierarchical decompositions utilize tree-based data structures (such as kd-trees or quadtrees) for dividing Ω into smaller regions, each represented using a tree node.
- Hybrid decompositions use a mixture of grid-based and hierarchical approaches, e.g., they organize the decomposition as a hierarchy of grids [22, 30].

In this article, we focus on hierarchical decompositions which have been commonly used in the literature for purposes such as range query answering [9, 36, 45], spatial indexing and modeling [32, 50], count-of-counts histograms [27], data summarization and visualization [3, 33], and synthetic data generation [19, 22]. Among hierarchical decompositions, we focus on two fundamental ones: quadtrees and kd-trees [9, 28, 34]. We propose methods to build quadtrees and kd-trees while protecting the privacy of each user's location l_i using LDP.

2.2 LDP

LDP is a popular privacy standard [8, 11, 15, 44] which enables each user to locally perturb their true data on their own device using a randomized algorithm Ψ and then send the perturbed output to the data collector (i.e., the server). After the server collects perturbed data from many users, it performs estimation to recover aggregate statistics pertaining to the general user population. In our context, the sensitive data that needs to be protected using LDP is each user's location. Accordingly, we formalize LDP as follows.

Definition 1 (ε-LDP). A randomized algorithm Ψ satisfies ε-LDP, where $\varepsilon > 0$, if and only if for any two inputs l_i, l_i^*, it holds that:

$$\forall y \in Range(\Psi) : \frac{\Pr[\Psi(l_i) = y]}{\Pr[\Psi(l_i^*) = y]} \leq e^{\varepsilon}, \tag{1}$$

where $Range(\Psi)$ stands for the set of all possible outputs of the algorithm Ψ.

Given the perturbed output y, ε-LDP ensures that the server (or any other third party who observes y) will not be able to distinguish between the user's actual location l_i and an incorrect location l_i^* beyond the probability odds ratio controlled by e^{ε}. The strength of the privacy protection is controlled by the parameter ε, commonly known as the *privacy budget*. Lower ε yields stronger privacy.

LDP satisfies the sequential composition property [13, 41, 49], which enables the total ε budget to be spent in multiple steps. This is beneficial in the construction of hierarchical spatial decompositions since we can construct each step of the hierarchy with a portion of the privacy budget, such that their sequential composition satisfies ε-LDP.

Definition 2 (Sequential Composition). Consider m algorithms $\Psi_1, \Psi_2, \ldots, \Psi_m$ such that each Ψ_j satisfies ε_j-LDP. Then, the sequential execution of $\Psi_1(l_i), \Psi_2(l_i), \ldots, \Psi_m(l_i)$ satisfies $(\sum_{j=1}^{m} \varepsilon_j)$-LDP.

The popularity of LDP has led to the development of several LDP protocols [17, 21, 40]. New systems and applications often use these protocols as building blocks. In this article, we use the *OUE* protocol as our building block due to its higher accuracy compared to many other protocols [40]. As in other protocols, OUE consists of two main components: (i) user-side encoding and perturbation to satisfy LDP on users' devices and (ii) server-side estimation after collecting perturbed data from the user population.

User-Side Perturbation in OUE. OUE assumes that the user's data is encoded as a unary bitvector (vector of bits), i.e., only one position in the user's vector contains a 1 bit and all remaining positions contain 0 bits. Let B_i denote user u_i's unary bitvector. The perturbation algorithm Ψ of OUE takes B_i as input and outputs perturbed bitvector B_i' as:

$$\Pr\Big[B_i'[j] = \Psi(B_i[j]) = 1\Big] = \begin{cases} \frac{1}{2} & \text{if } B_i[j] = 1 \\ \frac{1}{e^{\varepsilon}+1} & \text{if } B_i[j] = 0 \end{cases}. \tag{2}$$

In other words, each position $j \in [1, |B_i|]$ is considered independently from others, and the bit that exists in $B_i[j]$ is either kept or flipped according to the above probabilities. After this perturbation is complete, u_i sends the perturbed bitvector B_i' to the server.

Server-Side Estimation in OUE. The server receives perturbed bitvectors from many users, collectively denoted by $\{B_1', B_2', \ldots, B_n'\}$. Then, the server computes the reported counts of each position $j \in [1, |B_i|]$, which we denote by $\widehat{C}(j)$:

$$\widehat{C}(j) = \sum_{i=1}^{n} B_i'[j]. \tag{3}$$

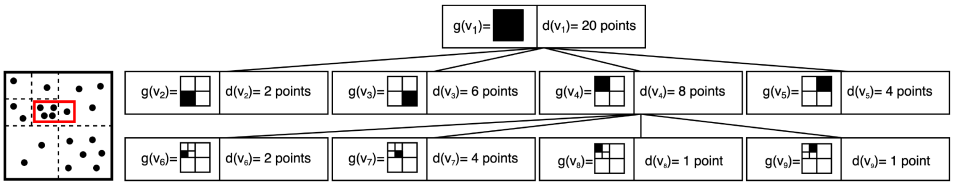


Fig. 1. Sample users with their locations represented as points (on the left) and the corresponding quadtree (on the right).

Finally, the server computes the estimate for position j, i.e., how many users have 1 bit in the jth position of their original bitvectors. This estimate, denoted by $\bar{C}(j)$, is computed as:

$$\bar{C}(j) = \frac{2 \cdot \left((e^{\varepsilon} + 1) \cdot \widehat{C}(j) - |\mathcal{U}|\right)}{e^{\varepsilon} - 1}. \tag{4}$$

2.3 Quadtrees

Quadtrees are a popular and well-known method for hierarchical spatial decomposition. A quadtree recursively divides the geographical space into four equi-sized quadrants (top left, top right, bottom left, bottom right) at each step. A sample quadtree is illustrated in Figure 1. The root corresponds to Ω as a whole. In the next level of the tree, Ω is divided into four quadrants, each of which corresponds to a child of the root node. The division of a node into four quadrants will keep occurring recursively until: (i) the maximum height limit h^* is reached, or (ii) the current node contains fewer location points than a pre-defined threshold θ.

We establish some notation and terminology with the help of Figure 1. Let Q denote a quadtree and let $\mathcal{V} = \{v_1, v_2, v_3, \ldots\}$ denote the set of nodes (vertices) in Q. The depth of a node v is denoted by $h(v)$, e.g., in Figure 1, $h(v_1) = 1$, $h(v_2) = h(v_3) = 2$, $h(v_7) = 3$. Nodes with the same $h(v)$ are found at the same depth of Q. Each node corresponds to a region of the geographical space as shown in Figure 1, e.g., v_1 corresponds to Ω, v_2 corresponds to bottom left quadrant of Ω, and so forth. We denote the geographical space corresponding to vertex v by $g(v)$. Furthermore, the number of data points located in the corresponding region of v is called the *density* of v and it is denoted by $d(v)$. In Figure 1, $d(v_1) = 20$, $d(v_2) = 2$, $d(v_3) = 6$, and so on. By definition, the root node v_1 has density $d(v_1) = n$ since all users are assumed to be located within Ω.

We denote the children of a node v by $c(v)$, e.g., according to Figure 1, $c(v_4) = \{v_6, v_7, v_8, v_9\}$. By definition of quadtrees, if a node has children, it will always have exactly four children. If a node does not have children, then it is called a *leaf* node. The parent of a node v is denoted by $p(v)$, e.g., $p(v_6) = p(v_8) = v_4$. In general, let v_i denote a node with set of children $c(v_i) = \{v_{i1}, v_{i2}, v_{i3}, v_{i4}\}$. It holds that:

$$g(v_i) = g(v_{i1}) \cup g(v_{i2}) \cup g(v_{i3}) \cup g(v_{i4}), \tag{5}$$

and the pairwise intersection $g(v_{ij}) \cap g(v_{ik})$ is empty for all $j \neq k$. Furthermore, the sum of densities of all children equals the density of the parent:

$$d(v_i) = d(v_{i1}) + d(v_{i2}) + d(v_{i3}) + d(v_{i4}). \tag{6}$$

One of the key applications of quadtrees is in answering spatial count (density) queries [9, 28, 50]. Consider a query q. Initially, the answer of q is set to 0, i.e., $ans_q = 0$. Then, starting from the root, the quadtree is traversed in top-down fashion. For each node v that is visited:

(1) If $g(v)$ is disjoint from q, then v is ignored.
(2) If $g(v)$ is fully contained in q, then ans_q is increased by $d(v)$.

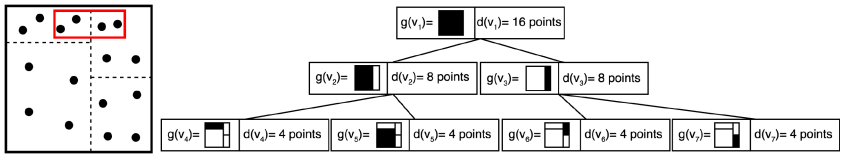


Fig. 2. Sample users with their locations represented as points (on the left) and the corresponding kd-tree (on the right).

(3) If $g(v)$ partially intersects q and v is not a leaf node, then every child of v with a region not disjoint from q must be visited.
(4) If $g(v)$ partially intersects q and v is a leaf node, then ans_q is incremented by:

$$ans_q = ans_q + d(v) \times \frac{||g(v) \cap q||}{||g(v)||}.$$

Here, $||\cdot||$ denotes the size of a geographical region.

We exemplify this process with the help of Figure 1. Let q be a query denoted using the red rectangle. Starting from v_1, we first find that $g(v)$ intersects with v_1, therefore v_1's children must be visited. Since v_2 and v_3 have no intersection with q, they are ignored. v_5 has intersection with q and it is a leaf node, therefore ans_q is incremented by $d(v_5) \times \frac{||g(v_5) \cap q||}{||g(v_5)||}$. v_4 has intersection with q but it is not a leaf node, therefore v_4's children must be visited next. Among v_4's children, v_6, v_8, and v_9 are ignored. Since v_7 intersects with q, ans_q is incremented by $d(v_7) \times \frac{||g(v_7) \cap q||}{||g(v_7)||}$.

2.4 Kd-trees

Another fundamental and popular method for hierarchical spatial decompositions is kd-trees (k-dimensional trees). At each step, a kd-tree alternates between the x-axis and the y-axis and divides the geographical space into two regions according to the median of the chosen axis. Since the division is according to the median, it is necessary to find the median in a data-dependent manner, and the resulting child nodes may have different geographic sizes. This is different from quadtrees in which the resulting child nodes have equal sizes. Similar to quadtrees, in kd-trees, the division of a node into two regions will keep occurring recursively until: (i) the maximum height limit h^* is reached, or (ii) the current node contains fewer location points than a pre-defined threshold θ.

A sample kd-tree is shown in Figure 2. The division at the root is performed with respect to the x-axis, according to the median of the location points. Hence, 16 points initially in v_1 are divided into v_2 and v_3 evenly. Note that the geographic sizes $g(v_2)$ and $g(v_3)$ are indeed different—since there are more points on the right compared to the left, the median is towards the right rather than the middle of the x-axis. Next, the y-axis is used when dividing v_2 and v_3. Since the median of v_2 is found independently from the median of v_3, the division of v_2 into v_4 and v_5 results in different sized nodes ($g(v_4)$ is small whereas $g(v_5)$ is large). In contrast, since the median of v_3 is towards the middle of the y-axis, the division results in similar sized v_6 and v_7. This highlights the need for determining the median of each node separately and in a data-dependent fashion.

For kd-trees, we inherit most of the notation established in Section 2.3 for quadtrees. To differentiate between a kd-tree and a quadtree, we denote a kd-tree by K. The nodes of K are denoted by $\mathcal{V} = \{v_1, v_2, v_3, \ldots\}$, the depth of node v is denoted by $h(v)$, the density of v is denoted by $d(v)$, the geographic space of v is denoted by $g(v)$, children of v are denoted by $c(v)$, and the parent of v is denoted by $p(v)$. In a kd-tree, a node can have no children or exactly two children. For node v_i and

its children $c(v_i) = \{v_{i1}, v_{i2}\}$, the following equation holds:

$$g(v_i) = g(v_{i1}) \cup g(v_{i2}). \tag{7}$$

We define two additional notations for kd-trees as follows: Let $g_x(v)$ denote the longitude bounds (x-axis bounds) of node v and let $g_y(v)$ denote the latitude bounds (y-axis bounds) of v. We denote by $min(g_x(v))$ the minimum longitude, $max(g_x(v))$ the maximum longitude, $min(g_y(v))$ the minimum latitude, and $max(g_y(v))$ the maximum latitude of v. We note that the application of kd-trees in answering spatial count queries is identical to that in quadtrees, with the difference that nodes in a kd-tree contain two children, whereas nodes in a quadtree contain four children.

3 Building Quadtrees under LDP

In this section, we describe two solutions for building quadtrees for spatial data under LDP. Our first solution is inspired by a differentially private quadtree building algorithm from the centralized DP literature, which we modify in order to adapt to LDP. We explain the centralized algorithm and the key modifications we need to perform to adapt it to LDP in Section 3.1. Then, our baseline solution is given in Section 3.2. We observe that although the iterative noise addition approach used in the baseline solution satisfies LDP, it is not desirable in practice since it causes large noise accumulation and therefore, low utility. This motivates the proposal of our new approach in Section 3.3, which yields higher utility than the baseline solution.

3.1 Quadtrees under Centralized DP

Consider the quadtree building algorithm presented in [50], which satisfies centralized DP. The algorithm starts by initializing quadtree Q with a root node v_1 such that $g(v_1) = \Omega$ and marks v_1 as unvisited. Then, the algorithm proceeds iteratively, and at each iteration, it checks if there is any unvisited node $v \in Q$. If there is an unvisited node, the algorithm computes the noisy density of v. For this purpose, the Laplace mechanism of DP can be used [13]. The noisy density is determined as: $\hat{d}(v) = d(v) + Lap(\lambda)$, where $Lap(\lambda)$ is a Laplace random variable with mean 0 and scale λ. Afterwards, node v is checked regarding whether it satisfies the splitting conditions, i.e., its noisy density is higher than split threshold θ and depth of resulting children will not exceed max depth limit h^*. Indeed, if $\hat{d}(v) \geq \theta$ and $h(v) + 1 \leq h^*$, then v is split into four children, the children are inserted to Q, and they are marked as *unvisited*. Otherwise, v is not split and becomes a leaf node. At this point, v has been processed; it is marked as *visited* and the algorithm proceeds to the next iteration (next unvisited node in Q). When all nodes in Q eventually become visited, the algorithm terminates and returns Q.

It can be shown that this algorithm satisfies ε-DP in the centralized setting when $\lambda \geq h^*/\varepsilon$. To verify this, let a new user with an arbitrary location be inserted (removal of an arbitrary user is very similar). This insertion will impact the noisy densities of nodes residing in exactly one root-to-leaf path in Q; since by properties of quadtrees, nodes with the same depth do not have any intersection in their geographic coverages $g(\cdot)$. Then, since the number of nodes residing in one root-to-leaf path in Q is at most h^*, adding Laplace noise calibrated to h^*/ε is sufficient to achieve ε-DP.

Our baseline solution for building quadtrees with LDP adapts the aforementioned algorithm from the centralized DP literature to LDP. Two key modifications are needed in this adaptation. First, the above algorithm visits nodes one-by-one in arbitrary order, as long as they are unvisited. In contrast, our baseline solution visits nodes in breadth-first order, i.e., depth = 1 in the first iteration, depth = 2 in the second iteration, and so forth. Notice that this has no adverse impact on utility or privacy (the aforementioned algorithm can be trivially modified to act in breadth-first order), but it offers us an important convenience in LDP: It enables us to estimate node densities with LDP iteratively such that in the first iteration all nodes with depth = 1 are estimated, in the second

Algorithm 1: Baseline Quadtree Solution

Input : $\mathcal{U}$, Ω, θ, h^*, ε
Output : Quadtree Q

1 $\hat{\varepsilon} \leftarrow \varepsilon/(h^* - 1)$
2 Initalize quadtree Q with root node v_1
3 Set $g(v_1) = \Omega$
4 $i \leftarrow 1$ `// current depth`
5 **while** $i \leq h^*$ **do**
    `// find densities at current depth`
6 $nodes \leftarrow$ list of nodes in Q with depth = i
7 **if** $|nodes| == 1$ **then**
8 Set $d(nodes[1]) = |\mathcal{U}|$ `// i = 1, root node only`
9 **else**
10 $estimates \leftarrow$ GET_ESTIMATES($nodes, \mathcal{U}, \hat{\varepsilon}$)
11 **for** $j = 1$ *to* $|nodes|$ **do**
12 Set $d(nodes[j]) = estimates[j]$
    `// for each node at current depth, determine if it should split or not`
13 **for** $j = 1$ *to* $|nodes|$ **do**
14 **if** $d(nodes[j]) \geq \theta$ *and* $i + 1 \leq h^*$ **then**
15 Split $nodes[j]$ into its four children and add the children to Q
16 **else**
17 Do not split $nodes[j]$
18 $i \leftarrow i + 1$
19 **return** Q

iteration all nodes with depth = 2 are estimated, and so forth. The second modification we perform is that instead of adding Laplace noise, we execute an LDP protocol (OUE) to perform node density estimation. While Laplace noise addition is a de facto mechanism to achieve centralized DP, it is not directly applicable to LDP, therefore the usage of an LDP protocol becomes necessary.

3.2 Baseline Solution for Quadtrees

Our baseline solution for building a quadtree under LDP is shown in Algorithm 1. Given the total privacy budget ε, on line 1, the algorithm computes $\hat{\varepsilon} = \varepsilon/(h^* - 1)$. Here, considering that the algorithm will traverse the quadtree depth-by-depth and h^* is the max depth parameter, $\hat{\varepsilon}$ is the privacy budget that the algorithm will spend at each depth. Since the density of the root node is always equal to $|\mathcal{U}|$ and the server can trivially know the size of the user population, no privacy budget is spent at depth = 1 (only the root node exists at depth = 1). Thus, dividing ε into $h^* - 1$ pieces is sufficient. On lines 2 and 3, the algorithm initializes the root node. Then, the main loop of the algorithm (lines 5–18) iterates depth-by-depth, and at each iteration, it estimates the densities of nodes at the current depth. The case where depth = 1 (only the root node exists) is handled between lines 7 and 8. At every other depth value, nodes' densities at that depth are computed with the help of the GET_ESTIMATES function (lines 10–12), which is explained in the next paragraph. GET_ESTIMATES satisfies $\hat{\varepsilon}$-LDP, and note that Algorithm 1 invokes it at most $h^* - 1$ times; thus, the overall algorithm satisfies ε-LDP by sequential composition. Finally, lines 13–17 are devoted to checking which nodes should split and which ones should not split. If and only if a node satisfies

Algorithm 2: GET_ESTIMATES Function

Input : $\mathcal{N}, \mathcal{U}, \hat{\varepsilon}$
Output: *estimates*

// Server-side
1 Server sends $\mathcal{N}$ to each user in $\mathcal{U}$
// User-side
2 **foreach** $u_i \in \mathcal{U}$ **do**
3 u_i constructs his/her bitvector B_i with length $|\mathcal{N}|$ such that:

$$\forall j \in [1, |\mathcal{N}|] : \; B_i[j] = \begin{cases} 1 & \text{if } l_i \in g(\mathcal{N}[j]) \\ 0 & \text{otherwise} \end{cases}$$

4 u_i perturbs B_i to satisfy $\hat{\varepsilon}$-LDP according to Equation 2
5 u_i sends resulting perturbed bitvector B'_i to server
// Server-side
6 *estimates* $\leftarrow$ [ ] // initialize empty
7 **for** $j = 1$ *to* $|\mathcal{N}|$ **do**
8 Server computes reported count of position j as: $\widehat{C}(j) = \sum_{i=1}^{|\mathcal{U}|} B'_i[j]$
9 Server computes estimate $\bar{C}(j)$ as: $\bar{C}(j) = \frac{2 \cdot \left((e^{\hat{\varepsilon}}+1) \cdot \widehat{C}(j) - |\mathcal{U}|\right)}{e^{\hat{\varepsilon}}-1}$
10 Server appends $\bar{C}(j)$ to *estimates*
11 **return** *estimates*

the splitting conditions, i.e., its noisy density is $\geq \theta$ and the height of its children will not exceed h^*, then the node will split.

The GET_ESTIMATES function, which is used by our baseline solution as well as our proposed solution, is described in Algorithm 2. Its inputs are the list of nodes $\mathcal{N}$, list of users $\mathcal{U}$, and privacy budget $\hat{\varepsilon}$. It returns a list called *estimates*, such that *estimates*$[j]$ is the estimated noisy density of node $\mathcal{N}[j]$. The execution of the GET_ESTIMATES function can be broken down into three steps. First, the server sends $\mathcal{N}$ to each user. Second, on the user side, each user u_i constructs a bitvector B_i, where $|B_i| = |\mathcal{N}|$. For each position j, $B_i[j]$ is determined according to whether u_i's real location l_i falls within the geographic boundaries of node $\mathcal{N}[j]$. That is, if l_i falls within $g(\mathcal{N}[j])$ then the j'th position of B_i is set to 1, otherwise it is 0. After constructing the B_i, it must be perturbed probabilistically in order to satisfy $\hat{\varepsilon}$-LDP. To do so, the user-side perturbation process of the OUE protocol is executed, as shown in Equation (2). The output of this process is the perturbed bitvector B'_i, which is sent to the server. Finally, the third step begins after the server receives perturbed bitvectors from all users. The server initializes *estimates* as an empty list. Afterwards, for j between 1 and $|\mathcal{N}|$, the server first computes $\widehat{C}(j)$ and then uses $\widehat{C}(j)$ to compute $\bar{C}(j)$ according to the equations on lines 8 and 9. $\bar{C}(j)$ is appended to *estimates* in each iteration, and eventually, *estimates* is returned by Algorithm 2.

Theorem 1. *GET_ESTIMATES satisfies $\hat{\varepsilon}$-LDP as long as for all nodes $v_a, v_b \in \mathcal{N}$, it holds that: $g(v_a) \cap g(v_b) = \emptyset$.*

Proof. If $g(v_a) \cap g(v_b) = \emptyset$ for all $v_a, v_b \in \mathcal{N}$, then by line 3 of Algorithm 2, the bitvector B_i will contain either one 1 bit since there exists a node in $\mathcal{N}$ such that $l_i \in g(\mathcal{N}[j])$, or zero 1 bits since there exists no such node. We need to show that, across all possible cases, the perturbation of OUE

(line 4 of Algorithm 2) satisfies:

$$\forall B_i^1, B_i^2, B' : \frac{\Pr[\Psi(B_i^1) = B']}{\Pr[\Psi(B_i^2) = B']} \leq e^{\hat{\varepsilon}}. \tag{8}$$

There are four possible cases; we show that Equation (8) is satisfied in all of them.

Case 1: B_i^1 contains zero 1 bits and B_i^2 contains zero 1 bits. In this case, based on Equation (2), it is clear that $\Pr[\Psi(B_i^1) = B'] = \Pr[\Psi(B_i^2) = B']$. Hence, Equation (8) is satisfied for any $\hat{\varepsilon} \geq 0$.

Case 2: B_i^1 contains one 1 bit and B_i^2 contains one 1 bit. In this case, the proof that Equation (8) is satisfied immediately follows from the fact that OUE satisfies LDP, and its proof is identical to the OUE privacy proof in [40].

Case 3: B_i^1 contains zero 1 bits and B_i^2 contains one 1 bit. Without loss of generality, say that the 1 bit occurs at position $B_i^2[j]$. Then:

$$\frac{\Pr[\Psi(B_i^1) = B']}{\Pr[\Psi(B_i^2) = B']} = \frac{\Pr[\Psi(B_i^1[j]) = B'[j]]}{\Pr[\Psi(B_i^2[j]) = B'[j]]}. \tag{9}$$

If $B'[j] = 0$, then using the perturbation probabilities from Equation (2), (9) becomes:

$$\frac{\Pr[\Psi(0) = 0]}{\Pr[\Psi(1) = 0]} = \frac{2e^{\hat{\varepsilon}}}{e^{\hat{\varepsilon}} + 1}. \tag{10}$$

Since $\hat{\varepsilon} \geq 0$, we have: $e^{\hat{\varepsilon}} + 1 \geq 2$. Thus, $\frac{2e^{\hat{\varepsilon}}}{e^{\hat{\varepsilon}}+1} \leq e^{\hat{\varepsilon}}$ holds.

On the other hand, if $B'[j] = 1$, then using the perturbation probabilities from Equation (2), (9) becomes:

$$\frac{\Pr[\Psi(0) = 1]}{\Pr[\Psi(1) = 1]} = \frac{2}{e^{\hat{\varepsilon}} + 1}. \tag{11}$$

Again, since $\hat{\varepsilon} \geq 0$, we have: $e^{\hat{\varepsilon}} + 1 \geq 2$. Thus, $\frac{2}{e^{\hat{\varepsilon}}+1} \leq 1 \leq e^{\hat{\varepsilon}}$ holds.

Case 4: B_i^1 contains one 1 bit and B_i^2 contains zero 1 bits. Without loss of generality, say that the 1 bit occurs at position $B_i^1[j]$. Then, similar to Case 3:

$$\frac{\Pr[\Psi(B_i^1) = B']}{\Pr[\Psi(B_i^2) = B']} = \frac{\Pr[\Psi(B_i^1[j]) = B'[j]]}{\Pr[\Psi(B_i^2[j]) = B'[j]]}. \tag{12}$$

If $B'[j] = 0$, then using the perturbation probabilities from Equation (2), (12) becomes:

$$\frac{\Pr[\Psi(1) = 0]}{\Pr[\Psi(0) = 0]} = \frac{e^{\hat{\varepsilon}} + 1}{2e^{\hat{\varepsilon}}}. \tag{13}$$

We want to verify if $\frac{e^{\hat{\varepsilon}}+1}{2e^{\hat{\varepsilon}}} \leq e^{\hat{\varepsilon}}$. This holds if and only if $2e^{2\hat{\varepsilon}} - e^{\hat{\varepsilon}} - 1 \geq 0$ for all $\hat{\varepsilon} \geq 0$. The function $2e^{2\hat{\varepsilon}} - e^{\hat{\varepsilon}} - 1$ is indeed ≥ 0 for all $\hat{\varepsilon} \geq 0$. Thus, $\hat{\varepsilon}$-LDP holds.

On the other hand, if $B'[j] = 1$, then using the perturbation probabilities from Equation (2), (12) becomes:

$$\frac{\Pr[\Psi(1) = 1]}{\Pr[\Psi(0) = 1]} = \frac{e^{\hat{\varepsilon}} + 1}{2}. \tag{14}$$

We want to verify if $\frac{e^{\hat{\varepsilon}}+1}{2} \leq e^{\hat{\varepsilon}}$. This is equivalent to verifying if $e^{\hat{\varepsilon}} + 1 \leq 2e^{\hat{\varepsilon}}$. Since $e^{\hat{\varepsilon}} \geq 1$ for all $\hat{\varepsilon} \geq 0$, it indeed holds that $e^{\hat{\varepsilon}} + 1 \leq 2e^{\hat{\varepsilon}}$. Thus, $\hat{\varepsilon}$-LDP holds. □

Theorem 2. *The baseline quadtree solution (Algorithm 1) satisfies ε-LDP.*

Proof. Algorithm 1 invokes GET_ESTIMATES at most $h^* - 1$ times, each time with budget $\hat{\varepsilon} = \varepsilon/(h^* - 1)$. Thus, by sequential composition, if each invocation of GET_ESTIMATES satisfies $\hat{\varepsilon}$-LDP, then Algorithm 1 satisfies ε-LDP. According to Theorem 1, GET_ESTIMATES satisfies $\hat{\varepsilon}$-LDP when $g(v_a) \cap g(v_b) = \emptyset$ for all $v_a, v_b \in \mathcal{N}$. In Algorithm 1, $\mathcal{N} = nodes$ is constructed on line 6 as the list of nodes in quadtree Q with the same depth. By definition of quadtrees (Section 2.3), all nodes at the same depth cover mutually exclusive geographical spaces, and it holds that no two nodes have a non-zero intersection. Hence, Algorithm 1 satisfies ε-LDP. □

3.3 Proposed Solution for Quadtrees

The main weakness of the baseline solution is that it divides the total ε privacy budget into $h^* - 1$ pieces, each to be used in one depth of estimation. Since the privacy budget used per estimation is smaller, higher noise gets added to satisfy privacy. This causes resulting node density estimations to contain large amounts of noise. Excessively noisy densities are also used in the decision to split or not split a node, further causing structural inaccuracies in the resulting quadtree, since a node that should not be split according to its real density ends up being split due its noisy density (or vice versa). Consequently, although ε-LDP is achieved, the quadtrees built using our baseline solution can have low similarity in terms of structure and node densities compared to a noise-free quadtree.

In order to address this problem, the key insight of our proposed solution is that instead of dividing ε into $h^* - 1$ pieces, it uses the whole ε in a single step. This single-step estimation is performed at the leaves of the quadtree (depth = h^*) and contains low noise, therefore the leaves contain high-quality density estimates. Then, leaves' densities are propagated upwards (in bottom-up fashion) towards the root, to populate the densities of remaining nodes. Finally, in top-down fashion, corrections and refinements are made in the structure of the quadtree.

The pseudocode of our proposed solution is shown in Algorithm 3. Its inputs and outputs are same as the baseline solution. It can be observed from Algorithm 3 that the proposed solution consists of four main steps, which are explained below.

Step 1: Construct Initial, Balanced Quadtree with Height = h^*. We construct an initial quadtree Q that is fully grown until height h^*. That is, we let each node in the quadtree split into its four children until h^* is reached. The resulting Q is fully balanced. Note that the θ threshold is not taken into account in constructing this initial Q.

Step 2: Density Estimation for Leaf Nodes with LDP. In the second step, we retrieve all leaf nodes in Q and obtain density estimates for each of them using the full privacy budget ε. This is achieved by a single invocation of the GET_ESTIMATES function with all leaves and full privacy budget ε. After this step is complete, for each leaf node $v \in Q$, its density $d(v)$ is determined.

Step 3: Bottom-Up Propagation of Densities. While densities of the leaves have been determined in step 2, densities of all remaining nodes (non-leaf nodes) are unknown. In this step, densities of non-leaf nodes are determined in bottom-up fashion. First, all nodes with depth $h^* - 1$ are handled, then, all nodes with depth $h^* - 2$ are handled, ... until the root node. For each non-leaf node, recall from Equation (6) that its density is equal to the sum of the densities of its four children.

Step 4: Top-Down Correction. The initial quadtree Q constructed in step 1 is fully balanced rather than taking into account the density threshold θ. As a result, it is possible that a node v which should not have children (because it fails the $d(v) \geq \theta$ condition) actually has children in Q. The goal of step 4 is to iterate through Q in top-down fashion and fix such situations. To do so, the algorithm starts at depth = 1 and moves down iteratively (depth = 2, depth = 3, ..., depth = $h^* - 1$). For each node at the current depth, if node v does not satisfy the $d(v) \geq \theta$ condition, its children along with the subtrees rooted at those children (i.e., if the child has any descendants) are removed from Q.

Theorem 3. *The proposed quadtree solution (Algorithm 3) satisfies ε-LDP.*

Algorithm 3: Proposed Quadtree Solution

Input : $\mathcal{U}$, Ω, θ, h^*, ε
Output: Quadtree Q

```
// Step 1: Construct initial tree
```
1 Initalize quadtree Q with root node v_1
2 Set $g(v_1) = \Omega$
3 **for** $i = 1$ *to* $h^* - 1$ **do**
4 | **foreach** *node* v *in* Q *with* $h(v) = i$ **do**
5 | | Split v into its four children and add the children to Q
```
// Step 2: Density estimation for leaf nodes with LDP
```
6 $leaves \leftarrow$ list of nodes in Q with depth = h^*
7 $estimates \leftarrow$ GET_ESTIMATES($leaves, \mathcal{U}, \varepsilon$)
8 **for** $j = 1$ *to* $|leaves|$ **do**
9 | Set $d(leaves[j]) = estimates[j]$
```
// Step 3: Bottom-up propagation of densities
```
10 **for** $i = h^* - 1$ *to* 1 **do**
11 | **foreach** *node* v *in* Q *with* $h(v) = i$ **do**
12 | | Initialize $d(v) \leftarrow 0$
13 | | **for** $v_{child} \in c(v)$ **do**
14 | | | $d(v) \leftarrow d(v) + d(v_{child})$
```
// Step 4: Top-down correction
```
15 **for** $i = 1$ *to* $h^* - 1$ **do**
16 | **foreach** *node* v *in* Q *with* $h(v) = i$ **do**
17 | | **if** $d(v) < \theta$ **then**
18 | | | Remove all children $c(v)$ and all subtrees rooted at those children from Q
19 **return** Q

Proof. Algorithm 3 invokes GET_ESTIMATES once with the full ε budget and the list *leaves* (line 7). According to Theorem 1, this invocation of GET_ESTIMATES satisfies ε-LDP if $g(v_a) \cap g(v_b) = \emptyset$ for all $v_a, v_b \in leaves$. The construction of *leaves* on line 6 ensures that all nodes in *leaves* are at the same depth. By definition of quadtrees (Section 2.3), all nodes at the same depth cover mutually exclusive geographical spaces. Hence, Algorithm 3 satisfies ε-LDP. □

4 Building Kd-trees under LDP

In this section, we describe our solution for building kd-trees for spatial data under LDP. Both quadtrees and kd-trees are hierarchical decompositions which recursively divide the geographical space. Hence, some of our algorithms from the previous section (such as the GET_ESTIMATES function) can be re-used as building blocks for building kd-trees under LDP. On the other hand, there are several conditions in which kd-trees can be preferable to quadtrees, which motivate us to study kd-trees. First, quadtrees are primarily designed for 2D spatial data, whereas kd-trees can handle data in k dimensions, where k can be larger than 2. Thus, kd-trees can be useful in higher-dimensional spatial data. Second, in quadtrees the geographical space is divided into four equi-sized quadrants regardless of the underlying data distribution; whereas in kd-trees, the space is divided into two regions according to the median of the data. This makes kd-trees more data-dependent and adaptive to the underlying data distribution. Third, because of their data-dependent

Algorithm 4: Proposed Kd-tree Solution

Input : $\mathcal{U}$, Ω, θ, h^*, ε, α
Output : Kd-tree K

1 $\hat{\varepsilon} \leftarrow \varepsilon/(h^* - 1)$
2 Initialize kd-tree K with root node v_1
3 Set $g(v_1) = \Omega$ and $d(v_1) = |\mathcal{U}|$
4 $i \leftarrow 1$ `// current depth`
5 $axis \leftarrow$ "x" `// alternate between x and y`
6 **while** $i \leq h^* - 1$ **do**
7 $nodes \leftarrow$ list of nodes in K with depth = i
8 $medians,\ c_1_dens,\ c_2_dens \leftarrow$ FIND_MEDIANS$(nodes, \mathcal{U}, \hat{\varepsilon}, \alpha, axis)$
`// for each node at depth i, determine if it should split`
9 **for** $j = 1$ *to* $|nodes|$ **do**
10 $v \leftarrow nodes[j]$
11 $median \leftarrow medians[j]$
12 **if** $d(v) \geq \theta$ *and* $i + 1 \leq h^*$ **then**
13 Split v into two children v_{c_1} and v_{c_2} according to *median*
14 Add v_{c_1} and v_{c_2} to K
15 Set densities of v_{c_1} and v_{c_2} `// Sec 4.3`
16 **else**
17 Do not split v
18 $i \leftarrow i + 1$
19 **if** $axis =$ "x" **then**
20 $axis \leftarrow$ "y"
21 **else**
22 $axis \leftarrow$ "x"
23 **return** K

nature, kd-trees are more likely to be balanced compared to quadtrees when the underlying data distribution is highly skewed or non-uniform.

In terms of methodology, as opposed to quadtrees, we cannot perform a single step of estimation and then propagate the densities upwards in kd-trees because the kd-tree requires finding the medians of each node, which are node-dependent and data-dependent. As a result, our solution learns medians from $\mathcal{U}$ in every depth of the kd-tree. We describe our general solution approach in Section 4.1 and then describe its details in Sections 4.2 and 4.3.

4.1 Overview of Our Kd-tree Solution

Our proposed solution for building a kd-tree under LDP is shown in Algorithm 4. On line 1, the algorithm divides the total ε budget into $h^* - 1$ pieces, such that each piece will be used in learning medians of nodes at depths $1, 2, \ldots, h^* - 1$. The kd-tree is initialized with root node v_1, and by definition of kd-trees, $g(v_1) = \Omega$ and $d(v_1) = |\mathcal{U}|$. Then, between lines 6–22, the algorithm iterates depth-by-depth, and at each iteration, it estimates the medians and performs necessary node splits at the current depth. Initially, the depth is set as $i \leftarrow 1$ (line 4). Considering that the kd-tree alternates between splitting the x-axis and y-axis, the initial axis for median computation and node

splitting is selected as x-axis (on line 5). Note that it is also possible to select the y-axis as the initial axis by trivially modifying line 5.

The main loop of the algorithm (lines 6–22) iterates depth-by-depth. First, it retrieves the *nodes* at the current depth (line 7). Then, for all *nodes*, it uses the FIND_MEDIANS function to retrieve the medians and densities of potential children (line 8). More details regarding the FIND_MEDIANS function are given in Section 4.2. Afterward, between lines 9–17, each node $v \in nodes$ is checked regarding whether it satisfies the splitting conditions, i.e., density $\geq \theta$ and depth $\leq h^*$ (line 12). If so, v is split into its two children v_{c_1}, v_{c_2} according to its *median*, and the children are added to K. For determining the densities of the children, it is possible to make use of the children densities (c_1_dens and c_2_dens) returned by FIND_MEDIANS. This is done on line 15; we describe two alternative strategies for this part in Section 4.3. Finally, between lines 18–22, the current depth i is advanced by 1, and the axis is changed from x to y or y to x in order to alternate between them. On the last line, the algorithm returns the final kd-tree K.

4.2 Finding Medians

The FIND_MEDIANS function is a key component of our kd-tree solution. This function takes as input a list of kd-tree *nodes*, user population $\mathcal{U}$, allocated privacy budget $\hat{\varepsilon}$, bucket count parameter α, and the *axis* that will be split. It returns three lists with sizes equal to the size of *nodes*: $medians[j]$ stores the estimated (using LDP) median of $nodes[j]$, $c_1_dens[j]$ stores the first child's estimated density, and $c_2_dens[j]$ stores the second child's estimated density. Its pseudocode is given in Algorithm 5. We explain the algorithm in three main steps.

Step 1: Server-Side Computation of Buckets. For finding the median of a node v, we make use of *buckets*. Buckets divide the geographic space $g(v)$ into small pieces with respect to the *axis*. The number of buckets is determined by the α parameter. Each bucket has an *owner*, i.e., if the bucket results from dividing v into small pieces, then the owner of this bucket is v. An example of dividing v into $\alpha = 4$ buckets with respect to the x-axis is shown in Figure 3. We can observe that $g(v)$ is split into four equal-sized buckets such that the horizontal axis (x-axis) is divided into 4, and the bounds of the y-axis remain the same across all buckets. This exemplifies the division of one v into four buckets; between lines 2–18 of Algorithm 5, this process is performed for all nodes v in *nodes*. All buckets are collectively stored in a list named *buckets*. Variables *upperlim*, *lowerlim*, *size* used in Algorithm 5 are also exemplified in Figure 3.

Step 2: Get LDP Estimates for Buckets from Users. Now consider that all buckets have been created and stored in the list named *buckets*. To find medians, we need to estimate the densities of all buckets according to users' locations. We make use of the GET_ESTIMATES function previously established in Algorithm 2 for this purpose. This function spends $\hat{\varepsilon}$ and returns the density of each bucket in *buckets*.

Step 3: Construct Medians, c_1_dens, and c_2_dens. Now that we have *buckets* and their densities, the goal of this step is to construct the three return values of the FIND_MEDIANS function: *medians*, c_1_dens, and c_2_dens. When computing the median and densities of the two children of node v, Algorithm 5 retrieves the relevant buckets (*relbuckets*) for this computation, i.e., those buckets for which the owner of the bucket is v. Then, Algorithm 5 calls the CALC_MED function, which is formally described in Algorithm 6.

We explain Algorithm 6 using the illustration in Figure 4, which is a continuation of the example in Figure 3. At the start of Algorithm 6, the relevant four buckets have been identified and their densities have been estimated (written inside the buckets in Figure 4). The goal is to find the median of $g(v)$ and the densities of the two resulting children if v was split according to this median. Algorithm 6 first computes the sum of the estimated densities (denoted by *dsum*), which is $dsum = 87$ in Figure 4. Then, the algorithm searches which bucket the median falls into, starting

Algorithm 5: FIND_MEDIANS Function

Input : $nodes, \mathcal{U}, \hat{\varepsilon}, \alpha, axis$
Output: $medians, c_1_dens, c_2_dens$

// Step 1: Server-side computation of buckets
$buckets \leftarrow$ Initialize empty list
foreach v *in* $nodes$ **do**
  if $axis =$ "x" **then**
    // Splitting by x-axis
    $upperlim \leftarrow max(g_x(v))$
    $lowerlim \leftarrow min(g_x(v))$
  else
    // Splitting by y-axis
    $upperlim \leftarrow max(g_y(v))$
    $lowerlim \leftarrow min(g_y(v))$
  $size \leftarrow (upperlim - lowerlim)/\alpha$
  $curr \leftarrow lowerlim$
  while $curr + size \leq upperlim$ **do**
    if $axis =$ "x" **then**
      Create *bucket* with $x_{min} = curr$, $x_{max} = curr + size$, $y_{min} = min(g_y(v))$ and $y_{max} = max(g_y(v))$
    else
      Create *bucket* with $x_{min} = min(g_x(v))$, $x_{max} = max(g_x(v))$, $y_{min} = curr$ and $y_{max} = curr + size$
    Set owner of *bucket* as v
    Append *bucket* to *buckets*
    $curr \leftarrow curr + size$
// Step 2: Get LDP estimates for buckets from users
$estimates \leftarrow$ GET_ESTIMATES$(buckets, \mathcal{U}, \hat{\varepsilon})$
for $j = 1$ *to* $|buckets|$ **do**
  Set density of $buckets[j]$ as $estimates[j]$
// Step 3: Construct medians, c_1_dens, and c_2_dens
Initialize $medians$, c_1_dens, c_2_dens as empty lists
foreach v *in* $nodes$ **do**
  $relbuckets \leftarrow$ Retrieve those buckets for which owner $= v$
  $med, c1, c2 \leftarrow$ CALC_MED$(relbuckets, axis)$
  Append med to $medians$
  Append $c1$ to c_1_dens
  Append $c2$ to c_2_dens
return $medians$, c_1_dens, c_2_dens

with the leftmost bucket. It keeps track of the sum of densities observed in the variable *currsum*, e.g., after visiting bucket 1 currsum is 12; after visiting bucket 2 currsum is 31; after visiting bucket 3 currsum is 47, and so on. When currsum becomes equal to or higher than $dsum/2$, then that means the most recently visited bucket contains the median (line 11). In that case, the median is retrieved, e.g., as the middle point of bucket 3 in Figure 4. Then, the density of the bucket containing the

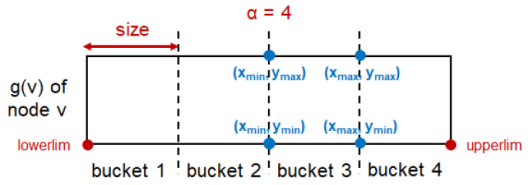


Fig. 3. Sample division of $g(v)$ into $\alpha = 4$ equi-sized buckets. Bounds x_{min}, x_{max}, y_{min}, and y_{max} shown in blue are for bucket 3. Since all buckets are derived from node v, the owner of all buckets is v.

Algorithm 6: CALC_MED Function

Input : *relbuckets*, *axis*
Output: *med*, *c*1, *c*2

1 $dsum \leftarrow$ Sum of densities of all buckets in *relbuckets*
2 **if** *axis* = "*x*" **then**
3 | Sort *relbuckets* in increasing order of their x_{min}
4 **else**
5 | Sort *relbuckets* in increasing order of their y_{min}
6 $currsum \leftarrow 0$
7 **for** $j = 1$ *to* $|relbuckets|$ **do**
8 | $bucket \leftarrow relbuckets[j]$
9 | $den \leftarrow$ density of *bucket*
10 | $currsum \leftarrow currsum + den$
11 | **if** $currsum \geq dsum/2$ **then**
| | `// This is the bucket containing median`
12 | | **if** *axis* = "*x*" **then**
13 | | | Retrieve x_{max} and x_{min} of *bucket*
14 | | | $med \leftarrow (x_{max} + x_{min})/2$
15 | | **else**
16 | | | Retrieve y_{max} and y_{min} of *bucket*
17 | | | $med \leftarrow (y_{max} + y_{min})/2$
18 | | **break**
19 $c1 \leftarrow currsum - \lfloor den/2 \rfloor$
20 $c2 \leftarrow dsum - c1$
21 **return** *med*, *c*1, *c*2

median is distributed among the first child and second child of v. Thus, in Figure 4 v_{c_1} gets a total density of $12+19+26/2=44$, whereas v_{c_2} gets a total density of $87-44$. These values are stored in the appropriate variables in Algorithm 6 and returned (lines 19–21) to Algorithm 5.

THEOREM 4. *The proposed kd-tree solution (Algorithm 4) satisfies ε-LDP.*

PROOF. Algorithm 4 invokes FIND_MEDIANS $h^* - 1$ times, each time with budget $\hat{\varepsilon} = \varepsilon/(h^* - 1)$. By sequential composition, if each invocation of FIND_MEDIANS satisfies $\hat{\varepsilon}$-LDP, then the proposed kd-tree solution satisfies ε-LDP. The only step of FIND_MEDIANS (Algorithm 5) which collects data from users is line 19, therefore it is sufficient to show that this line satisfies $\hat{\varepsilon}$-LDP. We observe that this line is a function call to GET_ESTIMATES with *buckets* and $\hat{\varepsilon}$. According to Theorem 1,

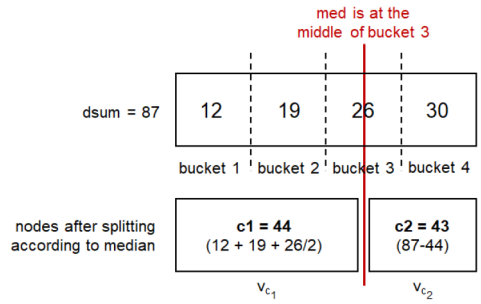


Fig. 4. Example for computing the median of v and estimated densities of two children v_{c_1} and v_{c_2} (Algorithm 6).

GET_ESTIMATES satisfies $\hat{\varepsilon}$-LDP when $g(v_a) \cap g(v_b) = \emptyset$ for all $v_a, v_b \in buckets$. The construction of $buckets$ in Step 1 of FIND_MEDIANS (in particular, lines 11–18) guarantees that $g(v_a) \cap g(v_b) = \emptyset$ for all $v_a, v_b \in buckets$. Hence, the proposed kd-tree solution satisfies ε-LDP. □

4.3 Determining Child Densities

Another key component of our kd-tree solution (Algorithm 4) is determining the densities of children v_{c_1} and v_{c_2}. The densities of the children can be determined directly using the results of LDP estimation, as described in the previous section (Figure 4). That is, line 15 of Algorithm 4 is replaced by:

$$\text{Set } d(v_{c_1}) \leftarrow c_1_dens[j]$$
$$\text{Set } d(v_{c_2}) \leftarrow c_2_dens[j].$$

We call this our initial strategy.

On the other hand, we experimentally observed that our initial strategy can yield erroneous densities since LDP estimations are noisy. Therefore, instead of using this strategy, we developed a second strategy in which we utilize the key property of kd-trees: since nodes are split according to their median, the resulting two children will have equal or near-equal densities. Thus, the density of the parent can be directly divided into two. Examples of this property were also shown in Figure 2. In this case, line 15 of Algorithm 4 is replaced by:

$$\text{Set } d(v_{c_1}) \leftarrow \frac{d(v)}{2}$$
$$\text{Set } d(v_{c_2}) \leftarrow \frac{d(v)}{2}.$$

We call this our improved strategy.

There is the following tradeoff between the initial strategy and the improved strategy. If the median computed under LDP is close to the real median (and therefore node v is split from the right location) then the two children will indeed have equal or near-equal densities by definition of kd-trees. However, if the median computed under LDP is far from the real median, then the kd-tree will contain error. In contrast, the accuracy of the initial strategy relies on the correctness of LDP estimations for bucket densities. Consequently, if median computation is less noisy than bucket density estimation, one should choose the improved strategy over the initial strategy. If vice versa is true, then one should choose the initial strategy over the improved strategy.

The noise level in bucket density estimation can be measured by finding the average error across the densities of all buckets in $relbuckets$ (Algorithm 6). The noise level in median computation can be measured using a metric such as the **Median Error (ME)** metric we will use in our experiments

(Equation (15)), which finds the difference between the true median and the LDP median. Although the two noise level measurements are not directly comparable, based on the experiments we conducted, we observed that estimating the median is less noisy under LDP since it is one numeric value, whereas bucket estimations are a vector of numeric values that are more prone to LDP noise. Furthermore, our experiments show that our median computation strategy is quite accurate and leads to low ME. Hence, we empirically observed that our improved strategy usually yields better results compared to the initial strategy, and therefore, we recommend using our improved strategy in general.

5 Experimental Evaluation

5.1 Experimental Setup

We implemented all algorithms in Python. In this section, we experimentally compare them under varying ε, θ, h^*, and α parameters, as well as using multiple evaluation metrics. We use four real-world location datasets in our experiments: Kaggle, Brightkite, Gowalla, and Foursquare. We also perform an experimental comparison against a state-of-the-art work for ASDQ-LDP [38] to demonstrate the practical benefits of our work and its superiority in this task. Each experiment is repeated 10 times and the average results are reported.

Kaggle. The Kaggle dataset contains trips of 442 taxis driving in the city of Porto. The dataset was originally made public for the taxi service prediction competition in ECML-PKDD; we downloaded it from Kaggle [14]. While the dataset contains full taxi trips (multiple location readings per trip), we pre-processed it by keeping only the starting location of each trip and treated the trip starting locations as the current locations of users in the user population. At the end of this processing, we ended up with 1,048,575 users and their latitude, longitude locations.

Brightkite. The Brightkite dataset contains users' location check-ins from a social network service provider called Brightkite [7]. From the full dataset, we extracted check-ins made in the United States, between longitudes −124.26 and −71.87 and latitudes 25.45 and 47.44.

Gowalla. Gowalla was also a location-based social network site where users contributed their data by sharing their locations [7]. Similar to Brightkite, we extracted check-ins made in the United States using the same latitude and longitude boundaries.

Foursquare. The Foursquare dataset contains location check-ins of users in Tokyo, between the time period from 12 April 2012 to 16 February 2013 [46]. We used this dataset without any pre-processing. It contains a total of 573,703 location check-ins. The minimum latitude is 35.51, maximum latitude is 35.87, minimum longitude is 139.47, and maximum longitude is 139.91.

5.2 Utility Metrics

We define our utility metrics using quadtrees (Q and Q'), but for kd-trees one may simply replace Q and Q' by K and K' and the remaining definitions of the metrics would stay the same. Let Q denote the noise-free quadtree that would be built without privacy protection, and let Q' denote the noisy quadtree built under ε-LDP. We use three utility metrics to measure the difference between Q and Q'. Higher the values of these metrics, higher the difference between Q and Q', and therefore higher the amount of utility loss.

AQE. We generate $N = 100$ random queries and compute their answers on Q and Q', denoted by ans_q and ans'_q respectively. Then, AQE measures the average error between ans_q and ans'_q across all queries as follows:

$$AQE = \frac{1}{N} * \sum_{i=1}^{N} \frac{|ans_{q_i} - ans'_{q_i}|}{max\{ans_{q_i}, b\}},$$

where q_i denotes the i'th query and b denotes a sanity bound that mitigates the effect of queries with extremely high selectivities (extremely low answers) [5, 18, 50]. We set the value of b as: $b = 2\% \times |\mathcal{U}|$.

TED. TED measures the structural difference between Q and Q'. Consider that we want to measure TED between two subtrees rooted at nodes v and v', such that $g(v) = g(v')$. $TED(v, v')$ is defined recursively as follows:

$$TED(v,v') = \begin{cases} 0 & \text{if } |c(v)| = 0 \text{ and } |c(v')| = 0 \\ num_desc(v) & \text{if } |c(v)| > 0 \text{ and } |c(v')| = 0 \\ num_desc(v') & \text{if } |c(v)| = 0 \text{ and } |c(v')| > 0 \\ \sum\limits_{\substack{x \in c(v),\, x' \in c(v') \\ s.t.\ g(x)=g(x')}} TED(x,x') & \text{if } |c(v)| > 0 \text{ and } |c(v')| > 0 \end{cases},$$

where $num_desc(v)$ denotes the number of descendent nodes that v has (excluding itself). The rationale is as follows: If both v and v' do not have any children, then they have zero TED (no structural difference). If one of them has children but the other does not, then all descendants must be counted as part of TED. If both v and v' have children, then their TED is computed recursively on pairs (x, x') where x and x' have matching geographical regions, i.e., x' in Q' is the noisy counterpart of x from Q.

Once TED between v and v' is defined as above, it can be used to measure TED between trees Q and Q' as: $TED(Q, Q') = TED(v_{root}, v'_{root})$ where v_{root} is the root node of Q and v'_{root} is the root node of Q'.

NDD. NDD measures the difference between Q and Q' by computing the differences between corresponding nodes' densities:

$$NDD(Q,Q') = \sum_{v \in Q} \varphi(v),$$

where:

$$\varphi(v) = \begin{cases} |d(v) - d(v')| & \text{if } \exists\, v' \in Q' \text{ s.t. } g(v) = g(v') \\ d(v) & \text{otherwise} \end{cases}.$$

In other words, NDD iterates over each node $v \in Q$ and checks if its counterpart exists in Q', i.e., $v' \in Q'$ with $g(v) = g(v')$. If it exists, then the difference between their densities is computed and added to NDD. However, due to structural differences between Q and Q', it is also possible that such a v' does not exist. In that case, $d(v')$ is assumed to be 0 and therefore $d(v)$ is added directly to NDD.

5.3 Results of Quadtree Algorithms

We first evaluate our quadtree algorithms (baseline solution and proposed solution) from Section 3. In the first set of experiments, we keep the h^* and θ parameters constant ($h^* = 4$, $\theta = 10{,}000$) and vary the ε parameter to observe its impact. The results are reported in Figure 5. Four popular (conventional) ε values are used: $\varepsilon = 0.1, 0.5, 1, 2$. Across all datasets and metrics, we observe that: (i) errors decrease as ε is increased, and (ii) the proposed solution yields lower errors than the baseline solution. The prior is an intuitive result because as ε increases, the noise caused by LDP decreases; therefore, it becomes possible to build more accurate quadtrees. With regards to the latter, especially on Brightkite and Gowalla datasets, the proposed solution makes remarkable improvement in terms of the AQE and NDD metrics when ε is low. The difference between the two solutions is relatively lower in terms of the TED metric. A similar observation holds for the Foursquare dataset—while the difference between the baseline solution and the proposed solution is

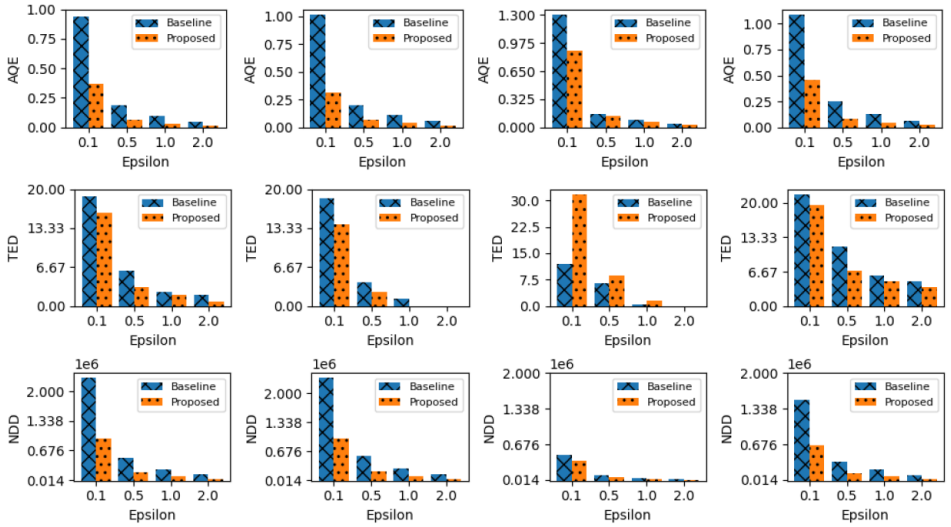


Fig. 5. Results of our baseline versus proposed quadtree solutions with AQE, TED, and NDD metrics (one metric each row) on Brightkite, Gowalla, Kaggle, and Foursquare datasets (left to right).

high in terms of AQE and NDD metrics, it is relatively less pronounced in terms of the TED metric. The only exception in which the proposed solution yields higher error than the baseline solution is the Kaggle dataset and the TED metric. The reason behind this observation is the significant skew in the spatial distribution of the Kaggle dataset. Users' locations in this dataset are heavily accumulated within a small range of latitude and longitude coordinates, and much fewer location readings exist outside this range. Consequently, in the noise-free quadtree, there exist few branches with high depth, whereas the remaining branches are shallow. However, the proposed solution, which constructs a fully balanced quadtree first and then performs top-down structural corrections in its final step, has a higher tendency to end up with a more breadth-balanced quadtree than the noise-free version, thus causing high TED.

Next, we fix $\theta = 10{,}000$ and vary the h^* and ε parameters. The results are shown in Table 1. We again observe that the proposed solution performs better than the baseline solution. For both solutions, it seems that as we increase h^*, errors tend to increase. For the baseline solution, this shows the ineffectiveness of splitting the original ε budget into $h^* - 1$ parts. Another interesting observation is that TED results are good when h^* is low, such as $h^* = 3$. This shows that for the first few levels of the quadtree (which are close to the root node), there is relatively small structural error. For example, when $h^* = 3$, TEDs are usually 0, meaning that our LDP quadtrees have identical structure to noise-free quadtrees. However, as we increase h^* to 4 and 5, there is an increasing amount of structural inequality as demonstrated by increasing TEDs. Overall, this shows that LDP quadtrees are more likely to have structural errors towards their leaf nodes whereas their structure close to the root node remains more accurate.

Finally, in Table 2, we fix $\varepsilon = 1$ and vary the h^* and θ parameters. For low h^* such as $h^* = 3$, different values of θ do not seem to cause substantial changes in AQE on Brightkite, Gowalla, and Foursquare datasets. However, for $h^* = 4$ and $h^* = 5$, changes to θ yield higher differences in terms of AQE. Usually, $\theta = 5{,}000$ or $10{,}000$ seem to be the ideal choice when $h^* = 4$ and $h^* = 5$ for the proposed solution. In contrast, higher θ such as $10{,}000$ or $20{,}000$ seem to be better for the baseline solution. This is because increasing θ decreases the risk of creating erroneous nodes caused by LDP noise. When θ is higher, it is less likely that LDP noise causes node densities to erroneously

Table 1. AQE, TED, and NDD of Quadtrees under Different Heights h^* with Threshold θ = 10,000

			AQE				TED				NDD ($\times 10^4$)			
			$\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$	$\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$	$\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
Brightkite	$h^* = 3$	Baseline	0.317	0.056	0.028	0.014	0.0	0.0	0.0	0.0	60.2	10.4	5.5	2.3
		Proposed	0.156	0.030	0.014	0.006	0.0	0.0	0.0	0.0	30.1	5.2	2.5	1.2
	$h^* = 4$	Baseline	0.941	0.189	0.095	0.046	18.8	6.0	2.4	2.0	229.5	51.7	26.0	13.8
		Proposed	0.369	0.067	0.032	0.015	16.0	3.2	2.0	0.8	94.1	18.5	9.7	4.0
	$h^* = 5$	Baseline	1.924	0.423	0.213	0.092	105.2	66.8	41.2	19.2	499.1	139.9	74.7	38.8
		Proposed	0.637	0.146	0.080	0.036	110.4	55.2	16.0	4.8	258.3	58.6	27.3	11.7
Gowalla	$h^* = 3$	Baseline	0.292	0.053	0.026	0.011	0.0	0.0	0.0	0.0	50.7	10.4	4.8	2.4
		Proposed	0.140	0.022	0.013	0.005	0.0	0.0	0.0	0.0	24.8	4.8	2.5	1.1
	$h^* = 4$	Baseline	1.015	0.201	0.108	0.057	18.4	4.0	1.2	0.0	234.2	56.5	28.4	14.0
		Proposed	0.313	0.069	0.039	0.016	14.0	2.4	0.0	0.0	95.9	20.9	9.5	4.3
	$h^* = 5$	Baseline	2.076	0.436	0.215	0.094	118.4	68.8	40.0	17.6	554.0	151.7	80.5	42.3
		Proposed	0.850	0.172	0.074	0.032	122.4	49.2	17.6	7.6	285.5	62.1	29.6	13.0
Kaggle	$h^* = 3$	Baseline	0.396	0.058	0.025	0.011	4.0	1.2	0.0	0.0	20.5	4.9	2.2	1.1
		Proposed	0.296	0.051	0.020	0.008	5.2	1.6	0.0	0.0	17.8	3.4	1.5	0.6
	$h^* = 4$	Baseline	1.302	0.155	0.085	0.039	12.0	6.4	0.4	0.0	47.0	9.4	5.1	2.5
		Proposed	0.885	0.137	0.065	0.026	31.6	8.8	1.6	0.0	37.2	7.4	3.2	1.4
	$h^* = 5$	Baseline	3.823	0.562	0.145	0.064	52.0	31.2	4.0	0.0	95.7	16.0	8.5	4.2
		Proposed	1.815	0.290	0.133	0.054	74.0	29.6	8.8	0.8	73.0	14.7	7.6	2.8
Foursquare	$h^* = 3$	Baseline	0.373	0.062	0.038	0.019	0.0	0.0	0.0	0.0	38.5	7.3	3.9	1.8
		Proposed	0.154	0.033	0.017	0.008	0.0	0.0	0.0	0.0	19.5	3.5	1.8	0.8
	$h^* = 4$	Baseline	1.085	0.249	0.128	0.060	21.6	11.6	6.0	4.8	149.5	36.0	20.1	9.8
		Proposed	0.462	0.084	0.045	0.024	19.6	6.8	4.8	3.6	65.8	14.4	8.0	3.8
	$h^* = 5$	Baseline	2.334	0.455	0.207	0.098	94.8	58.4	44.0	22.4	275.3	68.7	37.8	19.5
		Proposed	0.883	0.195	0.087	0.046	109.2	56.0	25.6	17.2	146.2	38.4	17.7	8.9

Table 2. AQE of Quadtrees with Privacy Budget $\varepsilon = 1$, Varying h^* and θ

		Baseline				Proposed			
		θ = 1,000	θ = 5,000	θ = 10,000	θ = 20,000	θ = 1,000	θ = 5,000	θ = 10,000	θ = 20,000
Brightkite	$h^* = 3$	0.027	0.029	0.032	0.022	0.015	0.015	0.012	0.012
	$h^* = 4$	0.100	0.099	0.103	0.084	0.036	0.036	0.035	0.039
	$h^* = 5$	0.278	0.229	0.195	0.182	0.078	0.072	0.085	0.082
Gowalla	$h^* = 3$	0.025	0.025	0.026	0.030	0.012	0.013	0.014	0.013
	$h^* = 4$	0.097	0.106	0.100	0.108	0.034	0.028	0.036	0.041
	$h^* = 5$	0.257	0.221	0.214	0.175	0.084	0.067	0.076	0.085
Kaggle	$h^* = 3$	0.046	0.030	0.023	0.024	0.031	0.024	0.021	0.020
	$h^* = 4$	0.170	0.085	0.073	0.075	0.089	0.065	0.073	0.062
	$h^* = 5$	0.396	0.268	0.150	0.161	0.173	0.156	0.177	0.160
Foursquare	$h^* = 3$	0.035	0.035	0.035	0.028	0.016	0.018	0.018	0.017
	$h^* = 4$	0.134	0.130	0.134	0.098	0.046	0.044	0.042	0.043
	$h^* = 5$	0.318	0.294	0.210	0.135	0.096	0.095	0.083	0.088

increase to higher than θ and cause an erroneous split. Similarly, higher θ implies shorter quadtrees for the baseline solution, which eliminates the creation of deeper nodes that have higher risk of being dominated by LDP noise.

5.4 Results of Kd-tree Algorithms

In this section, we evaluate our kd-tree algorithms from Section 4. In the first set of experiments, we keep $\theta = 10{,}000$ constant and vary the ε budget and h^* parameters. The results are shown in Table 3. First, recall that we had two strategies for determining child densities in kd-trees: the initial strategy and the improved strategy. Upon comparing them, we observe from Table 3 that the improved strategy yields lower AQE in the majority of the experiments. This shows that LDP median computation is indeed less noisy compared to LDP bucket density estimation. Therefore, in general, using the improved strategy rather than the initial strategy is preferable. Second, as ε increases, we observe that errors decrease both for the initial and improved strategies. This aspect is similar to quadtrees and it suits our expectations. Third, errors increase as h^* increases from 3 to 5, which is also similar to quadtrees. However, the speed of increase is slower in kd-trees compared to quadtrees. For example, although AQEs in quadtrees are lower than kd-trees when $h^* = 3$, vice versa can be true when $h^* = 4$ or 5. This shows that kd-trees are less affected by the increase in error as h^* increases. Fourth, we analyze the errors across different datasets. Interestingly, kd-trees yield lower error on Kaggle and Foursquare datasets, whereas these two datasets had the highest amount of error in the case of quadtrees. This shows that datasets and data distributions that are not handled well by quadtrees can be handled by kd-trees, and vice versa. Hence, considering that they work well on different datasets, kd-trees and quadtrees can complement each other in real-world tasks.

In the next set of experiments, we fix $\varepsilon = 1$ and $h^* = 4$, and vary the θ parameter. The results are shown in Table 4. We observe that there is no direct relationship between θ and AQE. For example, on the Brightkite dataset, AQE drops as θ is increased from 5,000 to 10,000, then increases as θ is increased from 10,000 to 20,000, and remains similar as θ is increased from 20,000 to 40,000. In contrast, on the Kaggle dataset, AQE drops consistently as θ is increased from 5,000 to 40,000. On the other hand, on the Foursquare dataset, AQE increases consistently as θ is increased from 5,000 to 40,000. Hence, it is not possible to establish a consistent relationship between the value of θ and AQE—the best value of θ can be different under different conditions.

5.5 Accuracy of LDP Median Computation

Recall from Section 4 that a significant component of our kd-tree solution is to find the medians of kd-tree nodes while satisfying LDP. In this section, we measure the accuracy of our proposed LDP median computation method, by comparing the medians found using our method versus the true medians. Let η denote the true median that would be found if all data could be accessed in plaintext (no LDP), and let η' denote the median found using our method satisfying LDP. The ME is defined as:

$$ME = \frac{|\eta' - \eta|}{max - min} * 100\%, \tag{15}$$

where max and min denote the bounds of the axis for which median is being computed (x-axis or y-axis).

The α parameter plays an important role in median estimation since it controls the number of buckets. When there are few buckets, then the density of each bucket is high (better signal-to-LDP-noise ratio) but median computation is coarse since intra-bucket distributions cannot be known. In the extreme case, when $\alpha = 1$, the median is always computed as the middle point of $g_x(v)$ or $g_y(v)$, i.e., no benefit is gained by data collection using LDP. On the other hand, when there are too many buckets, then the density of each bucket is low (LDP noise may overwhelm the true densities) but median computation is fine-grained since each bucket covers a small region. To better

Table 3. AQE of Kd-trees with Threshold θ = 10,000, Varying ε and h^*

			$\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$	$\varepsilon = 2$
Brightkite	$h^* = 3$	Initial	0.378	0.139	0.077	0.073
		Improved	0.236	0.087	0.017	0.012
	$h^* = 4$	Initial	0.685	0.178	0.091	0.058
		Improved	0.383	0.148	0.046	0.060
	$h^* = 5$	Initial	1.158	0.324	0.169	0.104
		Improved	0.534	0.188	0.181	0.079
Gowalla	$h^* = 3$	Initial	0.324	0.093	0.079	0.077
		Improved	0.244	0.051	0.021	0.007
	$h^* = 4$	Initial	0.593	0.200	0.161	0.157
		Improved	0.343	0.139	0.083	0.076
	$h^* = 5$	Initial	0.972	0.288	0.213	0.167
		Improved	0.491	0.193	0.159	0.122
Kaggle	$h^* = 3$	Initial	0.308	0.081	0.046	0.032
		Improved	0.095	0.023	0.015	0.012
	$h^* = 4$	Initial	0.614	0.129	0.081	0.050
		Improved	0.253	0.056	0.032	0.028
	$h^* = 5$	Initial	1.029	0.167	0.118	0.081
		Improved	0.465	0.090	0.061	0.040
Foursquare	$h^* = 3$	Initial	0.391	0.099	0.042	0.021
		Improved	0.166	0.031	0.022	0.011
	$h^* = 4$	Initial	0.637	0.193	0.096	0.051
		Improved	0.377	0.107	0.059	0.032
	$h^* = 5$	Initial	1.419	0.301	0.184	0.092
		Improved	0.645	0.205	0.097	0.065

Table 4. AQE of Kd-trees with the Improved Strategy, ε = 1.0 and $h^* = 4$

	θ = 5,000	θ = 10,000	θ = 20,000	θ = 40,000
Brightkite	0.098	0.046	0.104	0.100
Gowalla	0.083	0.083	0.094	0.078
Kaggle	0.036	0.032	0.029	0.021
Foursquare	0.056	0.059	0.065	0.067

θ is varied between 5,000 and 40,000.

understand which factor dominates and to choose a good α value for practical use, it is valuable to experimentally study the utility impacts of α.

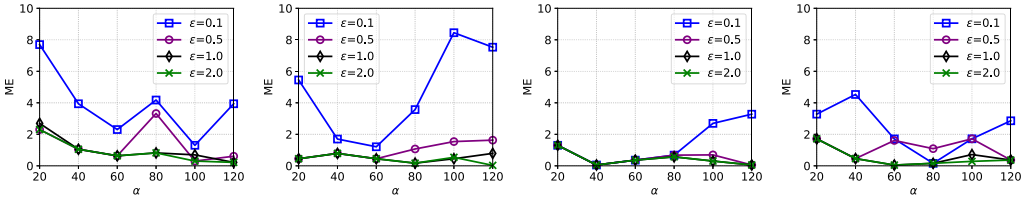


Fig. 6. ME of medians found using our LDP algorithm with varying ε and datasets (from left to right: Brightkite, Gowalla, Kaggle, and Foursquare).

In Figure 6, we plot the MEs across varying α between 20 and 120 for different ε values (0.1, 0.5, 1, 2) and datasets. It can be observed from Figure 6 that MEs are usually quite low. We observe ME $\geq 4\%$ only when $\varepsilon = 0.1$, but for other ε, MEs are generally $\leq 2\%$. Especially when ε is 1 or 2, we observe low MEs across all α values. This shows that our median computation method works well in practice, and enables accurate computation of the medians under LDP.

In the case of high ε values (such as 1 or 2), the impact of α on MEs is not very high. However, when privacy is stricter (such as $\varepsilon = 0.1$), MEs are relatively more impacted by the choice of α. In the low ε regimes, the change in ME across $\alpha \in [20, 120]$ resembles a U-shaped curve. When α is low (such as $\alpha = 20$), ME is high. This is because a low number of buckets causes median computation to be too coarse, and therefore even if our algorithm identifies the correct bucket, the precision of the median computation is negatively affected. On the other hand, when α is high (such as $\alpha = 100$ or 120), we observe that MEs can also be high. This is because few location points fall into each bucket, causing low bucket density. This hurts the correctness of identifying the right bucket for the median, which increases ME. Overall, to minimize error, our results and analyses show that instead of choosing α values that are too low or too high, it is preferable to choose values towards the middle, e.g., $\alpha = 60$ or 80.

5.6 Comparison against Prior Work

In this section, we perform an experimental comparison between our quadtree solution (the "proposed" version), kd-tree solution (using the "improved" strategy), and a state-of-the-art work for ASDQ-LDP [38]. Given a set of spatial density queries, ASDQ-LDP uses a four-step solution to answer the set of queries while satisfying LDP: partitioning, finding sensitivity, user-side noisy response computation, and server-side estimation. For partitioning, we use the Advanced Partitioning strategy (ADV) of ASDQ-LDP, which was experimentally shown to be the best choice in [38]. For noisy response computation and estimation, we use the **Optimized Bitvector Encoding (OBE)** protocol. Together, this corresponds to the OBE-ADV strategy in [38], which is the recommended strategy. To perform the experimental comparison, we generate $N = 600$ random queries and compute their true answers and noisy answers under LDP using each of the three approaches (our quadtree solution, our kd-tree solution, and ASDQ-LDP [38]) separately. In order to ensure a fair comparison, the same set of queries is used in all three approaches.

In Figure 7, we plot the AQEs of each of the three approaches with different ε values (0.1, 0.5, 1, 2) and datasets. We observe that our quadtree and kd-tree solutions achieve substantially lower errors compared to ASDQ-LDP in all cases. The improvements can be up to 10-fold in some settings. We further observe that the errors of ASDQ-LDP are quite high when $\varepsilon = 0.1$ and 0.5, but improve when $\varepsilon = 1$ and 2. Our kd-tree and quadtree solutions also have relatively higher errors under $\varepsilon = 0.1$, but their errors remain low for $\varepsilon \geq 0.5$. These results from Figure 7 agree with the results in Figure 5, since we had also observed in Figure 5 that errors decrease substantially as we move from $\varepsilon = 0.1$ to $\varepsilon \geq 0.5$.

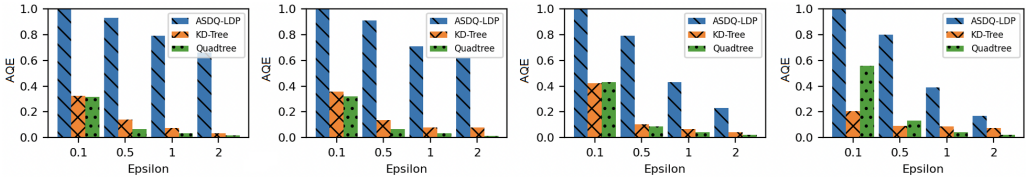


Fig. 7. Errors of quadtrees, kd-trees, and ASDQ-LDP [38] with varying ε and datasets (from left to right: Brightkite, Gowalla, Foursquare, and Kaggle).

Table 5. Execution Times of the Proposed Quadtree Solution and the Kd-tree Solution Using the Improved Strategy

		Quadtree				Kd-tree			
		$\theta = 1{,}000$	$\theta = 5{,}000$	$\theta = 10{,}000$	$\theta = 20{,}000$	$\theta = 1{,}000$	$\theta = 5{,}000$	$\theta = 10{,}000$	$\theta = 20{,}000$
Brightkite	$h^* = 3$	3.98	4.12	4.01	4.04	65.27	66.02	65.09	65.08
	$h^* = 4$	14.44	14.52	14.69	14.77	130.91	131.12	131.48	132.27
	$h^* = 5$	55.93	57.24	57.93	58.06	268.42	270.08	269.18	267.89
Gowalla	$h^* = 3$	4.01	4.22	4.18	4.09	62.84	61.79	62.06	62.20
	$h^* = 4$	14.41	14.38	14.84	14.32	130.12	130.42	130.13	129.99
	$h^* = 5$	55.12	57.43	51.83	55.38	256.33	256.27	257.14	253.82
Kaggle	$h^* = 3$	4.24	4.07	4.02	4.13	59.50	63.98	59.55	60.05
	$h^* = 4$	14.13	14.37	14.20	14.39	118.72	119.43	120.74	120.69
	$h^* = 5$	55.83	55.11	54.30	54.91	243.92	243.19	239.91	244.60
Foursquare	$h^* = 3$	2.21	2.17	2.22	2.17	35.76	35.61	35.59	35.70
	$h^* = 4$	7.83	7.83	7.72	7.89	73.00	71.22	71.53	71.81
	$h^* = 5$	30.69	31.64	32.13	31.32	146.92	147.06	144.22	145.07

Privacy budget is $\varepsilon = 1$, varying h^* and θ. Results are in seconds.

In addition to demonstrating the improvement achieved by our work compared to [38], Figure 7 also enables us to compare quadtrees and kd-trees with one another. When $\varepsilon = 0.1$, errors of quadtrees and kd-trees are usually similar (apart from the Kaggle dataset on which kd-trees achieve much lower error). For remaining ε values, errors of quadtrees are usually slightly lower compared to kd-trees. Quadtrees achieve particularly low errors of $\leq 2.5\%$ when $\varepsilon = 2$.

5.7 Execution Time Analysis

Finally, we experimentally analyze the execution times of our quadtree and kd-tree solutions. Similar to the previous section, we use our "proposed" solution for quadtrees and our "improved" strategy for kd-trees because of their higher utility. In Table 5, we report their execution times with varying datasets, h^*, and θ. The privacy budget is fixed to $\varepsilon = 1$, but we note that the budget does not have a noticeable impact on execution times. In simulation, the whole user population $\mathcal{U}$ is simulated on the same device sequentially (one after the other). However, in the real-world, users would be encoding and perturbing their data with LDP in parallel, on their own devices. Thus, to better represent the execution times of our solutions when applied to the real-world, we measure and report execution times as if user-side encoding and perturbation is done in parallel by all users.

We observe from Table 5 that the primary factor affecting execution times is h^*. In contrast, θ does not seem to have a significant impact. The super-linear increase in execution times with respect to h^* is especially evident in the case of quadtrees. Execution times are also similar for the three datasets with similar number of users, namely Brightkite, Gowalla, and Kaggle. Foursquare is the smallest dataset in terms of the number of users, and therefore many of the server-side operations and computations are also faster on Foursquare than the other datasets. Comparing kd-trees and quadtrees, we observe that quadtrees are usually much faster than kd-trees. The reason behind this observation is that the proposed quadtree solution collects perturbed data from the users in a single step and then performs bottom-up propagation and top-down correction according to the estimation results. Although it is relatively time-consuming to collect perturbed results from the users, execution times of the bottom-up and top-down steps are relatively low. Furthermore, there is no communication between the server and the users in these steps. In contrast, the kd-tree solution is iterative and needs to collect perturbed data from users at each depth. Thus, it necessitates multiple communication and data collection rounds between the server and the users. In each round, there are time costs associated with creating the buckets and estimating the median, caused by the FIND_MEDIANS function. Thus, the kd-tree solution incurs higher execution times than the quadtree solution.

6 Further Discussion and Analysis

6.1 Use of Alternative Privacy Mechanisms and Protocols

Throughout the article, we used the OUE protocol as a building block in our solutions. However, there exist alternative LDP mechanisms and protocols that can be used, such as **Generalized Randomized Response (GRR)**, RAPPOR, **Optimized Local Hashing (OLH)**, a properly calibrated **Exponential Mechanism (EM)**, and so forth [17, 21, 40]. The reason why we chose OUE is because of its higher utility [40]. Although it may be possible to modify GET_ESTIMATES to use another LDP protocol in place of OUE, we would generally expect this to decrease utility rather than increase it. On the other hand, if the goal is to improve the communication efficiency between the users and the server or to decrease user-side computation costs, then a simpler protocol such as GRR or EM, or a hash-based protocol such as OLH can be chosen. In this case, Algorithm 2 would need to be changed, and new privacy proofs would need to be produced. We note that privacy proofs of alternative protocols are non-trivial and not guaranteed, and they must be done case-by-case. Therefore, we leave the integration of alternative protocols into our quadtree and kd-tree solutions to future work.

6.2 When To Use Quadtrees versus Kd-trees

We developed solutions for two types of hierarchical spatial decompositions: quadtrees and kd-trees. One pertinent question is when to prefer quadtrees versus when to prefer kd-trees. Our results show that in many cases, the proposed quadtree solution yields lower errors than kd-trees, therefore in many cases, the proposed quadtree solution is preferable to improve utility. In some cases, quadtrees and kd-trees yield similar errors, e.g., low ε values in Figure 7. Among the four datasets, the only dataset in which kd-trees yield lower AQE than quadtrees is Kaggle. When we study the spatial distributions of the four datasets as shown in Figure 8, we observe that Kaggle is the dataset with the highest non-uniformity. At the beginning of Section 4, we stated that due to their data-dependent nature, kd-trees can be a better fit when the underlying data distribution is highly skewed or non-uniform. The experimental results support this intuition.

Overall, we recommend the usage of the proposed quadtree solution in general, especially when the data distribution is more uniform. In contrast, if it is suspected that the spatial data distribution is



Fig. 8. Spatial density distributions of the four datasets, from left to right: Brightkite, Gowalla, Foursquare, and Kaggle.

Table 6. AQE of Quadtrees with Foursquare Dataset, $\varepsilon = 1$, Baseline versus Proposed Quadtree Solutions

	$\theta = 5{,}000$		$\theta = 10{,}000$		$\theta = 20{,}000$		$\theta = 40{,}000$	
	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed
$h^* = 2$	0.006	0.008	0.008	0.008	0.007	0.009	0.008	0.008
$h^* = 3$	0.039	0.016	0.034	0.016	0.037	0.020	0.037	0.014
$h^* = 4$	0.123	0.043	0.112	0.048	0.098	0.047	0.063	0.039

non-uniform or skewed, then kd-trees, which rely on data-dependent medians, should be preferred. These guidelines are applicable across varying h^* and θ.

6.3 Impact of Data Sparsity and h^*

Among the two quadtree solutions, the proposed solution initially constructs a fully grown, balanced quadtree with height h^* in its first step. In the second step, density estimations of all leaf nodes in this tree are obtained using ε-LDP. Thus, there are a total of 4^{h^*} estimation results. One risk here is that the high number of estimations will cause high variance and error, especially when h^* increases and the data distribution is sparse.

First, we highlight that using the OUE protocol brings benefits which combat this risk. As shown in [40], estimations in OUE are unbiased and their variance does not depend on the domain size, i.e., the variance of each individual leaf's density estimation will be unaffected by the total number of leaves. This mitigates the risk of increased variance in individual estimations when h^* is higher. In contrast, if another protocol such as GRR was used, then h^* would have impacted the variance of each individual estimation.

Nevertheless, the third step of the proposed solution performs bottom-up propagation of density estimations (lines 10–14 of Algorithm 3). Each individual estimation is unbiased, hence the results of this bottom-up propagation will be unbiased. Yet, $d(v)$ of an intermediate node in the tree is determined by the density estimations of its children $\sum_{v_{child} \in c(v)} d(v_{child})$. Since $d(v_{child})$ of children are results of LDP estimations, the variance of the summation will increase. To analyze whether this increase causes problems under varying data sparsity and whether the proposed solution may underperform compared to the baseline solution under sparse data distributions, we performed the following experiment. On the Foursquare dataset, we built quadtrees with varying h^* and θ using our baseline solution and our proposed solution. Sparsity increases as h^* increases and θ decreases. We measure and report AQEs to compare the two solutions in Table 6.

Table 6 shows that errors of both the baseline solution and the proposed solution increase as data sparsity increases, i.e., h^* increases and θ decreases. First, consider h^*. Errors of both solutions are lowest when $h^* = 2$, and their errors are similar to one another. As h^* increases to 3 and 4,

Table 7. ME of Medians Found Using Our LDP Algorithm on Exponential Datasets with Varying β

	$\alpha = 60$	$\alpha = 80$	$\alpha = 100$	$\alpha = 120$
(closer to uniform) $\beta = \frac{1}{3}$	0.47	0.14	0.52	0.47
$\beta = \frac{1}{5}$	0.89	1.28	1.41	1.13
$\beta = \frac{1}{10}$	1.43	1.81	2.73	2.31
(more skewed) $\beta = \frac{1}{20}$	1.47	1.86	2.66	3.28

errors increase. However, we observe that the amount of increase is higher in the baseline solution compared to the proposed solution, therefore the proposed solution becomes more preferable as h^* becomes higher. Next, consider θ. Since $h^* = 2$ yields very shallow quadtrees, changing θ does not have much impact on AQE. However, for $h^* = 4$, we observe that having a sparse data distribution (i.e., low θ) causes high AQE, especially for the baseline solution.

Considering all results, we conclude that both solutions are negatively impacted by increasing data sparsity. However, the negative impact is more severe for the baseline solution, both in terms of h^* and θ parameters. Thus, the proposed solution maintains better performance than the baseline solution despite increasing sparsity.

6.4 Impact of Data Skewness on LDP Median Computation

Next, we analyze whether the skewness of the underlying data impacts the errors of our LDP median computation algorithm. In order to obtain data with varying skewness, we generate synthetic datasets using an Exponential distribution. The probability density function of the Exponential distribution is:

$$f(x;\beta) = \begin{cases} \frac{1}{\beta} * e^{-x/\beta} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}, \tag{16}$$

and its expected median is equal to $\ln(2) * \beta$. Here, β is the parameter that controls skewness. When β is small, the distribution is closer to uniform; whereas when β is large, the distribution becomes more skewed. We generate different datasets by sampling data from Exponential distributions with different β in each dataset. We assume a fixed population size $|\mathcal{U}| = 100.000$ for all datasets. Using $\varepsilon = 1$, we execute our LDP median computation algorithm with varying α and calculate the ME of medians found using our algorithm. Each experiment is repeated 20 times and the average MEs are reported in Table 7.

Table 7 shows that across all α values, as the distribution becomes more skewed, MEs increase. Increasing data skewness causes intra-bucket distributions to also become more skewed. Recall from Algorithm 5 that Calc_Med determines the median as the middle point of the corresponding bucket. This yields a more accurate result when the intra-bucket distribution is closer to uniform since the median of a uniform distribution is indeed at the middle point of the bucket's range. In contrast, in the case of skewed intra-bucket distribution, the true median is farther away from the middle of the bucket, which yields higher ME since our algorithm picks the middle. Nevertheless, although increasing skewness causes higher ME, MEs remain relatively low (below 4% error) even for very skewed datasets (e.g., $\beta = \frac{1}{20}$), showing that our algorithm remains effective across varying data skewness.

6.5 Time Complexity Analysis

In this section, we perform theoretical analyses regarding the time complexity of our quadtree and kd-tree solutions using the big-O notation.

Baseline Quadtree Solution. The main while loop in Algorithm 1 performs $O(h^*)$ iterations. In each iteration, the number of nodes is $O(4^{h^*})$. For $O(4^{h^*})$ number of nodes, the function GET_ESTIMATES is executed. Furthermore, there are two passes over all nodes: one on lines 11 and 12 and another one on lines 13–17. Therefore, denoting the complexity of GET_ESTIMATES by $O(\text{GE})$, the overall complexity of the baseline quadtree solution is $O(h^* \cdot \text{GE} + h^* \cdot 4^{h^*})$.

To quantify $O(\text{GE})$, we analyze the complexity of GET_ESTIMATES. We note GET_ESTIMATES will take as input a list of nodes with size $O(4^{h^*})$ and the population size is $O(|\mathcal{U}|)$. In real-world use of our baseline quadtree solution, each user executes the user-side encoding and perturbation in Algorithm 2 independently from others. This part is $O(4^{h^*})$ since it is linear in the size of the input list of nodes. The server-side estimation part between lines 8–10 is $O(4^{h^*} \cdot |\mathcal{U}|)$. On the other hand, when we execute Algorithm 2 in simulation, all users and the server are simulated on a single machine. In this case, the simulation of all users' user-side encoding and perturbation is $O(4^{h^*} \cdot |\mathcal{U}|)$ and the server-side estimation part remains $O(4^{h^*} \cdot |\mathcal{U}|)$. Therefore, overall, we can conclude that Algorithm 2 is $O(4^{h^*} \cdot |\mathcal{U}|)$.

Plugging $O(\text{GE})$ from the previous paragraph into $O(h^* \cdot \text{GE} + h^* \cdot 4^{h^*})$, we find that the baseline quadtree solution has $O(h^* \cdot 4^{h^*} \cdot |\mathcal{U}|)$ time complexity.

Proposed Quadtree Solution. We analyze Algorithm 3 step by step. Step 1 has $O(4^{h^*})$ time complexity due to constructing the nodes of the initial tree. Step 2 executes GET_ESTIMATES with $O(4^{h^*})$ nodes and $O(|\mathcal{U}|)$ population size; in addition, it performs a linear pass over all nodes causing $O(4^{h^*})$ complexity. Adding them both, the time complexity of Step 2 is dominated by GET_ESTIMATES, and it is $O(4^{h^*} \cdot |\mathcal{U}|)$. The time complexity of Step 3 as well as Step 4 is $O(h^* \cdot 4^{h^*})$ due to the nested loops. Adding the complexities of the four steps together, we find that the overall complexity of the proposed quadtree solution is $O(h^* \cdot 4^{h^*} + |\mathcal{U}| \cdot 4^{h^*})$. Typically, the population size is orders of magnitude larger than the h^* parameter, i.e., $|\mathcal{U}| >> h^*$. Assuming this holds, we can conclude that the complexity of the proposed quadtree solution is $O(|\mathcal{U}| \cdot 4^{h^*})$.

We note that in terms of asymptotic time complexity, the proposed quadtree solution has lower complexity than the baseline solution. This is an additional advantage that the proposed solution has over the baseline solution.

Proposed Kd-tree Solution. The main while loop in Algorithm 4 performs $O(h^*)$ iterations. In each iteration, the number of nodes is $O(2^{h^*})$. Each iteration executes FIND_MEDIANS with $O(2^{h^*})$ nodes, $O(|\mathcal{U}|)$ population size, and α parameter. In addition, it performs a linear pass over all nodes causing $O(2^{h^*})$ complexity (lines 9–17). In the next paragraph, we will establish that the complexity of FIND_MEDIANS dominates the latter.

We study Algorithm 5 to analyze the complexity of FIND_MEDIANS. In Step 1, $O(2^{h^*} \cdot \alpha)$ buckets are created. Step 2 executes GET_ESTIMATES with these buckets and $O(|\mathcal{U}|)$ population size, resulting in $O(2^{h^*} \cdot \alpha \cdot |\mathcal{U}|)$ complexity. In Step 3, the for loop iterates $O(2^{h^*})$ times. In each iteration, retrieving *relbuckets* is implemented in constant time by keeping the buckets sorted according to the node order and knowing that each node will result in exactly α buckets. Then, the execution of CALC_MED in Step 3 has complexity $O(\alpha \cdot \log(\alpha))$ due to the sorting in Algorithm 6. Combining the three steps, we find that the complexity of FIND_MEDIANS is $O(2^{h^*} \cdot \alpha + 2^{h^*} \cdot \alpha \cdot |\mathcal{U}| + 2^{h^*} \cdot \alpha \cdot \log(\alpha))$. Typically, the population size is orders of magnitude larger than $\log(\alpha)$, i.e., $|\mathcal{U}| >> \log(\alpha)$. Assuming this holds, we can conclude that the complexity of FIND_MEDIANS is $O(2^{h^*} \cdot \alpha \cdot |\mathcal{U}|)$.

Recall that the proposed kd-tree solution performs $O(h^*)$ iterations, and in each iteration, the complexity is dominated by Find_Medians which is $O(2^{h^*} \cdot \alpha \cdot |\mathcal{U}|)$. Therefore, the overall complexity of the proposed kd-tree solution is $O(h^* \cdot 2^{h^*} \cdot \alpha \cdot |\mathcal{U}|)$.

6.6 Impact of α on LDP Approximation Error

Recall that the α parameter controls the number of buckets in Find_Medians. This α can be considered as a parameter that affects how well the underlying distribution will be approximated under LDP. When α is small, the density of each bucket is high but the approximation is too coarse (i.e., it suffers from over-smoothing). On the other hand, when α is large, the approximation is very detailed but each bucket has too low density, which causes LDP noise to overwhelm the true densities. Thus, in addition to the experimental analysis provided in Section 5.5, here we provide some conceptual and theoretical insights into the impacts of α.

For a set of n data points, let $f(x)$ denote their true distribution and $\tilde{f}(x)$ denote their bucketized distribution. The **Integrated MSE (IMSE)** is defined as:

$$\text{IMSE} = \mathbb{E}\left[\int (f(x) - \tilde{f}(x))^2 dx\right]. \tag{17}$$

One way to theoretically find the optimal value of α would be to minimize IMSE. Several rules exist for achieving this in the context of choosing bucket sizes for histogram approximation, e.g., Sturges's rule [37], Scott's rule [35], and the Freedman–Diaconis rule [16]. However, many such rules require knowledge of the underlying data distribution, which makes them inapplicable to LDP since the server does not observe the true distribution. That is, in case of LDP, the server does not observe $f(x)$ but rather its noisy version $\hat{f}(x)$. In expectation, $\mathbb{E}[\hat{f}(x)] = f(x)$ since the OUE protocol is unbiased. Following [40], its variance is:

$$n \cdot \frac{4e^{\varepsilon}}{(e^{\varepsilon} - 1)^2}, \tag{18}$$

which can be introduced into the IMSE calculation. However, the optimization of IMSE is a challenging research problem, since arriving at a rule requires making assumptions regarding $f(x)$. It becomes even more challenging under LDP since IMSE must be optimized using $\hat{f}(x)$, for which the expected value and variance are available, rather than $f(x)$. Hence, we leave the solution to this theoretical optimization as a future work direction.

6.7 Impacts of Post-Processing (PP)

Since LDP protocols introduce perturbation on the user side to ensure privacy, the statistics estimated by the server become noisy, leading to utility loss. One approach to reducing utility loss in LDP is the use of PP methods [25, 42]. In this section, we explore the impacts of PP methods on our quadtree and kd-tree solutions. We integrate PP methods into our solutions as follows: For quadtrees, the integration is done in Algorithm 3. In this algorithm, the key step that applies LDP is line 7; the rest of the algorithm utilizes the estimates computed on line 7. Thus, we integrate PP methods between lines 7 and 8. For kd-trees, the key step that applies LDP is line 19 of Algorithm 5. Thus, we integrate PP methods between lines 19 and 20 of Algorithm 5. We integrate the following PP methods one by one, and empirically measure their AQEs.

Base-Pos. $estimates[j]$ should be non-negative for all j, but due to LDP perturbation, some of them can become negative. Base-Pos converts all negative estimates to 0:

$$estimates'[j] = \begin{cases} estimates[j] & \text{if } estimates[j] \geq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{19}$$

Table 8. AQEs of Quadtrees without Post-Processing (No-PP) and with Varying PP Methods

	$\varepsilon = 0.5$					$\varepsilon = 1.0$				
	No-PP	Base-Pos	Norm	Norm-Mul	Power	No-PP	Base-Pos	Norm	Norm-Mul	Power
Brightkite	0.067	0.064	0.057	0.060	0.067	0.031	0.031	0.029	0.028	0.035
Gowalla	0.071	0.073	0.062	0.067	0.075	0.038	0.036	0.038	0.038	0.038
Kaggle	0.108	0.119	0.100	0.126	0.121	0.051	0.048	0.046	0.057	0.050
Foursquare	0.096	0.095	0.089	0.102	0.091	0.039	0.039	0.036	0.039	0.039

Table 9. AQEs of Kd-trees without Post-Processing (No-PP) and with Varying PP Methods

	$\varepsilon = 0.5$					$\varepsilon = 1.0$				
	No-PP	Base-Pos	Norm	Norm-Mul	Power	No-PP	Base-Pos	Norm	Norm-Mul	Power
Brightkite	0.11	0.20	0.26	0.19	0.20	0.12	0.11	0.25	0.07	0.10
Gowalla	0.13	0.09	0.16	0.11	0.14	0.09	0.08	0.16	0.08	0.09
Kaggle	0.04	0.10	0.18	0.10	0.09	0.04	0.05	0.19	0.03	0.05
Foursquare	0.11	0.17	0.27	0.16	0.16	0.05	0.10	0.18	0.07	0.10

Norm processes the estimates by adding a constant σ to each of them, such that their sum will become $|\mathcal{U}|$. The value of σ is selected in a way that ensures the sum becomes equal to $|\mathcal{U}|$:

$$estimates'[j] = estimates[j] + \sigma, \quad \text{such that} \quad \sum_j estimates'[j] = |\mathcal{U}|. \tag{20}$$

Norm-Mul converts negative estimates to 0. Then, it uses a multiplicative factor γ so that the estimates' sum becomes equal to $|\mathcal{U}|$:

$$estimates'[j] = \begin{cases} 0 & \text{if } estimates[j] < 0 \\ \gamma \times estimates[j] & \text{if } estimates[j] \geq 0 \end{cases}. \tag{21}$$

Power. The intuition behind Power is that many real-world data distributions follow a statistical distribution, such as a Gaussian or power law distribution. Therefore, Power fits a distribution to the estimates and aims to minimize the expected square error. For example, with $estimates \sim \mathcal{P}$ where $\mathcal{P}$ is a power law distribution, Power post-processes the estimates in a way that minimizes:

$$\min_{\mathcal{P}} \mathbb{E}\left[\sum_j (estimates'[j] - estimates[j])^2\right]. \tag{22}$$

The PP methods above were selected and adapted to our context since they are prominent PP methods which are also suitable in our context [25, 42]. We compare AQEs of these PP methods versus the original versions of the algorithms that do not contain PP (denoted by No-PP). Consistent with our previous experiments, we set $\varepsilon = 0.5$ and 1, and we use $h^* = 4$ and $\theta = 10{,}000$. The results are shown in Tables 8 and 9 for quadtrees and kd-trees, respectively.

We observe from Table 8 that Norm is generally the best-performing PP method for quadtrees, and it also achieves utility improvement over No-PP. Other PP methods are sometimes better than No-PP, but sometimes they are worse. Thus, PP methods other than Norm do not seem to yield a consistent improvement. The improvement achieved by Norm is more significant in the case of $\varepsilon = 0.5$, but it is relatively less pronounced for $\varepsilon = 1.0$. Overall, for quadtrees, we find that using Norm is a good strategy to improve utility. We note that this is an intuitive finding—the fact that Norm achieves $\sum_j estimates'[j] = |\mathcal{U}|$ ensures that the bottom-up propagation step of the quadtree

solution will ensure $d(root) = |\mathcal{U}|$ and the remaining nodes' densities will also be consistent with respect to the population size.

On the other hand, for kd-trees, we observe from Table 9 that No-PP usually performs the best, especially for $\varepsilon = 0.5$. When $\varepsilon = 1.0$, Norm-Mul starts yielding some utility improvements on some datasets, but the improvements are relatively small. Thus, for kd-trees, we do not recommend the use of PP. There are two main reasons why PP does not work well on kd-trees. First, without PP, positive and negative bucket counts cancel each other out in Find_Medians. However, when we use PP and they do not cancel out, then the median is estimated less accurately. Second, since kd-trees divide the space according to the median, the densities of the resulting children nodes are equal (assuming the "improved" strategy is used). Thus, PP methods do not change the densities of the children. Combining these two reasons, we find that the potential error increase in median estimation, together with the lack of improvement in estimating densities, cause No-PP to be preferable for kd-trees.

6.8 Extending Kd-trees to Higher Dimensions

While our article focuses on two-dimensional geospatial data (as established by the data model and notation in Section 2.1), it is possible to apply kd-trees in higher dimensions, e.g., 3D, 4D, and beyond. To facilitate this, our proposed kd-tree solution can be extended as follows. First, notice that Algorithm 4 initializes the current axis as the x-axis, and then alternates between x- and y-axes (lines 19–22) since there are two dimensions. For higher dimensions, the algorithm should alternate between each dimension iteratively, e.g., first x, then y, then z, then back to x. Second, the Find_Medians function in Algorithm 5 would need to be extended so that splits in newly added dimensions can be performed. In its current state, Algorithm 5 is able to split the x-axis or the y-axis (lines 2–18); however, these parts can be extended to split other axes as well since other axes would be a mirror image of the current implementation. Finally, minor changes need to be made to Algorithm 6 so that sorting is performed according to newly added axes (lines 2–5) and the median is set correctly for the newly added axes (lines 12–18). Following these changes, it is possible to apply our LDP kd-tree solution for privacy-preserving decompositions in higher dimensions.

There are several potential applications for extending our LDP kd-tree solution to higher dimensions. First, one can utilize kd-trees for 3D spatial data involving latitude, longitude, and altitude. This can be useful for Earth data or drone path planning. Second, kd-trees are also used frequently in computer graphics and 3D modeling of objects. Third, motion tracking and path planning data—which can be sensitive, e.g., if it involves humans' motions—can benefit from LDP kd-trees in higher dimensions such as 3D and 4D. Finally, numeric data with low to medium-dimensional feature spaces can be represented and decomposed using kd-trees.

7 Related Work

LDP for Spatial Data. LDP has gained widespread acceptance as a popular privacy notion in recent years, and it has also been deployed in the industry [2, 8, 11, 15, 21, 48]. With the rising popularity of LDP, there is increasing interest in applying it to spatial data, considering the ubiquity of LBSs and the need to protect users' location privacy. Hong et al. [23] proposed a perturbation mechanism designed to reduce the error of each perturbed location under LDP. Wang et al. [39] developed an LDP version of Staircase Randomized Response to privately collect users' locations while they remain useful for LBS applications such as traffic density estimation and k-nearest neighbors. Yang et al. [47] studied the problem of collecting location trajectories under LDP. Cunningham et al. [10] proposed a technique that utilizes hierarchical n-grams for real-world trajectory sharing with LDP. Du et al. [12] proposed LDPTrace for synthesizing realistic location trajectories using aggregate statistics collected with LDP. Kim et al. [26] and Navidan et al. [31] applied LDP to indoor

positioning data. Recently, Tire and Gursoy [38] studied the problem of ASDQ-LDP. Although these works apply LDP to spatial data, they do not have the goal of building spatial decompositions.

Spatial Decompositions under Centralized DP. Spatial decompositions have been relatively less studied under LDP, but there exist several works on building spatial decompositions under the centralized form of DP. Xiao et al. [43] presented a spatial decomposition algorithm based on kd-trees for handling statistical queries (e.g., OLAP queries) under DP. Cormode et al. [9] proposed algorithms to build hierarchical spatial decompositions such as quadtrees and kd-trees under DP. Zhang et al. [50] developed the PrivTree algorithm, which enabled constructing a DP spatial decomposition without requiring a pre-defined recursion limit (height). Grid-based decompositions under DP, which are a popular alternative to hierarchical decompositions, have been introduced by Qardaji et al. [33] in the form of uniform and adaptive grids. They have later been used in other works such as [18, 19]. Hierarchical reference systems proposed by He et al. [22] employ hierarchically organized grids with different granularities, ordered from coarse to fine-grained.

More recently, Niknami et al. [32] proposed personalized DP and personalized noise addition for indexing geometric objects in spatial databases. Quadtrees and kd-trees are used as spatial indices. In the work of Li et al. [28], a data-dependent adaptive density grid decomposition is used at the first layer, and then a quadtree decomposition is adopted for further splitting. Yan et al. [45] proposed an unbalanced quadtree partitioning algorithm for improving query accuracy in publishing spatial data with DP. Similarly, Liu et al. [29] also proposed a quadtree algorithm for improving query accuracy under DP, by balancing noise error and uniformity error. A new decomposition structure called Homogeneous Tree Framework was proposed by Shaham et al. [36], which shares similarities to kd-trees. All of these works operate under the assumptions of centralized DP rather than LDP, hence they are not comparable to our work.

Spatial Decomposition and Aggregation under LDP. As stated above, although hierarchical spatial decompositions have been studied under centralized DP, their applications to LDP are relatively few. Chen et al. [6] proposed a variant of LDP, called **Personalized LDP (PLDP)**, for spatial data aggregation. However, Chen et al. [6] assume a safe region-based variant of LDP called PLDP rather than LDP. In addition, it performs clustering-based aggregation rather than hierarchical decomposition. The problem of estimating spatial frequency distributions was studied by Hong et al. [24]. In [38], ASDQ-LDP was experimentally compared with Hong et al. [24], and it was shown that ASDQ-LDP provides lower errors. Hence, in this article, we compare our solutions with ASDQ-LDP. There exist two works that perform grid-based decompositions under LDP: PrivAG proposed by Yang et al. [47] and the concurrent work AAG by Balioglu et al. [4]. However, these two works perform grid-based decompositions rather than hierarchical decompositions, hence they would not be applicable in contexts where a hierarchical decomposition is necessary or desired. In short, none of the above works study hierarchical decompositions such as quadtrees and kd-trees under LDP, which is the focus of our work. Finally, our prior work [1] proposed algorithms for building quadtrees under LDP. In this article, we improve [1] by proposing algorithms for kd-trees in addition to quadtrees, and by demonstrating utility improvements compared to another state-of-the-art work [38].

8 Conclusion

In this article, we studied the problem of building hierarchical spatial decompositions under LDP, focusing on two popular types of decompositions: quadtrees and kd-trees. We proposed two solutions for quadtrees: a baseline solution that adapts an iterative strategy inspired by the centralized DP literature and a newly proposed solution that relies on a single LDP data collection step. Furthermore, we proposed a solution for building kd-trees which relies on a novel algorithm for learning medians from the user population in every depth of the kd-tree. Two variants of

the kd-tree algorithm were developed based on their strategies for determining children nodes' densities ("initial" and "improved" strategies). We experimentally evaluated all algorithms using four real-world datasets, several ε values, and metrics. Results showed the superiority of the proposed solution compared to the baseline solution for quadtrees and the superiority of the improved strategy over the initial strategy for kd-trees. We also validated that our median computation method enables accurate estimation of nodes' medians under LDP. Finally, we showed that both our quadtree and kd-tree solutions achieve substantial utility improvement in ASDQ-LDP, compared to a state-of-the-art solution [38].

In future work, we plan to study other spatial decomposition methods, such as octrees and hybrid decompositions under LDP. In addition, we plan to study the adaptation and application of our LDP median computation method in alternative domains and data aggregation tasks, such as federated learning.

References

[1] Ece Alptekin and M. Emre Gursoy. 2023. Building quadtrees for spatial data under local differential privacy. In *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy (DBSec)*. Springer, 22–39.

[2] Apple. 2020. Learning with Privacy at Scale. Retrieved from https://machinelearning.apple.com/docs/learning-with-privacy-at-scale/appledifferentialprivacysystem.pdf

[3] Eugene Bagdasaryan, Peter Kairouz, Stefan Mellem, Adrià Gascón, Kallista Bonawitz, Deborah Estrin, and Marco Gruteser. 2022. Towards sparse federated analytics: Location heatmaps under distributed differential privacy with secure aggregation. *Proceedings on Privacy Enhancing Technologies* 4 (2022), 162–182.

[4] Berkay Kemal Balioglu, Alireza Khodaie, Ameer Taweel, and Mehmet Emre Gursoy. 2024. Grid-based decompositions for spatial data under local differential privacy. arXiv:2407.21624. Retrieved from https://arxiv.org/abs/2407.21624

[5] Rui Chen, Gergely Acs, and Claude Castelluccia. 2012. Differentially private sequential data publication via variable-length n-grams. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 638–649.

[6] Rui Chen, Haoran Li, A. K. Qin, Shiva Prasad Kasiviswanathan, and Hongxia Jin. 2016. Private spatial data aggregation in the local setting. In *Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 289–300.

[7] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. 2011. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 1082–1090.

[8] Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. 2018. Privacy at scale: Local differential privacy in practice. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1655–1658.

[9] Graham Cormode, Cecilia Procopiuc, Divesh Srivastava, Entong Shen, and Ting Yu. 2012. Differentially private spatial decompositions. In *Proceedings of the 28th IEEE International Conference on Data Engineering*. IEEE, 20–31.

[10] Teddy Cunningham, Graham Cormode, Hakan Ferhatosmanoglu, and Divesh Srivastava. 2021. Real-world trajectory sharing with local differential privacy. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2283–2295.

[11] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. 2017. Collecting telemetry data privately. In *Proceedings of the Advances in Neural Information Processing Systems*, 3571–3580.

[12] Yuntao Du, Yujia Hu, Zhikun Zhang, Ziquan Fang, Lu Chen, Baihua Zheng, and Yunjun Gao. 2023. LDPTrace: Locally differentially private trajectory synthesis. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1897–1909.

[13] Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[14] ECML/PKDD. 2015. Taxi Trajectory Prediction Dataset. Retrieved from https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i/data

[15] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1054–1067.

[16] David Freedman and Persi Diaconis. 1981. On the histogram as a density estimator: L2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie Und verwandte Gebiete* 57, 4 (1981), 453–476.

[17] M. Emre Gursoy, Ling Liu, Ka-Ho Chow, Stacey Truex, and Wenqi Wei. 2022. An adversarial approach to protocol analysis and selection in local differential privacy. *IEEE Transactions on Information Forensics and Security* 17 (2022), 1785–1799.

[18] Mehmet Emre Gursoy, Ling Liu, Stacey Truex, and Lei Yu. 2018. Differentially private and utility preserving publication of trajectory data. *IEEE Transactions on Mobile Computing* 18, 10 (2018), 2315–2329.

[19] Mehmet Emre Gursoy, Ling Liu, Stacey Truex, Lei Yu, and Wenqi Wei. 2018. Utility-aware synthesis of differentially private and attack-resilient location traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 196–211.

[20] M. Emre Gursoy, Vivekanand Rajasekar, and Ling Liu. 2020. Utility-optimized synthesis of differentially private location traces. In *Proceedings of the IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 30–39.

[21] Mehmet Emre Gursoy, Acar Tamersoy, Stacey Truex, Wenqi Wei, and Ling Liu. 2019. Secure and utility-aware data collection with condensed local differential privacy. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2365–2378.

[22] Xi He, Graham Cormode, Ashwin Machanavajjhala, Cecilia M. Procopiuc, and Divesh Srivastava. 2015. DPT: Differentially private trajectory synthesis using hierarchical reference systems. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1154–1165.

[23] Daeyoung Hong, Woohwan Jung, and Kyuseok Shim. 2021. Collecting geospatial data with local differential privacy for personalized services. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2237–2242.

[24] Daeyoung Hong, Woohwan Jung, and Kyuseok Shim. 2022. Collecting geospatial data under local differential privacy with improving frequency estimation. *IEEE Transactions on Knowledge and Data Engineering* 35, 7 (2022), 6739–6751.

[25] Jinyuan Jia and Neil Zhenqiang Gong. 2019. Calibrate: Frequency estimation and heavy hitter identification with local differential privacy via incorporating prior knowledge. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2008–2016.

[26] Jong Wook Kim, Dae-Ho Kim, and Beakcheol Jang. 2018. Application of local differential privacy to collection of indoor positioning data. *IEEE Access* 6 (2018), 4276–4286.

[27] Yu Hsuan Kuo, Cho Chun Chiu, Daniel Kifer, Michael Hay, and Ashwin Machanavajjhala. 2018. Differentially private hierarchical count-of-counts histograms. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1509–1521.

[28] Shuyu Li, Yue Geng, and Yingle Li. 2021. A differentially private hybrid decomposition algorithm based on quad-tree. *Computers & Security* 109 (2021), 102384.

[29] Gang Liu, Ziwen Tang, Bo Wan, Yanfei Li, and Yan Liu. 2022. Differential privacy location data release based on quadtree in mobile edge computing. *Transactions on Emerging Telecommunications Technologies* 33, 6 (2022), e3972.

[30] Mohamed F. Mokbel, Chi-Yin Chow, and Walid G. Aref. 2006. The new casper: Query processing for location services without compromising privacy. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, Vol. 6, 763–774.

[31] Hojjat Navidan, Vahideh Moghtadaiee, Niki Nazaran, and Mina Alishahi. 2022. Hide me behind the noise: Local differential privacy for indoor location privacy. In *Proceedings of the 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 514–523.

[32] Nadia Niknami, Mahdi Abadi, and Fatemeh Deldar. 2020. A fully spatial personalized differentially private mechanism to provide non-uniform privacy guarantees for spatial databases. *Information Systems* 92 (2020), 101526.

[33] Wahbeh Qardaji, Weining Yang, and Ninghui Li. 2013. Differentially private grids for geospatial data. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 757–768.

[34] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys* 16, 2 (1984), 187–260.

[35] David W. Scott. 1979. On optimal and data-based histograms. *Biometrika* 66, 3 (1979), 605–610.

[36] Sina Shaham, Gabriel Ghinita, Ritesh Ahuja, John Krumm, and Cyrus Shahabi. 2023. HTF: Homogeneous tree framework for differentially-private release of large geospatial datasets with self-tuning structure height. *ACM Transactions on Spatial Algorithms and Systems* 9, 4 (2023), 1–30.

[37] Herbert A. Sturges. 1926. The choice of a class interval. *Journal of the American Statistical Association* 21, 153 (1926), 65–66.

[38] Ekin Tire and M. Emre Gursoy. 2024. Answering spatial density queries under local differential privacy. *IEEE Internet of Things Journal* 11, 10 (2024), 17419–17436.

[39] Han Wang, Hanbin Hong, Li Xiong, Zhan Qin, and Yuan Hong. 2022. L-SRR: Local differential privacy for location-based services with staircase randomized response. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 2809–2823.

[40] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. 2017. Locally differentially private protocols for frequency estimation. In *Proceedings of the 26th USENIX Security Symposium*, 729–745.

[41] Tianhao Wang, Ninghui Li, and Somesh Jha. 2018. Locally differentially private frequent itemset mining. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE.

[42] Tianhao Wang, Milan Lopuhaa-Zwakenberg, Zitao Li, Boris Skoric, and Ninghui Li. 2020. Locally differentially private frequency estimation with consistency. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*.

[43] Yonghui Xiao, Li Xiong, and Chun Yuan. 2010. Differentially private data release through multidimensional partitioning. In *Proceedings of the 7th VLDB Workshop on Secure Data Management (SDM)*. Springer, 150–168.

[44] Xingxing Xiong, Shubo Liu, Dan Li, Zhaohui Cai, and Xiaoguang Niu. 2020. A comprehensive survey on local differential privacy. *Security and Communication Networks* 2020 (2020), 1–29.

[45] Yan Yan, Xin Gao, Adnan Mahmood, Tao Feng, and Pengshou Xie. 2020. Differential private spatial decomposition and location publishing based on unbalanced quadtree partition algorithm. *IEEE Access* 8 (2020), 104775–104787.

[46] Dingqi Yang, Daqing Zhang, Vincent W. Zheng, and Zhiyong Yu. 2015. Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 1 (2015), 129–142.

[47] Jianyu Yang, Xiang Cheng, Sen Su, Huizhong Sun, and Changju Chen. 2022. Collecting individual trajectories under local differential privacy. In *Proceedings of the 2022 23rd IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 99–108.

[48] Mengmeng Yang, Lingjuan Lyu, Jun Zhao, Tianqing Zhu, and Kwok-Yan Lam. 2020. Local differential privacy and its applications: A comprehensive survey. arXiv:2008.03686. Retrieved from https://arxiv.org/abs/2008.03686

[49] Qingqing Ye, Haibo Hu, Xiaofeng Meng, and Huadi Zheng. 2019. PrivKV: Key-value data collection with local differential privacy. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 317–331.

[50] Jun Zhang, Xiaokui Xiao, and Xing Xie. 2016. PrivTree: A differentially private algorithm for hierarchical decompositions. In *Proceedings of the 2016 International Conference on Management of Data*, 155–170.

Received 11 April 2024; revised 31 March 2025; accepted 30 May 2025