A Reciprocal Framework for Spatial *K*-Anonymity

Gabriel Ghinita, Keliang Zhao, Dimitris Papadias, and Panos Kalnis

Abstract—Spatial *K*-anonymity (SKA) exploits the concept of *K*-anonymity in order to protect the identity of users from location-based attacks. The main idea of SKA is to replace the exact location of a user *U* with an *anonymizing spatial region* (ASR) that contains at least *K*-1 other users, so that an attacker can pinpoint *U* with probability at most 1/*K*. Simply generating an ASR that includes *K* users does not guarantee SKA. Previous work defined the *reciprocity* property as a sufficient condition for SKA. However, the only existing *reciprocal* method, *Hilbert Cloak*, relies on a specialized data structure. In contrast, we propose a general framework for implementing reciprocal algorithms using any existing spatial index on the user locations. We discuss ASR construction methods with different tradeoffs on effectiveness (i.e., ASR size) and efficiency (i.e., construction cost). Then, we present case studies of applying our framework on top of two popular spatial indices (namely, R*-trees and Quad-trees). Finally, we consider the case where the attacker knows the query patterns of each user. The experimental results verify that our methods outperform *Hilbert Cloak*. Moreover, since we employ general-purpose spatial indices, the proposed system is not limited to anonymization, but supports conventional spatial queries as well.

Index Terms—Location-based Services, Anonymity, Privacy, Spatial Databases

1 Introduction

THE embedding of positioning capabilities (e.g., GPS) in mobile devices has triggered several exciting applications. However, at the same time, it has raised serious concerns about the risks of revealing sensitive information in *location based services* (LBS) [BS03, BWJ05]. Consider a user *U* that wants to issue a query about the nearest nightclub to an untrustworthy LBS through a non-secure channel, without being identified. *U* establishes a secure connection (e.g., SSL) with an *anonymizer*, which is a trusted server (services for anonymous web surfing are common). The anonymizer removes the user ID from the query and forwards it to the LBS. Nevertheless, the LBS requires the coordinates of *U* in order to process the query. If the LBS is malicious (e.g., it collects and sells personal data and habits for unsolicited advertisements), it can relate these coordinates to *U* through a variety of techniques. For instance, *U* may issue the query from a residence, in which case the coordinates can be converted to a street address and linked to *U* using an on-line white pages service. Furthermore, since the communication channel between the LBS and the anonymizer is not secure, an eavesdropper who physically observes *U* can easily associate the user with the query. In general, removing the ID is not enough for hiding the identity. Moreover, simply perturbing the exact location with an area around *U* may not suffice. If, for instance, *U* is in a sparse rural region, even a relatively large area may not enclose other users.

Several systems aim at solving these problems using the concept of *spatial K-anonymity* (SKA). SKA requires that an attacker can pinpoint the user that issues a query with probability not exceeding 1/*K*, even if, in the worst case, all user locations are known to the attacker. Assume that the anonymizer receives a location based query from *U*. Then, it could pick *K*-1 random users and forward *K* independent queries (including the real one) to the LBS. This simple method achieves SKA because the query could originate from any client with equal probability 1/*K*. However, it has several problems: (i) depending on the value of *K*, a potentially large number of locations are transmitted to the LBS, (ii) the LBS has to process *K* independent queries and send back all their results, and (iii) the anonymizer reveals the exact locations of *K* users, which is undesirable in many applications.

To overcome these problems, most existing systems use the framework of Fig. 1.1. (i) A user/client *U* sends its query and *anonymity requirement K* to the anonymizer, which maintains the current locations of numerous clients. (ii) The anonymizer removes the user ID and selects an *anonymizing set* (AS) that contains *U* and at least *K*-1 other clients in its vicinity. The *K*-

- *G. Ghinita and Panos Kalnis are with the Department of Computer Science, National University of Singapore, Email: {ghinitag, kalnis}@comp.nus.edu.sg*
- *K.Zhao and D. Papadias are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Email: cs_zkx@stu.ust.hk, dimitris@cse.ust.hk*

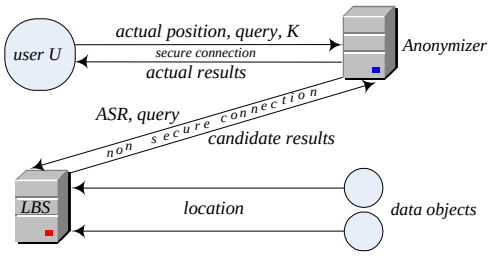
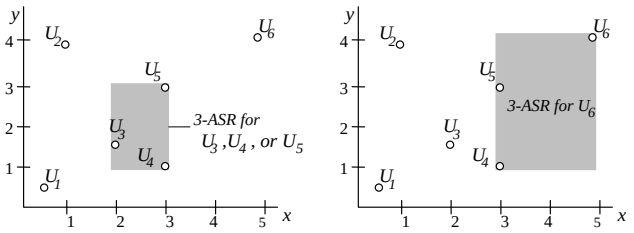


Fig. 1.1 Framework for spatial anonymization



(a) 3-ASR for U_3, U_4, or U_5 (b) 3-ASR for U_6

Fig. 1.2 Examples of spatial cloaking

anonymizing spatial region (K-ASR or ASR) is an area that spatially encloses AS. (iii) The anonymizer forwards the ASR to the LBS that stores the spatial data (e.g., nightclub dataset). (iv) The LBS processes the query and returns to the anonymizer a set of candidate results. (v) The anonymizer removes the false hits and forwards the actual result to U.

As an example, consider that U_3 in Fig. 1.2a issues a location-based nearest neighbor (NN) query with K=3. The anonymizer computes AS={U_3,U_4,U_5}, generates the corresponding ASR (shaded rectangle), and forwards it to the LBS. Because the LBS only obtains the ASR, but not the query point, it retrieves the NN for every possible location in the ASR. This candidate set is returned to the anonymizer that performs the filtering and forwards the actual NN to U_3. The process of replacing a specific location with an ASR is called *spatial cloaking*. A cloaking algorithm is *secure*, if it satisfies SKA. In our example, given the ASR, an attacker should not be able to infer U_3 as the query origin with probability that exceeds 1/3. Although the ASR can have arbitrary shape, it is common to use regular shapes, e.g., minimum bounding rectangles (MBRs), because they incur small network overhead (when transmitted to the LBS) and facilitate query processing. Furthermore, the ASR is independent of the query type; for instance, the ASR of Fig. 1.2a may have been created for a NN, or range query.

Nevertheless, simply generating an ASR that includes K clients is not sufficient for SKA. Consider an algorithm, called *Center Cloak* in the sequel, that given a query from U, finds its K-1 closest users and sets the ASR as the MBR that encloses them. In fact, a similar technique is proposed in [CML06] for anonymization in peer-to-peer systems (i.e., the K-ASR contains the query issuing peer and its K-1 nearest nodes). Given a query from U_3, U_4, or U_5, *Center Cloak* would generate the ASR of Fig. 1.2a. On the other hand, if the query originates from U_6, the ASR is the shaded rectangle in Fig. 1.2b. The second ASR violates SKA because an attacker can be sure that the query is issued by U_6, as the same ASR could not be generated for any other client. Specifically, any AS involving 3 users and created for U_4 or U_5, would contain U_3 (as in Fig. 1.2a), but not U_6. As we discuss in Section 2.2, most existing cloaking algorithms have similar problems[1].

To eliminate these problems, Kalnis et al. [KGMP07] introduced *reciprocity*, a sufficient property for SKA. Reciprocity requires that a set of users are always grouped together for a given K, or equivalently, each user in an AS lies in the ASRs of all other clients in the AS. Formally:

***Definition* [*Reciprocity*[2]].** Consider a user U issuing a query with anonymity degree K, anonymizing set AS, and anonymizing spatial region ASR. AS satisfies *reciprocity* if (i) it contains U and at least K-1 additional users, and (ii) every user in AS also generates the same anonymizing set AS for the given K.

If every AS satisfies reciprocity, an ASR may have originated from every user in the corresponding AS with equal probability $1/|\text{AS}|$, where $|\text{AS}|$ is the cardinality of AS. Because $|\text{AS}| \geq K$, the probability of identifying the query issuer does not exceed $1/K$. For instance, in Fig. 1.2a AS = {U_3,U_4,U_5} is reciprocal because it is generated (by *Center Cloak*) for each query with K=3 issued by U_3, U_4, or U_5. Therefore, the three users are indistinguishable to an attacker. In contrast, AS = {U_4,U_5,U_6} in Fig. 1.2b is not reciprocal because if the query were issued by U_4 or U_5, the AS would be {U_3,U_4,U_5} as in Fig. 1.2a (in other words, U_6 is not in the 3-ASR of U_4 and U_5). A cloaking algorithm is *reciprocal*, if *every* AS (i.e., for each possible user and K) satisfies reciprocity. It can be proved that reciprocal algorithms are secure.

In addition to being secure, spatial cloaking should be efficient and effective. *Efficiency* means that the cost of generating the ASR (at the anonymizer) should be minimized for better scalability and faster service. *Effectiveness* refers to the area of the ASR, which should also be minimized. Specifically, a large ASR incurs high processing overhead (at the LBS) and network cost (for transferring numerous candidate results from the LBS to the anonymizer). In real-world services, users may

[1] Additionally, *Center Cloak* compromises SKA in another way: often, the querying user U is closest to the ASR center. Thus, a simple "center-of-ASR" attack would correctly guess U with probability that far exceeds $1/K$, especially for large values of K.

[2] Reciprocity has been independently formulated as the *k-sharing* property in [CM07].

be charged depending on their anonymity requirements and the overhead that these requirements impose on the system. Reciprocity has a negative impact on effectiveness because, in general, it leads to relatively large ASRs. For instance, if U_4, U_5 and U_6 are grouped in the same AS as shown in Fig. 1.2b for a query of U_6, then this AS should also be used for queries (with the same K) issued by U_4 and U_5. On the other hand, if reciprocity were not required, the anonymizer could use the smaller ASR of Fig. 1.2a.

Our work is motivated by the fact that the only existing reciprocal technique, *Hilbert Cloak* [KGMP07], requires a specialized annotated B+-tree, which is used exclusively for spatial cloaking. In contrast, we envision a comprehensive anonymization service which, in addition to spatial cloaking, is capable of answering added-value queries, such as "find buddies in my vicinity". To support efficiently a variety of spatial queries, the anonymizer must index the moving users by a general purpose spatial index, such as an R*-tree or a Quad-tree. Consequently, spatial cloaking should also be implemented on top of any existing spatial index. Guided by these requirements, our contributions are:

(i) We propose a general framework for obtaining reciprocal (i.e., secure) cloaking algorithms based on an underlying spatial index. We develop an algorithm that, for a user U indexed by any tree structure, identifies the *partition sub-tree* which can be used independently to anonymize U, without violating reciprocity for any user.

(ii) We present two novel reciprocal partitioning algorithms, called *GH* and *AR*, which are based on our framework. Given a user U and the corresponding partition sub-tree, GH and AR generate the ASR for U. GH is best in terms of efficiency, whereas AR maximizes effectiveness.

(iii) We introduce and incorporate in our framework, the *frequency-aware reciprocity* property, which guarantees SKA even if the query frequency varies among users and is known to the attacker (e.g., a taxi driver asks more location-based queries than a casual commuter). Previous work assumed that all users have the same query frequency.

(iv) We present two implementations of the *reciprocal framework*, based on the R*-tree [BKSS90] and on the Quad-tree [S90], and evaluate the tradeoffs between efficiency and effectiveness. We show that, due to the superior clustering of the spatial indices, the proposed framework outperforms *Hilbert Cloak*.

The rest of the paper is organized as follows: Section 2 overviews the related work. Section 3 states our assumptions, and introduces the *reciprocal framework*. Section 4 presents the partitioning methods. Section 5 addresses the case of variable query frequencies. Section 6 contains the experimental evaluation and Section 7 concludes with directions for future work.

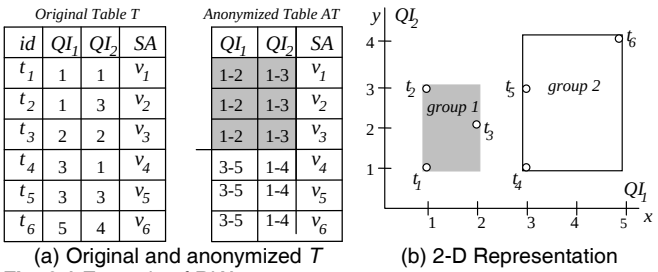


Fig. 2.1 Example of RKA

2 BACKGROUND

This section surveys relational and spatial *K*-anonymity methods and discusses the relationship among them.

2.1 Relational K-anonymity (RKA)

Relational *K-anonymity* has received considerable attention due to the need of several organizations to publish data (often called *microdata*) without revealing the identity of individual records. Consider a hospital authority that wishes to release information about its patients for medical research. Even if the identifying attributes (e.g., name) are removed, an attacker may be able to identify specific patients using combinations of other attributes (e.g., zipcode, gender, birth date), called *quasi-identifiers* (*QI*). A table is *K*-anonymized if each record is indistinguishable from at least *K*-1 other records with respect to the *QI* set [S01]. Records with identical *QI* values form an *anonymized group*. The most common form of anonymization is *generalization*, which involves replacing specific *QI* attribute values with more general ones.

Generalization can be applied to categorical and numerical attributes. In the first case, there is a predefined hierarchy of possible generalizations (e.g., a city name is replaced by the state or country). For numerical attributes, the existence of a hierarchy is not necessary, but values may be substituted by arbitrary ranges, whose extent depends on their density. Fig. 2.1a illustrates a generalization example for a table T involving two numeric *QI* and a sensitive attribute *SA* (to be discussed shortly). AT is a 3-anonymized version of T. AT contains groups, each with at least 3 tuples that have identical *QI* values. Fig. 2.1b contains a visualization of AT, where each group is represented by a 2-D range.

Several generalization algorithms have been proposed. Optimal algorithms that consider each attribute individually (i.e., single-dimensional) appear in [BA05] and [LDR05]. *Mondrian* [LDR06] is a multi-dimensional technique (i.e., it maps the Cartesian product of multiple attributes), which splits the space recursively similarly to *k*D-trees. Another multi-dimensional method based on R+-trees is presented in [IN07]. Finally, [AFK+06, XWP+06] describe generalization approaches based on clustering. Since all algorithms replace specific values with ranges, they incur information loss.

In some cases, RKA is not sufficient for protecting the privacy of microdata. Returning to the example of Fig. 2.1, assume that *AT* stores patients' data and the sensitive attribute is a disease. Ideally, it should not be possible to link a disease (e.g., v_1) to a specific record (e.g., t_1) with probability that exceeds 1/3. This condition holds if v_1 is different from v_2 and v_3. If, however, v_1 is the same as one or both v_2 and v_3, this probability increases to 2/3 or 1, respectively. To overcome this problem, Machanavajjhala et al. [MGKV06] propose the concept of *l*-diversity. A table is *l*-diverse if each anonymized group contains at least *l* "well-represented" sensitive attribute values. Ghinita et al. [GKKM07] develop efficient algorithms for RKA and *l*-diversity by mapping the multidimensional space to one dimension.

Permutation [XT06, ZKSY07] is an alternative to *QI* generalization that achieves *l*-diversity by randomly shuffling the sensitive values among records in each anonymized group. By avoiding generalization, permutation-based approaches have the potential of reducing information loss. However, the drawback is that *QI* values are disclosed in exact form, which makes the data vulnerable to linkage attacks (i.e., an adversary can confirm the presence of an individual in the microdata). Furthermore, as shown in [GKKM07], permutation-based methods such as [XT06, ZKSY07] do not account for *QI* proximity when creating groups, which may result in higher information loss than generalization-based counterparts.

2.2 Spatial *K*-anonymity (SKA)

We focus mainly on systems based on the user-anonymizer-LBS framework of Fig. 1.1. In *Casper* [MCA06], the anonymizer maintains the locations of the clients using a pyramid data structure, similar to a Quad-tree, where the minimum cell size corresponds to the anonymity resolution. Once the anonymizer receives a query from U, it uses a hash table on the user ID pointing to the lowest-level cell c where U lies. If c contains enough users (i.e., $|c| \geq K$) for the anonymity requirements, it forms the ASR. Otherwise ($|c|<K$), the horizontal c_h and vertical c_v neighbors of c are retrieved. If $|c \cup c_h| \geq K$ or $|c \cup c_v| \geq K$, the corresponding union of cells becomes the ASR. If both unions contain at least K users, the ASR is the one with the minimum cardinality. On the other hand, if $|c \cup c_h| < K$ and $|c \cup c_v| < K$, the anonymizer retrieves the parent of c and repeats this process recursively.

We use Fig. 2.2 to illustrate cloaking examples in *Casper*. Cells are denoted by the coordinates of their lower-left and upper-right points. Assume a query q with K=2. If q is issued by U_1 or U_2, the ASR is cell $\langle(0,2), (1,3)\rangle$. If q is issued by U_3 or U_4, the ASR is the union of cells $\langle(1,2), (2,3)\rangle \cup \langle(1,3), (2,4)\rangle$. Finally, if q is issued by U_5, the ASR is the entire data space. *Interval Cloak* [GG03] is similar to *Casper* in terms of both the data structure used by the anonymizer (a Quad-tree), and the cloaking algorithm. The main difference is that *Interval Cloak* does not consider neighboring cells at the same level when determining the ASR, but ascends directly to the ancestor level. For instance, a query with K=2 issued by U_3 or U_4 would generate the ASR $\langle(0,2), (2,4)\rangle$ (instead of $\langle(1,2), (2,4)\rangle$ for *Casper*). As we formally prove in the Appendix, *Casper* and *Interval Cloak* are secure only for uniform data because neither algorithm is reciprocal. In Fig. 2.2, although U_1 to U_4 are in the 2-ASR of U_5, U_5 is not in the 2-ASR of any of those users. Consequently, an attacker that detects an ASR covering the entire space can infer with high probability that it originates from U_5.

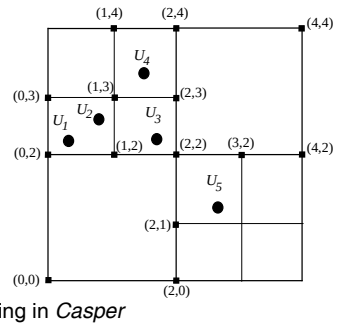


Fig. 2.2 Cloaking in *Casper*

To the best of our knowledge, the only provably secure spatial cloaking technique is *Hilbert Cloak* [KGMP07], which has also been implemented on a Peer-to-Peer system [GKS07]. *Hilbert Cloak* uses the Hilbert space filling curve [B71] to map the 2-D space into 1-D values. These values are then indexed by an *annotated B+-tree*, which supports efficient search by value or by rank (i.e., position in the 1-D sorted list). The algorithm partitions the 1-D sorted list into groups of K users (the last group may have up to $2K$-1 users). For a querying user U the algorithm finds the group where U belongs, and returns the MBR of the group as the ASR. The same ASR is returned for any user in a given group; therefore the algorithm is reciprocal.

While in all the above systems the cloaking mechanism is independent of the query type, processing at the LBS depends on the query. Range queries are straightforward: assume that U wants to retrieve the data objects within distance d from its current location. Instead of the position of U, the LBS receives an ASR and d. In order to compute the candidate results, it extends the ASR by d on all dimensions and searches for all objects in the extended ASR. On the other hand, for NN queries the candidate results can be retrieved using *range nearest neighbor search* [HL05], which finds the NN of any point inside an area. *Casper* and *Hilbert Cloak* include specialized processing techniques. We assume that the LBS implements these query processing mechanisms, and focus on spatial cloaking.

Finally, the privacy of user locations has also been studied in the context of related problems. *Clique Cloak* [GL05] combines spatial with *temporal cloaking*. Each query q specifies a temporal interval Δt that the corresponding user U is willing to wait. If within Δt, K-1 other clients in the vicinity of U also issue queries, all these queries are combined in a single ASR. Otherwise, q is rejected. *Probabilistic Cloaking* [CZBP06] does not apply the concept of SKA; instead, the ASR is a closed region around the query point, which is independent of the number of users inside. Given an ASR, the LBS returns the probability that each candidate result satisfies the query based on its location with respect to the ASR. Khoshgozaran and Shahabi [KS07] employ 1-D transformation and encryption to conceal both the spatial data and the queries from the LBS. Kamat et al. [KZTO05] propose a model for sensor networks and examine the privacy characteristics of different sensor routing protocols. Hoh and Gruteser [HG05] describe techniques for hiding the trajectory of users in applications that continuously collect location samples. The recent work in [GKK+08] shows how to obtain query privacy with cryptographic techniques, without relying on SKA. Nevertheless, the proposed cryptographic protocol incurs very high communication and computation cost, even when deployed in a parallel architecture.

2.3 Comparison between RKA and SKA

Relational and spatial k-anonymity have some important similarities, as well as differences. In terms of similarity, microdata tuples are often viewed as points in a multi-dimensional space, where each quasi-identifier attribute corresponds to a dimension (see Fig. 2.1b). Generalization, like spatial cloaking, replaces a point with a multidimensional range (i.e., an ASR). In fact, some relational generalization algorithms are directly motivated by spatial indices [LDR06, IN07], or clustering techniques [AFK+06, XWP+06]. Furthermore, the *certainty penalty* [XWP+06], used to measure information loss in RKA, is analogous to the effectiveness criterion in SKA that aims at minimizing the ASR.

Regarding the differences between RKA and SKA, RKA involves static data and a single value of K, whereas SKA deals with moving users and variable K. RKA methods precompute a partitioning of the *entire* table into a set of groups containing at least K tuples each. Every tuple belongs to exactly one group; therefore, the reciprocity property is satisfied by default. However, RKA methods are not suitable for SKA, due to efficiency considerations. Specifically, any RKA method (including bulk-loading of a spatial index, e.g., [IN07]), has computational complexity at least linear to the number of users. In contrast, SKA performs on-the-fly anonymization, involving only a small subset of the data in the neighborhood of the querying user. Although such an approach is scalable, unless special care is taken, a user may belong to multiple anonymized groups. Consequently, guaranteeing reciprocity is not trivial. Furthermore, since SKA is initiated by a querying user, *suppression*, often used in RKA to remove tuples that cannot be effectively generalized, should be avoided in SKA because it would lead to service denial for the corresponding user. Finally, as opposed to RKA, in SKA there is no concept of information loss; a small ASR is beneficial in terms of processing cost and network overhead, but a user will eventually obtain the correct results after they are filtered by the anonymizer, regardless of ASR size.

Enforcing SKA when the query frequency differs among users (Section 5) is a more difficult problem, and resembles the *t-closeness* [LTV07] paradigm in RKA. t-closeness requires that the distribution of sensitive values in each anonymized group must be the same as their table-wise distribution. However, existing t-closeness solutions are not suitable for variable frequency SKA for the same efficiency reasons cited earlier.

3 Reciprocal Framework

We consider the architecture of Fig. 1.1, where an anonymizer receives queries from geographically distributed users, removes the user IDs, hides their locations, and forwards the resulting ASRs to the LBS. Each query has a variable degree of anonymity K, which ranges between 1 (no privacy requirements) and the user cardinality (maximum privacy). The value of K is not subject to attacks since it is transferred from the client to the anonymizer through a secure channel. Queries are related to the position of the user (e.g., a user asking about its nearest restaurant, or all the restaurants within a range), but the type of query (i.e., NN or range) is not important for spatial cloaking.

We assume an attacker that (i) intercepts the ASR, (ii) knows the cloaking algorithm used by the anonymizer, and (iii) can obtain the current locations of all users. The first assumption implies that either the LBS is not trusted, or the communication channel between the anonymizer and the LBS is not secure. The second assumption is common in the literature since the data security techniques are typically public. The third assumption is motivated by the fact that users may often issue queries from the same locations (home, office), which could be identified through physical observation, triangulation, telephone catalogs etc. In the worst case, an attacker may be able to obtain the positions of all users in the AS of the query. Since it is difficult to model the exact knowledge available to the attacker, the third assumption is necessary in order to prove theoretically that the anonymization method is secure under the most pessimistic scenario. On the other hand, because in practice the attacker does not have all user

locations, it is important that the anonymization method does not reveal the position of any user, to avoid giving away additional information.

Similar to [GG03, GL05, CML06, MCA06, KGMP07] we focus on snapshot queries, where the attacker uses current data, but not historical information about repetitive queries by the same user at a specific location or time. This assumption is reasonable in practice because if a client obtains the items of interest (e.g., the closest restaurant), it is unlikely to ask the same query from the same location again in the future[3]. Finally, for ease of presentation, we consider that a query originates from any client with equal probability. We will remove this assumption in Section 5.

The anonymizer indexes the user locations by a hierarchical (i.e., tree-based) spatial index (e.g., R*-tree, Quad-tree, etc). Let U be the user issuing a query. We propose a general spatial cloaking algorithm, called *Reciprocal*, which traverses the tree and generates a reciprocal AS that contains U and at least K-1 users in its vicinity. The resulting ASR is the area that encloses all elements of the AS. Fig. 3.1 illustrates the pseudo-code for the reciprocal framework. Let N be the leaf node that contains U. *Reciprocal* traverses the tree in a bottom-up fashion, starting from N. The important observation here is that even if N contains enough ($\geq K$) points (we use the terms point, user and client, interchangeably) for the anonymity requirements, we still have to traverse the tree bottom-up (lines 1-2), if there is a node N' at the same level such that $0 < |N'| < K$ because N' may contain a user U' whose AS includes U.

Let AN be the ancestor of N when the bottom-up traversal stops. Each node at the level of AN is either empty (non-balanced trees such as the Quad-tree can have empty nodes at any level), or contains at least K users in its sub-tree. This implies that the AS can be determined locally within AN because all other queries (originating outside AN) do not need to include users of AN in their AS. Having established that AN can autonomously generate a K-ASR, *Reciprocal* traverses AN top-down towards U (lines 3-4) as long as each sub-tree has at least K points[4]. Let PN be the node in AN where the top-down traversal stops. PN includes U in its sub-tree and some of its child nodes have fewer than K points. PN is called the *partition node*, and corresponds to the lowest ancestor of U where we can achieve reciprocity. This is because all nodes in the sub-tree of AN and at the level of PN or above, contain at least K points, and thus can generate ASRs without using any points in PN.

[3] We emphasize that, in the case of continuous queries, our solutions can be immediately adapted to support historical reciprocity using the framework introduced in [CM07].

[4] While bottom-up traversal considers the cardinality of all nodes at a level, top-down only considers the cardinalities along a single path.

PN may contain numerous ($>>K$) points, which is likely to yield very large ASRs. *Partition* (line 5) eliminates this problem by grouping these points into disjoint *buckets*. The users in the same bucket b_U as U form the AS for the query. Several partitioning methods can be used (see Section 4), provided that:

(i) each bucket contains at least K and no more than $2K$-1 points. The lower bound is due to the K-anonymity requirement. The upper bound is due to the fact that if the cardinality of a bucket exceeds $2K$-1, the bucket can be split into smaller ones, each containing at least K users.

(ii) partitioning is independent of the query point. Each user in the node will generate the same partitioning for the same K. This property guarantees reciprocity.

After determining the AS, we form the ASR as the minimum bounding rectangle (MBR) covering AS. Note that the MBR may enclose some additional users that are not in AS. Compared to the fixed cells of *Casper* and *Interval Cloak*, MBRs adapt more effectively to the density around the query, i.e., if the query lies in an area with numerous users, the ASR is likely to be small. The disadvantage is that the MBR reveals the coordinates of points on its boundaries. Furthermore, in case that there are K (or more) users at the same location, the ASR may degenerate to a single point and disclose the positions of these users. A simple way to overcome these problems is to superimpose a grid where the cell size corresponds to the anonymity resolution. Then, the ASR sent to the LBS is the minimum enlargement that aligns the MBR to the grid. For the following discussion we omit this modification because the cell size depends on the application requirements for the anonymity resolution. Furthermore, spatial cloaking should be secure even if the attacker has complete knowledge of all the user positions.

Algorithm *Reciprocal* (query issuing user U, anonymity requirement K, node N)
// *initially N is the leaf node containing U*
1. While there is a non-empty node at the same level as N with $< K$ users
2. N = parent of N *//bottom-up traversal*
3. While N is not a leaf and (each child of N is either empty or contains $\geq K$ users)
4. N = child of N that contains U *//top-down traversal*
5. ASR=*Partition*(U, K, N)

Fig. 3.1 Reciprocal Cloaking

Reciprocal can be applied in conjunction with main-memory, or disk-based, and space-partitioning, or data-partitioning indices. The following example demonstrates *Reciprocal* on top of a Quad-tree. We will present R*-tree examples in Section 4.

Example [*Quad-tree Cloak*]: Fig. 3.2a illustrates an example where 6 clients are indexed by a Quad-tree (level 1 corresponds to the leaf cells). Assume a query with

K=2 originating from U_1. Since the cell ⟨(0,2), (1,3)⟩ of U_1 already contains 2 clients, *Casper* (and *Interval Cloak*) would use it directly as the ASR. This violates reciprocity because there are four level-1 cells that contain a single point; e.g., a query with K=2 from any of these cells could include U_1 in its AS. In contrast, *Quad-tree Cloak* (QC) ascends to level 2, where there still exist non-empty cells (e.g. ⟨(0,0), (2,2)⟩) with fewer than K users. Finally, QC reaches the root and sets AN=PN=⟨(0,0), (4,4)⟩. The same partition node is obtained for all users given K=2. In the above query, PN contains 6 points, although only 2 are necessary for the anonymity requirements. *Partition* groups these 6 points into buckets of 2 or 3 (i.e. K to 2K-1), and includes in AS the users from the same bucket b_U as U_1. Assuming that AS={U_1, U_2, U_6}, the ASR is the shaded MBR of Fig. 3.2a.

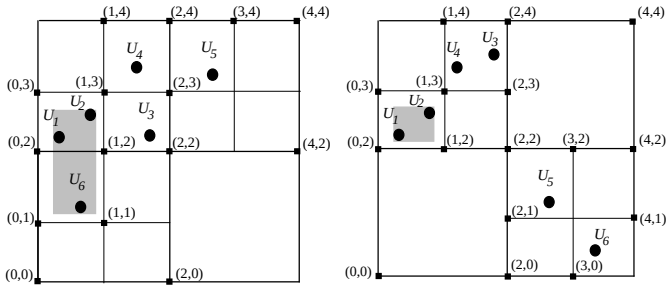


Fig. 3.2 Examples of reciprocal *Quad-tree Cloak* (QC)

Fig. 3.2b illustrates a second example, which also involves top-down traversal. Given again a query with K=2 from U_1, the bottom-up traversal stops at level 2 with AN=⟨(0,2), (2,4)⟩ because all non-empty cells at this level have at least 2 points. Furthermore, both non-empty children of AN, ⟨(0,2), (1,3)⟩ and ⟨(1,3), (2,4)⟩, also include 2 points each. Therefore, QC descends to level 1 and sets the partition node to PN=⟨(0,2), (1,3)⟩. Since this cell contains only U_1 and U_2, *Partition* returns directly the MBR of these users, without performing grouping. In general, if $|PN|$<2K, then there is a single bucket containing all the points in PN.

Theorem 1. *Reciprocal guarantees spatial K-anonymity*

Proof. We show that each AS generated by *Reciprocal* satisfies reciprocity, by retracing the steps of the algorithm. The bottom-up traversal terminates at an ancestor node AN such that each node at the level of AN is either empty or contains at least K users. Therefore, no user in AN belongs to the AS of any other user outside AN, and vice versa. The top-down traversal determines a partition node PN, that satisfies similar conditions, i.e., each sibling of PN (under the same parent) is either empty or has at least K points in its sub-tree. Thus, an AS can be assembled locally in PN without violating reciprocity. Finally, *Partition* generates buckets that by definition obey reciprocity, since each bucket contains at least K users, and each query with the same K from a user in PN will lead to exactly the same bucket.

Note that RKA methods based on spatial indices (KD-trees [LDR06] or R+-trees [IN07]) assume a fixed value of K, and compute a partitioning of the entire table into a set of groups containing at least K tuples. On the other hand, we consider variable K for each query, dynamic datasets, and we only care about the group containing the query. In this setting, methods such as [LDR06, IN07] would require bulk-loading an index for each query, leading to unnecessary (and very high) cost. Another simple alternative would be to maintain the spatial index incrementally. Given a query with a requirement K, we could load all the points and apply *Partition*(U, K, *root*), i.e., directly set AN = PN = *root*, without performing bottom-up and top-down traversals. As opposed to RKA generalization techniques, *Partition* returns a single group (instead of the entire anonymized table). However, this approach would also be inefficient because it has to access all the user locations, whereas *Reciprocal* only retrieves the users necessary for building the ASR.

Reciprocal needs the cardinality of the node with the minimum number of points per level. These numbers (i.e., one per level) can be explicitly stored and updated when there is change in the tree structure. Alternatively, if the index has a minimum node utilization M (e.g., R-trees), we can set the minimum cardinality at level i to its lower bound M^i (leaves are at level 1). This does not affect correctness, but may have a negative impact on performance, if the actual minimum cardinality is significantly higher than the lower bound. Furthermore, the top-down traversal requires the number of points in each entry of an intermediate node (line 3). We assume that this number is stored with the corresponding entry. Such structures are called *aggregate* indices, and have been used extensively in spatio-temporal data warehouses [TP05]. Finally, the location updates issued by the users are handled by the default algorithms of the indices, without any effect on anonymization.

4 Partitioning Methods

Given a partition node PN, *Partition* (line 5 in Fig. 3.1) splits the users inside the sub-tree of PN into buckets containing between K and 2K-1 users. Sections 4.1 and 4.2 present alternative partitioning algorithms with different tradeoffs in terms of efficiency and effectiveness. Although both techniques can be used with any spatial index, the examples assume an *aggregate* R*-tree (aR*-tree [TP05]), i.e., an R*-tree where each intermediate node entry stores the total number of points in the corresponding sub-tree. The resulting implementation is called *R-Tree Cloak* (RC). For ease of presentation, we

assume that the minimum node cardinality M^i per level i is M, where M is the R*-tree minimum node utilization (usually 40% of the node capacity).

4.1 Greedy Hilbert Partitioning (GH)

Let LN be the leaf node containing the query issuer. We first consider that partitioning takes place at the leaf level, i.e., $K \leq M$ and PN=LN. Similar to *Hilbert Cloak* [KGMP07], GH sorts the points in LN according to their Hilbert value. The Hilbert space filling curve transforms the multi-dimensional coordinates of each user U into an 1-D value $H(U)$. Fig. 4.1 illustrates the Hilbert curve for a 2-D space using a 8x8 space partitioning. A point U is assigned the value $H(U)$ of the cell that covers it. If two users are near each other in the 2-D space, they are likely to be close in the 1-D transformation. Given a query with required anonymization degree K, GH assigns the first K points (in the Hilbert order) to the first bucket, the next K points to the second bucket and so on. Consequently, each bucket contains exactly K users, except for the last one that may include up to 2K-1 users. Let $r(U)$ be the rank of U in the Hilbert order ($1 \leq r(U) \leq |LN|$). The bucket b_U of U contains all clients whose ranks are in the range [s, e], where $s = r_U - (r_U - 1) \bmod K$ and $e = s + K - 1$ (unless b_U is the last bucket).

Fig. 4.1 elaborates the application of GH to a leaf node containing 10 users, whose IDs are ordered according to their Hilbert value. Consider a query from U_7 with K=5. The rank of U_7 is $r(U_7)$=7. The bucket containing U_7 starts at s =7 - 6 *mod* 5 = 6 and ends at e=10, i.e., it contains all users U_6 to U_{10}. Its ASR is the MBR (shaded rectangle at the upper-right corner) covering the corresponding points. Any query with K=5 originating from these users will generate the same b_U, AS and ASR, thus, guaranteeing reciprocity. Note that GH constructs on-the-fly only b_U, as the remaining buckets are irrelevant to the query. Fig. 4.1 illustrates another ASR (shaded rectangle at the lower-left corner) for a query with K=3 originating from one of U_1 to U_3.

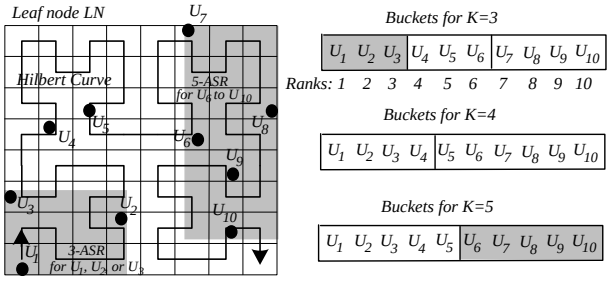


Fig. 4.1 GH partitioning for (leaf) level 1

In case that partitioning takes place above the leaf level, GH could simply load the entire sub-tree of the partition node PN and compute b_U (and its ASR) as above, similarly to *Hilbert Cloak*. However, this process is not necessary since we only need b_U (and not the other buckets at this level). Fig. 4.2 shows an example, where the query issuer U is in leaf node LN_4. The leaves are numbered according to their Hilbert order in the parent PN. I.e, each node is assigned the Hilbert value of the cell that covers its center in the data space defined by the MBR of PN. The cardinality of each leaf node is shown in the corresponding entry of PN.

If K=30, the bucket b_U includes 5 users from LN_3, 10 users from LN_4 and 15 users from LN_5. The nodes that must be accessed are LN_4, PN, and LN_3. Inside LN_3, only the 5 last users in the Hilbert order (in the data space defined by the MBR of LN_3) contribute to b_U, while the rest are assigned to the first bucket (not computed). Note that since the entire LN_5 is included in b_U the node is not visited, but its MBR is simply *merged* to that of the bucket. In some cases the leaf node containing U may fall on the boundary between two buckets. In Fig. 4.2, if K=20, the first 5 users of LN_4 are assigned to the second bucket, and the remaining to the third one. Depending on the position of U in the Hilbert order, either of these two buckets constitutes b_U.

Fig. 4.3 illustrates the general GH method. First, GH computes the extent of the bucket b_U that contains U. Recall that this requires the rank of U in the Hilbert order of N. The function *compute-rank* performs this computation in a recursive manner. Specifically, r_U is the rank of U in LN plus the sum of cardinalities of all nodes that precede the ancestors of U in the path from LN to PN. For instance, if K=30 in Fig. 4.2, then r_U is the rank of U among the points of LN_4 added to the cardinalities of LN_1 to LN_3. Once b_U has been determined, all the leaf nodes that contribute points to b_U participate in the ASR construction through the *merge* function. The merging process is also recursive: If an entry E is totally included within the bucket, it causes the replacement of the ASR with a larger one, whose maximum (minimum) coordinate on each axis is the maximum (minimum) between the corresponding coordinates of E and the original ASR. If E is only partially included, we have to read its contents and repeat this process; there can be at most two such entries per level.

GH involves accessing only (i) the nodes in the path LN to PN (i.e., one node per level) (ii) leaf nodes that are partially (but not totally) included in b_U (i.e., at most two nodes). The first set of nodes is used for the com-

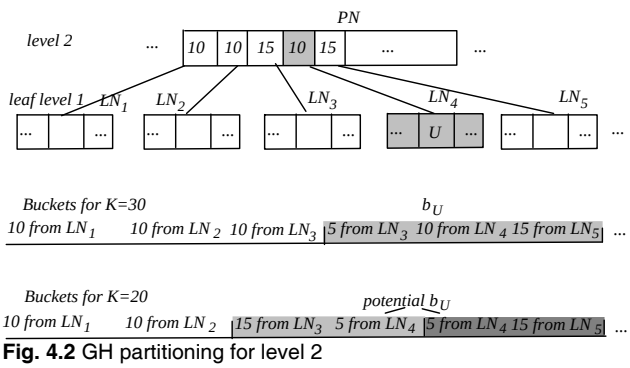


Fig. 4.2 GH partitioning for level 2

GH-partitioning(query issuing user U, anonymity requirement K, partition node PN)
1. ASR =∅
2. r_U = *compute-rank*(PN,0)
3. $s = r_U - (r_U - 1)$ *modulo* K ; $e = s + K - 1$ // *extent of* b_U
4. If $|PN| - e < K$ // b_U *is the last bucket*
5. $e = |PN|$; $s = e - (e \bmod K) - K + 1$
6. For each entry E of PN intersecting $b_U=[s,e]$ //*E is point or node*
7. ASR = *merge*(E,ASR)

compute-rank(N, r_U)
8. *list*=sort entries of N according to their Hilbert value in the data space defined by the MBR of N
9. If N is a leaf node
10. $r_U = r_U$ +position of U in *list*
11. Else // N *is an intermediate node*
12. Let E be the entry that contains U
13. For each entry E' before E in *list*
14. $r_U = r_U + |E'|$
15. r_U =*compute-rank*(E, r_U)
16. Return r_U

merge(E, ASR)
17. If E is totally included in $b_U = [s,e]$
18. For each dimension d
19. $ASR_{d\text{-}min} = min(ASR_{d\text{-}min}, E_{d\text{-}min})$
20. $ASR_{d\text{-}max} = max(ASR_{d\text{-}max}, E_{d\text{-}max})$
21. Else // E *intersects but is not included in* b_U
22. For each entry E' of E that intersects $b_U = [s,e]$
23. ASR = *merge*(E', ASR)

putation of r_U. Other intermediate nodes are not necessary since their contribution to r_U is determined by their cardinalities, which are stored with their parent entries (lines 13-14). Furthermore, leaf nodes that do not intersect b_U are ignored, whereas the MBRs of those totally included in b_U, are directly aggregated in the ASR.

For index structures that impose a minimum occupancy constraint M, such as the R-tree, the PN node is situated at height at most $\lceil \log_M K \rceil$. At each level below the PN node, at most two nodes are accessed, hence the I/O cost is $O(\log_M K)$. The computation complexity of GH includes: (i) sorting of entries according to Hilbert values (line 8) in each accessed node, which takes $O(M \cdot \log_2 M \cdot \log_M K)$, (ii) computation of bucket extent (lines 3-5) which has O(1) cost, and (iii) determining the ASR extent (17-23) with $O(M \cdot \log_M K)$ cost. Therefore, the overall computational complexity is $O(M \cdot \log_2 M \cdot \log_M K)$.

4.2 Asymmetric R-tree Split (AR)

The AR partitioning method is inspired by the R*-tree construction algorithm[5], which is known to have good locality properties. A straightforward approach is to apply the *R*-split* [BKSS90] on the partition node, after setting the minimum node utilization to K. Specifically, *R*-split* first sorts all points by their x-coordinates.

[5] Although AR is inspired by R*-tree, the method can be used on top of any spatial index including the Quad-tree (see experimental evaluation).

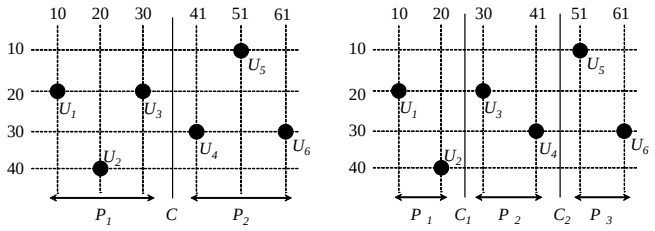


Fig. 4.4 R*-tree split vs AR

Then, it considers every division of the sorted list in two nodes N, N' so that each node contains at least K points, and computes the perimeters of N and N'. The overall perimeter on the x-axis equals the sum of all the perimeters. The process is repeated for the y-axis, and the axis with the minimal overall perimeter becomes the split dimension. Subsequently, *R*-split* examines again all possible divisions on the selected dimension, and selects the one that yields the minimum overlap between the MBRs of the resulting nodes. The split is recursively applied on each partition with more than $2K$ users.

R-split* has some shortcomings with respect to the problem at hand. First it attempts to minimize factors such as perimeter and overlap of the resulting nodes, whereas we aim at minimizing the ASR area. Even if we modify the algorithm to consider only the ASR area, *R*-split* can still lead to *fragmentation*, i.e., a split may create partitions with a large number of redundant users, such that no subsequent splits are possible. As an example, consider that we want to partition the 6 points of Fig. 4.4a into buckets, so that each bucket contains at least K=2 users. The split point that minimizes the sum of resulting areas is x=C, which eliminates the largest gap (i.e., "dead area") between partitions P_1 and P_2. No further split can be performed, since each new node contains 3 users.

To address the problem of fragmentation, AR takes into account both the area and the cardinality of the resulting partitions. Specifically, AR generates partitions P_1 and P_2 that minimize the objective function:

$$[ASR(P_1) + ASR(P_2)] \cdot |P_1| \cdot |P_2|$$

subject to the constraint that $|P_1|$ and $|P_2|$ are at least K. AR favors unbalanced splits, which are desirable, since they achieve low fragmentation. Continuing the example in Fig. 4.4b, any of the split points C_1 or C_2 would yield split cost $(200+620) \cdot 2 \cdot 4 = 6560$, compared to $2 \cdot 400 \cdot 3 \cdot 3 = 7200$ generated by C. Hence, AR would split on either C_1 or C_2, and subsequently allow a second split, resulting in three ASRs, with a total weighted ASR area of $2 \cdot (200+110+200) = 1020$, compared to 2400 for *R*-split*.

Fig. 4.5 shows the pseudocode for AR. Lines 6-15 of *compute-ASR*(U,N) identify the best *split_point* (according to the objective function) for splitting node N by looping over all dimensions and split points in the

```
AR (query issuing user U, anonymity requirement K, partition node PN)
1. Load all points in PN
2. compute-ASR (U, PN)

compute-ASR(U, N)
3.  If |N| < 2K
4.    return MBR(N)
5.  min_split_cost = ∞
6.  For d = 1 to #dimensions // for each dimension
7.    list_d = sort points according to d coordinate
8.    For point = K to |N| - K
9.      P1 = list_d[1 .. point]
10.     P2 = list_d[point +1 .. |list_d| ]
11.   split_cost = (ASR(P1) + ASR(P2))· |P1| · |P2|
12.   If split_cost < min_split_cost
13.         min_split_cost = split_cost;
14.         split_point = point;
15.         split_dim = d;
16. If rank(U) in list_split_dim ≤ split_point
17.   N'=points in list_split_dim[1 .. split_point])
18. Else // U is in the second node of the split
19.   N'=points in list_split_dim[split_point+1.. |list_split_dim| ])
20. Return compute-ASR(U, N')
```

Fig. 4.5 Asymmetric R-tree Split (AR)

range K to $|N|-K$. Let $list_{split\_dim}$ be the list of points sorted on the split dimension. The position of U in $list_{split\_dim}$ determines the partition N' that contains it. If U is before *split_point*, then N' includes all points of $list_{split\_dim}$ in the range [1, *split_point*]. Otherwise, N' includes all points in the range [*split_point*+1, $|list_{split\_dim}|$]. In either case, N' is split recursively. Note that the other partition of N is not split as it is not necessary for the computation of b_U.

Similarly to GH, if an index with minimum node occupancy is used, the *PN* node is situated at height at most $\lceil \log_M K \rceil$. However, this time all nodes under *PN* need to be accessed, with an I/O cost of $O(1+M+M^2+\ldots+M^{\Box})$ where $\Box = \lceil \log_M K \rceil$, which equals to $O(K)$. The computation complexity of AR is a function of K and $|PN|$: at each split of a partition P with more than $2K-1$ points, a sorting phase is employed, with cost $|P| \cdot \log |P|$. In the worst case, each split is unbalanced, and yields two partitions with cardinalities $|P|-K$ and K; the former is split further, until it has less than $2K$ points. The complexity is:

$$\sum_{i=2}^{|PN|/K} (iK)\log(iK) = K\left(\sum_{i=2}^{|PN|/K} i(\log i + \log K)\right) = O\left(\frac{|PN|^2}{K}\log|PN|\right)$$

The proposed partitioning techniques provide different tradeoffs of efficiency and effectiveness. GH, which is very localized, is fast in terms of both I/O and CPU cost but may yield large ASRs. On the other hand, AR is more expensive, since it has to read the entire sub-tree of *PN* and perform CPU-intensive computations, but it yields smaller ASRs. The choice of the partitioning technique depends on the application characteristics. If, for instance, the anonymizer charges clients according to their usage, and the LBS is a public service, it may be preferable to use GH. On the other hand, if the LBS imposes limitations (e.g., on the number of results, processing time, etc) AR is a better choice. In Section 6, we experimentally evaluate these tradeoffs.

5 SKA WITH VARIABLE QUERY FREQUENCIES

So far, we assumed that every user may issue a query with equal probability. However, in practice, the query frequency distribution among users can be skewed. For instance, a taxi driver may issue numerous queries due to the nature of his occupation. In this section we extend the *reciprocal framework* to variable query frequencies. Assuming the worst case scenario, we consider that the attacker knows the query frequencies of all users (e.g., by obtaining billing records).

The definition of SKA is the same as for uniform query frequencies, but the reciprocity property as discussed so far is not sufficient to guarantee SKA. Consider, for instance, AS=$\{U_1,U_2,\ldots,U_K\}$, with user query frequencies $F_1,F_2,\ldots,F_{|K|}$ and that U_1 has twice the query frequency of the other users in AS. Even if AS satisfies reciprocity, based on the knowledge of frequencies, an attacker can pinpoint U_1 as the source with probability $F_1/(F_1+F_2+\ldots+F_K) = 2/(K+1) > 1/K$ for all values of $K>1$. If a query has anonymity degree K, in order to preserve SKA it is necessary that, $F_i/(F_1+F_2+\ldots+F_K) \leq 1/K$, $\forall U_i \in$ AS. Below, we generalize the reciprocity requirement to incorporate information about query frequencies:

***Definition* [*Frequency-Aware Reciprocity* (FQR)].** Consider a user U with query frequency F issuing a query with anonymity degree K, anonymizing set AS = $\{U_1,U_2,\ldots,U_{|AS|}\}$, and anonymizing spatial region ASR. AS satisfies the *frequency-aware reciprocity* (FQR) property if (i) it contains U, (ii) every $U_i \in$ AS generates the same anonymizing set AS for the same value of K and (iii) $\forall U_i \in$ AS, it holds that $F_i/(F_1+F_2+\ldots+F_{|AS|}) \leq 1/K$.

An immediate consequence of condition (iii) is that $K \cdot F_{max} \leq (F_1+F_2+\ldots+F_{|AS|})$, where F_{max} is the maximum query frequency of any user in AS. Note that the reciprocity property discussed in the previous sections is a special case of FQR where all users have equal query frequency.

The *reciprocal framework* can be extended to achieve FQR by incorporating frequency-related information. Assume frequency is represented as the number of queries issued by each user in a previous time interval. For each sub-tree, i.e. internal index node N, we store the sum of frequencies F of users rooted at N, together with the maximum frequency F_{max} in the sub-tree. N can accommodate by itself any query with $K < F/F_{max}$. The algorithm of Fig. 3.1 remains the same, except from line 3, which changes to:

3. While N is not a leaf and (each child of N is empty or $K < F/F_{max}$)

Next, we discuss how GH can be extended to accommodate FQR. Recall that, after the partition node *PN* has been determined, GH sorts the points according to Hilbert values, and creates buckets that contain at least K consecutive points. In the case of *Frequency-Aware GH* (FQGH), each point is conceptually replicated a number of times equal to its query frequency. Hence, each point appears multiple times in the Hilbert sequence, although it is physically stored only once in the index, along with its frequency. The resulting sequence is split into buckets of $K \cdot F_{max}$ each, where F_{max} is the maximum frequency that occurs in *PN*. Fig. 5.1 illustrates an example: each node stores the additional frequency information. At the level 2 *PN* node, the total number of queries in the sub-tree is F=28, whereas F_{max}=7. Assume a query with K=2: the splitting into buckets is performed with respect to $K \cdot F_{max} = 2 \cdot 7 = 14$, and buckets B_1 and B_2 are obtained.

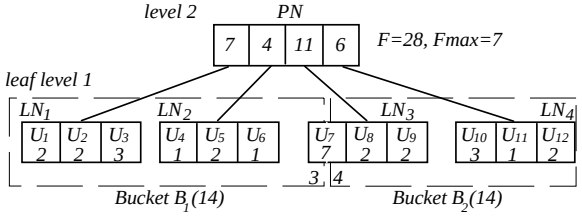


Fig. 5.1 FQGH partitioning, K=2

Since the split is performed with respect to frequencies, it is possible for a user to belong to more than one bucket. However, because the bucket size is at least $K \cdot F_{max}$, it is straightforward to show that a user can belong to at most two buckets. Assume that querying user U contributes with a fraction p of its queries to B_1, and (1-p) to B_2. Then, B_1 will be chosen as ASR with probability p, and B_2 with (1-p). In Fig. 5.1, U_7 contributes with 3/7 of its points to B_1, and 4/7 to B_2; hence, if U_7 issues a query with K=2, the respective generation probabilities for the two buckets are 0.43 and 0.57.

Similar to GH, FQGH only needs to access at most two leaf nodes for each query, therefore it is efficient. Furthermore, the Hilbert sorting is performed based on user locations, and it is oblivious to the query frequencies; hence, the complexity of FQGH is similar to that of GH. AR can be extended to accommodate FQR in a similar manner. However, in practice, query frequency distribution is expected to be skewed, in which case partitioning techniques that require the retrieval of the entire *PN* sub-tree are not practical because a much larger number of users than K are required to achieve SKA. We experimentally verify this claim in the next section.

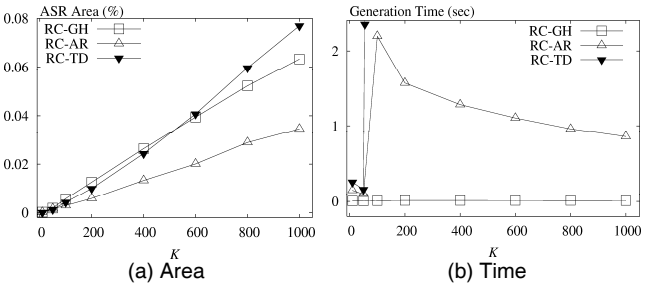


Fig. 6.1 RC: Partitioning methods versus K

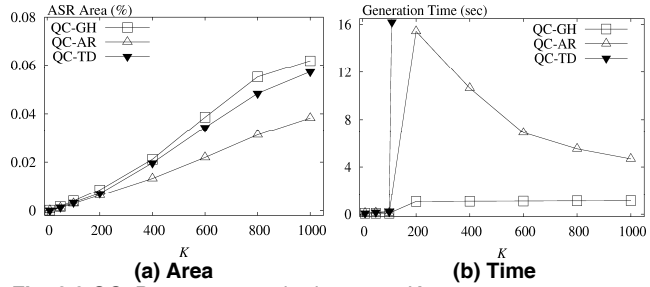


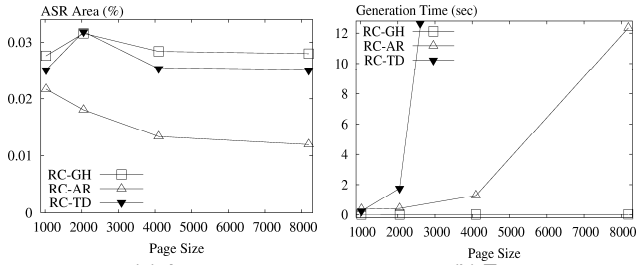
Fig. 6.2 QC: Partitioning methods versus K

6 Experimental Evaluation

We implemented a C++ prototype of the anonymizer and deployed it on an Intel Xeon 2.8GHz machine running Linux OS. The anonymizer indexes the user locations, which are taken from the NA dataset (available at www.rtreeportal.org) containing 569K intersections of the North American road network. K ranges from 10 to 1000. In each experiment, we generate 1000 queries originating at random users. Effectiveness is measured as the average ASR area, expressed as a percentage of the entire data space. Efficiency is measured in terms of average ASR generation time. The average cost per random I/O is 5ms, and each index has a cache equal to 10% of the entire index. For I/O efficiency, we implemented Quad-trees using linear representation [A84], which is easily embeddable into B^+-trees.

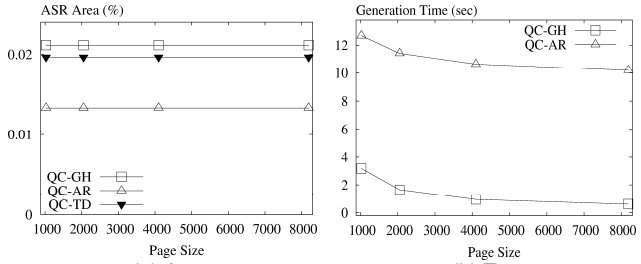
6.1 Evaluation of Partitioning Techniques

First, we consider the RC implementation of *Reciprocal* and compare the proposed partition methods (GH and AR) against a benchmark from the RKA literature. Specifically, we adapt *Top Down* (TD), a divisive clustering-based approach that builds anonymized groups with cardinality bounded between K and $2K$–1 [XWP+06]. The adaptation works as follows. Once the partition node *PN* has been determined, all points of *PN* form one large cluster. TD chooses as seeds two of the most distant points (through an approximate, iterative, linear technique) and divides the cluster among the seeds, so that the extents of the resulting clusters are minimized. The process is repeated recursively for all resulting clusters with cardinality $2K$ or higher. After completion of this step, some clusters (called *runts*) may have fewer



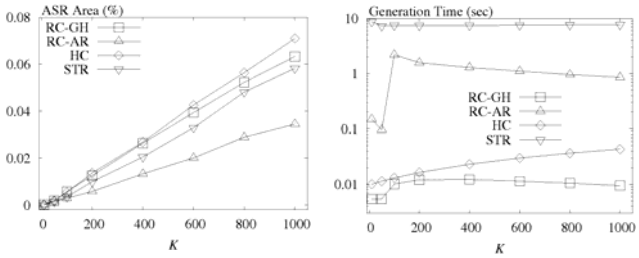
(a) Area (b) Time

Fig. 6.3 RC: Partitioning methods versus page size



(a) Area (b) Time

Fig. 6.4 QC: Partitioning methods versus page size



(a) Area (b) Time

Fig. 6.5 RC-GH and RC-AR versus *HC* and *STR* bulk loading

than K items. To preserve the K-anonymity requirement, a runt may either be merged with another runt, or borrow points from one of the clusters with more than K items. The algorithm terminates when all clusters have at least K items. TD has $O(|PN|^2)$ computation complexity and $O(K)$ I/O cost.

Fig. 6.1 illustrates the ASR area and generation time as a function of K, for 4KB page size. AR has the clear advantage in ASR area, while GH is considerably faster. Note that generation time exhibits a jump after K=80 for all methods except GH. For the 4KB page size, the minimum occupancy of the underlying R*-tree index is 85. Hence for $K \leq 85$, ASRs are generated within one leaf node (at level 1). As K increases beyond this threshold, the ASR is created in a partition node PN at level 2. GH retrieves only a small number of leaf nodes (under PN). On the other hand, AR and TD need to scan the entire sub-tree of PN, leading to significantly more I/Os. Furthermore, the processing time, which is a function of the input size, increases accordingly. For a fixed number of data points under PN, the generation time of AR decreases with larger K because the number of splits drops (i.e., there are fewer, larger buckets). TD is expensive for partitioning at level 2 (in some cases up to 100 sec per query) and omitted for K >80.

Fig. 6.2 repeats the same experiment for the Quad-tree (QC) implementation of *Reciprocal*. While the ASR area is similar to RC, the generation time is considerably higher for QC due to the lack of balance in the index structure, resulting in a large number of points under the PN node.

In Fig. 6.3 we vary the page size, and measure the ASR area and generation time for RC, when K = 400. As the page size increases, ASRs need to span across fewer leaf nodes. Therefore, we expect the effectiveness to improve, as the good locality properties of the underlying R*-tree index are better exploited. For page sizes from 2 to 8KB, this is indeed the case. However, initially GH exhibits an increasing trend because, for 1KB page size, the K = 400 setting coincides with the minimum occupancy at level 2. Hence, a point of convergence occurs, which helps GH to obtain smaller ASRs. A larger page size also translates into increased generation time, as the cardinality of the partition node increases. TD is very expensive for sizes exceeding 2KB (for 8 KB page size, it needs 400sec per query). The cost of AR grows due to the recursive splits. GH is rather insensitive to the page size since it computes a single bucket, independently of node cardinality.

Fig. 6.4 shows the same experiment for QC. Observe that the page size does not affect the ASR area, which only depends on the Quad-tree hierarchy. On the other hand, a larger page increases the occupancy of leaf nodes, and reduces the I/O cost, as shown in Fig. 6.4b (TD is omitted due to very high values).

Summarizing, GH is the most efficient partitioning method, whereas AR is the most effective. The performance of TD is unsatisfactory, as it is extremely expensive and produces ASRs with quality comparable to GH. Regarding the R-tree and Quad-tree implementations, they offer similar ASR areas, but RC is faster. Based on the above, RC-GH is the method of choice for efficiency (e.g., when the anonymizer charges clients according to their usage and the LBS is a public service) and RC-AR the winner when effectiveness is more important (e.g., free anonymizer service and expensive LBS).

6.2 Comparison with Hilbert Cloak (HC) and R-Tree Bulk Loading

We compare the RC-GH and RC-AR methods against two baseline solutions: *Hilbert Cloak* (HC) [KGMP07], and *Sort-Tile-Recursive* (STR) [LEL97] – a state-of-the-art bulk-loading method for R-trees. The STR baseline relies on the same principle as [IN07], i.e., bulk-loading a spatial index with an occupancy constraint of at least K users per leaf node. However, [IN07] employs an R+-tree index that does not allow overlaps between node

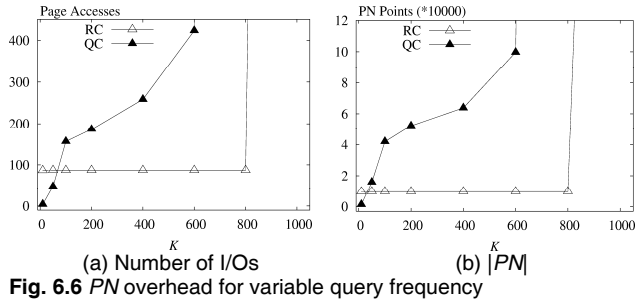


Fig. 6.6 *PN* overhead for variable query frequency

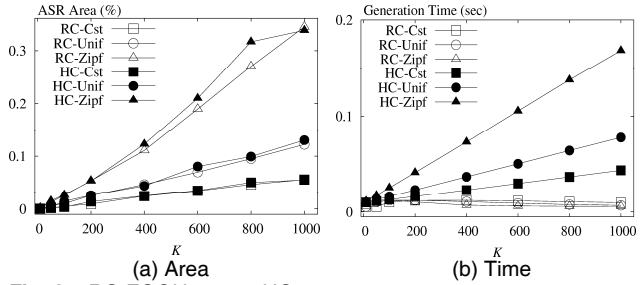


Fig. 6.7 RC-FQGH versus HC_f

extents. In contrast, SKA does allow ASR extents to overlap, therefore we use STR in conjunction with R*-trees[6] that obtain lower extents for leaf nodes than their R+-tree counterparts.

Fig. 6.5 shows the relative performance of RC-GH, RC-AR, HC and STR. In terms of ASR area, STR is slightly better than RC-GH, but considerably worse than RC-AR. Furthermore, since STR performs partitioning with respect to the entire dataset, it incurs high overhead. As shown by the log-scale graph in Fig. 6.5b, the time required by STR is close to 10 seconds, up to one order of magnitude worse than RC-AR. Therefore, STR is not suitable for on-line queries where users require short response times. On the other hand, RC-AR performs a high-quality partitioning of users with respect to a single index sub-tree within a short time.

RC-GH is slightly better than HC in terms of ASR size and up to one order of magnitude faster. Although HC applies a Hilbert sorting method similar to RC-GH and does not incur the overhead of finding the *PN* node, it still needs to retrieve from the disk O(*K*) leaf entries. In contrast, RC-GH, which maintains MBR information in the internal nodes, only needs to access two leaf nodes per query. Note that the RC-GH generation time exhibits an initial increase with increasing *K*, as the *PN* node moves from the leaf level to level 2. RC-AR generates significantly smaller ASRs, but it is much slower than both RC-GH and HC.

6.3 Variable Query Frequencies

As discussed in Section 5.2, local partitioning methods that require loading the entire *PN* node (e.g., AR, TD) are not I/O and CPU efficient, when the query frequency distribution is skewed. We support our claim with an experiment which measures the I/O cost to retrieve the *PN* node, and the number of points included in *PN*. We generated 1000 queries, each assigned to a user according to the zipf distribution with parameter 0.8. Page size is 4 KB. The results are shown in Fig. 6.6. Due to its unbalanced structure, QC incurs higher I/O cost than RC, and it requires retrieving the entire dataset for values of $K > 600$. Although RC incurs less I/O, for $K > 800$, *PN* corresponds to the root node of the index; therefore, all points need to be retrieved. Consequently, AR and TD are impractical for skewed frequency distribution.

Finally, we evaluate RC-FQGH, which is feasible for skewed query distributions because it does not retrieve the entire *PN* sub-tree. For comparison, we use a frequency-aware variant of HC (called HC_f), which is similar to RC- FQGH, except that partitioning is applied to the entire user set, as opposed to the *PN* node. We consider 1000 random queries with constant (*Cst*), uniform (*Unif*) and zipf-0.8 distribution (*Zipf*). Fig. 6.7 shows that guaranteeing privacy for variable query frequency comes at an additional increase in ASR size, which grows with the skewness of the frequency distribution. RC-FQGH is slightly better in terms of ASR area, but the advantage of the reciprocal framework is clear in terms of generation time, where RC-FQGH is much faster than HC_f for all query distributions.

7 Conclusions

In this paper we proposed a *reciprocal framework* that allows the implementation of a variety of secure algorithms for spatial *K*-anonymity on top of a spatial index. We also extended the framework to support users with variable query frequencies. We demonstrated the versatility of our framework by using it to implement a variety of partitioning techniques on top of two popular spatial indices. Finally, we showed experimentally that our methods outperform the only existing secure technique.

In the future, we plan to address more attack scenarios, such as attacks based on user preferences. Assume that each user is interested in certain types of queries, e.g., traffic conditions, restaurants, etc. An attacker may use the additional knowledge to infer the query source. To prevent this, users can be classified into groups according to their interests. Then, spatial diversity would take into account these groups when forming ASRs; i.e., an ASR should contain users with *similar* interests, from the same group. Another interesting problem concerns *continuous* SKA [CM07]. In this setting, a client poses a long running query about its surroundings (e.g., "find the nearest gas station"), whose results are updated as

[6] We used in our STR implementation the *Spatial Index Library*, available online at http://research.att.com/~marioh/spatialindex

the client moves. The cloaking algorithm should generate a continuously changing ASR in a way that it does not reveal information about the user through inspection of the individual ASR snapshots.

ACKNOWLEDGEMENTS

This work was partially supported by grant HKUST 618406 from Hong Kong RGC.

REFERENCES

[A84] Abel, D. J. A B+ tree structure for large quadtrees. Computer Vision, Graphics, and Image Processing 27(1): 19-31, 1984.

[AFK+06] Aggarwal G., Feder T., Kenthapadi K., Khuller S., Panigrahy R., Thomas D., Zhu A.. Achieving Anonymity via Clustering. *PODS*, 2006.

[B71] Butz A.R.. Alternative Algorithm for Hilbert's Space-Filling Curve. *IEEE Trans. on Computers*, April ,1971.

[BA05] Bayardo R.J., Agrawal R. Data Privacy through Optimal k-Anonymization. *ICDE*, 2005.

[BKSS90] Beckmann N., Kriegel H.P., Schneider R., Seeger B. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[BS03] Beresford A.R., Stajano F. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46-55, 2003.

[BWJ05] Bettini C., Wang S., Jagodia S. Protecting Privacy Against Location-based Personal Identification. *VLDB Workshop on Secure Data Management*, 2005.

[CZBP06] Cheng R., Zhang Y., Bertino E., Prabhakar S., Preserving User Location Privacy in Mobile Data Management Infrastructures. *Privacy Enhancing Technology Workshop*, 2006.

[CM07] Chow C.Y., Mokbel M. Enabling Private Continuous Queries for Revealed User Locations. *SSTD* ,2007.

[CML06] Chow C., Mokbel M., Liu X. A Peer-to-Peer Spatial Cloaking Algorithm for Anonymous Location-based Services. *ACM GIS*, 2006.

[GL05] Gedik B., Liu L. Location Privacy in Mobile Systems: A Personalized Anonymization Model. *ICDCS*, 2005.

[GKS07] Ghinita G., Kalnis P., Skiadopoulos S. PRIVÈ: Anonymous Location-based Queries in Distributed Mobile Systems. *WWW*, 2007.

[GKKM07] Ghinita G., Karras P., Kalnis P., Mamoulis N, Fast Data Anonymization with Low Information Loss. *VLDB*, 2007.

[GKK+08] Ghinita G., Kalnis P., Khosgozaran A., Shahabi C., Tan K.-L. Private queries in location-based services: anonymizers are not necessary. *SIGMOD*, 2008.

[GG03] Gruteser M., Grunwald D. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. *USENIX MobiSys*, 2003.

[HG05] Hoh B., Gruteser M. Protecting Location Privacy through Path Confusion. *SecureComm*, 2005.

[HL05] Hu H., Lee D.L. Range Nearest-Neighbor Query. *IEEE TKDE*, 18(1):78-91, 2006.

[IN07] Iwuchukwu, T., Jeffrey F. Naughton, J. K-Anonymization as Spatial Indexing: Toward Scalable and Incremental Anonymization. *VLDB*, 2007.

[KGMP07] Kalnis P., Ghinita G., Mouratidis K., Papadias D. Preventing Location-Based Identity Inference in Anonymous Spatial Queries. *IEEE TKDE*, to appear.

[KS07] Khoshgozaran A., Shahabi C. Blind Evaluation of Nearest Neighbor Queries Using Space Transformation to Preserve Location Privacy. *SSTD* ,2007.

[KZTO05] Kamat P., Zhang Y., Trappe W., Ozturk C. Enhancing Source-Location Privacy in Sensor Network Routing. *ICDCS*, 2005.

[LDR05] LeFevre K., DeWitt D.J., Ramakrishnan R. Incognito: Efficient Full-domain K-Anonymity. *SIGMOD*, 2005.

[LDR06] LeFevre K., DeWitt D.J., Ramakrishnan R. Mondrian Multidimensional *K*-Anonymity. *ICDE*, 2006.

[LEL97] Leutenegger S.T., Edgington J.M., Lopez M.A. STR: A simple and efficient algorithm for R-tree packing. Institute for Computer Applications in Science and Engineering (ICASE), TR-97-14, 1997.

[LTV07] Li N., Li T., Venkatasubramanian S. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. *ICDE*, 2007.

[MGKV06] Machanavajjhala A., Gehrke J., Kifer D., Venkitasubramaniam M. *l*-Diversity: Privacy Beyond *K*-Anonymity. *ICDE*, 2006.

[MCA06] Mokbel M.F, Chow C.Y, Aref W.G. The New Casper: Query Processing for Location Services without Compromising Privacy. *VLDB*, 2006.

[S01] Samarati P. Protecting Respondents' Identities in Microdata Release. *IEEE TKDE*, 13(6):1010-1027, 2001.

[S90] Samet H. The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1990.

[TP05] Tao Y., D. Papadias. Historical Spatio-Temporal Aggregation. *ACM TOIS*, 23(1), 61-102, 2005.

[XT06] Xiao X., Tao Y. Anatomy: Simple and Effective Privacy Preservation. *VLDB*, 2006.

[XWP+06] Xu J., Wang W., Pei J., Wang X., Shi B., Fu A., Utility-Based Anonymization Using Local Recoding. *SIGKDD*, 2006.

[ZKSY07] Zhang Q., Koudas N., Srivastava D., Yu T. Aggregate Query Answering on Anonymized Tables. *ICDE*, 2007.

APPENDIX A

Among the systems reviewed in Section 2.2, *Casper* and *Interval Cloak* perform spatial cloaking, using the same architecture and following the same assumptions as our techniques. Next, we show formally that both approaches are not secure. Recall that the shape of an ASR in *Casper* can be either a square, or the horizontal/vertical union of two adjacent cells under the same parent. We first analyze the case of square ASRs assuming that an attacker detects the ASR of Fig. A.1a. Then, s/he can infer that it was created due to a query from a user U in A, B, C, D. If U is in cell A, the required degree of anonymity K_A must be in the range $[M_A+1, |A|+|B|+|C|+|D|]$. $M_A = |A| + max\{|B|, |C|\}$ is due to the fact that neither $A \cup B$, nor $A \cup C$ contains sufficient points (otherwise the ASR would be $A \cup B$, or $A \cup C$). Similar to K_A, we can calculate the ranges of K_B, K_C and K_D which have the same maximum value $|A|+|B|+|C|+|D|$, but different lower bounds $M_B = |B| + max\{|A|, |D|\}$, $M_C = |C| + max\{|A|, |D|\}$ and $M_D = |D| + max\{|B|, |C|\}$, respectively.

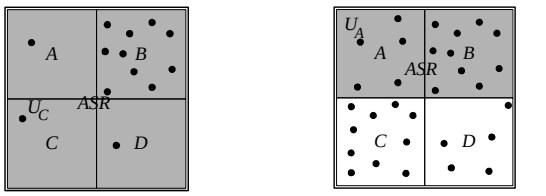
Summarizing, the ASR is generated by a query originating from (i) A with anonymity K_A, i.e., $|A|\cdot(|A|+|B|+|C|+|D|-M_A)$ events, or (ii) B with K_B, i.e., $|B|\cdot(|A|+|B|+|C|+|D|-M_B)$ events, or (iii) C with K_C, i.e., $|C|\cdot(|A|+|B|+|C|+|D|-M_C)$ events, or (iv) D with K_D, i.e., $|D|\cdot(|A|+|B|+|C|+|D|-M_D)$ events. The total number of events is $(|A|+|B|+|C|+|D|)^2-|A|\cdot M_A-|B|\cdot M_B-|C|\cdot M_C-|D|\cdot M_D$. Given no additional knowledge about the query frequency and the anonymity degree distributions, the attacker considers that these events have equal probabilities, e.g., s/he assumes that the query originates from A with probability:

$$P_A=\frac{|A|\cdot\left(|A|+|B|+|C|+|D|-M_A\right)}{\left(|A|+|B|+|C|+|D|\right)^2-|A|\cdot M_A-|B|\cdot M_B-|C|\cdot M_C-|D|\cdot M_D}$$

Within A, each individual user can issue the query with equal probability $P_A/|A|$. For SKA to be preserved, it must hold that $P_A/|A| \le 1/K_A$. Since the maximum value of K_A is $|A|+|B|+|C|+|D|$, we have $P_A/|A| \le 1/(|A|+|B|+|C|+|D|)$. Applying the same reasoning to $P_B/|B|$, $P_C/|C|$ and $P_D/|D|$ and some algebraic simplifications, we derive the following system of linear inequalities:

$$\begin{cases} M_A \ge \dfrac{|B|\cdot M_B+|C|\cdot M_C+|D|\cdot M_D}{|B|+|C|+|D|} \\ M_B \ge \dfrac{|A|\cdot M_A+|C|\cdot M_C+|D|\cdot M_D}{|A|+|C|+|D|} \\ M_C \ge \dfrac{|A|\cdot M_A+|B|\cdot M_B+|D|\cdot M_D}{|A|+|B|+|D|} \\ M_D \ge \dfrac{|A|\cdot M_A+|B|\cdot M_B+|C|\cdot M_C}{|A|+|B|+|C|} \end{cases}$$

The solution to the above system has the only form $M_A = M_B = M_C = M_D$. $M_A=M_D$ implies that $|A|=|D|$, and $M_B=M_C$ that $|B|=|C|$. In other words, each pair of diagonal cells should have the same cardinality; otherwise *Casper* fails to preserve SKA. As an example consider Fig. A.1a, where A, C and D contain one user each, and B includes 10 users ($M_A=M_B=M_D=11$, $M_C=2$). Assuming that the query originates from U_C in cell C, then K_C must be in the range [3, 13]. The attacker will infer U_C as the origin with probability $P_C/|C|=11/35$, which exceeds $1/K_C$ for $4 \le K_C$. Thus, the anonymity of U_C is breached for all, but one, queries involving this ASR.



(a) Square ASR (b) 2x1 Rectangular ASR

Fig. A.1 Examples of *Casper* ASRs

Having established that diagonal neighbors must have the same cardinality (in order not to compromise square ASRs), we will show that the horizontal and vertical neighbors must also satisfy the same condition. Assume a rectangular ASR consisting of cells A and B (see Fig. A.1b). Clearly, the query may have originated from a user U in A or B. If U is in A, the required degree of anonymity K_A must be in the range $[|A|+1, |A|+|B|]$. This is because if $K_A \le |A|$, the ASR would not include B (as the points in A would suffice). Otherwise, if $K_A > |A|+|B|$, the ASR should be larger than the union of A and B. Similarly, if the query is issued by any user from B, the degree of anonymity K_B is in the range $[|B|+1, |A|+|B|]$. By applying the previous methodology, we conlude that $|A|$, $|B|$, $|C|$, $|D|$ must all be equal to guarantee anonymity.

Therefore, *Casper* achieves SKA only when each cell (at any level) contains exactly the same number of users as its neighbors, i.e., *only for perfectly uniform user distribution*. The analysis also applies to the simpler case of *Interval Cloak*.