PRIVACY-PRESERVING QUERY TRANSFORMATION AND PROCESSING IN LOCATION BASED SERVICES

GABRIEL GHINITA

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2008

Abstract

The increasing trend of embedding positioning capabilities (e.g., GPS) in mobile devices has created unprecedented opportunities for the widespread use of Location Based Services (LBS). Mobile users are able to formulate spatial queries, such as "find the closest restaurant to my current position". For such applications to succeed, privacy and confidentiality are essential. Commonly, privacy-enhancing techniques rely on encryption to safeguard communication channels, and on pseudonyms to protect user identities. Nevertheless, an LBS query contains the current location of the user, which may be mapped to the user's identity through a variety of means, such as signal triangulation, or physical observation. Hiding the user location is a challenging task, and a primordial requirement for LBS privacy.

This thesis presents a framework for private queries in location-based services. First, we study in depth the location privacy problem in the context of *spatial K-anonymity (SKA)*, an extension of the *K-anonymity* paradigm, widely used for privacy preservation in relational databases. To enforce SKA, we adopt a three-tier architecture, with an *Anonymizer Service (AS)* that acts as an intermediary between the users and the LBS, and anonymizes queries by cloaking user locations. We identify the *reciprocity* property, a sufficient condition to guarantee privacy for a snapshot of user locations, and develop two SKA algorithms which provide a trade-off between privacy requirements and query processing overhead. We also devise algorithms to process range and nearest-neighbor anonymized queries at the LBS side.

Next, we extend our results by showing how reciprocity can be effectively and efficiently enforced using hierarchical spatial indices, such as Quad-trees and R-trees. We also develop a stronger version of reciprocity - *frequency-aware* reciprocity - which addresses the scenario when an attacker possesses additional background knowledge about the relative frequencies of issuing queries among distinct users.

Most existing work in LBS query privacy assumes a centralized AS, which must handle the frequent updates of user locations, as well as the overhead

of anonymizing queries. Furthermore, the AS is a single-point-of-attack, and, if compromised, the privacy of all users is threatened. We address these limitations by devising a decentralized architecture for LBS anonymization: users organize themselves into a P2P network, and cooperate to anonymize queries. We propose two such P2P systems, which provide a trade-off between privacy requirements and scalability.

Finally, we take a step further from the SKA paradigm, and propose a novel LBS privacy approach, based on *Private Information Retrieval (PIR)*. PIR comprises of a two-party cryptography-based protocol that allows a client to retrieve the desired information from a server, without the server learning what information was requested. We show that PIR eliminates the need to trust a third-party anonymizer, as well as other users. Furthermore, since location information is encrypted (not just cloaked, as in the case of spatial K-anonymity), this method is resilient to any type of location-based attack. For instance, PIR-based privacy protects against correlation attacks in the case of private continuous queries (i.e., a user asks the same query from different locations at consecutive timestamps), a problem which has not been efficiently solved yet within the SKA paradigm. The PIR approach provides superior privacy, and incurs a reasonable overhead in practice.

Acknowledgments

I would like to thank my supervisor, Dr. Panos Kalnis, for his guidance and support throughout my Ph.D. studies. I would also like to thank the members of my examination committee for their interest and time spent on this PhD dissertation: Dr. Li Mong Lee and Dr. Chee Yong Chan from National University of Singapore, and Dr. George Kollios (external reviewer) from Boston University.

I am also grateful for their support and advice, as well as the numerous interesting research discussions, which represented the source of valuable ideas, to: Dr. Dimitris Papadias (Hong Kong University of Science and Technology), Dr. Nikos Mamoulis (Hong Kong University), Dr. Kian-Lee Tan (National University of Singapore), Dr. Yufei Tao (Chinese University of Hong Kong), Dr. Cyrus Shahabi (University of Southern California), Dr. Kyriakos Mouratidis (Singapore Management University), Dr. Panagiotis Karras (University of Zurich), Dr. Spiros Skiadopoulos (University of Peloponnese), Dr. Man Lung Yiu (Aalborg University) and Mr. Xiaokui Xiao (Chinese University of Hong Kong).

Contents

1 Introduction **1**

1.1 Contributions and Thesis Organization 5

2 Related Work **10**

2.1 K-anonymity 10

2.2 Spatial K-anonymity. Assumptions and Goals 12

2.3 Existing SKA Techniques 16

2.4 Related Spatial Query Processing Techniques 21

2.5 Related P2P Systems 23

2.6 Private Information Retrieval 24

3 SKA Framework for LBS Privacy **26**

3.1 Introduction 26

3.2 Nearest Neighbor Cloak 27

3.3 Reciprocity 28

3.4 Hilbert Cloak 29

3.5 Location-Based Service Query Processing 32

3.5.1 $\mathcal{C}k$NN - Circular Range kNN 32

3.5.2 R-trees and $\mathcal{C}k$NN 35

3.6 Experimental Evaluation 40

3.6.1 Anonymizer Evaluation 40

3.6.2 Location-Based Service Evaluation 44

3.7 Discussion 51

4 Reciprocal Framework for SKA **52**

4.1 Introduction 52

4.2 Algorithm for Reciprocal Cloaking 52
4.3 Partitioning Methods 57
4.3.1 Greedy Hilbert Partitioning (GH) 57
4.3.2 Asymmetric R-tree Split (AR) 62
4.3.3 Dynamic Programming Hilbert (DH) 64
4.3.4 Top-Down Clustering (TD) 66
4.3.5 Discussion 66
4.4 SKA With Variable Query Frequencies 67
4.5 Experimental Evaluation 70
4.5.1 Evaluation of Partitioning Techniques 70
4.5.2 Comparison with Hilbert Cloak (HC) 76
4.5.3 Variable Query Frequencies 77
4.6 Discussion 79

5 Decentralized Query Anonymization 80

5.1 Introduction 80
5.2 PRIVÉ 81
5.2.1 *Hilbert Cloak* with a B^+-tree index 83
5.2.2 Protocol Overview 84
5.2.3 Protocol Operations 86
5.2.4 Fault Tolerance and Load Balancing 89
5.3 MOBIHIDE 92
5.3.1 The Correlation Attack 94
5.3.2 Protocol Overview 95
5.3.3 Protocol Operations 97
5.3.4 Fault-tolerance and Load Balancing 99
5.4 Experimental Evaluation 102
5.4.1 PRIVÉ protocol 103
5.4.2 MOBIHIDE protocol 111
5.4.3 PRIVÉ and MOBIHIDE Comparison 114
5.5 Discussion 119

6 PIR Framework for LBS 120

6.1 Introduction 120

6.2 Computational PIR Protocol 121
6.3 PIR and Location-dependent Queries 124
6.4 Approximate Nearest Neighbors 125
6.4.1 Approximate NN using Hilbert ordering 125
6.4.2 Generalization to 2-D partitionings 128
6.5 Exact Nearest Neighbors 129
6.5.1 Grid Granularity 132
6.6 Optimizations 133
6.6.1 Compression 133
6.6.2 Rectangular vs. Square PIR Matrix 133
6.6.3 Avoiding Redundant Multiplications 135
6.6.4 Parallelism 138
6.7 Experimental Evaluation 138
6.7.1 1D and 2D Approximate NN 139
6.7.2 Exact Methods 141
6.7.3 Execution Time Optimizations 143
6.7.4 User CPU Time 144
6.7.5 PIR vs. Anonymizer-based Methods 144
6.8 Discussion 146

7 Conclusions and Future Work 148
7.1 Summary of Contributions 148
7.2 Directions for Future Research 150

A Analysis of Privacy in *Casper* and *Interval Cloak* 159

List of Tables

5.1 Privé Protocol Terminology 86

6.1 Summary of notations 121
6.2 Grid Granularity for ExactNN 141

List of Figures

1.1 Hiding identity with pseudonyms is not sufficient 2
1.2 Example: "Find the nearest hospital". 3
1.3 Framework for Spatial K-anonymity (SKA) 4
1.4 PIR framework 7
1.5 Thesis Roadmap 9

2.1 Distance from MBR center for *Center Cloak* (K=10) 15
2.2 Example of *Interval Cloak* and *Casper* 17
2.3 Location anonymity compromise in the presence of outliers . 19
2.4 Example of *Clique Cloak* 19
2.5 Example of continuous NN search 22

3.1 Example of *NNC* 27
3.2 K-ASR Reciprocity Example, K=5 28
3.3 Hilbert Curve (left: 4×4, right: 8×8) 30
3.4 Example of *Hilbert Cloak* 31
3.5 The 1-NNs of $\mathcal{C}$ are p_1 and p_2 33
3.6 $\mathcal{C}k$NN example: perpendicular bisector does not intersect $\mathcal{C}$. 34
3.7 The perpendicular bisector intersects $\mathcal{C}$ 35
3.8 Find the 1-NNs of a circular range $\mathcal{C}$ 36
3.9 Check if E may contain qualifying objects 37
3.10 The MBR and the MER of $\mathcal{C}$ 38
3.11 North-America (NA) dataset 40
3.12 Area of rectangular K-ASR 41
3.13 K-ASR generation time 42
3.14 Rectangular vs SA K-ASR, *Nearest Neighbor Cloak* 43

3.15 *center-of-ASR* attack, K= 50 44
3.16 kNN queries, varying k, $N = 50,000$, $K = 80$ 45
3.17 kNN queries, varying K, $k = 2$ neighbors, $N = 50,000$ 46
3.18 kNN queries, varying N, $k = 2$, $K = 80$ 47
3.19 Range queries, $N = 50,000$, varying K 48
3.20 *NNC*, rectangular vs SA K-ASR, $k = 2$, $N = 50,000$ 49
3.21 *NNC*, rectangular vs SA K-ASR, $k = 2$, $K = 80$ 50

4.1 Reciprocal Cloaking 53
4.2 Partitioning with a Quad-tree 55
4.3 GH partitioning for (leaf) level 1 58
4.4 GH partitioning for level 2 59
4.5 Greedy Hilbert - general method 61
4.6 R*-tree split vs AR 63
4.7 Asymmetric R-tree Split (AR) 64
4.8 GH and DH partitions for K=4 65
4.9 Reciprocal Cloaking Change for Variable Frequency 68
4.10 FQGH partitioning, K=2 69
4.11 R-tree Cloak (RC). Partitioning methods versus K 71
4.12 Quad-tree Cloak (QC). Partitioning methods versus K 72
4.13 RC versus page size 73
4.14 QC versus page size 74
4.15 RC-GH and RC-AR versus HC 76
4.16 PN overhead for variable query frequency 77
4.17 RC-FQGH versus HC_f 78

5.1 Architecture of Privé 82
5.2 *Hilbert Cloak* with Annotated B^+-tree 84
5.3 Distributed Index Structure, α=2 85
5.4 User Join and Relocation, α=2 87
5.5 User Relocation Pseudocode 88
5.6 K-request, α=2, K=6 89
5.7 K-request 90
5.8 Load Balancing Mechanism 91

5.9 Hilbert sequence ring 92
5.10 K-ASR construction in MobiHide 93
5.11 MobiHide implementation over Chord 96
5.12 Join and Split, α=2 98
5.13 Pseudocode for K-Request 99
5.14 Leader Election Protocol 100
5.15 Dataset 102
5.16 Privé Join/Leave Operation 103
5.17 Privé K-request Operation 104
5.18 Privé K-request Operation 105
5.19 Privé Percentage of users involved in query 106
5.20 Privé Relocation 107
5.21 Privé Relocation Level 108
5.22 Privé Failure Recovery 108
5.23 Privé Load Balancing 109
5.24 MobiHide Join 111
5.25 MobiHide K-Request Operation 112
5.26 MobiHide Load Balancing 113
5.27 MobiHide Fault Tolerance 114
5.28 Anonymity Strength 116
5.29 K-ASR Area 117
5.30 Scalability, $K = 40$ 118

6.1 PIR example. u requests X_{10} 123
6.2 9 POIs on a 8×8 Hilbert curve 126
6.3 Approximate NN using Hilbert 127
6.4 Protocol for approximate NN 127
6.5 2-D approximate NN 129
6.6 Exact nearest neighbor 130
6.7 Protocol for exact NN 131
6.8 Finding the optimal grid granularity 132
6.9 Rectangular PIR matrix M 134
6.10 Pre-compiled optimized execution plan 135
6.11 Execution plan for one row 136

6.12 PIR Optimizer Architecture 138
6.13 Variable k, Sequoia set (62K POI) 140
6.14 Variable data size, $k = 768$ bits 140
6.15 Approximation Error 141
6.16 Variable k, Sequoia set (62K POI) 142
6.17 Variable data size, $k = 768$ bits 142
6.18 DM Optimization, Sequoia set 143
6.19 Parallel execution, Sequoia set 144
6.20 User CPU time 145
6.21 PIR vs. K-anonymity, Sequoia set 145

A.1 Examples of Casper ASRs 161

Chapter 1

Introduction

In recent years, mobile devices with positioning capabilities (e.g., GPS) have gained tremendous popularity. Navigation systems are already widespread in the automobile industry and, together with wireless communications, facilitate exciting new applications. General Motor's OnStar system, for example, supports on-line rerouting to avoid traffic jams and automatically alerts the authorities in case of an accident. More applications based on the users' location are expected to emerge with the arrival of the latest gadgets (e.g., iPAQ hw6515, Mio A701), which combine the functionality of a mobile phone, PDA and GPS receiver. For such applications to succeed, the privacy and confidentiality issues are of paramount importance.

Consider the example in Figure 1.1: Bob uses his GPS-enabled mobile phone to find the nearest betting office. This query can be answered by a Location Based Service (LBS) in a publicly available web server (e.g., Google Maps). Since Bob does not want to disclose to Eve (an eavesdropper) his gambling habits, instead of directly sending the query to the LBS, he uses a *pseudonym*[1] service, which is a trusted server (services for anonymous web surfing are commonly available nowadays). He establishes a secure connection (e.g., SSL) with the pseudonym service, which removes the user id and forwards the query to the LBS. The answer from the LBS is also routed to Bob through the pseudonym service.

Nevertheless, the query itself unintentionally reveals sensitive informa-

[1] http://www.torproject.org/

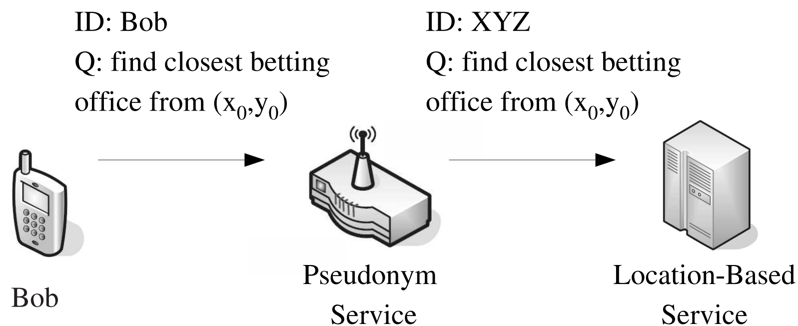


Figure 1.1: Hiding identity with pseudonyms is not sufficient

tion. In our example, the LBS requires the coordinates of the user in order to process the nearest neighbor (NN) query. Since the LBS is not trusted, Eve can collaborate with the LBS and acquire the location of Bob and his query result (i.e., betting office). The next step is to relate the coordinates to a specific user. Eve may choose from a variety of techniques such as physical observation of Bob, triangulating his mobile phone's signal[2], or consulting publicly available databases. If, for instance, Bob uses his phone within his residence, Eve can easily convert the coordinates to a street address (most on-line maps provide this service) and relate the address to Bob by accessing an on-line white pages service.

A broad discussion on the risks of revealing sensitive information in location-based services can be found in [16]. In practice, users would be reluctant to access a service that may disclose their political/religious affiliations or alternative lifestyles. Furthermore, given that the LBS is not trusted, users might be hesitant to ask innocuous queries such as "find the closest gas station" or "which are the restaurants in my vicinity" since, once their identity is revealed, they may face unsolicited advertisements, e-coupons, etc.

To address these privacy threats, most existing solutions rely on the K-anonymity [53, 58] paradigm, which has been used for publishing census data and hospital records. A dataset is said to be K*-anonymized*, if each

[2]Phone companies can estimate the location of the user within 50-300 meters, as required by the US authorities (E911).

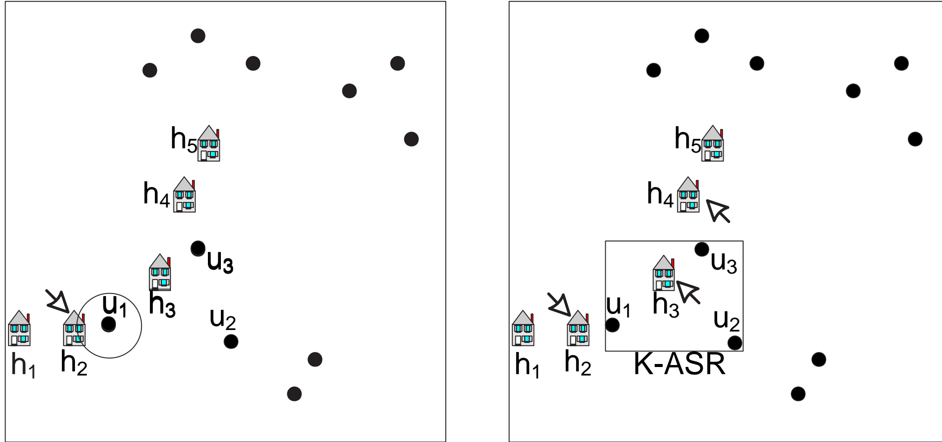


Figure 1.2: Example: "Find the nearest hospital".

record is indistinguishable from at least $K-1$ other records with respect to certain identifying attributes. In location based services, the corresponding *Spatial K-anonymity (SKA)* concept translates as follows: given a query, guarantee that an attack based on the query location cannot identify the query source with probability larger than $1/K$, among other $K-1$ users.

Typically, users ask Range or Nearest-Neighbor (NN) queries with respect to their location. For example, user u_1 in Figure 1.2(left) (users are shown as black dots), may ask: "Find the nearest hospital to my present location" (the answer is h_2). In order not to reveal his exact location, u_1 employs the use of an *Anonymizer Service* (*AS*), which hides user locations. Commonly, the three-tier architecture of Figure 1.3 is employed, where the AS acts as an intermediate tier between the users and the LBS. Users send their locations and queries to the centralized AS, through a secure connection. In our case, u_1 sends to AS the query content (i.e. "find the closest hospital"), and the required degree of anonymity K (note that, K is based on individual privacy criteria, and may vary among queries). For each received query, the anonymizer removes the id of the user, and constructs an *Anonymizing Spatial Region* (*ASR* or *K-ASR*), which is an area that encloses the query source, as well as at least $K-1$ other users. Continuing the running example in Figure 1.2(right), upon receiving the query request from u_1, the AS identifies a set of additional two users (i.e., u_2 and u_3) and

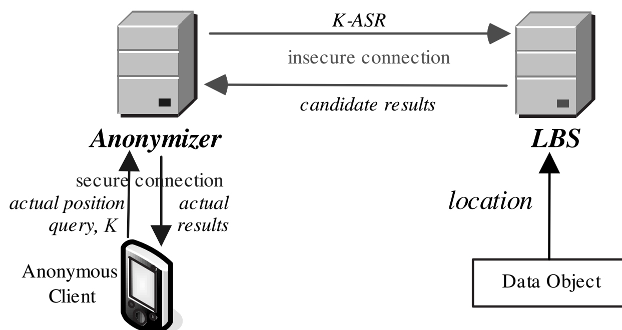


Figure 1.3: Framework for Spatial K-anonymity (SKA)

assembles the corresponding ASR.

The anonymizer then sends the ASR to the LBS, which cannot know which of the enclosed users is the query source. The LBS returns to the anonymizer a set of *candidate results* that satisfy the query condition for any possible point in the ASR. This set includes all hospitals inside the ASR (e.g., h_3), as well as the NN of any point on the ASR perimeter [35]. In the example, the result set consists of h_2, h_3 and h_4. Note that, the number of returned results, as well as the processing cost at the LBS, is dependent on the spatial extent of the ASR; therefore, small ASRs are preferred.

The LBS may be compromised, or it may be malicious itself. Therefore, in the worst case, an adversary may have complete knowledge of all K-ASRs received by the LBS. An SKA method should provide privacy under this scenario, as well.

Existing methods for spatial K-anonymity (reviewed in Chapter 2) have at least one of the following shortcomings: *(i)* They compromise the query issuer's identity for certain user location distributions. In most cases, the privacy of outliers is exposed. *(ii)* They sacrifice quality of service (QoS), i.e., some queries must be delayed or dropped, in order to preserve user privacy. *(iii)* They are ineffective, i.e., they generate large ASRs, resulting in high query processing cost, and increased communication to transfer a large number of candidate results from the LBS back to the AS. *(iv)* They focus exclusively on cloaking mechanisms, and lack algorithms for query processing at the LBS. We address all of these limitations, as described next.

1.1 Contributions and Thesis Organization

The remainder of this dissertation is organized as follows: In Chapter 2, we give a background on LBS query privacy, and survey the related work in the area. Subsequently, we introduce our specific contributions:

- In Chapter 3, we adopt the centralized anonymizer service architecture of Figure 1.3, and address the LBS query privacy problem through a comprehensive set of techniques. Specifically, we identify an important property of ASRs, *reciprocity*, which is a sufficient condition to guarantee query privacy for a snapshot of user locations. Intuitively, reciprocity requires that whenever user u_i includes u_j in its corresponding ASR, u_j also includes u_i in its ASR when it issues a query. We propose two cloaking algorithms: *Nearest Neighbor Cloak* and *Hilbert Cloak*. *Nearest Neighbor Cloak* builds K-ASRs based on user proximity, and significantly outperforms existing techniques in terms of K-ASR size. On the other hand, *Hilbert Cloak* builds upon the reciprocity property, and never reveals the query source, regardless of the user location distribution. Note that, *Hilbert Cloak* is the first technique in literature to provide privacy guarantees for LBS queries.

  Moreover, we address the issue of anonymized query processing at the LBS. Specifically, we adopt an existing algorithm [35] to compute the k nearest neighbors[3] (kNN) of rectangular regions, as opposed to points. We also investigate the use of K-ASRs with non-rectangular shape. In particular, we consider circular-shape K-ASRs, and we develop a novel algorithm to compute the kNN of circular regions. Our experiments reveal that circular K-ASRs reduce the number of redundant results, hence the communication cost between the anonymizer and the LBS.

- Existing work on LBS query privacy assumes that the attacker does not have any prior knowledge on the frequency of issuing queries among various users. However, this is not the case in practice. Users with certain occupations may have a considerably higher frequency of is-

[3]Note that k, the number of nearest neighbors, is different from K, the degree of anonymity.

suing queries. For instance, a taxi driver, or a real estate agent, are likely to issue many more daily queries than an office worker.

Revisiting the example of Figure 1.2, consider the $3-ASR$ enclosing u_1, u_2 and u_3 . If the attacker knows that the frequency of u_1 issuing a query is 2 times larger than that of either u_2 or u_3, then the probability of identifying u_1 as query source becomes $2/4 = 1/2 > 1/K$ for $K = 3$. Therefore, the privacy requirement of u_1 is no longer met. In Chapter 4, we address this scenario: we extend the reciprocity property to account for variable query frequencies among users, and we propose algorithms that preserve privacy even if the attacker possesses query frequency knowledge.

Moreover, we give a general methodology to enforce the reciprocity property (and its frequency-aware counterpart) using a generic spatial index. Specifically, we propose methods to achieve reciprocity with Quad-trees and R-trees. Such methods allow seamless integration of query-privacy services with already existing applications, facilitating the adoption of privacy-aware LBS.

- So far, we have focused on the centralized anonymizer service architecture. Nevertheless, such an approach has several shortcomings: the centralized anonymizer is a bottleneck due to handling query requests, frequent updates of user locations and result post-processing. Furthermore, the anonymizer represents a single point of attack: the complete knowledge of the locations and queries of all users is a serious privacy threat, if the anonymizer is compromised. Even if there is no attack, the centralized anonymizer may be subject to governmental control, and may be banned or forced to disclose sensitive user information (similar to the legal case of the Napster file-sharing service).

  In Chapter 5, we consider a distributed architecture for anonymous location-based queries, which addresses the above-mentioned limitations. Mobile users self-organize into a fault-tolerant, P2P overlay network, and cooperate to assemble K-ASRs. We propose two such protocols: *(i)* The PRIVÉ protocol implements the *Hilbert Cloak* anonymization technique in a decentralized fashion. The structure of the

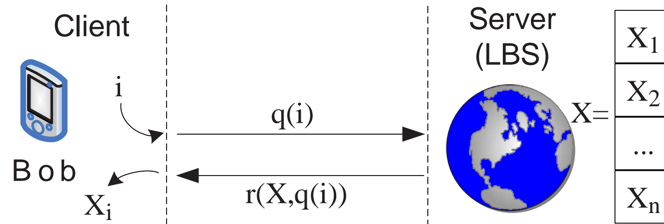


Figure 1.4: PIR framework

network resembles a distributed B$^+$-tree (each mobile user corresponds to a data point), with additional annotation to support efficiently the Hilbert-based K-ASR construction. PRIVÉ avoids the single point of attack of the centralized AS, since the state of the system is distributed in numerous users. However, it may incur slow response time at the high levels of the network tree, during peak load. *(ii)* MOBIHIDE is a scalable P2P anonymization system based on the Chord [57] DHT. It uses a randomized version of *Hilbert Cloak*, which prevents any hotspots in the system. MOBIHIDE does not offer the same theoretical privacy guarantees as PRIVÉ, but it does provide strong privacy in practice. Therefore, we propose two alternative solutions, representing a clear trade-off between privacy and scalability.

- Finally, we move one step beyond the SKA paradigm, and devise a *Private Information Retrieval* (*PIR*)-based solution to LBS query privacy. SKA assumes the existence of a trusted third party anonymizer service, as well as a large number of cooperating LBS users, who are willing to constantly report their location to the AS. Furthermore, users are assumed to be non-malicious, i.e. they do not collude against a target user. Our proposed PIR framework relies on cryptographic techniques, and relinquishes these assumptions: no trusted third-party (either AS or mobile users) is required. Furthermore, no expensive maintenance of locations for a large population of subscribed users is necessary.

  Recent research on PIR [19, 42] resulted in protocols that allow a client to privately retrieve information from a database, without the

database server learning what particular information the client has requested. Most techniques are expressed in a theoretical setting, where the database is an n-bit binary string X (see Figure 1.4). The client wants to find the value of the ith bit of X (i.e., X_i). To preserve privacy, the client sends an encrypted request $q(i)$ to the server. The server responds with a value $r(X, q(i))$, which allows the client to compute X_i. We focus on *computational* PIR, which relies on the fact that it is computationally intractable for an attacker to find the value of i, given $q(i)$. Furthermore, the client can easily determine the value of X_i based on the server's response $r(X, q(i))$.

In Chapter 6, we extend existing PIR protocols for binary data to the LBS domain, and we propose approximate and exact techniques to privately answer NN queries. As opposed to SKA techniques, where the user location is cloaked, but some location-information is still revealed (i.e., the K-ASR area which encloses the query source), the PIR approach does not disclose any spatial information whatsoever, since location data is encrypted. Hence, the PIR method is resilient against any type of location-based attack, including *correlation* attacks, which can be staged when a user issues continuous queries (i.e. the same query is asked at consecutive timestamps, from distinct locations).

Figure 1.5 provides a roadmap of the thesis.

This thesis contains work already accepted for publication, as well as work currently under review. Specifically, Chapter 3 is based on the IEEE TKDE article in [39]. The work in Chapter 4 is currently under review with the VLDB Journal. The PRIVÉ and MOBIHIDE P2P systems presented in Chapter 5 have been published in the proceedings of the International World Wide Web Conference (WWW) [29] and International Symposium on Spatial and Temporal Databases (SSTD) [28], respectively. The work in Chapter 6 is currently under review with the SIGMOD 2008 conference. Furthermore, our research on LBS privacy has provided us with important insights on the related problem of privacy in relational databases, resulting in two other research papers (not included in this thesis, as their focus is not on LBS privacy): a VLDB 2007 paper [30] which uses multi-to-1D mapping

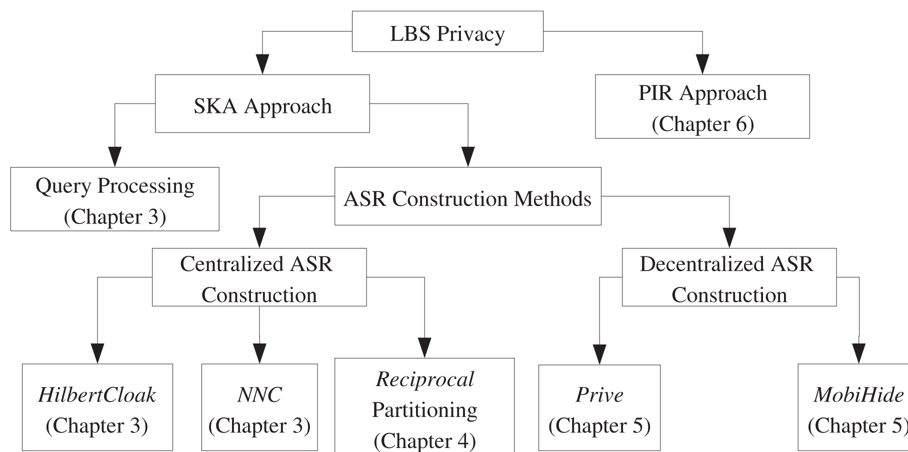


Figure 1.5: Thesis Roadmap

to anonymize relational data, and an ICDE 2008 paper [31], which addresses privacy-preserving publication of transaction (or “market-basket”) data.

Chapter 2

Related Work

This chapter provides background on the LBS query privacy problem, and surveys existing LBS privacy techniques. In Section 2.1, we briefly discuss the K-anonymity paradigm in relational databases, while in 2.2 we present Spatial K-anonymity, and introduce its assumptions and objectives. In Section 2.3, we survey existing SKA techniques, and highlight their limitations. Section 2.4 focuses on processing of anonymized queries (i.e., ASRs) at the LBS. In Section 2.5, we survey P2P techniques that are relevant to our distributed anonymization architecture of Chapter 5, whereas related work on Private Information Retrieval is overviewed in Section 2.6.

2.1 K-anonymity

Extensive research efforts have focused on privacy-preserving publishing of relational data. In this context, released *microdata* (e.g. detailed census or medical records) should not be linked to specific individuals. Adam and Wortmann [3] survey methods for computing aggregate functions (e.g., *sum, count*) under the condition that the results do not reveal any specific record. Agrawal and Srikant [9] employ random perturbation to prevent re-identification of records, by adding noise to the data. In [36], it is shown that an attacker could filter the random noise, and hence breach data privacy, unless the noise is correlated with the data. However, randomly perturbed data is not “truthful” [45], in the sense that it contains records which do not

exist in the original data. Furthermore, random perturbation may expose privacy of outliers when an attacker has access to external knowledge.

Published microdata may contain *quasi-identifier* attributes (QID), such as age, or zipcode, which may be joined with public databases (e.g. voting registration lists) to re-identify individual records. To address this threat, Samarati and Sweeney [53, 58] introduced K-anonymity, a privacy-preserving paradigm which requires each record to be indistinguishable among at least $K-1$ other records with respect to the set of QID attributes. Records with identical QID values form an *equivalence class*, or *anonymized group*. K-anonymity can be achieved through *generalization*, which maps detailed attribute values to value ranges, and *suppression*, which removes certain attribute values or records from the microdata. The process of data anonymization is called *recoding*, and it inadvertently results in information loss. Several privacy-preserving techniques have been proposed, which attempt to minimize information loss, i.e. maximize utility of the data.

Meyerson et al [48] proposed an approximate algorithm that minimizes the number of suppressed quasi-identifier values; the approximation bound is $O(K \cdot logK)$. Aggarwal et al [6] improved this bound to $O(K)$, while Park et al [52] further reduced it to $O(logK)$.

More recent works adopt the generalization of quasi-identifiers. Bayardo et al [12] and LeFevre et al [43] proposed optimal K-anonymity solutions for *single-dimensional* recoding, which performs value mapping independently for each attribute. LeFevre et al [44] introduced *Mondrian*, an heuristic solution for *multi-dimensional* recoding, which performs mapping for the Cartesian product of multiple attributes. Mondrian outperforms optimal single-dimensional solutions, due to its increased flexibility in forming anonymized groups. Methods discussed so far perform *global* recoding, where a particular detailed value is always mapped to the same generalized value. In contrast, *local* recoding allows distinct mappings across different anonymized groups. Clustering-based local recoding methods are proposed in [5, 66]. Xiao and Tao [64] consider the case where each individual requires a different degree of anonymity, whereas Aggarwal [4] shows that anonymizing a high-dimensional relation leads to unacceptable loss of information due to the dimensionality curse.

K-anonymity prevents re-identification of individual records, but it is vulnerable to *homogeneity* attacks, where many (or all) of the records in an anonymized group share the same sensitive attribute (SA) value. ℓ -diversity [47] addresses this vulnerability, and creates anonymized groups in which at least ℓ SA values are "well-represented". Any K-anonymity technique can be adapted to account for SA value diversity, by changing the group validation condition. Nevertheless, K-anonymity techniques use generalization or suppression, and may result in high information loss, especially for high-dimensional QID. Ghinita et al [30] employ multi-dimensional to 1-D transformations to solve efficiently the K-anonymity and ℓ-diversity problems, while [31] presents a technique for privacy-preserving publication of high-dimensional transaction (or "market-basket") data.

Anatomy [63] introduced a novel approach to achieve ℓ-diversity: instead of generalizing QID values, it decouples the SA from its associated QID, and *permutes* the SA values among records. Since QID are published directly, the information loss is reduced. A similar approach is taken in [67].

t-closeness is another privacy paradigm introduced in [46], which attempts to reproduce in each anonymized group the overall distribution of SA values of the entire published table. However, the method proposed to transform the dataset may incur high information loss in practice. Finally, Xiao and Tao [65] have proposed m-invariance, a privacy model for publishing sequential data releases.

2.2 Spatial K-anonymity. Assumptions and Goals

In the LBS domain, K-anonymity was first introduced in [33]. Spatial K-anonymity *(SKA)* prevents an attacker from learning exact user locations. Given a query from user u, SKA techniques replace the exact location of u with an Anonymizing Spatial Region (ASR or K-ASR) that encloses u, as well as $K-1$ other users. Formally:

Definition 2.1. ***[Spatial* K*-anonymity(SKA)]*** *Let H be a set of* K *distinct user entities with locations enclosed in an arbitrary spatial region* K-*ASR. A user $u \in H$ is said to possess* K-*anonymity with respect to* K-

ASR if the probability of distinguishing u among the other users in H does not exceed 1/K. *We refer to* K *as the required degree of anonymity.*

Note that, SKA does *not* depend on the size of the K-ASR. In the extreme case, the K-ASR can degenerate to a point, if K users are at the same location. In general, we prefer small K-ASRs, in order to minimize the processing cost at the LBS and the communication cost between the LBS and the mobile user. Nevertheless, some applications may impose a lower bound on the size of the K-ASR; for instance, it may be forbidden by law to disclose exact user locations [16]. In such a case, the K-ASR can be trivially enlarged to satisfy the lower bound, by symmetrical scaling in all directions. The same procedure can also be used to avoid having users on the perimeter of the K-ASR.

SKA is commonly performed by an *Anonymizer Service (AS)*, or simply anonymizer. The anonymizer is a trusted server, which collects the current location of users and anonymizes their queries. Each query has a required degree of anonymity K, which ranges between 1 (no privacy requirements) and the user cardinality (maximum privacy). We assume that an attacker has complete knowledge of *(i)* all the ASRs ever received at the LBS, *(ii)* the cloaking algorithm used by the anonymizer, and *(iii)* the locations of all users. The first assumption states that either the LBS is not trusted (e.g., a commercial service that collects unauthorized information about its clients for unsolicited advertisements), or the communication channel between the anonymizer and the LBS is not secure. The second assumption is common in the security literature since the data privacy algorithms are usually public.

The third assumption is motivated by the fact that users may often (or always) issue queries from the same locations (home, office), which may be easily identified through public databases, telephone directories, etc. Furthermore, they may reveal their locations by issuing queries without privacy requirements. In scenarios with highly mobile users, the attacker may not be able to learn exact user locations. However, one can argue that in these cases spatial K-anonymity is not important, because *(i)* the user ids are removed by the anonymizer anyway, and *(ii)* a query at a random position does not necessarily reveal information about the identity of the corresponding user.

However, in practice, a determined attacker may be able to acquire (through triangulation, public databases, physical observation, etc.) the locations of at least a few users in the vicinity of the targeted victim.

Similar to existing work on SKA [21, 33, 49] we focus on *snapshot* queries, where the attacker uses current data, but not historical information about movement and behavior patterns of particular clients[1] (e.g., a user often asking a particular query at a certain location or time). We also assume that the value of K is not subject to attacks since it is transferred from the client to the anonymizer through a secure channel.

Given a query, the anonymizer removes the user id, applies cloaking to hide the user's location through an ASR, and forwards the ASR to the LBS. The cloaking algorithm is said to preserve spatial K-anonymity, if the probability of the attacker pinpointing the query source under the above assumptions does not exceed $1/K$.

Note that simply generating an ASR that includes K users is not sufficient for spatial K-anonymity. Consider for instance, a naïve algorithm, called *Center Cloak* (CC) in the sequel, which given a query from u, finds his $K-1$ closest users, and sets the ASR as the minimum bounding rectangle (MBR) or circle (MBC) that encloses them. In fact, a similar technique is proposed in [21] for anonymization in peer-to-peer systems, i.e., the K-ASR contains the query issuing peer and its $K-1$ nearest nodes. CC is likely to disclose the location of u under the *center-of-ASR* attack. Specifically, let $index_u$ be the position of u in the sequence of users enclosed by the K-ASR, sorted in ascending order of their distance from the center of the K-ASR; for example, if $index_u = 1$, then u is the closest user to the center. The *center-of-ASR* attack is successful if $P[index_u = 1] > 1/K$, i.e., if the probability of u being the closest user to the center exceeds $1/K$.

Figure 2.1 shows the distribution of the positions of u inside an MBR enclosing its 9 NNs (for details of the experimental setting, see Section 3.6). In most cases, u is close to the center of the 10-ASR (i.e., $P[index_u = 1] > 1/10$). Hence, an attacker with knowledge of the cloaking algorithm (assumption *ii*) may easily pinpoint u as the query source. Note that, since the

[1] In Chapter 6 we present a technique which guarantees privacy for continuous queries as well; however, that technique relies on PIR, and not on SKA

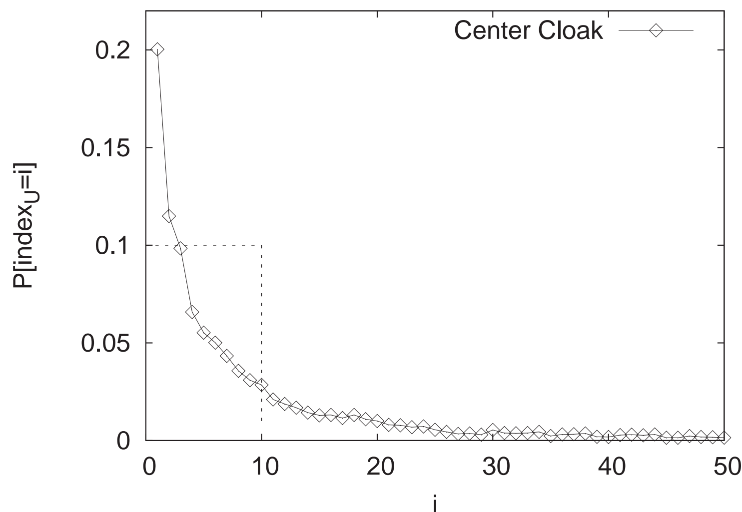


Figure 2.1: Distance from MBR center for *Center Cloak* (K=10)

MBR may enclose more than 10 users it is possible to get $P[index_u = i] > 0$ for $i > 10$. The dashed line in the graph corresponds to the “flat” index distribution obtained by an ideal anonymization technique, which would always generate 10-ASRs with exactly 10 users.

In addition to the preservation of spatial K-anonymity, we define the following objectives of cloaking:

1. The generated ASR should be as small as possible.

2. The cloaking algorithm should not compromise the quality of service (QoS).

3. The ASR should not reveal the exact location of any user.

Goal 1 is induced by the fact that a large ASR incurs higher processing overhead (at the LBS) and network cost (for transferring a large number of candidate results from the LBS to the anonymizer). In real-world services, users may be charged depending on the overhead that the anonymization requirements impose on the system. Note that, as long as the anonymity requirements of the user are satisfied, the size of the ASR is irrelevant in terms of K-anonymity. Goal 2 states that systems that delay or reject service requests, such as *Clique Cloak* [27] (reviewed in Section 2.3), are

unacceptable. In general, since temporal cloaking compromises QoS, we focus our attention on spatial cloaking. Goal 3 ensures that the anonymizer does not help the attacker obtain the locations of users through the cloaking algorithm (although, as discussed before, he may obtain them through other means). The disclosure of exact locations by a service is undesirable to most users (independently of their queries), and in some cases forbidden by law. As an example, consider that the anonymizer picks $K - 1$ random users and sends K independent queries (including the real one) to the LBS. This method achieves spatial K-anonymity, but reveals the exact locations of K users. Furthermore, it has several efficiency problems: *(i)* depending on the value of K, a potentially large number of locations are transmitted to the LBS and *(ii)* the LBS has to process K independent queries and send back all their results.

Let u be the user issuing a query. The proposed cloaking algorithms first generate an anonymizing set (AS) that contains u and at least $K - 1$ users in u's vicinity. The ASR is an area that encloses all users in AS. Although the ASR can have arbitrary shape, we use minimum bounding rectangles (MBR) or circles (MBC) because they incur small network overhead (when transmitted to the LBS) and facilitate query processing. Note that, in addition to AS, the ASR may enclose some additional users that fall in the corresponding MBR or MBC.

2.3 Existing SKA Techniques

Most previous work on location-based services adopts the concept of K-anonymity using the framework of Figure 1.3: a user sends his position, query and K to the anonymizer, which removes the id of the user and transforms his location through cloaking. The generated K-ASR is forwarded to the LBS which processes it and returns a set of candidates, containing the actual results and false hits. The first cloaking[2] technique, called *Interval Cloak* [33] is based on quadtrees. A quadtree [54] recursively partitions the space into quadrants until the points in each quadrant fit in a page/node.

[2]Beresford and Stajano [15] introduce the concept of mix zone, which is similar to the K-ASR, but do not provide concrete algorithms for spatial cloaking.

Figure 2.2 shows the space partitioning and a simple quadtree assuming that a node contains a single point. The anonymizer maintains a quadtree with the locations of all users. Once it receives a query from a user U, it traverses the quadtree (top-down) until it finds the quadrant that contains U and fewer than $K-1$ users. Then, it selects the parent of that quadrant as the K-ASR and forwards it to LBS.

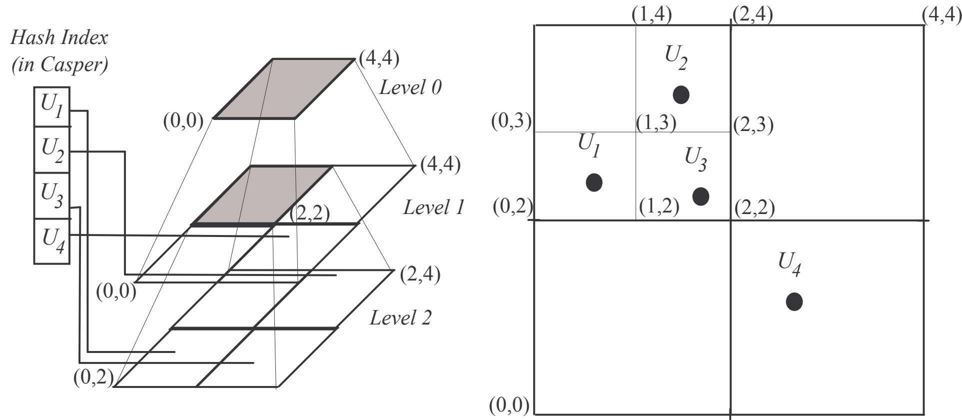


Figure 2.2: Example of *Interval Cloak* and *Casper*

Assume that in Figure 2.2, U_1 issues a query with K=2. Quadrant[3] $\langle(0,2),(1,3)\rangle$ contains only U_1, so its parent $\langle(0,2),(2,4)\rangle$ becomes the 2-ASR. Note that the ASR may contain more users than necessary; in this example it includes U_1, U_2, U_3, although 2 users would suffice for the privacy requirements. A large ASR burdens the query processing cost at the LBS and the network overhead for transferring a large number of candidate results from the LBS to the anonymizer. In order to overcome this problem, Gruteser and Grunwald [33] combine *temporal cloaking* with spatial cloaking, i.e., the query may wait until K (or more) objects fall in the user's quadrant. In our example, the query of U_1 will be executed when a second user enters $\langle(0,2),(1,3)\rangle$, in which case $\langle(0,2),(1,3)\rangle$ is the 2-ASR sent to the LBS.

Similar to *Interval Cloak*, *Casper* [49] is based on quadtrees. The anonymizer uses a hash table on the user id pointing to the lowest-level quadrant where the user lies. Thus, each user is located directly, without having

[3]We use the coordinates of the lower-left and upper-right points to denote a quadrant.

to access the quadtree top-down. Furthermore, the quadtree can be adaptive, i.e., contain the minimum number of levels that satisfies the privacy requirements. In Figure 2.2, for instance, the second level for quadrant $\langle(0,2),(2,4)\rangle$ is never used for $K \geq 2$ and can be omitted. The only difference in the cloaking algorithms of *Casper* and *Interval Cloak* is that *Casper* (before using the parent node as the K-ASR) also considers the neighboring quadrants at the same level of the tree. Assume again that in Figure 2.2 U_1 issues a query and K=2. *Casper* checks the content of quadrants $\langle(1,2),(2,3)\rangle$ and $\langle(0,3),(1,4)\rangle$. Since the first one contains user U_3, the 2-ASR is set to $\langle(0,2),(2,3)\rangle$, which is half the size of the 2-ASR computed by *Interval Cloak* (i.e., $\langle(0,2),(2,4)\rangle$).

However, *Interval Cloak* and *Casper* may compromise location anonymity in the presence of outliers. Consider the example of Figure 2.2 assuming that $K=2$. If a query originates from U_1, U_2, or U_3, the 2-ASR of *Interval Cloak* is quadrant $\langle(0,2),(2,4)\rangle$. Similarly, the 2-ASR of *Casper* is the concatenation of two sibling quadrants at level 2 (e.g., $\langle(0,2),(1,3)\rangle$ and $\langle(1,2),(2,3)\rangle$). On the other hand, if a query originates from U_4, the 2-ASR is the entire data-space $\langle(0,0),(4,4)\rangle$ for both *Interval Cloak* and *Casper*. Thus, an attacker can identify U_4 for all 2-ASRs that cover the entire data-space.

For illustration purposes, in the above examples we assumed that the attacker knows K, although as discussed in Section 2.2, K is not subject to attacks. Nevertheless, even for variable and unknown K, the presence of outliers may compromise spatial anonymity. We demonstrate the problem for *Interval Cloak* and *Casper* using Figure 2.3. There is a single user U_1 in quadrant $\langle(0,0),(1,1)\rangle$ and $N-1$ users in $\langle(1,1),(2,2)\rangle$, where N is the user cardinality. Quadrant $\langle(1,1),(2,2)\rangle$ may be subdivided further, but this is not important for our discussion. Each user has equal probability to issue a query, and the degree of anonymity required by different queries distributes uniformly in the range $[1,N]$. The term *event* signifies the issuance of a query with anonymity degree K at a random user U. Then, an ASR covering the entire data space is generated by *(i)* a query originating from U_1 and $2 \leq K \leq N$ (i.e., $N-1$ events), or *(ii)* a query originating from another user and $K=N$ (i.e., $N-1$ events). Thus, if the attacker detects such an ASR

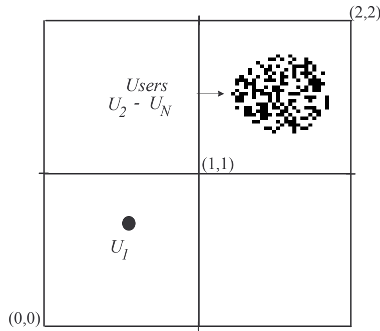


Figure 2.3: Location anonymity compromise in the presence of outliers

and has knowledge of the user distribution (assumption *iii* in Section 2.2), then he concludes that it originated from U_1 with probability 1/2. Thus, the spatial anonymity of U_1 is breached for all values $K > 2$.

In general, following a similar analysis, we show in Appendix A that, if any two quadrants contain a different number of users, the location anonymity is compromised (for all values of K exceeding a threshold) in the quadrant containing the smaller number.

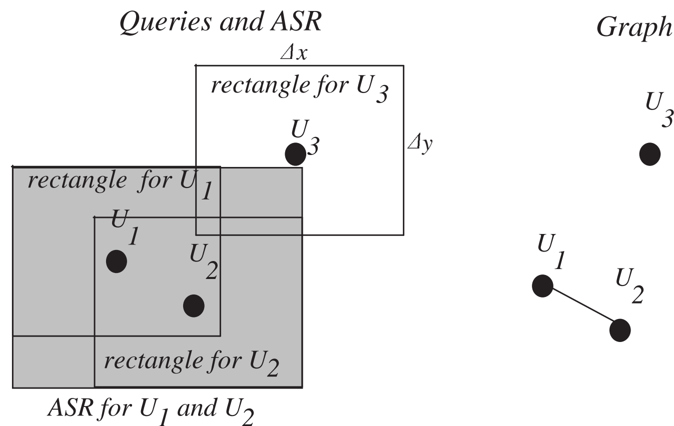


Figure 2.4: Example of *Clique Cloak*

In *Clique Cloak* [27], each query defines an axis-parallel rectangle whose centroid lies at the user location and whose extents are $\Delta x, \Delta y$. Figure 2.4 illustrates the rectangles of three queries located at U_1, U_2, U_3, assuming that they all have the same Δx and Δy. The anonymizer generates a graph where

a vertex represents a query: two queries are connected if the corresponding users fall in the rectangles of each other. Then, the graph is searched for cliques of K vertices and the minimum bounding rectangle (MBR) of the corresponding rectangles forms the ASR sent to the LBS. Continuing the example of Figure 2.4, if K=2, U_1 and U_2 form a 2-clique and the MBR of their respective rectangles is forwarded so that both queries are processed together. On the other hand, U_3 cannot be processed immediately, but it has to wait until a new query (generating a 2-clique with U_3) arrives. *Clique Cloak* allows users to specify a temporal interval Δt such that, if a clique cannot be found within Δt, the query is rejected. The selection of appropriate values for $\Delta x, \Delta y, \Delta t$ is not discussed in [27].

Chow and Mobkel [20] identified, independently from our work, the *K-sharing* property, which is similar to the *reciprocity* that we propose[4] in Chapter 3. The authors of [20] also consider an extension of *K-sharing*, which aims to prevent correlation attacks, i.e. attacks based on history of user movement. If a user issues a continuous query, i.e. a sequence of shapshot queries from different locations at consecutive timestamps, the attacker can corroborate information from all snapshots to infer the query source. [20] protects against correlation attacks as follows: At the initial timestamp t_0, it builds ASR_0, which encloses a set AS of at least K users. At a subsequent timestamp t_i, the algorithm computes a new anonymizing region ASR_i that encloses the same users in AS, but contains their locations at timestamp t_i. There are two drawbacks: *(i)* As users move, the resulting CR can grow very large, leading to prohibitive query cost. *(ii)* If a user in AS disconnects from the service, the query must be dropped.

Location anonymity has also been studied in the context of related problems. *Probabilistic Cloaking* [18] preserves the privacy of locations without applying spatial K-anonymity. Instead, *(i)* the ASR is a closed region around the query point, which is independent of the number of users inside and *(ii)* the location of the query is uniformly distributed in the ASR. Given an ASR, the LBS returns the probability that each candidate result satisfies the query, based on its location with respect to the ASR. Kamat et al. [40]

[4]Note that, our work in [29] pre-dates the work in [20], therefore the reciprocity property that we propose is the first work to provide privacy guarantees

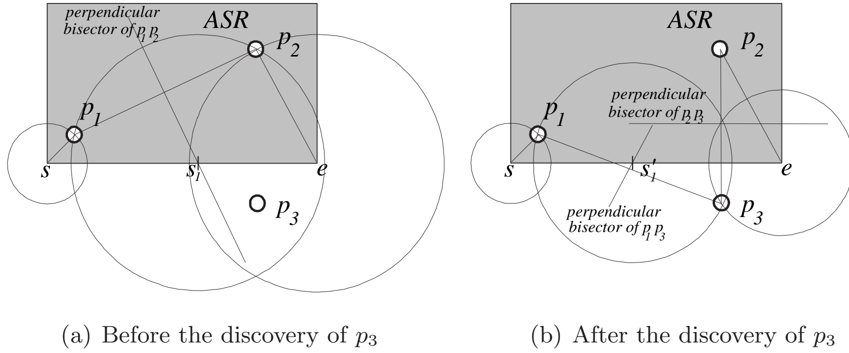
propose a model for sensor networks and examine the privacy characteristics of different sensor routing protocols. Hoh and Gruteser [34] describe techniques for hiding the trajectory of users in applications that continuously collect location samples. Chow et al. [21] study spatial cloaking in peer-to-peer systems.

An encryption-based approach is considered in [41]: In a preprocessing phase, a trusted third party transforms (using 2-D to 1-D mapping) and encrypts the database. The database is then uploaded to the LBS, which does not know the decryption key. All users possess tamper-resistant devices which store the decryption key, but they do not know the key themselves. Users send encrypted queries to the LBS and decrypt the answers to extract the results. The method assumes that none of the tamper-resistant devices is compromised. If this condition is violated, the privacy of all users can be compromised. Moreover, there is no guarantee against correlation attacks, in which an attacker combines information from multiple queries issued by the same user from distinct locations.

2.4 Related Spatial Query Processing Techniques

The LBS maintains the locations of points-of-interest and answers cloaked queries. The most common spatial queries, and the focus of the existing systems, are ranges and nearest neighbors (NN). While the cloaking mechanism at the anonymizer is independent of the query type, query processing at the LBS depends on the query. Range queries are usually straightforward; assume that a user U wants to retrieve the data objects within distance d from his current location. Instead of the position of U, the LBS receives (from the anonymizer), an ASR that contains U (as well as several other users) and d. In order to compute the candidate results, the LBS extends the ASR by d in all dimensions and searches for all objects in the extended ASR. The set of candidates is returned to the anonymizer which filters out false hits and returns the actual result to U.

The processing of NN queries is more complicated. If the ASR is an axis-parallel rectangle (as in *Interval Cloak*, *Casper* and *Clique Cloak*), then the candidate results can be retrieved using *range nearest neighbor* search



(a) Before the discovery of p_3 (b) After the discovery of p_3

Figure 2.5: Example of continuous NN search

[35], which finds the NN of any point inside a rectangular range. Assume the example of Figure 1.2(right). The LBS must return the NN of every possible location in the ASR. Such candidate data points lie inside (e.g., h_3), or outside the ASR (e.g., h_2, h_4). For instance, h_4 would be the NN for user u_3, or another user situated at the top-right corner of the ASR.

Figure 2.5 shows an example of the application of range nearest neighbor search for three points of interest stored at the LBS, denoted by $p_1 \ldots p_3$. The initial set of candidates contains all points (p_1, p_2) inside the input range (i.e., the ASR). Then, four *continuous NN* (CNN) queries [60], one for each side of the ASR, retrieve the remaining candidates. Consider, for instance, the CNN query for the bottom side se. The initial candidates split se into two intervals: ss_1 and s_1e, where s_1 is the point where the perpendicular bisector of p_1p_2 intersects se. Currently, the NN of every point in ss_1 is p_1, whereas the NN of every point in s_1e is p_2. The three *vicinity circles* in Figure 2.5a, are centered at s, s_1, e and their radii equal the distances between s and p_1, s_1 and p_1 (or p_2), and e and p_2, respectively. The only data points that can be closer to se (than p_1 and p_2) must fall inside some vicinity circle.

Continuing the example, p_3 falls inside the last two vicinity circles and updates the result as shown in Figure 2.5b. Specifically, s_1' is the point where the perpendicular bisector of p_1p_3 intersects se: p_1 becomes the NN of every point in ss_1', and p_3 the NN of every point in $s_1'e$. Note that the vicinity circles shrink as new data points are discovered. The process terminates

when no more points are found within the vicinity circles. It can be shown [35] that four CNN queries for the four sides of the ASR find all candidate objects. A similar technique (also for rectangular ranges) is presented for *Casper* in [49]; in Section 3.5, we develop a method capable of processing circular ranges.

2.5 Related P2P Systems

In Chapter 5, we will introduce two P2P protocols for distributed anonymization of LBS queries. We further give a brief overview of the most prominent P2P systems related to our work.

Key and range search has been studied extensively in distributed environments. Several structured Peer-to-Peer systems (e.g, Chord [57]) support distributed key search with $O(\log N)$ complexity. The drawback of such systems is that they cannot support efficiently node annotation. Without node annotation, the communication cost for satisfying the reciprocity property (which guarantees K-anonymity) is $O(N)$; this cost is too high for large scale systems. Closer to our work is the P-tree [22], which supports range queries by embedding a B$^+$-tree on top of an overlay network. No global index is maintained; instead each node maintains its own B$^+$-tree-like structure. BATON [38] also addresses range queries, by embedding a balanced tree onto an overlay network. It uses additional cross-links to prevent hotspots, and achieves $O(\log N)$ complexity for search and maintenance. Similar to Chord, these systems cannot support efficiently node annotation.

Hierarchical clustering in distributed environments has been an active research topic in recent years. In [11], a hierarchical-clustering routing protocol for wireless networks is presented. The NICE project [10] proposes a scalable application-layer multicast protocol, based on delivery trees built on top of a hierarchically connected control topology. Nodes participating in a multicast group are organized into a multi-layer hierarchy of clusters with bounded size. NICE trees obtain delays in the order of $O(\log N)$, where N is the size of the multicast group, and there is an upper bound of $O(\log N)$ in terms of control state maintained per node. Our protocols also use hierarchical clustering of mobile users, but the requirements of total ordering

and annotation impose particular challenges that have not been addressed by existing research.

2.6 Private Information Retrieval

In Chapter 6, we develop an LBS privacy solution that relies on *Private Information Retrieval (PIR)*. Our work builds on the theoretical results for the PIR problem, which is defined as follows: a server S holds a database with n bits, $X = (X_1 \ldots X_n)$. A user u has a particular index i and wishes to retrieve the value of X_i, without disclosing to S the value of i. The PIR concept was introduced by Chor et al [19] in an information theoretic setting, requiring that even if S had infinite computational power, it could not find i. In this context it was proved that in any solution with a single server, u must receive the entire database (i.e., $O(n)$ cost). The communication cost can be reduced to $n^{O(\frac{\log \log K}{K \log K})}$ if the database is replicated in K non-colluding servers [14]. Nevertheless, in practice, it is sufficient to ensure that S cannot find i with polynomial-time computations; this problem is known as *Computational PIR*. Kushilevitz et al [42] showed that the communication cost for a single server is $O(n^{\varepsilon})$, where ε is an arbitrarily small positive constant. Our work employs Computational PIR.

Several approaches employ cryptographic techniques to privately answer NN queries in relational data. Most of them are based on some version of the secure multiparty computation problem [32]. Let two parties A and B hold objects a and b, respectively. They want to compute a function $f(a, b)$ without A learning anything about B and vice versa. They encrypt their objects using random keys and follow a protocol, which results into two "shares" S_A and S_B given to A and B, respectively. By combining their shares, they compute the value of f. In contrast to our problem (which hides the querying user from the LBS), existing NN techniques assume that the query is public, whereas the database is partitioned into several servers, neither of which wants to reveal their data to the others. [62] assumes vertically partitioned data and uses secure multiparty computation to implement a private version of Fagin's [24] algorithm. [55] follows a similar approach, but data is horizontally partitioned among the servers. The computation cost is $O(n^2)$

and may be prohibitive in practice. [7] also assumes horizontally partitioned data, but focuses on top-k queries.

More relevant to our problem is the work of [37] which uses PIR to compute the NN of a query point. The server does not learn the query point and the user does not learn anything more than the NN. To achieve this, the method computes private approximations of the Euclidean distance by adapting an algorithm [25] that approximates the Hamming distance in $\{0,1\}^d$ space (d is the dimensionality). The cost of [37] is $\tilde{O}(n^2)$ for the exact NN and $\tilde{O}(\sqrt{n})$ for an approximation through sampling. The paper is mostly of theoretical interest, since the $\tilde{O}$ notation hides polylogarithmic factors that may affect the cost; the authors do not provide any experimental evaluation of the algorithms.

Chapter 3

SKA Framework for LBS Privacy

3.1 Introduction

This chapter presents our comprehensive SKA framework for LBS query privacy. Our framework includes techniques for generating K-ASRs at the anonymizer, as well as algorithms to process transformed queries at the LBS. Similar to existing SKA work, we consider a centralized architecture[1], with an intermediate AS server between the mobile users and the LBS (see Figure 1.3). Furthermore, we assume that an attacker does not have a priori knowledge of the user query frequencies (i.e., a query may originate from any user with equal probability). We remove this assumption in Chapter 4.

In Section 3.2 we propose the *Nearest Neighbor Cloak* cloaking technique, which clearly outperforms existing methods in terms of K-ASR size. Section 3.3 introduces the *reciprocity* concept, a sufficient condition to achieve privacy, based on which, in Section 3.4, we propose the *Hilbert Cloak* algorithm. In Section 3.5 we focus on anonymized query processing at the LBS.

[1] Later in Chapter 5 we remove the centralized AS, and propose a decentralized solution

3.2 Nearest Neighbor Cloak

Nearest Neighbor Cloak (*NNC*) is a randomized variant of *Center Cloak* (presented in Section 2.2), and is not vulnerable to *center-of-ASR* attacks. Given a query from U, *NNC* first determines the set S_0 containing U and his K-1 nearest users. Then, it selects a random user U_i from S_0 (the probability of selecting the initial user U is $1/K$) and computes the set S_1, which includes U_i and his K-1 nearest neighbors (NNs). Finally, *NNC* obtains $S_2 = S_1 \cup U$, i.e., S_2 corresponds to the anonymizing set. This step is essential, since U is not necessarily among the NNs of U_i. The K-ASR is the MBR or MBC enclosing all users in S_2.

Example 3.1. Figure 3.1 shows an example of *NNC*, where U_1 issues a query with K=3. The 2 NNs of U_1 are U_2, U_3, and $S_0 = \{U_1, U_2, U_3\}$. *NNC* randomly chooses U_3 and issues a 2-NN query, forming $S_1 = \{U_3, U_4, U_5\}$. The 3-ASR is the MBR enclosing $S_2 = \{U_1, U_3, U_4, U_5\}$. *NNC* can be used with variable values of K. It is not vulnerable to the *center-of-ASR* attack since the probability of U being near the center of the K-ASR is at most $1/K$ (due to the random choice). Furthermore, as we show in the experimental evaluation of Section 3.6, the ASR is much smaller than that of *Interval Cloak* and *Casper*. □

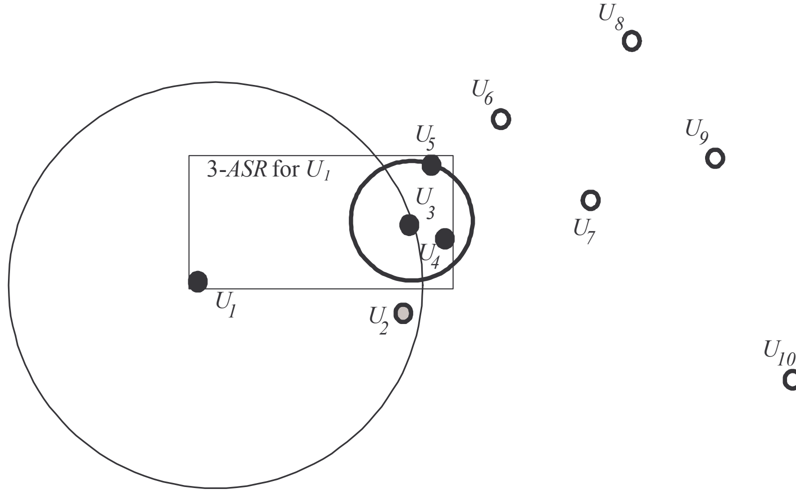


Figure 3.1: Example of *NNC*

However, *NNC*, as well as *Interval Cloak* and *Casper*, may compromise location anonymity in the presence of outliers. Consider that in Figure 3.1, an adversary knows the locations of the users and the value of K. Then, he can be sure that the query originated from U_1 because if it were issued by any other user (U_3, U_4, U_5) in the 3-ASR, the ASR would not contain U_1. Next, we introduce the *reciprocity* principle, which is sufficient to guarantee query privacy, regardless of user location distribution.

3.3 Reciprocity

We identify the following property that is sufficient for a K-ASR construction technique in order to preserve user privacy:

Definition 3.2. ***[K-ASR Reciprocity]*** *Consider a user u_q issuing a query and its associated* K-*ASR A_q. A_q satisfies the reciprocity property iff there exists a set of users AS lying inside A_q such that* (i) $|AS| \geq K$*,* (ii) $u_q \in AS$ *and* (iii) *every user $u \in AS$ lies in the K-ASRs of all other users in AS.*

Example 3.3. Fig. 3.2 shows an example with ten users. For K=5, the K-ASR of users $u_1, u_3, u_4, u_8, u_{10}$ is area A_1 and the K-ASR of users u_2, u_5, u_6, u_7, u_9 is area A_2. In this example, ASRs of all users satisfy the reciprocity property. For instance, for user u_1, if we set $AS = \{u_1, u_3, u_4, u_8, u_{10}\}$, we may easily verify that AS satisfies all the requirements of the reciprocity property. □

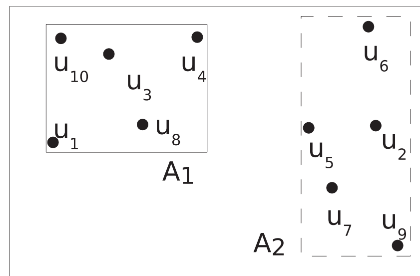


Figure 3.2: K-ASR Reciprocity Example, K=5

Theorem 3.4. *For a given snapshot of user locations, and regardless of the query distribution among users, a K-ASR construction technique* guaran-

tees *spatial K-anonymity if* every *generated K-ASR satisfies the reciprocity property.*

Proof. We assume the worst case scenario, where an attacker knows the exact location of all users in the system (from an outside source). The attacker possesses a set $\mathbf{A}$ of K-ASRs associated to user queries.

Consider K-ASR $A_q \in \mathbf{A}$. The attacker attempts to infer the user u_q that constructed A_q. Since A_q satisfies the reciprocity property, there exists a set of users AS (lying inside A_q) such that *(i)* $|AS| \geq K$, *(ii)* $u_q \in AS$ and *(iii)* every user $u \in AS$ lies on the K-ASRs of all other users in AS.

Moreover, since every K-ASR satisfies the reciprocity property, it follows that when the attacker inspects any K-ASR that includes u_q, he will observe the same set of users AS. Therefore, for all users u in AS, the probability P_u of being the query issuer is:

$$P_u = P_{u_q} = \frac{1}{|AS|} \leq \frac{1}{K}$$

Hence, the K-anonymity property is satisfied. □

In general, *Interval Cloak*, *Casper* and *NNC* do not satisfy reciprocity as they violate condition *(iii)*. For instance, in the example of Figure 2.3, although users $U_2 \dots U_N$ lie in the K-ASR of U_1, U_1 is not in the K-ASR of $U_2 \dots U_N$ for $2 \leq K < N$. Similarly for *NNC*, although in Figure 3.1 $U_3 \dots U_5$ are in the 3-ASR of U_1, U_1 is not in the 3-ASR of $U_3 \dots U_5$.

In view of this property, an optimal K-ASR construction algorithm would partition the user population into K-ASRs that possess the reciprocity property, such that the sum of areas of the resulting K-ASRs is minimized. However, finding this optimal K-anonymity solution, which is similar to finding the optimal K-anonymous generalization of a dataset, is an NP-Hard problem [48]. Next, we introduce an efficient algorithm that enforces reciprocity, and at the same time generates K-ASRs with low spatial extent.

3.4 Hilbert Cloak

Hilbert Cloak(*HC*) uses the Hilbert space-filling curve [50] to generate small

(but not necessarily optimal) ASRs for variable values of K. The Hilbert space filling curve transforms the 2-D coordinates of each user into a 1-D value $H(U)$. Figure 3.3 illustrates the Hilbert curves for a 2-D space using a 4×4 and 8×8 space partitioning. With high probability [50], if two points are in close proximity in the 2-D space, they will also be close in the 1-D transformation. A major benefit of Hilbert (and similar) curves, is that they permit the indexing of multidimensional objects through one-dimensional structures (e.g., B$^+$-trees).

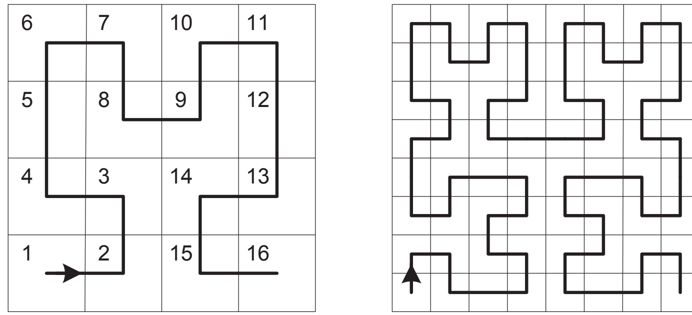


Figure 3.3: Hilbert Curve (left: 4×4, right: 8×8)

Given a query from user U with anonymity requirement K, HC sorts the Hilbert values and splits them into K-buckets. Each K-bucket has exactly K users, except the last one which may contain up to $2{\cdot}K$-1 users. Let $H(U)$ be the Hilbert value of U and $rank_U$ be the position of $H(U)$ in the sorted sequence of all locations. HC identifies the K-bucket containing $rank_U$. The users in that K-bucket constitute the corresponding AS. Figure 3.4 illustrates an example, where the user ids indicate their Hilbert order. For K=3, the users are grouped into 3 buckets (the last one contains 4 users). When any of U_1, U_2 or U_3 issues a query, HC returns the first bucket (shown shaded) as the AS; the MBR (or MBC) of that bucket becomes the 3-ASR.

HC is reciprocal because all users in the same bucket share the same K-ASR; therefore, it guarantees spatial anonymity according to Theorem 3.4. Furthermore, it can deal with variable values of K by not physically storing the K-buckets. Instead, it maintains a balanced sorting tree, which indexes the Hilbert values. When a user U initiates a query with anonymity degree K, HC performs a search for $H(U)$ in the index and computes $rank_U$. From

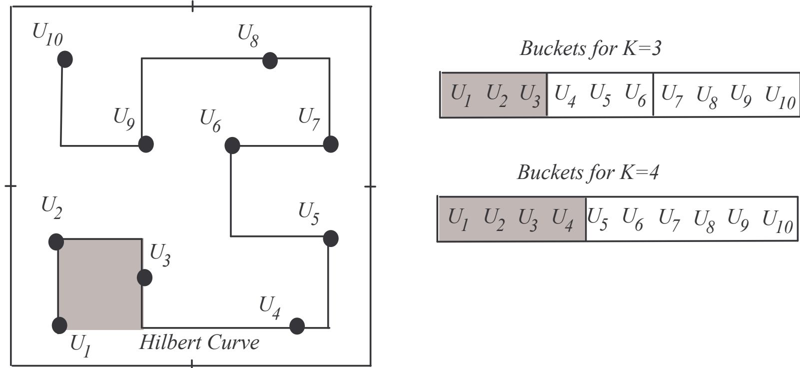


Figure 3.4: Example of *Hilbert Cloak*

$rank_U$, we calculate the start and end positions defining the K-bucket that includes $H(U)$, as follows:

$$start = rank_U - (rank_U \mod K), \qquad end = start + K - 1 \qquad (3.1)$$

The complexity of the in-order tree traversal is $O(N)$, where N is the number of indexed users. To compute $rank_U$ efficiently, we use an aggregate tree [59], where each node w stores the number w_{count} of nodes in its left subtree (including itself). Using this data structure, $rank_U$ is calculated in $O(logN)$ as follows: we initialize $rank_U$ to zero and perform a normal lookup for $H(U)$. For every node w we visit, we add w_{count} to $rank_U$ only if we follow a right-hand branch. The complexity of maintaining the aggregate information is $O(logN)$ because changes are propagated from the leaves to the root. Since the complexity of constructing the K-ASR is $O(logN + K)$, whereas search, insert and delete cost $O(logN)$, the data structure is scalable. Therefore, *HC* is applicable to a large number of mobile users who update their location frequently and have varying requirements for the degree of anonymity. Note that, while our description assumes a main memory index, the technique can be easily extended to secondary memory by using B^+-trees (we address ASR construction with various disk-based index structures in Chapter 4).

3.5 Location-Based Service Query Processing

The Location-Based Service (LBS) receives the query from the anonymizer, processes it and sends the results back to the anonymizer. In our implementation, the data in the LBS are indexed by an R*-Tree [13]; our methods, however, are independent of the index structure. We support two types of queries:

1. Range queries: The LBS receives the query range which is either an axis-parallel rectangle $\mathcal{R}$ or a circle $\mathcal{C}$. Processing is straight-forward; the R-tree is traversed from the root to the leaves and any object inside $\mathcal{R}$ (or $\mathcal{C}$) is returned.

2. kNN queries: This case is more complex, since the LBS must find the k nearest neighbors of the entire range. For rectangular ranges, we adopt the Range Nearest Neighbor ($\mathcal{R}k$NN) algorithm [35] (see Section 2.4 for details). The rest of this section describes our $\mathcal{C}k$NN algorithm, which computes the kNNs of circular ranges.

3.5.1 $\mathcal{C}k$NN - Circular Range kNN

Similar to rectangular ranges [35], the set of kNNs of a circular range $\mathcal{C}$ also consists of two subsets of objects: *(i)* all the objects inside $\mathcal{C}$ and *(ii)* the kNNs of the circumference of $\mathcal{C}$. The objects in *(i)* are retrieved by a range query; in the rest of the section, we present the novel $\mathcal{C}k$NN-Circ algorithm which computes the kNNs of the circumference of $\mathcal{C}$. Intuitively $\mathcal{C}k$NN-Circ is similar to CNN (see Section 2.4). However, some of the properties of 1-D shapes which are used in CNN (e.g., continuity by the definition of [60]) do not hold for 2-D shapes, rendering the problem more complex.

Conceptually, $\mathcal{C}k$NN-Circ partitions the circumference of $\mathcal{C}$ into disjoint arcs, and associates to each arc the data objects nearest to it. Consider the example of Figure 3.5, where p_1, p_2 and p_3 are the data objects. Let s_0, s_1 be the intersection points of the *perpendicular bisector* of p_1p_2 (denoted by $\perp p_1p_2$) with $\mathcal{C}$, i.e., $|p_1s_0| = |s_0p_2|$ and $|p_1s_1| = |s_1p_2|$. Assuming that the center c of $\mathcal{C}$ is the origin of the coordinate system, the *polar* coordinates of s_0 are $(r, \hat{s_0})$, where r is the radius of $\mathcal{C}$ and $\hat{s_0}$ is the (anti-clockwise) angle

between the x-axis and the vector $c\vec{s}_0$. Similarly, the polar coordinates of s_1 are $(r, \hat{s_1})$. The NN of every point in the arc $[\hat{s_0}, \hat{s_1}]$ is p_1; we denote this as: $[\hat{s_0}, \hat{s_1}] \rightarrow p_1$. Likewise $[\hat{s_1}, \hat{s_0}] \rightarrow p_2$, since any point in the arc $[\hat{s_1}, \hat{s_0}]$ is closer to p_2 than to any other object. Therefore, the set of NNs of $\mathcal{C}$ is $\{p_1, p_2\}$. Note that p_3 is not in this set, even though it is closer to $\mathcal{C}$ than p_2, because p_1 is closer than p_3 to any point on $\mathcal{C}$; we say that p_1 *covers* p_3.

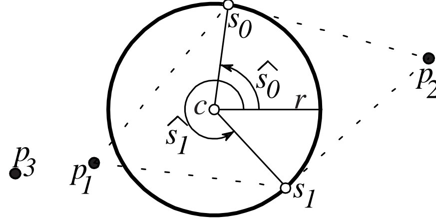


Figure 3.5: The 1-NNs of $\mathcal{C}$ are p_1 and p_2

Let $\mathcal{D} = \{p_1, p_2, \ldots, p_n\}$ be the set of all data objects. $\mathcal{C}k$NN-Circ maintains a list SL of mappings $[a, b] \rightarrow p_i$, where a, b are angles defining an arc on $\mathcal{C}$, $0 \leq a < b \leq 2\pi$, and $p_i \in \mathcal{D}$ is the object which is closest to every point of arc $[a, b]$ than any other object $p_j \in \mathcal{D}$. The $\mathcal{C}k$NN-Circ pseudocode is shown in Figure 3.8.

In the example of Figure 3.6a, let $p_1 \in \mathcal{D}$ be the first object encountered by the algorithm. Since SL is initially empty, p_1 is closest to the entire $\mathcal{C}$. Without loss of generality, we pick two points $s_0, s_0' \in \mathcal{C}$, where $\hat{s_0} = 0$ and $\hat{s_0'} = 2\pi$ (i.e., they are the same point), and insert the mapping $[\hat{s_0}, \hat{s_0'}] \rightarrow p_1$ into SL (line 2 of the pseudocode). For each subsequent point $p \in \mathcal{D}$, the algorithm traverses SL (line 4) and examines all existing mappings $[a, b] \rightarrow q$. There are three possible cases:

Case 1: $\perp pq \cap \mathcal{C} = \varnothing$ or $\perp pq$ is tangent to $\mathcal{C}$ (lines 5-6). This case is exemplified[2] in Figure 3.6b. The only existing mapping is $[\hat{s_0}, \hat{s_0'}] \rightarrow p_1$, and p_2 is processed next. Any point on the right-hand side of $\perp p_1 p_2$, is closer to p_1. Therefore, the entire $\mathcal{C}$ is closer to p_1 than to p_2. Since the mapping to p_1 already exists, there is no change in SL. Furthermore, even if there

[2]For simplicity, all objects are shown outside $\mathcal{C}$. However, the algorithm also works for objects inside $\mathcal{C}$.

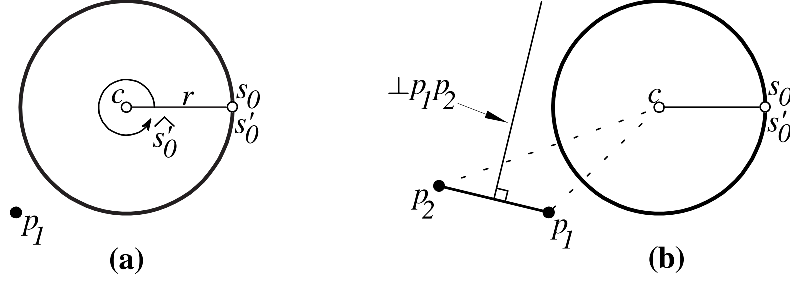


Figure 3.6: $\mathcal{C}k$NN example: perpendicular bisector does not intersect $\mathcal{C}$

were more mappings inside SL, it would not be necessary to compare with p_2, since p_1 covers p_2. On the other hand, if p_2 was at the right-hand side (and p_1 on the left), then p_2 would be closer to $\mathcal{C}$ than p_1. In this case, the algorithm would remove the $[\hat{s_0}, \hat{s'_0}] \rightarrow p_1$ mapping from SL and add a new one $[\hat{s_0}, \hat{s'_0}] \rightarrow p_2$ (line 6).

Case 2: $\perp pq \cap \mathcal{C} = \{s_0, s_1\}$ and either $\hat{s_0} \in [a, b]$ or $\hat{s_1} \in [a, b]$ (lines 12-14). This case is illustrated in Figure 3.7a: both p_1 and p_2 have already been processed, and there are two mappings in SL: $[\hat{s_1}, \hat{s'_1}] \rightarrow p_1$ and $[\hat{s'_1}, \hat{s_1}] \rightarrow p_2$. Let p_3 be the next object to be processed. p_3 is compared against the existing mappings. For the first one (i.e., $[\hat{s_1}, \hat{s'_1}] \rightarrow p_1$), $\perp p_1p_3$ intersects $\mathcal{C}$ at s_2 and s'_2. Note that $\hat{s'_2} \notin [\hat{s_1}, \hat{s'_1}]$, so it is not considered further. On the other hand, $\hat{s_2} \in [\hat{s_1}, \hat{s'_1}]$ and p_3 is closer to s_1 than p_1. Therefore (line 13), the arc is split into two parts $[\hat{s_1}, \hat{s_2}]$ and $[\hat{s_2}, \hat{s'_1}]$, which are assigned to p_3 and p_1, respectively. Similarly, for the second mapping (i.e., $[\hat{s'_1}, \hat{s_1}] \rightarrow p_2$), $\perp p_2p_3$ intersects $\mathcal{C}$ at s_3, s'_3. Only $\hat{s_3} \in [\hat{s'_1}, \hat{s_1}]$, so the arc is split into $[\hat{s'_1}, \hat{s_3}]$ and $[\hat{s_3}, \hat{s_1}]$, which are assigned to p_2 and p_3, respectively. After updating, $SL = \{[\hat{s_2}, \hat{s'_1}] \rightarrow p_1, [\hat{s'_1}, \hat{s_3}] \rightarrow p_2, [\hat{s_3}, \hat{s_1}] \rightarrow p_3, [\hat{s_1}, \hat{s_2}] \rightarrow p_3\}$. The last two mappings can be combined (i.e., $[\hat{s_3}, \hat{s_2}] \rightarrow p_3$) since they are consecutive and are mapped to the same object.

Case 3: $\perp pq \cap \mathcal{C} = \{s_0, s_1\}$ and both $\hat{s_0}, \hat{s_1} \in [a, b]$ (lines 9-11). This case is illustrated in Figure 3.7b: again, both p_1 and p_2 have already been processed, and $SL = \{[\hat{s'_1}, \hat{s_1}] \rightarrow p_1, [\hat{s_1}, \hat{s'_1}] \rightarrow p_2\}$. Next, p_3 is compared to the first mapping of SL. Note that $\perp p_1p_3$ intersects $\mathcal{C}$ at s'_2, s_2 and both $\hat{s'_2}, \hat{s_2} \in [\hat{s'_1}, \hat{s_1}]$. Therefore (line 10), the arc is split into three

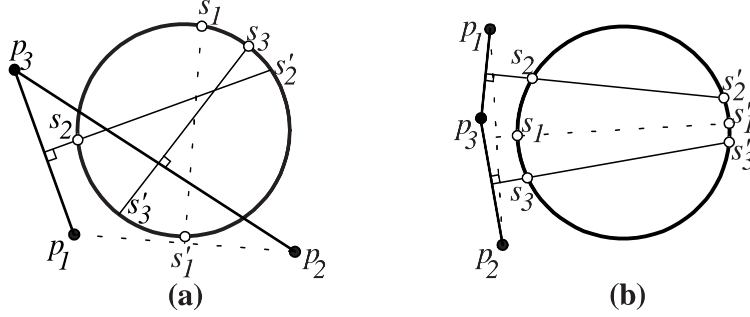


Figure 3.7: The perpendicular bisector intersects $\mathcal{C}$

parts and since p_3 is closer to s'_1 than p_1 the corresponding mappings are: $[\hat{s'_1}, \hat{s'_2}] \rightarrow p_3, [\hat{s'_2}, \hat{s_2}] \rightarrow p_1, [\hat{s_2}, \hat{s_1}] \rightarrow p_3$. Similarly, after considering $\perp p_2p_3$, $[\hat{s_1}, \hat{s'_1}]$ is also split into three parts. Finally, after combining the consecutive mappings, $SL = \{[\hat{s'_2}, \hat{s_2}] \rightarrow p_1, [\hat{s_2}, \hat{s_3}] \rightarrow p_3, [\hat{s_3}, \hat{s'_3}] \rightarrow p_2, [\hat{s'_3}, \hat{s'_2}] \rightarrow p_3\}$.

For simplicity, the pseudocode of Figure 3.8 computes only the 1-NNs. To compute the kNNs, instead of a single object, the arcs in our implementation are mapped to an ordered list of k objects: $[a, b] \rightarrow (p_1, \ldots, p_k)$, where p_1 is the nearest neighbor of arc $[a, b]$, p_2 is the second NN of arc $[a, b]$, etc. The procedure is called for each position i ($1 \leq i \leq k$) of the ordered list. In the ith call, if an object $p \in \mathcal{D}$ already exists in position j ($1 \leq j \leq i-1$), then p is not considered for that mapping. Also, if an arc is split, the objects in positions $1 \ldots i-1$ (i.e. the $i-1$ nearest neighbors found already) are not altered. The worst case complexity of $\mathcal{C}k$NN is $O(|\mathcal{D}|^k)$, since any object may cause an arc split. In practice, however, the algorithm is faster, because the objects which are far away from $\mathcal{C}$ do not cause splits.

3.5.2 R-trees and $\mathcal{C}k$NN

In order to use the $\mathcal{C}k$NN algorithm with an R-tree, we employ a branch-and-bound heuristic. Starting from the root, the R-tree is traversed either in Depth-First or in Best-First [60] manner. When a leaf entry (i.e., object) p is encountered, the $\mathcal{C}k$NN algorithm is used to check whether p is closer to $\mathcal{C}$ than any of the objects in the current mappings (i.e., p is a *qualifying object*) and updates SL accordingly. For an intermediate entry E we avoid

$\mathcal{C}k$NN-Circ($\mathcal{D}$: the set of objects)

1. **for** every object $p \in \mathcal{D}$ **do**
2. **if** $SL = \varnothing$ **then** $SL := \{[0, 2\pi] \to p\}$
3. **else**
4. **for** every interval $\varphi \equiv [a, b] \to q, \varphi \in SL$ **do**
5. **if** $\perp pq \cap \mathcal{C} = \varnothing$ **or** $\perp pq$ is tangent to $\mathcal{C}$ **then**
6. **if** $|p\mathcal{C}| < |q\mathcal{C}|$ **then** $SL := (SL - \varphi) \cup \{[a, b] \to p\}$
   else break
7. **else**
8. **let** s_0, s_1 be two points such that $\perp pq \cap \mathcal{C} = \{s_0, s_1\}$
9. **if** $\hat{s_0} \in [a, b]$ **and** $\hat{s_1} \in [a, b]$ **then**
   // Assume $\hat{s_0} < \hat{s_1}$ (the other case is symmetric)
10. **if** $|p\mathcal{C}_a| < |q\mathcal{C}_a|$ **then** $SL := (SL - \varphi) \cup$
    $\cup \{[a, \hat{s_0}] \to p, [\hat{s_0}, \hat{s_1}] \to q, [\hat{s_1}, b] \to p\}$
    // $\mathcal{C}_a$, $\mathcal{C}_b$ are the endpoints of arc $[a, b]$
11. **else** $SL := (SL - \varphi) \cup$
    $\cup \{[a, \hat{s_0}] \to q, [\hat{s_0}, \hat{s_1}] \to p, [\hat{s_1}, b] \to q\}$
12. **else if** $\hat{s_0} \in [a, b]$ **or** $\hat{s_1} \in [a, b]$ **then**
    // Let only $\hat{s_0} \in [a, b]$ ($\hat{s_1} \in [a, b]$ is symmetric)
13. **if** $|p\mathcal{C}_a| < |q\mathcal{C}_a|$ **then** $SL := (SL - \varphi) \cup$
    $\cup \{[a, \hat{s_0}] \to p, [\hat{s_0}, b] \to q\}$
14. **else** $SL := (SL - \varphi) \cup \{[a, \hat{s_0}] \to q, [\hat{s_0}, b] \to p\}$
15. **else if** $|p\mathcal{C}_a| < |q\mathcal{C}_a|$ **then**
    $SL := (SL - \varphi) \cup \{[a, b] \to p\}$
16. **return** SL

$\mathcal{C}k$NN($\mathcal{D}$: the set of objects)

1. **call** $\mathcal{C}k$NN-Circ($\mathcal{D}$)
2. **return** $\{p : p \in \mathcal{D} \wedge p \text{ is inside } \mathcal{C}\} \cup$
   $\cup \{p : p \text{ belongs to a mapping of } SL\}$

Figure 3.8: Find the 1-NNs of a circular range $\mathcal{C}$

visiting its subtree if it is impossible to contain any qualifying object.

Figure 3.9 presents an example where p_1 and p_2 are the current 1-NNs of $\mathcal{C}$. Next, an entry E from an intermediate node of the R-tree is encountered. We observe the following:

Lemma 3.5. *Let MBR_E be an axis-parallel MBR and let* st *be the side which is closest to circle $\mathcal{C}$. If* st *does not contain any of the kNNs of $\mathcal{C}$, then MBR_E cannot contain any kNN.* □

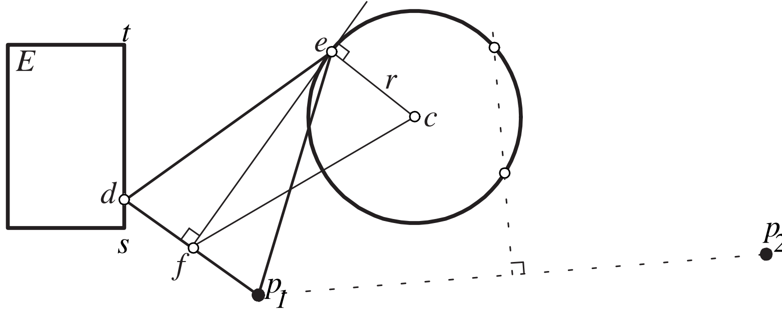


Figure 3.9: Check if E may contain qualifying objects

The proof is straight-forward, since any point in the MBR will be further away from $\mathcal{C}$ than the closest point on st. In our example, the right side st of E is closer to $\mathcal{C}$. Assume there is a point d on st, such that the perpendicular bisector $\perp dp_1$ is tangent to $\mathcal{C}$, and let $e \equiv \perp dp_1 \cap \mathcal{C}$. Then we get the following system of equations[3]:

$$\begin{cases} |ce| = r \\ |p_1 e| = |de| \\ |p_1 e|^2 - |p_1 f|^2 = |cf|^2 - r^2 \end{cases} \quad (3.2)$$

The first equation is derived from the fact that $e \in \mathcal{C}$, while the second one is because the distance from any point on $\perp dp_1$ to d and p_1 is equal. The third equation results from the application of the Pythagorean theorem on the orthogonal triangles p_1fe and fec which have a common side ef. After substituting the points with their Cartesian coordinates, we get the following system (note that $x_f = \frac{x_d + x_{p1}}{2}, y_f = \frac{y_d + y_{p1}}{2}$, since f is the middle of dp_1):

[3]If a different side of E is closer to $\mathcal{C}$, the equations are modified accordingly.

$$\begin{cases} (x_e - x_c)^2 + (y_e - y_c)^2 = r^2 \\ (x_d - x_e)^2 + (y_d - y_e)^2 = (x_{p1} - x_e)^2 + (y_{p1} - y_e)^2 \\ (x_{p1} - x_e)^2 + (y_{p1} - y_e)^2 - \frac{(x_d - x_{p1})^2 + (y_d - y_{p1})^2}{4} = \\ = \left(\frac{x_d + x_{p1}}{2} - x_c\right)^2 + \left(\frac{y_d + y_{p1}}{2} - y_c\right)^2 - r^2 \end{cases}$$

There are three equations and three unknowns: x_e, y_e, y_d. If there is a real solution to this system, under the condition $(x_d, y_d) \in st$, then there *may* be a qualifying object inside the subtree of E. Else all objects in E are further away from $\mathcal{C}$ than the current objects in SL, so the subtree under E can be pruned.

Solving this system, however, is slow (in the order of 100's of msec in an average computer); given that an entry E must be checked against many objects, the running time is prohibitively long. Therefore, in our implementation, we use the $\mathcal{R}k$NN algorithm to traverse the R-tree and employ the $\mathcal{C}k$NN algorithm only for the objects at the leaf-level. Our strategy is based on the following observation:

Lemma 3.6. *Let $\mathcal{C}$ be a circle, MER the maximum enclosed axis-parallel rectangle of $\mathcal{C}$ and S the set of kNNs of MER's perimeter. Let p_i be an object, such that p_i is inside MER and $p_i \notin S$. Then p_i cannot be a kNN for any point of $\mathcal{C}$.*

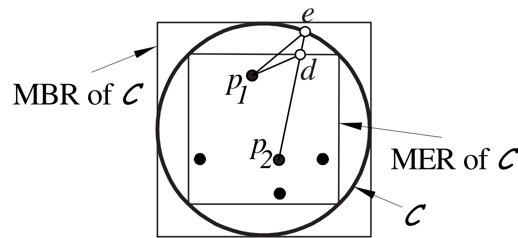


Figure 3.10: The MBR and the MER of $\mathcal{C}$

Proof. Assume the lemma does not hold. Figure 3.10 shows an example where p_2 is inside MER and $p_2 \notin S$. Assume that p_2 is the NN of point $e \in \mathcal{C}$. Let d be the point where the line segment p_2e intersects the perimeter of MER, and p_1 be the object which is the NN of d. It follows from our

hypothesis that: $|p_2e| < |p_1e|$. Using the triangular inequality, we get: $|p_2d| + |de| < |p_1d| + |de| \Rightarrow |p_2d| < |p_1d|$ which is a contradiction, since p_1 is the NN of d. Therefore, the lemma holds. □

We construct the Minimum Bounding Rectangle[4] MBR and the Maximum Enclosed Rectangle MER of $\mathcal{C}$ (the side-length of MER is $\sqrt{2}r$). Conceptually, our implementation works in three steps:

1. Use the $\mathcal{R}k$NN algorithm to find the set S_1 of kNNs of MBR (including all the objects inside MBR). Recall that S_1 is a superset of the kNNs of any point inside MBR; therefore, it contains all the kNNs of $\mathcal{C}$.

2. Use CNN (see Section 2.4) to find the set S_2 of kNNs of *only* the perimeter of MER. Use Lemma 3.6 and S_2 to prune objects from S_1.

3. Call the $\mathcal{C}k$NN algorithm with the objects remaining in S_1.

In practice, these steps can be combined. In a single traversal of the R-tree, steps (1) and (2) can be used at the intermediate levels to prune the tree and step (3) is applied on the leaf-level objects.

[4]For a set of users $U_{1...n}$, the MBR of $\mathcal{C}$ is not the same as their corresponding anonymizing rectangle $\mathcal{R}$.

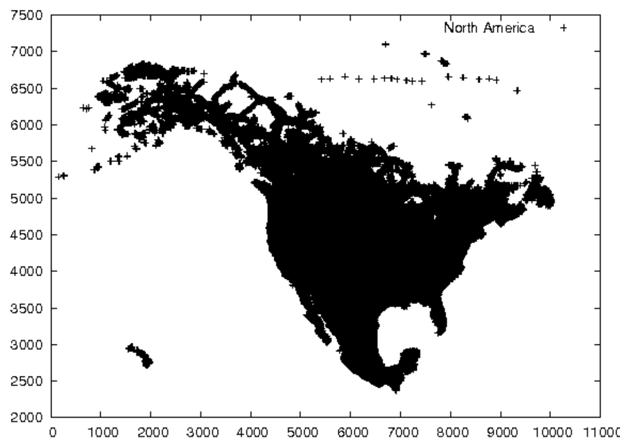


Figure 3.11: North-America (NA) dataset

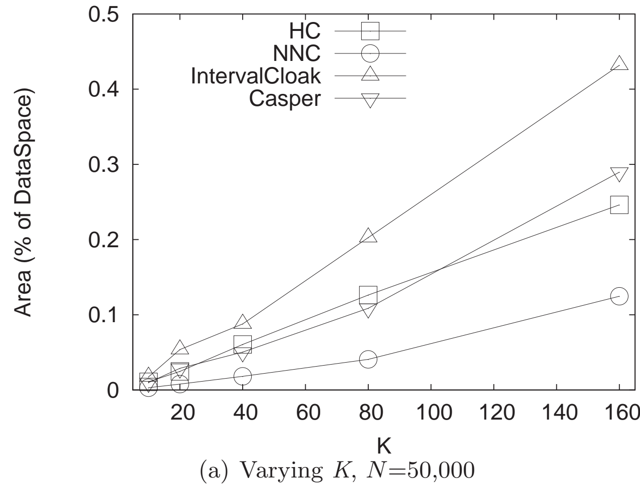
3.6 Experimental Evaluation

This section evaluates our proposed anonymization and query processing algorithms. We implemented prototypes for both the anonymizer and the LBS using C++. All experiments were executed on an Intel Xeon 2.8GHz machine with 2.5GB of RAM and Linux OS. Our workload for user positions and landmarks/points of interest consists of the NA dataset [61], which contains 569K locations on the North-American continent (Figure 3.11). Performance is measured in terms of CPU time, I/O time and communication cost. At the anonymizer we employed main-memory structures, therefore we measured only the CPU time. At the LBS, we used an R*-Tree and measured the total time (i.e., I/O and CPU time); in all experiments we maintained a cache with size equal to 10% of the corresponding R*-Tree. The communication cost was measured in terms of number of candidates sent from the LBS back to the anonymizer.

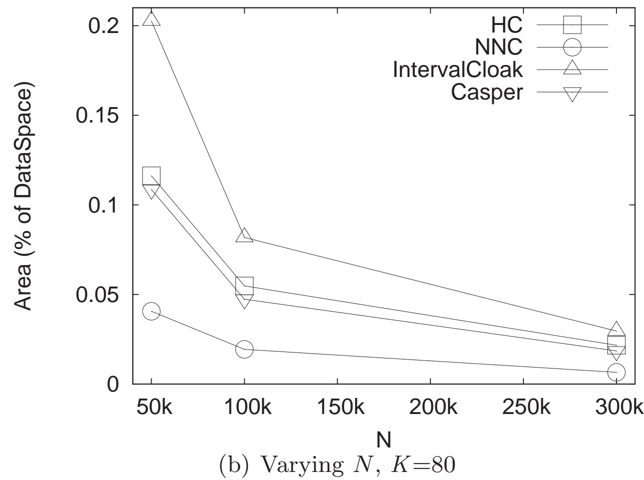
In the following, Section 3.6.1 focuses on cloaking algorithms at the anonymizer, whereas Section 3.6.2 evaluates query processing at the LBS.

3.6.1 Anonymizer Evaluation

We compare the proposed *Nearest Neighbor Cloak* (*NNC*) and *Hilbert Cloak* (*HC*) against *Casper* and *Interval Cloak* (*IC*). The first experiment measures the area of rectangular K-ASRs. Recall that we wish to minimize the



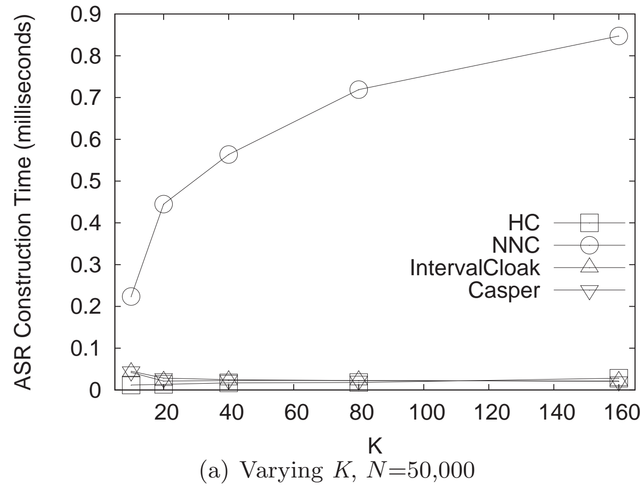
(a) Varying K, N=50,000



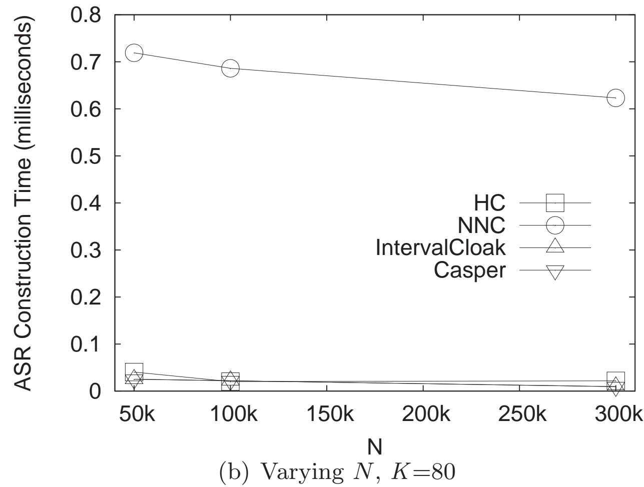
(b) Varying N, K=80

Figure 3.12: Area of rectangular K-ASR

ASR area, since it affects the processing time at the LBS and the communication cost between the LBS and the anonymizer. First, we fix the number of users $N = 50,000$ and vary the degree of anonymity K. The K-ASR area is expressed as a percentage of the entire data space. We generated $1,000$ queries originating at random users. Figure 3.12a shows the average area per query. Clearly *IC* is the worst algorithm, whereas *NNC* is the best. *HC* and *Casper* exhibit similar behavior to each other. All algorithms scale linearly with K in terms of ASR area. Figure 3.12b, shows the K-ASR area for $K = 80$ and varying N. Since the extent of the data space remains constant, an increase in user population translates to higher user density, hence



(a) Varying K, N=50,000



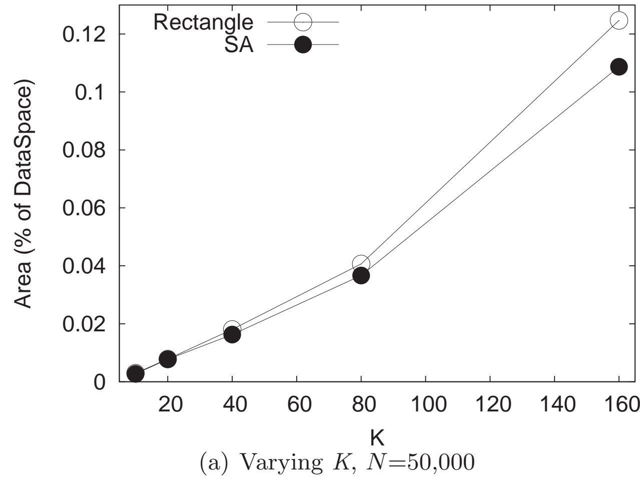
(b) Varying N, K=80

Figure 3.13: K-ASR generation time

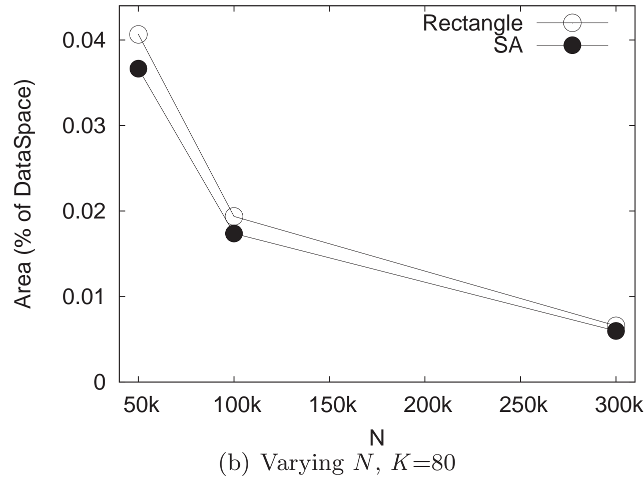
reduced K-ASR size for all methods. The relative performance among the algorithms remains the same. Observe that *HC* and *Casper* outperform *IC*, and generate ASRs with roughly twice the area of *NNC*.

Figure 3.13 shows the average ASR generation time (in milliseconds) for varying K and N. *HC*, *IC* and *Casper* behave similarly. *NNC*, on the other hand, has a significantly larger generation time, due to the more costly nearest-neighbor search. Nevertheless, we will show in the following that *NNC* is best in terms of overhead at the LBS.

So far, we focused on rectangular K-ASRs. However, depending on the user distribution, circular K-ASRs may have smaller size. Here we adopt



(a) Varying K, N=50,000



(b) Varying N, K=80

Figure 3.14: Rectangular vs SA K-ASR, *Nearest Neighbor Cloak*

a simple optimization: first we identify the set of users which belong to a K-ASR. Then we calculate the minimum bounding rectangle $\mathcal{R}$ and the minimum enclosing circle $\mathcal{C}$ of the K-ASR, and select the shape with the *smallest area*. We call this method SA. *NNC* is more suitable to be combined with SA, since the nearest neighbor search tends to identify circular clusters of users. Figures 3.14a and 3.14b compare the rectangle-only approach against the SA optimization for varying K and N, respectively. SA manages to reduce the K-ASR area by up to 15%.

Finally, we measure the anonymity strength of the above-mentioned al-

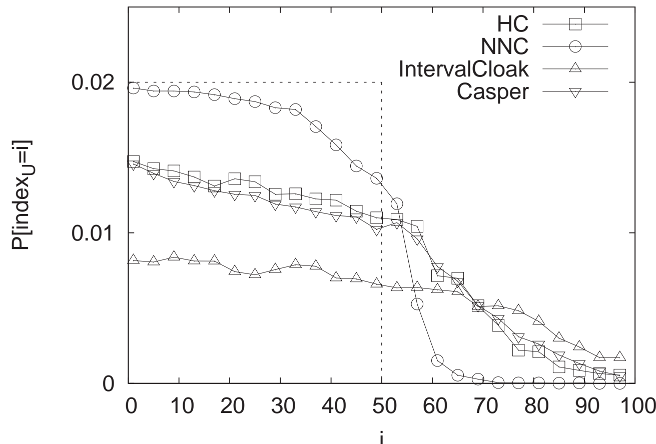


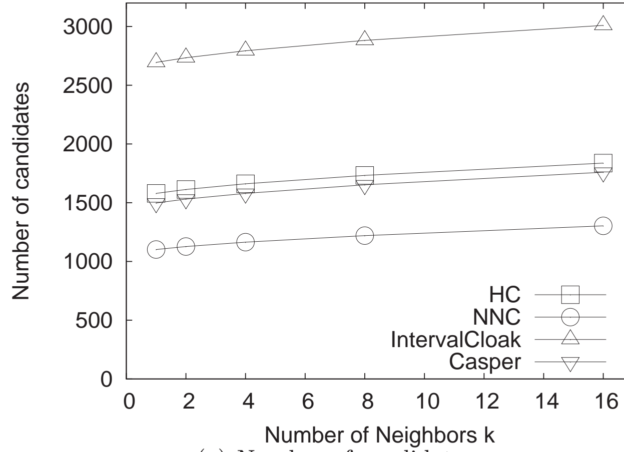
Figure 3.15: *center-of-ASR* attack, $K = 50$

gorithms against the *center-of-ASR* attack[5]. We consider a workload of 1000 queries, originating at a set of random users, with $K = 50$. Figure 3.15 shows the probability $P[index_U = i]$ (the experiment is similar to that of Section 2.2). Recall that $index_U = 1$ means that user U is the closest to the center of the K-ASR. Furthermore, the dashed line corresponds to the distribution of $index_U$ for the ideal anonymization technique. All studied algorithms preserve privacy in the case of the *center-of-ASR* attack. *NNC* is close to the ideal distribution and there are few cases where the K-ASR encloses more than K users, which explains the relatively small ASR size observed in the previous experiments. *HC* and *Casper* exhibit similar behavior to each other, but include a larger number of redundant users inside the K-ASR, compared to *NNC*; this is why $P[index_U = i] > 0$ for $i > K$. However, they are both better than *IC*.

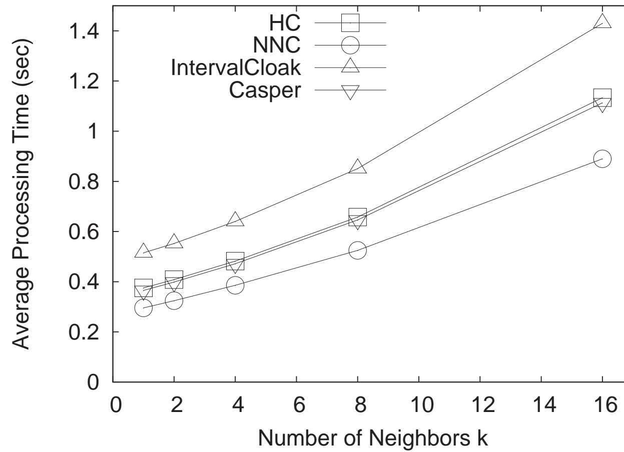
3.6.2 Location-Based Service Evaluation

For this experiment, we generate $1,000$ queries originating at random users. The corresponding K-ASRs are sent to the LBS and the queries are executed against the entire NA dataset, which is indexed by an R*-tree. For all K-ASR generation techniques, we compare the average processing time (i.e, CPU plus I/O time) per query, and the size of the candidate set. The

[5]Although we formally proved that *Hilbert Cloak* guarantees location anonymity, we include this experiment for illustration purposes.



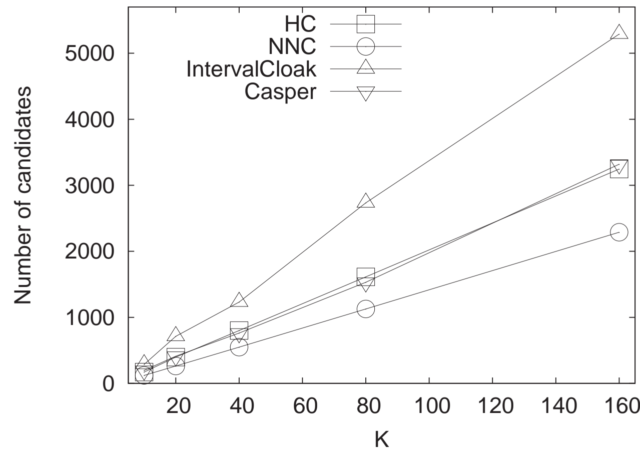
(a) Number of candidates



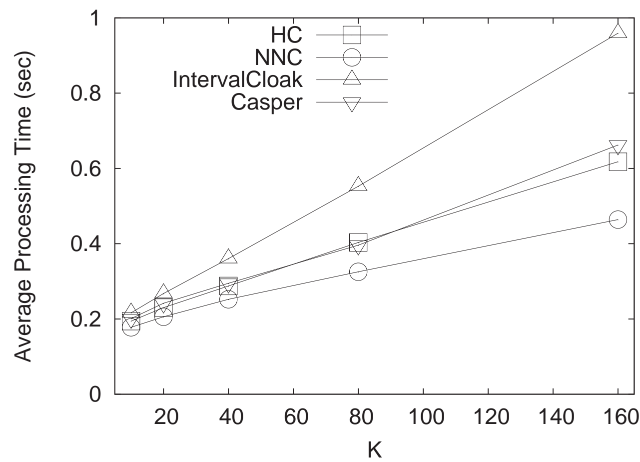
(b) Avg. processing time (sec)

Figure 3.16: kNN queries, varying k, $N = 50,000$, $K = 80$

latter is a superset of the actual result, and it reflects the communication cost between the LBS and the anonymizer. First, we focus on kNN queries. Figure 3.16 shows the performance for varying number of nearest neighbors k. *NNC* generates a significantly lower number of candidates compared to the other techniques. This is expected, since the sizes of the corresponding K-ASRs are also smaller. *HC* and *Casper* generate up to 50% more candidates than *NNC*. However, they both outperform *IC* by a large margin. In terms of processing time, *NNC* is the fastest, with *HC* and *Casper* considerably better than *IC*.



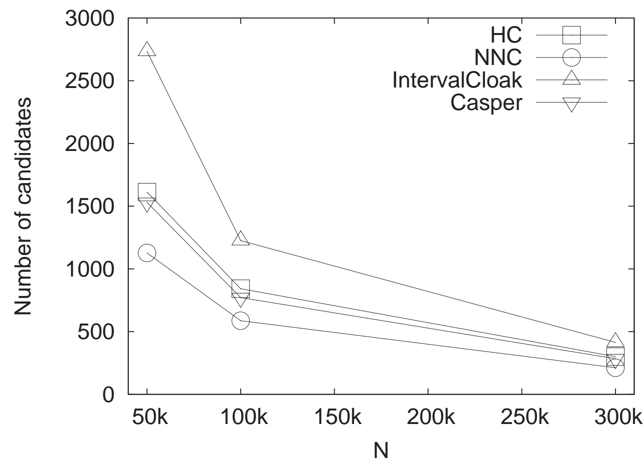
(a) Number of candidates



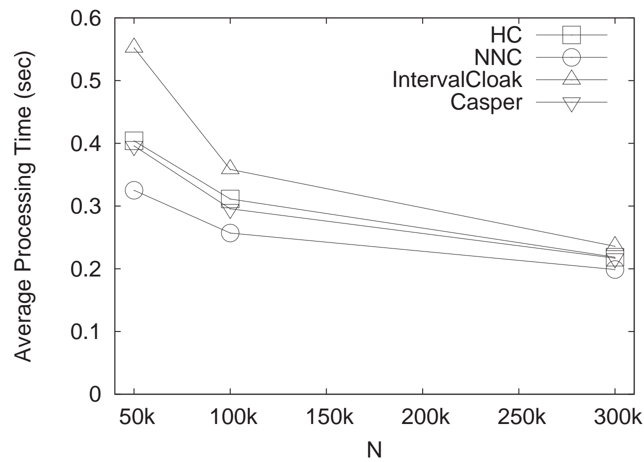
(b) Avg. processing time (sec)

Figure 3.17: kNN queries, varying K, $k = 2$ neighbors, $N = 50,000$

In Figure 3.17 we fix the number of neighbors $k = 2$ and vary the degree of anonymity K. Again, *NNC* performs best, followed by *HC* and *Casper*. The difference is more significant for larger K values, as the average size of the K-ASR increases.



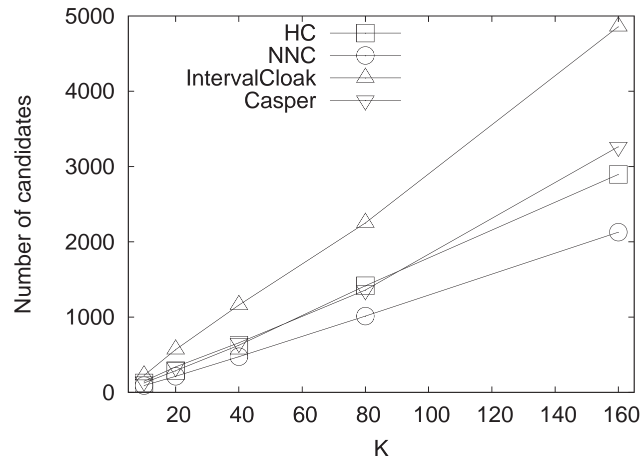
(a) Number of candidates



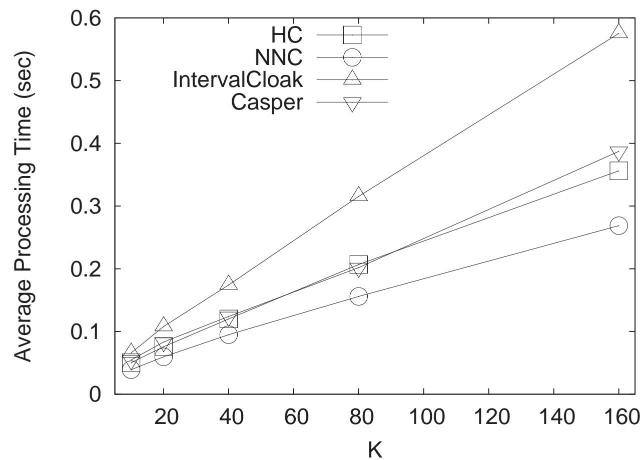
(b) Avg. processing time (sec)

Figure 3.18: kNN queries, varying N, $k = 2$, $K = 80$

Figure 3.18 shows the number of candidates and processing time for varying N. Note that more users lead to higher density, thus smaller K-ASRs. Consequently, the number of candidates and the average processing time decrease with N.



(a) Number of candidates

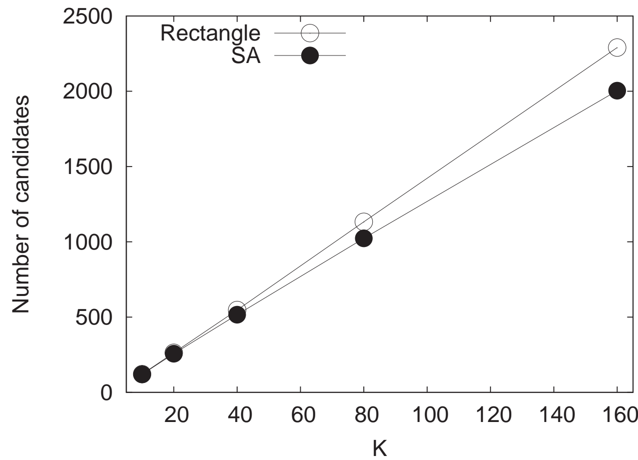


(b) Avg. processing time (sec)

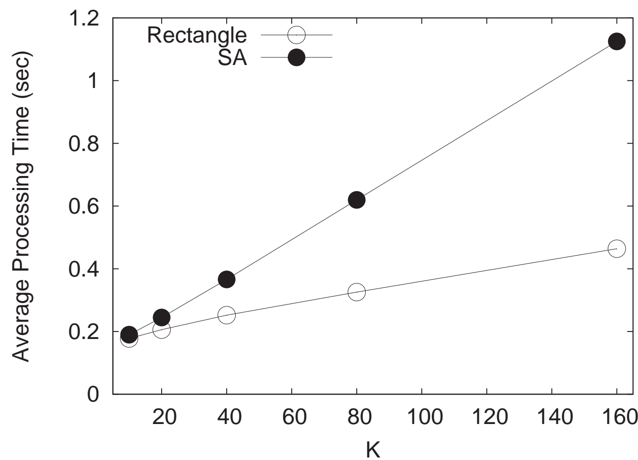
Figure 3.19: Range queries, $N = 50,000$, varying K

We also evaluated the performance of the four techniques for range queries. The results are presented in Figure 3.19 for varying K and $N = 50,000$. Again, we observe a significant advantage of *NNC* over the other techniques, while *HC* and *Casper* outperform *IC* in terms of both processing cost and candidate set size. The trends for varying N are similar.

The previous results were obtained for rectangular K-ASRs. We also



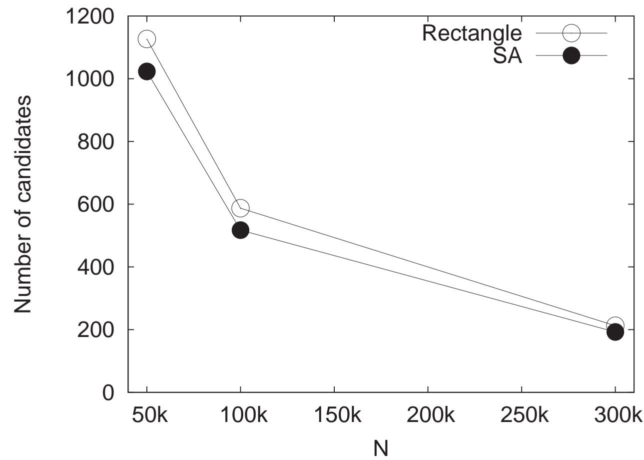
(a) Number of candidates



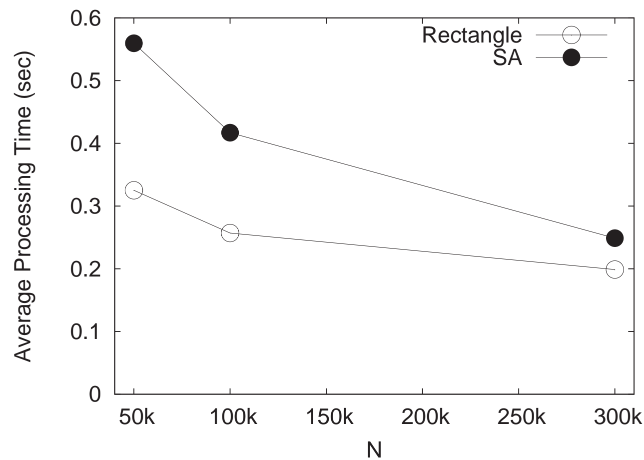
(b) Avg. processing time (sec)

Figure 3.20: *NNC*, rectangular vs SA K-ASR, $k = 2$, $N = 50,000$

investigated the effect of the SA (i.e., smallest area) optimization on query processing. For a given K-ASR, if SA generates a circular range $\mathcal{C}$, we employ $\mathcal{C}k$NN to execute the corresponding kNN query. For our workload, SA generated circular ranges for around 45% of the K-ASRs when K was small, and up to 90% for large values of K. Figure 3.20 compares SA against the rectangles-only approach for $k = 2$ neighbors and varying K. SA re-



(a) Number of candidates



(b) Avg. processing time (sec)

Figure 3.21: *NNC*, rectangular vs SA K-ASR, $k = 2$, $K = 80$

duces the number of candidates by up to 18%, compared to the rectangular K-ASR. The tradeoff is the increased processing time. The same relative performance is observed in Figure 3.21, where we vary N.

3.7 Discussion

Our proposed anonymization algorithms are clearly superior compared to the existing approaches. The *HC* algorithm provides privacy guarantees under all user and query distributions, and its overhead in terms of ASR generation time, query processing time and communication cost is similar to *Casper*, the most recent and most efficient technique. On the other hand, *NNC* clearly outperforms *Casper* in terms of overhead at the LBS, while offering similar anonymity strength.

The LBS is likely to maintain huge volumes of data and disk-based data structures, while the anonymizer typically uses memory-based data structures. For this reason, the query overhead at the LBS is considerably larger than at the anonymizer (observe that time is measured in milliseconds in Figure 3.13 instead of seconds in Figure 3.16b). Under these circumstances, the reduced LBS processing cost offers *NNC* an important performance advantage, despite its increased K-ASR generation time.

The choice between *Hilbert Cloak* and *Nearest Neighbor Cloak* involves a clear trade-off between privacy guarantees on one hand, and processing overhead on the other. If provable anonymity guarantees are required, *Hilbert Cloak* is the only option. Nevertheless, *Nearest Neighbor Cloak* also achieves strong anonymity for most of the cases, and may be acceptable for applications where outliers do not constitute an anonymity threat (e.g., very frequent user movement) and efficiency is crucial.

Finally, there is a tradeoff between rectangular-only K-ASRs and the SA optimization. The cost of $\mathcal{C}k$NN at the LBS is higher than $\mathcal{R}k$NN. However, $\mathcal{C}k$NN reduces the number of candidates. Therefore, $\mathcal{C}k$NN is preferable if the communication cost is more important than the processing cost at the LBS. In practice, this happens if a single anonymizer sends queries to several LBSs. In this case the bandwidth of the single anonymizer is shared among all connections. Thus, it is important to minimize the communication cost, whereas the processing cost is distributed among the LBSs.

Chapter 4

Reciprocal Framework for SKA

4.1 Introduction

In the previous chapter, we have introduced the reciprocity concept, a sufficient condition to guarantee privacy for snapshot LBS queries. In this chapter, we propose a general framework to implement reciprocity in conjunction with a spatial index. We investigate several methods to group the set of users into reciprocal partitions, which provide a trade-off between the size of the resulting K-ASR, and the time required to generate it.

We also extend the reciprocity principle to address the scenario where an attacker has additional background knowledge on the frequencies of generating queries for each LBS user.

4.2 Algorithm for Reciprocal Cloaking

We consider the architecture of Figure 1.3, 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 required degree of anonymity K, which ranges between 1 (no privacy requirements) and the user cardinality (maximum privacy).

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 K-ASR is the area that encloses all elements of the AS. Figure 4.1 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 term user and point 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 as N such that $0 < |N'| < K$, because N' may contain a user U' whose AS includes U.

Reciprocal (query issuer U, anonymity degree K, node N)

    //initially N is the leaf node containing U

1. **while** ($\exists$ non-empty node at the same level as N with $< K$ users)
2. $N = N.parent$ //bottom-up traversal
3. **while** (N not leaf) **and**
   ($\forall$ child of N is either empty or contains $\geq K$ users)
4. $N =$ child of N that contains U //top-down traversal
5. $ASR = Partition(U, K, N)$

Figure 4.1: Reciprocal Cloaking

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[1]. Let PN be the node in AN where the top-down traversal stops. PN includes U in its sub-tree and

[1] While bottom-up traversal considers the cardinality of all nodes at a level, top-down only takes into account the cardinalities at a single path

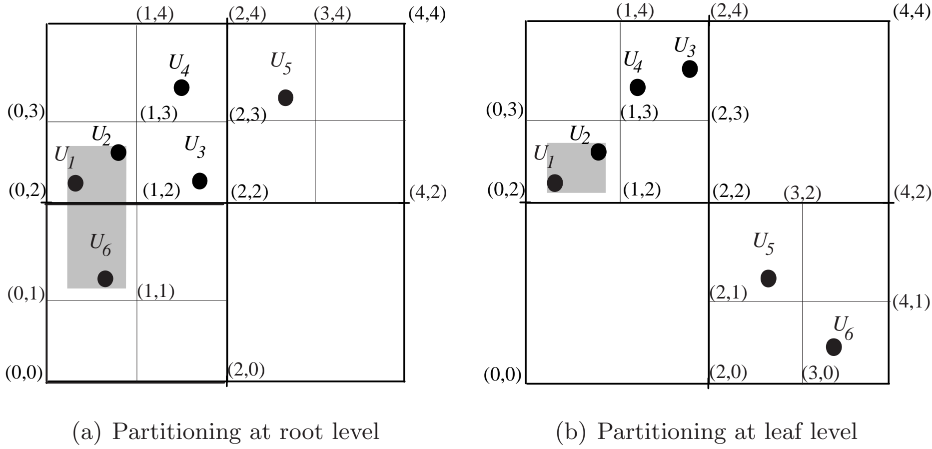
some of its child nodes have fewer than K points. PN is called the partition node, and corresponds to the lowest ancestor of U where we can achieve reciprocity. This is because all nodes in the sub-tree of AN and at the level of PN or above, contain at least K points, and thus can generate ASRs without using any points in PN.

PN may contain numerous ($\gg K$) points, which is likely to yield very large ASRs. The *Partition* routine (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. As we will discuss in Section 4.3, several partitioning methods can be used, 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, i.e., each user in the node will generate exactly 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 a pre-defined anonymity resolution [49]. 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



(a) Partitioning at root level

(b) Partitioning at leaf level

Figure 4.2: Partitioning with a Quad-tree

should be secure even if the attacker has complete knowledge of all the user positions.

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 4.3.

Example 4.1 (Quad-tree Cloak)**.** Figure 4.2a 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 $\langle(0,2),(1,3)\rangle$ 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. $\langle(0,0),(2,2)\rangle$) with fewer than K users. Finally, QC reaches the root and sets $AN = PN = \langle(0,0),(4,4)\rangle$. The same partition node is obtained for all users given $K = 2$. $\square$

In the above query, PN contains 6 points, although only 2 are required 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 Figure 4.2a. Figure 4.2b illustrates a second example, which in addition involves the top-down traversal phase. Given again a query with $K = 2$ from U_1, the bottom-up traversal stops at level 2 with $AN = \langle(0,2),(2,4)\rangle$ because all non-empty cells at this level have at least 2 points. Furthermore, both non-empty children of AN, $\langle(0,2),(1,3)\rangle$ and $\langle(1,3),(2,4)\rangle$, also include 2 points each. Therefore, QC descends to level 1 and sets the partition node to $PN = \langle(0,2),(1,3)\rangle$. 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 4.2. 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. □

A simple method for guaranteeing reciprocity 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. In fact, this is similar to the relational K-anonymity (RKA) generalization techniques surveyed in Section 2.1, except that SKA requires a single group (instead of the entire anonymized table). Clearly, this approach would be inefficient because it has to access all the user locations. Furthermore, it would probably be ineffective (i.e., would lead to large ASRs), since it does not take advantage of the existing grouping of users in the index nodes.

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 indexes, and have been used extensively in spatio-temporal data warehouses [59]. Finally, note that, the location updates issued by the users can be handled automatically by the corresponding algorithms of the indices, without any effect on the anonymization process.

4.3 Partitioning Methods

Given a partition node PN, Partition (line 5 in Figure 4.1) splits the users inside the sub-tree of PN into buckets containing between K and $2K-1$ users. In the sequel we discuss alternative partitioning algorithms. Sections 4.3.1 and 4.3.2 present two novel methods with different tradeoffs in efficiency and effectiveness. Sections 4.3.3 and 4.3.4 adapt two RKA techniques to our framework. Although all techniques can be used with any spatial index, the examples assume an aggregate R*-tree (aR*-tree [59]), 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 per level i is M^i, where M is the R*-tree minimum node utilization (usually 40% of the node capacity).

4.3.1 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 (introduced in Section 3.4), GH sorts the points in LN

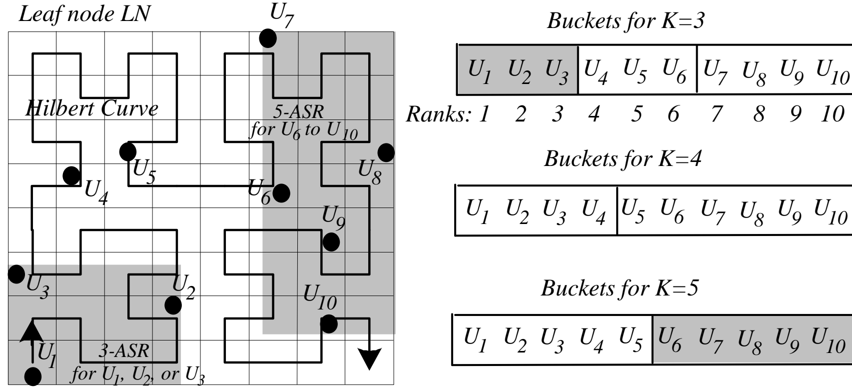


Figure 4.3: GH partitioning for (leaf) level 1

according to their Hilbert value. The Hilbert space filling curve transforms the multi-dimensional coordinates of each user U into an 1-D value $H(U)$. Figure 4.3 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. Following this approach, 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 $s = r_U - (r_U - 1) \mod K$ and $e = s + K - 1$ (unless b_U is the last bucket).

Example 4.3. Figure 4.3 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. Figure 4.3 illustrates

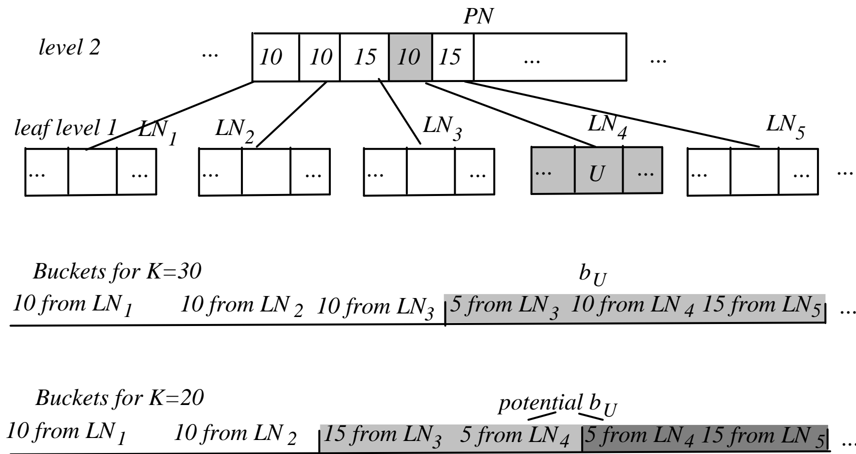


Figure 4.4: GH partitioning for level 2

another ASR (shaded rectangle at the lower-left corner) for a query with $K = 3$ originating from one of U_1 to U_3. □

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, which would be similar to using Hilbert Cloak. However, this process is not necessary since we only need b_U (and not the other buckets at this level). Figure 4.4 shows an example, assuming that the query issuer U is in leaf node LN_4. The leaves are numbered according to their Hilbert order in the parent PN; specifically, 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 for generating b_U 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 Figure 4.4, 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.

Figure 4.5 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 Figure 4.4, then r_U is the rank of U among the points of LN_4 plus the cardinalities of LN_1 to LN_3. Once b_U has been determined, all leaf nodes that contribute points to b_U participate in the ASR construction through the *merge* function. The merging process is also recursive. Specifically, 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 from the path LN to PN (i.e., one node per level) and *(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 computation 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 Figure 4.5) 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 $O(M \cdot \log_2 M \cdot \log_M K)$.

GH-partitioning(query issuer U, anonymity degree K, partition node PN)

1. $ASR = \emptyset$
2. $r_U = compute_rank(PN, 0)$
3. $s = r_U - (r_U - 1) \mod K$; $e = s + K - 1$ // extent of b_U
4. **if** $|PN| - e < K$ // b_U is the last bucket
5. $e = |PN|$; $s = e - (e \mod K) - K + 1$
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. $list$ = entries of N sorted in Hilbert order
9. **if** N is leaf node
10. $r_U = r_U +$ position of U in $list$
11. **else** // N is an internal 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. $ASR_{d-min} = min(ASR_{d-min}, E_{d-min})$
20. $ASR_{d-max} = max(ASR_{d-max}, E_{d-max})$
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. $ASR = merge(E', ASR)$

Figure 4.5: Greedy Hilbert - general method

4.3.2 Asymmetric R-tree Split (AR)

The AR partitioning method is inspired by the R*-tree construction algorithm[2], which is known to have good locality properties. A straightforward approach is to apply the R*-split [13] on the partition node, after setting the minimum node utilization to K. Specifically, R*-split first sorts all points by their x-coordinates. 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 $2K - 1$ users.

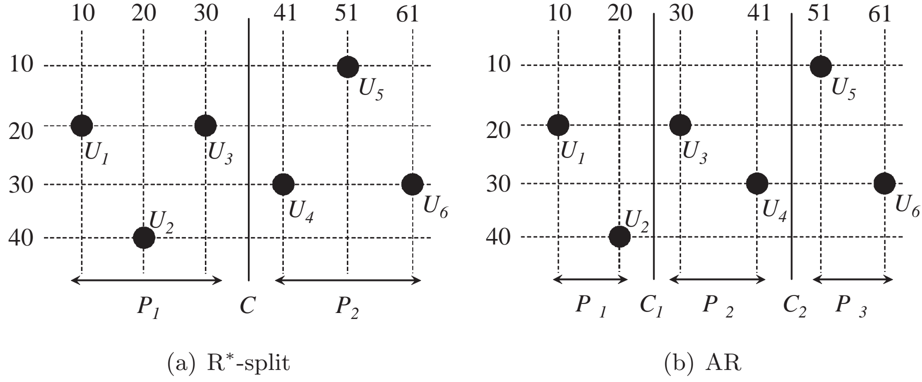
R*-split has some shortcomings with respect to the problem at hand. First it attempts to minimize factors such as perimeter and overlap of the resulting nodes, whereas we aim at minimizing the ASR area. Even if we modify the algorithm to consider only the ASR area, R*-split can still lead to fragmentation, 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 Figure 4.6a 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:

$$[ASR(P_1) + ASR(P_2)] \cdot |P_1| \cdot |P_2| \tag{4.1}$$

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.

[2]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).



(a) R*-split (b) AR

Figure 4.6: R*-tree split vs AR

Continuing the example in Figure 4.6b, 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.

Figure 4.7 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 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 $O(1 + M + M^2 + \ldots + M^\alpha)$ where $\alpha = \lceil log_M K \rceil$, 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

AR (query issuing user U, anonymity requirement K, partition node PN)
1. load all points in PN
2. $compute_ASR(U, PN)$

$compute\text{-}ASR(U,\ N)$
3. **if** $|N| < 2K$
4. **return** MBR(N)
5. $min_split_cost = \text{inf}$
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. $P_1 = list_d[1..point]$
10. $P_2 = list_d[point + 1..|list_d|]$
11. $split_cost = [ASR(P_1) + ASR(P_2)] \cdot |P1| \cdot |P2|$
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 $list_{split_dim} \leq$ split_point
17. $N' =$ points in $list_{split_dim}[1 \ldots split_point]$
18. **else** // U is in the second node of the split
19. $N' =$ points in $list_{split_dim}[split_point + 1 \ldots |listsplit_dim|]$
20. **return** $compute_ASR(U, N')$

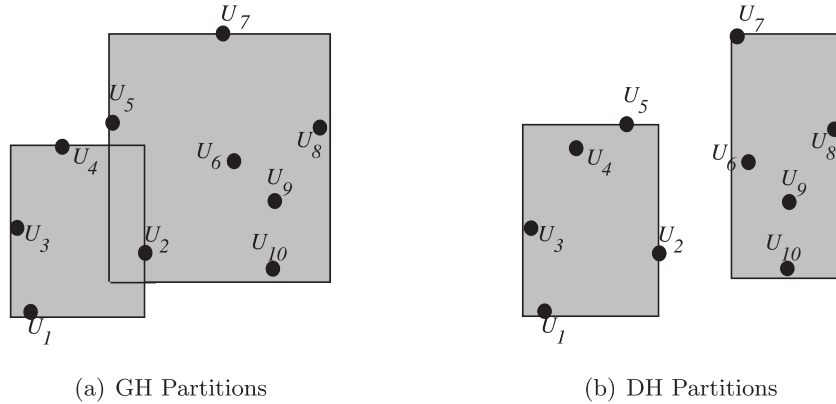
Figure 4.7: Asymmetric R-tree Split (AR)

it has less than $2K$ points. The complexity is:

$$\sum_{i=2}^{|PN|/K} (iK) \log iK = K \left[\sum_{i=2}^{|PN|/K} i(\log i + \log K) \right] = O\left(\frac{|PN^2|}{K} \log |PN| \right) \tag{4.2}$$

4.3.3 Dynamic Programming Hilbert (DH)

DH is an adaptation of the RKA method proposed in [30]. We use DH as a benchmark in our experimental evaluation, as well as a case study that



(a) GH Partitions

(b) DH Partitions

Figure 4.8: GH and DH partitions for K=4

demonstrates the versatility of our framework. DH is motivated by the fact that greedy selection may generate sub-optimal buckets. Figure 4.8a shows the GH partitioning for $K = 4$ and the leaf node contents of Figure 4.3. Observe that the MBR of the second bucket (containing U_5 to U_{10}) is rather large. The problem is caused by U_5, which would be better grouped with the points of the first bucket. DH applies dynamic programming to find the best clustering, given the Hilbert order. For instance, in Figure 4.8b DH would include U_5 in the first bucket since the small increase of its MBR is compensated by the large decrease in the ASR of the second bucket. The sum over all ASR areas multiplied by their cardinality is an estimation of effectiveness (i.e., ASR processing cost at the LBS). Hence, we aim at minimizing the total weighted ASR area:

$$WASR = \sum_{\forall ASR A} area(A) \cdot |A| \tag{4.3}$$

DH first loads all the data points of the partition node PN and sorts them. Let e be a position in the Hilbert order. $WASR_{min}(e)$ is the optimal weighted sum of ASR areas (of all buckets), when the first e points have been partitioned. $ASR(s, e)$ is the area of the MBR containing all points between positions s and e ($s < e$). The intuition behind DH is that the total weighted area of the best partitioning equals the minimum sum of *(i)* the total weighted area of the best partitioning for the first s users plus *(ii)* the ASR of the last bucket, containing the remaining $|PN| - s$ points, multiplied

by its cardinality. Formally:

$$WASR_{min}(|PN|) = \min_{|PN|-2K<s\leq|PN|-K} \left\{ \begin{array}{r} WASR_{min}(s)+ \\ (|PN|-s)\cdot ASR(s+1,|PN|) \end{array} \right\} \tag{4.4}$$

The range $|PN| - 2K < s \leq |PN| - K$ is due to the fact that each bucket must include between K and $2K - 1$ users. The expression above can be calculated by Dynamic Programming. Because each bucket can have up to $2K - 1$ users, the computation complexity of Dynamic Programming becomes $O(K \cdot |PN|)$. DH needs to load all points inside PN. Therefore, the I/O cost is $O(K)$ (i.e., the same as AR).

4.3.4 Top-Down Clustering (TD)

For completeness, our evaluation includes one more recent method from the RKA literature as a benchmark. Specifically, we adapt *Top Down* (TD), a divisive clustering-based approach that builds anonymized groups with cardinality bounded between K and $2K - 1$ [66]. The adaptation works as follows. Once the partition node PN has been determined, all points of PN are loaded and 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 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.

4.3.5 Discussion

The proposed partitioning techniques provide different tradeoffs of efficiency and effectiveness. At one end, GH (which is very localized) is fast in terms of both I/O and CPU cost, but may yield large ASRs. At the other end, AR, DH and TD are more expensive, since they have to read the entire

sub-tree of PN and perform CPU-intensive computations, but they usually yield smaller ASRs. The choice of 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 (or DH, TD) is a better choice. In Section 4.5 we experimentally evaluate these tradeoffs.

4.4 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. Assume 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 query 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 4.4 (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 $K \cdot F_{max} \leq (F_1 + F_2 + \ldots + F_N)$, 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 *Reciprocal* algorithm of Figure 4.1 remains the same, except from line 3, which changes to:

---

3. **while** (N not leaf) **and** ($\forall$ child of N is empty or has $K < F/F_{max}$)

Figure 4.9: Reciprocal Cloaking Change for Variable Frequency

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 (i.e., user) 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.

Example 4.5. Figure 4.10 illustrates the application of FQGH: 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. □

Since the split is performed with respect to frequencies, it is possible for a user to belong to more than one bucket. However, since 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

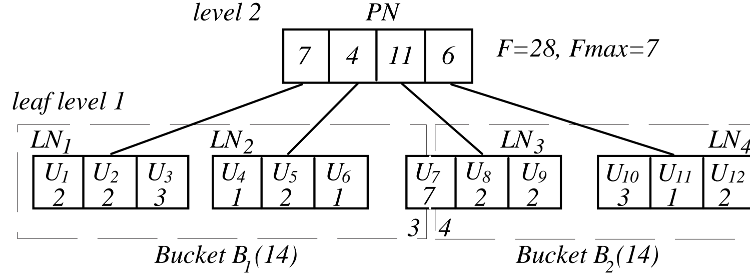


Figure 4.10: FQGH partitioning, K=2

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 Figure 4.10, 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.

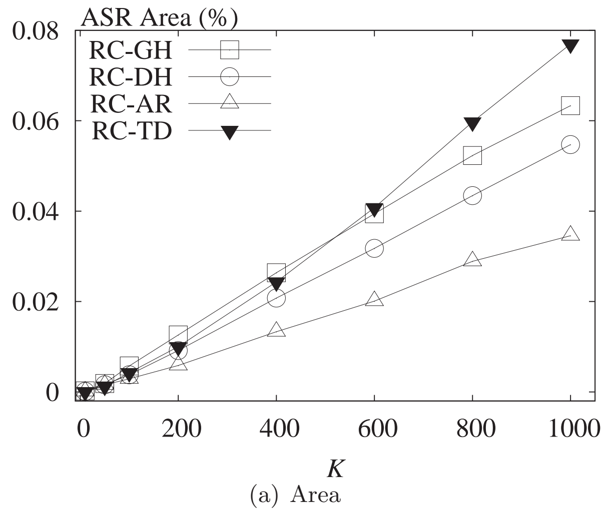
Similar to GH, FQGH only needs to access at most two leaf nodes for each query, therefore it is efficient. Furthermore, the Hilbert sorting is performed based on user locations, and it is oblivious to the query frequencies; hence, the complexity of FQGH is similar to that of GH. AR, DH and TD can be 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 subtree (e.g., AR, DH, TD) 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.

4.5 Experimental Evaluation

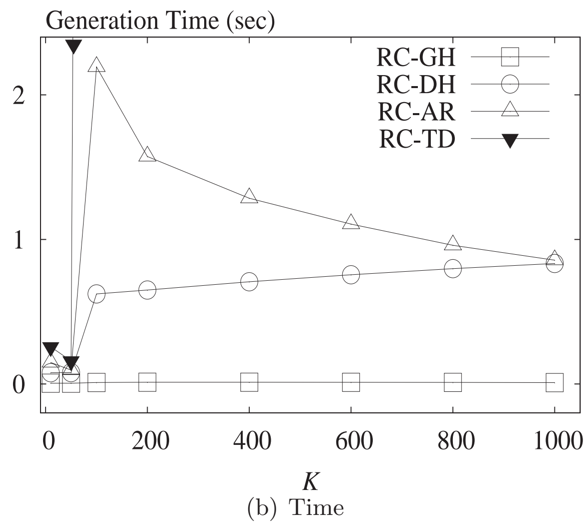
We implemented a C++ prototype of the anonymizer and deployed it on an Intel Xeon 2.8GHz machine running Linux OS. The anonymizer indexes the user locations, which are taken from the NA dataset (available at www.rtreeportal.org) containing 569k intersections of the North American road network. K ranges from 10 to $1,000$. In each experiment, we generate $1,000$ 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 $5ms$. For I/O efficiency, we implemented Quad-trees using linear representation [2], which is easily embeddable into B^+-trees. Section 4.5.1 evaluates the partitioning methods of Section 4.3. Section 4.5.2 compares R-tree Cloak (RC) and Quad-Tree Cloak (QC) against Hilbert Cloak [39], the only existing method which is reciprocal (hence secure). In Section 4.5.3 we address the variable query frequency scenario.

4.5.1 Evaluation of Partitioning Techniques

First, we consider the RC implementation of *Reciprocal* and compare different partition methods (i.e., GH, AR, DH and TD). Figure 4.11 illustrates the ASR area and generation time as a function of K, using a fixed page size of 4KB. AR has the clear advantage in terms of ASR area, while GH and DH both outperform TD. In terms of generation time, 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, DH 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



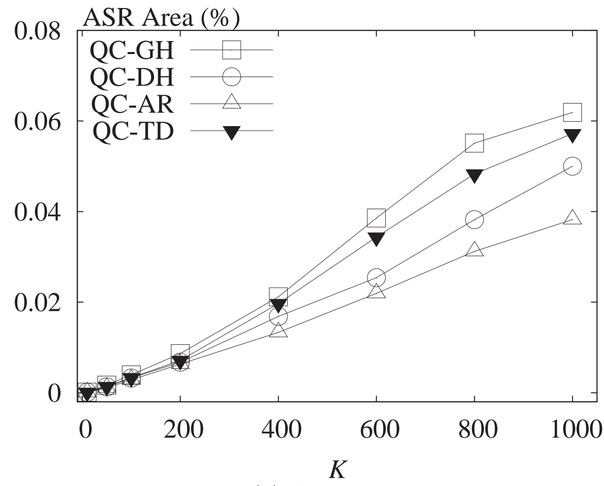
(a) Area



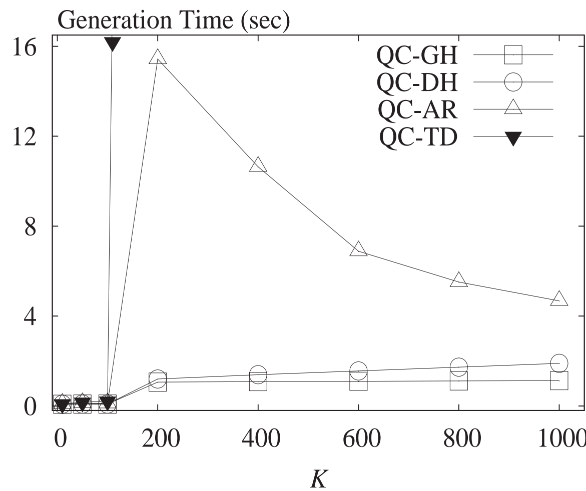
(b) Time

Figure 4.11: R-tree Cloak (RC). Partitioning methods versus K

splits drops (i.e., there are fewer, larger buckets). On the other hand, the cost of DH increases linearly with K. TD is expensive for partitioning at level 2 (in some cases up to 100 sec per query) and omitted for $K > 80$.



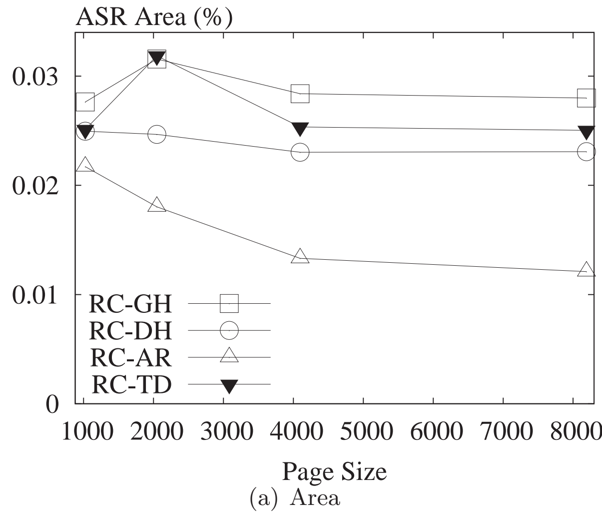
(a) Area



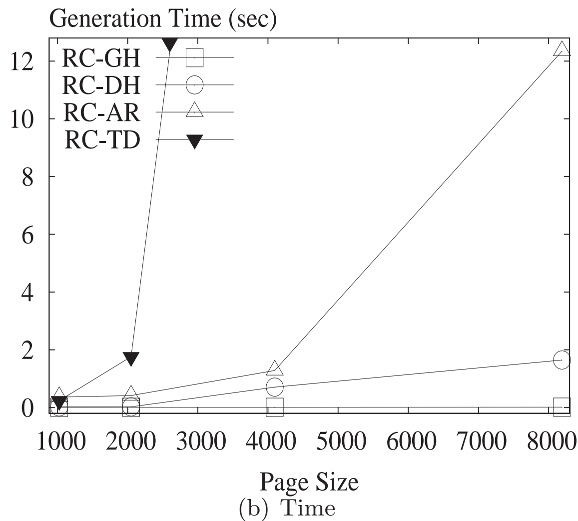
(b) Time

Figure 4.12: Quad-tree Cloak (QC). Partitioning methods versus K

Figure 4.12 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.



(a) Area

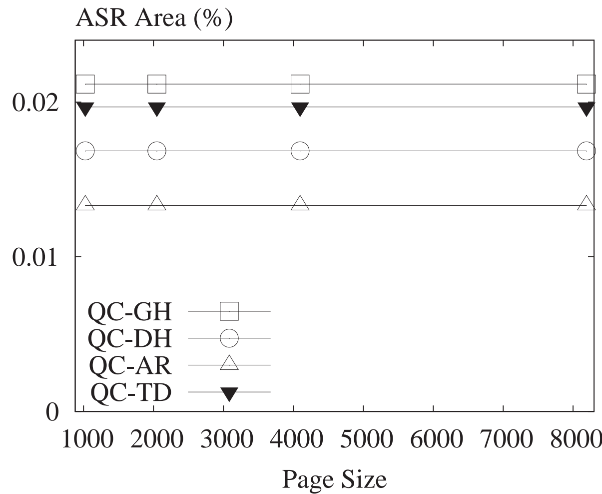


(b) Time

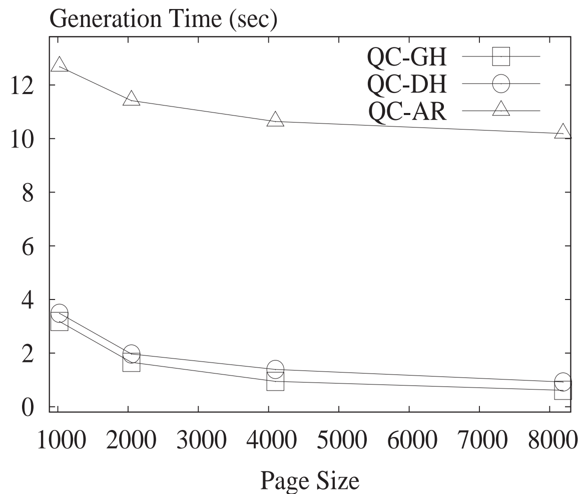
Figure 4.13: RC versus page size

In Figure 4.13 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, at the transition from 1 to 2KB, GH and DH exhibit 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 and DH 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 8KB page size, it needs 400sec per query). The cost of AR grows faster than that of DH due to the recursive splits. GH is rather insensitive to the page size since it computes a single bucket, independently of node cardinality.



(a) Area



(b) Time

Figure 4.14: QC versus page size

Figure 4.14 shows the same experiment for QC. Observe that the page size does not affect the ASR area, since construction 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 Figure 4.14b (TD is omitted due to very high values).

Summarizing, among the various local partitioning techniques GH is the

method of choice when the priority is efficiency. On the other hand, AR is the best technique in terms of effectiveness. DH offers a trade-off between the two: it obtains smaller ASR area than GH, and it is usually faster than AR. The performance of TD is unsatisfactory, as it is extremely expensive while producing ASRs with quality comparable to GH. Regarding the R-tree and Quad-tree implementations, they offer similar ASR areas, but RC is more efficient. 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). Next, we compare RC-GH and RC-AR against Hilbert Cloak (HC).

4.5.2 Comparison with Hilbert Cloak (HC)

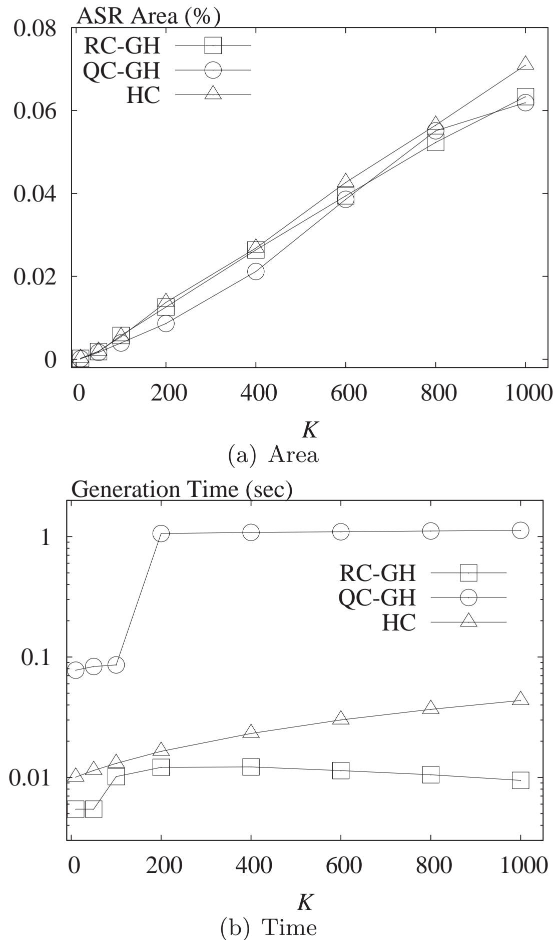


Figure 4.15: RC-GH and RC-AR versus HC

Figure 4.15 shows the relative performance of RC-GH, RC-AR and HC. RC-GH is slightly better than HC in terms of ASR size and up to one order of magnitude faster, as can be observed from the log-scale graph in Figure 4.15b. 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.

4.5.3 Variable Query Frequencies

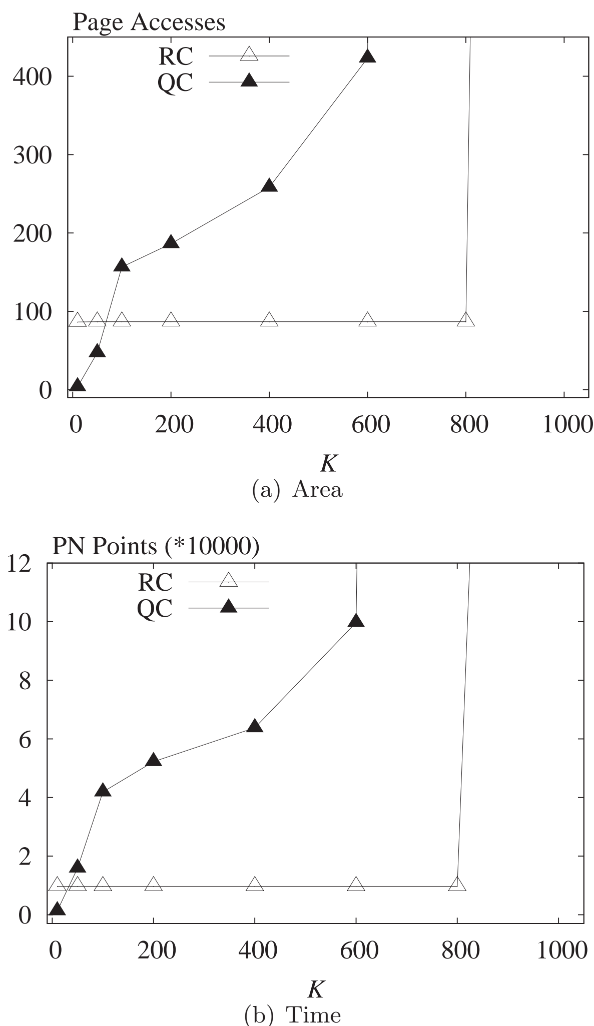
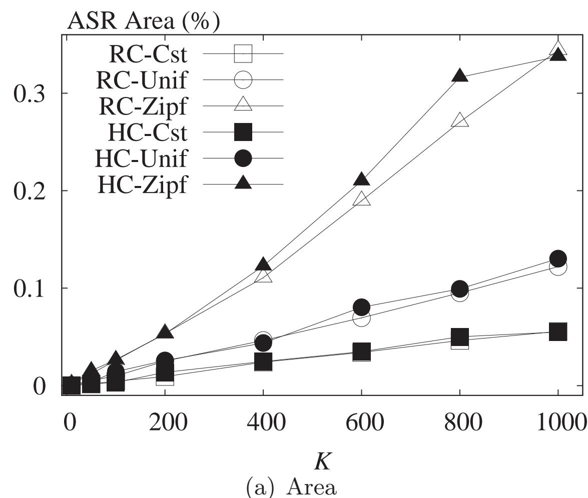


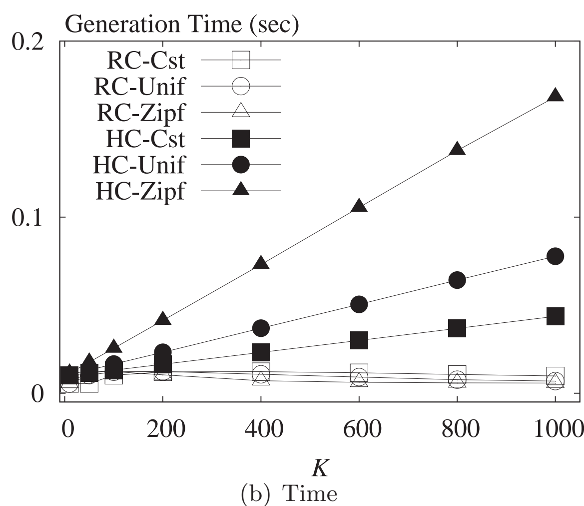
Figure 4.16: PN overhead for variable query frequency

As discussed in Section 4.4, local partitioning methods that require loading the entire PN node (e.g., AR, DH, 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 $1,000$ queries, each assigned to a user according to the zipf distribution with parameter 0.8. Page size is 4KB. The results are shown in Figure 4.16. 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, DH and TD are impractical for skewed frequency distribution.



(a) Area



(b) Time

Figure 4.17: RC-FQGH versus HC_f

Next, 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 $1,000$ random queries with constant (Cst), uniform (Unif) and zipf-0.8 distribution (Zipf). Figure 4.17 shows that, guaranteeing privacy for variable query frequency comes at an addi-

tional increase in ASR size, which grows with the skewness of the frequency distribution. RC-FQGH is slightly better in terms of ASR area. On the other hand, 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.

4.6 Discussion

Our proposed reciprocal framework for spatial K-anonymity offers a systematic methodology to enforce SKA on top of index structures which are already wide-spread in LBS applications. The framework is versatile, and allows a broad range of partitioning techniques: GH offers excellent efficiency, with an ASR generation time up to one order of magnitude faster than competitor methods. GH is the ideal choice if the anonymizer service has limited resources, and handles a large number of queries.

In terms of ASR effectiveness, AR is the method of choice, obtaining ASRs with roughly 50% the area of those obtained by GH. In scenarios where the LBS is the bottleneck, or quotas/charges are imposed on users based on the amount of LBS processing incurred, AR is preferred.

For the variable query frequency setting, the frequency-aware flavor of GH (FQGH) outperforms the only existing reciprocal (i.e. secure) anonymization method - *Hilbert Cloak* - by up to one order of magnitude in terms of efficiency, with similar ASR extent.

As stated in the assumptions from Section 2.2, we only employ SKA for a snapshot of user locations. In this setting, our proposed reciprocal framework guarantees user privacy. However, in the case of continuous queries, an attacker can correlate information from ASRs generated at different timestamps in order to expose the querying user. Later in Section 6 we will propose a private information retrieval protocol which guarantees privacy in the case of continuous queries as well.

Chapter 5

Decentralized Query Anonymization

5.1 Introduction

So far, most existing approaches utilized the *centralized* architecture of Figure 1.3, where a trusted anonymizer server acts as an intermediate tier between the users and the LBS. All users subscribe to the anonymizer and continuously report their location while they move. Each user sends his query to the anonymizer, which constructs the appropriate K-ASR and contacts the LBS. The LBS computes the answer based on the K-ASR, instead of the exact user location; thus, the response of the LBS is a superset of the answer. Finally, the anonymizer filters the result from the LBS and returns the exact answer to the user.

A centralized anonymizing service has the following shortcomings *(i)* The anonymizer server is a bottleneck due to handling query requests, frequent updates of user locations and result post-processing. *(ii)* The anonymizer is a single point of failure; the system cannot function without it. *(iii)* The complete knowledge of the locations and queries of all users is a serious security threat, if the anonymizer is compromised. Even if there is no attack, the centralized anonymizer may be subject to governmental control, and may be banned or forced to disclose sensitive user information (similar to the legal case of the Napster file-sharing service).

In this chapter, we propose distributed architectures for anonymous location-based queries, which address the problems of centralized solutions. Mobile users self-organize into a P2P overlay network, and cooperate to assemble K-ASRs. The bottleneck of the centralized server is avoided. Moreover, since the state of the system is distributed in many users, distributed solutions are resilient to attacks.

Our specific contributions are two P2P systems: *(i)* In Section 5.2, we introduce Privé, a hierarchical P2P network that implements the *Hilbert Cloak* algorithm presented in Section 3.4. The structure of Privé resembles a distributed B^+-tree (each mobile user corresponds to a data point), with additional annotation to support efficiently the Hilbert-based K-ASR construction. Privé offers privacy guarantees for snapshot LBS queries, but it may exhibit slow response time under heavy load, due to its hierarchical organization. *(ii)* In Section 5.3, we present the MobiHide P2P network, which employs a randomized version of *Hilbert Cloak*: MobiHide does not enforce reciprocity, hence it does not offer privacy guarantees under all scenarios. Nevertheless, we show that it provides strong privacy in practice, and its scalability is clearly superior to Privé. Therefore, our two decentralized solutions offer a trade-off between privacy guarantees and system scalability.

5.2 Privé

Figure 5.1 depicts the architecture of Privé. We assume a large number of users who carry mobile devices (e.g., mobile phones, PDAs) with embedded positioning capabilities (e.g., GPS). The devices have processing power and access the network through a wireless protocol such as WiFi, GPRS or 3G. Moreover, each device has a unique network identity (e.g., IP address) and can establish point-to-point communication (e.g., TCP/IP sockets) with any other device in the system through a base station (i.e., the two devices do not need to be within communication range of each other). For security reasons, all communication links are encrypted.

In addition, we assume the existence of a trusted central *Certification Server* (CS), where users are registered. Prior to entering the system, a

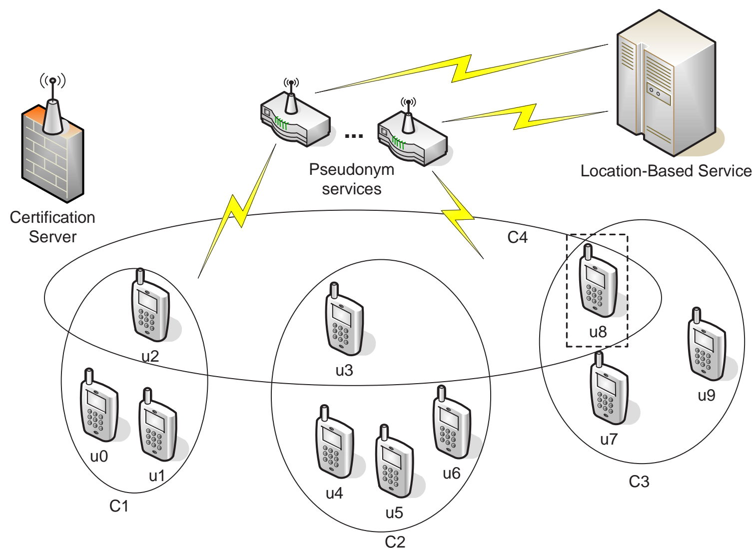


Figure 5.1: Architecture of PRIVÉ

user u must authenticate against the CS and obtain a certificate. Users having a certificate are trusted by all other users. Typically, a certificate is valid for a few hours; it can be renewed by recontacting the CS. Apart from the certificate, the CS returns to u the IP addresses of some users who are currently in the system. u uses this list to identify an entry point to the distributed network. Note that the CS does not know the locations of the users and does not participate in the anonymization process. Therefore the workload of the CS is low (i.e., no location updates); moreover, it does not store any sensitive information.

Each user corresponds to a peer. Peers are grouped into clusters, according to their location. Within each cluster, peers elect a cluster head, and the set of heads is grouped recursively to form a tree. To achieve load balancing, the cluster heads are rotated in a round-robin manner. By definition, cluster heads belong to multiple levels of the tree. In Figure 5.1, for instance, there is a two-level hierarchy, where users u_2, u_3, u_8 are the heads of cluster C_1, C_2 and C_3, respectively; also, u_8 is the head of the upper layer cluster C_4.

Assume user u_1 asks a query. u_1 initiates a distributed procedure to build a K-ASR, in cooperation with other users (the details of the protocol

are presented in Section 5.2.3). Once the K-ASR is ready, u_1 needs to send it to the LBS. In order to hide his IP address, u_1 uses a pseudonym. To obtain a pseudonym, any existing service for anonymous web surfing can be used[1].

Note that the pseudonym service does not know the location of any user. Moreover, the auxiliary users inside the K-ASR collaborate only to hide the location, but do not know the exact query of u_1; therefore, a single point of attack is avoided.

5.2.1 *Hilbert Cloak* with a B$^+$-tree index

In Section 3.4, we have described the details of the *Hilbert Cloak* algorithm, and mentioned that it can be efficiently implemented with any type of annotated index structure. The index must efficiently support the K-ASR formation operation, which boils down to determining the *start* and *end* values for a certain bucket, as given by eq. (3.1).

In PRIVÉ, we use an annotated B$^+$-tree (similar to the aR-tree [51]), which stores the number of leaf nodes in each of its subtrees.

Example 5.1. Figure 5.2 shows the application of *Hilbert Cloak* with an annotated B$^+$-tree index. For each internal node entry e, we store the number of leaf entries that are rooted at e; annotation counters are shown in parentheses. Assume we want to determine a K-ASR for entry 37, with $K = 6$. First, we compute the rank of entry 37 (Figure 5.2a): we follow the path in the tree from root to the leaf that contains 37, and at each internal node we add to the rank value the sum of all counters in the node situated at the left of the followed pointer. At the leaf layer, we add to the rank the local rank value of key 37 in its leaf, and obtain rank 8 (ranks start from 0). Then, we calculate the bucket delimiters using eq. (3.1), and obtain the interval [6..11]. Next (Figure 5.2b), we perform a range search to locate the entries with ranks [6..11]. Observe that this operation uses the annotation counters, rather than the B$^+$-tree keys. Sub-ranges at each level are deter-

[1]Since each user can access his preferred pseudonym service, that service is not a bottleneck or a single point of failure. The pseudonym, as opposed to the location anonymizer, does *not* need to pool together a large number of users

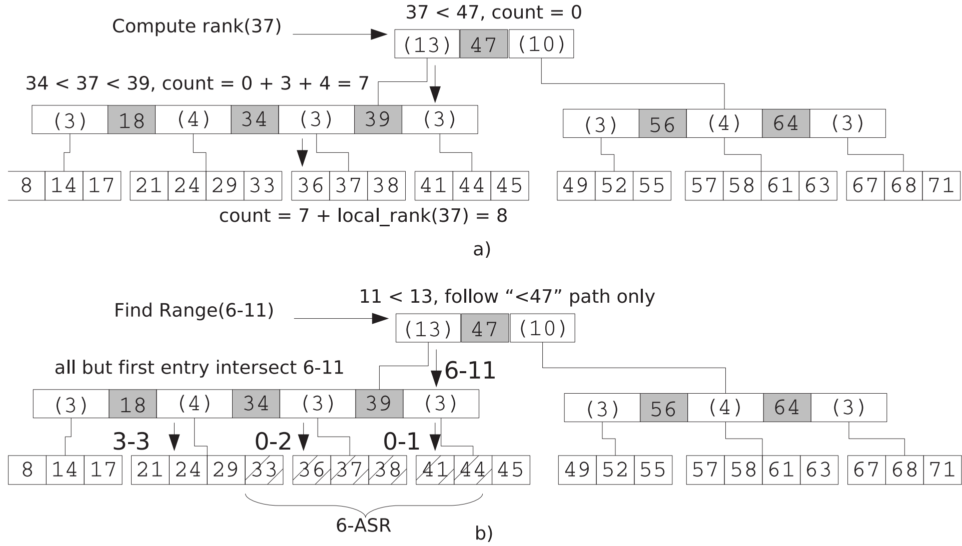


Figure 5.2: *Hilbert Cloak* with Annotated B$^+$-tree

mined by splitting the initial range based on subtree sizes; the offset for the recursive call at entry e is determined as the initial start value minus the sum of counters of all entries in the node preceding e. The resulting K-ASR is highlighted in the diagram. □

The data structure is scalable, since the complexity of constructing the K-ASR is $O(\log N + K)$, whereas search, insert and delete cost is $O(\log N)$.

5.2.2 Protocol Overview

PRIVÉ mimics the functionality of a B$^+$-tree in a distributed setting. Each mobile user u has an associated index entry consisting of an ID (e.g., IP address), and the Hilbert value $H(u)$ of his location as index key. A node (leaf or internal) in the B$^+$-tree corresponds to a *cluster* of users, with size bounded between α and 3α, where α is a fixed system parameter. We use the terms *cluster* and *index node* interchangeably. The maximum cluster size is 3α, instead of the usual 2α for B$^+$-trees, to prevent cascading splits and merges (i.e., a split followed by a user departure), which are costly in the distributed environment.

Every user belongs to a leaf level cluster (level 0), and the contents of

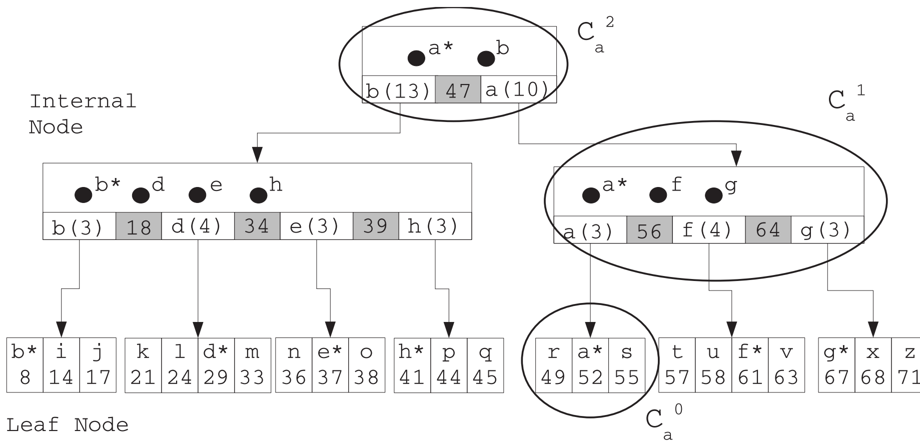


Figure 5.3: Distributed Index Structure, α=2

each cluster are disjoint (see Figure 5.3). The users of each cluster C elect a leader called $head(C)$. The head (marked with an asterisk) handles all index operations on behalf of the users in the cluster. Cluster heads are recursively grouped to form a tree; therefore, they belong to multiple levels of the tree. We denote by C_u^i, the level i cluster which includes user u. In our example, user u_a is the head of cluster C_a^0 at level 0, and also the head of clusters C_a^1 and C_a^2; therefore, it belongs to every level of the tree. There is a single cluster at the top of the hierarchy, denoted as *top*. The cluster head of *top* is denoted by *root* (u_a in the example). In our protocol description, we use remote procedure call convention to specify interactions between users. The notation $u.func(params)$ denotes the invocation of subroutine $func$ with parameters *params* at user u.

Each cluster is associated with its *state* information. The state of a leaf level cluster consists of an ordered list of (IP address, $H(u)$) pairs (user coordinates can be derived from the $H(u)$ value). The state of an upper layer cluster with m elements consists of a list of m user addresses, separated by $m-1$ key values used to direct the search; the process is similar to a B^+-tree, with the role of memory pointers fulfilled by the IP addresses of users. Each internal node entry is annotated with a counter (depicted in parentheses) representing the total number of users at the subtree under the entry. Only the head needs to know the state of the cluster. However, in our implementation, we replicate the state on every user within the cluster,

Table 5.1: Privé Protocol Terminology

Notation	*Definition*
C_u^i	level i cluster user u belongs to
$head(C)$	cluster leader of cluster C
$parent(C)$	the parent-cluster of cluster C
top	the cluster at the top of the hierarchy
$root$	$head(top)$
$u.func(params)$	RPC call for $func$ at user u

to improve fault tolerance (in Section 5.4, we discuss the tradeoff between fault tolerance and maintenance cost). The Privé hierarchy has at most $\log_\alpha N$ layers, where N is the total number of users. Since the cluster size is bounded and a user may belong to at most one cluster at each level, there is an upper bound of $O(\alpha \log_\alpha N)$ on the membership state stored at a user. The Privé protocol terminology is summarized in Table 5.1.

5.2.3 Protocol Operations

The index supports four operations: *join*, *departure*, *relocation* and *K-request* (i.e., a request for a K-ASR with anonymization degree K). We establish two performance metrics for Privé: *(i) latency*: the number of hops an index operation requires to complete. The latency is equal to the longest tree path followed as a result of the operation. Multiple paths may be followed in parallel during an operation. *(ii) communication cost*: the number of messages generated by an index operation.

Join. User join corresponds to a B^+-tree insertion operation. Newly joining users authenticate at the certification server and receive the address of a user already inside the system. Without loss of generality, we assume that joining users know the root, since the root can be reached from any user in $O(\log_\alpha N)$ cost. We stress that since we require an index structure with annotation (in order to determine the absolute ranks of users), all joins must occur through the root. To avoid overloading the root, we devise a load-balancing mechanism (Section 5.2.4). User join has $O(\log_\alpha N)$ complexity in terms of latency and $O(\log_\alpha N + \alpha)$ communication cost; the second term is for updating the cluster state in all the users of the affected cluster.

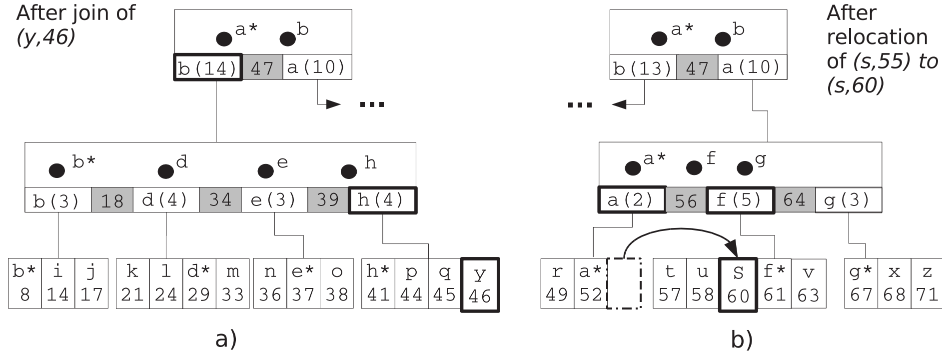


Figure 5.4: User Join and Relocation, α=2

Consider user u_y with Hilbert value $H(u_y) = 46$ that joins the index of Figure 5.3: u_y contacts u_a (at the root level) who forwards the join request to u_b and updates u_b's annotation counter in C_a^2 to 14. u_b then forwards the request to u_h, whose annotation counter in C_b^1 is updated to 4. Figure 5.4(a) shows the join outcome. User join may trigger a cluster split, handled similarly to a B$^+$-tree node split; the head initiating the split leads one of the resulting clusters, and appoints a random initial cluster node to lead the other.

Departure (informed). User departure is similar to a B$^+$-tree deletion. The effect of deletion must be propagated to *root* to update the annotation counters. Deletion has $O(\log_\alpha N)$ latency and $O(\log_\alpha N + \alpha)$ communication cost. If the cluster size decreases below α, the head triggers a merge operation with the neighbor leaf-level cluster that has fewer members (to avoid a cascaded split). The head of the resulting cluster can be any of the initial heads, except if one of them (e.g., u_a) is also head at the higher level. If so, u_a will be chosen as leader, to minimize membership changes.

Relocation. User mobility is treated as an entry update, which in a B$^+$-tree translates into a deletion and an insertion. Since users are likely to change location often, we optimize this process by performing local reassignment of users to nearby clusters. Due to the good locality properties of Hilbert ordering, the number of clusters involved in relocation is likely to be small. Annotation counter updates are only performed by affected clusters; this way, updates are not propagated all the way to the root. The upper

```
u.RelocateMyself() /*executed by moving user*/
  determine new key value H_u = Hilbert(u.x, u.y)
  call head(C_u^0).Relocate(u, H_u,0)
u.Relocate(relocated_user,H,l)
  if (H in indexed key range at level l )
    if (l = 0)
      add relocated_user to leaf user list; return
    else
      let n be the next hop for H
      call n.Relocate(relocated_user, H,l − 1)
  else
    call head(parent(C_u^l)).Relocate(relocated_user, H,l + 1)
```

Figure 5.5: User Relocation Pseudocode

bound on relocation latency is $O(\log_\alpha N)$, but in most cases relocation only involves a few clusters, at the low layers of the index. The pseudocode for user relocation is given in Figure 5.5.

Consider user u_s from Figure 5.3 who relocates to a new position with Hilbert value 60. He forwards the request to $u_a = head(C_s^0)$. u_a cannot keep u_s within the same leaf entry, since the new value is outside the interval [49..55]. Since $u_a = head(C_a^1)$, with no additional message, u_a decides that u_s can be relocated to C_f^0, forwards the request to u_f and updates the annotation counters of u_a and u_f accordingly. Figure 5.4(b) illustrates the relocation outcome.

***K*-request.** This operation corresponds to the *Hilbert Cloak* algorithm described in Section 5.2.1. Consider the example in Figure 5.6, where user u_m issues a K-request with K=6. The request follows the path: $u_m \rightarrow u_d \rightarrow u_b \rightarrow u_a$ (solid arrows in Figure 5.6(a)). The root u_a determines the K-bucket (i.e., $start$ = 6, end = 11) and sends a K-ASR request to u_b (dotted arrows in Figure 5.6(a)). u_b sends in parallel requests for partial K-ASRs with ranges [6..6], [7..9] and [10..11] to u_d, u_e and u_h, respectively. u_b, which is the head of the lowest-layer cluster that completely covers the K-bucket (shown hashed in Figure 5.6(b)) collects the partial K-ASRs, assembles the final query K-ASR and sends it back to the query issuer on the reverse path of the request. Note that, the cluster head that covers the K-bucket

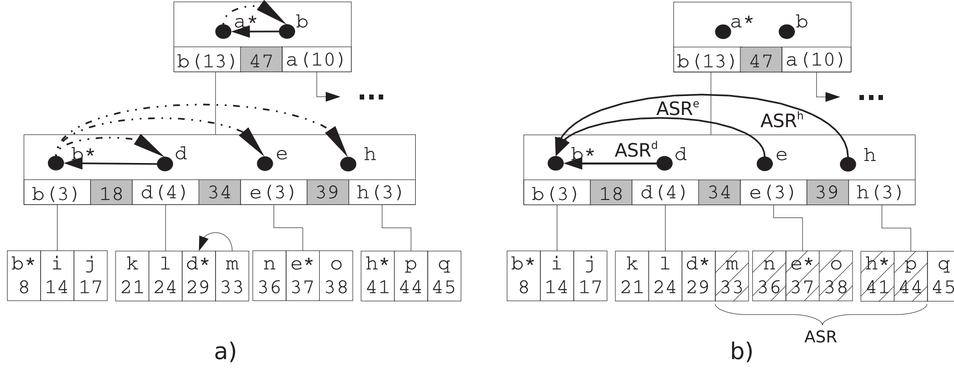


Figure 5.6: K-request, α=2, K=6

sustains the highest load among all other users involved in the query. This potential load imbalance issue is addressed in Section 5.2.4. A K-request has $O(\log_\alpha N)+O(\log_\alpha K)$ latency and $O(\log_\alpha N)+O(K/\alpha)$ communication cost. The pseudocode for K-request is shown in Figure 5.7. Once the K-ASR is constructed, the query issuer (i.e., u_m) can send the anonymized query to the LBS through any preferred pseudonym service.

5.2.4 Fault Tolerance and Load Balancing

PRIVÉ implements a soft-state based mechanism to deal with user failures or disconnections without notification[2]. Each cluster leader sends periodically (i.e., every δt seconds) a *membership_update* message to all cluster members. The message contains the membership list of the current cluster C and that of $parent(C)$. Cluster members respond to these messages; if a cluster member does not respond to two consecutive messages, it is considered disconnected and removed from the cluster. The change is broadcast by the cluster head to the remaining cluster members.

If a non-head cluster member u does not receive a *membership_update* from its head for a $2\delta t$ period, it initiates a *leader election* process. Alternatively, when u attempts to initiate a operation, such as query or relocation, but cannot contact the cluster head for two consecutive attempts, it triggers the leader election protocol without waiting for the timer to expire. u

[2]Similar fault-tolerance and load-balancing mechanisms have also been proposed for hierarchical wireless networks [11]

```
u.K-request() /*executed by query source*/
  determine key value H_u = Hilbert(u.x, u.y)
  call head(C_u^0).ForwardRequest(H_u, 0, 0)
u.ForwardRequest(H, count, l)
  if (l = 0) count = rank_H in leaf entry
  else count+ = sum of annotation counters of keys < H
  if (u is root)
    compute start and end using eq (3.1)
    K-ASR = root.findMBR(start, end, root_height)
  else call head(C_u^{l+1}).ForwardRequest(H, count, l + 1)
u.findMBR(start,end,l)
  if (l = 0) /*leaf level*/
    return MBR of members with local rank in [start,end]
  find set of next hops U for range [start,end]
  MBR = ∅
  for u' ∈ U
    MBR = MBR ∪ u'.findMBR(start_u', end_u', l − 1)
  return MBR
```

Figure 5.7: K-request

checks the membership it had at the last update, and chooses as leader (i.e., *new_head*) the user with the smallest identifier. It then sends a *transfer_head* message to *new_head*, which in turn sends a membership update message to all cluster users and also contacts $head(parent(C))$ to notify the change in leadership. *new_head* will replace the old head in all layers where the latter was leader before disconnection.

The hierarchical structure can cause significant differences between the load sustained by cluster heads and ordinary cluster members, as well as among cluster heads at different layers of the hierarchy. To alleviate the inherent imbalance, we propose a cluster head *rotation* mechanism, where users take turns in fulfilling the cluster head role. Since the promotion to cluster head translates into presence at a higher layer of the hierarchy, the rotation also ensures that users equally share the load at different layers.

Rotation is triggered when a node reaches a certain load threshold, denoted by *load unit*. In wireless devices, the communication cost is dominant.

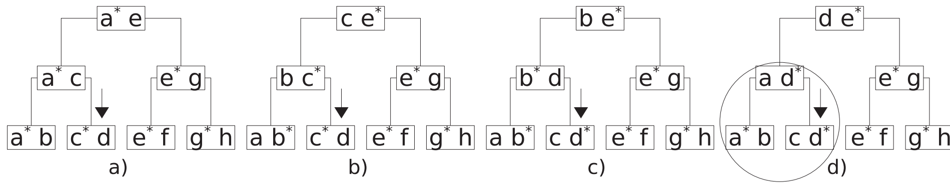


Figure 5.8: Load Balancing Mechanism

It is also important from the user's perspective, since mobile phone operators charge by the amount of transferred data. Therefore, in PRIVÉ the load is best represented by the number of messages sent and received by the user.

When user u reaches one load unit, it triggers a head rotation in all the clusters it currently heads, starting with its highest layer. For each node along the path to its level 0 cluster, the member with the least load is appointed as new head. Note that, since u stores the membership state about all clusters it belongs to at different layers, the appointment of a new leader can be done directly by u, without the need for a complex protocol or additional messages. Choosing the cluster member with the lowest load prevents the newly appointed head to start a fresh rotation soon after promotion.

Figure 5.8 illustrates the rotation mechanism. For simplicity, all clusters have size 2. Assume all queries originate at user u_d with K=4. After u_a reaches one load unit, it hands over the root role to u_e (at layer 2) from the right-hand subtree. Also, at layer 1, u_c becomes the head and is automatically promoted to layer 2. Similarly, at layer 0, u_b becomes the head and is promoted to layer 1; the result is shown in Figure 5.8(b). Next, u_c reaches its load unit, because more requests pass through it (it must inject queries and collect partial K-ASRs). u_c triggers a rotation at level 1 and appoints u_b as cluster head (see Figure 5.8(c)). Subsequently, u_b may be the next one to reach the load threshold, and start a new rotation in the left subtree. Observe that at step (d), the left subtree has already performed a complete rotation round, whereas the right subtree has only performed one change. Hence, our rotation mechanism alleviates hotspots (an entire subtree shares the load generated by u_d) and at the same time provides a degree of *fairness*, not allowing a localized hotspot to affect a large partition

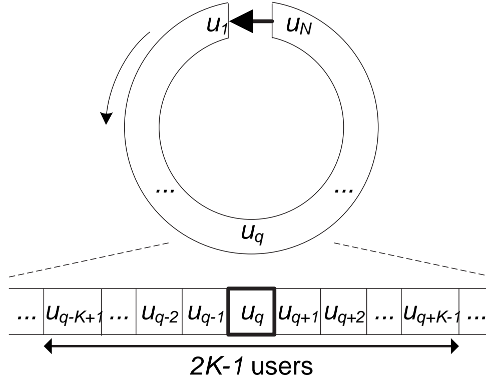


Figure 5.9: Hilbert sequence ring

of the index.

The granularity of load unit choice is important in practice, in order to achieve a good tradeoff between load balancing and communication cost, since a rotation may incur a number of messages as large as $O(\alpha \log_\alpha N)$. In practice, the load unit value can be set to a multiple of the rotation cost, i.e. $\beta \times \alpha \log_\alpha N$, where $\beta \in \mathbb{N}$. This way, the communication overhead of changing cluster leaders will not exceed an $1/\beta$ fraction of the usual protocol operation cost.

5.3 MobiHide

In this section, we introduce MobiHide, a P2P system which employs a randomized K-ASR construction technique to offer query source anonymity, and is scalable to a large number of mobile users. Similar to Privé, MobiHide is using the Hilbert ordering of the users' locations. However, instead of grouping users into fixed partitions, it forms a K-ASR by randomly choosing K consecutive users, including the querying user.

Let $[u_1, \ldots, u_N]$ be the sequence of all users, ordered by their Hilbert value. To allow random K-ASR selection for the users at the start and end of the sequence, the 1-D space becomes a ring (or torus), instead of an array. Therefore, u_1 is after u_N (and u_N is before u_1). Figure 5.9 presents an example, where u_q is the user who issues a query. There are K ways to choose a set of consecutive K users which includes u_q: $[u_{q-K+1} : u_q], [u_{q-K+2} :$

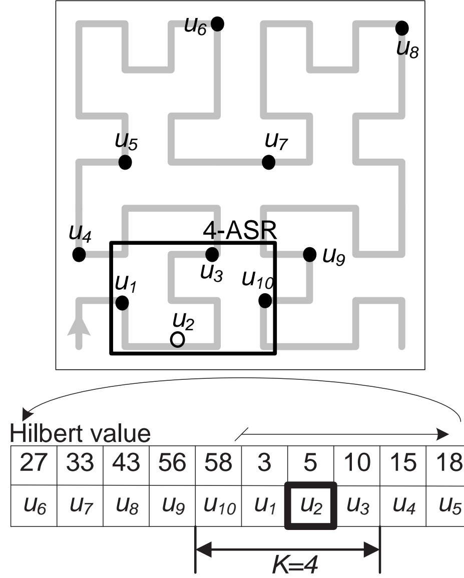


Figure 5.10: K-ASR construction in MobiHide

$u_{q+1}], \ldots, [u_q : u_{q+K-1}]$. This is equivalent to choosing a random offset $l \in [0,\ K-1]$, representing the offset of u_q in the resulting sequence. For example, if $l = 0$, the resulting sequence is $[u_q : u_{q+K-1}]$. Observe that we only need information in the neighborhood of u_q in order to select the sequence (as opposed to Privé, which needs the global ranking). Therefore, MobiHide works in a fully decentralized manner, and can be deployed on top of a scalable structure such as Chord.

Example 5.2. Figure 5.10 shows the MobiHide K-ASR construction, where u_2 is the querying user. Let $K = 4$ and assume that u_2 randomly selects offset $l = 2$. According to the Hilbert ordering, the resulting sequence of users is $[u_{10}, u_1, u_2, u_3]$. The corresponding K-ASR is the *minimum bounding rectangle* (MBR) which encloses these four users. In this particular example it was necessary to wrap around the Hilbert sequence (from u_{10} to u_1). Observe that the "jump" in Euclidean distance due to wrapping, is not necessarily larger than other "jumps" that may occur within the sequence (e.g., from user u_8 to u_9). Therefore, the average size of the K-ASRs (thus the query cost) is not affected significantly by wrapping. We investigate further this issue experimentally in Section 5.4. □

Theorem 5.3. *If all users issue queries with the same probability (i.e.,*

uniform distribution), MobiHide **guarantees** *query anonymity.*

Proof. Denote by P_Q the probability of a user issuing a query (same for all users). The query source generates a random offset $l \in [0,\ K-1]$; we denote by $\langle u, l\rangle$ the event of user u generating a set of users with offset l. The probability $P_{\langle u,l\rangle} = P_Q/K$. Refer to Figure 5.9, where u_q is issuing a query. Obviously, u_q must belong to the set associated to his query. To guarantee anonymity, the probability of identifying u_q as the query source must not exceed $1/K$. We denote by A_q any set of users that includes u_q, and by P_{A_q} the probability of such a set being generated. We denote by P_{u_i} the probability of user u_i being the source of the query associated with A_q. Then, $P_{u_i} > 0$ only for users $[u_{q-K+1} : u_{q+K-1}]$, and by symmetry, $P_{u_{q-j}} = P_{u_{q+j}}$. We have:

$$P_{u_q} = \sum_{l=0}^{K-1} P_{\langle u_q,l\rangle} = P_Q, \tag{5.1}$$

$$P_{u_i} = \sum_{l=i-q}^{K-1} P_{\langle u_i,l\rangle} = \frac{K-i+q}{K} P_Q, i > q \tag{5.2}$$

$$P_{u_q} + 2\sum_{i=q+1}^{q+K-1} P_{u_i} = P_{A_q} \tag{5.3}$$

The probability of pinpointing u_q as the query source is

$$\frac{P_{u_q}}{P_{A_q}} = \frac{P_Q}{\left(1 + 2\sum_{i=1}^{K-1} \frac{K-i}{K}\right) P_Q} = \frac{1}{K}, \tag{5.4}$$

hence user u_q is K-anonymous. □

5.3.1 The Correlation Attack

In practice, the query distribution is not always uniform. In the extreme case, the same user (e.g., u_q) would send all queries and he would be included in all K-ASRs. An attacker can intersect the K-ASRs and pinpoint u_q as the querying user with high probability. It is more realistic, however, that many users ask queries, even if the query distribution is skewed. In this case, intersecting the K-ASRs is unlikely to compromise the system, since the

random sequence selection in MobiHide distributes the anonymized regions in the entire space. In order to succeed, the attacker should know the *exact locations of all users*, to be able to reconstruct the Hilbert sequence. Then, he could find the users included in each K-ASR by reverse-engineering the K-ASR construction mechanism, and speculate that the users who appear more frequently are the ones who issued the queries.

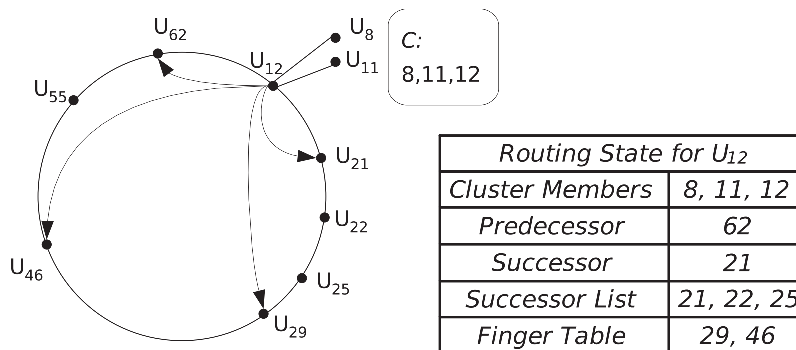
Consider the extreme case where the attacker knows the exact location of all users and intercepts the set $\mathcal{R}$ of K-ASRs. We formalize the correlation attack as follows: *(i)* Construct a histogram F with the number of occurrences of every user in any of the queries. *(ii)* For each $R \in \mathcal{R}$: infer the query source as the user in R with the highest number of occurrences in F.

The correlation attack gives an attacker powerful means to infer the query source. Privé guarantees anonymization against this type of attack, but as discussed in Section 5.1, may not scale well as the number of users increases. MobiHide cannot offer theoretical guarantees when the query distribution is extremely skewed. However, we believe that in practice this attack is hard to stage, since it is difficult for an attacker to know the exact locations of all users at each snapshot. Furthermore, we show experimentally (Section 5.4) that the probability of identifying the querying user in MobiHide is very close to the theoretical bound $1/K$, even if the attacker knows all users' locations and the query distribution is skewed. Finally, observe that MobiHide does not suffer from the *center-of-ASR* attack (see Section 2.2) because, by construction, the probability of u_q to be closest to the center of the K-ASR is $1/K$.

5.3.2 Protocol Overview

MobiHide users organize themselves into a Chord [57] P2P system. Chord is a Distributed Hash Table (DHT), where each peer (or node) has an m-bit key (the Hilbert value in our case), and it stores a routing table with pointers to other nodes (see Figure 5.11). The routing table at peer n with key key_n consists of:

- a *successor* and *predecessor* pointer to the node with the key that



Routing State for U12	
Cluster Members	8, 11, 12
Predecessor	62
Successor	21
Successor List	21, 22, 25
Finger Table	29, 46

Figure 5.11: MobiHide implementation over Chord

immediately follows (respectively, precedes) key_n on the ring

- a *successor list*, used mainly for fault tolerance, with a list of consecutive peers that follow n on the ring
- a *finger table*, with m pointers to nodes that are situated at 2^i distances from n $(i = 0, 1, .., m - 1\}$.

We denote by $H(u)$ the Chord key of user u. Assume that each user is mapped to a distinct Chord node. When user u wants to pose a query, he initiates the K-ASR construction procedure, denoted by *K-request*. u generates a random offset $l \in [0,\ K-1]$, and contacts the set P of l predecessors and the set S of $K-1-l$ successors on the Chord ring. The resulting K-ASR is the MBR that encloses users in $P \cup S \cup \{u\}$. The complexity of a K-request is $O(K)$ overlay hops.

Since K can be large (e.g., 50-100) in practice, we wish to reduce the number of hops, and hence the latency of K-request. We introduce an additional level of hierarchy, such that each overlay node represents a *cluster of users*, rather than a single user. Similar to Privé, each cluster has between α and 3α-1 users, where α is a system parameter. If the cluster reaches 3α, a split is performed and an additional ring node is created. If the size falls below α, a merge operation with another overlay node is performed[3]. We chose 3α, instead of 2α, as the upper bound on size, to minimize frequent

[3]Obviously, if more keys fall within a Chord segment, there will also be proportionally more nodes in that segment; therefore, hot-spots are avoided.

merge and split operations. Each cluster has a representative, or cluster *head*, which is part of the Chord ring. In the example of Figure 5.11, u_{12} is the head of cluster $\{u_8, u_{11}, u_{12}\}$. The head's key on the ring is the maximum of all keys inside its cluster, in order to preserve the key ordering on the ring. The cluster membership is maintained by the head, and is replicated to all cluster members, to enhance fault-tolerance. Heads are rotated periodically to achieve load-balancing. We denote by C_u the cluster that contains user u, and by CH_u the head of C_u.

We further describe how various operations are performed in MobiHide.

5.3.3 Protocol Operations

Similar to Privé, for each operation, we consider two performance metrics: *(i) latency*: the time to completion, measured as the number of overlay hops on the longest path followed. Multiple paths may be followed in parallel. *(ii) communication cost*: the communication cost of an operation, measured as the number of transmitted messages (communication cost typically prevails over CPU cost).

Join and Departure. User join is illustrated in Figure 5.12a. User u with key $H(u) = 71$ authenticates at the certification server[4] and receives the address of user u_{bs} inside the system. u_{bs} issues a search for key $H(u)$, which returns the address of u_{85}, the successor of 71 on the ring. u contacts u_{85} and joins cluster C. Hence, $C_u \equiv C$ and $CH_u \equiv u_{85}$. Upon u's join, CH_u checks the new size of cluster C_u, and if $size(C_u) = 3\alpha$, CH_u splits his cluster into two halves, in increasing order of key values. He appoints one of his cluster members, CH'_u, as head of the newly formed cluster. All nodes in the initial cluster are notified. CH_u and CH'_u also notify their predecessor and successor on the ring. CH'_u inherits a large part (if not all) of the finger table of CH_u; the rest of the table is determined through the Chord stabilization process [57].

In our example, the new size of C is 6 and $\alpha = 2$, so u_{85} triggers a split operation (Figure 5.12b). u_{85} divides his cluster C into two halves,

[4] MobiHide uses the same architectural components as shown in Figure 5.1 for Privé, except that the user organization is different

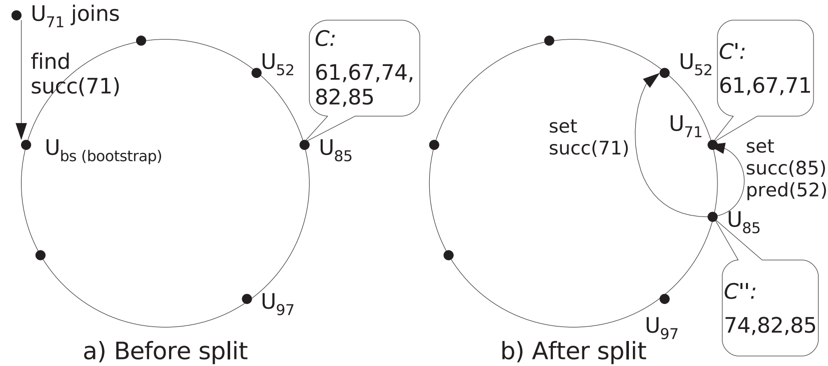


Figure 5.12: Join and Split, α=2

C' with members 61, 67 and 71, and C'' with members 74, 82 and 85. u_{71} is appointed as head of C', while u_{85} remains head for C''. u_{85} sets his predecessor pointer to u_{71}, and notifies the former predecessor u_{52} to change its successor from u_{85} to u_{71}. The complexity of join is $O(\log N - \log \alpha)$ latency and $O(\log N - \log \alpha + \alpha)$ communication cost (the last term stands for notifying all cluster members).

User u can depart gracefully, or fail; failure is addressed in Section 5.3.4. When u departs gracefully, he notifies his cluster head CH_u, who updates the cluster membership. If the departing node is cluster head, he appoints one of his members as new head. A merge can be triggered by departure. In this case, user CH_u triggering the merge contacts randomly either his successor s or predecessor p on the Chord ring to merge[5]. CH_u transfers his members (including himself) to the merging peer and ceases to be cluster head. All members are notified and the successor and predecessor pointers are updated.

Relocation. When user u moves to a new location, his Hilbert value $H(u)$ changes. If the new $H'(u)$ falls within the key range of other users in cluster C_u, u only needs to inform his cluster head of the key change. Otherwise, u performs a graceful departure followed by a join. Since Hilbert ordering preserves locality, it is likely that the relocation will be within a small distance from the initial ring position. The worst case complexity of relocation

[5]Alternatively, an interrogation phase can find which of s or p has fewer members, and merge with that one (to avoid cascaded splits and to equalize cluster sizes).

```
u.findASR(H,K)
  compute rank_H in sorted order of C_u
  generate random offset l
  before = max(0,l - rank_H)
  after = max(0,K-l + rank_H − size(C_u))
  if (after > 0)
    succ.FwdReq(after,1)
  if (before > 0)
    pred.FwdReq(before,−1)
  wait for partial MBRs
  K-ASR = union of all received MBR

u.K-request(K)
  call CH_u.findASR(H(u),K)
u.FwdReq(count,direction)
  if (direction == 1) /*Look Forward*/
    return MBR of first count keys
    if (count > size(C_u))
      succ.FwdReq(count − size(C_u),1)
  else                  /*Look Backward*/
    return MBR of last count keys
    if (count > size(C_u))
      pred.FwdReq(count − size(C_u),−1)
```

Figure 5.13: Pseudocode for K-Request

is $O(\log N - \log \alpha)$ latency and $O(\log N - \log \alpha + \alpha)$ communication cost.

***K*-request.** To generate a K-ASR, u forwards a K-request to his cluster head CH_u (unless u himself is the cluster head). CH_u generates a random offset $l \in [0,\ K-1]$. Then, CH_u examines the membership list of his cluster C_u and determines how many users in C_u will belong to the K-ASR. CH_u computes the values $before$ and $after$ corresponding to the number of users in K-ASR that are *outside* C_u and precede (respectively, follow) the set of keys in C_u. CH_u issues a request for the MBR[6] of these members to his predecessor p and successor s. In p and s the same procedure is followed recursively, until K users are found. CH_u waits for all answers, and assembles the K-ASR as the union of the received MBRs. The pseudocode for K-request is given in Figure 5.13. The complexity is $O(K/\alpha)$ in terms of both latency and communication cost. Once the K-ASR is assembled, u can submit it to the LBS using his preferred pseudonym service.

5.3.4 Fault-tolerance and Load Balancing

MobiHide inherits the good fault-tolerance properties of Chord [57]. Similar to Chord, some of the pointers to other peers (i.e., successor and predecessor pointers, the successor list and the finger table) may be temporarily corrupted (e.g., when a user fails). Such pointers are corrected periodically through a stabilization process. In addition to stabilization, MobiHide

[6] CH_u only acquires the MBR, not the exact location of users in other clusters.

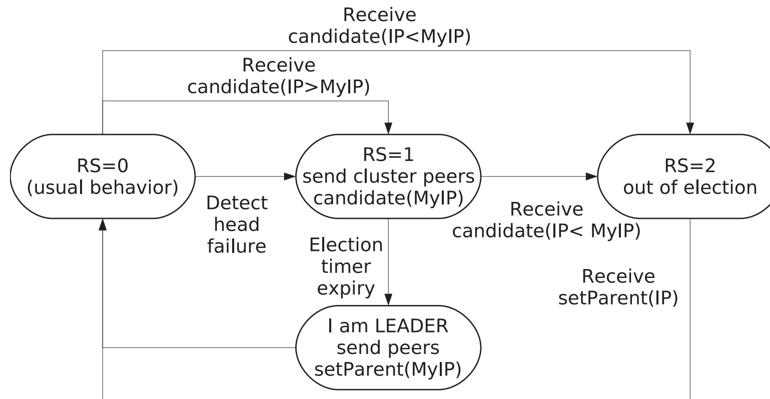


Figure 5.14: Leader Election Protocol

implements an intra-cluster maintenance mechanism. Each cluster head periodically (i.e., every δt seconds) checks if all cluster members are alive, by sending *beacon* messages; beacons contain the current cluster membership in addition to the successor and predecessor nodes of the head on the Chord ring. If a user fails to respond for $2\delta t$ seconds, he is considered failed and is removed from the cluster. Similarly, a non-head node that does not receive a beacon from his head for $2\delta t$ seconds, concludes that the head has failed and initiates a leader election protocol (see Figure 5.14). The RecoveryState (RS) variable of each node indicates whether the node is in normal operation ($RS = 0$) or participates in the election protocol. Since the cluster membership is replicated at all cluster nodes, recovery is facilitated. Upon detecting leader failure, node n enters the $RS = 1$ state, sends a $candidate(n.IP)$ message to all peers in the cluster and sets an election timer large enough to allow other peers to respond to the candidature proposal. When a node receives the $candidate(IP)$ message, it initiates its own candidature only if its address is smaller than IP; otherwise, it enters the $RS = 2$ state and waits for a $setParent$ message. The user with the smallest address declares himself leader and notifies all other cluster members, as well as the predecessor and successor on the ring.

To prevent unequal load sharing, a simple rotation mechanism is enforced among cluster members. The rotation is triggered when a certain load threshold is reached. This threshold is measured in terms of number of messages sent/received, since the communication cost is predominant in

terms of both energy consumption and fees payed to the service provider. When the cluster head CH transfers leadership to another cluster member CH', he transfers his routing state on the Chord ring and the cluster membership to CH'. Observe that the Chord key does not change, since it is the maximum key among all cluster members. Therefore, the overhead for the P2P network is minimal.

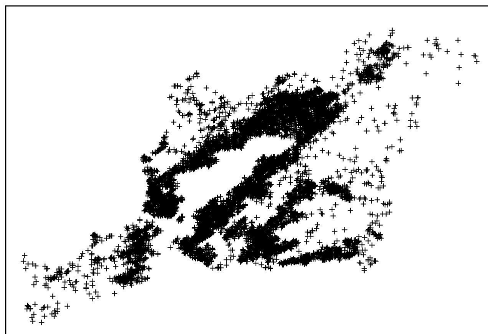
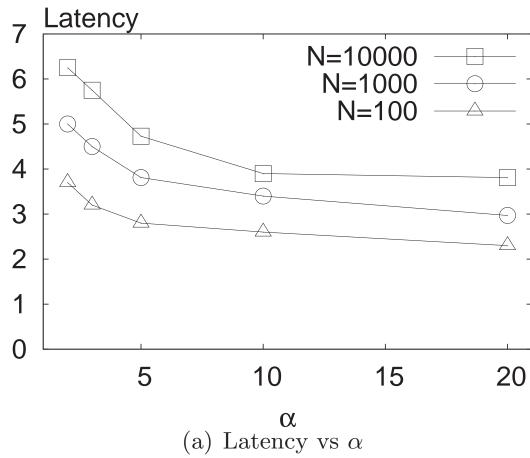


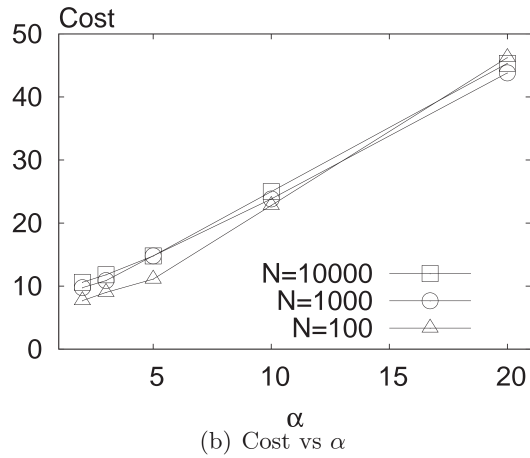
Figure 5.15: Dataset

5.4 Experimental Evaluation

To evaluate our distributed anonymization protocols, we have implemented event-driven packet level simulators in C++. Our PRIVÉ implementation is developed on top of the NICE [10] protocol suite for multicast delivery networks. For MOBIHIDE, we have used the Chord DHT [57] implementation in the p2psim [1] suite, a packet-level simulator for P2P systems. Since we are mostly interested in the overlay-layer performance, we consider a full mesh topology with lossless 500ms round-trip time links between any pair of users. Furthermore, we only consider packet loss as an effect of queueing at the processing nodes, and not as a result of link faults. Our workload consists of user locations and movement patterns, and is generated using the *Network-based Generator of Moving Objects* [17], which models user movement on public road networks. We consider user velocities ranging from 18 to 68km/h. We present our results for a data set consisting of the San Francisco bay area (Figure 5.15), with number of users N varying from $1,000$ to $10,000$. We vary the anonymization degree K from 10 to 160. We consider both uniform and Zipfian distributions of queries over the set of users. If not stated differently, we set $\alpha = 5$ (see Section 5.2.2 and 5.3.2). We compare PRIVÉ and MOBIHIDE against the only other existing distributed spatial anonymization system, *CloakP2P* [21]. In Section 5.4.1 we evaluate the PRIVÉ protocol, whereas in Section 5.4.2 we evaluate MOBIHIDE. Section 5.4.3 compares directly the two protocols.



(a) Latency vs α

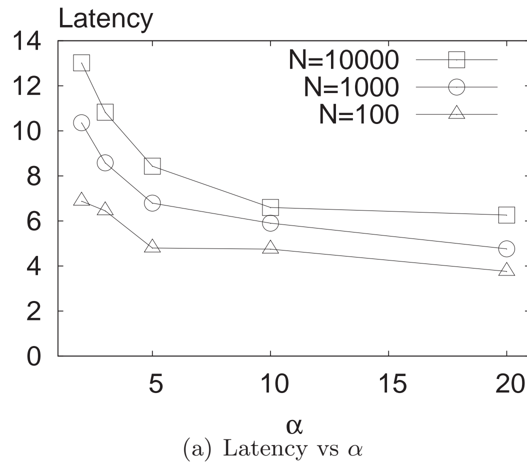


(b) Cost vs α

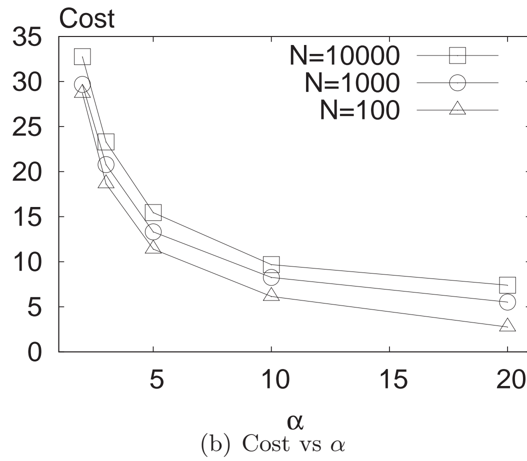
Figure 5.16: PRIVÉ Join/Leave Operation

5.4.1 Privé protocol

Join and Departure. In a system with N users, we perform $0.1N$ random user joins, followed by $0.1N$ random user departures. Figure 5.16(a) shows the join latency measured as hop count from the time a user issues a join request until he receives a join response message from its leaf-level head. We observe that the latency is lower than the theoretical $1 + \log_\alpha N$, as a user may appear in multiple levels and can avoid sending redundant messages to himself. The communication cost (i.e., total messages) per join and departure operation (Figure 5.16(b)) varies linearly with α, since ev-



(a) Latency vs α

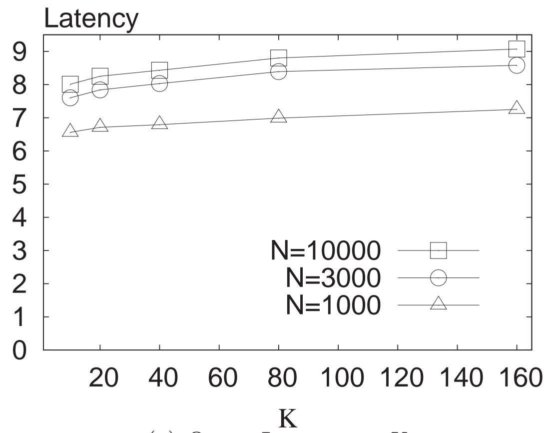


(b) Cost vs α

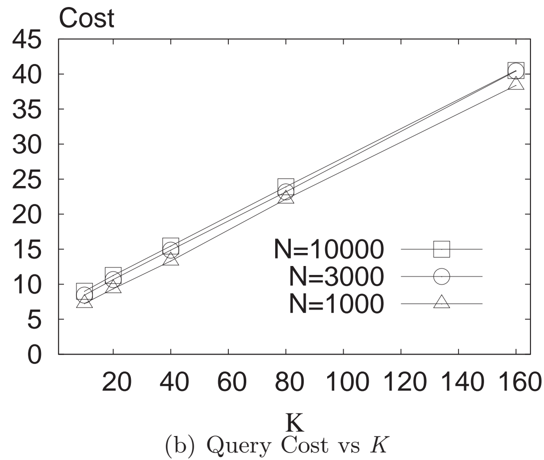
Figure 5.17: PRIVÉ K-request Operation

ery join/departure translates into a *membership_update* broadcast message within one leaf-level cluster. Note the role of α in the latency-cost trade-off: an increase of α decreases latency as $\log_\alpha N$, but triggers a linear cost increase in membership notification. A larger α also increases the cost of periodic cluster membership maintenance.

***K*-request.** Figure 5.17(a) and 5.17(b) show the K-request latency and communication cost for varying α, where K=40. Larger α decreases the latency as the height of the index decreases. The communication cost also decreases, as fewer leaf-level cluster heads need to be contacted to build



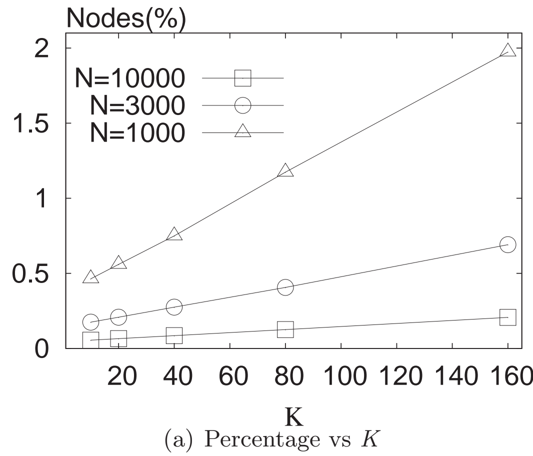
(a) Query Latency vs K



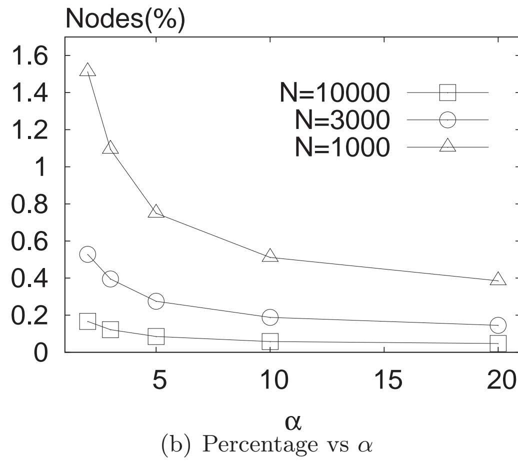
(b) Query Cost vs K

Figure 5.18: PRIVÉ K-request Operation

the K-ASR. However, α cannot grow very large from index maintenance considerations. Figure 5.18(a) and 5.18(b) show the latency and communication cost variation with anonymization degree K, $\alpha = 5$. Latency is only marginally affected by K (the dominant factor in latency is $\log_\alpha N$, since in practice $K \ll N$), while the communication cost grows linearly with K.



(a) Percentage vs K

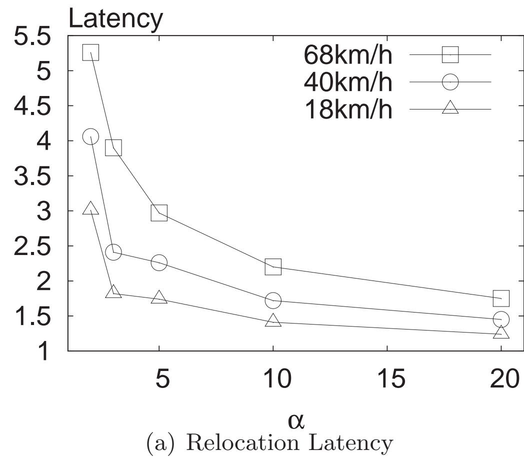


(b) Percentage vs α

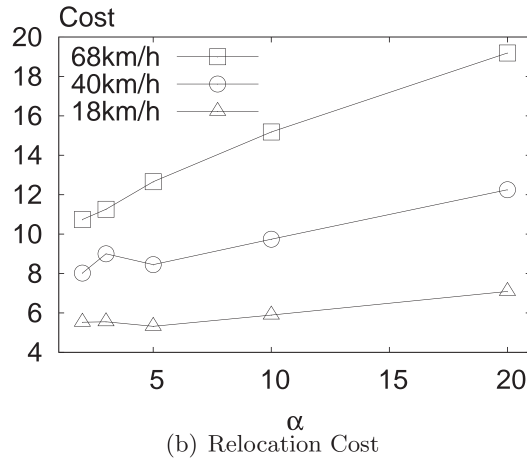
Figure 5.19: Privé Percentage of users involved in query

The percentage of the user population involved in answering a single K-request operation is shown in Figure 5.19(a) and 5.19(b). For small N values, at most 2% of all users are needed to answer a K-request, while for larger N, less than 0.5% of the users are required.

Relocation. Privé addresses user mobility by using an index update algorithm that attempts to resolve relocation at the lower levels of the hierarchy, in order to reduce both latency and communication cost. In our simulated scenario, we consider $10,000$ users across 20 consecutive time frames, with half of the indexed users moving at each time frame. We consider three ve-



(a) Relocation Latency



(b) Relocation Cost

Figure 5.20: PRIVÉ Relocation

locities: 68, 40 and 18km/h. Figure 5.20(a) and 5.20(b) show that relocation is efficiently handled: for the moderate $\alpha = 10$ value, the relocation is done on average in 2.5 hops for fast-moving users and 1.5 hops for slow-moving users. The dominant communication cost is that of the membership change propagation; for $\alpha = 10$ this cost is roughly a quarter compared to the cost of an index deletion followed by insertion for the $68km/h$ case, and 1/8 for $18km/h$.

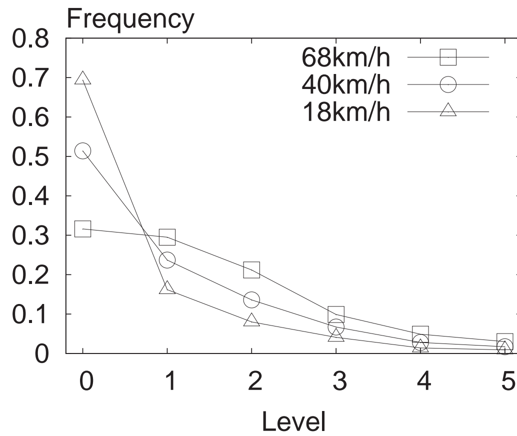


Figure 5.21: PRIVÉ Relocation Level

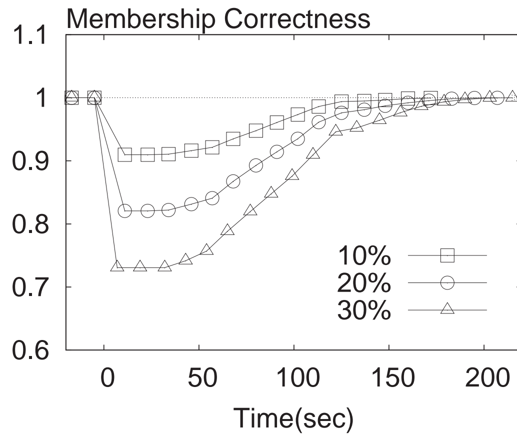
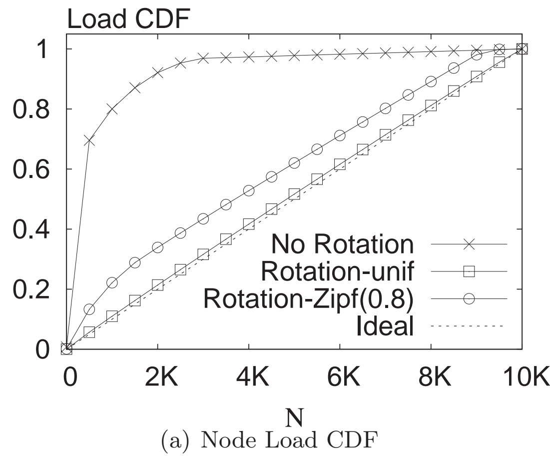


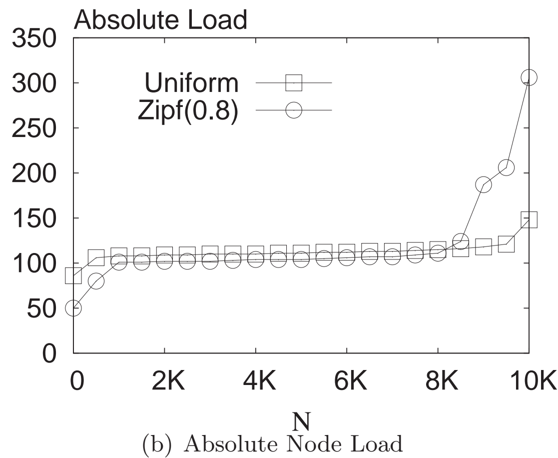
Figure 5.22: PRIVÉ Failure Recovery

Fig 5.21 shows the frequency of relocations completed at various levels of the hierarchy for a 6-level, $\alpha = 3$, $10,000$ users system. Most relocations are solved at the low levels of the hierarchy: for slow movement, 70% are solved at the leaf level and 86% at levels 0 and 1; for fast movement, 32% of relocations are completed at the leaf level, 63% at levels 0 and 1, and 86% at levels 0, 1 or 2.

Fault-tolerance. Starting with a system having correct cluster membership, we fail simultaneously 10, 20 or 30% of the nodes. We use maintenance timer values of 30 seconds for refreshing cluster membership and 60 seconds for purging a failed member. Figure 5.22 shows the evolution of membership state correctness over time (1 represents completely correct state). The



(a) Node Load CDF



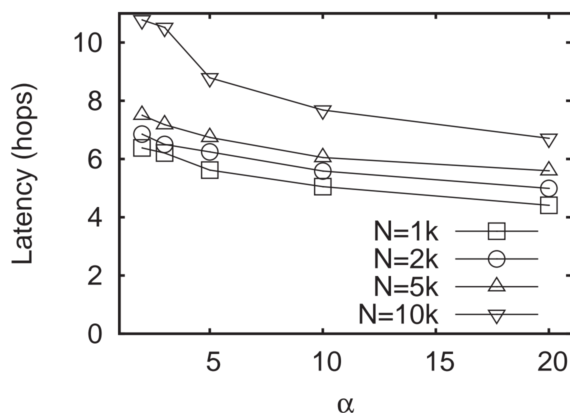
(b) Absolute Node Load

Figure 5.23: Privé Load Balancing

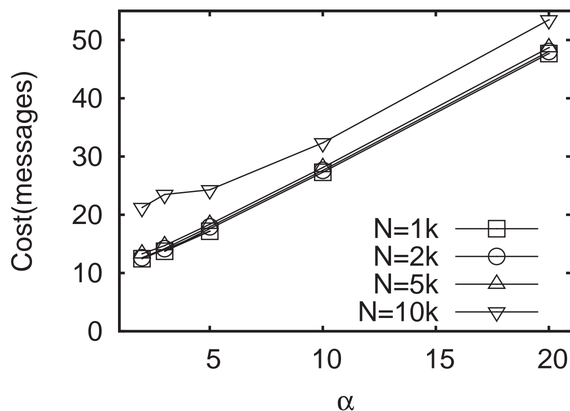
system recovers to a correct state within 3 purge cycles (138 sec) for 10% failure and 4 purge cycles (197 sec) for 30% failure.

Load-balancing. We measure the load incurred by each user for a $10,000$ users system, $\alpha = 5$, K=80, *load unit* = 200 messages (load unit is discussed in Section 5.2.4) and a simulated time of 1 hour, during which an average of 8 queries/user were generated. We consider both uniform and skewed (Zipf 0.8) query source distribution. Figure 5.23(a) shows the cumulative distribution function (CDF) of *sorted* user loads. The load is highly unbalanced if no rotation is performed, with 10% of users sustaining more than 80% of

the load. With rotation, for uniform query distribution, the load is close to the ideal one (i.e., diagonal line). For skewed query distribution, most of the users share equal load, while part of the users (roughly 10%) share a slightly higher load, as dictated by the fairness requirement discussed in Section 5.2.4. This is illustrated better in Figure 5.23(b) which shows the absolute load of each user.



(a) Latency

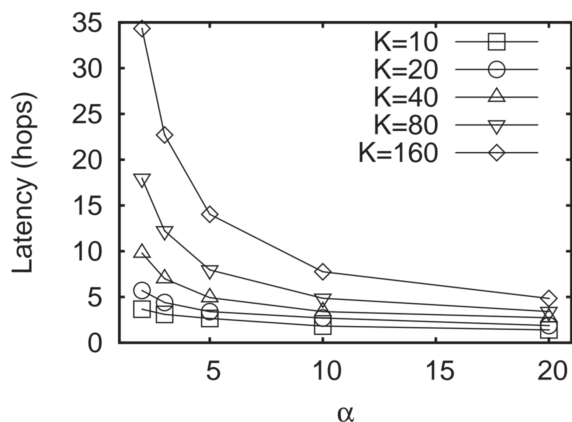


(b) Communication cost

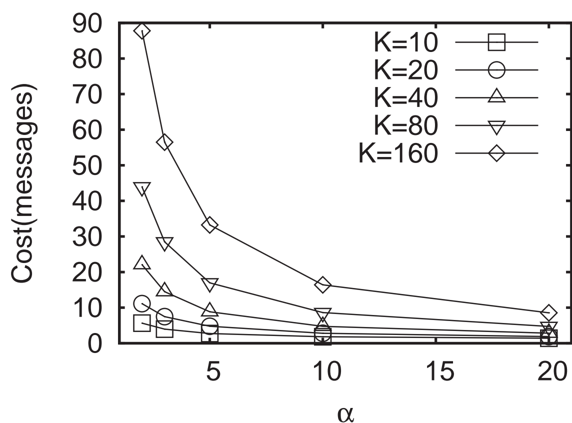
Figure 5.24: MobiHide Join

5.4.2 MobiHide protocol

Join. In this experiment, we measure the latency (i.e., number of hops) and communication cost (i.e., total number of messages) for the user join operation. Starting from a stable system, an additional 10% of the initial user population joins randomly the system. Figure 5.24(a) shows the latency for $N = 1$K, 2K, 5K and 10K users, for varying α (recall that the cluster size is between α and 3α). The plot confirms the theoretical expected complexity $O(\log N - log\alpha)$. For low α values, we observe a slight increase, due to the increasing proportion of split operations. In terms of communication cost (see Figure 5.24(b)), the dominant factor is $O(\alpha)$ due to the intra-cluster



(a) Latency



(b) Communication cost

Figure 5.25: MobiHide K-Request Operation

notification. There is a tradeoff between join latency and communication cost in terms of α. For low α values, the cluster maintenance cost is lower, but the latency increases. Furthermore, a low α also causes increased latency and communication cost during K-requests, as we will show shortly. Our experiments suggest that a value $5 < \alpha < 10$ is likely to yield good results in practice.

***K*-Request.** We consider a 10K user population with 10K uniformly distributed queries. Figure 5.25(a) and 5.25(b) show the average latency and communication for constructing the K-ASRs (α is varied). Both the latency

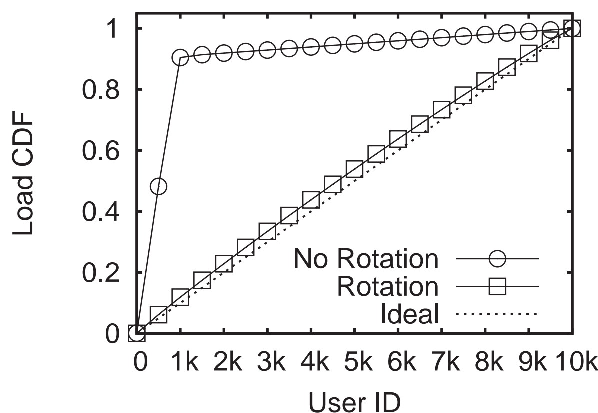


Figure 5.26: MobiHide Load Balancing

and communication cost are favored by larger α values. However, a compromise must be reached among the K-Request performance, maintenance cost and system scalability. Larger α determines higher maintenance cost and also yields a more centralized system, with inferior peak-load performance.

Load Balancing. Due to the hierarchical nature of MobiHide, the cluster heads that participate on the Chord ring bear more load than other cluster members. Here, we evaluate the rotation mechanism of MobiHide which aims at distributing the load evenly. We set α=5, K=20 and simulated a 10K user network, where an average of 3.6quh are generated. The total simulated time is 3 hours, and a rotation is triggered at every 300 messages received by a node. Figure 5.26 shows the cumulative distribution function (CDF) of the *sorted* node loads. Without rotation, the roughly 1,000 cluster heads (i.e., $10000/2\alpha$ as 2α is the average cluster size) bear 90% of the system load. With rotation, the load balancing is very close to the ideal (i.e., linear CDF, plotted as dotted line). Note that, for a load unit setting of 300 and a rotation cost of 2α messages, the rotation overhead is only $2\alpha/300 = 3\%$. This overhead can be decreased further by increasing the load unit.

Fault Tolerance. In this experiment we evaluate the fault-tolerance features of MobiHide. We consider 10K users and α=5. Chord performs periodical maintenance for its pointers. The respective timers are set at 3sec for the successor/predecessor, 10sec for the successor list and 30sec for

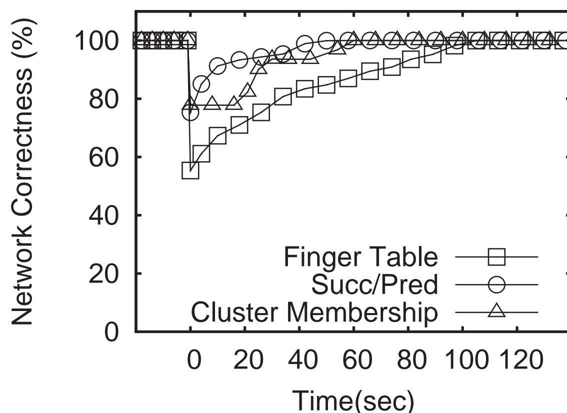


Figure 5.27: MobiHide Fault Tolerance

the finger table pointers. The intra-cluster beacon timer $\delta t = 10sec$. We consider three network correctness metrics: *(i)* the intra-cluster correctness, measured as the ratio of correct cluster membership entries out of the total entries, *(ii)* the succ/pred correctness, measured as the ratio of correct successors/predecessors over the total number of successor/predecessor pointers, and *(iii)* we define similarly the correctness of finger tables. Note that, for correct execution of K-request operations, only the successor/predecessor and intra-cluster membership need to be 100% accurate; the finger table pointers are only used for join and relocation operations, and their inaccuracy can only cause a slight increase in latency. Figure 5.27 shows the evolution in time of the three metrics, starting with a correct network, when 25% of the users fail simultaneously; $t = 0$ is the time of failure. We observe that the succ/pred and intra-cluster correctness are established after $60sec$. For the intra-cluster correctness, it takes the system roughly three purge intervals ($6\delta t$) to detect head failure, elect new leaders and establish correct cluster membership. The finger table is restored after $120sec$.

5.4.3 Privé and MobiHide Comparison

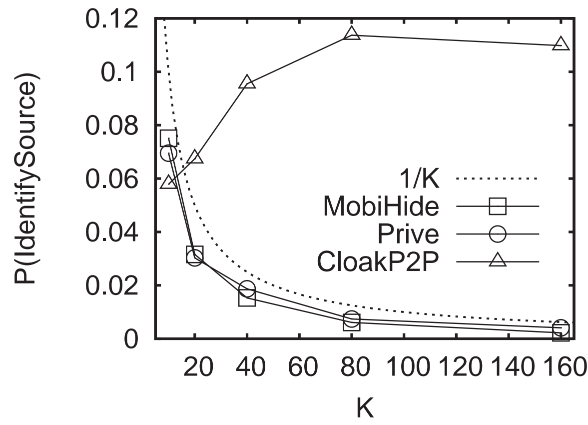
Anonymity Strength.

In Section 3.4, we have proved that *Hilbert Cloak* guarantees anonymity against location-based attacks, under any query distribution. Furthermore,

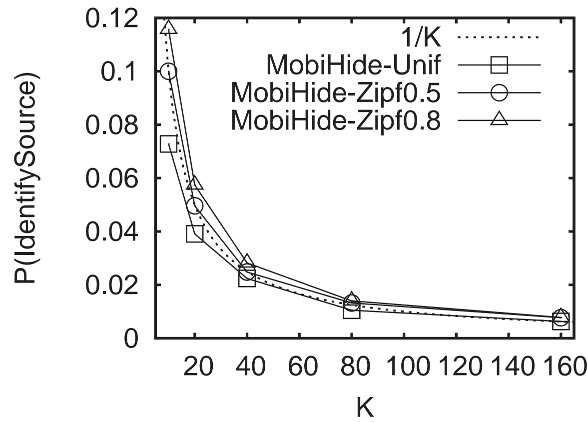
in Section 5.3, we proved theoretically that MobiHide guarantees K-anonymity for uniform query distribution.

First, we focus on the *center-of-ASR* attack, and we show a head-to-head comparison of MobiHide and Privé against *CloakP2P* [21]. We assume that an attacker knows (from an external source) the locations of all users, and employs a simple strategy which infers the query source as u_c, the user who is nearest to the center of the K-ASR. We consider a $10,000$ users scenario in which $10,000$ random queries are issued, according to a Zipf (i.e., skewed) query distribution with $\vartheta = 0.8$. In Figure 5.28(a) we plot the probability of u_c being the query source, for various values of K. The dotted line represents the value $1/K$; ideally, the performance of the algorithms should be *under* that line. For *CloakP2P*, if K=40, the probability of u_c being the query source is 10%, four times the $1/K$=2.5% maximum allowed bound. For larger values of K, the situation gets worse, as the number of users included in the K-ASR increases. The users are likely to come uniformly from all directions; hence, u_c is disclosed as the query source. On the other hand, Privé and MobiHide always satisfy the privacy bound. Note that, even if the anonymizing sets contain exactly K users, the corresponding MBRs may enclose a few more. This is why the results for Privé and MobiHide are not identical to the $1/K$ line.

In Figure 5.28(b) we consider the correlation attack (see Section 5.3.1). We assume the extreme case, where the attacker knows the exact locations of all users (recall that this attack is unlikely to occur in practice). We show the results for uniform and Zipf query distribution, with $\vartheta = 0.5$ and $\vartheta = 0.8$. As expected, for uniform distribution, anonymity is always preserved. Actually, in this case MobiHide behaves almost identical to Privé (not shown in the graph). Anonymity is also entirely preserved for $\vartheta = 0.5$. As the distribution becomes more skewed, MobiHide may fail to preserve anonymity by a small margin. In most cases, however, the probability of identifying the query source is very close to the theoretical bound $1/K$. In the worst case, for K= 160, $\vartheta = 0.8$, the probability of identifying the query source was $1.2/K$.



(a) *center-of-ASR* attack



(b) Correlation Attack

Figure 5.28: Anonymity Strength

Observe that in Figure 5.28(b) we did not consider *CloakP2P*, as it can be easily compromised by the much simpler *center-of-ASR* attack. Since it fails to provide anonymity in many cases, we will not consider *CloakP2P* any further.

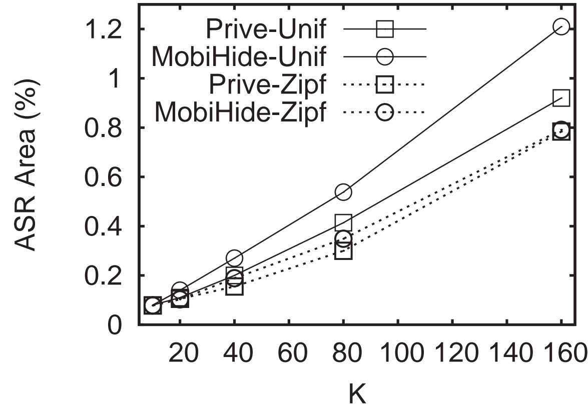


Figure 5.29: K-ASR Area

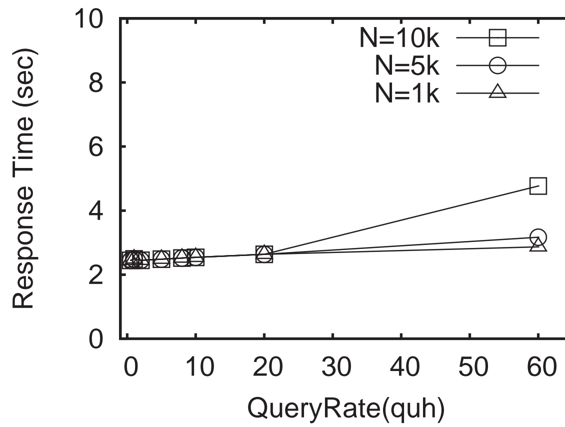
K-ASR Size.

MobiHide wraps around the Hilbert sequence in order to handle users near the start/end of the sequence. In some cases, this may yield K-ASRs with larger area, compared to Privé; consequently, the query processing cost will increase. To investigate this issue, we considered uniform and Zipf ($\vartheta = 0.8$) query distributions over a set of 10K users and varying K. In Figure 5.29 we plot the average area of the K-ASRs as a percentage of the entire dataspace. Observe that for the Zipf distribution the two systems behave almost identical, while for uniform distribution MobiHide generates 25% larger K-ASRs in the worst case. Therefore, we tradeoff at most 25% in additional query processing cost, but we obtain far superior system scalability as we will show next.

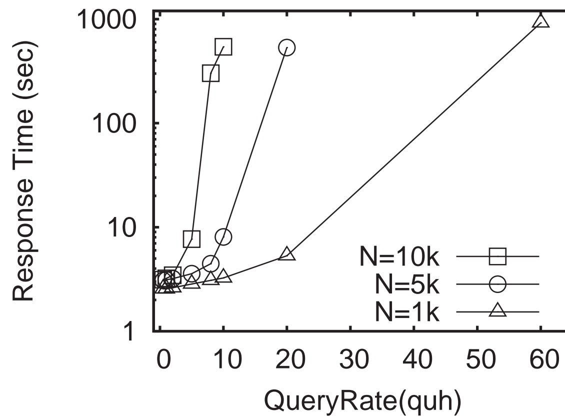
Scalability (response time).

The most important advantage of MobiHide is its increased scalability due to the highly decentralized structure. Here, we evaluate the response time of the system for 1K, 5K and 10K users. The querying users are selected with a Zipf ($\vartheta = 0.8$) distribution[7]. We use exponential distribution to model the query rate, and the mean is varied between 0.5 and 60quh (*Queries per User per Hour*). Processing time at each node is exponentially distributed with

[7] MobiHide behaves even better for uniform distribution of the querying user.



(a) MobiHide



(b) Privé

Figure 5.30: Scalability, $K = 40$

mean $50ms$. This is a realistic processing time that includes CPU processing and network buffer access. We set K=40 and inject queries for a period of $600sec$. From Figure 5.30(a), we can see that the response time is short (i.e, does not exceed $5sec$) even for large user populations and high query rates. Note that the experiment assumes unbounded message queues at the nodes; therefore the drop rate of requests is 0. We also considered bounded queues (size = 100); in the worst case, the drop rate was 3.4%.

In Figure 5.30(b) we repeated the same experiment for Privé. Observe that the response time grows sharply with the query rate, due to delays at the root node. For 10K users and $10quh$ the response time is almost

600sec (whereas, MobiHide needs only 2.5sec). Again, these results are for unbounded queues. For the bounded case (queue size = 100), the drop rate was 26% for 8quh; for 10quh the drop rate surges as high as 60%.

5.5 Discussion

Most previous work on spatial K-anonymity assumes a centralized anonymizer service architecture [33, 27, 49], whereas the only proposed distributed anonymization system [21] provides weak privacy features, and is vulnerable even to the simple "center-of-ASR" attack.

Our two proposed P2P anonymization infrastructures, Privé and MobiHide, address the limitations of previous work, and remove the central anonymizer bottleneck/single point-of-attack, while at the same time providing strong privacy and good system scalability. Our two proposed techniques provide an interesting trade-off between privacy and scalability. If theoretical guarantees on K-anonymity are required, Privé is the method of choice. On the other hand, if response time is of primary importance, even under periods of peak system load, MobiHide is the preferred technique, with good resilience to both "center-of-ASR" and correlation attacks.

Chapter 6

PIR Framework for LBS

6.1 Introduction

In this chapter, we propose a novel approach to LBS privacy, based on *Private Information Retrieval (PIR)*. This is the first work that integrates location privacy and PIR, with the following advantages over SKA approaches:

i. PIR eliminates the need for a trusted third party, whether it takes the form of an anonymizer service, or other users. For this reason, PIR offers much stronger privacy guarantees, and prevents attacks based on collusion among users, which SKA is vulnerable to

ii. since PIR privacy is not dependent on other users, it eliminates the need of expensive maintenance of the locations of a large number of subscribed users

iii. PIR does not disclose any information about user location, not even in perturbed (cloaked) form; location information is completely abstracted, and therefore any type of location-based attack is thwarted. In particular, PIR guarantees privacy for continuous queries

The rest of this chapter is organized as follows: Section 6.2 outlines an existing PIR protocol for binary data, which we use as a building block in our techniques. Section 6.3 discusses the advantages of our PIR framework, compared to existing spatial cloaking techniques. We introduce methods for approximate nearest-neighbor (NN) search in Section 6.4 and exact NN

Symbol	Description
k	Modulus Bits
q_1, q_2	$k/2$-bit primes
$N = q_1 \cdot q_2$	Modulus
n	Number of Data Objects
m	Object Size (bits)
$t = \lceil \sqrt{n} \rceil$	PIR Matrix Dimension
$M_{1:t,1:t}[1:m]$	PIR Matrix (binary array)
$y_{1:t}$, array of k-bit numbers	PIR Query
$z_{1:t}[1:m]$, array of k-bit numbers	PIR Reply

Table 6.1: Summary of notations

methods in Section 6.5. In Section 6.6, we present two optimizations targeted to reduce the computational overhead of PIR. We present the results of our experimental evaluation in Section 6.7.

6.2 Computational PIR Protocol

Computational PIR [42] relies on the *Quadratic Residuosity Assumption (QRA)*, which states that it is computationally hard to find the quadratic residues in modulo arithmetic of a large composite number $N = q_1 \cdot q_2$, where q_1, q_2 are large primes (see Table 6.1 for a summary of notations).

Define

$$\mathbb{Z}_N^* = \{x \in \mathbb{Z}_N | gcd(N, x) = 1\}, \tag{6.1}$$

the set of numbers in $\mathbb{Z}_N$ which are prime with N (gcd is the greatest common divisor). Then the set of *quadratic residues (QR)* modulo N is defined as:

$$QR = \{y \in \mathbb{Z}_N^* | \exists x \in \mathbb{Z}_N^* : y = x^2 \mod N\}. \tag{6.2}$$

The complement of QR with respect to $\mathbb{Z}_N^*$ constitutes the set of *quadratic non-residues (QNR)*.

Let

$$\mathbb{Z}_N^{+1} = \{y \in \mathbb{Z}_N^* | \left(\frac{y}{N}\right) = 1\}, \tag{6.3}$$

where $\left(\frac{y}{N}\right)$ denotes the Jacobi symbol [26]. Then, exactly half of the numbers in $\mathbb{Z}_N^{+1}$ are in QR, while the other half are in QNR. According to QRA,

for $y \in \mathbb{Z}_N^{+1}$, it is computationally intractable to decide whether $y \in QR$ or $y \in QNR$. Formally, define the quadratic residuosity predicate Q_N such that:

$$Q_N(y) = 0 \Leftrightarrow y \in QR \tag{6.4}$$

Then, if q_1 and q_2 are $\frac{k}{2}$-bit primes, for every constant c and any function $C(y)$ computable in polynomial time, there exists k_0 such that

$$\forall k > k_0, \Pr_{y \in \mathbb{Z}_N^{+1}}[C(y) = Q_N(y)] < \frac{1}{2} + \frac{1}{k^c} \tag{6.5}$$

Hence, the probability of distinguishing between a QR and a QNR is negligible for large-enough k.

Let $t = \lceil \sqrt{n} \rceil$ and consider that the database X is organized as a square $t \times t$ matrix M (the matrix is padded with extra entries if n is not a perfect square). Let $M_{a,b}$ be the matrix element corresponding to X_i that is requested by the user u. u randomly generates modulus N (similar to a public key in asymmetric cryptography), and sends it to the server, together with query message $y = [y_1 \dots y_t]$, such that $y_b \in QNR$, and $\forall j \neq b, y_j \in QR$.

The server computes for every row r of M the value

$$z_r = \prod_{j=1}^{t} w_{r,j} \tag{6.6}$$

where $w_{r,j} = y_j^2$ if $M_{r,j} = 0$, or y_j otherwise[1]. The server returns $z = [z_1 \dots z_t]$. Based on the *Euler criterion*, u computes the following formula:

$$\left(z_a^{\frac{q_1-1}{2}} = 1 \mod q_1\right) \wedge \left(z_a^{\frac{q_2-1}{2}} = 1 \mod q_2\right) \tag{6.7}$$

If Equation 6.7 is *true*, then $z_a \in QR$ else $z_a \in QNR$. Since u knows the factorization of N, Equation 6.7 can be efficiently computed using the Legendre symbol [26]. The user determines the value of $M_{a,b}$ as follows: If $z_a \in QR$ then $M_{a,b} = 0$, else $M_{a,b} = 1$.

Example 6.1. Figure 6.1 shows an example, where $n = 16$. u requests X_{10}, which corresponds to $M_{2,3}$. Therefore, u generates a message $y =$

[1]According to [56] the formula can be simplified as follows: $w_{r,j} = y_j$ if $M_{r,j} = 1$, otherwise $w_{r,j} = 1$

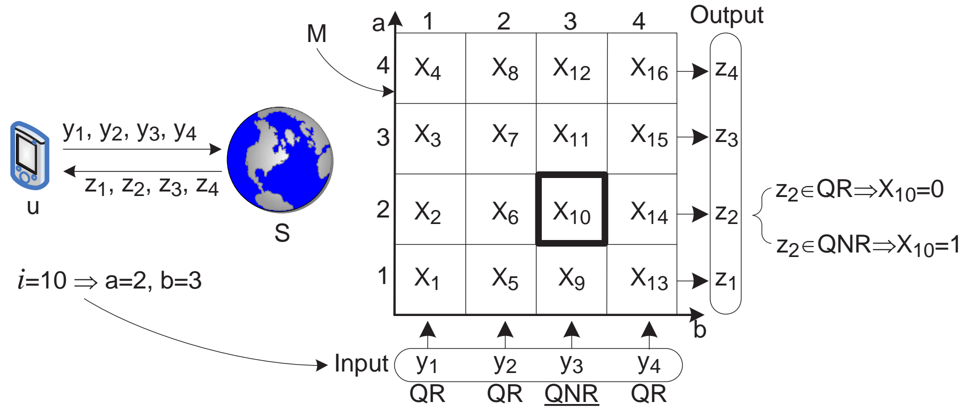


Figure 6.1: PIR example. u requests X_{10}

$[y_1, y_2, y_3, y_4]$, where $y_1, y_2, y_4 \in QR$ and $y_3 \in QNR$. The server replies with the message $z = [z_1, z_2, z_3, z_4]$. If $z_2 \in QR$ then u concludes that $X_{10} = 0$, else $X_{10} = 1$. □

The protocol requires $O(n)$ multiplications at the server, and $O(\sqrt{n})$ communication cost. The latter can be reduced to $O(n^{\varepsilon})$, $0 < \varepsilon < 1/2$, by applying the method recursively [42]. Although the recursive variation is asymptotically better that the basic one, our experiments revealed that the overhead of the recursion is not justified in practice.

The previous protocol retrieves privately one bit of information. The same idea can be extended to retrieve an object p_i which is represented as an m-bit binary string. Let D be a database containing n objects: $D = \{p_1, p_2, \ldots p_n\}$. Again, the server generates a matrix M with the difference that each cell contains an m-bit object. Conceptually, this is equivalent to maintaining m matrices $M[1], M[2], \ldots M[m]$, one for each bit of the objects. Assume that u requests object p_i. Same as the 1-bit case, u sends a message $y = [y_1 \ldots y_t]$. However, the server applies y to each one of the m matrices, resulting to m answer messages: $z[1], z[2], \ldots z[m]$. u receives these messages and computes all m bits of p_i. The communication and computation cost increase to $O(m\sqrt{n})$ and $O(m \cdot n)$, respectively. We use

$$PIR(p_i)$$

to denote that a user u retrieves privately an object p_i from the server, using

the described protocol.

6.3 PIR and Location-dependent Queries

There are two privacy issues in location-dependent queries: *(i)* The user must hide his identity (e.g., username, IP address, etc). This is orthogonal to our problem and can be achieved through a widely available anonymous web browsing service (that service does not learn the location of u). *(ii)* The user must hide his location. Similar to previous research on spatial K-anonymity (see Section 2.2), our PIR framework focuses on this issue. The advantages of our approach are:

PIR does not disclose *any* spatial information. As opposed to CR-based methods (which only perturb location, but still disclose the CR), no location information is disclosed. Instead, the data (i.e., POIs) are retrieved based on object index, by employing the provably private PIR protocol. This approach prevents any type of attack based on user location. In Sections 6.4 and 6.5, we develop methods to find the NN of a user with exactly one PIR request, irrespectively of his location.

PIR protects against correlation attacks. Assume that u asks a continuous query as he moves. Existing methods generate one cloaking region CR_i per location, but all CR_i will include u. By intersecting the set of users in all CR_i, an attacker can identify u with high probability; this is called *correlation attack*. Observe that this attack is possible because the CR reveals spatial information. Since the PIR framework does not reveal any spatial information, u is protected against correlation attacks.

PIR reduces significantly the identification probability. Let U be the set of all possible users (e.g., all mobile phone users within a country); $|U|$ is typically a large number (i.e., in the order of millions). From the server's point of view, the PIR request may have originated from any $u_i \in U$. Therefore, the probability to identify u as the querying user, is $1/|U|$. In contrast, existing techniques require a subset of users $U' \subset U$ to subscribe to the anonymization service; typically $|U'| \ll |U|$. Moreover, the number of users in the cloaking region CR must be $K \ll |U'|$, else CR grows large and the query cost becomes prohibitive (typically K is in the order of 10^2

[39, 49]). Therefore, the probability $1/K$ of identifying u is several orders of magnitude larger than that of the PIR framework.

PIR does not require any trusted third party, since privacy is achieved through cryptographic techniques. Existing techniques, on the other hand, need: (i) An anonymizer, which is a single point of attack, and (ii) A large set U' of subscribed users, all of whom must be trustworthy, since malicious users may collaborate to reveal the location of u. Furthermore, users in U' must accept the cost of sending frequent location updates to the anonymizer, even if they do not ask queries.

PIR reduces the number of disclosed POI. Existing SKA techniques may disclose a large set of candidate POIs (see the experimental evaluation of Section 3.6). Since the database is a valuable asset of the LBS, users may be charged according to the result size. We will show that PIR techniques disclose far fewer POIs.

6.4 Approximate Nearest Neighbors

In this section we describe our *ApproxNN* method, which employs the PIR framework to retrieve privately the nearest point of interest (i.e., NN) of u from a LBS. We show that a good approximation of the NN can be found with only one PIR request. For simplicity, in Section 6.4.1 we describe our method using the 1-D Hilbert ordering. In Section 6.4.2 we generalize to 2-D partitionings, such as kd-trees and R-trees.

6.4.1 Approximate NN using Hilbert ordering

The Hilbert space filling curve is a continuous fractal that maps the 2-D space to 1-D. Let p_i be a POI and denote its Hilbert value as $H(p_i)$. The Hilbert ordering of a database $D = \{p_1, p_2, \dots p_n\}$ is a list of all objects sorted in ascending order of their Hilbert values. Figure 6.2 shows an example database with 9 POIs $D = \{p_1, \dots p_9\}$, where $H(p_1) = 6$, $H(p_2) = 15$, etc. The granularity of the Hilbert curve is 8×8. The granularity does not affect the cost of our method, therefore it can be arbitrarily fine.

If two POIs are close in the 2-D space, they are likely to be close in the Hilbert ordering, as well [50]. Therefore, an approximation of the NN of u

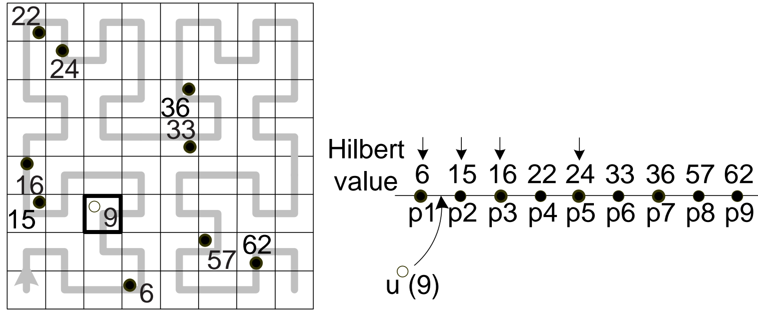


Figure 6.2: 9 POIs on a 8×8 Hilbert curve

is the POI p_i whose Hilbert value $H(p_i)$ is closest to $H(u)$. Since the POIs are sorted on their Hilbert value, we can use binary search to compute the approximate NN in $O(\log n)$ steps. In our example, $H(u) = 9$, therefore we retrieve $p_5 \rightarrow p_3 \rightarrow p_2 \rightarrow p_1$. The answer is p_1 since its distance from u in the 1-D space is $|H(p_1) - H(u)| = |6 - 9| = 3$, which is the smallest among all POIs. Note that the answer is approximate; the true NN is p_2.

There are two problems with this approach: First, since the search must not reveal any information, $O(\log n)$ costly private requests for $PIR(p_i)$ must be performed. Second, a side effect of the PIR protocol is that each $PIR(p_i)$ retrieves not one, but $\sqrt{n}$ POIs. Recall the example of Figure 6.1, where u is interested in X_{10}. The server returns z_1, z_2, z_3, z_4, from which u can compute the entire column 3 of M, i.e., $X_9, X_{10}, X_{11}, X_{12}$. Consequently, the binary search will retrieve $O(\sqrt{n}\log n)$ POIs, which represent a large fraction of the database.

Observe, however, that each PIR request is intuitively analogous to a "page access" on a disk. Therefore, the POIs can be arranged in a B$^+$-tree, where each node contains at most $\lceil\sqrt{n}\rceil$ POIs. The B$^+$-tree for our running example is shown in Figure 6.3.a; since there are 9 POIs, the capacity of each node is 3. Each entry in the root has a key and a pointer to a leaf. All Hilbert values in a leaf are less or equal to the corresponding root key. Each leaf node corresponds to one column of the PIR matrix M (see Figure 6.3.b). Note that M stores the POIs without their Hilbert value. Without loss of generality, we assume that each POI consists of its coordinates: $p_i = (x_i, y_i)$;

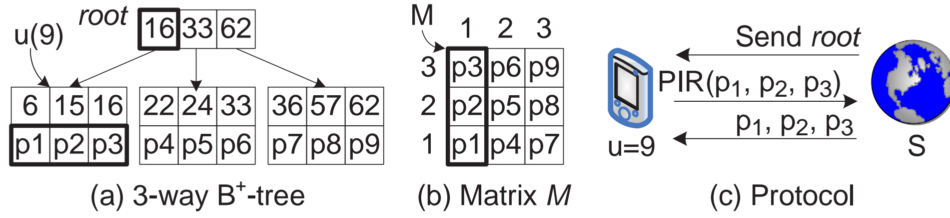


Figure 6.3: Approximate NN using Hilbert

Approximate NN Protocol

User u: Initiate query

Server: Send root node

User u: Let b be the column that includes u

$y = [y_1 : y_{\sqrt{n}}]$, $y_b \in QNR$, and $\forall j \neq b, y_j \in QR$

Send y

Server: Send $z[1:m] = [z_1 : z_{\sqrt{n}}][1:m]$

User u: Calculate distance to all POIs in column b

Return the approximate NN

Figure 6.4: Protocol for approximate NN

more complex objects are easily supported. During query processing the server sends to u the root node (i.e., $\langle 16, 33, 62 \rangle$). In the example $H(u) = 9 \leq 16$, therefore u must retrieve privately the first column of M. This is done with one request $PIR(\{p_1, p_2, p_3\})$. Next, u computes his NN from the set $\{p_1, p_2, p_3\}$. The answer is p_2, which happens to be the exact NN. Note that by retrieving several POIs in the neighborhood of u, the approximation error decreases; however, the method remains approximate.

Observe that the height of the tree is always $\log_{\sqrt{n}} n = 2$. The fact that u asks for the root node does not reveal any information to the server, since all queries require the root. Therefore, the server sends the root, which contains only Hilbert values but no POIs, in a low-cost plain format (i.e., does not use PIR). Consequently, the NN is computed with only *one* PIR request (i.e., one column of M). Figure 6.4 shows the protocol. The communication cost is $O(\sqrt{n})$ and u retrieves up to $\sqrt{n}$ POIs; for instance, if the LBS contains 10^6 POIs, u retrieves 0.1% of them. In Section 6.7 we show that existing methods, which employ an anonymizer, typically retrieve more POIs.

6.4.2 Generalization to 2-D partitionings

The previous method can be extended to 2-D partitionings. The only requirement is that data must be partitioned into at most $\sqrt{n}$ buckets, each containing up to $\sqrt{n}$ POIs. Consider the case of *kd*-tree [23]. The original insertion algorithm partitions the space either horizontally or vertically such that every partition contains one point. We modify the algorithm as follows: Let n' be the number of POIs in the current partition (initially $n' = n$), and let g be the number of remaining available partitions (initially, there are $\sqrt{n}$). We allow splits that create partitions e_1 and e_2 such that $|e_1| + |e_2| = n'$ and

$$\lceil |e_1|/\sqrt{n}\ \rceil + \lceil |e_2|/\sqrt{n}\ \rceil \leq g. \tag{6.8}$$

Then, the algorithm is recursively applied to e_1 and e_2, with $\lceil |e_1|/\sqrt{n}\ \rceil$ and $\lceil |e_2|/\sqrt{n}\ \rceil$ remaining partitions respectively. Out of the eligible splits, we choose the most balanced one.

In the example of Figure 6.5.a there are $n = 9$ POIs, and 3 available buckets. The points are split into region A which contains $|A| = 3$ POIs and BC which contains $|BC| = 6$ POIs. BC is further split into B (where $|B| = 3$) and C (where $|C| = 3$). The resulting *kd*-tree has 2 levels. The root contains regions A, B, C and the leaf level contains 3 nodes with 3 POIs each, which are arranged in a PIR matrix M. Query processing follows the protocol of Figure 6.4. Since u is in region C, column 3 is retrieved; the NN is p_2.

As another case study, consider the R-tree. Originally, each node would store between $f/2$ and f objects, where f is the node capacity; internal nodes contain minimum bounding rectangles (MBR) which enclose the objects of their children. We modify the R-tree construction algorithm such that there are 2 levels and the root contains no more than $\sqrt{n}$ MBRs. Let n' be the number of POIs in the current partition. The original algorithm checks all possible partitionings with $|e_1| + |e_2| = n'$ POIs, along the x and y-axis. It selects the best one (e.g., lowest total area, or total perimeter, etc) and continues recursively. We modify this algorithm to validate a split only if Equation 6.8 is satisfied. Figure 6.5.b shows an example where MBRs

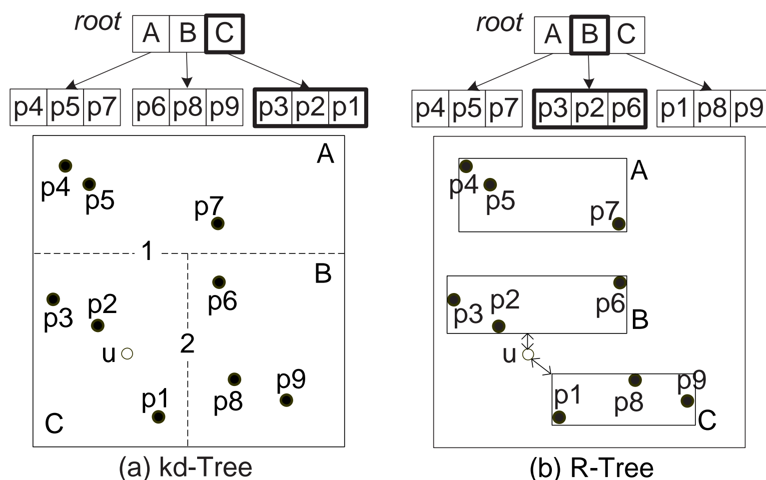


Figure 6.5: 2-D approximate NN

A, B, C contain 3 POIs each. The leaf nodes are arranged in a PIR matrix M and query processing follows the protocol of Figure 6.4. u is closer to MBR B, therefore column 2 is retrieved and the NN is p_2.

Both 2-D methods return the approximate NN by retrieving $\sqrt{n}$ POIs. The communication cost is $O(\sqrt{n})$. Therefore, in terms of cost, they are the same as the Hilbert-based method. The only difference is the approximation error, which depends on the characteristics of the dataset (e.g., density, skew). The case studies of the *kd*-tree and R-tree demonstrate a general method for accommodating any partitioning in our PIR framework. The choice of the appropriate partitioning for a specific dataset is outside the scope of this thesis. Note that, all variations of ApproxNN can also return the approximate i^{th}-Nearest Neighbor, where $1 \leq i \leq \sqrt{n}$.

6.5 Exact Nearest Neighbors

In this section we present a method, called *ExactNN*, which returns the POI that is the exact nearest neighbor of user u. In a preprocessing phase, ExactNN computes the Voronoi tessellation [23] of the set of POIs (see Figure 6.6). Every Voronoi cell contains one POI. By definition, the NN of any point within a Voronoi cell is the POI enclosed in that cell. ExactNN

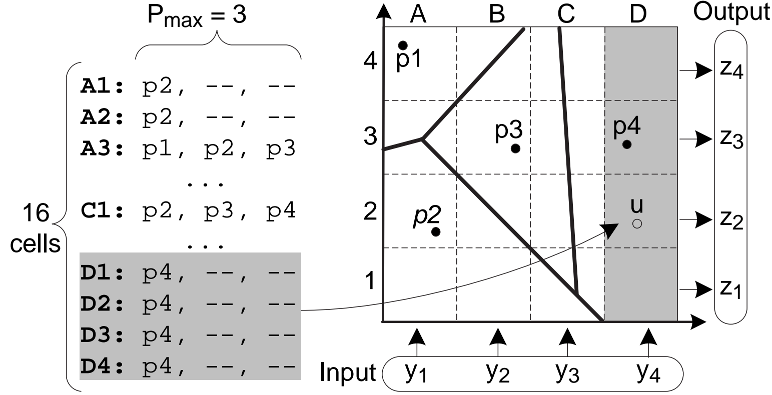


Figure 6.6: Exact nearest neighbor

superimposes a regular $G \times G$ grid on the Voronoi diagram. Then, for every cell c of the grid, it determines all Voronoi cells that intersect it, and adds the corresponding POIs to c. Hence, cell c contains all potential NNs of every location inside it. For example, Figure 6.6 depicts a 4×4 grid, where cell $A1$ contains $\{p_2\}$, cell $B2$ contains $\{p_2, p_3\}$, etc. During query processing, u learns the granularity of the grid; therefore he can calculate the cell that encloses his location (i.e., $D2$ in our example). Then, u issues a private request $PIR(D2)$; from the contents of $D2$ u finds his NN (i.e., p_4).

In contrast to ApproxNN methods, the objects of the PIR matrix M of ExactNN are not the POIs. Instead, each object in M corresponds to the contents of an entire grid cell c. For instance, our example contains 4 POIs (i.e., p_1, p_2, p_3, p_4), but M contains 16 objects, since there are 16 cells in the grid. In the previous section, n (i.e., the number of objects in M) was the same as the number of POIs. To avoid confusion, n still refers to the number of objects in M (i.e., $n = 16$ in the example) and we use $|POI|$ to denote the number of POIs.

All objects in M must have the same number of bits, otherwise the server may infer the requested cell based on the amount of bits transferred. Let P_{max} be the maximum number of POIs per grid cell. If a cell has fewer than P_{max} POIs, we add dummy POIs as placeholders. In our example, $P_{max} = 3$ because of cells $A3$ and $C1$. Therefore, all other cells are padded with dummy POIs. For instance, cell $A1$ becomes $\{p_2, -, -\}$. Recall from

Exact NN Protocol

User u: Initiate query

Server: Send grid granularity G

User u: Let b be the column that includes u

$y = [y_1 : y_{\sqrt{n}}]$, $y_b \in QNR$, and $\forall j \neq b, y_j \in QR$

Send y

Server: Send $z[1 : m] = [z_1 : z_{\sqrt{n}}][1 : m]$

User u: Let a be the row that includes u

Discard dummy POIs in z_a

Calculate distance to real POIs in z_a

Return the exact NN

Figure 6.7: Protocol for exact NN

Table 6.1 that m denotes the number of bits of each object in M. Since there are P_{max} POIs in each object, $m = |p_i| \cdot P_{max}$, where $|p_i|$ is the number of bits in the representation of each POI.

Since the number of objects in M is $n = G^2$, depending on the granularity of the grid, n may be larger or smaller than the number of POIs. P_{max} (hence m, too), also depends on G. Therefore the communication and computation cost of ExactNN depend on G. In Section 6.5.1 we discuss how to select an appropriate value for G.

The protocol for ExactNN is shown in Figure 6.7. It is similar to the ApproxNN protocol, with one difference: Let $\langle a, b \rangle$ be the cell that contains u, where a is the row and b the column. u issues a private request $PIR(\langle a, b \rangle)$. Recall that, in addition to $\langle a, b \rangle$, the byproduct of this request are the POIs of the entire column b. ApproxNN would utilize the extra POIs to improve the approximation of the result. On the other hand, the extra results are useless for ExactNN, since the exact NN is always in $\langle a, b \rangle$. A possible concern is that ExactNN reveals to the user $G \cdot P_{max}$ POIs, which may be more than those revealed by ApproxNN. In practice, however, this is not a problem because column b includes many duplicates. For example, cells $D1, D2, D3, D4$ in Figure 6.6 all contain the same POI p_4; therefore the request $PIR(D2)$ reveals only p_4 to the user. In Section 6.6.2 we discuss an optimization which reduces further the number of revealed POIs.

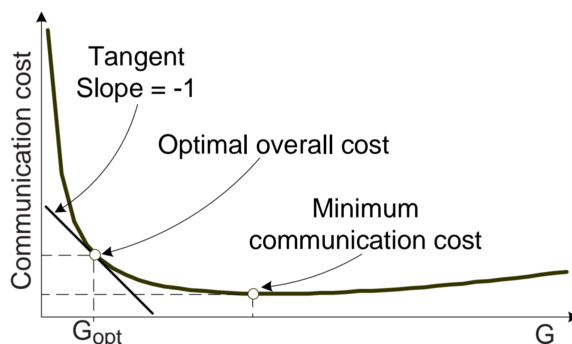


Figure 6.8: Finding the optimal grid granularity

6.5.1 Grid Granularity

For a particular choice of grid granularity G, the PIR protocol overhead of ExactNN is $k \cdot G + k \cdot m \cdot G$ communication (the first term corresponds to request y; the second to reply z), and $O(m \cdot G^2)$ server computation (recall that $m = |p_i| \cdot P_{max}$). By increasing G (i.e., finer grid), P_{max} may decrease or remain the same, depending on the data characteristics. Figure 6.8 shows the general form of the communication cost, as a function of G. Initially the cost decreases fast because P_{max} decreases, but later the cost increases again at finer granularity, as P_{max} reaches a lower bound (either 1, or the maximum of duplicate POIs). We could select the value of G that minimizes the communication cost, but there is a tradeoff, as the CPU cost increases quadratically to G. We could include the CPU cost in the graph and find the granularity that minimizes the total cost (expressed as response time). This would require the exact CPU speed and network bandwidth; the latter is problematic, since the bandwidth of each user differs. A good tradeoff is to select the granularity G_{opt} near the point where the rate of decrease of the communication cost slows down. That is the point where the slope of the tangent of the cost function becomes -1.

In practice, since P_{max} is not known in advance, the graph of Figure 6.8 is generated as follows: First, we compute the Voronoi diagram of the dataset. Then, we select a set of values G_i using random sampling. For each of these values, we superimpose the resulting grid on the Voronoi diagram,

and calculate P_{max} by counting the POIs in each cell. The communication cost is $C_i(G_i) = k \cdot G_i + k \cdot m \cdot G_i$. Finally, we apply curve fitting on the $\langle G_i, C_i(G_i) \rangle$ points to obtain the complete curve.

6.6 Optimizations

This section presents optimizations that are applicable to the previous methods. By employing these optimizations, the communication cost is reduced by as much as 90%, whereas the computation cost is reduced by up to 40% for a single CPU and more for multiple CPUs.

6.6.1 Compression

The size of z (i.e., the server's answer) is $k \cdot m \cdot r$ bits, where r is the number of rows in the PIR matrix M. However, there is a lot of redundancy inside z. Consider the example of Figure 6.6. Cells $A4, B4, C4, D4$ have at least one dummy object each. Assuming that the dummy object corresponds to bits $m_i \dots m_j$, then all $z_1[m_i : m_j]$ results will be the same. Since each one of these results is k bits, the redundancy is significant. In our implementation we use standard compression techniques to compress the result. Our experiments showed that, in many cases, compression may save up to 90% of the communication cost.

6.6.2 Rectangular vs. Square PIR Matrix

In the previous sections the PIR matrix M is assumed to be square. However, M can have any rectangular shape [42] with r rows and s columns (see Figure 6.9). The shape of M does not affect the CPU cost, since the number of multiplications does not change. On the other hand, the communication cost becomes: $C(r, s) = k \cdot s + k \cdot m \cdot r$, where the first part is the size of the user's request $y_{1..s}$ and the second part is the size of the server's answer $z_{1..r}$. $C(r, s)$ is minimized for:

$$r = \left\lceil \sqrt{\frac{n}{m}} \right\rceil, \quad s = \left\lceil \frac{n}{r} \right\rceil \tag{6.9}$$

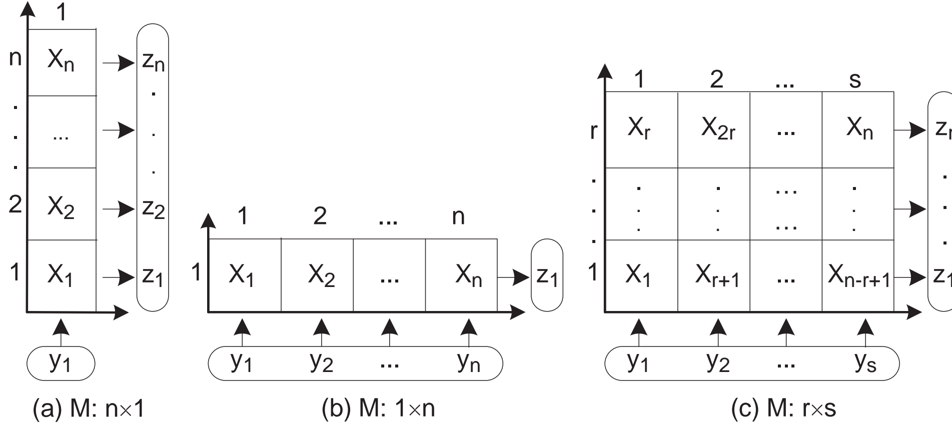


Figure 6.9: Rectangular PIR matrix M

If each object has 1 bit (i.e., $m = 1$), $C(r, s)$ is minimized for $r = s = \sqrt{n}$ (i.e., square matrix). In our ExactNN method, on the other hand, $m \gg 1$; therefore, the communication cost is minimized for r smaller than s. Rectangular matrices have an additional benefit: they can reduce the number of POIs that the user learns. Consider the example of Figure 6.9.a, where $r = n$ and $s = 1$. The server returns $z_{1..n}$, therefore, the user learns n POIs. On the other hand, in Figure 6.9.b $r = 1$ and the server returns only 1 POI. By using rectangular M in the ExactNN algorithm, the user learns up to $r \cdot P_{max}$ POIs. This is much less than the $\sqrt{n} \cdot P_{max}$ POIs that a square matrix would reveal.

Rectangular M could also reduce the communication cost in the ApproxNN methods, since $m \gg 1$. However, there is a drawback: Recall that the ApproxNN methods organize POIs in an index, whose root node is always sent to the user. The size of the root is equal to the number of columns s. In the extreme case (i.e., for large enough m), Equation 6.9 results in $s \approx n$, therefore the root node reveals the entire database to the user, defeating the purpose of PIR. The minimum number of revealed POIs (i.e., $O(\sqrt{n})$) is achieved for square M. In our implementation we use a square matrix M for the ApproxNN methods.

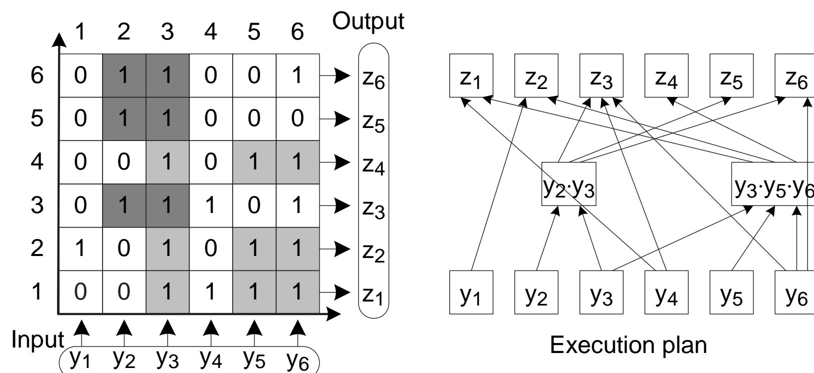


Figure 6.10: Pre-compiled optimized execution plan

6.6.3 Avoiding Redundant Multiplications

From Equation 6.6 (Section 6.2), it is clear that a PIR request requires $m \cdot n$ multiplications with $y_i \in y$. Each y_i is a k-bit number; to ensure that factorization is hard, k needs to be in the order of hundreds. Therefore, the CPU cost of the multiplications is high. Nevertheless, many multiplications are redundant, since they are repeated several times. In this section we propose an optimization technique, which employs data mining to avoid redundant multiplications. Although in this work we only evaluate the effectiveness of the proposed optimization for the location privacy problem, our technique is general and can be used in other PIR applications.

By using the simplification of [56] (Section 6.2), in each row of the PIR matrix we only need to consider the '1' bits. For example, in Figure 6.10, the result for row 1 is: $z_1 = y_3 \cdot y_4 \cdot y_5 \cdot y_6$. Observe that the partial product $y_{356} = y_3 \cdot y_5 \cdot y_6$ appears in rows 1, 2 and 4. If y_{356} is computed once, it can be reused to compute $z_1 = y_{356} \cdot y_4$, $z_2 = y_{356} \cdot y_1$ and $z_4 = y_{356}$, thus saving many multiplications. The same idea applies to y_{23}, which appears in rows 3, 5 and 6.

Intuitively, the previous idea can be implemented as a "cache". When a new PIR request arrives, the server starts processing it and stores the partial results in the cache. If a partial product is repeated, the corresponding partial result is retrieved from the cache. Unfortunately, the number of possible partial products is 2^s, where s is the number of columns in M.

BuildExecutionPlan

Input: transaction T_i (from row i of M),
list of frequent itemsets IT

```
1.  ExecPlan_i = ∅
2.  foreach itemset it_j ∈ IT
3.    if (¬T_i ∧ it_j = 0) /*it_j is part of T_i*/
4.      ExecPlan_i = ExecPlan_i ∪ {it_j}
5.      T_i = ¬it_j ∧ T_i
6.      if (T_i = 0) /*no more '1's in T_i*/
7.        break
8.  output ExecPlan_i
```

Figure 6.11: Execution plan for one row

s can be in the order of thousands, therefore the method is prohibitively expensive for on-line use.

Observe that, although the result depends on the input y, the set of multiplications depends only on the server's data and is the same for any PIR request. Therefore, similarly to pre-compiled query plans in databases, we generate in an off-line phase an optimized execution plan that avoids redundant multiplications. Then, during query processing, the server routes the input y through the operators of the plan, in order to compute fast the result z. The execution plan for our running example is shown in Figure 6.10.

In the off-line phase, we employ data mining techniques to identify redundant partial products. Following the data mining terminology, each item corresponds to one column of matrix M, whereas each transaction corresponds to a row of M. For example, row 1 in Figure 6.10 corresponds to transaction $T_1 = 001111$. A '1' bit means that the corresponding item belongs to the transaction. There are $r \cdot m$ transactions with s items each. An itemset corresponds to a partial product. In order to avoid many multiplications, we must identify frequent and long itemsets. We use the *Apriori* algorithm [8]. Initially, Apriori considers all itemsets with one item and prunes those that do not appear in at least f_{min} transactions. Then, it considers all possible combinations with two of the remaining items and continues recursively with itemsets containing more items.

Accessing the execution plan incurs an overhead on query execution.

Therefore, the frequency and length of the discovered itemsets must be large enough such that the savings from the multiplications are more than the overhead. The cut-off values for frequency and length can be estimated by measuring the actual multiplication time of the particular CPU. Moreover, by decreasing f_{min} the running time of Apriori increases. Therefore, f_{min} must be selected such that Apriori finishes within a reasonable time. Note that the identification of frequent itemsets is a costly operation, therefore it is not appropriate for databases with frequent updates. However, in many LBSs updates are infrequent (e.g., hospitals change rarely). Similar to data warehouses, our method is appropriate for batch periodic updates (e.g., once per night).

Let $IT = (it_1, it_2, \ldots)$ be the list of frequent itemsets sorted in descending order of itemset length. In the example of Figure 6.10, $IT = (001011, 011000)$ which corresponds to y_{356} and y_{23}. We use the following greedy algorithm to build the execution plan for row z_i: Let T_i be the transaction that corresponds to z_i. We traverse the list IT and select the first (i.e., longest) itemset it_j which appears in T_i. The rationale for this heuristic is that longer itemsets correspond to longer partial products, hence they are preferred for their higher potential in multiplication savings. We include it_j in the execution plan of T_i, remove from T_i all items in it_j (this step is necessary in order to ensure correctness) and repeat the process for the rest of itemsets in IT. The pseudocode is shown in Figure 6.11 (lines 3 and 5 use bitwise operations for performance reasons). The same process is repeated for all rows of M.

Figure 6.12 shows the architecture of the PIR optimizer. Once a query is received, the server checks for each row the associated execution plan $ExecPlan_i$: for each itemset $it \in ExecPlan_i$, the server checks whether the partial product of it has already been tabulated in table $PROD$; if so, it is used directly, otherwise, the server computes the product and stores it in $PROD$ to be used for subsequent rows. The overhead of this technique consists of the lookup in the $PROD$ table, which can be efficiently manipulated as a hash table, having as key the signature of it. The experiments show that, by using the optimized execution plan, the computation cost is reduced by up to 40%.

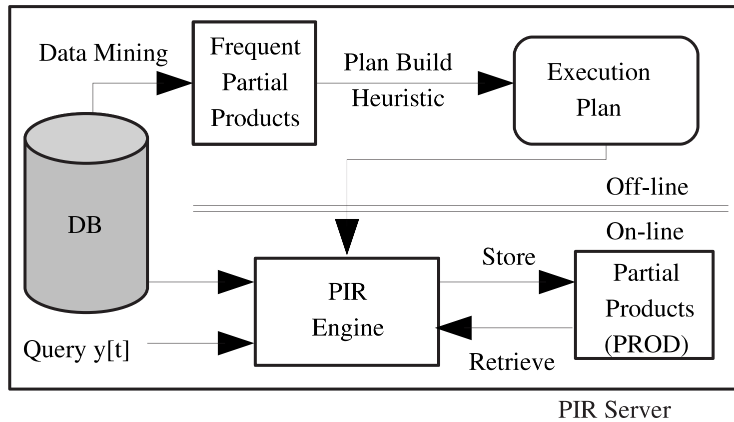


Figure 6.12: PIR Optimizer Architecture

6.6.4 Parallelism

The PIR framework involves a large number of multiplications in a regular pattern. Consequently, the computations can be easily parallelized. The parallel computing infrastructure can vary from multicore CPU, to multi-CPU to computer cluster. Matrix M is partitioned horizontally in as many partitions as the available CPUs, and each CPU receives the corresponding partition in an off-line phase. During query processing, all CPUs receive the input vector y and calculate their part of the result. Communication is minimal (only the input and output) since each partition does not depend on the others. Therefore, parallel implementations achieve almost linear speedup. In our experiments we used up to 8 CPUs resulting to almost 7 times faster execution time.

6.7 Experimental Evaluation

We developed a C++ prototype of the proposed PIR framework. We tested the methods using both synthetic (uniform and Gaussian) and real (Sequoia[2], 65K POIs in California) datasets. Our experimental testbed consisted of a Pentium 4 3.0GHz machine with 2GB of RAM, running Linux OS.

[2]http://www.rtreeportal.org

We employed the *GMP*[3] library for operations with large integers (required by PIR), and the *zlib*[4] library for data compression. In our experiments, we measured the communication cost, as well as the computation cost at the server, which is the dominating factor for PIR. The CPU time includes the compression of the result before returning it to the client (which only accounts for a small fraction of the total CPU time). We also measured the computation cost at the client. We varied k (i.e., modulus bits) between 256 and 1280, and the number of POIs between $10,000$ and $100,000$. Each POI consists of its (x, y) coordinates (i.e., 64 bits).

6.7.1 1D and 2D Approximate NN

First we compare the approximate NN methods. 1D refers to the Hilbert variant, whereas 2D refers to the R-tree variant. Figure 6.13.a shows the server CPU time with varying k for the real Sequoia set. Recall that, for approximate methods, n is the number of POIs. The CPU time is very similar for both 1DApprox and 2DApprox, since in both cases it mainly depends on the data size. CPU time varies approximately as $k\sqrt{k}$, which is the average complexity of the multiplication algorithms implemented in GMP.

Figure 6.13.b, shows the communication cost, which is linear to k. The cost for 2DApprox is slightly lower due to compression. Compression is more effective for 2DApprox, especially for skewed data, because the R-tree clustering algorithms have good locality, and many POIs with similar coordinates are grouped together in the same "index node" (i.e., PIR matrix column), therefore increasing data redundancy. For $k = 768$, the communication cost is 1MB.

Figure 6.14.a shows the CPU time for varying data size (synthetic sets) and $k = 768$. The CPU time is linear to n, since the number of multiplications is proportional to the number of '1' bits in the data. The communication cost follows the expected theoretical dependency of $\sqrt{n}$, as shown in Figure 6.14.b. Compression is more effective with Gaussian data, because

[3]http://gmplib.org/

[4]http://www.zlib.net

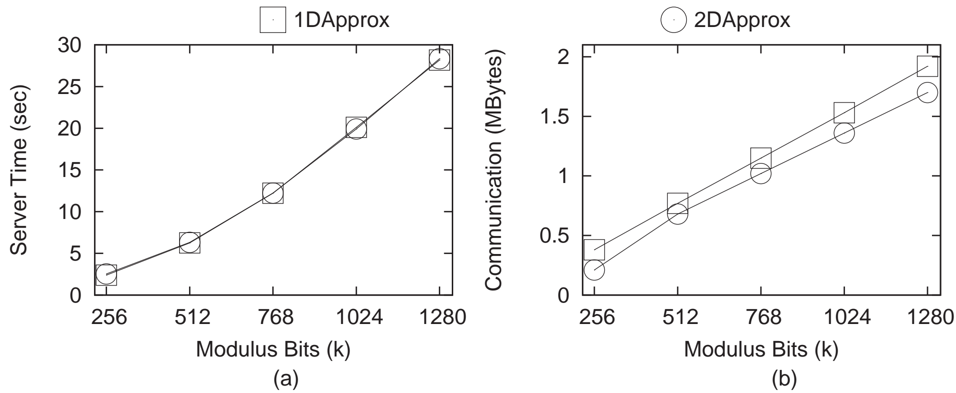


Figure 6.13: Variable k, Sequoia set (62K POI)

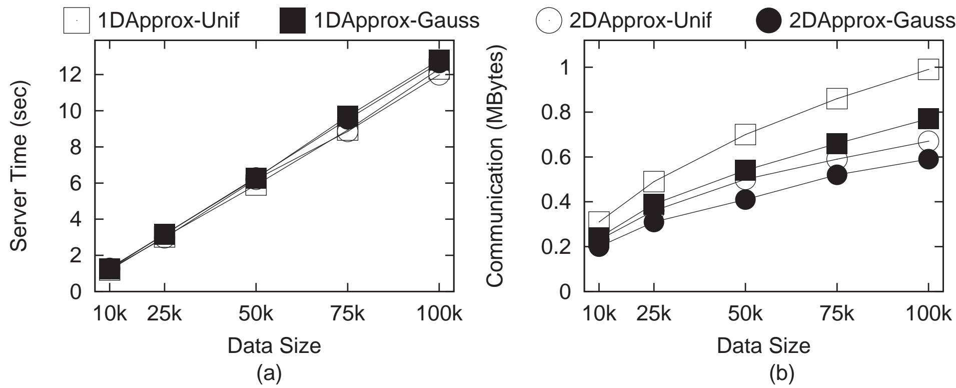


Figure 6.14: Variable data size, $k = 768$ bits

there are more POIs with nearby (possibly identical) coordinates, increasing redundancy.

Next, we investigate the approximation error of the proposed techniques. We generate 1000 queries originating at random locations that follow the POI distribution (this is a reasonable assumption, since the dataset is likely to correspond to an urban area, for instance). Given query point q, the returned result r and actual NN p, we express the approximate NN error as $err = (dist(q, r) - dist(q, p))/maxD$, where $maxD$ is the side of the (square) data space.

Figure 6.15 shows the average error for 1DApprox and 2DApprox. The error is slightly larger for uniform data, as POIs are scattered in the entire dataspace. For Gaussian data, the clustering of POIs in the PIR matrix

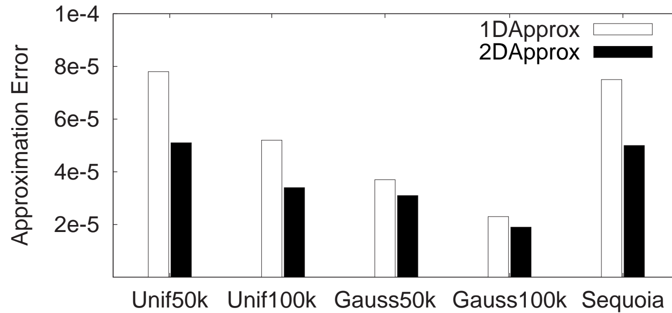


Figure 6.15: Approximation Error

	10K	**25K**	**50K**	**75K**	**100K**
Uniform	20x20	22x22	28x28	32x32	36x36
Gaussian	42x42	61x61	78x78	108x108	122x122
Sequoia	104x104				

Table 6.2: Grid Granularity for ExactNN

is more effective, leading to better accuracy. Furthermore, the error decreases when data density increases. The error is always under 0.01% of the dataspace size.

1DApprox and 2DApprox have similar CPU time and comparable communication cost, since they both follow the same 2-level tree approach. The choice between the two depends on the characteristics of the data and is outside the scope of this work. Due to the similar performance, we only consider 1DApprox for the rest of the experiments.

6.7.2 Exact Methods

We evaluate the performance of ExactNN in comparison with 1DApprox. The grid size of ExactNN was determined as described in Section 6.5.1. Table 6.2 shows the resulting grid size for each dataset. Figure 6.16.a depicts the CPU time versus k for the real dataset. The trend is similar to approximate methods, but the absolute values are higher for ExactNN, due to the larger size of the PIR matrix (recall that the m value for ExactNN may be considerably larger than that for 1DApprox). In Section 6.7.3 we evaluate methods that reduce the CPU time. Figure 6.16.b confirms that the communication cost is linear to k.

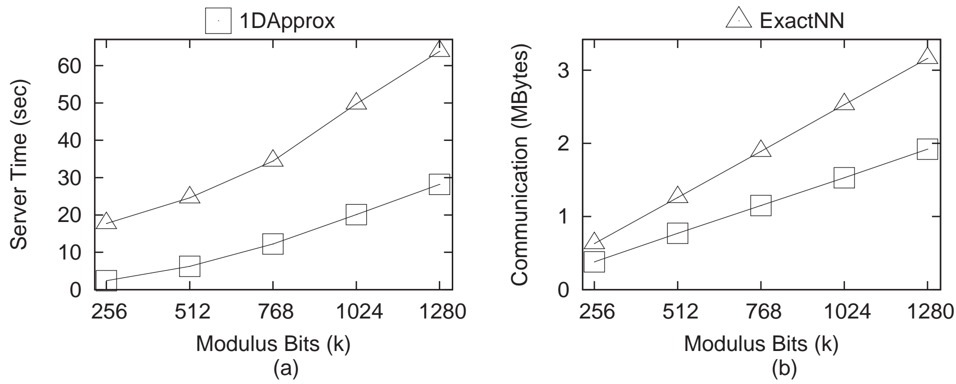


Figure 6.16: Variable k, Sequoia set (62K POI)

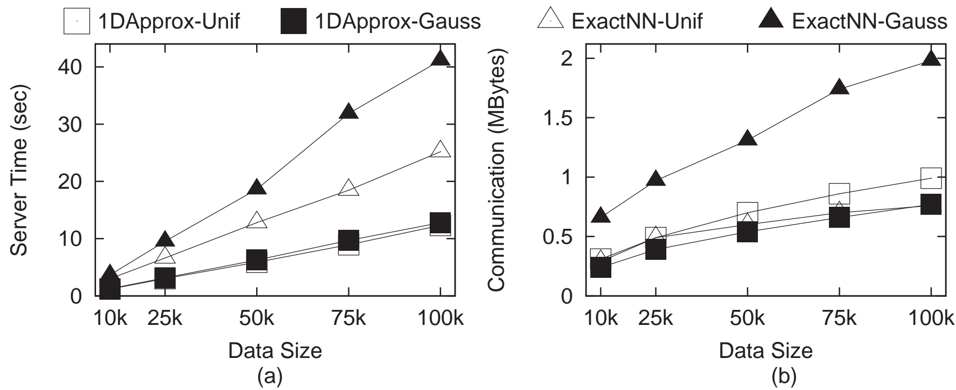


Figure 6.17: Variable data size, $k = 768$ bits

Figure 6.17.a shows the CPU time versus the data size. Recall that n for ExactNN depends on the grid granularity, and is *not* equal to the data size. For uniform data, the number of grid cells (i.e., n value) required to maintain a constant P_{max} grows proportionally with data size, therefore the CPU time increases linearly. On the other hand, for skewed data, in order to maintain a value of m which provides low communication cost, it may be necessary to use a finer grid, resulting in increased CPU time. However, the results show that the CPU time is almost linear to the number of POI, confirming that the heuristic for choosing the grid granularity is effective. The good choice of granularity is also reflected in the communication cost (Figure 6.17.b). Observe that, for Gaussian data, P_{max} (hence m) increases, and consequently the communication cost increases.

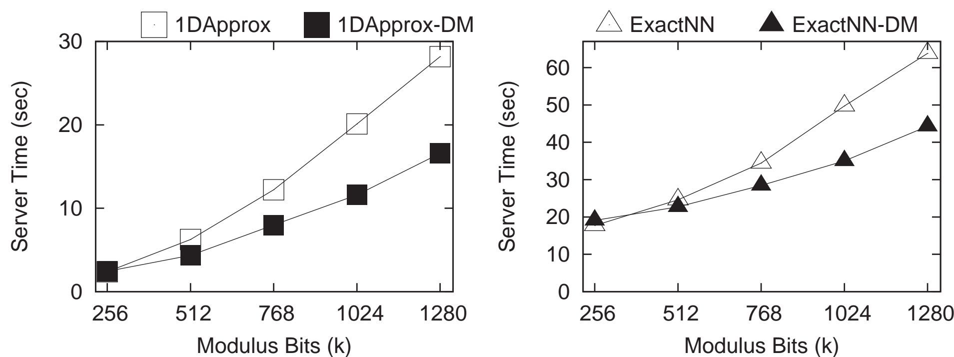


Figure 6.18: DM Optimization, Sequoia set

6.7.3 Execution Time Optimizations

In this experiment we evaluate our optimizer, which employs data mining (DM) to reduce the CPU cost of PIR at the server. We run the Apriori algorithm on the real dataset and retain all frequent itemsets with a support of at least 5%. Figure 6.18 shows the results: for small k values, the gain in execution time is less significant, because multiplications are relatively inexpensive. However, as k increases, the benefit of avoiding redundant multiplications becomes clear: the CPU time is reduced by up to 41% for 1DApprox, and 32% for ExactNN.

The PIR computations are suitable for parallel execution. We implemented a *Message Passing Interface (MPI)* version of the server, and tested it on a Linux cluster with Intel Xeon 2.8 GHz nodes. In Figure 6.19, we show the effect of parallel processing. We vary the number of CPUs from 1 to 8; note that, since each individual CPU is slower than the one used in the previous experiments, the 1-CPU time is slightly larger. The speed-up obtained is almost linear for 1DApprox, where we obtained improvements by a factor of 7.25 for 8 CPUs. For ExactNN, the speed-up is slightly lower, up to 6.1 for 8 CPUs, because the dummy objects correspond to a lot of '0' bits and result in load imbalance among the CPUs. We expect better performance with a more sophisticated load-balancing algorithm. For a typical value of $k = 768$ bits, 1DApprox finishes in 1sec, whereas ExactNN needs 6sec.

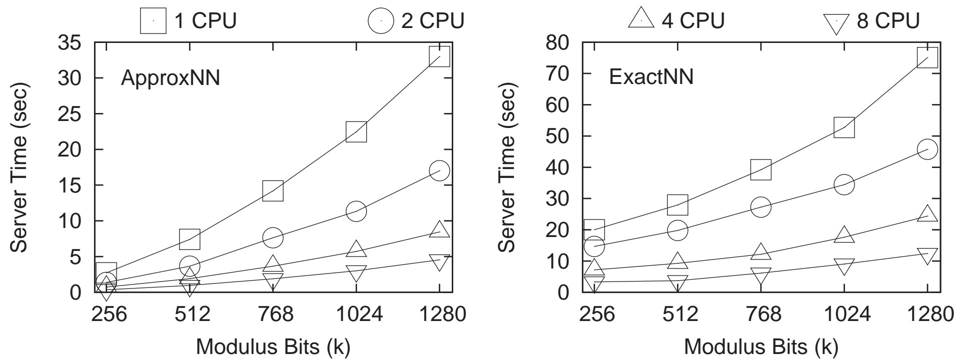


Figure 6.19: Parallel execution, Sequoia set

6.7.4 User CPU Time

The user is typically equipped with a slow PDA; therefore he cannot afford expensive computations. However, our experiments show that the CPU cost for the user is low. In Figure 6.20.a we use the real dataset and vary k. The user needs to generate random k-bit numbers and perform QR/QNR verifications of the k-bit replies; therefore the CPU time is linear to k. For typical $k = 768$, the CPU time does not exceed 0.5sec. In Figure 6.20.b we set $k = 768$ and vary the data size (we use the Gaussian dataset). When the data size increases, so does the number of columns in matrix M. Consequently, the size of the query vector y , as well as the size of the reply vector z, increases. Note that, the CPU time is lower for ExactNN, due to the use of rectangular matrices, which reduce the size of vector z. The resulting CPU time is always lower than 0.4sec.

6.7.5 PIR vs. Anonymizer-based Methods

We compare our methods with *Hilbert Cloak (HC)* [39], which offers privacy guarantees for snapshot queries, and outperforms other cloaking-based location privacy techniques in terms of overhead, i.e. size of cloaking region (CR). Direct comparison is difficult, since the architectures are completely different and there are many unknowns (e.g., how many users subscribe in the anonymizer service, how often they update their location, how often they ask private queries, etc). Instead we study the number of POIs that the user learns from each query (recall from Section 6.3 that the user is charged by

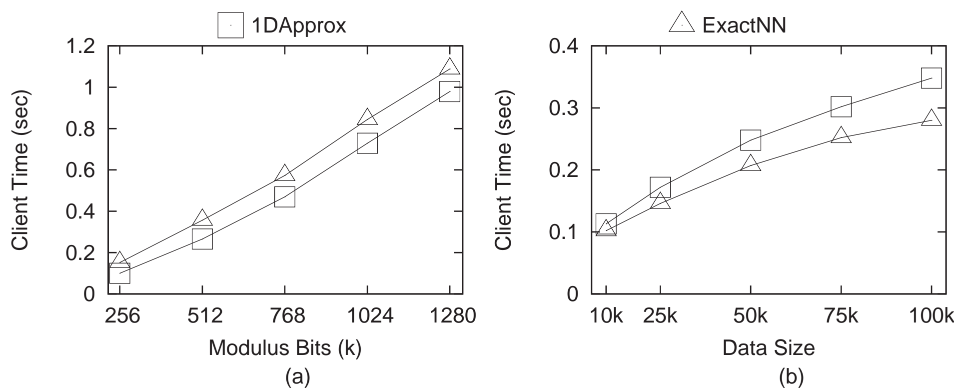


Figure 6.20: User CPU time

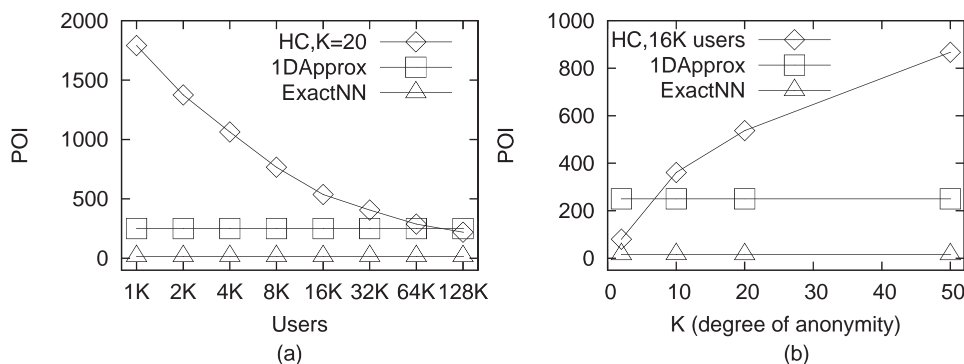


Figure 6.21: PIR vs. K-anonymity, Sequoia set

the number of retrieved POIs).

We consider the Sequoia dataset, and for HC we generate a number of subscribed users between 1K and 128K, at locations that follow the POI data distribution (as discussed in Section 6.7.1). Figure 6.21.a shows the number of disclosed POI for varying number of subscribed users, and a value of anonymity degree K of 20 (i.e., 5% probability of identifying the source). If the number of subscribed users is low, the size of the generated CR is large, and a huge number of POIs are included in the result. Only for a very large number of subscribers does the POI count become comparable with that of 1DApprox, which is roughly 250 for the Sequoia set. The number of disclosed POIs is even lower for ExactNN (i.e., 15 POIs in average), due to the rectangular PIR matrix. This result shows that, in order to maintain a reasonable degree of disclosed POIs (i.e., a compact CR),

cloaking-based methods need to have a large number of subscribed users. This translates into a high cost of location updates (mobile users change location frequently), and also poses privacy concerns, since all users must be trustworthy.

In Figure 6.21.b we fix the number of subscribed users to 16,000 and vary K. HC achieves similar performance to ExactNN for $K < 10$, which means that the identification probability exceeds 10%. The PIR methods are constant because they do not need any subscribed user. Moreover the identification probability is $\frac{1}{|U|} \ll \frac{1}{K}$, where U are all possible users (see Section 6.3).

6.8 Discussion

Our experimental evaluation shows that, although PIR techniques are relatively expensive compared to usual query execution, the overhead is still reasonable. For the real dataset and a typical value of $k = 768$ bits, the communication cost for 1DApprox and ExactNN is roughly 1MB and 2MB, respectively. The corresponding CPU time at the server is 1sec and 6sec, respectively (by employing optimization and/or using multiple CPUs). The CPU time at the user is 1sec at most, and the number of disclosed POIs (hence the resulting financial cost of using the LBS), is low.

Existing SKA approaches have many hidden efficiency issues, such as handling location updates, and managing a large number of user requests. In addition, they have important drawbacks of qualitative nature: First, they lack privacy guarantees for continuous queries (i.e., correlation attack), and fail completely if some of the users are malicious. Second it may not be commercially feasible to gather the required large number of subscribers who will offer continuously their resources for a sporadic benefit. Third, there may be legal reasons which prohibit the anonymizer to gather locations of users.

Compared to previous work, the PIR framework architecture is simpler, more secure (i.e., does not require an anonymizer or collaborating trustworthy users), and is the first one to protect against correlation attacks. Nevertheless, in the absence of a parallel computing infrastructure, the com-

putational cost incurred by PIR may be high in comparison with SKA. For this purpose, we plan to study in future work methods to further reduce the computational overhead of PIR.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

This thesis has focused on a comprehensive framework for private queries in Location Based Services (LBS). We have identified the main objectives and assumptions behind LBS query privacy, and we have systematically built solutions that address the limitations of existing techniques. In summary, our contributions are:

Secure SKA Algorithms. We have first considered the already established setting in most existing work, i.e. Spatial K-anonymity (SKA) within a centralized Anonymizer Server (AS) architecture. In Chapter 3, we have identified the *reciprocity* property, a sufficient condition to guarantee SKA for a snapshot of user locations. Our work was the first to provide privacy guarantees in the above-mentioned setting. We have proposed two SKA algorithms: *Nearest Neighbor Cloak* and *Hilbert Cloak*. *Nearest Neighbor Cloak* uses a randomized variation of NN search, and significantly outperforms existing techniques in terms of K-ASR size, by a factor of up to 4 times. *Hilbert Cloak* builds upon the reciprocity property, and provides provable privacy guarantees, independently of the user location distribution.

Anonymized Query Processing at LBS. The LBS overhead incurred by the processing of anonymized queries is an important concern. In Chap-

ter 3, we have introduced a novel algorithm for finding the NN of a circular region, as opposed to rectangular regions which were considered previously. We have shown that by using circular ASRs, the LBS overhead can be reduced by a significant margin.

SKA Reciprocity with Variable Query Frequency. We have also considered the scenario in which a determined attacker has additional background knowledge on the query frequency of various users. In Chapter 4, we extended the reciprocity property to account for differences in probability of issuing a query at distinct users.

Reciprocal Framework for SKA. In Chapter 4 we have introduced a methodology for building reciprocal ASRs in a systematic manner. We have proposed a family of partitioning methods based on hierarchical spatial indices, with various trade-offs between ASR size and generation time. The reciprocal framework also addresses the variable query frequency setting. Our AR partitioning method outperforms existing solutions by a factor of 2 in terms of ASR size, while the proposed GH method (and its frequency-aware counterpart) incurs an ASR generation time up to an order of magnitude lower than competitor methods.

Decentralized LBS Query Anonymization. Motivated by the limitations of the centralized AS architecture, we have considered in Chapter 5 a distributed architecture for LBS query anonymization. Users self-organize in a P2P overlay network, and cooperate to anonymize queries. We proposed two different P2P protocols, PRIVÉ and MOBIHIDE, which provide a trade-off between privacy guarantees and response time. PRIVÉ implements the *Hilbert Cloak* algorithm in a distributed fashion and offers privacy guarantees. MOBIHIDE relies on a randomized version of *Hilbert Cloak*, which allows a fully-decentralized implementation on top of the Chord [57] DHT. MOBIHIDE guarantees privacy for uniform query distribution, and offers excellent scalability with the number of subscribed users, with a response time of under 5 seconds in the worst case.

PIR-based LBS Privacy. Finally, in Chapter 6, we proposed a completely novel approach to LBS privacy, based on *Private Information Retrieval (PIR)*. This approach has several fundamental advantages over its SKA-based counterparts: specifically, (i) it offers strong privacy guarantees,

that do not depend on the existence of a large number of trusted third-parties, in the form of the AS and its subscribed users. (ii) it eliminates the need for the maintenance of locations for a large population of mobile users and (iii) it thwarts any type of location-based attack, as it does not disclose any location information whatsoever to the LBS server (not even in perturbed form). We have also shown the benefits of PIR techniques in terms of commercial considerations: the number of points of interest disclosed, which is a good estimator of the financial cost incurred by LBS users, is one order of magnitude smaller for PIR than for SKA-based techniques.

7.2 Directions for Future Research

We envision extending this research along the following directions:

- A challenging problem is to ensure anonymity for users issuing continuous spatial queries. Intuitively, preserving anonymity is more difficult in this case: asking the same query from successive locations may disclose the identity of the querying user, who will be included in all ASRs. Although we have addressed this problem with our PIR approach, the issue remains open under the SKA paradigm. Our SKA methods can be extended for processing continuous queries as follows: a snapshot technique (e.g., *NNC*, *HC*) is first employed to determine the set *AS* of users included in the ASR for the initial snapshot of the query; this anonymizing set is "frozen" for the rest of the query lifetime. The MBR of *AS* is then used as ASR at subsequent snapshots[1]. However, as users move in different directions, such an approach may yield large ASRs. Furthermore, if one of the users in *AS* disconnects, it compromises the privacy of the other users. Continuous queries involve several complex issues, and constitute a promising topic for further work.

- Another interesting aspect to enhance the privacy offered by SKA methods is preventing "background knowledge" attacks, when the attacker has additional information about the preferences of certain

[1] Such an approach has been proposed in [20], as discussed in Chapter 2.

users. For instance, if Bob, a rugby fan, asks for the location of the closest rugby club, and the associated ASR contains only female users in addition to Bob, the attacker may infer Bob as query source with higher probability. A solution to this problem would be to group users into partitions according to their areas of interest (e.g., users who query frequently about restaurants, or night clubs, etc). Then, when a query is issued, the corresponding ASR is generated with users from the same interest group as the query source, such that each user in the ASR has an equally likely probability of having asked the query.

- Our P2P anonymization methods currently assume a communication network infrastructure (such as IP connectivity), where users can establish point-to-point connections. An interesting direction for future work is to devise protocols for infrastructure-less networks, in which only mobile devices within communication range can connect to each other (for instance, using Wi-Fi or Bluetooth connections). Furthermore, it would be interesting to develop real-life prototypes of the proposed decentralized anonymization systems, in order to confirm their feasibility in practice.

- Although it offers much stronger privacy guarantees, and works under more relaxed assumptions than SKA, our PIR LBS privacy approach may incur increased computational and communication cost. In the future, we plan to further investigate specific LBS privacy techniques that result in lower cost, as well as general optimizations for PIR protocols that would help reduce the incurred overhead.

Bibliography

[1] *p2psim: The Peer-to-Peer Network Simulator. http://pdos.csail.mit.edu/p2psim.*

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

[3] N. R. Adam and J. C. Wortmann. Security-Control Methods for Statistical Databases: A Comparative Study. *ACM Computing Surveys*, 21(4):515–556, 1989.

[4] C. C. Aggarwal. On k-Anonymity and the Curse of Dimensionality. In *VLDB*, pages 901–909, 2005.

[5] G. Aggarwal, T. Feder, K. Kenthapadi, S. Khuller, R. Panigrahy, D. Thomas, and A. Zhu. Achieving Anonymity via Clustering. In *Proc. of ACM PODS*, pages 153–162, 2006.

[6] G. Aggarwal, T. Feder, K. Kenthapadi, R. Motwani, R. Panigrahy, D. Thomas, and A. Zhu. Approximation Algorithms for k-Anonymity. *Journal of Privacy Technology*, (Paper number: 20051120001), 2005.

[7] G. Aggarwal, N. Mishra, and B. Pinkas. Secure Computation of the k th-Ranked Element. In *Proc. of Int. Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 40–55, 2004.

[8] R. Agrawal, T. Imielinski, and A. N. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proc. of ACM SIGMOD*, pages 207–216, 1993.

[9] R. Agrawal and R. Srikant. Privacy-Preserving Data Mining. In *Proc. of ACM SIGMOD*, pages 439–450, 2000.

[10] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM*, 2002.

[11] S. Banerjee and S. Khuller. A Clustering Scheme for Hierarchical Control in Wireless Networks. In *Proc. of IEEE INFOCOM*, 2001.

[12] R. Bayardo and R. Agrawal. Data Privacy through Optimal k-Anonymization. In *Proc. of ICDE*, pages 217–228, 2005.

[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of ACM SIGMOD*, pages 322–331, 1990.

[14] A. Beimel, Y. Ishai, E. Kushilevitz, and Jean-Fran. Breaking the barrier for information-theoretic private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 261–270, 2002.

[15] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[16] C. Bettini, X. SeanWang, and S. Jajodia. Protecting Privacy Against Location-Based Personal Identification. In *VLDB Workshop on Secure Data Management (SDM)*, 2005.

[17] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[18] R. Cheng, Y. Zhang, E. Bertino, and S. Prabhakar. Preserving user location privacy in mobile data management infrastructures. In *Int. Workshop on Privacy Enhancing Technologies*, pages 393–412, 2006.

[19] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995.

[20] C.-Y. Chow and M. F. Mokbel. Enabling Private Continuous Queries for Revealed User Locations. In *Proc. of SSTD*, pages 258–275, 2007.

[21] C.-Y. Chow, M. F. Mokbel, and X. Liu. A Peer-to-Peer Spatial Cloaking Algorithm for Anonymous Location-based Services. In *ACM International Symposium on Advances in Geographic Information Systems*, 2006.

[22] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying P2P Networks using P-trees. In *Proc. of WebDB*, pages 25–30, 2004.

[23] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2nd edition, 2000.

[24] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *Proc. of ACM PODS*, pages 216–226, 1996.

[25] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. N. Wright. Secure Multiparty Computation of Approximations. In *Int. Colloquium on Automata, Languages and Programming (ICALP)*, 2001.

[26] D. E. Flath. *Introduction to Number Theory.* John Wiley & Sons, 1988.

[27] B. Gedik and L. Liu. Location Privacy in Mobile Systems: A Personalized Anonymization Model. In *Proc. of ICDCS*, pages 620–629, 2005.

[28] G. Ghinita, P. Kalnis, and S. Skiadopoulos. MobiHide: A Mobile Peer-to-Peer System for Anonymous Location-Based Queries. In *Proc. of SSTD*, pages 371–380, 2007.

[29] G. Ghinita, P. Kalnis, and S. Skiadopoulos. PRIVE: Anonymous Location-based Queries in Distributed Mobile Systems. In *Proc. of Int. Conference on World Wide Web (WWW)*, pages 371–380, 2007.

[30] G. Ghinita, P. Karras, P. Kalnis, and N. Mamoulis. Fast Data Anonymization with Low Information Loss. In *Proc. of VLDB*, pages 758–769, 2007.

[31] G. Ghinita, Y. Tao, and P. Kalnis. On the Anonymization of Sparse, High-Dimensional Data. In *Proc. of ICDE*, page to appear, 2008.

[32] O. Goldreich. *The Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

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

[34] B. Hoh and M. Gruteser. Protecting Location Privacy Through Path Confusion. In *Proc. of SecureComm*, pages 194–205, 2005.

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

[36] Z. Huang, W. Du, and B. Chen. Deriving private information from randomized data. In *Proc. of ACM SIGMOD*, 2005.

[37] P. Indyk and D. P. Woodruff. Polylogarithmic Private Approximations and Efficient Matching. In *Proc. of Theory of Cryptography Conference (TCC)*, pages 245–264, 2006.

[38] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: a Balanced Tree Structure for P2P networks. In *Proc. of VLDB*, 2005.

[39] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias. Preventing Location-Based Identity Inference in Anonymous Spatial Queries. *IEEE TKDE*, 19(12):1719–1733, 2007.

[40] P. Kamat, Y. Zhang, W. Trappe, and C. Ozturk. Enhancing Source-Location Privacy in Sensor Network Routing. In *Proc. of ICDCS*, pages 599–608, 2005.

[41] A. Khoshgozaran and C. Shahabi. Blind Evaluation of Nearest Neighbor Queries Using Space Transformation to Preserve Location Privacy. In *Proc. of SSTD*, pages 239–257, 2007.

[42] E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: Single database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[43] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Incognito: Efficient Full-Domain K-Anonymity. In *Proc. of ACM SIGMOD*, pages 49–60, 2005.

[44] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Mondrian Multidimensional k-Anonymity. In *Proc. of ICDE*, 2006.

[45] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Workload-aware Anonymization. In *Proc. of KDD*, pages 277–286, 2006.

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

[47] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-Diversity: Privacy Beyond k-Anonymity. In *Proc. of ICDE*, 2006.

[48] A. Meyerson and R. Williams. On the Complexity of Optimal K-anonymity. In *Proc. of ACM PODS*, pages 223–228, 2004.

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

[50] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE TKDE*, 13(1):124–141, 2001.

[51] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *Proc. of SSTD*, pages 443–459, 2001.

[52] H. Park and K. Shim. Approximate algorithms for K-anonymity. In *Proc. of ACM SIGMOD*, 2007.

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

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

[55] M. Shaneck, Y. Kim, and V. Kum. Privacy Preserving Nearest Neighbor Search. In *Int. Workshop on Privacy Aspects of Data Mining (PADM)*, 2006.

[56] R. Sion and B. Carbunar. On the Computational Practicality of Private Information Retrieval. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2007.

[57] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[58] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *Int. J. of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[59] Y. Tao and D. Papadias. Historical spatio-temporal aggregation. *ACM Trans. Inf. Syst.*, 23(1):61–102, 2005.

[60] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *Proc. of VLDB*, pages 287–298, 2002.

[61] Y. Theodoridis. The R-tree-portal, 2003.

[62] J. Vaidya and C. Clifton. Privacy-Preserving Top-K Queries. In *Proc. of ICDE*, pages 545–546, 2005.

[63] X. Xiao and Y. Tao. Anatomy: Simple and Effective Privacy Preservation. In *Proc. of VLDB*, 2006.

[64] X. Xiao and Y. Tao. Personalized Privacy Preservation. In *Proc. of ACM SIGMOD*, 2006.

[65] X. Xiao and Y. Tao. m-invariance: Towards privacy preserving republication of dynamic datasets. In *Proc. of ACM SIGMOD*, 2007.

[66] J. Xu, W. Wang, J. Pei, X. Wang, B. Shi, and A. Fu. Utility-Based Anonymization Using Local Recoding. In *Proc. of SIGKDD*, pages 20–23, 2006.

[67] Q. Zhang, N. Koudas, D. Srivastava, and T. Yu. Aggregate Query Answering on Anonymized Tables. In *Proc. of ICDE*, 2007.

Appendix A

Analysis of Privacy in *Casper* and *Interval Cloak*

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 Figure 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. For instance, s/he assumes that the query originates from A with probability:

$$P_A = \frac{|A|\cdot(|A|+|B|+|C|+|D|-M_A)}{(|A|+|B|+|C|+|D|)^2-|A|\cdot M_A-|B|\cdot M_B-|C|\cdot M_C-|B|\cdot M_D} \tag{A.1}$$

Within A, each individual user can issue the query with equal probability $P_A/|A|$. For SKA to be preserved, it must hold that $P_A/|A| \leq 1/K_A$. Since the maximum value of K_A is $|A|+|B|+|C|+|D|$, we have $P_A/|A| \leq 1/(|A|+|B|+|C|+|D|)$. 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:

$$M_A = \frac{|B|\cdot M_B+|C|\cdot M_C+|D|\cdot M_D}{|B|+|C|+|D|} \tag{A.2}$$

$$M_B = \frac{|A|\cdot M_A+|C|\cdot M_C+|D|\cdot M_D}{|A|+|C|+|D|} \tag{A.3}$$

$$M_C = \frac{|A|\cdot M_A+|B|\cdot M_B+|D|\cdot M_D}{|A|+|B|+|D|} \tag{A.4}$$

$$M_D = \frac{|A|\cdot M_A+|B|\cdot M_B+|C|\cdot M_C}{|A|+|B|+|C|} \tag{A.5}$$

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 Figure 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 $K_C \geq 4$. Thus, the anonymity of U_C is breached for all, but one, queries involving this ASR.

Having established that the 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.

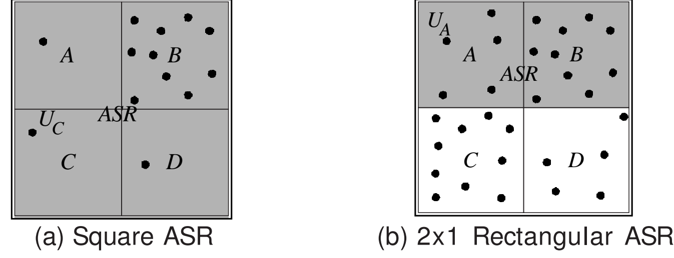


Figure A.1: Examples of Casper ASRs

Assume a rectangular ASR consisting of cells A and B as shown in Figure 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 $K_A \leq |A|$, 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 $[|B|+1, |A|+|B|]$.

The ASR is generated by (i) a query originating from A with K_A, i.e., $|A| \cdot |B|$ events, or (ii) a query originating from B with K_B, i.e., $|B| \cdot |A|$ events. Given that these events have equal probabilities, the attacker assumes that the query originates from A or B with $P_A = P_B = |A| \cdot |B| / (2 \cdot |A| \cdot |B|) = 1/2$. Within A or B, each individual user can issue the query with equal probability $P_A/|A| = 1/(2 \cdot |A|)$ or $P_B/|B| = 1/(2 \cdot |B|)$, respectively. SKA requires that $P_A/|A| \leq 1/K_A$ and $P_B/|B| \leq 1/K_B$. Because the maximum value of K_A and K_B is $|A|+|B|$, it must hold that $1/(2 \cdot |A|) \leq 1/(|A|+|B|)$, and $1/(2 \cdot |B|) \leq 1/(|A|+|B|)$, which are simultaneously satisfied only when $|A| = |B|$. In case that $|A| \neq |B|$, Casper fails to preserve SKA. For instance, in Figure A.1b ($|A| = |D| = 5$, $|B| = |C| = 10$), assume that the ASR is generated due to a query from U_A with K_A in $[6, 15]$. The attacker will pinpoint U_A with probability $P_A/|A| = 1/10$, which compromises anonymity for all values of K_A in the range $[11, 15]$.

In conclusion, 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 of Interval Cloak is similar

to Casper; except that (i) the ASR is always square, and (ii) $M_A = |A|$, $M_B = |B|$, $M_C = |C|$ and $M_D = |D|$, because if a cell does not contain enough users, the method uses directly its parent. Thus, the previous system of inequalities implies that in order to guarantee anonymity, it should hold that $|A| = |B| = |C| = |D|$, meaning that Interval Cloak is also applicable only to uniform datasets.