Efficient management and secure sharing of mobility traces

Dai Hai Ton That

université PARIS-SACLAY

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

NNT : 2016SACLV003

THESE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PREPAREE A
L'UNIVERSITE VERSAILLES SAINT-QUENTIN EN YVELINES

ÉCOLE DOCTORALE N°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat: Informatique

Par

M. Dai Hai TON THAT

Titre de la thèse:

Gestion efficace et partage sécurisé des traces de mobilité

Titre de la thèse en anglais:

Efficient Management and Secure Sharing of Mobility Traces

Thèse présentée et soutenue à Versailles, le 29 janvier 2016 :

Composition du Jury :

Mme. Karine ZEITOUNI, Professeur, UVSQ, Directrice de thèse
M. Iulian SANDU POPA, Maitre de conférences, UVSQ, INRIA, Co-Directeur de thèse
Mme. Agnès VOISARD, Professeur, Freie Universität Berlin, Rapporteur
M. Philippe RIGAUX, Professeur, CNAM, Rapporteur
M. Vincent ORIA, Professeur, New Jersey Institute of Technology, Examinateur
M. David GROSS-AMBLARD, Professeur, Université de Rennes 1, Président du Jury
M. Bruno DEFUDE, Professeur, Telecom SudParis, Examinateur

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Gestion efficace et partage sécurisé des traces de mobilité

Mots clés: données spatio-temporelles, indexation, stockage en mémoire flash, appareils mobiles, protection de la vie privée, collecte participative

Résumé: Aujourd'hui, les progrès dans le développement d'appareils mobiles et des capteurs embarqués ont permis un essor sans précédent de services à l'utilisateur. Dans le même temps, la plupart des appareils mobiles génèrent, enregistrent et de communiquent une grande quantité de données personnelles de manière continue. La gestion sécurisée des données personnelles dans les appareils mobiles reste un défi aujourd'hui, que ce soit vis-à-vis des contraintes inhérentes à ces appareils, ou par rapport à l'accès et au partage sûrs et sécurisés de ces informations. Cette thèse adresse ces défis et se focalise sur les traces de localisation. En particulier, s'appuyant sur un serveur de données relationnel embarqué dans des appareils mobiles sécurisés, cette thèse offre une extension de ce serveur à la gestion des données spatio-temporelles (types et operateurs). Et surtout, elle propose une méthode d'indexation spatio-temporelle (TRIFL) efficace et adaptée au modèle de stockage en mémoire flash. Par ailleurs, afin de protéger les traces de localisation personnelles de l'utilisateur, une architecture distribuée et un protocole de collecte participative préservant les données de localisation ont été proposés dans PAMPAS. Cette architecture se base sur des dispositifs hautement sécurisés pour le calcul distribué des agrégats spatio-temporels sur les données privées collectées.

Title: Efficient Management and Secure Sharing of Mobility Traces

Key words: spatio-temporal data, indexing, flash storage, mobile devices, privacy-preserving, participatory sensing.

Abstract: Nowadays, the advances in the development of mobile devices, as well as embedded sensors have permitted an unprecedented number of services to the user. At the same time, most mobile devices generate, store and communicate a large amount of personal information continuously. While managing personal information on the mobile devices is still a big challenge, whether on account of the inherent constraints of these devices, or towards the safe and secure access and share of these information. This dissertation addresses these challenges while focusing on the location traces. In particular, relying on an embedded relational data server in secure mobile devices, we offer an extension to spatio-temporal data management (types and operations). More importantly, we also propose an efficient indexing technique for spatio-temporal data (TRIFL) designed for flash storage. Besides, in order to protect the user's personal mobility traces, a distributed architecture and a privacy-aware protocol for participatory sensing, have been proposed in PAMPAS. PAMPAS relies on secure hardware solutions for distributed computing of spatio-temporal aggregates on the collected private data.

Summary

Nowadays, the advances in the development of mobile devices, as well as embedded sensors have permitted an unprecedented number of services to the user. At the same time, most mobile devices generate, store and communicate a large amount of personal information continuously. While managing personal information on the mobile devices is still a big challenge, sharing and accessing this information in a safe and secure way is always an open and hot topic. Personal mobile devices may have various form factors such as mobile phones, smart devices, stick computers, secure tokens or etc. It could be used to record, sense, store data of user's context or environment surrounding him/her. The most common contextual information is the user's location. Personal data generated and stored on these devices is valuable for many applications or services to users, but it is sensitive and needs to be protected in order to ensure the individual privacy. In particular, most mobile applications have access to accurate and real-time location information, raising serious privacy concerns for their users.

In this dissertation, we dedicate the two parts to manage the location traces, i.e., the spatio-temporal data on mobile devices. In particular, we offer an extension to spatio-temporal data types and operators for embedded environments. These data types reconcile the features of spatio-temporal data with the embedded requirements by offering an optimal data presentation called Spatio-temporal object (STOB) dedicated to embedded devices. More importantly, in order to optimize query processing, we also propose an efficient indexing technique for spatio-temporal data called TRIFL designed for flash storage. TRIFL stands for TRajectory Index for Flash memory. It exploits unique properties of trajectory insertion and optimizes the data structure for the behavior of flash and the buffer cache. These ideas allow TRIFL to archive much better performance in both Flash and magnetic storage compared to its competitors.

Additionally, we also investigate the protection of user's sensitive information in the remaining part of this thesis by offering a privacy-aware protocol for participatory sensing applications called PAMPAS. PAMPAS relies on secure hardware solutions and proposes a user-centric privacy-aware protocol that fully protects personal data while taking advantage of distributed computing. To that end, we also propose a partitioning algorithm and

an aggregate algorithm in PAMPAS. This combination drastically reduces the overall costs making it possible to run the protocol in near real-time at a large scale of participants, without any personal information leakage.

Résumé

Aujourd'hui, les progrès dans le développement d'appareils mobiles, ainsi que des capteurs embarqués ont permis un développement sans précédent de services à l'utilisateur. Dans le même temps, la plupart des appareils mobiles génèrent, stockent et de communiquent une grande quantité de données personnels de manière continue. Tandis que la gestion des données personnelles dans les appareils mobiles est encore un grand défi, le partage et l'accès à ces informations d'une manière sûre et sécurisée est aussi un sujet largement ouvert. Les appareils mobiles personnels peuvent avoir différents facteurs de forme tels que les téléphones mobiles, les objets intelligents, tokens sécurisés, etc. Ces objets peuvent être utilisés pour enregistrer, capter, et stocker des données du contexte de l'utilisateur ou de l'environnement qui l'entourent. L'information contextuellela plus courante est l'emplacement de l'utilisateur. Les données personnelles générées et stockées dans ces appareils sont précieuses pour de nombreuses applications ou de services à l'utilisateur, mais ellessont aussi sensibles et doivent être protégées afin d'assurer la vie privée des individus. En particulier, la plupart des applications mobiles ont accès à des informations précises et en temps réel sur la localisation de l'utilisateur, ce qui préoccupe graves les utilisateurs.

Dans cette thèse, nous consacrons ses deux parties à la gestiondes traces de localisation, i.e., les données spatio-temporelles des appareils mobiles. En particulier, nous proposons une extension d'une base de données pour les environnements embarqués incluant des types de données spatio-temporelleset des opérateurs spécifiques. Ces types de données réconcilient les caractéristiques des données spatio-temporelles avec les contraintes matérielles des appareils mobiles. Plus important encore, afin d'optimiser le traitement des requêtes, nous proposons également une technique d'indexation efficace pour les données spatio-temporelles appelée TRIFL conçus pour le stockage de type mémoire NAND flash. TRIFL exploite les propriétés uniques des données de trajectoire et optimise la structure de données pour le comportement de la mémoire flash. Ces optimisations permettent à TRIFL d'avoir unebien meilleure performance en mémoire flash et aussi pour les disques magnétiques par rapport à ses concurrents.

Dans la deuxième partie de cette thèse, nous étudions également la protection des in-

formations sensibles de l'utilisateur par un protocole sécurisé (appelées PAMPAS) pour les applications de collectes participatives. PAMPAS repose sur des solutions matérielles sûres et propose une architecture centrée sur l'utilisateur qui protège pleinement les données personnelles tout en profitant descalculs distribués. Pour ce faire, nous proposons un algorithme de partitionnement de l'espace d'observation pour la collecte. Cette combinaison permet de réduire drastiquement les coûts globaux de calcul permettant d'exécuter le protocole en temps quasi réel à grande échelle des participants, sans aucune fuite d'informations personnelles.

Acknowledgement

Though only my name appears on the cover page, many people have significantly and insightfully contributed to this dissertation. Without the direct or indirect support from those people, I would have never completed my thesis. Therefore this acknowledgement is to express my thankfulness to all of them.

My first deepest gratitude is to my advisor, Prof. Karine ZEITOUNI. It has been an honor for me to be one of her PhD students. I really appreciate all her great contribution in time, ideas and funding during my PhD thesis. I can't image how my thesis goes without her excellent and conscientious guidance. It would be one of the most exciting experiences in my life to work more than 3 years in such friendly, open and excellent research team headed by Prof. Karine where I have been always trusted and freed to explore on my own. I will also never forget how she encouraged me when I was down, her patient and insight support significantly helped me overcome many crisis situations and finish this dissertation. If I have a wish to come true in the future, then I wish to be a nice, friendly, successful and talented scientist and professor like her.

To my co-advisor, associate Prof. Iulian SANDU POPA, I would like to express my heartfelt gratitude for everything he has taught me in both research and life. It was an amazing fortune for me to have a change to work tightly with one of the most young and talented research scientist like him. At first, I would like to acknowledge him for putting my first foot step on the research-life by opening my mind in every aspects of research career. I would also like to sincerely thank to him for his solid and insightful contribution in both ideas and writing to this dissertation. I greatly thank to him for always being on my side and spending many weekends in the office in order to help me to overcome the hardest parts of my thesis. Also, I am very grateful to him for his changing my own definition about the "supervisor" due to his kindness, friendliness and conscientiousness he has been to me during the time we have worked together. Actually, it's hard to memorize all of meaningful advice he has been given to me in both professional and personal life. If there is only one sentence for me to describe my gratitude to him then it should be "to be one of his first PhD students would be always my great honor and proudness".

It is a big shortcoming if I don't mention about the help of the members in SMIS (INRIA)

team. At first, I am grateful to Prof. Benjamin NGUYEN for his recommendation of me to my advisors. His kindness support has changing my life since it put me in front of a big opportunity to work and study in PRISM laboratory. Second, I would like to thank to Prof. Philippe PUCHERAL, the leader of SMIS team, for his fruitful comments and his indirect contribution to this thesis through KISS project. Third, special thanks are due to Prof. Luc BOUGANIM for fruitful discussions related to Flash storage that significantly improve the first part of this dissertation. I am also thankful to Mr. Alexei TROUSSOV for very interesting discussions and for his precious advice in my implementation during my internship. Many thanks to Dr. Nicolas ANCIAUX, Dr. Tristan ALLARD, Dr. Quoc-Cuong TO, Saliha LALLALI, Paul TRAN-VAN, Athanasia KATSOURAKI and other members in SMIS team for their fruitful talks and cheerful moments.

I gratefully acknowledge Prof. Cristian BORCEA and Prof. Vincent ORIA for the short-duration collaboration we have had and for their meaningful contribution to the second part of this thesis. I would also specially thank to Mr. Paul ZEITOUNI for his very friendly and conscientious instruction in the implementation part of my demonstration.

To all my friends in PRISM laboratory: Dr. Naila Bouchemal, Dr. Ahmed Kharrat, Dr. Fatiha Amanzougarene, Dr. Qingfeng Fan, Wenjun Yuan (Clement), Hanane Ouksili, Kim Tam Huynh, Dr. Isma Sadoun, Kenza Menouer, Maria Koutraki, Dr. Mohamed-Amine Adouane, Dr. Zakaria Bendifallah, Ali Masri, Raef Mousheimish, Ticiana Linhares and other PhD students, I am very grateful for all their cheerfulness and friendship. Thank you for all their friendly help, their understanding and their encouragement in many moments of crisis. It was my great pleasure to know all of them and be a friend with these very nice, young and talented people. I would also like to thank for the excellent and friendly atmosphere they have contributed to our research and living environment that made me be free to explore on my own. Wherever and whatever I will be in the future, we are always friends and I will keep with me these funny and exciting memories we have had in PRISM laboratory.

I would sincerely thank to the other members in ADAM team: Prof. Mokrane Bouzghoub, Associate Prof. Béatrice Finance, Associate Prof. Yehia Taher, Associate Prof. Laurent Yeh, Associate Prof. Zoubida Kedad, Associate Prof. Stéphane Lopez, Associate Prof. Nicoleta Preda and secretaries: Ms. Chantal Ducoin and Ms. Isabelle Moudenner for their kindness help.

I also gratefully acknowledge my landlords (i.e., Ms. Phan Thi Sen and Mr. Duong Quan Vi) and all my Vietnamese student friends for all their kindness care and love.

Last but not least, I would like to express my heartfelt gratitude to my parents, my brothers and my girlfriend for all their endless love and encouragement. I really appreciate all their moral support in all my pursuits.

Dai Hai TON THAT

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

In the last decade, we have witnessed an explosion in the adoption of mobile devices (e.g., mobile phones, smartphones, tables, smartwatches, etc.) by users. The trend is clear and indicates that mobile devices are steadily replacing desktops and even laptop computers at least for personal usage and activities. For instance, in the Unites States the percentage of users owning a smartphone increased from 35% in 2011 to 64% in 2014, while for tablets the market grew from 8% to 42% during the same period [88]. Meanwhile, the desktop and laptop computers have maintained a steady rate, at best, of around 80%. Similar trends have been also observed in other developed or developing countries [86], [87], [89]. The success of mobile devices is predicted to last, sustained by the continuous advances in hardware technology and the myriad of new applications currently being developed. Moreover, the arrival of the Internet of Things [4] can only be beneficial to the generalization of mobile devices and to increasing their diversity.

An important consequence of this massive adoption of mobile devices is the exponential increase in the personal data production and consumption [120]. There are two main reasons to explain this phenomenon. First, mobile devices are mostly connected to the internet allowing users to conduct their typical online activities (e.g., mailing, web browsing, social networking, e-commerce). Second, mobile devices such as smartphones are equipped with sensors (e.g., GPS, camera, microphone, temperature, accelerometers) or can be easily connected to other portable smart sensors (e.g., smartwatch or other sensors enabling the quantified self [65], [102], [119]). All these sensors permit automatic or manual acquisition of quality data at high rates. Also, an important observation is that the GPS sensor is a first class citizen among the sensors embedded in mobile devices. Indeed, all daily personal data (e.g., pictures, the environmental noise or pollution, tweet, posts in a social network) can be geolocalized and timestamped with GPS sensor.

This profusion of personal data represents an unprecedented potential for applications and business, which led the World Economic Forum to affirm that "personal data is the new oil" [121]. However, with great opportunities come also important challenges. We highlight here two of them. The first challenge is related to the management (i.e., storage, indexing, querying and sharing) of large amounts of (spatio-temporal) personal data. The data management has to consider the specificities of spatio-temporal data and the advances in the hardware technology (e.g., new storage devices based on Negative-AND Flash memory (NAND flash memory)). For instance, in many cases, the data is produced and consumed in stream (e.g., traffic or pollution data) requiring to be processed, indexed and queried in real-time. Also, the data can be sent in the Cloud (i.e., to a remote service) or kept at the user-side, leading to centralized, decentralized or hybrid architectures. If stored locally, the data management has to take into account the hardware constraints of the specific user's device (e.g., limited RAM and CPU power, battery consumption, specific storage such as NAND flash).

The second important challenge in this context is privacy. Until now, the enthusiasm for new opportunities brought by this massive creation of personal data, has thwarted privacy concerns. Nevertheless, the risk of a backlash is growing as new devices and new services bring us closer to the dystopias described in the science fiction literature. This risk is well documented and the nature of the solution appears to be consensual, i.e., it is necessary to increase the control of the individuals over their personal data [21], [120], [73]. In this context, the World Economic Forum indicated the need for a data platform that allows individuals "to manage the collection, usage and sharing of data in different contexts and for different types and sensitivities of data" [121].

At the same time, the protection of users' privacy should not hinder the development of new applications that are beneficial for both users and the society. We believe that the advent of secure smart objects (e.g., secure smart card) holds the promise of offering tangible security guarantees to users' mobile devices. Such tamper-resistant devices are flourishing today, e.g., Mobile Security Card (produced by Giesecke & Devrient), Personal Portable Security Device (produced by Gemalto and Lexar), Multimedia SIM card or Secure Portable Token [10], [111], and can be used to securely manage personal data [11].

1.2 Motivation

Integrating mobile technology and positioning devices has led to the production of large amounts of spatio-temporal data every day. A wide range of applications such as traffic management and location-based services (LBS), rely on these data. The generated spatio-temporal data streams are meaningful for many applications such as traffic analysis, customized insurance (pay-as-you-drive), reconstructing the circumstances of an accident in

road safety, mobility and population exposure analysis, etc. Spatio-temporal streams may be saved for legal reasons (e.g., to provide an evidence for the insurance), or for individual use (e.g., experience sharing, individual gas consumption monitoring or eco-driving).

In this thesis, we focus on two major aspects related to the management of spatio-temporal data streams. The first aspect is motivated by the generalization of storage devices based on NAND flash. Due to several important features, such as high performance, low power consumption and shock resistance, NAND flash has become the most popular stable storage medium for mobile devices and sensor. Also, Solid State Drives (SSDs) built on flash are replacing magnetic disk drives (HDDs) in laptop and desktop computers, and even in high-end enterprise servers. However, despite its many advantages, NAND flash has also peculiar characteristics compared with traditional magnetic disks (e.g., asymmetric read-write performance, erase-before-write mechanism). Moreover, the characteristics and performance dramatically change when comparing low-end flash devices (e.g., SD cards) with high-end flash devices (e.g., SSDs). Therefore, to fully benefit from the high performance of flash storage devices, we need to reconsider the spatio-temporal indexing techniques that have been designed for magnetic hard-disks in the context of flash storage.

The second aspect is motivated by the fact that geolocation data is inherently sensitive since it may reveal private information about the personal activities and behaviors [26], [47]. There exist many approaches in the area of location privacy [40]. For example, some approaches aim at preserving the current location, essentially in the scenario of LBS, while the other relies on the whole trajectory anonymization [71]. Different solutions have been proposed in the literature which use or not a trusted anonymizer, cryptography or location perturbation/generalization. However, to the best of our knowledge, there is no solution adapted to the context of secure mobile devices.

Secure personal hardware has paved a new way by using a user-centric architecture [8], [111], [113] as an alternative approach to the typical server-centric architecture. A decentralized, user-centric architecture exhibits better properties than a centralized architecture in terms of privacy (i.e., no need to trust a central server) and security (i.e., due to decentralization). Nevertheless, dealing with data management in the decentralized architecture also raises several challenges. A first important challenge comes from the severe hardware constraints of the secure portable devices. Such devices offer state-of-the-art security guarantees (i.e., they are tamper-resistant). But, this security comes at a price, e.g., a low power CPU and a tiny RAM (at most 100KB). Therefore, we need to devise adapted data management techniques that take into account both the hardware constraints of secure mobile devices and the specificity of spatio-temporal data. Such techniques will allow users to manage their personal data locally in their devices. At the same time, users may want to share their data or participate in global computations that can benefit the entire community (e.g., behavioral or epidemiological studies, participatory sensing). Hence, we

also need to devise secure global computation protocols to aggregate data from a large number of users and thus, to reestablish all the global treatments that can be done in a centralized architecture. Again, the challenge here comes from the limited resources and availability of users' secure devices, and also from the fact that these global computations have to be performed while preserving users' privacy.

Finally, this thesis work is also motivated and strongly related to the ANR KISS (Keep your personal Information Safe and Secure) project [3] (detailed in Chapter 6). The idea promoted in KISS is to embed, in trusted devices, software components capable of acquiring, storing and managing securely various forms of personal data (e.g., salary forms, invoices, banking statements, geolocation data) depending on the applications. These software components form a Personal Data Server which can remain under the holder's control. The scientific challenges include: embedded data management issues tackling regular, streaming and spatio-temporal data (e.g., geolocation data), crypto-protected distributed protocols to implement private communications and secure global computations.

1.3 Contributions

There are three main contributions in this thesis, listed in the order they are introduced in the thesis, as following:

- *We propose TRIFL, an efficient and generic TRajectory Index for FLash.* TRIFL is designed around the key requirements of trajectory indexing and flash storage. TRIFL is generic in the sense that it is efficient for both simple flash storage devices such as the SD cards and more powerful devices such as the solid state drives. In addition, TRIFL is supplied with an online self-tuning algorithm that allows adapting the index structure to the workload and the technical specifications of the flash storage device to maximize the index performance. Moreover, TRIFL achieves good performance with relatively low memory requirements, which makes the index appropriate for many application scenarios.

- *We propose PAMPAS, a Privacy-Aware Mobile PArticipatory Sensing system based on secure mobile probes.* Mobile participatory sensing could be used in many applications such as vehicular traffic monitoring, pollution tracking, or even health surveying. However, its success depends on finding a solution for querying large numbers of users which protects user location privacy and works in real-time. PAMPAS is a privacy-aware mobile distributed system for efficient data aggregation in mobile participatory sensing. In PAMPAS, mobile devices enhanced with secure hardware, called secure probes, perform distributed query processing, while preventing users from accessing other users' data. Secure probes exchange data in encrypted

form with help from a supporting server infrastructure. PAMPAS uses two efficient, parallel, and privacy-aware protocols for location-based aggregation and adaptive spatial partitioning of secure probes. Our experimental results and security analysis demonstrate that these protocols are able to collect, aggregate and share statistics or derived data in real-time, without any privacy leakage.

- *We implemented a prototype in the context of the secure Personal Data Server.* This prototype has two major components that were embedded in a secure portable token. The first component is an extension of an embedded relational database engine (i.e., PlugDB [82]) to deal with spatio-temporal data. This extension consists in new data types and new operators allowing to locally manage spatio-temporal data in conjunction with the classical data types and operations in the relational model. The second component is a secure global aggregation protocol based on PAMPAS and applied to the context of traffic monitoring using secure devices. We call this component PPTM (Privacy-aware Participatory Traffic Monitoring). The implementation of PPTM can be executed in many secure tokens in parallel and shows the feasibility of doing privacy-preserving participatory sensing using secure mobile devices.

1.4 Organization of the thesis

The thesis is organized in nine chapters that are grouped in three parts. Figure 1.1 outlines the organization of this thesis. The document starts with the current chapter, in which we introduce the general context, the motivation and the contributions of the thesis.

The first part of this document is composed by two Chapters (i.e., Chapters 2 and 3) related to indexing spatio-temporal data. Specifically, in Chapter 2, we study the state-of-the-art on the relevant indexing methods in both magnetic disks and flash storage. Then, we analyze the limitations of existing methods when considered in the combined context of trajectory data and flash storage by pointing out the specificities of spatio-temporal data and flash storage. Following the challenges described in Chapter 2, we introduce in Chapter 3 an indexing method (i.e., TRIFL) dedicated to spatio-temporal trajectory indexing in Flash storage. We present in detail the structure of TRIFL and its cost model, before evaluating it in comparison with relevant state-of-the-art indexing methods.

The seconds part covers Chapters 4 and 5 and deals with global computations that preserve the user location privacy. In Chapter 4, we discuss the related work relevant to user location privacy and in particular, the systems related to mobile participatory sensing applications and their respective limitations. Then, we present PAMPAS, a privacy-aware mobile distributed system for efficient and secure data aggregation in mobile participatory sensing, in Chapter 5.

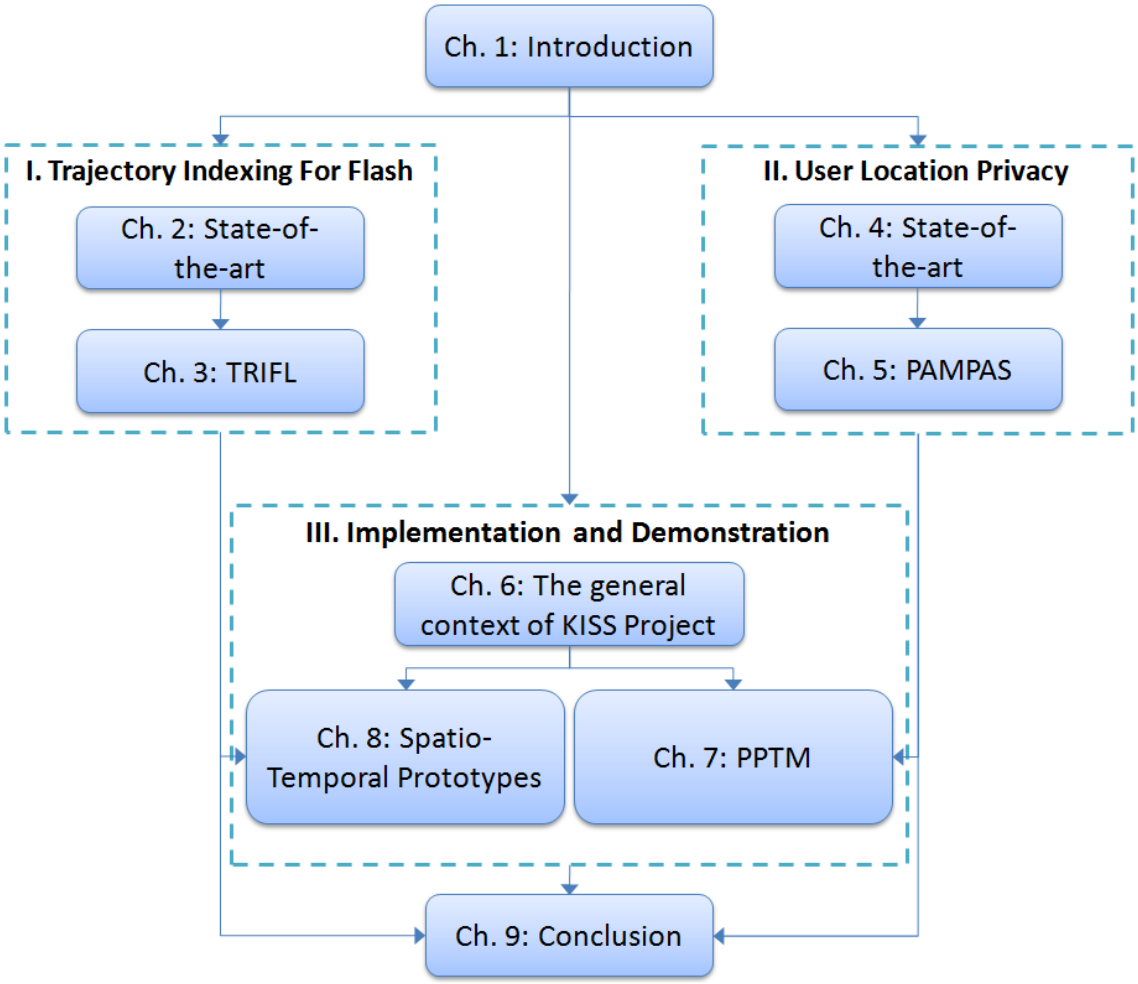

Figure 1.1: Organization of the thesis

The third part comes as a result of the first two parts and covers Chapters 6, 7 and 8. In this last part, we present the implementation of a prototype using use-cases inspired by real applications. Chapter 6 gives an overview of the ANR KISS project that inspired and guided the work in this thesis. We also point out our contributions in the implementation of the Personal Data Server promoted in KISS. We demonstrated the usefulness of PAMPAS in the real scenario of online traffic monitoring in Chapter 7 (i.e., PPTM: Privacy-Aware Participatory Traffic Monitoring Using Mobile Secure Probes). Lastly, in Chapter 8, we present the implementation and evaluation of a spatial-temporal database extension for secure hardware.

Finally, we conclude the thesis in Chapter 9 by summarizing the contributions and indicate some interesting directions of future work resulting from this thesis.

Part I

Trajectory Indexing for Flash storage

Chapter 2

State of the Art

In this Chapter, we present the context and motivation of the proposed TRajectory Index for Flash storage (TRIFL), as well as the relevant state-of-the-art. Specifically, this Chapter has the following structure. The issue of trajectory indexing for TRIFL is discussed in Section 2.1. We define the problem statement and list the contributions in Section 2.2. Finally, Section 2.3 is dedicated to the related works.

It is worth noticing that the contributions of this thesis, and especially TRIFL, can be included in the area of spatio-temporal databases, or more specifically, in the field of moving objects databases [45]. The works in this field started more than two decades ago, motivated by the need to store, model, and query spatio-temporal data, i.e., data related to both space and time (e.g., such es the trajectories of moving objects and their interactions). Beside, spatio-temporal databases can be seen as the follow-up of spatial databases [42] (e.g., allowing to query spatial data such as points, lines and regions) and temporal databases [80], [105] (e.g., allowing to query time-varying data and to record efficiently the changes in data over time). Today, spatio-temporal databases are a mature, well established branch in the databases area and a detailed presentation of the works in this field would be rather counter productive. Instead, we focus and present in details only the related work relevant to the contributions of this thesis.

2.1 Trajectory indexing in Flash memory

The convergence of mobile computing, wireless communications, and sensors has led to an exponential production and consumption of data in the last years. In many cases, the data are geo-localized and time-stamped [22], [52]. This results in a massive flow of spatio-temporal data to be inserted and queried. Many applications have been flourishing as a consequence of this massive data production in different domains such as location-based services [45], [20], participatory sensing [81], or traffic management [38], [57], [109].

The database research community has actively supported the development of such applications that are built on spatio-temporal data by providing technical solutions ranging from the data integration and indexing level to the data visualization and mining level. Probably one of the most active topics in this area is the spatio-temporal data indexing. Mokbel and colleagues [69] listed around 35 spatio-temporal access methods proposed up to 2003 and the list has not stopped growing ever since [72]. There are several reasons to this profusion of indexing techniques. First, spatio-temporal indexing covers several types of data, e.g., historical trajectory data that is mainly static [16], [19], [55], [95], [106], trajectory flows that continuously add new data [24], [84], [96], or moving object tracking that only considers the current and estimated near-future positions of the moving objects [68], [117], [107], [83]. Second, there are many types of queries related to mobility [68], e.g., range queries, nearest neighbors (NN), reverse NN, spatio-temporal join. Third, the type of movement itself leads to different indexing approaches, as the moving object (MO) can be free (e.g., pedestrians, animals) [16], [19], [24], [55], [106] or constrained by a transportation network (e.g., vehicles, trains) [25], [95].

Since a few years a new fundamental parameter has made its entry on the database scene: the NAND flash storage. Due to several important features, such as high performance, low power consumption and shock resistance, NAND flash has practically become the most popular stable storage medium for embedded devices and sensor networks. Moreover, Solid State Drives (SSDs) built on flash are replacing magnetic disk drives (HDDs) in personal computers. And with the capacity increasing every year and prices continuing to drop, SSDs appear as a viable alternative to HDDs even for large enterprise servers. Meanwhile, SSDs are already integrated between the main memory and secondary storage in the architecture of enterprise servers to extend and improve the database caching.

The advent of flash storage has had an important impact on databases [29]. The reason is simple: NAND flash has peculiar characteristics compared with traditional magnetic disks (e.g., fast random read, asymmetric read-write performance, erase-before-write mechanism, etc.) as described in the next section. Therefore, to fully benefit from the high performance of flash storage devices, many of the data storage and indexing techniques that have been designed for magnetic hard-disks in the last forty years have to be reconsidered in the context of flash storage.

2.2 Problem statement and contributions

The adaptation of databases to flash storage started more than a decade ago and many solutions have been proposed ever since [6], [61], [93], [122], [123], [64]. While a significant number of these works focus on data indexing, to our knowledge, none of the existing works addresses the problem of indexing trajectory data in flash.

In this Chapter we tackle the fundamental problem of indexing trajectory flows in flash storage. A flow of trajectories can be seen as a dataset that expands continuously through data insertions and is queried simultaneously. The proposed method can optimize both range queries and NN queries, which are the base types of queries over trajectory data. The proposed index works for both free and constrained trajectory flows.

Why is this problem challenging? The first key challenge is related to trajectory data. Some of the existing indexing methods for magnetic disks can be used in conjunction with the existing flash-aware B^+-tree indexes to index trajectory flows (see Section 2.3). Whereas this type of approach represents a straightforward solution, the resulting indexes will have suboptimal performance. The reason is that the specific features of trajectory data are not considered by the existing tree indexing techniques for flash, which focus on typical data indexing (i.e., primary or secondary key indexing). In particular, trajectory datasets are characterized by massive insertions, while deletions and modifications of the existing data are rare. Indeed, new (parts of) trajectories can be added continually by logging the individual updates coming from tracking a group of moving objects or as periodical large batches of insertions. Therefore, the data insertions can be timely (i.e., trajectories transmitted in real time) or deferred (i.e., recorded trajectories transmitted at a latter point in time). In both cases, the index should be able to handle the insertions efficiently, while processing the users' queries. Also, the index performance should not degrade in time as a consequence of this massive updating.

The second key challenge is related to flash storage. NAND flash is an electronic memory and unlike magnetic disks has no mechanical movement (i.e., no seek and rotational delays). Therefore, the latency of random reads (RRs) is similar to the latency of sequential reads (SRs) in flash and RRs can be up to two orders of magnitude faster in flash than in HDDs [75]. However, flash memory badly supports fine-grain data writes. The memory is organized in blocks containing a number of pages (e.g., typical values are 64 or 128 pages). The write granularity is a page, but a page cannot be rewritten without erasing first the block that contains it (i.e., the erase-before-write mechanism). In addition, the pages must be written in sequential order in an erased block. Also, the number of times a block can be erased before it wears out is limited (e.g., 10^4 erase cycles). As a result, random writes (RWs) can be up to two or even three orders of magnitude more costly than sequential writes (SWs) at least in basic flash devices such as the SD cards [99].

In addition, flash memory is used in several types of storage devices that cover a large spectrum of physical specifications. For instance, only for the SD cards devices the ratio RW/SW can vary from one to three orders of magnitude, while the ratio RW/RR can extend even further [99]. Furthermore, recent SSD devices (e.g., SSD OCZ [75]) manage to overcome the major drawback of flash memory, i.e., very costly random writes, through a combination of internal parallelism, large DRAM cache and complex software imple-

menting the Flash Translation Layer (FTL). For such devices, the RW latency is similar to the SW latency. Therefore, the typical optimizations that are based only on reducing the number of RWs are less useful in this case. Instead, the index should take advantage of the highly efficient large granularity I/Os in flash, i.e., transform the RWs into large granularity SWs. Thus, the second major challenge is to have a generic index structure, i.e., the index should be capable of adapting to the wide range of physical specifications of flash devices.

Finally, the third important challenge concerns the index memory requirements. The typical approach [76], [101], [61] for indexing insert intensive data is to buffer the updates in memory and to commit them in batch in order to amortize the cost of individual updates. Such an approach, which was employed in the context of magnetic disks [76], [101] becomes even more relevant in the context of flash storage [61], [6] since flash memory, unlike magnetic disks, badly supports in-place updates. On the other hand, typical trajectory indexing techniques partition the space before indexing the data in each partition [16], [19], [24], [95]. As a consequence, numerous low level indexes have to be maintained in parallel. However, buffering the new index entries for each index component can require an important amount of cache memory that may not always be available. Memory is still an expensive resource as even in the case of enterprise servers many applications share the same cache buffer pool. Besides, memory limitation is obvious in the case of embedded devices such as smart phones, smart objects and sensors. With the advent of Internet of Things, the usage of such embedded devices is generalizing since they offer an increased environmental sustainability by avoiding energy consuming data transfers between smart objects and remote servers especially when large amounts of often seldom used data are concerned [64], [116], [118]. For such devices, it is thus important to lower the index memory footprint without significant performance loss.

To respond to these three key challenges, we propose TRIFL, a generic TRajectory Index for FLash storage. TRIFL is generic in two senses. First, the global structure of TRIFL is based on well-established state-of-the-art index structures for trajectory data on HDD storage. These methods adaptively partition the space and then index the temporal dimension of the trajectories in each partition with a B^+-tree. TRIFL reuses this idea of spatial partitioning and temporal indexing, but replaces the B^+-tree indexing with alternative indexing methods that are adapted to flash devices. However, different from the existing methods that consider only the query load in the space partitioning computation,TRIFL computes an optimal spatial partitioning of the data by considering both the queries and the insertions in the workload.

Second, TRIFL is supplied with a cost model and an online algorithm that allow TRIFL to be self-tunable with respect to the characteristics of the flash storage device and of the workload. Moreover, TRIFL combines a physical and a logical partitioning of the data,

which makes the index structure adapted to both basic flash devices (e.g., SD cards) and powerful flash-based devices (e.g., SSDs). Interestingly enough, as a byproduct, TRIFL surpasses the existing indexes even with HDD storage. Finally, TRIFL provides high performance with relatively low memory requirements, which makes it appropriate for a wide range of scenarios.

Specifically, the contributions of our TRIFL are as following:

- We propose a low-level index structure combining a novel type of Append-Only B^+-trees and Time Interval Indexes to index the temporal dimension of trajectories in flash. These structures drastically reduce the number of RWs in flash, while opening the way for large granularity write I/Os.
- We alternate between a logical and a physical partitioning of the trajectory data to further benefit from the fast large-granularity I/Os in flash by transferring almost small-granularity I/Os into large granularity I/Os.
- We provide the index structure with a cost model and an online algorithm to make it self-tunable with respect to the performance specifications of the flash storage and the index workload.
- We experimentally evaluated TRIFL in comparison with four state-of-the-art indexes on two flash devices having different characteristics and on a HDD device. The results show that TRIFL provides throughput values that are on average one order of magnitude higher than its competitors depending on the insert/query ratio and the type of insertions in the workload.

2.3 Related work

In this section[1] we present existing work related to trajectory indexing and to flash indexing. First, we focus on the trajectory indexing problem and present the existing approaches with an emphasis on the space partitioning methods. Then, we discuss the typical approaches for tree indexing in flash.

2.3.1 Trajectory indexing on magnetic disks

Spatio-temporal indexing is a hot research topic since the mid-90. The initial works [16], [19], [25], [55], [95], [106] focused on indexing large, but mainly static, historical trajectory datasets. With the increased possibility of tracking MOs and obtaining their trajectories in real-time (or offline in large batches), subsequent methods [24], [84], [96] considered the

[1] All figures in this section are taken from the cited references with the permissions of the authors.

problem of indexing trajectory flows. This is a particularly hard problem to solve, since the index structure has to be capable of integrating continuously a large number of incoming trajectory data, while accelerating the users' queries over the indexed data. Moreover, as the continuous data insertion may degrade the query performance, the index should also be able of self-tuning to keep up the near-optimal performance of the structure. Many different types of queries, e.g., range queries, nearest neighbors (NN), reverse NN, spatio-temporal join or aggregate queries, are optimized by these access methods. However, a great majority of the existing works [16], [19], [24], [25], [55], [84], [95], [96], [106] focus on range and NN queries, since these are the most frequent queries over trajectory data.

There are two main approaches to indexing trajectory data. A first group of techniques are based on data partitioning [25], [55], [84], [106]. These methods use multidimensional R-tree-like or R-tree-based structures to index both the spatial and the temporal dimensions of trajectory data. A second group of methods are based on space partitioning [16], [19], [24], [95], [96]. Such methods partition first the space and then index only the time dimension of the trajectories in each partition with a B^+-tree. Arguably, it was shown [16], [19], [24], [95], [96] that space partitioning methods are globally more efficient than data partitioning methods. Space partitioning indexes offer superior query performance than data partitioning indexes since trajectory datasets exhibit large amounts of overlapping in both the spatial and the temporal dimensions. And the difference in performance becomes even more important in the case of indexing trajectory flows. The main reason is that B^+-trees are more update efficient than R-trees and also work better in a concurrent environment that combines queries and updates. We describe below in more detail the methods using space partitioning, since the global structure of TRIFL is based on these methods.

We present some well-known spatio-temporal indexing methods based on space partitioning.

SETI, a Scalable and Efficient Trajectory Index. SETI [19] was proposed to partition the 2D space with a uniform grid or grid-like structure. In particular, at the first level the space is partitioned by non-overlapping cells in order to classify the trajectory segment into these cells. This means each segment mainly located belongs to this cell. Then, in the second level the trajectory segments (also called trajectory units) belonging to one cell are clustered and indexed on the time dimension with a B^+-tree or a 1D R-tree. However, this method suffers from two main drawbacks. First, using a uniform space partitioning does not consider the fact that trajectory data have skewed distributions both in space and in time, which leads to high variability of the execution time of queries. Second, they do not provide analytical models to compute the optimal number of cells of the grid with respect to the data size and the query workload, which leads to suboptimal performance of the index.

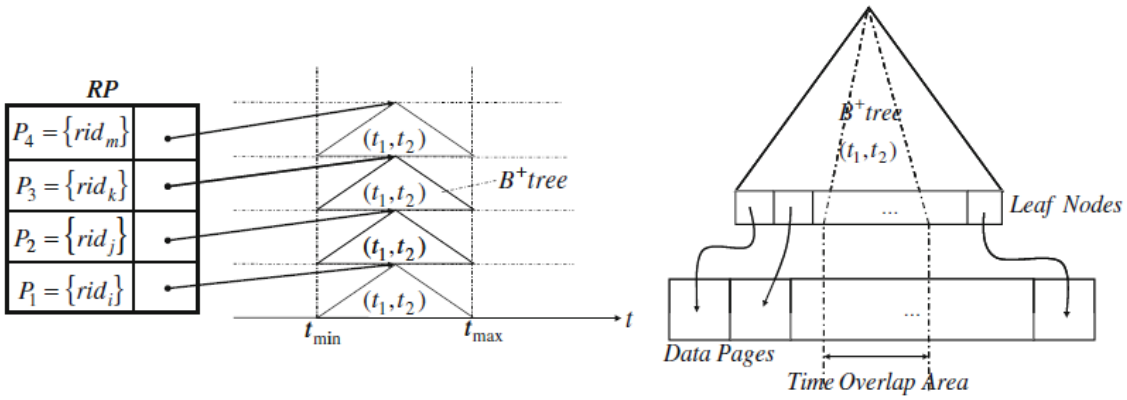

Figure 2.1: Examples of PARINET index structure

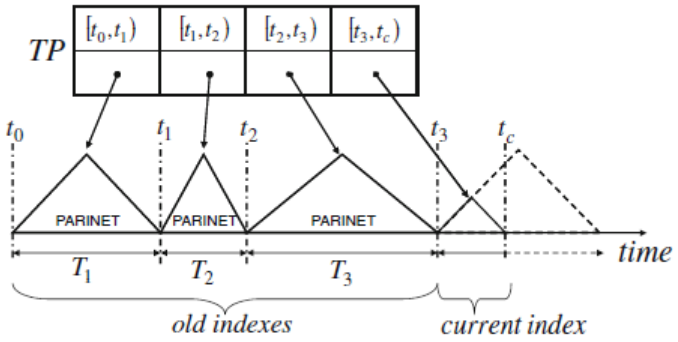

Figure 2.2: Examples of TPARINET index structure

TrajStore, PARINET and T-PARINET. More recent methods [95], [24] fix these previous shortcomings by employing an adaptive grid partitioning of the space, where the cell size depends on the local density of the data. In addition, these methods are supplied with a cost model that allows to compute the optimal cell size for a given query workload. Another important observation is that in many cases the movement of the objects is constrained by a transportation network. In such cases, it is more efficient to represent and index the trajectories with reference to a network space instead of a 2D space [25]. Besides, the queries are generally formulated relatively to the network space for constrained MOs. The problem of indexing in-network trajectory flows is considered in [96], where the T-PARINET index is proposed. Figure 2.1 and 2.2 show the index structure of PARINET and T-PARINET respectively. As in the above mentioned methods, T-PARINET proceeds by partitioning the graph network into network regions. The size of the regions depends on the spatio-temporal density of the data and on the query workload. Then, the trajectory units in each region are indexed on time with a B^+-tree.

To the best of our knowledge, all the existing methods for trajectory indexing are devised for HDD storage and do not consider the peculiar characteristics of flash memory. Therefore, they will offer suboptimal performance if used directly on flash storage. Moreover,

the existing methods that provide tuning algorithms to compute an optimal partitioning of the space, consider only the queries in their cost models. However, the insert performance of the index is essential in the case of trajectory flows (given the high insertion rate), especially when a basic flash device is used for storage (given the poor RW performance). TRIFL uses the global idea of space partitioning and time indexing of the existing methods, but modifies the low level storage and time indexing with structures that are adapted to flash storage. In addition, the spatial partitioning of TRIFL considers both the query cost and the insert cost of the index workload.

2.3.2 Tree indexing in NAND Flash

The first works to adapt the classical databases techniques to flash storage initiated more than a decade ago. Since then, many solutions regarding the data storage [58], [64], the data indexing [6], [61], [93], [122], [123] and the data caching have been devised for flash storage. The existing works cover different application domains ranging from sensor networks [64], [118] to embedded database systems [6], [123], [116] or large databases stored on SSDs [61], [93]. A majority of the existing works focus on indexing and within this majority many papers propose alternatives to the B^+-tree index that are fitted to flash storage [6], [61], [93], [122], [123]. We focus on these works in the remainder of this section, since a flash-aware B^+-tree may be used for indexing the temporal dimension of trajectory data.

The first works that propose flash-aware B^+-tree indexes [6], [58], [64], [122], [123] focus on the poor random write performance of flash devices. Therefore, the key idea in these approaches is to buffer the updates in log structures that are written sequentially and to leverage the fast (random) read performance of flash memory to compensate the loss of optimality of the lookups.

BFTL. The architecture of BFTL [122] is shown in Figure 2.3. In particular, BFTL [122] stores the index entries of a node in multiple physical pages and maintains a logical B-tree with the help of an in-memory table, which maps each node to the physical pages. This means B-Tree index services requested by higher level are handled by BFTL with the help of its small reservation buffer and node translate table (in-memory) before generate real requests to Flash Translation Layer (FTL) by using an efficient commit policy in order to reduce the random write as well as maximal the large granularity IOs. However, this approach leads to a significant degradation of the search performance, since many flash accesses are required to search a single tree node. Also, BFTL requires additional RAM memory to store the mapping table. For trajectory indexing, the memory consumption of a B-tree plays an important role, as the index partitioning entails a large number of trees belonging to the same global structure.

Lazy Adaptive-tree (LA-tree). Similar to BFTL, LA-Tree [6] shares the same idea of

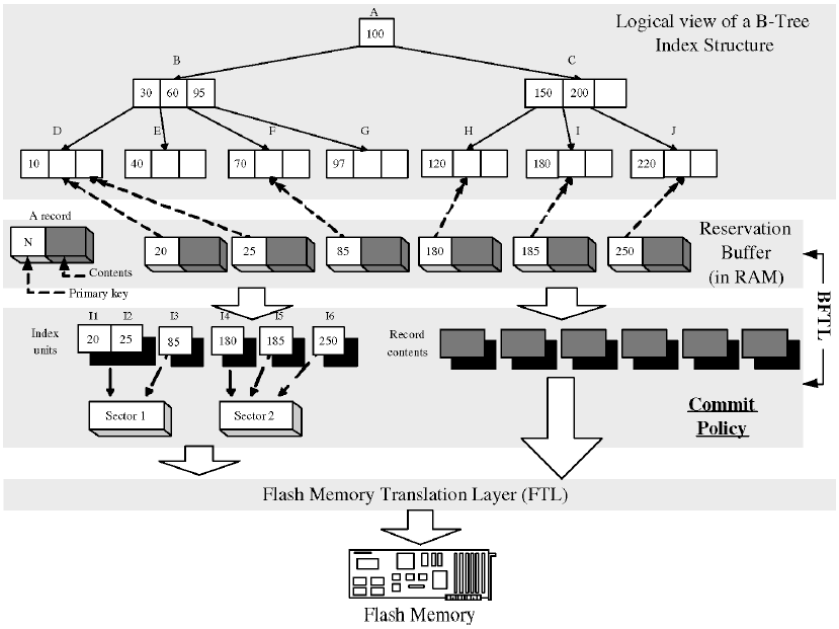

Figure 2.3: The architecture of BFTL

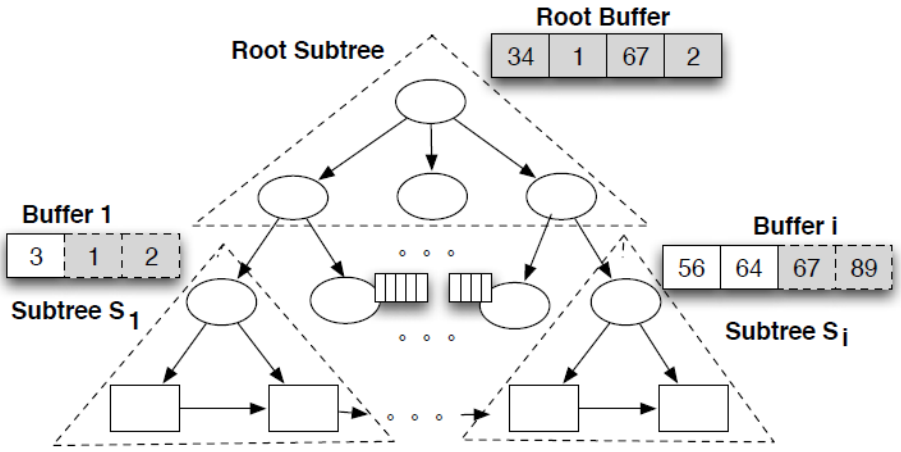

Figure 2.4: The LA-Tree

delay the changes a longer a possible. An example of LA-Tree is shown in Figure 2.4. LA-tree uses flash-resident buffers, which are attached to various levels of the B^+-tree to log the updates. The updates are integrated in the B^+-tree only when the leaf level buffers become full. The benefit of lazy updates is that they reduce the total number of flash accesses since they are done in batch. Also, LA-tree uses an online algorithm that adapts the size of the buffers to the type of the index workload. Larger buffers are used for update-intensive workloads, whereas smaller buffer are more efficient in case of lookup-intensive workloads.

MicroHash and MicroGridFile. The work in [64] proposes two index structures,

MicroHash and MicroGridFile, for efficiently indexing temporal environmental and geographical data recorded by flash-based sensors. The main idea in these structures is to completely eliminate the need for RW operations in flash. Trajectory data indexing is possible with these methods, but the supported queries are only temporal or spatial predicates and not spatio-temporal range or NN queries.

FAST. FAST [97], [98] proposes a generic framework that can be applied on top of data partitioning tree indexes to boost their performance on flash storage without changing the underlying index structures. FAST uses an in-memory buffer for the updates, which are intelligently flushed to reduce the write cost in flash. However, the focus in FAST is on the flushing policies and the efficient implementation of a crash recovery mechanism, which is mainly complementary to this work.

MILo-DB. MILo-DB [10] is an embedded relational DBMS engine for mass-storage portable tokens. MILo-DB is designed to cope with the conflicting constraints of NAND flash and scarce RAM of portable devices. To tackle such constraints, MILo-DB proposes a log-only based storage and indexing scheme to avoid any RWs. To obtain scalability, MILo-DB reorganizes periodically the indexes, as it is done in TRIFL. However, since MILo-DB targets selection and join indexes for RDBMSs, the proposed techniques are less relevant in the context of trajectory flows.

FD-Tree. A common drawback of the above listed methods is that the proposed optimizations are limited to only reducing the number of random write operations. The very high latency of RWs remains a valid hypothesis for basic flash devices such as the SD cards [99]. However, current SSDs [75] can handle RWs as fast as SWs by leveraging the internal parallelism of such devices. In addition, large granularity reads and writes also benefit from this internal parallelism and enable a throughput that is up to one order of magnitude larger than with the page granularity I/Os. Consequently, more recent proposals of flash-aware B^+-trees were designed to exploit such features. Among the most generic tree indexes for flash, there is FD-tree [61] whose design includes both the reduction of RWs and the support for large granularity I/Os. FD-tree uses the idea of a log-structured merge-tree (LSM-tree) [76] and adapts it to SSDs. FD-tree consists of a small B^+-tree on top of several levels of sorted runs of increasing size (see Figure 2.5). All the RWs are limited to the head B^+-tree which is normally cached in the main memory. Whenever the head tree becomes full, it is merged with the sorted runs below. Hence, the RWs are transformed into SWs. Besides, the merge operations are performed using large granularity I/Os. Also, compared with the LSM-tree, FD-Tree improves the search performance of the range queries since the sorted runs allow retrieving several key entries in one I/O. Other B^+-trees for SSDs focus only on exploiting the internal parallelism of such devices. For instance, Parallel IO B-tree [93] combines the large granularity I/Os with the parallel execution of a set of I/O requests to improve the performance of the B^+-tree on SSDs.

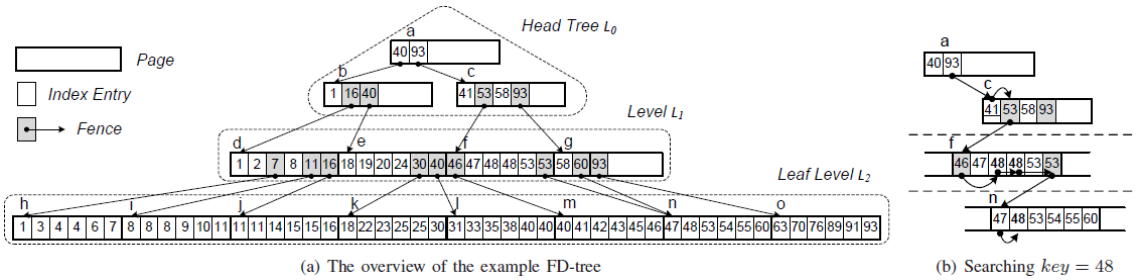

Figure 2.5: An example of FD-Tree

However, such optimizations are mostly orthogonal with the index structure and can be applied to other database structures.

LSM-tree and bLSM-tree. We also mention the works in [76], [101] that propose a log-structured merge-tree. Although, these index structures are designed for magnetic disks, they share some common points with the flash-aware structures (e.g., FD-tree [61]) and TRIFL. The LSM-tree [76] is an efficient index structure for workloads having high insertion rates. The LSM-tree consists in one in-memory B-tree component to buffer the updates and one on-disk B^+-tree component that indexes the disk resident data. Periodically, the two components are merged to integrate the in-memory data and free the memory. The benefit of such an approach is twofold. First the updates are integrated in batch, which amortizes the write cost per update. Second, the merge operation uses sequential I/Os, which reduces the disk arm movements and thus, highly increases the throughput. If the indexed dataset becomes too large, the index disk component can be divided into several disk components of exponentially increasing size to reduce the write amplification of merges. bLSM [101] fixes several limitations of the LSM-tree. Among the improvements, the main contribution is an advanced merge scheduler that bounds the index write latency without impacting its throughput. TRIFL also employs a merge mechanism. However, the logged updates are stored directly on flash to reduce the memory consumption. In addition, the merge process is optimized by the use of large granularity I/Os and does not consider arm movements, which is proper to magnetic disks and not to flash devices. All this leads to different storage and index structures in TRIFL.

The existing flash-aware B^+-tree structures represent a straightforward solution for indexing trajectory flows. Nevertheless, such solutions may be suboptimal as they do not consider the important particularities of trajectory indexing. First, a trajectory index has to integrate efficiently a continuous and massive flow of new data without letting the lookup performance degrade much. Second, trajectory indexes perform a spatial partitioning beforehand [16], [19], [24], [96], which implies that a high number of temporal indexes has to be efficiently managed at the same time. Therefore, the main memory footprint of each local index has to be very small to satisfy this constraint. But this requirement is

conflicting with the existing flash indexes that have to buffer the updates in memory in order to avoid costly RWs or to execute large granularity I/Os.

Chapter 3

TRIFL: A Generic Trajectory Index for Flash Storage

In this Chapter, we present the global framework of TRIFL. We first describe the trajectory data model and the query model supported by TRIFL in Section 3.1. Then we introduce the global index structure of TRIFL in Section 3.2. Note that the framework introduced in this section is generic and independent of the type of used storage. The cost models of TRIFL as well as an online self-turning algorithm are also addressed in Section 3.3. Section 3.4 evaluates the performance of TRIFL and details the comparison of TRIFL with its competitors. Finally, We summary our designed principles in Section 3.5 before concluding this Chapter in Section 3.6.

3.1 Scope of TRIFL

3.1.1 Data model

TRIFL is designed to continuously index flows of MOs' trajectories up to the current time and to efficiently process the spatio-temporal queries over the recorded history. A *trajectory* is defined as a sequence of trajectory units. A unit is generated between two consecutive location updates belonging to the same MO and the time intervals of the units are not necessarily of equal size. An update contains the identifier of the trajectory, the location and the associated time instant: $(trid, loc, t)$. Each unit is recorded as a tuple in the database: $(trid, [loc_1, loc_2], [t_1, t_2])$. Within each unit, it is assumed that the MO moves at constant speed, i.e., a linear interpolation is considered over each interval, which is common to the existing trajectory indexes.

TRIFL indexes trajectories generated by both free MOs and constrained MOs. Only the representation of the location is different between the two types of movement. In the case

of free MOs [16], [24], the location is represented by the 2D coordinates, i.e., $loc = (x, y)$. As for constrained movement [25], [96], the location is represented with reference to a network space (usually represented as a directed graph), i.e., $loc = (rid, pos)$, where *rid* is the road (or edge) identifier and *pos* is the relative position on the road as measured from start point of a road. Both the free and the constrained movement representations have been extensively studied in previous works [16], [24], [25], [96].

3.1.2 Query model

TRIFL can index two types of queries, i.e., range queries and point nearest neighbor queries, which are the classical queries over trajectory data. Typical examples of range queries are: *"which are the trajectories that crossed that region between 10am and 11am today?"* or *"find the number of MOs that traversed a given road section at a certain time (interval)"*. While a nearest neighbor query example is: *"which were the three closest MOs to the indicated location at 12am today?"*

A query is composed of a spatial part and a temporal part: $Q = (Q_S, Q_t, k)$. The parameter k applies only to NN queries and indicates the number of requested nearest neighbors. Q_t is a time interval, which can degenerate to a point (a time instant). Q_S depends on the query type and on the movement type. In the case of a range query, Q_S indicates a spatial interval. For free MOs, Q_S is a 2D region (e.g., a rectangle). For constrained MOs, Q_S is a set of road sections, i.e., $Q_S = (rs_1, rs_2, ...rs_n)$, where $rs_i = \left(rid_i, \left[pos^i_{11}, pos^i_{11}\right]\right)$ and the road sections are disjoint. In the case of a nearest neighbor query, Q_S indicates a location either in the 2D space or a network space.

3.1.3 The global index structure of TRIFL

The global structure of TRIFL is inspired from the existing index structures that employ a space partitioning (conform to Section 2.3.1 in Chapter 2). The reasons are two-fold. First, space partitioning index structures are more efficient for both the search [16], [95], [19] and the update operations [96]. Trajectory data exhibit large amounts of overlapping in both the spatial and the temporal dimensions, which greatly affects the search effectiveness of multi-dimensional indexes [16], [95] that are typically based on the classical R-tree or its variants. In addition, the update operations are more costly in the R-tree [56] that in the B-tree [59] especially in a concurrent environment that combines queries and updates [69]. The reasons for this claim are manifold: (i) in addition to the split propagation when a node overflows as in B-trees, R-trees have to recursively update the ancestor keys in case a leaf's minimum bounding rectangle (MBR) changes; (ii) to account for the strategies aiming to minimize the overlap of the MBRs of the nodes, many leaves may be visited which increases the cost; (iii) the split of an overflowing node in an R-tree is more complex

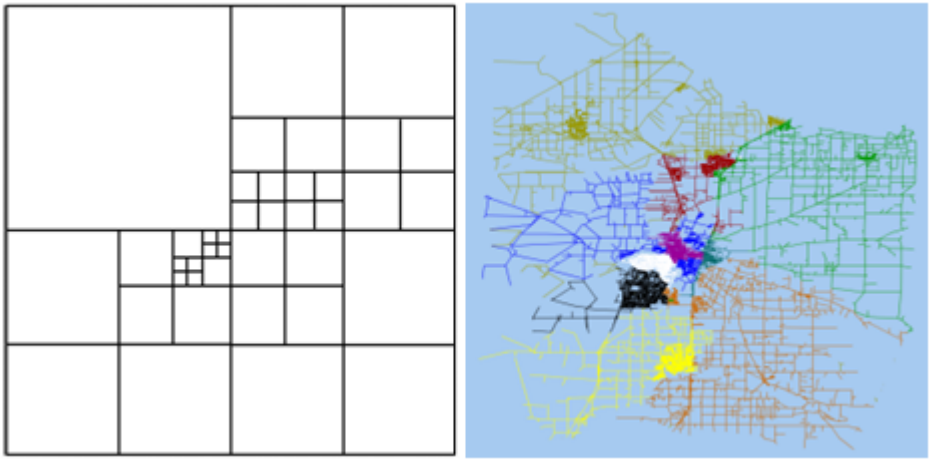

Figure 3.1: Examples of 2D and network space partitioning in TRIFL

than in a B-tree due to the lack of linear ordering among the keys. Hence, since the cost of updates and the potential impact it has over different nodes in R-trees are higher than in B-trees, the execution of a mixed workload with updates and queries will likely perform better with a B-tree based index than with those using R-trees, whatever the technique for concurrency is. All this makes space partitioning methods an obvious choice for indexing trajectory flows. Second, these methods are more easily tunable, i.e., are provided with algorithms that compute the space partitioning offering an optimal lookup performance with respect to a data load and a query workload.

Given a flow of trajectories, a TRIFL index is created to index the incoming trajectory data belonging to a time interval $[t_i, t_{i+1}]$, where t_i corresponds to the index creation time and t_{i+1} corresponds to the closing time of the index and depends on the data flow rate. At t_{i+1} a new TRIFL structure is created to store and index the flow and this process continues indefinitely. We provide in Section 3.3.2.2 a cost model to compute the time interval $|T_i| = t_{i+1} - t_i$ of a TRIFL index.

TRIFL partitions first the indexed space into n partitions by using either an adaptive grid partitioning for free trajectories or a network partitioning for constrained trajectories (see figure 3.1). TRIFL uses a specific cost model (see Sections 3.3.1 and 3.3.2) to compute the space partitioning. In short, the space partitioning algorithm of TRIFL takes into account the spatial data distribution to produce spatial regions that are balanced with respect to the amount of data in each region. Therefore, the spatial partitioning can periodically adapt to the temporal changes of the MO distribution in space. Then, for each grid cell or network region, TRIFL builds a dedicated index structure that stores the trajectory units in that partition and indexes the units on the time dimension (see figure 3.2).This structure is composed of a novel Append Only B^+-tree and of a Time Interval Index, which are presented in Section 3.2.

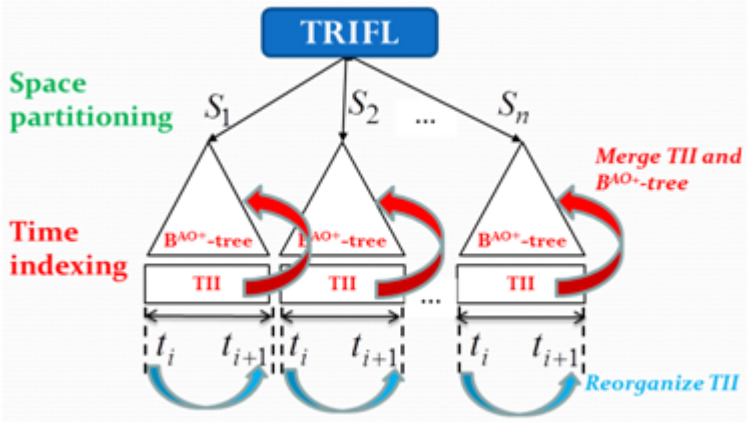

Figure 3.2: The global structure of TRIFL

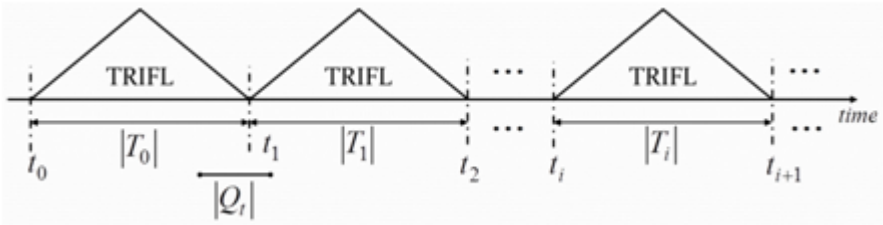

Figure 3.3: Temporal partitioning in TRIFL

3.1.3.1 Temporal partitioning of TRIFL

A trajectory data flow is generally bounded in space (i.e., the data is generated in a predefined observed 2D region or road network), but not necessarily bounded in time (i.e., the data harvesting can extend over long periods of time). As indicated above, a TRIFL index has a bounded time span $[t_i, t_{i+1}]$ (see figure 3.3). In this section we discuss the benefit of the temporal partitioning for trajectory flow indexing. The detailed algorithm used in TRIFL to compute the length of the time interval covered by a TRIFL index is given in Section 3.3.2.2.

Only a few methods [96], [63] for indexing trajectories propose a temporal index partitioning. BB^x-tree [63] proposes very short time span indexes, i.e., covering only 1.5 times the maximum time interval of a trajectory unit. Also, the lifespans of two consecutive indexes overlap significantly. Consequently, the queries intersect several indexes and their performance is significantly degraded. T-PARINET [96] also proposes a temporal index partitioning. However, the cost model that computes the partitioning is designed for B^+-trees on magnetic disks and is not relevant to our context. As for the "general" purpose index methods that propose multi-component indexes [76], [101] or multi-level indexes [61], their approach is to create indexes of exponentially increasing size to obtain a trade-off between the query and the insert cost. This type of approach is not appropriate in our

case since it deteriorates the insert performance without improving the query performance. Therefore, we propose in Section 3.3.2.2 a temporal partitioning algorithm that is adapted to the specific structure of TRIFL.

3.1.3.2 Query processing

TRIFL supports all the typical index operations, i.e., search, insertion, deletion and update. Given a range query $Q = (Q_s, Q_t)$, the data lookup in TRIFL is performed in three steps. First, TRIFL identifies the spatial partitions that that intersect Q_s, i.e., the spatial filtering step. Then, it uses the time indexes in the selected partitions to get all the trajectory units that intersect Q_t, i.e., the temporal filtering step. Finally, a refinement step is required to eliminate the false positives, which consists intesting the exact spatio-temporal intersection between the candidate units and Q. The update operations are discussed in Section 3.2.2.

NN queries are evaluated in two steps, which can be repeated several times. In the first step, the indicated location is expanded to a spatial window, i.e., a square or a network region depending on the trajectory type, centered in the search point. Therefore, the initial NN query is transformed into a rage query. In the second step, the obtained range query is evaluated. If the number of returned MOs is superior or equal to k, no additional search is necessary. The list of MOs' identifiers is sorted based on the distance to the query position and the top k MOs are returned as the final result. Otherwise, the spatial window is expanded and a new range query is generated. Note that the spatial part of the new query is computed as the difference between the previous and the current expansion to avoid a repetitive search over the same spatial regions. The returned MOs are added to the current set of MO identifiers. The algorithm stops when the set contains at least k MO identifiers. Assuming that the spatio-temporal distribution of the indexed data is known, the expansion size can be set inversely proportional to the spatial and temporal density of the data observed at the query location and the query time interval. Higher data densities indicate a higher probability to find the NN closer to the query point location and vice-versa.

3.2 Detailed structure of TRIFL

In this section we present the detailed structure of TRIFL. We present first the index design principles. Based on these principles, we introduce the data structure in TRIFL. Finally, we describe the mechanisms that allow TRIFL to support large granularity I/Os.

3.2.1 Design principles

TRIFL is designed around five principles. The first two principles are related to the problems of trajectory indexing, the following two principles consider the characteristics of flash devices, whereas the last principle covers both aspects.

P1. Efficiently process the data insertions. Indexing trajectory flows incurs a high number of data insertions. Typically, the index workload comprises both lookup queries and trajectory insertions with a much higher rate of the latter. Therefore, it is necessary to process efficiently the insertions even at the cost of moderate query performance degradation.

P2. Low buffer cache memory requirement. The data partitioning (typically employed by trajectory indexes) incurs a high number of local indexes that need to be handled simultaneously. Therefore, for increased efficiency (i.e., to decrease the number of RRs and RWs for data insertions) a local index should have low cache memory requirements, as the buffer cache is shared among many indexes. In other words, the typical strategy consisting in buffering in memory a large number of updates before integrating them in batch in the index (e.g., FD-tree [61], LSM-tree [76]) may be inefficient with many indexes and limited cache size.

P3. Favor the sequential writes over the random writes. The index structure should avoid as much as possible the RWsand favor instead the SWs, since RWs are poorly supported by flash memory, which is reflected in the performance of flash devices (especially with basic devices such as the SD cards).

P4. Favor large granularity I/Os over page granularity I/Os. Multiple pages I/Os are much more efficient than single page I/Os especially for devices that have internal parallelism such as the recent SSDs. Therefore, the index structure should exploit this feature for increased performance.

P5. Endow the index structure with self-tuning to keep up the near-optimal performance. The continuous insertion of new data may lead to a degradation of the lookup performance. The index should be able to automatically detect and correct the loss in performance over time.

Note that some of these principles are conflicting. For example, a large buffer cache may be required to reduce the number of RWs and satisfy $P3$, $P4$ and $P5$, which is conflicting with the requirement of $P2$. Also, with a small buffer cache, many in-place updates may be required to process the insertions as requested by $P1$, which is conflicting with $P3$. The existing methods (see Section 2.3.1 in Chapter 2) and their straightforward adaptation (see Section 2.3.2 in Chapter 2) do not comply with all the above listed principles. On the one hand, the methods that are optimized for HDD storage do not consider the $P3$ and $P4$ principles that are mostly related to flash storage. For example, the insertions in the B^+-tree are executed as in-place updates at a page granularity. On the other hand, the

methods that are optimized for flash (cf. with $P3$ and $P4$) and also optimize the insertions (cf. with $P1$) require buffering many insertions in memory before committing in block to the disk (e.g., FD-tree [61]). While this is not an issue in case of a single index structure, it may become problematic when a large number of such indexes are simultaneously active and the cache memory is limited (cf. with $P2$).

To reconcile all these principles, TRIFL uses appropriate low-level index structures that permit to maintain the index as a log. The benefit is manifold. Logs are known to be insert efficient structures (as required by $P1$) since all the modifications are directly appended to the log storage area. Hence, RWs are avoided and replaced with SWs (as required by $P3$). The second important consequence is that large granularity SWs can be employed to update the index structure (as required by $P4$) at the cost of a small buffer cache. The index structures in TRIFL are also designed to have a small and adaptive memory footprint (as required by $P2$), i.e., it requires only a few cache pages per index partition. Lastly, TRIFL structures leverage the fast large granularity I/Os in flash to periodically reorganize the index structure and compensate the loss in the search performance of a log-like structure (as required by $P5$).

3.2.2 Storage and indexing in TRIFL

TRIFL partitions the trajectory data on the space dimension and then clusters and indexes the trajectory units in each partition on the time dimension. TRIFL considers the peculiarity of the time dimension, i.e., the time takes monotonically increasing values, and the fact that the data can be obtained either in real-time, i.e., from tracking a set of MOs, or offline, i.e., as a batch of previously recorded trajectories. We refer to the first type of insertions as *timely insertions* and to the second type as *deferred insertions*. To handle the timely insertions, TRIFL uses an *Append-Only* B^+-tree (B^{AO+}-tree), which is a novel B^+-tree-like structure optimized for the insertions of monotonically increasing keys in flash. To handle the deferred insertions, TRIFL uses *a Time Interval Index* (TII) structure. Both structures are designed in accordance with the principles $P1$-$P4$. Note that we do not make any assumption with regard to the ratio between the timely and deferred insertions, as TRIFL can handle timely or deferred insertions mixed in any proportion.

3.2.2.1 $B^{AO+} - tree$

The B^{AO+}-tree (see figure 3.4) is a modified B^+-tree that supports only the search and the insert operations. In addition, a new key can be inserted in a B^{AO+}-tree only if it is larger than all the already inserted keys. Thus, all the insertions take place in the right-most leaf of the tree. The B^{AO+}-tree indexes the t_2 time instant of the trajectory units (cf. Section 3.1). The data units are stored in the leaf nodes of the index.

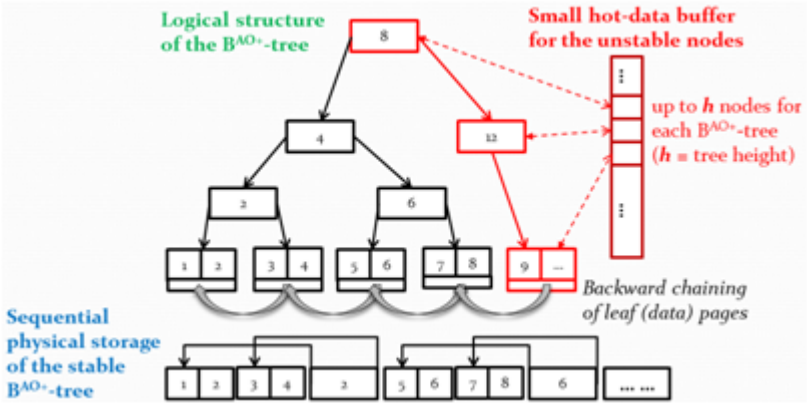

Figure 3.4: Basic example of B^{AO+}-tree

The tree consist of two parts: *a stable part* containing most of the tree nodes, which cannot be affected by new insertions (in black in figure 3.4), and an *unstable part* containing the right-most path from the root to the right-most leaf, which is continuously modified by the incoming data (in red in figure 3.4). Note that the unstable part of the tree contains at most h nodes, where h is the height of the tree (typically equal to two or three). A tree node is written to the stable part only when it is full. To accelerate the temporal range search, a new leaf node is back-chained with the foregoing leaf before being written to the stable part. Hence, the stable part requires only SWs (cf. our design principle $P3$), while all the RWs are limited to only a few nodes (cf. $P2$ and $P3$), which are typically amortized by the buffer cache.

The append-only feature of the B^{AO+}-tree makes it much more insert efficient than the classical B^{+}-tree, but also more search efficient since the fill factor is 100% for the B^{AO+}-tree. Given a time interval $Q_t = [t_1, t_2]$ of a spatio-temporal query, we search first the leaf containing t_2 and then use the backward chaining to retrieve all the leaves containing units that overlap in time with Q_t. In addition, if the tree is physically clustered (i.e., the tree is stored in a contiguous area on flash), the scanning of the leaf nodes can be done with large granularity I/Os to increase the search efficiency.

3.2.2.2 *Time Interval Index (TII)*

All the deferred insertions in a partition are stored within a second structure (see figure 3.5). A TII is a set of disjoint and adjacent time intervals that cover the current time span of the index. The number of time intervals is established at the index creation based on the cost model presented in Section 3.3. Each time interval is associated with a linked list of data pages, which contain trajectory units that overlap with the time interval. A trajectory unit that overlaps with more than one interval is duplicated and inserted in all the overlapping intervals. Similar to the B^{AO+}-tree, the TII has a stable part containing

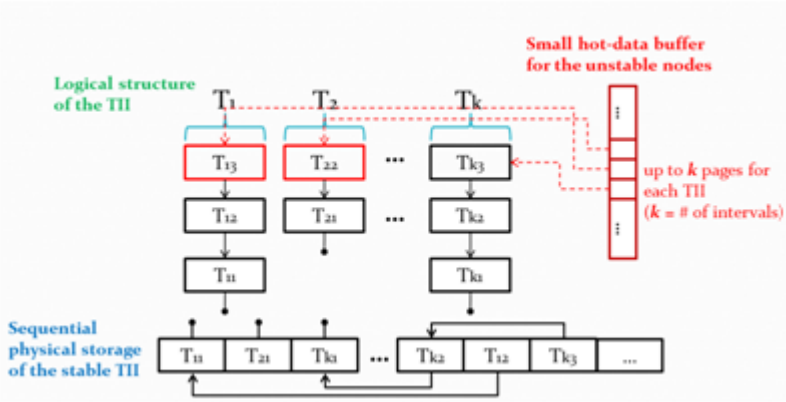

Figure 3.5: Basic example of a TII

full data pages and an unstable part containing one TII descriptor page and at most one non-full data page for each interval. All the writes in the stable part are sequential. The descriptor of the TII stores the physical addresses of the last stable data page in each interval as well as the temporal distribution of the units (required by the TII reorganization as described below). The new data pages are inserted as the head of the interval lists. When a new unstable page is filled, it becomes stable and it is appended at the end of the stable zone. Given Q_t, all the page chains of the intervals that overlap with Q_t have to be searched.

Note that new time intervals are appended to the TII with the extension of the lifespan of the index. Note also that the stable and the unstable parts of the indexes use two different storage areas in flash to increase the write efficiency. Specifically, in our implementation, the stable and unstable parts of the indexes are stored in distinct files. We use two files to store the stable parts of the index (see Section 3.2.4) and one file for the unstable parts of the index. To physically separate the storage areas of the two types of files, we partition the flash storage device and store each type of file on a different partition.

TRIFL also supports delete and update operations. TRIFL processes deleted trajectory units as deferred insertions, i.e., it inserts the deleted data in the TII. The updates are executed as the deletion of the old value followed by the insertion of the new value. At the search time, the deleted units that match the query will be loaded first and will be kept in memory to filter the "ghost" units in the B^{AO+}-tree and in the TII. Nevertheless, the delete and update operations are uncommon for applications using trajectory data and should not play a significant role in the overall index performance.

3.2.3 Two-level local index reorganization

The B^{AO+}-tree and the TII structures conform to the above stated principles $P1$-$P3$. We explain in Section 3.2.4 the mechanism used by TRIFL to implement the principle $P4$. The

trade-off for having very efficient insertion, low memory requirement, and fast sequential writes, is the loss in the optimality of the search operations. The reason is the suboptimal search performance over the data stored in the TII. However, TRIFL limits and controls the degradation of the search performance (as requested by the design principle $P5$) by means of two restructuring index operations (see figure 3.2).

The first operation *is merging the B^{AO+}-tree and the TII* into a new BAO+-tree. The idea is to integrate the TII data into the B^{AO+}-tree, since the latter offers better search performance. The process consists in reading all the data from the B^{AO+}-trees and the TIIs, sorting, merging and writing it back to new B^{AO+}-trees (see Section 3.3.1). Nevertheless, since the merge uses only block reads and writes, it is highly efficient in flash. The merge also permits the index to absorb the deleted data. In addition, the merge leads to a physical data clustering in each partition, which allows for improved query performance (see Section 3.2.4).

The second operation is the TII *reorganization* that is performed after each merge operation. The TII reorganization consists in changing the number of time intervals of the TII in a partition and keeping balanced the lengths of each time interval in order to reduce the cost of sequential searching in TII's time intervals. A higher number of intervals leads to better lookup performance, but decreases the insertion performance, since more intervals increase the probability of duplicating the data units overlapping with two intervals. More intervals require also more cache memory to efficiently manage the unstable part of the TII. Hence, the best value for the number of intervals depends on the insert/query ratio. A detailed cost model and an online algorithm that automatizes these operations are provided in Section 3.3.

3.2.4 Supporting large granularity I/Os

All the read and write operations over the index structure as presented up here are executed at a flash page granularity. TRIFL employs two mechanisms to increase the index performance by leveraging the (much) more efficient large granularity I/Os of flash devices.

3.2.4.1 Trading cache for block sequential writes

The insertion of new data units requires the creation of new index pages. Such pages are associated first to the hot-data buffer containing the index pages of all the unstable parts of the index. Once such a page becomes full, it is committed and appended at the end of the stable part of the index. Thus, the pages are written sequentially in the stable part.

TRIFL alters a part of the available cache memory by using it as a buffer that stores temporarily full index pages before appending them in block to the end of the stable index. TRIFL computes the size of the temporal buffer (TB) that maximizes the index perfor-

mance, based on three factors: the buffer cache size, a function that indicates the write performance of the flash device with variable block sizes, and an empirically determined function that indicates the index performance with the buffer cache size (see figure 3.6).

3.2.4.2 Logical versus physical clustering

The spatial data partitioning of TRIFL implies the clustering of the trajectory units of the same partition. To optimize the index performance, TRIFL alternates between a logical and a physical clustering of the data. TRIFL divides the stable index storage area in two parts: logically clustered and physically clustered. Typically, there is one data file associated to each storage area. The first stable storage area is used for the initial insertion of the data. All the write operations in this area are done at large granularity (see Section 3.2.4). However, the read operations of the search queries cannot benefit from the same optimization, since the data pages of different partitions are interleaved (i.e., a logical clustering).

The physically clustered[1] area is engendered by the B^{AO+}-tree and TII merges (cf. Section 3.2.3) when all the data in the logically clustered and physically clustered areas are merged and rewritten into a new physically clustered part. During this operation, all the reads and writes are done at large granularity using the TB. Since these merge operations are triggered sequentially among partitions (cf. Section 3.3.1), the data are physically clustered by partition. Hence, the index uses large granularity I/Os for the lookup operations in this area. In particular, the leaf nodes of a clustered B^{AO+}-tree are fetched in block when doing a range search. The size of the block is dynamically determined based on the temporal selectivity of the query and the number of index entries at query time.

3.3 Cost models of TRIFL

We introduced in Section 3.2.3 two index operations that enable TRIFL to contain the degradation of the search performance by integrating periodically the TII data into the B^{AO+}-tree and by reorganizing the TII. In this section, we present first a cost model and an online algorithm that permit TRIFL to have a self-tunable structure. The cost model introduces the formulas that estimate the index performance for a given configuration and workload. The online algorithm uses this cost model to automatically trigger the index optimize operations. Then, we discuss the algorithms used to compute the spatial partitioning and the temporal time span of a TRIFL index.

[1]The term "physically clustered" denotes a block of data pages having consecutive addresses. Since only the Flash Translation Layer (FTL) has access to the physical addresses of the flash device, the page addresses can only be consecutive with respect to the FTL addressing. Nevertheless, this type of "physical clustering" is sufficient to benefit from increased I/O performance.

Parameters	Description
RR	Random page read latency in flash
$SR(B)$	Sequential page read latency by block of B pages
$SW(B)$	Sequential page write latency by block of B pages
I_p	Current number of TII intervals in partition p
T_p	Current index time span in partition p (measured in time units)
$Pages_p^{TII}$	Number of pages of the TII in partition p
$Pages_p^{BAO}$	Number of pages of the B^{AO+}-tree in partition p
M	Number of pages of the buffer cache
np	Number of partitions in the index
$\|\widetilde{T}_{unit}\|$	Average time length of the inserted trajectory units
$\|\widetilde{Q}_t\|$	Average time length of time interval of the queries in the workload (average temporal selectivity of the queries)

Table 3.1: Notations used in this Chapter

3.3.1 Online self-tuning algorithm

To assess the degradation of the query performance, we estimate the query execution time for a given index configuration. The total execution time is computed as the sum of the execution time of all the I/Os that are required to answer the query. We neglect the CPU and the main-memory operation costs. The notations used in this section are listed in Table 3.1.

3.3.1.1 Query cost

Given a range query $Q = (Q_s, Q_t)$, the execution time t^Q is the sum of the search time in each partition p that intersects with Q_s.

$$t^Q = \sum_{P \cap Q_s} t_p^Q \tag{3.3.1}$$

In a partition p, the index retrieves all the data pages that overlap with Q_t from both the B^{AO+}-tree and the TII, i.e.

$$t_p^Q = t_p^{BAO} + t_p^{TII} \tag{3.3.2}$$

The search in the B^{AO+}-tree requires a few RRs to reach the first leaf node from the root, followed by a number of reads of the leaf nodes. Therefore,

$$t_p^{BAO} = RR.h_p^{BAO} + Read.Pages_p^{BAO}.\frac{|Q_t|}{|T_p|} \tag{3.3.3}$$

where $Read = RR$ if the B^{AO+}-tree is logically clustered or $Read = SR(B)$ if the B^{AO+}-tree is physically clustered.

The search in a TII requires one RR to retrieve the TII descriptor, followed by a number of reads to retrieve all the data pages of the intervals that overlap Q_t. Thus,

$$t_p^{TII} = RR + Read.\frac{Pages_p^{BAO}}{I_p}.\left(1 + I_p.\frac{|Q_t|}{|T_p|}\right) \tag{3.3.4}$$

where:$Read$ has the same meaning as above.$\frac{Pages_p^{BAO}}{I_p}$ represents the average number of data pages per time interval, while $\left(1 + I_p.\frac{|Q_t|}{|T_p|}\right)$ indicates the average number of intersected intervals considering that $|Q_t| < |T_p|$.

TRIFL uses these formulas to measure the query performance degradation, i.e., the difference in the query cost between the actual index configuration and the optimized index configuration (see Section 5.2).

3.3.1.2 Merge cost

To merge the B^{AO+}-tree and the TII in a partition, we apply a typical sort-merge algorithm, i.e., the TII is first sorted on t_2 and then merged with the B^{AO+}-tree. The cost of the merge operation in each partition p is:

$$t_p^{merge} = [RR + SW(B)].\left(\lceil \log_M \left(Pages_p^{TII}\right)\rceil.Pages_p^{TII} + Pages_p^{BAO}\right), if M < Pages_p^{TII} \tag{3.3.5}$$

Or

$$t_p^{merge} = [RR + SW(B)].\left(Pages_p^{TII} + Pages_p^{BAO}\right) \tag{3.3.6}$$

To benefit from the high performance block reads, TRIFL employs also an opportunistic "merge" of the B^{AO+}-tree, even if the TII is empty. To this end, the B^{AO+}-tree is copied from the insertion (logically clustered) storage area to the physically clustered storage area. In this case, the cost of the opportunistic merge is:

$$t_p^{merge} = [RR + SW(B)].\left(Pages_p^{BAO(1)} + Pages_p^{BAO(2)}\right) \tag{3.3.7}$$

Where $Pages_p^{BAO(1)}$ and $Pages_p^{BAO(2)}$ indicate the number of index pages in the first area and the second area respectively.

Note that in most cases (i.e., even for moderate to low query rates) the merge is triggered frequently due to its relatively low cost. Therefore, the reads in Formulas (3.3.6) and (3.3.7) are usually executed in block, i.e., $Read = SR(B)$, since all the unclustered data can be accommodated in the buffer cache.

3.3.1.3 Choosing the number of intervals of the TII

TRIFL uses a TII structure in each partition to index the deferred data insertions and the data deletions. A TII consists in a number of time intervals that is initialized at the index creation. The number of intervals of the TII in a partition can be changed after executing a merge in that partition. As indicated in Section 3.2.3, putting more intervals leads to better lookup performance, but decreases the insert performance. Given a mix of insertions and queries, we determine below the number of intervals that maximizes the index throughput.

LEMMA 1. The number of intervals of the TII of a partition that maximizes the index throughput over an index lifespan T_p and a workload containing α queries with an average temporal selectivity of $\widetilde{Q}_t$ is:

$$I_p = min \left\{ \sqrt{\frac{|T_p|}{|\widetilde{T}_{unit}| \left(\frac{|\widetilde{Q}_t|}{|\widetilde{T}_p|} + \frac{2.SW(B)}{\alpha.RR} \right)}}; \frac{M}{np} - \widetilde{h}_{BAO} \right\} \tag{3.3.8}$$

PROOF. The query cost over the TII in a partition is computed by Formula (4), assuming that there are $Pages_p^{TII}$ data pages and I_p intervals. Note that the number of written data pages $Pages_p^{TII}$ is higher than the actual number of pages $Pages_{0_p}^{TII}$ received by the index, because of the duplication of the data units (cf. Section 3.2.2):

$$Pages_p^{TII} = Pages_{0_p}^{TII}.(1 + DuplicateProbability) \tag{3.3.9}$$

where $DuplicateProbability = \frac{|\widetilde{T}_{unit}|}{|\widetilde{T}_{interval}|} = \frac{|\widetilde{T}_{unit}|.I_p}{|\widetilde{T}_p|}$.

Thus, the cost of insertion is: $t^{ins} = SW(B).Pages_p^{TII}$. The total cost of the workload over the time period T_p is: $t^{work} = \alpha.t^Q + t^{ins}$, where α is the number of queries. Assuming a uniform distribution of the queries in time, we obtain:

$$t^{work} = \alpha.\left[RR + \frac{1}{2}.RR.\frac{Pages_p^{TII}}{I_p}.\left(1 + I_p.\frac{|\widetilde{Q}_t|}{|\widetilde{T}_p|}\right)\right] + SW(B).Pages_p^{TII} \quad (3.3.10)$$

where

$$Pages_p^{TII} = Pages_{0_p}^{TII}.\left(1 + \frac{|\widetilde{T}_{unit}|.I_p}{|\widetilde{T}_p|}\right) \quad (3.3.11)$$

and the $\frac{1}{2}$ factor accounts for the fact that the queries "see" on average only half of $Pages_p^{TII}$ in the interval T_p. If we consider t^{work} as a function of I_p and combine Formulas 3.3.10 and 3.3.11, then t^{work} is minimized when I_p is equal to the first value in Formula 3.3.8 in Lemma 1.

The insert cost t^{ins} above considers the ideal case in which the unstable pages of a TII can be buffered in cache before being written to the stable TII. This assumption holds if the buffer cache can accommodate all the hot data pages of the index, i.e. $M \geq np.\left(\widetilde{h}_{BAO} + \widetilde{I}\right)$, where $\widetilde{h}_{BAO}$ and $\widetilde{I}$ indicate the average height of the B^{AO+}-trees and the average number of the TIIs' intervals in the index. On the other hand, the number of RWs increases rapidly when the size of M decrease and $M < np.\left(\widetilde{h}_{BAO} + \widetilde{I}\right)$ (see Figure 3.6). Therefore, to avoid increasing the number of RWs and maintain the high level of the insert performance, we limit the number of intervals in a partition to $\frac{M}{np} - \widetilde{h}_{BAO}$, which corresponds to the second value in Formula 3.3.8. Obviously, the system assigns the minimum value of 1 to I_p whether $\frac{M}{np} - \widetilde{h}_{BAO}$ is less than 1.

3.3.1.4 Online self-tuning algorithm

TRIFL implements a self-tuning algorithm that is based on the above cost model and a few statistics about the index structure, which are collected and updated continuously. The online algorithm (see Algorithm 1) monitors several index structure parameters. These parameters are then used to estimate the benefit and the cost of the merge in each index partition. However, the reorganizing operations are triggered for the entire index (line 6 in Algorithm 1) to benefit from the large granularity I/Os and reduce the reorganization cost (see Section 3.3.1). The benefit of the merge in a partition p is computed as the difference between the total actual query cost since the last merge $t_p^{Q\_lm}$ and an optimal (hypothetical) query cost $t_p^{Q\_opt}$ considering that all insertions are integrated directly in a clustered B^{AO+}-tree. The algorithm triggers the index reorganization if the benefit in hindsight of the merge operation outweighs the operation cost (first condition at line 5 in Algorithm 1). After each merge, the algorithm recomputes the number of intervals I_p (with Formula 3.3.8) in each partition to adapt the TIIs to the index workload (line 7 in Algorithm 1).

Another issue is that the query degradation is caused by the data inserted since the last

ALGORITHM 1. Online Self-Turning Algorithm

Monitored parameters in each partition: $\alpha, \widetilde{Q}_t, \widetilde{T}_{unit}, T_p, Pages_p^{TII}, Pages_p^{BAO}$

1. while (true)
2. update the monitored parameters
3. **for** each partition p **do**
4. compute $t_p^{Q\_lm}, t_p^{Q\_opt}$ and t_p^{merge}
5. if $\left(\sum_{p \in partitions}\left(t_p^{Q\_lm} - t_p^{Q\_opt}\right) > \sum_{p \in partitions} t_p^{merge} \right.$
$\left. OR \quad \frac{\sum_{p \in partitions} t_p^{Q\_lm}}{\sum_{p \in partitions} t_p^{Q\_opt}} > query_{deg}^{th}\right)$ then
6. execute merge (B_p^{AO+}-tree, TII_p), $\forall p \in partitions$
7. compute I_p, $\forall p \in partitions$

merge. However, the cost of the merge is directly proportional to the global indexed data size. Hence, the merge cost continuously increases as more data is inserted into the index. This means that, assuming an approximately constant query rate, the query degradation increases over time with the increase of the merge cost. The self-tuning algorithm uses a user-defined parameter that defines an upper bound of the admitted query degradation, i.e.,$query_{deg}^{th}$. An index merge is also triggered if the ratio between the actual and the optimal query cost is larger than the query degradation threshold (second condition at line 5 in Algorithm 1).

3.3.2 Partitioning in TRIFL

This section presents the spatial and the temporal partitioning in TRIFL.

3.3.2.1 Spatial partitioning

As with the existing space partitioning indexes (see Section 2.3.1 in Chapter 2), TRIFL partitions the space before indexing the trajectory units in each partition on time. A few of the trajectory indexes for magnetic disks [16], [24], [96] propose cost models to compute the best spatial partitioning with respect to a query workload. Yet, these methods do not consider the insert performance in the computation of the spatial partitioning. This is an important limitation in our context that combines the flash memory (having poor random write performance) and trajectory flows (implying a large number of insertions). Therefore, TRIFL computes the spatial partitioning by taking into account both the queries and the insertions in the workload. In this part, we discuss the spatial partitioning only in the case of road networks. Nonetheless, a most similar approach can be employed to partition a 2D space by using an adaptive grid partitioning as in [16], [24]. Also, the cost model we

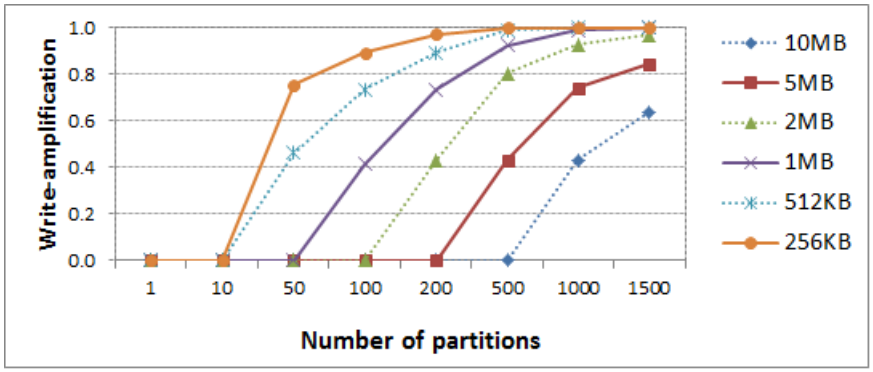

Figure 3.6: Write-amplification depending on the cache size and the number of partitions

introduced above is independent of the type of the indexed space.

The spatial partitioning of a road network in TRIFL sums up to finding the number of partitions np that minimizes an index workload. This is due to the fact that the partitioning of the graph network into np network regions can be easily computed by the existing graph partitioning algorithms [53]. Hence, the number of partitions np is sufficient as input to compute the actual network partitioning, i.e., which network edge belongs to which partition. Highly efficient implementations of such partitioning algorithms are also available. We employed METIS [41] to compute the spatial partitioning in TRIFL. Also, if a spatio-temporal distribution of the data is provided, the partitioning takes it into account and produces network regions that are balanced with respect to the amount of data in each region (see figure 3.1).

Let $\widetilde{\rho}$ be the average temporal density of the indexed trajectory flow (i.e., number of inserted data units per time unit). Let $\{Q^i\}$ be the set of queries composing the query load. Let $|T_i|$ be the length of the index time span (see Section 3.3.2.2). Then, the total insert cost is:

$$t^{ins} = \widetilde{\rho}.|T_i|.\left[\frac{SW(B)}{BS} + wa_{cache}(np, M).(RW + RR)\right] \tag{3.3.12}$$

BS is the block size (number of entries) of the data pages (leaf nodes in the B^{AO+}-trees or data pages in the TII). We do not consider the cost of writing the non-leaf nodes of the B^{AO+}-trees in each partition since this part is negligible in the overall cost. wa_{cache} is a write-amplification factor that indicates the probability for a data unit in the hot data buffer to be temporarily written in flash before being flushed in the stable part of the index, depends on the number of partitions np and the size of the buffer cache M (see figure 3.6). As indicated in Section 3.2.2, the newly inserted data units are temporarily stored in the hot data buffers of the corresponding partition. Once a hot data page is filled, it is flushed in the stable area of the index. Each partition has only a few hot data

pages (in red in figures 3.3 and 3.4). If the number of partitions np is small, all these hot data pages can be accommodated in the buffer cache and the write-amplification wa_{cache} is equal to 0. Conversely, for a large number of partitions and/or a small buffer cache M, some of the hot data pages will be evicted from the buffer cache before being filled and written to the stable area. At worst, a data page has to be read into the buffer cache and evicted from the buffer cache for each newly inserted data unit in that page (i.e., a wa_{cache} equal to 1). Therefore,the insertion cost is minimized with a number of partitions $np \leq np_{wa=0}$, where $np_{wa=0}$ is the maximum number of partitions for which wa_{cache} is equal to 0.

The total query cost for a query load $\{Q^i\}$ is:

$$t^{query} = \sum_{Q \in \{Q^i\}} t^Q \tag{3.3.13}$$

where

$$t^Q = \sum_{p \bigcap Q_S} \left(RR.h_p^{BAO} + SR(B).\frac{\widetilde{\rho}}{np}.|Q_t| \right) \tag{3.3.14}$$

The first term in Formula 3.3.14 corresponds to the cost of traversing the B^{AO+}-tree in each partition intersected by the query spatial window. The second term corresponds to the cost of scanning the leaf data pages intersected by the query temporal interval. $|Q_t|$ is the length of the query time interval. Here we consider that each partition consists in a clustered B^{AO+}-tree since the unclustered B^{AO+}-tree and the TII are temporary structures that are absorbed and clustered through index merges. Similar to the insert cost, the number of partitions influences the query cost. By increasing np, the spatial extent of the partitions decreases. This can increase the number of partitions to be visited by the queries, leading to a higher cost of the first term in Formula 3.3.14. At the same time a large np decreases the data volume in each partition, which lowers the cost of the second term in Formula 3.3.14).

The best value for np is the value that minimizes the global index workload, i.e. $t^{work}(np) = t^{ins}(np) + t^{query}(np)$. To find the best np value, TRIFL simply estimates t^{work} for several np values in a given range (see Algorithm 2), e.g., from 1 to 1000 partitions with a step of 10, before the index creation and picks the np with the smallest t^{work}. The upper bound of the np depends on the extent of the indexed space and on the density of the data flow, i.e., larger spaces and denser data flows typically require more partitions.

ALGORITHM 2. Spatial Partitioning Algorithm

Input: $\left\{Q^i\right\}, \widetilde{\rho}, |T_i|, max\_np$ network graph NG

1. $np = 1; t_{min}^{work} = t^{work}(1);$
2. **for** $p = 2$ to max_np **do**
3. compute spatial partitioning GraphPartitioning(p, NG);
4. compute $t^{work}(p)$;
5. if $\left(t_{min}^{work} < t^{work}(p)\right)$ then
6. $np = p; t_{min}^{work} = t^{work}(p);$
7. return GraphPartitioning(np, NG);

3.3.2.2 Temporal partitioning

We present in this section the algorithm that computes the timespan of an index component in TRIFL. In Section 3.1.3 we discussed the benefit of a temporal partitioning of the index structure. TRIFL triggers periodically a merge operation when the query performance degrades (see Algorithm 1). Since the cost of the merge is directly proportional to the global indexed data size, the cost continuously increases as more data is inserted into the index. Therefore, the temporal index partitioning limits the merge cost of an index component assuming that the volume of data collected in a bounded time interval is also limited.

At the same time, the temporal partitioning increases the query cost of the queries having query time intervals Q_t that overlap with two index components (see figure 3.3) for two reasons. First, these queries have to visit two indexes instead of one, which can double the index access cost (i.e., the first term in equation 3.3.14). Second, a more subtle issue regarding the queries is caused by the fact the data is scanned using large granularity I/Os in a partition (i.e., the second term in equation 3.3.14). Recall that the index merge leads to a physically clustered index, which allows data pages to be efficiently fetched in block when doing a range search (see Section 3.2.4). The size of the block (i.e., number of data pages) is dynamically determined based on the temporal selectivity of the query and the number of index entries at query time, i.e., $B = \frac{\widetilde{\rho}}{np}.|Q_t|$ from equation 3.3.14. Typically, larger block sizes lead to better query performance. Therefore, the index scan cost (i.e., the second term in equation 3.3.14) is also increasing for the overlapping queries since the data is fetched in two smaller blocks instead of one large block. Assuming a uniform distribution of the query time interval, for the index component i having a timespan length $|T_i|$, the ratio of queries that overlap with two indexes is $r = \frac{|\widetilde{Q}_t|}{|T_i|}$, where $|\widetilde{Q}_t|$ indicates the average length of the query time intervals. To have a small impact on the query performance, the ratio r has to be small, i.e., $r \ll 1$. In our implementation we empirically assigned a value of 0.05 to r, which is sufficiently small to have only a marginal effect on the query

performance. This imposes a minimum timespan of an index component of

$$|T_i^{min}| = \frac{|\widetilde{Q}_t|}{r} \tag{3.3.15}$$

From the index reorganization cost perspective, the timespan of an index component should be as shorter as possible to reduce the cost of index merges. Therefore, taking into account the above reasoning about the query performance, we set the timespan of a TRIFL index component i to $|T_i^{min}| = \frac{|\widetilde{Q}_t|}{r}$, where $|\widetilde{Q}_t|$ is the observed average length of the query time intervals and r is a user defined parameter (0.05 in our implementation).

3.4 Experimental evaluation

We present in this section the experimental evaluation of the TRIFL framework in comparison with the representative indexing techniques in magnetic disk and flash storage. The experimental setup is described in Section 3.4.1. In Section 3.4.2 we analyze the detailed insert and query performance of the tested methods. The global index performance for different workloads is discussed in Section 3.4.3. We detail the scalability of TRIFL* in comparison with other methods in section 3.4.4. Then, Section 3.4.5 compares the robustness of the tested methods with the cache size. In Section 3.4.6 we discuss the importance of the spatial and temporal partitioning for indexing trajectory flows. Section 3.4.7 presents the evaluation of the cost models. Finally, we sum up the experimental results in Section 3.4.8.

3.4.1 Experimental setup

All the experiments have been conducted on a workstation having an Intel Core(TM) i5-2400 3.1GHz CPU with 2GB of main memory (note: the experiments do not require so much memory) running Windows XP. Trajectory indexing within an SD card could be of interest (e.g. for sensor devices [64]). However, in a majority of the expected use-cases, modern SSDs will be the prevalent flash storage medium. Hence, we selected a commodity SSD having representative characteristics of contemporary SSDs and consider this storage device as the baseline. At the same time, TRIFL is designed to be generic for any type of flash storage. Therefore, to fully assess the properties of TRIFL, we consider in our experiments an SD card, but also an HDD. The rationale is that, compared against each other, these three devices exhibit quite different behaviors (see the next section) and permit evaluating the importance of all the optimizations in TRIFL (e.g., reduction of RWs, usage of large granularity I/Os, index memory footprint, etc.).

3.4.1.1 Selected storage devices

We selected one SD card (i.e., Kingston Ultimate XX 233X SDHC 8GB), one SSD (i.e., OCZ Vertex 4 128GB) and one HDD (i.e., Western Digital 1TB SATA 7200RPM) as representative storage devices. The SD card was accessed via a USB 2.0 adapter, while the SSD and the HDD were connected by a SATA interface. These storage devices have very different I/O performance as indicated in Figure 3.7. We measured the average performance with both random I/Os, which by definition are executed at a page (2KB) granularity, and sequential I/Os executed both at a page granularity (i.e., 2KB) and at a block granularity (i.e., 512KB). The SD card exhibits the asymmetric read-write performance. The read operations (i.e., RRs and SRs(2KB)) in the SD card are 10 times faster than the SWs(2KB) and 35 times faster than the RWs. On the other hand, the RWs in the SSD are at the same cost as the SWs executed at a page granularity (i.e., 2KB). The large granularity I/Os improve significantly the throughput of all the devices. The speedup due to large I/Os is about a factor of 10 for the reads and writes in the SSDs and the reads in the SD card. Moreover, large I/Os are particularly beneficial for the writes in the SD card as the speedup is about a factor of 100.

The selected HDD shares a common characteristic with the flash devices, i.e., increased throughput with sequential large granularity I/Os. Nevertheless, compared with the two tested flash devices, the HDD has peculiar characteristics (see Figure 3.7). On the one hand, with the HDD the random write has a similar (lower) cost to the random read as with the SSD. On the other hand, the page sequential I/Os are much more efficient than the random I/Os in the HDD, while there is very little difference between random and sequential page I/Os in the flash devices. Finally, the large I/Os significantly improve the sequential writes in the HDD (by a factor of 20) as in the SD card (a factor of 100), while for the SSD the improvement is relatively low (a factor of 5).

3.4.1.2 Tested methods

We implemented in C the global framework of TRIFL introduced in Section 3.1. The framework employs the standard OS file system facilities for storage. However, to achieve accurate performance measurements, we used file access primitives that bypass the file system caching. Therefore, all I/Os are immediately flushed to disk and are not buffered before in the OS queue of the disk controller. To stress the importance of the index tuning, we measured the performance of TRIFL with and without the online self-tuning algorithm. We denote hereafter by TRIFL and TRIFL* the proposed index structures. The only difference between the two instances is that TRIFL* includes the online self-tuning algorithm, which triggers the index reorganization operations.

We also implemented four alternative data storage and indexing methods to TRIFL. Since

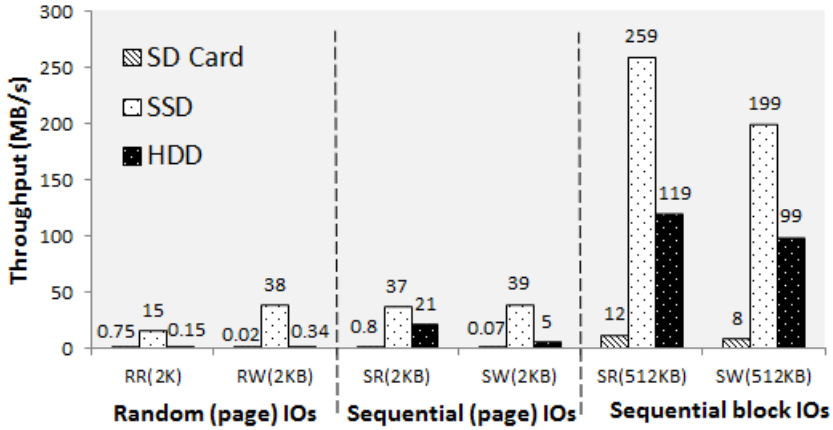

Figure 3.7: Measured throughput of the tested storage devices for I/Os at page or block granularities

the main novelty of TRIFL is the local, low level data indexing, we use the same spatial partitioning with three of the alternative tested methods and replace only the local indexing method. The fourth competitor [25] uses a specific type of spatial partitioning as described below. The first alternative to the proposed index is based on the classical B^+-tree, as in the state-of-the-art trajectory indexes [16], [19], [24], [96]. Similar to TRIFL, these methods partition first the space and then index the data in each partition on time with a B^+-tree. The second alternative replaces the B^+-tree by a representative flash-aware tree index, i.e., the FD-tree [61]. We chose FD-tree for two main reasons. First, FD-tree offers excellent performance in flash compared with other flash-aware trees and to the B^+-tree [61]. Second, the design of FD-tree (see Section 2.3.2 in Chapter 2) is generic as it considers both the asymmetric R/W performance and the large granularity I/Os in flash, which makes it more adapted to our context that considers the wide range of flash devices. The third alternative uses the LSM-tree [76]. Although designed for magnetic disks, the LSM-tree has a few characteristics that make it an interesting candidate for indexing data flows in flash (see Section 2.3.2 in Chapter 2). LSM-tree buffers all the updates in RAM and commits them in batch when the RAM is filled. This precludes any random writes, amortizes the cost of individual updates, and allows for large granularity I/Os. Hence, LSM-tree is very efficient for processing insert oriented workloads.

Finally, we also consider an index structure for road network trajectories, i.e., the MON-tree [25], that is based on the multi-dimensional R-tree. MON-tree is a combination of a 2D R-tree (called the top R-tree) used for indexing the network roads and a set of 2D R-trees (called the bottom R-Trees) for indexing the objects' movements along the roads. Therefore, MON-tree uses a bottom 2D R-tree for each road in the network. MON-tree considers two kinds of possible roads: an edge-oriented model (in which each network edge is a road) and a route-oriented model (in which a road covers several concatenated edges,

i.e., a path in the network).We implemented MON-tree with the route-oriented model (see the details in Section 3.4.1.3) since this configuration leads to better index performance [25].

We implemented the B^+-tree and the LSM-tree and used the available implementation of the FD-tree [62]. We implemented MON-tree using the online implementation of R-tree (available at superliminal.com/sources/sources.html). We also implemented an Least Recently Used (LRU) buffer cache that was used for all the tested methods.

3.4.1.3 Index workload

We used the generator of moving objects in networks proposed in [17], [67] to create synthetic datasets that are representative enough in terms of trajectory variety and data size. The results reported in the part are based on a trajectory flow generated in the road network of Stockton (San Joaquin County, CA), which is an average size road network having 24123 segments and 18496 nodes. We generated a trajectory flow containing 4.15 million units from 50 thousand vehicles over a period of 1000 time units, which contains sufficient data to cover the life span of a TRIFL index. Based on this dataset, we simulate three types of insertion loads that contain 100% *deferred insertions*, 100% *timely insertions*, and a *mixed* 50% timely - 50% deferred insertions. Each workload includes also the random deletion of 5% of the trajectories.

We generated a set of 500 range and NN queries that are placed randomly in space and in time. The query set contains for two-thirds range queries and for one-third NN queries. Also, half of the range queries are 2D queries, i.e., Q_s is a 2D square window, while for the other half of the range queries Q_s is a path in the network. The spatial and the time interval were fixed to 2.5%, 5% and 10% of the total space and time of the dataset for the 2D queries with equal probability. For the path queries the spatial interval is 0.25%, 0.5% and 1% respectively of the total number of roads in the network, while the time interval was fixed to the same size as for the 2D queries, i.e., 2.5%, 5% and 10% of the total time of the dataset. The resulting average selectivity of the queries in number of selected trajectories varies from 0.03% to 0.44% for the 2D queries and from 0.32% to 1.44% for the path queries. The selectivity of the NN queries is set to 25, 50 and 100 nearest neighbors for a time interval 2.5%, 5% and 10% respectively of the total time of the dataset.

Based on the trajectory dataset and query set, we generated workloads having insert/query ratios varying from 1 to 10000, 1 corresponding to a query intensive workload and 10000 corresponding to an insert intensive workload. However, to confine the execution time of the tests to a reasonable amount of time, we had to limit the maximum number of queries in the workloads to 50000. Therefore, for the workloads that have the ratios 1 and 10, we inserted first 98% and 85% of the data and then measured the index performance for

the remaining mix of inserts and queries. We consider the query/insert ratio of 100 as the baseline, since a single trajectory object requires many insertions. For example, the average length of a trajectory in our dataset has 83 units.

For each workload, each tested method and each tested storage device, we measured the total time required to solve the workload as well as the detailed I/Os executed by the indexes (i.e., the number of RRs, RWs, SR(B)s, SW(B)s). We used the typical value of the page size in flash, i.e., 2KB. The default cache size was fixed to 10MB (i.e., approximately 5% of the maximum index structure size) for all the methods mostly because the competing methods (B^+-tree, FD-tree and LSM-tree) require a relatively large cache to have good performance with many local indexes. However, TRIFL* works also well with lower cache sizes (see more details in Section 3.4.3). With a cache size of 10MB, the spatial partitioning algorithm (see Algorithm 2) partitions the network space in 500 partitions[2]. Also, the number of intervals of the TIIs for TRIFL/TRIFL* is equal to 10 as computed by Formula 3.3.8 (see Section 3.3.1) in this configuration. Similar to the FD-tree, the data insertions and the restructuring operations in TRIFL are executed at large granularity I/Os, i.e., in blocks of 256 pages, since this granularity offers the best index performance. Furthermore, since MON-tree performs better on a route-oriented model, we concatenated the segments of the above Stockton road network to generate long routes. From the 24123 segments of Stockton, we generated 6039 routes using a direction-based partitioning as proposed in [25]. Then, a 2D R-tree was created to index the trajectory units contained by each route since a trajectory unit can be seen as a two-dimensional segment. Specifically, in a 2D R-tree the first dimension indexes the time interval of the unit, while the second dimension indexes the relative positions on the respective route.

3.4.2 Insertion and query performance

Figure 8 shows the average I/O insert performance (left) and I/O query performance (right) with the three types of insertion flows and an insert/query (i/q) ratio of 100. We observed similar trends at different i/q ratios. TRIFL and TRIFL* have by far the best insert performance, which is also constant across the range of the insertion flows. TRIFL optimizes the data insertions by buffering in cache the hot index pages and by committing these pages in batch when they become stable. At the same time, TRIFL creates only a small number of hot pages for each local index, which limits the total cache requirements and increases the insertion efficiency. LSM-tree and FD-tree also try to buffer the data before writing them in batch to flash. However, the LSM-tree merges the in memory index pages with the on disk pages at each RAM flush, which increases the insertion cost. Differently, TRIFL* only triggers an index merge whenever the query performance

[2]To simplify the experimentation, we took the number of partitions (i.e., 500) that offer near-optimal performance for both the SD card and the SSD as indicated by the cost model.

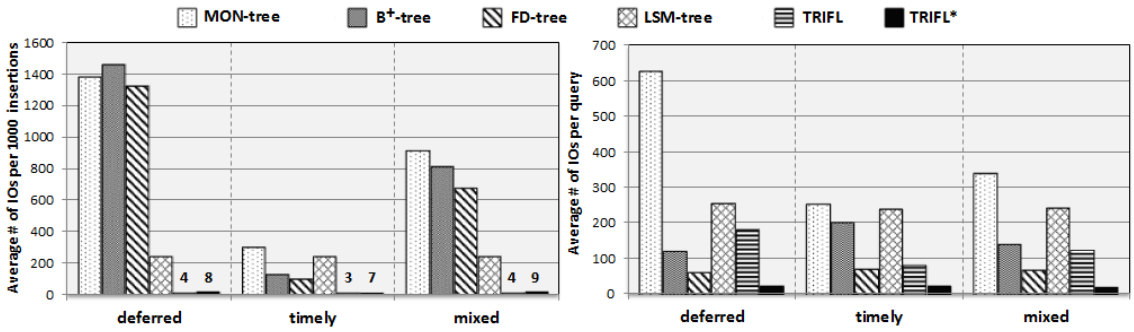

Figure 3.8: I/O insert (left) and query (right) performance of tested methods at 100 i/q ratio

degrades significantly. Similar to TRIFL, LSM-tree has stable insert performance for the three types of insertion flows.

FD-tree uses a relatively small head B^+-tree of 512KB to buffer the updates before flushing them in batch to subsequent index levels. For optimal insert performance, the head tree has to be entirely located in cache. Nevertheless, given the large number of local indexes and the limited amount of cache, i.e., 10MB, the insert performance of the FD-tree is degraded and similar to the insert performance of the B^+-tree, which is not particularly optimized for high insertion rates. MON-tree has similar insert performance with the B^+-tree with deferred and mixed insertion workloads, but is significantly worse with timely insertions.

As expected, varying with the cache locality of the insert workload, the B^+-tree, the MON-tree and the FD-tree have the best insert performance with timely insertions and the worse performance with deferred insertions, and intermediate performance with mixed insertions. With the timely updates all the insertions take place in the right-most leaf node of the B^+-trees or the head B^+-trees of the FD-tree. Given the very low number of nodes affected by updates, there is a high probability for a cache hit when modifying these nodes, which in turn massively reduces the number of random I/Os. Oppositely, the deferred workload inserts data randomly and therefore any index leaf node can be affected by an update. Therefore, the probability of a cache hit drastically reduces with the deferred insertions, which greatly increases the number of random I/Os.

On the other hand, the query performance of TRIFL varies with the type of the insertion flow, since the deferred insertions go into the TII. The more deferred insertions there are, the larger the number of I/Os is for TRIFL. TRILF* overcomes this query performance variability with the help of the self-tuning algorithm. The price to pay is a slightly higher insertion cost. However, the I/O increase in the insertion cost is largely compensated by the I/O decrease of the query cost. Indeed, TRIFL* has the best query performance measured in the number of I/Os. The reason is that for the part of the data that is

physically clustered through index merges, the queries are evaluated using large granularity I/Os, which reduces the number of I/Os to process queries. The FD-tree also uses large granularity I/Os for query processing and exhibits low query costs. But in an FD-tree, the data are split among several index levels, which decreases the efficiency of retrieving the data in large blocks. The B^+-tree and the LSM-tree require a high number of I/Os to process queries since these methods do not use large granularity I/Os. In addition, an LSM-tree has several index components, which increases even more the query processing costs compared with the B^+-tree.MON-tree has the poorest query performance. The trajectory data exhibit large amounts of overlapping in both the spatial and the temporal dimensions, which requires searching several paths from the root to the leaves in the R-trees of the MON-tree. The query performance of the MON-tree is also sensitive to the type of the workload and degrades significantly with deferred insertions.

3.4.2.1 Detailed I/O performance

The number of I/Os plays an important role in the overall index performance. But the type of I/Os is even more important for flash devices since an I/O can contain many pages. Figure 3.9 details the total number of pages that have to be read/written by each method to solve the mixed workload, as well as the type of these page accesses. Figure 3.10 gives the total the number of I/Os executed by each method with the same mixed insertion flow. TRIFL requires only a few SW(B) for insertions and is the most efficient method for data insertions. The rest of the methods have comparable insert performance if we consider the total number of pages read and written (Figure 3.9, left). However, the differences in insert performance become prominent if we consider the total number of I/Os (Figure 3.10, left). Most of the page access of TRIFL* are executed using large granularity I/Os, which are very efficient in flash and also in the HDD. Hence, the insert performance of TRIFL* is similar to TRIFL. LSM-tree and FD-tree also use large granularity I/Os for data insertion processing. However, these methods still require a significant number of RRs and RWs compared with TRIFL*. The B^+-tree and the MON-tree have the worst insert performance since all the I/Os are done at a page granularity.

Nevertheless, if we consider the number of I/Os (Figure 3.10, right) and not the number of accessed pages, TRIFL* is the most efficient method since most of the accessed pages are retrieved in block. FD-tree also uses large granularity I/Os to process queries, but the number of block I/Os is only a small percentage from the total number of disk accesses.

3.4.2.2 Insert and query time

Figure 3.11 presents the insertion time (left) and the query time (right) on the three test storage devices. In general, the insert and query time are related to the number of

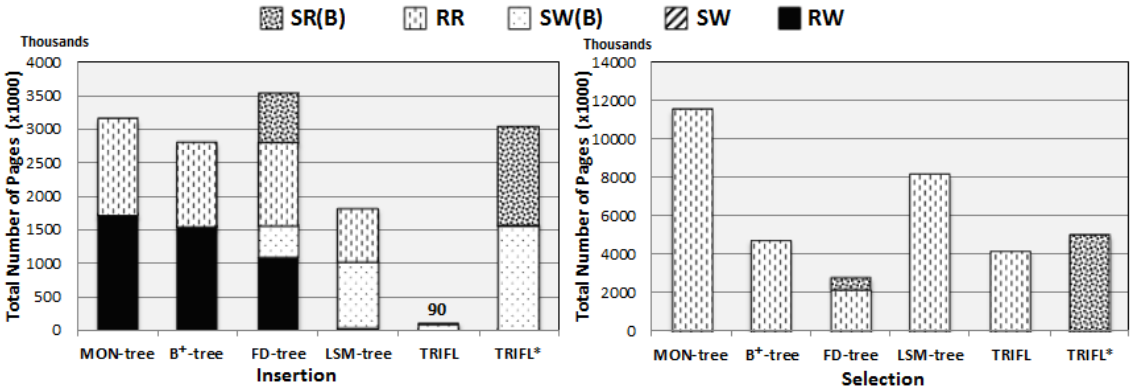

Figure 3.9: Detailed insert (left) and query (right) performance of the tested methods for the mixed insertion flow (in number of pages)

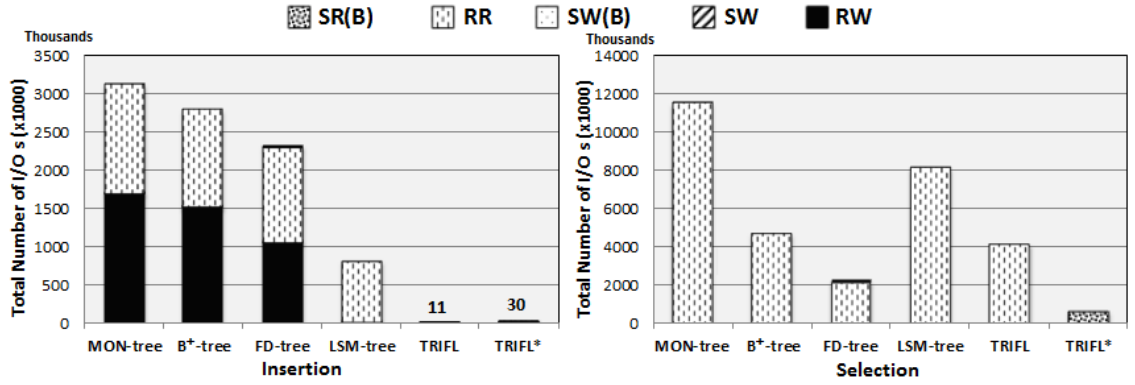

Figure 3.10: Detailed insert (left) and query (right) performance of the tested methods for the mixed insertion flow (in number of I/Os)

executed I/Os (and not the number of accessed pages). Regarding the insertion, TRIFL and TRIFL* exhibit (much) better performance than their competitors on all the tested devices. LSM-tree also offers very good insert performance that is not affected by the type of the insertion flow. As expected, the B^+-tree, the MON-tree and the FD-tree have good insert performance with timely data due to the workload cache locality, but offer poor insert performance with deferred and mixed data flows. The reason is that these methods require a very large buffer cache for processing random data inputs over a high number of local indexes; otherwise many random reads and writes will be generated.

The query execution time changes the ranking between the methods. TRIFL* offers by far the best query performance for every kind of workloads and devices. The FD-tree comes next having better query execution time than TRIFL and the B^+-tree.The main reason is that TRIFL* and FD-tree use large granularity I/Os to process the queries (96% for TRIFL* and 22% for FD-tree). In contrast, TRIFL and B^+-tree exhibit lower query performance, which also varies with the type of the data insertion flow. For example,

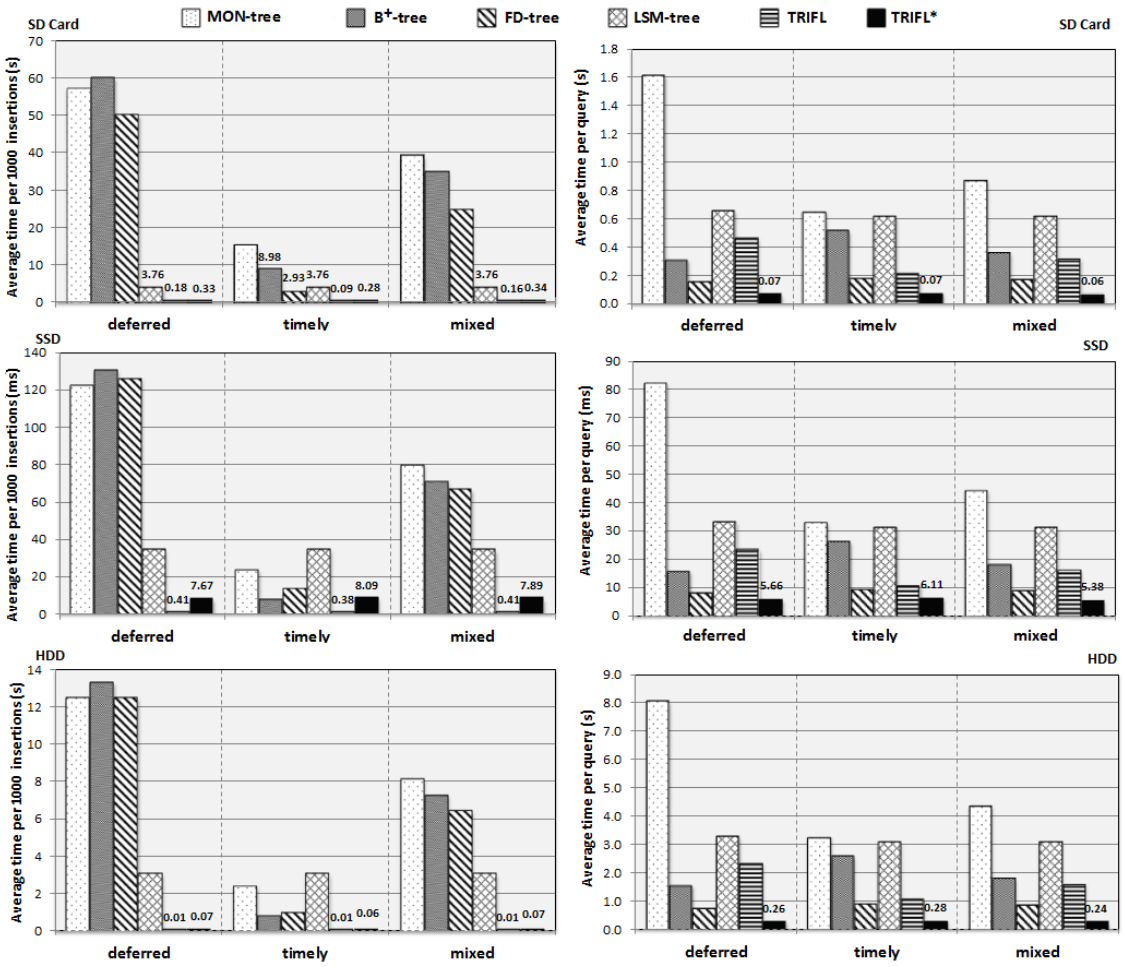

Figure 3.11: Insertion (left) and query (right) time for the tested methods on the SD card, the SSD and the HDD at i/q ratio of 100

TRIFL requires a smaller amount of time for query processing than the B^+-tree for timely and mixed insertions but consumes more time than the B^+-tree with deferred insertions where most of the inserted data go into the TII. The LSM-tree has stable query time with different types of insertion loads, while the MON-tree has the most variable query time with changing workloads. However, both methods exhibit poor query performance.

Overall, TRIFL proves to have the best insert performance, but not the best query performance which also depends on the type of insertions (i.e., timely or deferred). In contrast, compared with TRIFL, TRIFL* slightly trades insert performance to obtain significant gains in query performance. TRIFL* also shows much better performance compared with all four tested methods for both insertions and queries.

3.4.3 Index throughput

The main challenge in indexing the trajectory flows is to efficiently process the massive index insertions, while processing the queries over the indexed data. Hence, the index throughput, i.e., the number of operations processed by the index in a time unit (e.g., one second), is the main indicator of the global index performance. Figure 3.12 shows the relative throughput of the tested methods for the deferred and the timely insertion flows at different insert/query ratios. For better readability of these graphs, we take the throughput of the B^+-tree as a base line, and show for each method the ratio between its throughput and the throughput of the B^+-tree. The first general observation is that the throughput of each method increases with the increase of the i/q ratio. This is normal since an insert operation is much cheaper than a query. The second general observation is that TRIFL* offers the best throughput no matter the values of the testing parameters. This is as expected since TRIFL* has both excellent query and insert performance as discussed in Section 3.4.2. Also, the performance differences between TRIFL* and the other methods are higher with the deferred data flows. TRIFL delivers comparable throughput values with TRIFL* only for workloads having large i/q ratios, i.e., for insert-oriented workloads, which indicates the importance of the tuning process. The FD-tree has throughput values comparable to TRIFL* only on the SSD device and low i/q ratios. Similarly, the LSM-tree is competitive only with the SD card device and large i/q ratios. The performance of the B^+-tree is comparable to that of the FD-tree and the LSM-tree. The B^+-tree manages to be more efficient than the FD-tree and the LSM-tree on the SSD and HDD for the highest i/q ratio. Given both the poor insert and query performance, MON-tree has the lowest throughput in most cases and for this reason, we do not include MON-tree in following remaining parts of the evaluation.

To sum up, on the SD card, TRIFL* offers a throughput that is 3.3X to 334X (for deferred insertions) and 3.4X to 27X (for timely insertions) larger than FD-tree,4.5X to 12X and 5.7X to 15X larger than LSM-tree, and 7.1X to 379X and 6.6X to 51.7X larger than B^+-tree, for i/q ratios between 1 and 10000. Similarly, on the SSD the throughput of TRIFL* is 1.7X to 81.8X and 2X to 12.8X larger than FD-tree, 4.8X to 21.3X and 5X to 31.5X larger than LSM-tree, and 3.1X to 84.4X and 2.5X to 6.7X larger than B^+-tree. Even on the HDD the throughput of TRIFL* is 3.9X to 270X and 4.6X to 21.9X larger than FD-tree, 13.5X to 70.4X and 11.9X to 70.2X larger than LSM-tree, and 9.9X to 283X and 16.5X to 10.1X larger than B^+-tree. Compared with TRIFL, the throughput of TRIFL* is on average 5X (for deferred insertions) and 2.5X (for timely insertions) higher than TRIFL on SD Card, 3X and 1.6X on SSD, and 10X and 4.8X on HDD. Overall, TRIFL* provides a throughput that is on average 31X higher than its competitors depending on the i/q rate and the type of insertions.

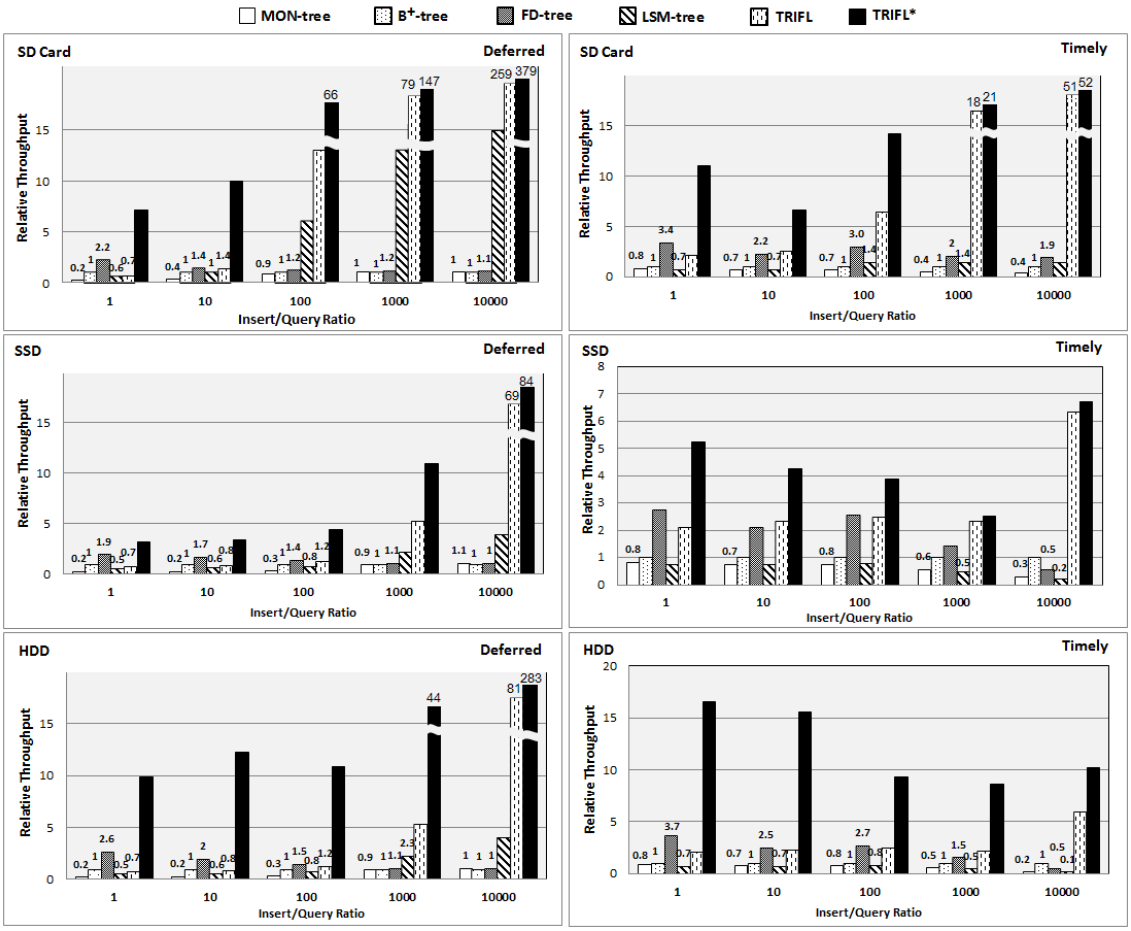

Figure 3.12: Relative throughput for the tested methods on the SD card, the SSD and the HDD at different i/q ratios and types of insertions (deferred and timely)

3.4.4 Scalability with the road network size and the data density

In this section, we analyze the scalability of TRIFL* in comparison with the other tested methods, i.e., the B^+-tree, the FD-tree, the LSM-tree and the basic TRIFL (without self-tuning). In particular, we seek to respond to the following two questions: (i) what is the impact of the road network size on the index performance? And (ii) what is the impact of the average temporal density $\widetilde{\rho}$ of the indexed data (see Section 3.3.2.1) on the index performance? The network size can influence the query performance since the query size may be proportional with the network size as it is the case in our query loads (see Section 3.4.1.3). The data density reflects the traffic density (i.e., number of moving objects at a certain time instant in the network) and may influence both the insert performance and the query selectivity.

To measure the index scalability, we proceed as follows. We use two road networks: (i) a "small" network, i.e., the road network of Stockton city, which has 24123 segments

and 18496 nodes (the same was used in all the above experiments); and (ii) a "large" network, which is also based on the road network of Stockton, but covering a larger area and having 52738 segments and 48576 nodes. The larger Stockton map was generated with the MNTG generator [Mokbel et al. 2013] (available at mntg.cs.umn.edu). Then, we generate [17] on each road network two datasets: (i) a "small" dataset containing 50 thousand trajectories with 4.1 million units over a period of 1000 time units; and (ii) a "large" dataset containing 100 thousand trajectories with 8.7 million units over a period of 1000 time units. Therefore, the data density in the larger dataset is twice the density of the smaller dataset since the global temporal interval is the same for the two datasets. In the following graphs, we identify each of the four experimental settings by: (i) SN-SD (small network - small dataset); (ii) SN-LD (small network - large dataset); (iii) LN-SD (large network - small dataset); (iv) LN-LD (large network - large dataset). For each setting, we use the *mixed* insertion workload that combines timely and deferred insertions and an i/q ratio of 100. We used the same queries as in the previous experiments (see Section 3.4.1.3)with the small network and generated new queries with the same properties (i.e., same relative spatial and temporal extent) for the larger network.

We discuss the index scalability by showing directly the speedup of TRIFL* in comparison with the other tested methods, since TRIFL* has in general much better performance. Figure 3.13 shows the query time speedup of TRIFL* compared with the other tested methods, i.e., the ratio between the query time of each of the four competing methods (B^+-tree, FD-tree, LSM-tree and normal TRIFL) and the average query time of TRIFL*. A first observation (not shown in Figure 3.13) that applies to all methods is that the query cost is practically independent of the network size and only depends on the temporal data density. The query cost depends on the number of intersected partitions and the amount of data retrieved in each partition. Since in our tests the query size is relative to the network size, the number of intersected partitions by the queries is similar with the two networks. Therefore, the variation in the query cost is mainly determined by the variation of the temporal data density. The higher the density is, the larger the amount of scanned data in each partition is. The second observation is that TRIFL* has in general much better query scalability with larger data densities than the other competing methods. The explanation is that TRIFL* uses large granularity I/Os to process queries and the size of the block is proportional to the amount of scanned data in a partition. This allows TRIFL* to keep the query cost low even when the data density increases. Among the competitors, the FD-tree is the only method that also uses large granularity I/Os to process queries and has similar query scalability with TRIFL*. However, the query cost with the FD-tree is about twice the query cost with TRIFL*.

Figure 3.14 shows the insert time speedup of TRIFL* compared with the other methods, i.e., the ratio between the insert time of the other methods and the average insert time

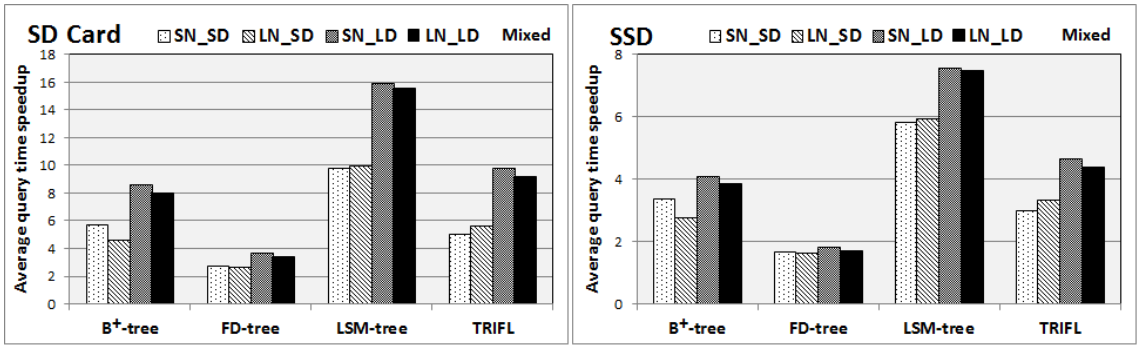

Figure 3.13: Average speedup of the query time of TRIFL* compared with the other methods with different network sizes and data densities

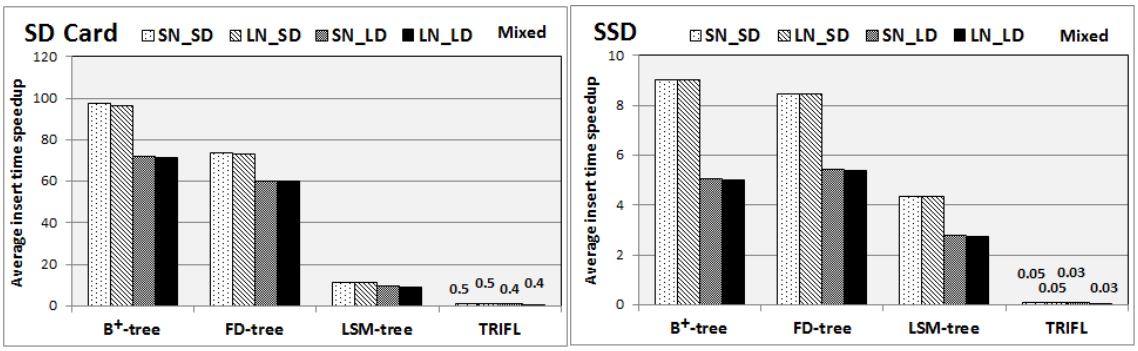

Figure 3.14: Average speedup of the insertion time of TRIFL* compared with the other methods with different network sizes and data densities

of TRIFL*. As with the queries, the map size does not have any influence on the insert performance of any method (not shown in Figure 3.14). Only the temporal data density, i.e., the number of trajectory units inserted over a time interval, affects the insert performance. The insert performance remains the same with the larger dataset for the B^+-tree and TRIFL and it decreases for FD-tree,LSM-tree and TRIFL*. The explanation is that FD-tree,LSM-tree and TRIFL* periodically (re)merge the indexed data, and the cost of the merge increases with the index size. Nevertheless, the increased query performance compensates the merge cost for TRIFL* and the FD-tree. Also, the increase of the merge cost is marginally larger for TRIFL* than for the FD-tree. Regardless the decrease of the insertion speedup, TRIFL* still has much better insert performance, while the query scalability is slightly better than in FD-tree. Besides, with increasingly sized datasets, the merge cost in TRIFL* is bounded by the temporal partitioning as discussed in Section 3.4.6.2.

Figure 3.15 shows the throughput speedup of TRIFL* compared with the other methods, i.e., the ratio between the average throughput of the other methods and the average throughput of TRIFL*. The throughput results sum up the query and the insertion costs presented above. On the SD card, the speedup of TRIFL* compared with the LSM-tree

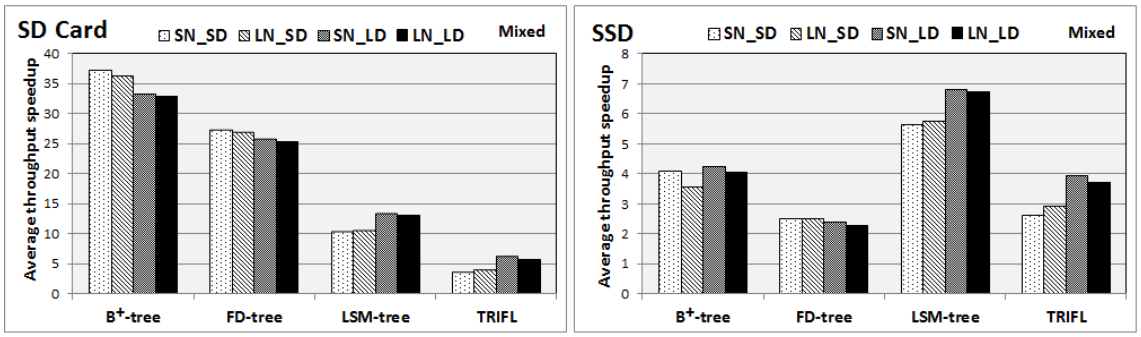

Figure 3.15: Average speedup of the throughput of TRIFL* compared with the other methods with different network sizes and data densities

and TRIFL slightly increases with the larger dataset, while it slightly decreases compared with the B^+-tree and the FD-tree. On the SSD, the speedup of TRIFL* compared with the LSM-tree and TRIFL increases with the larger dataset, it remains approximately constant compared with the B^+-tree, and it slightly decreases compared with the FD-tree. Overall, TRIFL* exhibits good scalability with both the map size and the data density, and it offers a significantly larger throughput compared with the other methods.

3.4.5 Index performance with various cache sizes

Indexing trajectory flows in flash raises conflicting challenges. On the one hand, the high insertion rate and the potentially high RW cost in flash require buffering many updates in cache before committing them in batch for increased efficiency. On the other hand, to effectively process queries the data has to be partitioned (see Section 3.4.6.1), which leads to a large number of indexes that are concurrently accessed. In this section, we analyze the robustness of the index performance with smaller cache sizes. We use a workload with an i/q ratio of 100 and mixed insertions and vary the cache size from 256KB to 10MB. For each cache size, we compute the number of partitions of the index with the cost model presented in Section 3.3.2.1. Typically, for a given cache size the insert performance of TRIFL* largely depends on the number of partitions (see Figure 3.6 in Section 3.2.4). If the number of partitions is large, the write amplification becomes greater than 0, which greatly increases the insert cost.

Figure 3.16 indicates the measured throughput for the tested methods for a given cache size. The number of partitions corresponding to each cache size is also indicated in Figure 3.16. Overall, TRIFL* performs significantly better than the other tested methods. Without surprise, the efficiency of all the methods decreases with the reduction of the cache size. However, TRIFL* manages to be more robust to the cache size reduction. The explanation is that the reduction of the cache size leads to a lower number of index partitions. This decreases the spatial filtering efficiency of the indexes which in turn in-

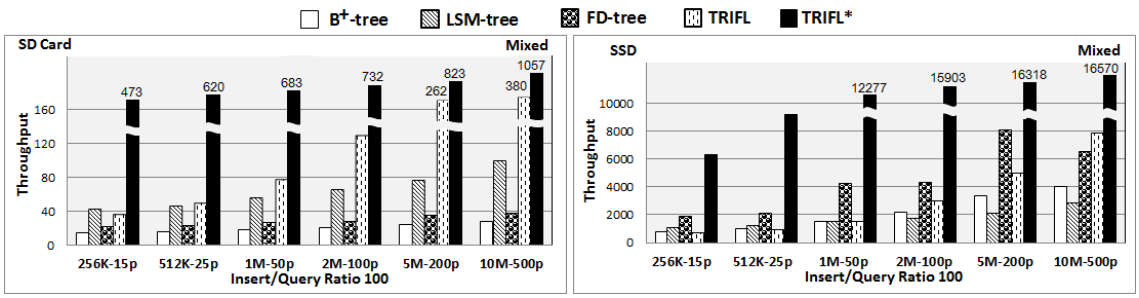

Figure 3.16: Throughput as a function of the cache size

creases the query processing cost. TRIFL* compensates the loss of the spatial filtering by employing larger granularity I/Os in the temporal filtering step. For instance, the throughput of TRIFL* is practically the same with cache sizes from 2MB to 10MB with the SSD device. The FD-tree also uses large granularity I/Os and has a good performance with the SSD storage. However, the poor insert performance makes that the FD-tree is much less efficient overall than TRIFL*.

3.4.6 Importance of spatio-temporal partitioning in TRIFL

3.4.6.1 Spatial partitioning

All the competing methods (i.e., the FD-tree, the LSM-tree and the B^+-tree) suffer from poor insert performance with a few exceptions (i.e., the LSM-tree works well on the SD card, and the FD-tree and the B^+-tree work well with timely insertions). The poor insert performance significantly degrades the global throughput of these methods especially for insert-oriented workloads. Interestingly, the FD-tree and the LSM-tree are particularly tuned to have good insert performance. However, these methods require a minimum amount of cache to attain their optimal performance (e.g., 256 pages for the FD-tree). Taken in the context of a trajectory index, the cache memory requirement explodes because of the spatial partitioning which incurs a large number of local indexes.

Therefore, given the conflicting requirements between cache and multiple indexes, one idea could be to give up the index partitioning and create a single global index on the time dimension of the dataflow. The obvious drawback is that the index search efficiency decreases since there is no more spatial filtering. Hopefully, this should be overcompensated by the gain in the insert efficiency. Figure 3.17 shows the relative throughput of the tested methods without spatial partitioning on SD card and SSD with the mixed insertion workload and an i/q ratio of 100 and 10MB of cache. We take the throughput of the B^+-tree as a base line for the comparison. We also add the throughput of the partitioned TRIFL* (TRIFL*_multi) to point out the importance of the spatial partitioning for trajectory indexing.

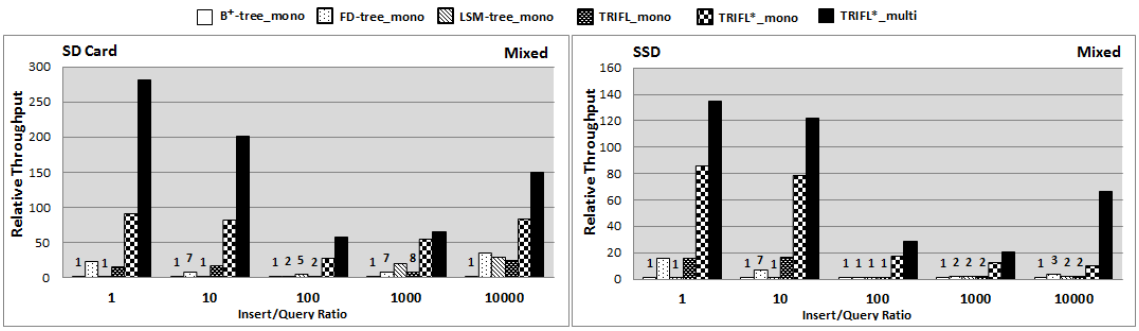

Figure 3.17: Relative throughput with mono-partition indexes versus TRIFL*

Globally, the throughput of TRIFL*_multi is much larger than the mono partition indexes. Actually, the gap between TRIFL*_multi and the competing methods increases in general (except for the largest i/q ratio workload) showing that the spatial partitioning is more important in the overall performance than the cache size. TRIFL*_mono is the only method that has comparable performance with TRIFL*_multi. The large granularity I/Os of TRIFL* manage to partially compensate the loss of the spatial filtering in TRIFL*_mono. However, the query performance is significantly better with partitioning even for TRIFL*.

3.4.6.2 Temporal partitioning

Trajectory flows are potentially unbounded in time. In Section 3.1.3.1 we discussed the importance of having a temporal index partitioning in the TRIFL framework. We also proposed a temporal partitioning model in Section 3.3.2.2. In this section we evaluate the temporal partitioning in TRIFL. To this end, we compare the performance of a multi-component TRIFL* with a single-component TRIFL* with a large trajectory flow containing up to 17,5 million data units arriving in a window of 5000 time units. The temporal partitioned TRIFL* creates a new index component after each 1000 time units. Figure 3.18 details the throughput of the multi-component TRIFL* and the single-component TRIFL* on the SD card and the SSD. For this test, we used an i/q ratio of 100, mixed insertions and a 10MB cache size.

With both flash devices, the multi-component TRIFL* has nearly stable throughput while the throughput of the single-component TRIFL decreases significantly with the increase of the indexed data volume. The reason is that TRIFL* with temporal partitioning limits the index reorganization cost, while the merge cost of TRIFL* without temporal partitioning linearly increases over time. On the other hand, the query cost in the multi-component TRIFL* is higher than in the single-component TRIFL* since some of the queries require searching in two index components. However, the increase of the search cost is small, i.e., less than 5% in our setting, and is not significant in the overall index performance.

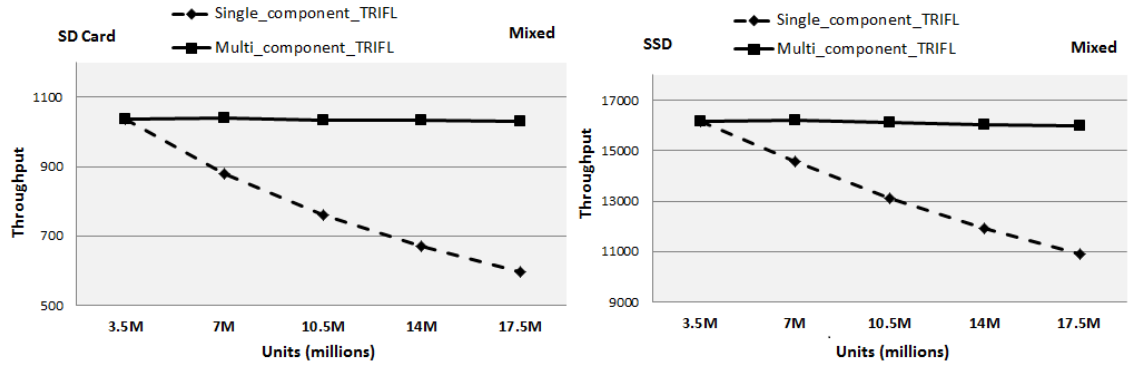

Figure 3.18: Multi-component TRIFL* versus single-component TRIFL*

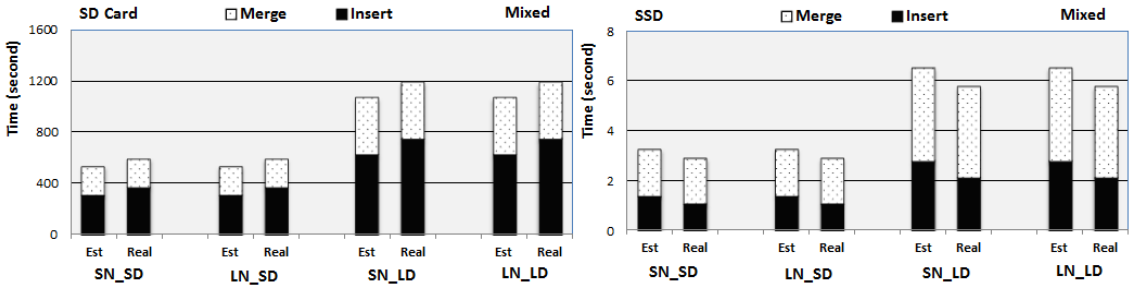

Figure 3.19: Insert and merge cost model evaluation

3.4.7 Cost model evaluation

In Section 3.3, we presented a cost model that allows TRIFL* to tune its structure. The cost model proposes estimations for insert, merge and query costs. These estimations are then used to self-tune the index structure of TRIFL*. In particular, the online self-tuning algorithm (Algorithm 1 in Section 3.3.1) compares the merge cost with the query cost to automatically trigger a merge operation in TRIFL*. Moreover, the spatial partitioning algorithm (Algorithm 2 in Section 3.3.2) uses the estimated workload cost (i.e., the sum of the query cost and the insertion cost) to determine the best spatial partitioning for a new TRIFL component index. Given the importance of the cost model in TRIFL, we evaluate in this section the accuracy of the proposed model. To this end, we compare the real costs of insertions, merges and queries with the estimated costs, as computed by the cost model. We test the cost model in four different settings, i.e., the same settings used in Section 3.4.4, since these tests vary the road network, the insert workload and the query workload.

In Figure 3.19, we compare the real costs of the insertions and merges with the estimated values as computed by Formulas 3.3.6 and 3.3.12 with different index configurations and workloads. The results indicate that both the insertion cost and the merge cost are estimated accurately by the cost model. With the SD card, the estimated insertion cost is on average 84% accurate in the four experiments, while the estimated merge cost is 99%

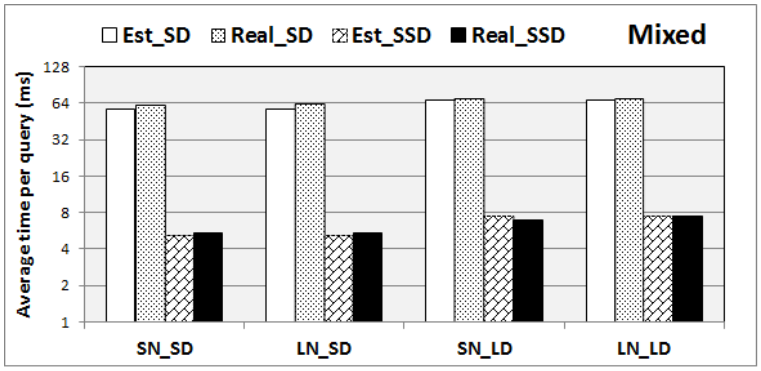

Figure 3.20: Query cost model evaluation

accurate. Similarly, with the SSD, the estimated insertion cost is on average 69% accurate, while the estimated merge cost is 97% accurate with the SSD. Mainly, the estimation error of the insert cost is produced by the write-amplification factor, i.e., wa_{cache} (see Section 3.3.2.1). wa_{cache} indicates the probability for a data unit in the hot data buffer to be temporarily written in flash before being flushed in the stable part of the index and its value is also estimated based on the cache size and the number of index partitions (see Figure 3.6). Although the estimated values for the wa_{cache} are fairly accurate, even small deviations can introduce significant errors in the insert cost since the wa_{cache} is multiplied with the RR and RW costs (see Equation 3.3.12), which are typically much larger than the SW(B) cost (see Figure 3.7).

Figure 3.20 shows the average estimated time as computed by Formulas 3.3.1 to 3.3.4 and the average real time of queries in the four experiments and with the two flash devices. Globally, the query costs are estimated accurately by the cost model. With the SD card, the estimated query cost is on average 94% accurate in the four experiments, while the estimated query cost is 96% accurate with the SSD. The real query costs are in general slightly higher than the estimated costs. One explanation is that TRIFL* uses large granularity I/Os in the clustered index area, which may slightly increase the number of retrieved data pages in comparison with the exact number of data pages required by a query. Also, we obtained similar accuracy levels of the cost model with the HDD storage but for conciseness we omit the detailed results.

3.4.8 Summary of the experimental results

We experimentally evaluated in Section 3.4 the proposed method to index trajectory flows. In this context, the index workload consists in both queries and massive data insertions. We discussed in Section 3.4.2 the insert and query performance of TRIFL* in comparison to three state of the art index methods. The I/O insert performance of TRIFL* proved excellent and robust with the type of insertions (i.e., timely or deferred). This is due

to the way TRIFL* buffers the updates (which amortizes the cost of individual updates) and appends them to the index structure (which precludes costly random writes). At the same time, TRIFL* also offers good I/O query performance compared with the other tested methods. TRIFL* manages to maintain high query efficiency through periodic index reorganizations. Moreover, the performance difference between TRIFL* and the competing methods becomes even more prominent when analyzing the types of the I/Os. Most of the I/Os engendered by insertions and queries in TRIFL* are large granularity I/Os, which are cost-effective in flash and further improve the index efficiency.

We then evaluated in Section 3.4.3 the global index performance, i.e., the index throughput, using workloads with various insert/query ratios. Without surprise, based on the very good insert and query performance and its self-tuning mechanism, TRIFL* offers much better throughput than the competing methods for both insert-oriented and query-oriented workloads on all the tested storage devices. The throughput of TRIFL* is in general one order of magnitude higher than the throughput of the other tested methods. In fact, compared with TRIFL* these methods can offer competitive performance only for particular workloads and storage devices. For example, the LSM-tree has good performance with insert-oriented workloads on the SD card, while the FD-tree works well with query-oriented workloads on SSD.

The large number of partitions of a trajectory index being concomitantly accessed by updates and queries may require an amount of cache proportional to the number of indexes, which may not be always available. In Section 3.4.5 we analyzed the robustness of the index performance with the cache size. Thanks to its cost model that adapts the spatial partitioning to the workload and the cache size, TRIFL* shows robust performance with a throughput that remains competitive even with small cache sizes.

We confirmed the performance scalability of TRIFL* with both large network size and high data density in comparison with the other methods in Section 3.4.4. The results show that only the temporal data density affects the insert and query performance, while the network size has marginal impact on the performance of the tested methods.

In the first part of Section 3.4.6, we experimentally evaluated the importance of the spatial partitioning in the overall index performance. The experimental results show that the spatial partitioning is necessary since it improves the performance of all the tested methods even with workloads containing a relative small number of queries. Then, we evaluated in the second part of Section 3.4.6 the importance of the proposed temporal partitioning in TRIFL*. By periodically creating new index components over time, TRIFL* manages to maintain a near-optimal index throughput regardless of the continuous increase in the indexed data volume.

Finally, we validated the proposed cost model used to estimate the insert, merge and query costs in Section 3.4.7. The obtained empirical results are very close to the values computed

with the cost model. This allows TRIFL* to adapt smoothly its structure to any changes in the workload.

3.4.9 Experimental evaluation with a real dataset

Although real trajectory datasets are becoming more and more available, we preferred to evaluate the proposed index framework with synthetic datasets. The reason is that available real datasets are still not representative enough in space and time. That is, real datasets typically contain data from a limited number of users with a high degree of spatial overlapping among trajectories and a low degree of temporal concurrency between trajectories. This explains the fact that most of the works on trajectory indexing still prefer synthetic datasets to real datasets since, ironically, the synthetic datasets are more representative of real use-cases (i.e., trajectories cover the entire space and present a high degree of concurrency between the MOs) and are also parametrically adjustable.

Nevertheless, we evaluate in this section the proposed index in comparison with the chosen alternative methods with a real dataset. The dataset was collected by Microsoft Research in the GeoLife project [124] and is available online (research.microsoft.com). This dataset contains trajectories from 182 users collected over a period of 5 years from 04/2007 to 08/2012. Geolife data was recorded in many cities located in China, USA and Europe. However, the majority of the data was created in Beijing, China. In our tests, we extracted data from Geolife dataset so that it can be comparable with the above presented experiments. We selected all the trajectories recorded in the Beijing region within a period of 3 years from 08/2009 to 08/2012, which represents a dataset containing 4.5 million trajectory units. Although the amount of units is similar with our synthetic datasets, there are two major differences between the real dataset and the synthetic datasets. First, given the low number of tracked users, the trajectories in the real dataset are sparse in the time dimension. Second, the trajectories correspond to different kinds of movement (e.g., vehicles, bicycles, walks) and are represented in the two-dimensional space. Hence, we kept all the kinds of trajectories and indexed the data using the free movement model. To this end, we used an adaptive grid (see Section 3.1.2) to partition the 2D space for the real dataset instead of a graph partitioning as with the synthetic dataset. However, the resulted index structure has 400 partitions, which is similar to the number of partitions used with the synthetic dataset.

Based on this real dataset, we generated an insertion workload containing only *timely* insertions to simulate a continuous real-time tracking of the users. We also generated a query workload with queries having similar properties as in the synthetic datasets. However, since the real dataset contains free trajectories, we only consider 2D and NN queries and no path queries with this dataset. As in Section 3.4, we generated workloads having insert/query ratios varying from 1 to 10000, 1 corresponding to a query intensive workload

and 10000 corresponding to an insert intensive workload. Also, the cache size was fixed to the default size of 10MB.

Figure 3.21 presents the insertion time (left) and the query time (right) with different indexing methods on the SD card and the SSD. Regarding the insert performance, TRIFL and TRIFL* show in general (much) lower insertion time than their competitors on both the SD card and the SSD. The B^+-tree has also good insert performance on the SSD due to the workload cache locality (i.e., timely insertions) and the small cost of random writes on the SSD. However, the insert performance of the B^+-tree is poor on the SD card. Also, the FD-tree and the LSM-tree have large insertion costs in comparison with TRIFL and TRIFL*.

Regarding the query costs, TRIFL* offers by far the best performance on both storage devices. TRIFL and FD-tree come next having better query execution time than the B^+-tree and the LSM-tree. The good query performance of TRIFL* is explained by the use of large granularity I/Os and by the high degree of the node filling factor of the B^{AO+}-trees. The FD-tree also uses large granularity I/Os to process the queries but to a smaller extent than TRIFL*. Besides, the FD-tree and the B^+-tree are penalized by small node filling factor, which is particularly pronounced with timely insertions. TRIFL also has a very good node filing factor as TRIFL*, but the lack of large granularity IOs in the query processing makes it less efficient than TRIFL*. Finally,the LSM-tree exhibitsthe lowest query performance on both the SDcard and the SSD because it has to search in multiple components in each partition index.

Overall, TRIFL proves to have the best insert performance, but not the best query performance. By contrast, TRIFL* trades insert performance to obtain significant gains in query performance.

Figure 3.22 shows the relative throughput of the tested methods with the real dataset at different i/q ratios. For better readability of these graphs, we take the throughput of the B^+-tree as a base line, and show for each method the ratio between its throughput and the throughput of the B^+-tree. The first general observation is that the throughput of each method increases with the increase of the i/q ratio. This is normal since an insert operation is much cheaper than a query. The second general observation is that TRIFL* offers the best throughput no matter the values of the testing parameters. This is as expected since TRIFL* has both excellent query and insert performance as discussed above. TRIFL delivers comparable throughput values with TRIFL* only for workloads having large i/q ratios, i.e., for insert-oriented workloads. This indicates the importance of the tuning process to achieve good performance in general with various i/q ratios. The FD-tree has throughput values comparable to TRIFL* only on the SSD device and low i/q ratios. By contrast, the LSM-tree offers the worst performance on the SSD and on the SD card with low i/q ratios. The performance of the B^+-tree is comparable to that of the

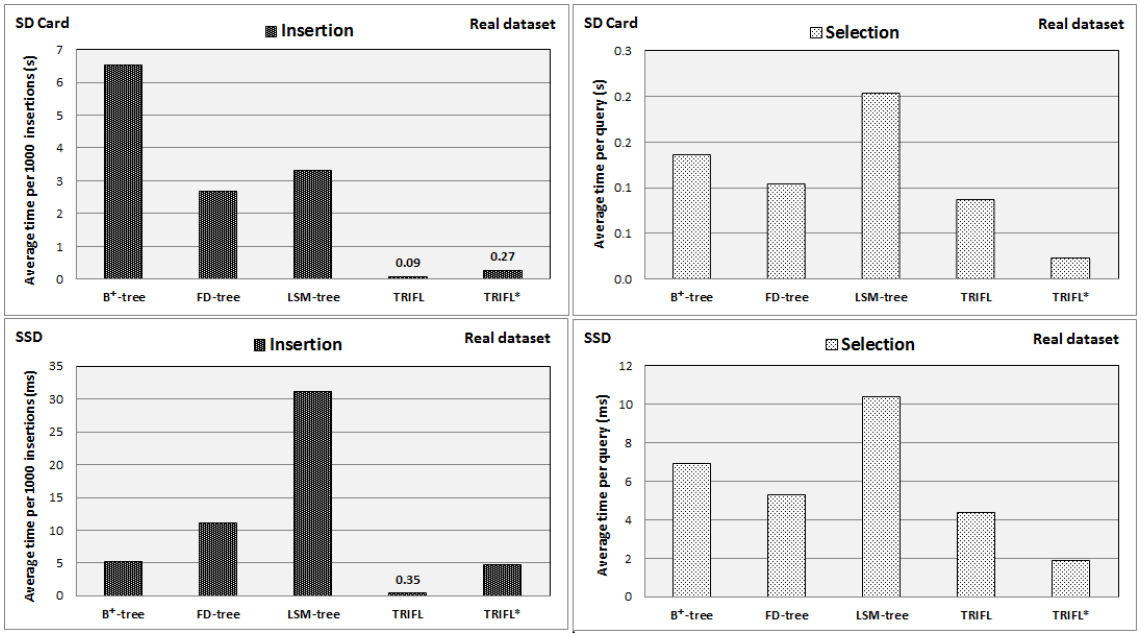

Figure 3.21: Insertion (left) and query (right) time for the tested methods on the SD card and SSD at i/q ratio of 1000

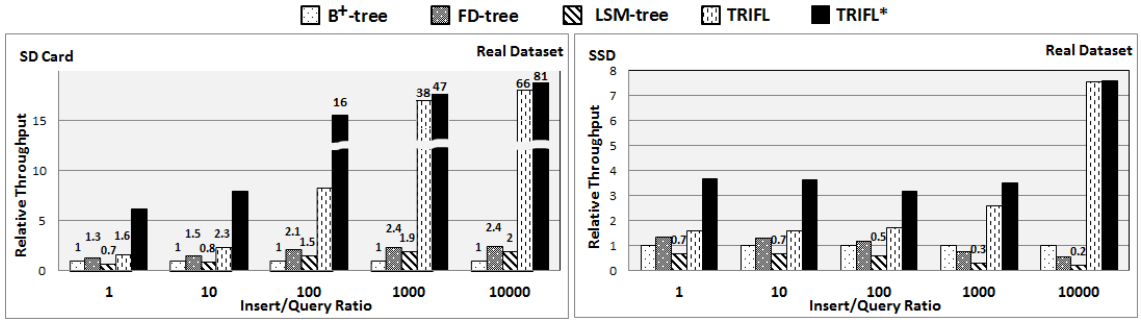

Figure 3.22: Relative throughput for the tested methods on the SD card and the SSD at different i/q ratios and timely insertions

FD-tree and the LSM-tree. The B^+-tree manages to be more efficient than the FD-tree and the LSM-tree on the SSD and for the highest i/q ratio.

Globally, the results with the real dataset confirm the performance trends with the synthetic dataset and the timely insertions (see Sections 3.4.2 and 3.4.3). The main difference between the two sets of results is that the absolute throughput values of all the tested methods are larger with the real dataset. The explanation is that the trajectories in the real dataset are much sparser in both space and time, which significantly increases the query selectivity and therefore, decreases the query costs for all the tested methods. At the same time, the insert performance of all the methods is generally the same with the real and the synthetic datasets.

3.5 Discussion

In the light of the above experimental results, we analyze in this section the reasons that make TRIFL framework appropriate for any kind of flash storage and also more efficient at indexing trajectory flows in flash than the other tested methods.

TRIFL is designed to reconcile the requirements of trajectory indexing with the characteristics of flash storage (see Section 3.2.1). The existing access methods either consider the specificity of trajectory indexing or the characteristics of flash storage. Since these requirements can be conflicting, they can only be addressed efficiently if considered together. To this end, TRIFL uses adapted low level storage structures, i.e., B^{AO+}-trees and TIIs. These structures are specifically devised to conform to all the design principles listed in Section 3.2.1. In particular, with respect to the flash storage design principles, the main role of these structures is to transform the random writes engendered by the updates into sequential writes as described in Section 3.2.2. Therefore, TRIFL deliberately trades query performance in order to avoid the costly random writes in flash and improve the insert performance (conform to $P1$). Consequently, supporting large granularity writes becomes possible since all the updates in TRIFL are initially appended (logged) to the index structure. Thus, the obtained benefit is twofold since first, the RWs are transformed in SWs(conform to $P3$), and second, the obtained SWs are committed with large I/Os(conform to $P4$). In addition, supporting the large granularity reads is the consequence of an index reorganization operation, which is the merge between the B^{AO+}-trees and the TIIs. The merge produces clustered partitions, which in turn allows retrieving data in large, consecutive blocks and improves the query performance(conform to $P5$). Finally, given their small hot-data buffer requirements, the B^{AO+}-tree and TII structures allow for better buffer cache utilization(conform to $P2$), which in the context of trajectory indexing significantly improves the index performance, again, by reducing the costly random reads and writes. In conclusion, the proposed B^{AO+}-tree and TII structures constitute the foundation of TRIFL and play an essential role in the index performance since these structures permit to transform the random writes into sequential writes, reduce the index cache footprint and open the way for employing large granularity I/Os.

The experimental results confirm that both the reduction of the random writes and the large granularity I/Os play an important role in the index performance. The B^{+}-tree generates many random writes and cannot employ large I/Os in our framework (see Figure 3.9 and 3.10). Compared with TRIFL, the performance gap is the largest (see Figure 3.12) with the deferred, insert oriented workloads (since more random I/Os are generated in this setting). This also applies to the FD-tree, which has similar insert performance (i.e., many random writes) to the B^{+}-tree. However, the FD-tree has much better query performance than the B^{+}-tree since it uses large I/Os in the query processing. Hence, the FD-tree offers much better throughput with query oriented workloads on both the SD card and

the SSD. In comparison with the FD-tree, the LSM-tree has the opposite behavior. The LSM-tree has very good insert performance since it uses large I/Os and does not generate any random write. However, the LSM-tree has very poor query performance. Therefore, the LSM-tree offers its best performance with the insert oriented workloads since the insert performance is the most relevant in this case.

Overall, all these methods are less efficient than TRIFL in both reducing the number of random writes and in using large granularity I/Os. The number of random writes is extremely low with TRIFL, while most of the I/Os are large granularity I/Os (see Figure 3.9 and 3.10). Therefore, given a storage device, the higher the random write cost is and the higher the throughput with large granularity I/Os is, the better the relative performance of TRIFL will be in comparison to the other tested methods. Also, for all devices, the speedup of TRIFL is much more pronounced with the deferred insert-oriented workload since more random writes are generated in this case by the FD-tree and the B^+-tree. Moreover, TRIFL's performance is more robust to cache size variations than the competing methods (see Figure 3.16).

3.6 Conclusion

The advent of NAND flash storage, which is becoming the preferred storage medium in mobile devices, sensors, personal computers, and high-end servers, raises new challenges regarding the data storage and indexing. In this thesis we addressed the problem of indexing trajectory data flows in flash storage. To this end, we introduced first a global framework called TRIFL for indexing range and NN queries over free and constrained trajectory flows. Then, we endowed TRIFL with a new indexing structure designed for flash storage. The indexing structure of TRIFL is generic with respect to the main characteristics of flash memory such as the asymmetric-read-write performance and the fast large-granularity I/Os, which makes TRIFL adapted for both basic storage devices such as the SD cards and powerful devices such as the SSDs. Also, TRIFL is supplied with an online self-tuning algorithm that allows it to be adaptable with respect to the performance specifications of the flash storage and the index workload. Moreover, TRIFL achieves good performance with relatively low memory requirements, which makes the index appropriate for many application scenarios. The experimental evaluation of TRIFL in comparison with three representative index structures shows that TRIFL is very efficient and generic, since it offers in general a much larger throughput than its competitors on an SD card, on an SSD and even on an HDD.

TRIFL is designed for flash storage but the experimental results showed that TRIFL also has excellent performance on HDD. The reasons are twofold. First, part of the design principles of TRIFL concern the characteristics of trajectory data and are orthogonal to the

type of employed storage. Second, magnetic disk storage shares common features with flash storage, e.g., increased performance of large granularity I/Os (although not due to the same technical reasons; on HDD a large granularity I/O corresponds to number of sequential I/Os, which minimizes the arm movements and hence increases the performance). At the same time, magnetic disks have other characteristics that are not shared by flash storage such as the symmetric read-write performance. As future work we plan to investigate if the TRIFL framework can be further improved for magnetic disks and therefore be generalized for both flash and magnetic disk storage.

Part II

PAMPAS: Privacy-Aware Mobile Participatory Sensing Using Secure Probes

Chapter 4

State of the Art

In the part II of this thesis, we address the problem related to privacy in personal location data of participatory sensing systems that is carried at two Chapters (i.e., this Chapter and the next one). This Chapter introduces the context as well as the motivations of this topic. At first we introduce the motivation and the requirements in the Section 4.1. Then, we present the related work in the Section 4.2.

4.1 General context

There is an increasing interest in mobile participatory sensing for urban monitoring, which appears to be a better alternative to traditional infrastructure-based sensing to cope with the high installation and maintenance costs, as well as the coverage limitation. Many projects have been conducted recently around the world - or are still ongoing - in the area of environmental participatory sensing, among which Citi-Sense in Oslo, CamMobSens in Cambridge, MetroSense in Dartmouth, and OpenSense in Switzerland, to cite but a few[1]. For a recent review, refer to [85]. Also, many applications that exploit the sensing features of smart-phones are already available. Examples include community based traffic monitoring (e.g., INRIX, Waze or Navigon[2]), or noise mapping [28]. In addition, the emerging lightweight low-cost sensors are changing the paradigm of environmental monitoring, according to the US Environmental Protection Agency, which proposes a toolbox[3] for citizen sensing of air quality.

In these scenarios, the citizens act as mobile probes and contribute to spatial aggregate statistics, which in turn, benefit to the whole community, e.g. in dynamic traffic navigation or air quality mapping and alerts. Various statistics are of interest: basic count and density, average of reported measures by location and time, or more complex geo-statistical

[1]For a recent review, refer to the European COST Action EuNetAir: http://www.eunetair.it
[2]INRIX: http://inrix.com, Waze: http://www.waze.com and Navigon: http://navigon.com
[3]http://www.epa.gov/heasd/airsensortoolbox/

operations such as spatial interpolation [74]. Unfortunately, most current mobile participatory sensing systems (MPSS) require users to reveal their locations to trusted monitoring servers. This raises serious privacy concerns and prevents a wide adoption of the system since, by knowing the user location traces, an attacker can easily identify the participants and infer their personal habits, activities, health-related information or any other sensitive personal data [47], [26]. Several works consider the MPSS problem such as [18], [27], [28], [92], [110], [81]. However, most approaches require trusting a proxy server [27], [110], [81] while others are too costly [18], [28], or sacrifice the accuracy for privacy [92].

Hence, providing a high-quality MPSS, while protecting the users' privacy, is still a challenge. Recently, the emergence of personal secure devices has opened new perspectives in personal data protection. Be it a secure portable token [8], [111] communicating with the user smartphone or plugged inside it (e.g., Google Vault[4]), a tamper-resistant hardware security module securing the on-board computer of a vehicle [30], or the secure TrustZone CPU [12], [108] of the ARM cortex-A series equipping most of mobile devices today, all such secure devices offer tangible, hardware-based security guarantees. Their secure data processing capability can be employed in a distributed, privacy-by-design architecture, providing an alternative to the traditional server-centric architecture.

This part presents PAMPAS, a Privacy-Aware Mobile Participatory Sensing system for efficient mobile distributed query processing in the context of MPSS. PAMPAS provides high-level user privacy and satisfies the MPSS real-time constraints by ensuring acceptable communication and computation costs. These features give users strong incentives for participation, in addition to the benefits they get from MPSS applications [2], [39].

In PAMPAS, all participants have a mobile device enhanced with secure hardware (i.e., any of the types described above), called a secure probe (SP). SPs act as probes for the target phenomenon, perform distributed query processing, and share aggregates or other type of derived data with the users. The secure hardware prevents users from accessing other users' data during the distributed computation. Secure probes exchange data in encrypted form with help from a supporting server infrastructure (SSI). To provide real-time results, PAMPAS employs an efficient parallel location-based aggregation protocol which partitions the probes according to their geographic distribution. The construction and the maintenance of these partitions aim at reducing and balancing the workloads on worker SPs. To avoid leaking private information about these partitions to SSI, PAMPAS uses a privacy-aware protocol for adaptive spatial partitioning of secure probes.

We implemented and validated PAMPAS using representative secure hardware platforms. We used two applications for experiments, traffic and noise monitoring, with two synthetic mobility datasets representing a small and a medium-size city. Using these applications, we compared PAMPAS with a state-of-the-art secure aggregation protocol described in [111].

[4]http://www.cnet.com/news/googles-project-vault-is-a-security-chip-disguised-as-an-micro-sd-card/

The experimental results show that PAMPAS outperforms this protocol in terms of latency and scalability. Also, the security analysis demonstrates that PAMPAS avoids the leakage of any individual private data.

4.2 Related work

4.2.1 Traditional systems

Traditional system architectures used in MPSS such as [27], [100], [110] rely on a powerful centralized server to collect data from mobile participants, process it, and publish the result. The works in [27], [110] are some instances that use a server-centric architecture to monitor the users' environment conditions or traffic network. For example, [27] uses the sensors in the mobile devices of the volunteers to gather noise levels, which is sent to a central server for building the noise pollution map. In [110], the cars periodically send their locations to a centralized server that computes the global traffic density. Many commercial systems, such as INRIX, WAZE and Navigon, are based on similar solutions. This server-centric model is straightforward and easy to deploy, run or maintain. However, this basic approach also raises serious privacy concerns and prevents a wide adoption of the system since, by knowing the user locations, an attacker can easily identify the participants and infer their personal habits and activities [47], [26].

4.2.2 Privacy-aware approaches

Many recent works deal with the privacy issue in participatory sensing [22], and an important part focus on the traffic monitoring use-cases or environment monitoring systems [18], [48], [92] [23], [37],[103], [28], [60], [90]. They can be grouped in three classes, i.e., based on a server-centric architecture, on cryptographic protocols or secure hardware approaches.

4.2.2.1 Server-centric approaches

Virtual Trip Lines (VTLs). The model proposed in [48] deal with the privacy issue by distributing the traffic monitoring service implementation across several specialized servers and by providing a spatio-temporal cloaking of the users under the VTLs. This system was designed based on the ideas of virtual trip lines (VTLs) and associate cloaking techniques which means the grid of geographic makers in this systems indicates where the vehicles should provide their location updates. Indeed, VTLs is composed by a set of entities such as probe vehicles, ID proxy server, VTL generator and traffic monitoring service servers (VTL servers)(see Figure 4.1). Each component in VTL system takes responsibility of

All figures in this section are taken from the cited references with the permissions of the authors.

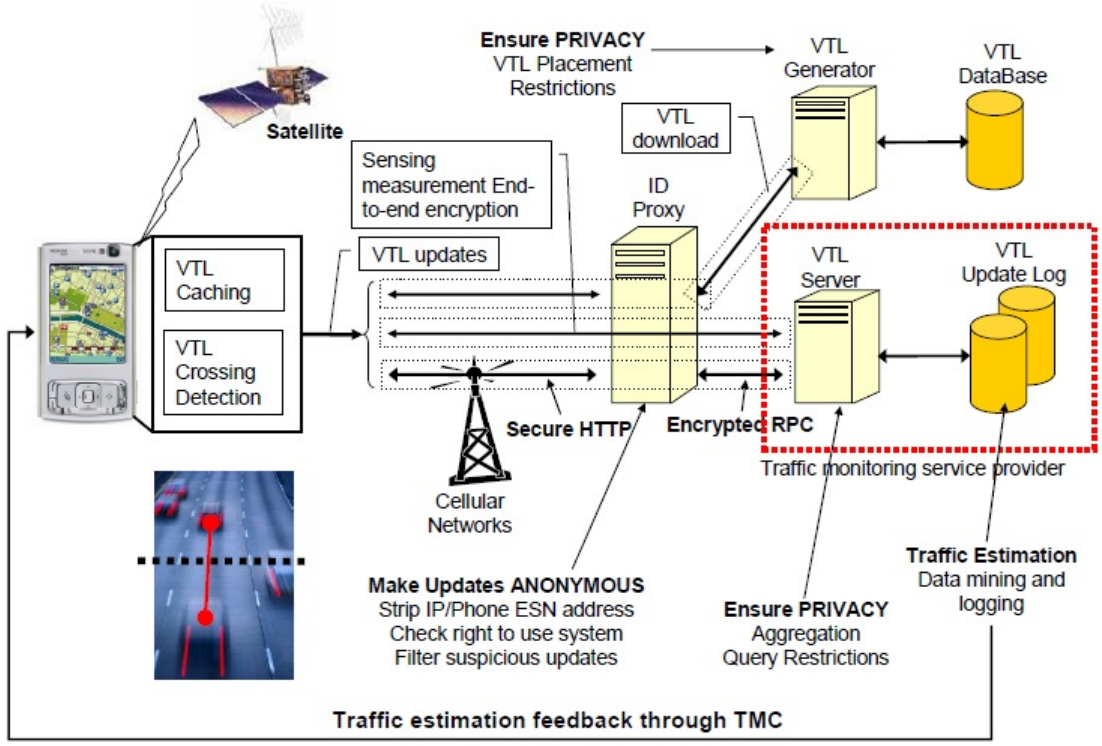

Figure 4.1: The architecture of VTL

specific task. For example, ID proxy servers authenticate each client to make sure all their received data are sent from authorized vehicles before encrypting and forwarding all these data to VTL servers via secure links. VTL generator determines the position of VTLs, store them and distribute to probes when received the requests. Lastly, VTL servers obtain all the encrypted updates from probes but don't have any knowledge about vehicle identities. VTL servers do the statistic based on these data. To this end, by distributing the services of system into many components VTLs system could protect again any single-component attacks. Although the attack of a single system component prevents linking the identity and location of the users, choosing privacy-insensitive locations for VTLs is tricky and limits the traffic information to a part of the road network. Also, the problem of multi-component attack (or collusion) remains, as well as the high cost of building such a complex system distributed over several components.

Spot Me if you can (SpotMe). SpotMe [92] is also trying to deal with the problem of traffic monitoring with the high privacy constraint. SpotMe [92] proposes a different approach consisting in mixing the real user's location with fake locations before posting them to a central server with the help of a small software (i.e., around 30KB) on user's side. Then, the server estimates the aggregated user locations by using the probability theory. By this way, the real locations of user are blurred among their fake locations and thus Spotme doesn't need to trust to servers. However, the traffic estimation errors can

be important (around 20%), while the number of observed spatial units cannot exceed a few hundreds. Furthermore, SpotMe involves higher communication costs because of the large number of fake locations, while linkability may still a problem for users that send many consecutive location updates, which limits the usability of this approach to sporadic updates.

AnonySense. Indeed, AnonySense using the participating users to collect the data and proposed a system composed by many entities such as mobile nodes (MNs), registration authority (RA), task services (TS), report service (RS), application (App), access point (AP) and mix network (MIX). Each component takes responsibility to a specific role. Since, any single malicious entity in this system can't infer much information from system. In other word, AnonySense protects quite well any kinds of single-component attack. However, the assumption of there is no collusion of many components in this system to invade user's privacy is unrealistic. Second, building such infrastructure is costly.

Poolview. Also designing a system based on participatory sensing, Poolview [37] tackles the problems of user-privacy by applying user's data perturbation algorithm together with noise-based solution. However, this system need to trade off between user's privacy and the accuracy of the system. The higher required privacy level, the higher error ratio. Furthermore, no matter what the algorithms applied for user's perturbation, there's no guarantee that user's data is always protected.

PriSense. Shi et al. [103] proposed a people-centric urban sensing system to tackle the problems of aggregation statistics. Using people-centric to obtain the sensed data, PriSense inherits all of the advantages of this architecture such as widespread, low cost, user-mobility and etc. In the architecture of Prisense, they classified the system into two main components: powerful aggregation severs (ASs) and participanting individuals (Nodes). The nodes check the queries from their connected AS via one-hop downlinks and contribute their data to this AS if they are relevant. To be more secure, PriSense also provides pairs of public/private keys between ASs and nodes. Each AS needs to compute its aggregate data before doing the final aggregation between ASs. Final result is sent back to the querier through a service provider. This system shows to be very secure but it still shares the common shortcoming of server-centric architecture, that is to trust a server or group of servers.

By employing a fully decentralized, user-centric architecture, PAMPAS avoids all the above listed problems, since all the computations are delegated to the participants. Moreover, the trust is enforced by using cheap but highly secure, tamper-resistant hardware at the user side (see Section 5.1 in Chapter 5).

4.2.2.2 Cryptographic approaches

Another way to protect the users' privacy is to use secure cryptographic protocols [18], [28], [60], [90]. Such solutions can offer formal guarantees on location privacy and accountability to protect against users trying to upload large amounts of fake samples. Typically, the cryptographic solutions are based on homomorphic encryption schemes allowing a central-server [60], [90] or the users [18] to aggregate the samples directly on the cyphertext. We present below some recent studies in this trend.

Haze. Haze in [18] introduced a general approach in order to generate traffic-statistics using user-centric architecture. Haze consists three phases: Setup, data upload and aggregation. The general ideas of Haze are to gather information from authentication users (in data upload period), encrypt these data by using the key generated in the setup period. Lastly, Aggregation process will be delegated to a list of selected authorized users. Once final aggregate results are obtained, it is published by the service provider. However, this approach also reveals some shortcoming. At first, there is no guarantee that the users are not malicious. Secondly, the results show that it's quite costly and thus it's hard to be applied in our context with the real-time constraints and continuous with a large number of participants.

NoiseTubePrime. Drosatos et al. [28] proposed a system called NoiseTubePrime in order to monitor the noise pollution using cloud computing technique and homomorphic encryption. The general architecture of NoiseTubePrime is shown in Figure 4.2. Concretely, participants in this system obtain data using their sensor integrated in their mobile devices. These sensed data have to be encrypted before delivering to NoiseTubePrime Agencies (i.e., which is built on top of cloud system architecture) in order to calculate the statistic results without descrypt these data using homomorphic encryption technique. This work shows to archive a high level of privacy as the computing entities don't have any information of participant while it also takes advantages of distributed computing resources via could computing systems. However, similar to any other homomorphic systems, NoiseTubePrime also shares some shortcomings that are presented later on.

PrivMobileSensing. Another protocol for mobile sensing that supports sum, min aggregates using the homomorphic encryption in combination with a keyed-hash message authentication code (HMAC) to guarantee that server can not infer anything accept the final aggregate statistics. This work [60] is quite similar to the work proposed in Popa et al. that is presented in the next paragraph, thus it shares the same drawbacks.

PrivStats. Popa et al. [90] guarantee an approach for aggregation statistics without leakage any user information. Also using homomorphic technique but Popa el al. still keeps server-centric architecture (see Figure 4.3). The authors designed the solution based on the combination between cryptographic and accountability protocol. This solution prevents any attack from both server's side and user's side to infer user's information or to bias the

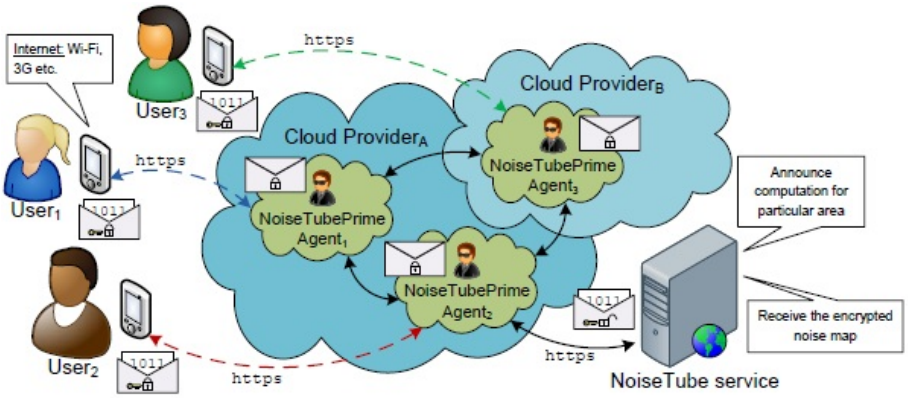

Figure 4.2: The general architecture of NoiseTubePrime

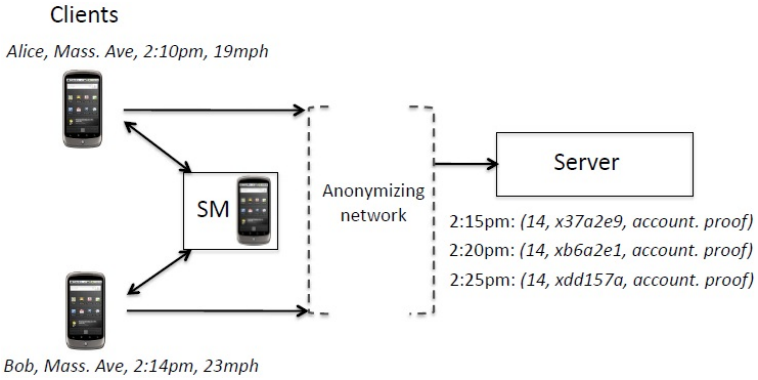

Figure 4.3: The architecture of PrivStats

aggregate result. The upload messages have to be encrypted and follow the accountability checking process before arrive to server. The server then does the aggregation based on the cipher texts of the uploads by using homomorphic encryption scheme.

However, the cryptographic methods have to face two major limitations. First, homomorphic encryption only allows the computation of basic aggregate functions (e.g., count, average, sum, standard deviation), while more advanced functions require fully homomorphic encryption schemes, which are not computational feasible today. Second, even with the basic aggregate functions, the cryptographic protocols can incur a (very) large computation and communication cost. Hence, the existing works typically limit the size of the monitored space (e.g., the number of roads) and the monitoring pace. Therefore, such solutions cannot meet the scalability and the real-time requirements of a MPSS at the same time, and are not generic w.r.t. the type of aggregate function. Moreover, depending on the encryption protocol, the accuracy of the aggregate result may also be impacted since only a (low) number of discrete range values can be computed with such protocols [18], [28], [60], [90].

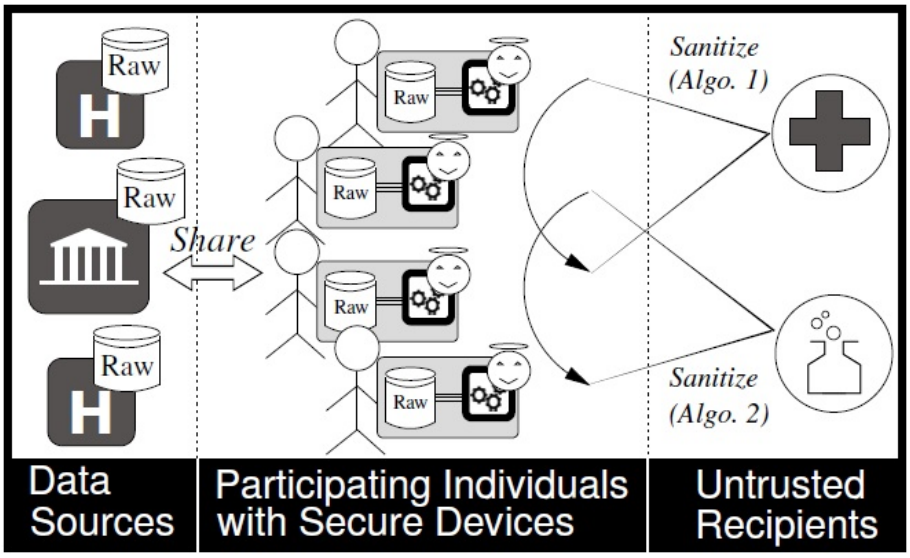

Figure 4.4: The protocol of METAP

4.2.3 Secure hardware approaches

Recent works have also proposed the use of secure hardware at the user-side [8], [111], which has been proposed independently from the context of participatory sensing. The trust in such a distributed architecture in which all computation is done by user devices arises from two sources:

- The decentralization, i.e., there is no central-server to be trusted or to be exposed to attacks having a large benefit/cost ratio.
- The (tamper-resistant) secure hardware at the user-side, which protects the devices against physical attacks (even from the device holder).

METAP. In [8], Allard et al. propose METAP, a generic privacy-preserving data publishing protocol in the context of an architecture composed of low power secure devices and a powerful but un-trusted server in order to release sanitized data to third parties. In particular, METAP proposed an user-centric architecture by using secure hardware devices (i.e., secure tokens) integrated on participating individuals (i.e., participants). The overview of this proposed approach is shown in Figure 4.4. Instead of using servers, METAP delegated the processing workload to a group of participating individuals having a secure device. However, this data publishing protocol does not consider the case of spatio-temporal sensed values and cannot be used in participatory sensing aggregation.

Secure protocol. To et al. [111] propose a similar architecture, but consider the problem of executing basic SQL queries over a distributed database without revealing any sensitive information to central servers. The proposed secure aggregation protocol is generic and

can be used to perform many types of computations within the user-centric architecture. Concretely, To et al. designed a basic and secure mechanisms in order to compute the queries and share data between participants and servers. The detail of this protocol is shown in Figure 4.5. The system is composed by two main entities Trusted Data Servers (TDSs) and Supporting Server Infrastructure (SSI) in order to execute the queries from the queriers. At first, the queriers could send their queries to SSI so that participating TDSs could download and compute the aggregate data. During the computation process, TDSs could interchange the computed data or intermediate result via SSI. Since, all of the calculation are executed in parallel on TDSs' side, the main roles of SSI in this model are only for communication and temporary storage. All of data exchanged or temporary stored in SSI has to be encrypted to avoid any effort of SSI to read or infer user's information. The final aggregate result is sent back to the querier through the SSI. The authors also assumed that TDSs only answer the authorized queries which means the final results will not reveal any user's sensitive information. In detail, the authors proposed four kinds of protocols (i.e., Generic query evaluation, secure aggregation, noise-based and Equi-depth histogram-based protocols) which are both mainly contain 3 main phases: collection phase, aggregation phase and filter phase (see Figure 4.5. First, the SII collects the related tuples sent by the TDSs in the collection period. Each tuple (or fake tuple) is encrypted using non-deterministic encryption so that the SSI cannot gain any knowledge from these updates. All the TDSs share a secret key to encrypt/decrypt the updates. Note that, although an TDS can decrypt the updates, a user is not allowed to access the decrypted data in her SP and that the tamper-resistant hardware protects the transiting data also from the user herself. At the end of the collection period, the SSI triggers the aggregation period. In this phase, only a (small) subset of TDSs are involved. The SSI shuffles the collected samples and partitions the sample set such that the number of updates in a partition can fit the RAM resources of an TDS (otherwise, the persistent Flash storage of the SP has to be used incurring a much higher computation cost). Then, each sample partition is sent to an TDS, which decrypts the data and computes a partial aggregate result for the received updates. The encrypted partial aggregate results are sent back to the SSI, which re-shuffles and re-partitions them. The process repeats until the final aggregate result is obtained. Note that at each iteration, the number of TDSs participating in the computation decreases with the remaining number of samples to be aggregated. In particular, in the last round, a single TDS is selected to produce the final aggregate result. Finally, the filtering period consists in eliminating the not related result and delivering the result to queries.

However, considered in our context, the generic protocol proposed in [111] incurs high computation and communication costs, especially with a large number of users or aggregate groups. There are three reasons for that: (i) since a worker probe can aggregate only samples belonging to the same aggregate groups, the computation can require many iterations; (ii) the number of aggregate groups processed by a probe in an iteration can

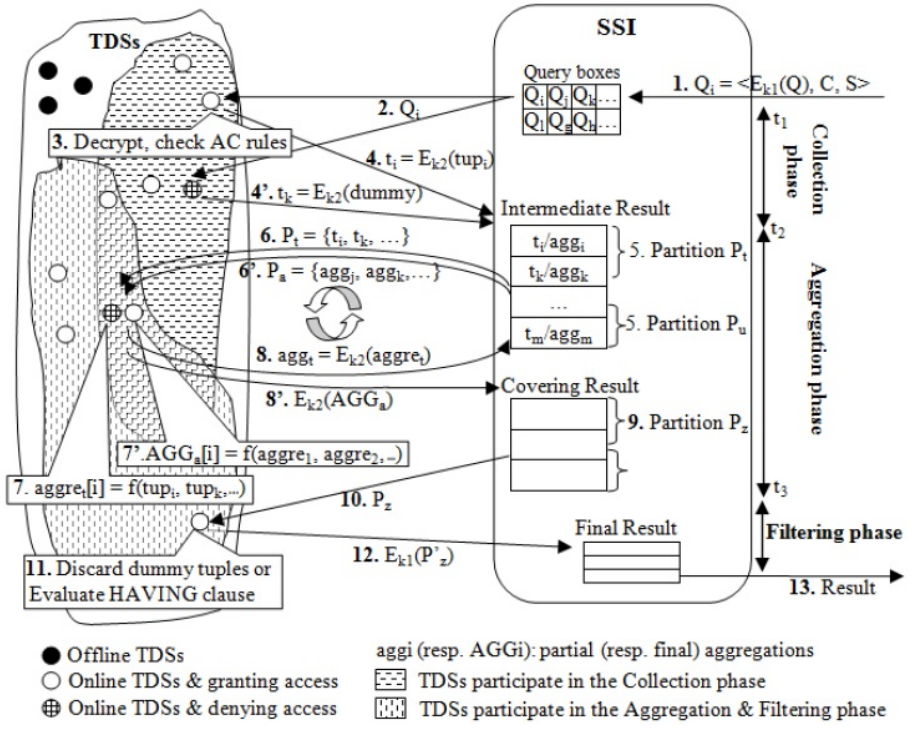

Figure 4.5: The querying protocol

exceed its available RAM size, which highly increases the computation cost; (iii) in case of holistic aggregate functions (e.g., median or top-k), all the samples in an aggregate group have to be known by the processing probe, thus the aggregation phase cannot be parallelized, which greatly affects the scalability of the system.

PAMPAS shares a similar architecture with the above mentioned works. However, the existing aggregation protocols are not adapted to the mobile participatory sensing computations. In particular, the high dynamicity of the participants, the continuous nature of the computation and the complexity of the aggregate statistics, make the existing protocols inefficient and not scalable to be used in our system.

A centralized solution based on secure hardware could also be devised using recent proposals to ensure shielded application execution over untrusted servers. For example, Haven [14] extends the hardware level protection features provided by the Intel SGX architecture from code snippets to the entire OS. But there are limitations: this solution slows down the computation substantially; the entire security architecture depends on the chip manufacturer's ability to protect the secret keys; programmers will miss certain features, such as process creation, that are not supported.

Chapter 5

PAMPAS

5.1 System overview

This section presents the system architecture of PAMPAS, the threat model in our system, and the data and computation model of the system. Based on these components, we derive the requirements for the PAMPAS protocols.

5.1.1 System architecture

PAMPAS relies on a hybrid architecture combining secure elements at the user side (secure probes – SP) and a supporting server infrastructure (SSI) that enables secure exchange of messages between the mobile users (see Figure 5.1). SPs and SSI jointly run two protocols to exchange sensed sample updates, continuously compute the spatially aggregated results, and periodically partition SPs according to their location. This architecture fully protects the users' privacy w.r.t. the SSI.

Compared to a purely decentralized peer-to-peer (P2P) architecture, this hybrid architecture has the salient advantage of not consuming any resources from the participants to maintain the P2P overlay, which is important given the low resources and availability of the user devices. In addition, it exchanges messages between SPs in $O(1)$ hops as opposed to the typical $O(logN)$ hops in P2P networks.

Secure Probe (SP). Each user holds a secure portable device, which can be implemented by any type of (tamper-resistant) secure devices (see Figure 5.2) flourishing today and described in Section 4.1 in Chapter 4. Whatever its commercial name and form factor, a secure device, called secure probe (SP) hereafter, embeds at least a secure microcontroller (MCU) (e.g., a smart card chip) for computation connected to a large NAND Flash memory (e.g., an SD card) for data storage. An SP plays three roles: (i) a mobile probe, (ii) a processing node, and (iii) a query issuer. Indeed, the SP sends encrypted samples

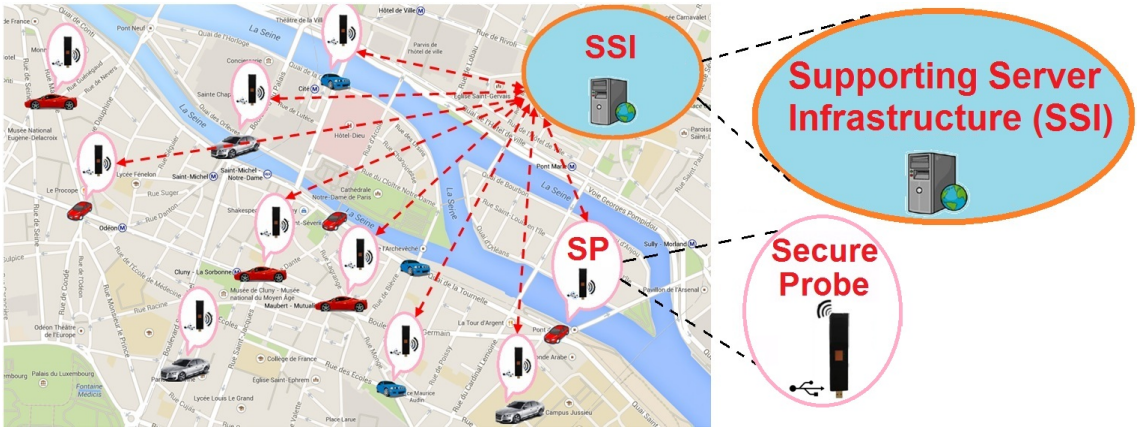

Figure 5.1: System architecture

Figure 5.2: Examples of secure tokens

(containing spatiotemporal sensed measures) to SSI, participates in the data aggregation, and receives the final results from other SPs with the help of SSI. Given their high-level of security, SPs are considered trusted in our system. However, this feature comes at a price. The MCU usually has a low power CPU and a tiny RAM (a few tens of KB). Also, in some cases, SPs may be battery powered. In addition, SPs have low availability since they can be connected/disconnected as required by the users. Therefore, all the computation and communication with the SPs have to be highly optimized.

Supporting Server Infrastructure (SSI). Different from the typical server-centric architecture, the SSI in our system acts only as a coordinator for exchanging messages between the SPs and for temporary storage purposes. Since the SSI is not trusted, all the temporary results stored in the SSI are encrypted using non-deterministic encryption.

In conclusion, the security and privacy in PAMPAS arise from the combination of secure hardware with a high degree of distribution of the architecture (i.e., all computations are executed by some of the SPs). The challenge is then to be able to continuously compute any type of aggregate functions in real-time in this user-centric architecture given the low resources of the SPs.

5.1.2 Threat model

The attackers in PAMPAS could be either users or the owners of SSI. Their goal is to collect private user information (e.g., location or sensing data). Using this private information,

attackers can determine the user identities and learn their activities and behaviors. Our goal is to ensure that users cannot read the raw data reported by other users. The SSI must not be able to read the user raw data, and, in addition, must not be able to infer personal user information from aggregated data even if it uses external background information.

Even though the users are untrusted, we assume that all the SPs are trusted, which is reasonable considering that the tamper-resistance of the MCU provides a high level of protection against physical and side-channel attacks, and in particular for the data residing in RAM since the RAM memory is located inside the MCU. We also assume that the hardware manufacturer is trusted and protects the secrets embedded in SPs. In addition, all the persistently stored data in the NAND Flash is cryptographically protected.

Furthermore, we assume an honest-but-curious threat model for SSI (i.e., the SSI obeys the protocol it is supposed to do, but may try to infer anything it can from the data or behaviors is sees). Considering a malicious SSI (i.e., the server tampers with the protocol is of little interest, since a malicious SSI can be easily detected (e.g., the SPs that aggregate the data verify if their own samples are present in the data sent by the server) leading to critical financial/legal consequences for the service.

Finally, we assume that the communication between SPs and SSI is anonymous, e.g., by using a proxy forwarder or an anonymization network (e.g., Tor) We assume such systems are able to hide the packet origin from an adversary, so that privacy cannot be compromised by a malicious server searching to recognize the origin of the uploaded messages. Let us emphasize that IP anonymity is not enough to protect the user privacy in MPSS because identity information could be determined from the location and sensing data.

5.1.3 Data and computation models

Data model. PAMPAS is designed to be generic with respect to the type of computation required by participatory sensing applications. In most cases, such applications require the aggregation of geo-localized and time-stamped sensed values collected by the sensing devices of the participants. Therefore, a participant's device periodically generates an update in the form $sample = (location,\ time,\ value)$, which is encrypted and sent to the SSI. PAMPAS does not impose any restriction on the generation frequency of samples, which may depend on the application sample generation policy. However, the system should be efficient and scalable for a large number of participants and a high generation frequency of samples. Also, the participants' privacy should be fully protected independent of the number and spatiotemporal distribution of the samples. Furthermore, PAMPAS considers two types of locations corresponding to the two typical types of movements of users: (i) free movements in the two-dimensional space, i.e., $location = (x,\ y)$; (ii) movements constrained by a transportation network (e.g., road or railroad network), i.e., $location = (rid,\ pos)$, where *rid* is the road identifier and *pos* is the relative position on

```
SELECT SpatialAggregate(value), [COUNT(*)]
FROM ParticipatorySensingStream
        [WINDOW x seconds SLIDE x seconds]
GROUP BY spatialUnit
HAVING predicateOnSpatialAggregate
```

Figure 5.3: Supported spatial-temporal aggregates in PAMPAS

the road. Finally, the *value* corresponds to the sensed measure (e.g., traffic speed, noise level, etc.).

Query model. Given a stream of *samples* and an aggregate function, PAMPAS produces a spatiotemporal aggregation of the sample stream such as the stream-SQL-like [51] query formulation in Figure 5.3. The aggregation is temporal since the result is computed *continuously* over time as long as it is required or whenever the number of participants exceeds some predefined threshold. In this way, the spatiotemporal evolution of the measure of interest is monitored over time. To this end, PAMPAS divides the stream using a sliding time window (see Figure 5.3) and computes an aggregate result based on all the samples generated in the time window. The final aggregation result is a spatial function representing the evolution in space of the observed measure in the respective time window. For instance, the result can be the noise heat-map in the covering area of a city or the average travel time in a road network. As with the duration of observation, we do not impose any restrictions regarding the extent of the observed space.

Spatial units. As shown in the above query, spatial aggregates are based on some discrete referential space, i.e., a finite set of spatial units. Without loss of generality, we consider two types of referential spaces corresponding to the two types of users' movements. In the case of free movement, we consider a uniform grid and each grid cell corresponds to a *spatial unit*. The size of the units is determined based on the application requirements, space size, number of participants, etc. In the case of constrained movement by a transportation network, we consider that a spatial unit corresponds to a network (road) segment, i.e., the network path connecting two adjacent network nodes (e.g., the road segment between two intersections). In both cases, the number of spatial units can be large (e.g., hundreds of thousands). The COUNT in the query model is optional and is required in the aggregation protocol to check the probes partitioning.

Aggregate functions. PAMPAS can compute most types of aggregate statistics required by participatory sensing applications. Practically, our system can compute any type of function having reasonable time and space complexity to be computed in real-time given the relative low CPU power and little RAM of the SP. For illustrative purpose, we consider three classes of functions in this part:

- (i) *Typical algebraic functions*: count, sum, average, standard deviation. Such ag-

gregate functions are the most popular in the works related to participatory sensing [18], [28], [90]. These functions allow for example to compute the average travel time or the traffic density (count the number of cars) in a road network.

- (ii) *Specific functions*: inverse distance weighting (IDW). Applications may require specific aggregate functions. For instance, an application monitoring the noise pollution in the city could use the IDW function to compute, from the inputted samples, a heat-map of the noise level in the entire space [74].

- (iii) *Holistic functions*: median, percentile, top-k. Such functions are also frequently used in statistical computations. Their particularity is that the computation of the result requires accessing the entire sample set and cannot be achieved incrementally by accessing only subsets of samples as with the previous two classes of functions.

An important observation is that the cryptographic solutions cannot be applied for specific or holistic functions (see Section 4.2 in Chapter 4). Also, the holistic functions cannot be computed efficiently in a distributed architecture by the secure protocol proposed in [111] (as shown in Section 5.5).

5.1.4 Protocol requirements

In the light of the above description of the proposed user-centric architecture, the PAMPAS protocols have to deal with the following challenges: (i) *Privacy*: By keeping all the sensitive data in the SPs, the adopted user-centric architecture matches this requirement in contrast with a server-centric architecture. In short, the computation protocol should not reveal to the SSI any additional information about the participants' paths besides what the SSI can infer from the aggregate result. (ii) *Generality*: the protocols should be able to compute any type of function over the spatiotemporal sensed measures by the mobile users and covering a large observation space. This is different from the works based on cryptographic approaches in which, typically, only basic computation (e.g., simple aggregates like sum, average) can be achieved and only in specific locations over limited periods of time. (iii) *Efficiency*: the protocols should be highly efficient to be able to *continuously* compute the aggregate results in *real-time* with very *limited resources*. Indeed, for economic and security reasons, the SPs used for data processing have low resources and limited availability. Hence, it is necessary to minimize the computation and communication costs of the PAMPAS protocols. (iv) *Scalability*: the protocols should allow PAMPAS to scale to a large number of participants (e.g., up to millions of users), high sampling frequencies, and very large regions. (v) *Accuracy*: PAMPAS should continuously reflect the sensed measures with good precision. In other words, protecting the users' privacy should not impact the accuracy of the aggregate result computed by the protocols.

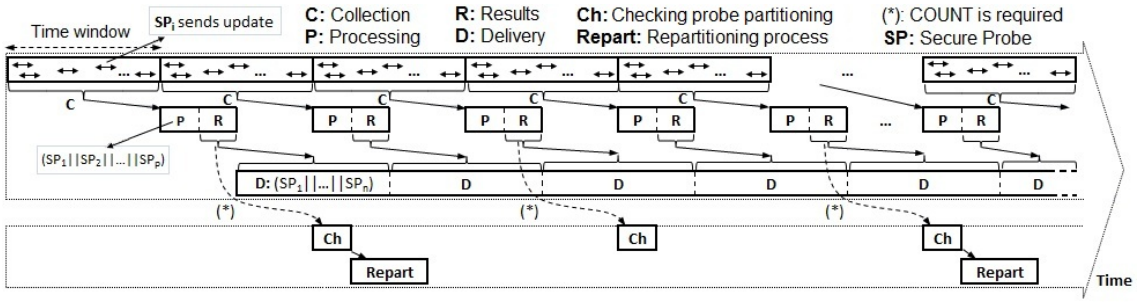

Figure 5.4: Workflow representation of the global protocol in PAMPAS

5.2 Global aggregation protocol

The global privacy-preserving protocol in PAMPAS consists in three phases that are repeatedly executed in pipeline (see Figure 5.4). First, the SSI collects all the sensing updates sent by the participants for a period equal to the sliding time window (i.e., the *collection period*). Each update is encrypted using non-deterministic encryption so that the SSI cannot gain any knowledge from these updates. All the SPs share a secret key[1] to encrypt/decrypt the updates. Note that, although an SP can decrypt the updates, a user is not allowed to access the decrypted data in her SP and that the tamper-resistant hardware protects the transiting data also from the user herself. Therefore, as for the SSI, the users have access only to the final results and not to the raw data.

At the end of the collection period, the SSI triggers the *processing period*. In this phase, only a small subset of SPs, which are randomly selected by the SSI, are involved. The SSI partitions the collected samples such that the number of updates in a partition can fit the RAM resources of an SP (otherwise, the persistent Flash storage of the SP has to be used incurring a much higher computation cost). Then, each sample partition is sent to an SP, which computes a partial aggregate result for the received updates. The encrypted results are sent back to the SSI. Finally, the *delivery period* consists of delivering the current partial aggregate results to the queriers. Each querier needs to perform the final aggregation of these partial encrypted results.

Algorithms 1 and 2 give the detailed description of the operations executed by the SSI and the SPs respectively. In the following, we denote by E_k and nE_k the symmetric deterministic and non-deterministic encryption with the key k, and by E_k^{-1} and nE_k^{-1} the opposite decryption operations.

To address the performance limitations of the existing protocols [111] (see Section 4.2 in Chapter 4), the aggregation protocol in PAMPAS groups the participants based on their location, which permits processing together the generated samples in a group by a single

[1]To increase security, the secret key is renewed periodically. The key is generated randomly by a randomly chosen SP.

Algorithm 1: PAMPAS protocol on the SSI-side

***collection_period()*:**
  /* Receive encrypted updates from SPs */
  while $(true)$ **do**
    $message = (E_k(G_i), nE_k(sample))$
    $store(message) \rightarrow list[E_k(G_i)][currentTimeWindow]$

***processing_period()*:**
  foreach i *in* $\{E_k(G_i)\}$ **do**
    /* feed in parallel the randomly selected SPs */
    randomly select $SP_i \in E_k(G_i)$
    while $message \leftarrow list[E_k(G_i)][lastTimeWindow]$ **do**
      $send(message, SP_i)$
  foreach i *in* $\{E_k(G_i)\}$ **do**
    /*Receive the final results from worker SPs*/
    $enc\_result_i^{final} = (E_k(G_i), nE_k(result))$

***delivery_period()*:**
  foreach i *in* $\{E_k(G_i)\}$ **do**
    /*Push $result_i$ to all requesting SPs*/
    $multicast(enc\_result_i^{final}, \{SP_k\})$

Algorithm 2: PAMPAS protocol on the SP-side

***collection_period()*: /* for all SPs */**
  /* Generate and send the sensing update: $update(G_i, sample)$ */
  $message = (E_k(G_i), nE_k(sample))$ $send(message, SSI)$
  /* Send a fake sample to the SSI with probability $P_{G_i}^{fake}$ */
  if $rand(0, 100) >= P_{G_i}^{fake}$ **then**
    $fake\_message = (E_k(G_i), nE_k(fake\_sample))$ $send(fake\_message, SSI)$

***processing_period()*: /* only for the selected SPs, one for each G_i */**
  while *message = receive(SSI)* **do**
    $sample = nE_k^{-1}(message)$
    $result = result \oplus sample$
  $enc\_result_i^{final} = (E_k(G_i), nE_k(result))$
  $send(enc\_result_i^{final}, SSI)$

***delivery_period()*: /* for all SPs */**
  /* Pull the results for required $\{G_i\}$ from the SSI */
  foreach i *in* $\{E_k(G_i)\}$ **do**
    $send\_request(E_k(G_i), SSI)$
    $result_i^{final} = nE_k^{-1}(receive(SSI))$

SP. To this end, the users also send the deterministically encrypted value of the spatial unit they are currently located in, in addition to the non-deterministically encrypted value of the sample, i.e., $message = (E_k(groupID), nE_k(sample))$ (see Algorithm 1, line 4 and Algorithm 2, line 3). Consequently, the SSI can group the messages based on the encrypted unit number and then send each group of samples to a different SP for aggregation (see lines 7-9 in Algorithms 1 and 2). By doing so, the advantage is manifold. First, the processing period is guaranteed to terminate in a single iteration, since each involved SP produces directly the aggregation result corresponding to a spatial unit. This greatly improves both the computation and the communication cost of the aggregation process. Second, data processing by an SP is also efficient since only one aggregate is computed, which greatly reduces the RAM requirements and avoids/reduces the usage of the persistent storage. Third, the final aggregate result is also partitioned and the queriers can demand the results only for specific spatial units, which further improves the communication cost.

However, despite all these benefits, the above approach has one fundamental shortcoming both originating from the skewed spatial distribution of the participants. Although the exact location of the updates and the unit ID are hidden, the SSI knows the number of participants in each spatial unit. If the SSI has access to side information about the spatial distribution of the users (e.g., global traffic density information), it may use this information to infer the (approximate) location of the participants and compromise their privacy.

5.3 Probe partitioning protocol

To solve the privacy problems that could appear due to the skewed spatial distribution of the participants, PAMPAS continuously partitions the set of participants based on their current location and the spatial units of the query. Similar to the global aggregation protocol, this privacy-aware partitioning protocol is executed by SPs. The idea is to group SPs located in adjacent spatial units such that the number of SPs in the resulted groups is approximately the same. Therefore, a group G_i will cover several spatial units and include all the SPs in these units.

The probe partitioning has to be recomputed periodically to keep the groups balanced since the users' distribution in space changes over time. Moreover, the groups should contain users located in closely situated spatial units to maximize the lifetime of a partitioning. The challenge is to implement a partitioning algorithm that can be executed periodically at SPs because the typical spatial partitioning algorithms are much too costly to be considered in our context (i.e., limited-resources SPs).

Our algorithm is based on the following idea. We use a space-filling curve to index the spatial units of the application query. A space-filling curve has the property to map

Algorithm 3: Checking probe partitioning (SP-side)

```
1  check_probe_partitioning():/* one randomly selected SP */
2      /* Pull all the results from the SSI */
3      foreach i in {E_k(G_i)} do
4          send_request(E_k(G_i), SSI)
5          enc_result_i^final = receive(SSI)
6          allCounts[G_i][] ← E_k^{-1}(enc_result_i^final)
7          update localy stored counts for spatial units /* also required to compute
               the probability to generate fake samples */
8          compute weights[G_i] = SUM(allCounts[G_i][])
9      compute standard_deviation(weights)
10     if standard_deviation(weights) < threshold then
11         send_for_broadcast(nE_k(allCounts), SSI)
12     else
13         execute probes_repartitioning()
```

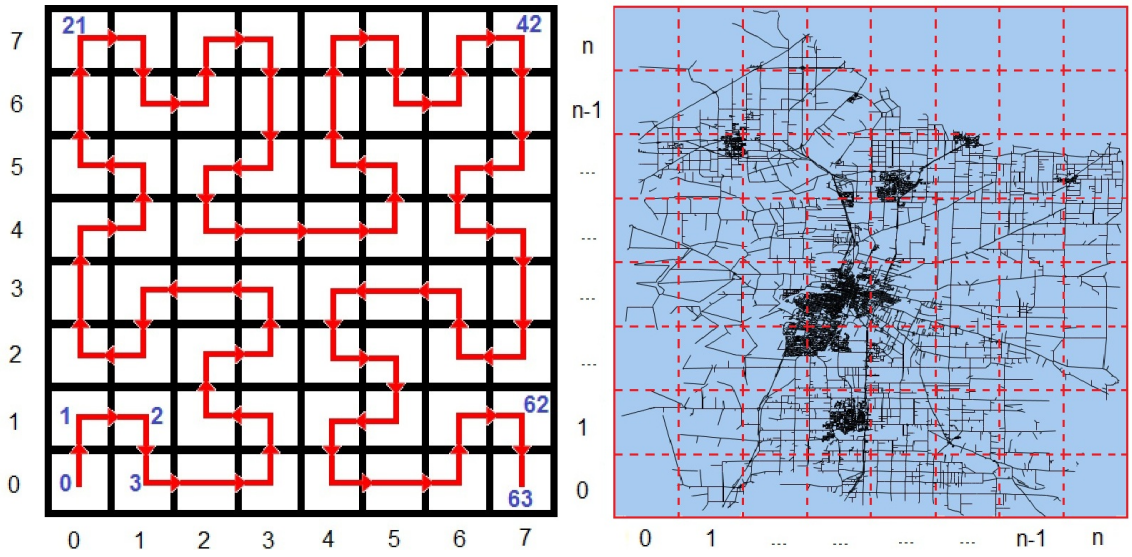

Figure 5.5: Hilbert indexing of spatial units

a multidimensional space to a one-dimensional space such that, for two objects that are close in the original space, there is a high probability that they will be close in the mapped target space [13]. Then, we sort the spatial units on the space-filing curve index. Once sorted, an approximate balanced grouping can be checked and computed in O(G) space complexity and O(N) time complexity, where N is the number of spatial units and G the number of probes groups.

Indexing the spatial units. Several space-filling curves have been proposed [13]. The most prominent ones include the snake-like curve [13], Peano curve [35], Gray code [31],

[32], Hilbert curve [34], [50] or z-ordering curve [66], [78], [77], [79]. However, the comparative tests in [1], [50], [33] show that the Hilbert and z-ordering are outstanding among these choices for spatial data. Consequently, in our system, we use Hilbert curves [34], [50], but other types of space-filling curves can be used as well to index the spatial units considered by the participatory sensing application (e.g., z-curves). In the case of free movement, the indexing is straightforward since the space is already partitioned with a uniform grid (see Figure 5.5 left). Then, we cover the grid cells with the Hilbert curve corresponding to the grid granularity and label each cell with the obtained Hilbert index. In the case of constrained movement, the indexing requires two steps. First, we cover the transportation network with a uniform grid (see Figure 5.5 right). The grid granularity is chosen such that the number of network segments intersecting with a grid cell is low for most of the cells. Then, the grid cells are indexed with a Hilbert curve and each network segment is labeled with the Hilbert index of the cell containing the segment center. In case several segments are contained by a cell, the segments are sorted by the x-coordinate and the y-coordinate of their centers and labeled accordingly. Once the spatial units are indexed, they are sorted on the index value and the sorted unit vector is broadcasted to all the participants to be used in the probe partitioning phase.

Checking and repartitioning the probe grouping. Periodically, our system verifies if the current probes partitioning is still balanced with respect to the number of probes in each group. The verification frequency depends on the dynamicity of users in space. In PAMPAS, the checking and repartitioning processes can be executed often (i.e., every few seconds) due to their low cost. When a partitioning checking is triggered, the system computes a *count* aggregate in addition to the application aggregate function (see Figure 5.3), which gives the actual number of users (SPs) in all the spatial units. The count aggregate result is then pushed to an SP randomly chosen by the SSI . The checker SP decrypts the results and updates the weights of the sorted spatial unit vector (lines 4-7 in Algorithm 3). This operation has O(N) complexity assuming that a small index containing the partitions frontiers is kept in memory by the SP (which requires only G Flash addresses to be kept in RAM). At the same time, the SP computes in memory the count by group (since the groups are sent one by one by the SSI, line 8 in Algorithm 3) and compares the counts. If the balancing of the current probes partitioning is within the predefined limits, the checker SP sends the current values to all the other SPs (i.e., exchanged encrypted through SSI), which update the weights of the spatial units with the new count values. Otherwise, the checker SP computes a new partitioning.

Once the sorted vector of spatial units is updated with the new weight values, the probe repartitioning can be efficiently computed in O(N) and O(G) time and space complexity respectively (see Algorithm 4). The algorithm consists in reading the weights in the order of the Hilbert index and determining the partition borders. To set the partition borders

Algorithm 4: Repartitioning process (SP-side)

```
1  PROBE_REPARTITIONING():/* one randomly selected SP */
2      compute QI_comp and QI_comm for current G
3      while true do
4          /* adjust the number of groups G */
5          if QI_comp > QI_comm then
6              tG = 2 * G
7          else
8              tG = G/2
9          /* repartition for tG */
10         avgGroupWeight = SUM(allCounts[])/tG
11         currentGroupWeight = 0
12         for i = 0 to NUM_SPATIAL_UNITS − 1 do
13             currentGroupWeight+ = allCounts[i]
14             if currentGroupWeight ≈ avgGroupWeight then
15                 newPartitionMilestones[].add(i)
16                 currentGroupWeight = 0
17         /* check if the new partitioning for tG is better than for G*/
18         compute tQI_comp and tQI_comm for tG
19         if tQI_comp + tQI_comm < QI_comp + QI_comm then
20             G = tG; QI_comp = tQI_comp
21             QI_comm = tQI_comm
22         else
23             break
24     message = allCounts[]||newPartitionMilestones[]
25     send_for_broadcast(nE_k(message), SSI)
```

we use a greedy algorithm, which adds spatial units to a group as long as the total weight of the group is lower than a threshold value (lines 12-16 in Algorithm 4). The threshold is computed as the ratio between the total number of probes and the number of groups (line 10 in Algorithm 4), and represents the average number of users per group. The partitioning result is a list of G *milestone* indicating the group borders in the sorted index of spatial units (line 16 in Algorithm 4). The result is then encrypted and delivered, through SSI, to all users (lines 24-25 in Algorithm 4), which update their partitioning data and generate new samples accordingly starting from the next computation window.

The proposed probes partitioning algorithm has low complexity and can be efficiently executed even with the low resources of an SP. However, the partitioning algorithm cannot guarantee a certain degree of balancing of the partition weights. Yet, the partitioning balancing is required to avoid leaking any information regarding the spatial distribution of users. To deal with this problem, the SPs generate fake samples in all the probe groups

having a number of users lower than the maximum size group. Therefore, in the collection period of each computing time window, an SP sends probabilistically a dummy sample in addition to the real sample. The probability to send a fake sample is proportional with the difference between the maximum size group and the number of users in the SP's group, and inverse proportional with the number of users in the group. The same approach is used to hide the number of spatial units in each group. At the end of the aggregation phase, each aggregating SP adds to the result a number of fake values equal to the difference between the maximum number of units in the groups and the number of units in the current group. In this way, all the partial aggregate results received by the SSI have the same size and the SSI cannot infer the number of cells in any group.

Choosing the Number of Probe Groups. The cost of the aggregation protocol is composed of the computation cost at the SP side and the communication cost between the SSI and the SP. The number of probes groups impacts both the computation and the communication costs. Specifically, the computation cost decreases with the increase in the number of groups and attains the minimum value when the number of groups is equal to the number of spatial cells, i.e., an SP is used to aggregate the samples for each spatial unit. But increasing the number of groups leads to a higher imbalance in the groups' weights, which in turn requires injecting more fake samples and enlarges the communication cost. Therefore, modifying the number of groups has opposite effect on the computation and the communication cost.

$$QI_{comp} = Max_{i=1,G}[Comp\_time_i] - Max_{j=1,\#spatialUnits}[Comp\_time_j] \quad (1)$$

$$QI_{comm} = \frac{size(sample)}{bandwidth} \sum_{i=1}^{G} \{Max_{j=1,G}[Count_j(probes)] - Count_i(probes)\} + \frac{size(sample)}{bandwidth} \sum_{i=1}^{G} \{Max_{j=1,G}[Count_j(spatialUnits)] - Count_i(spatialUnits)\} \quad (2)$$

PAMPAS computes two quality indicators to measure the impact of the number of groups on the computation and communication costs, i.e., QI_{comp} and QI_{comm}, as defined by Formulas (1) and (2). QI_{comp} estimates the degradation of the computation time at the SP side generated by the fact that several spatial cell aggregates are delegated to one SP instead of using one SP for each cell. Estimating the computation time is fairly simple since the time is typically linear with the number of samples to be processed by the SP, assuming that the aggregation can be entirely processed in RAM. However, the cost model can be extended to the case in which it is required to access the secondary storage.

QI_{comm} estimates the degradation of the communication cost caused by the imbalance of the partitions. The first term indicates the overhead incurred by the fake samples generated to balance the probes groups, while the second term measures the overhead of generating fake results to balance the number of aggregates in each group.

Each time an SP computes the probes partitioning, it also computes the values of QI_{comp} and QI_{comm} (line 2 in Algorithm 4). If $QI_{comp} > QI_{comm}$, the SP multiplies by two the number of groups and re-partitions the probes. If $QI_{comp} < QI_{comm}$ the SP divides by two the number of groups and re-partitions the probes (lines 5-8 in Algorithm 4). The SP continues to adjust the number of groups until $QI_{comp} + QI_{comm}$ has minimum value (lines 19-24 in Algorithm 4), meaning that the aggregation cost is near optimal. Thus, this process allows adapting the number of groups to the number and the spatial distribution of the probes.

5.4 Security analysis

The SSI does not have the encryption key, so it cannot access the transiting data it sees. In addition, the non-deterministic encryption protects the data against frequency-based attacks. The SSI may also try to buy an SP and pass for a user to gain access to the encryption key. However, this would be useless since the tamper-resistance of an SP protects the secret, the data and the code execution against physical attacks. Therefore, in this case, the SSI would gain access only to the final aggregate result. The same would happen if the SSI colludes with the querier, i.e., a user or an application having access to the aggregation result. Finally, since the samples are communicated through anonymizers, the SSI cannot identify the senders or link consecutive messages from the same user.

Another option for the SSI would be to infer some information from the deterministically encrypted group id values. Nevertheless, the SSI cannot perform a frequency-base attack using the encrypted group id, since all the groups contain approximately the same number of messages. Therefore, the SSI cannot infer the corresponding (approximate) location of a group or the topological neighborhood of the groups (which would be the first step to attack the users' privacy). Hence, the only knowledge the SSI acquires is the number of groups and its evolution over time, which does not endanger the users' privacy. Note that even if the SSI has somehow access to the full partitioning information and the corresponding encrypted id, a user is still hidden under the corresponding partition area and within the crowd in the same group (let us recall that the messages are sent anonymously so it is hard for the server to link the messages coming from the same user). Hence, the protocols are secure and fully protect the privacy of the users.

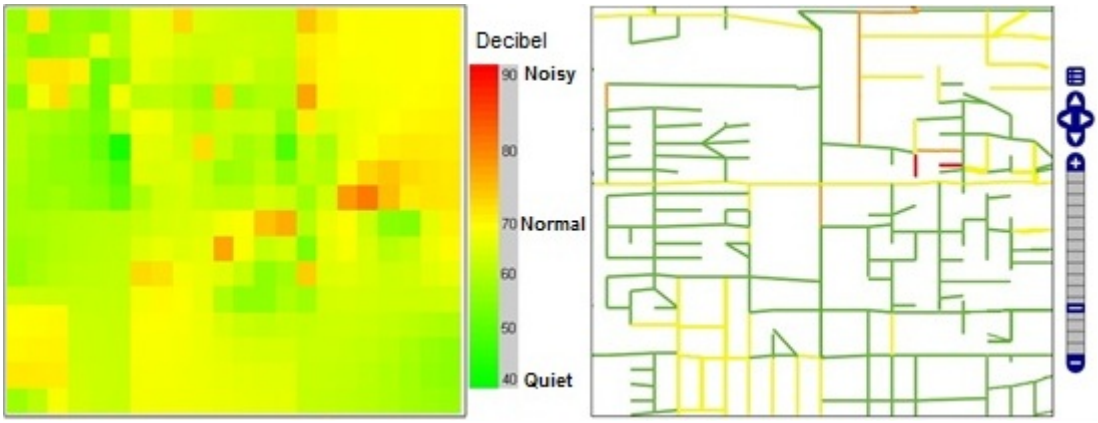

Figure 5.6: Aggregation maps for two applications: noise monitoring (left) and traffic monitoring (right)

5.5 Experimental evaluation

The goals of our experimental evaluation are twofold: (i) compare the execution time and scalability of PAMPAS with those of a state-of-the-art protocol described in [111]; (ii) quantify the cost and scalability of our partitioning protocol. We implemented and validated PAMPAS through emulations using a development board which has a hardware configuration representative for secure hardware platforms. As applications for our experiments, we used traffic monitoring and noise monitoring with two synthetic datasets representing a small and a medium-size city. Figure 5.6 illustrates our graphic interface for these applications; it shows the aggregate results for the noise heat-map and the average travel time for the road network. A demo of our prototype was presented in [113] using a traffic monitoring scenario.

5.5.1 Experimental setting

Hardware platform. Our experimental evaluation uses two types of hardware, a PC and a development board. The PC (3.1 GHz i5-2400, 8GB RAM, running Windows 7) plays the role of the SSI and also displays the aggregate results in a graphical form. The board plays the role of an SP, and it is equipped with an MCU with a 32-bit RISC CPU at 120MHz, a crypto-coprocessor implementing AES and SHA in hardware, 128KB of static RAM and 1MB of NOR Flash to store the software stack. It also includes a smartcard chip hosting the cryptographic credentials (i.e., the secrete encryption keys) and an SD card reader allowing for a large storage capacity. We use a commodity SD card (Samsung SDHC Essential Class 10 of 32GB) as secondary Flash storage. The PC and the SP board communicate over a 100Mbit Ethernet connection. Importantly, our implementation limits the RAM consumption to only 30KB to validate the proposed protocols with less

powerful secure devices. To emulate multiple SPs, we execute sequentially on the board the aggregate computations and communications for all the worker SPs and measure the "parallel" execution time as the maximum aggregation time in the execution sequence. The experimental results were also validated [113] using several secure tokens with the same HW configuration as the board.

Baseline system. To underline the importance of the optimizations in PAMPAS, we implemented the *secure protocol* proposed in [111] and took it as the baseline. This protocol can be applied without modification to aggregate the samples collected in each time window. Note that in [111], two more protocols are proposed that are even more expensive than the secure protocol if considered in our context.

Datasets and aggregate functions. We created synthetic datasets to test the efficiency and scalability of PAMPAS. We used the well-known Brinkhoff generator [17] to generate mobility traces on two real road networks of the cities of Oldenburg (Germany) and Stockton (San Joaquin County, CA). Oldenburg is a small size network having 7035 road segments, while Stockton is an average size road network having 24123 segments. Depending on the network size, we generated traces corresponding to a medium and large number of users. With Oldenburg, the medium and large datasets contain 47 thousand and 270 thousand users respectively. With Stockton, the medium and large datasets contain 200 thousand and 1.35 million users respectively. The spatial distribution of the traces follows the network spatial density.

To show the generality of PAMPAS, we selected three aggregate functions, i.e., average, IDW [74] and median, corresponding to the three aggregate types described in Section 5.1.3. We associate the average and median aggregates with the traffic monitoring application, i.e., compute the average travel time and the median speed for each road segment in a road network. Hence, these two scenarios consider the constrained movement type. The IDW aggregate is associated to the noise-level monitoring application and a free movement type. In this case, we use the same generated mobility traces, but consider them in the 2D space instead of the network space. Also, we use a 64x64 grid (i.e., 4096 spatial units) to partition the 2D space for the free movement scenario. The speed sample values are directly generated by the moving objects generator, while the noise values are generated by us randomly and proportionally to the number of probes in the spatial unit.

5.5.2 Performance evaluation

Execution time. Figure 5.7 shows the aggregation time for the three functions for both protocols with 200,000 probes in Stockton. The aggregation time is global, i.e., it includes both the computation and communication time. The results indicate that PAMPAS is very efficient since it requires only a few seconds to compute the aggregate results for all the tested functions. On the other hand, the baseline protocol is much more costly

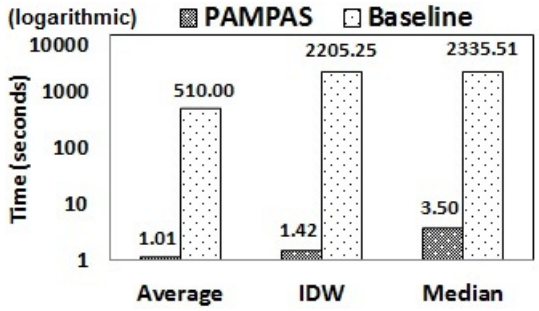

Figure 5.7: Aggregation time

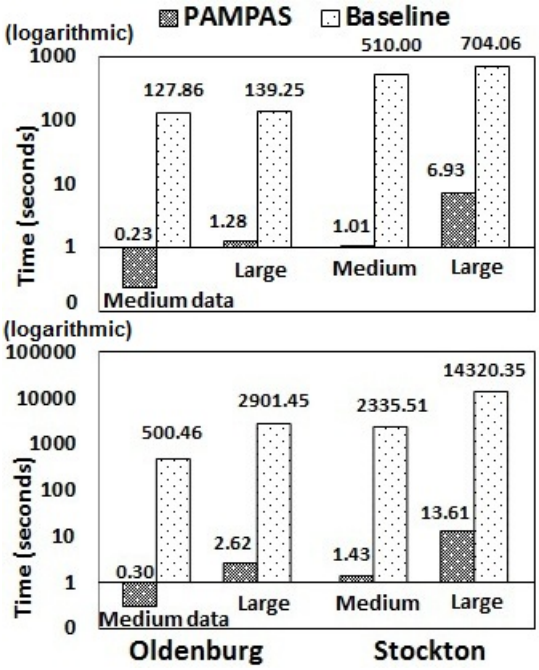

Figure 5.8: Scalability of the PAMPAS and baseline protocols with average function (top) and median function (bottom)

(especially for complex aggregate functions) leading to aggregation times up to three orders of magnitude higher than PAMPAS. These results demonstrate that PAMPAS achieves the goal of working in real-time (e.g., the drivers can see the traffic conditions in real-time).

Scalability. We further test the scalability of the protocols with different number of probes, spatial units, and aggregate functions. Figure 5.8 shows the aggregation time for the two protocols for the average (top graph) and median (bottom graph) functions with medium and large number of users on both road networks. The results confirm that only PAMPAS is scalable w.r.t. all the varying input parameters. In the worst case, the computation time attains 14 seconds to compute the median speed for 1.3 million samples covering 24,000 spatial units.

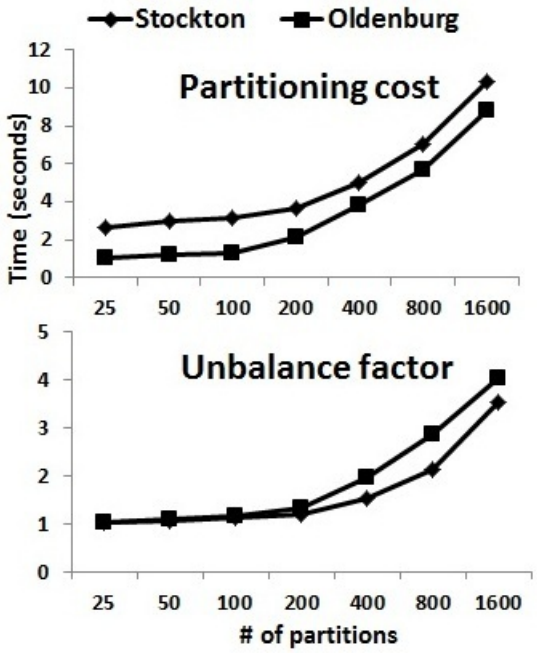

Figure 5.9: The partitioning costs (top) and the partitioning imbalance factor (bottom) with different number of partitions

The baseline protocol does not scale with the number of samples and especially with the number of spatial units. Practically, the baseline can provide real-time aggregation only for a small number of spatial units (i.e., 7000 in Oldenburg) and basic aggregate functions (e.g., average). The very limited RAM resources of the SPs and the impossibility to efficiently parallelize the aggregate computation make the baseline inadequate for the requirements of participatory sensing applications.

Cost and scalability of partitioning protocol. Figure 5.9 (top) presents the partitioning computation time for both Oldenburg and Stockton networks. A new partitioning can be computed in a few seconds by an SP. This means that the checking and probes re-partitioning can be executed frequently, which allows PAMPAS to adapt to even fast changes in the spatial distribution of the probes. Most of the partitioning cost resides in reading and writing the partitioning data to the secondary Flash storage. This also explains the increase of the partitioning time with the number of partitions, since in this case the I/O operations are executed at a smaller granularity, which is more costly.

Figure 5.9 (bottom) indicates that the partitioning imbalance factor, i.e., the ratio between the maximum and the average partition size, increases with the number of partitions. The imbalance factor is an important indicator in PAMPAS since the higher the imbalance, the higher the number of fake injected samples and, therefore, the communication cost.

Figure 5.10 shows the impact of the number of partitions on the global aggregation time as well as on the computation and communication cost, which compose the total time. As

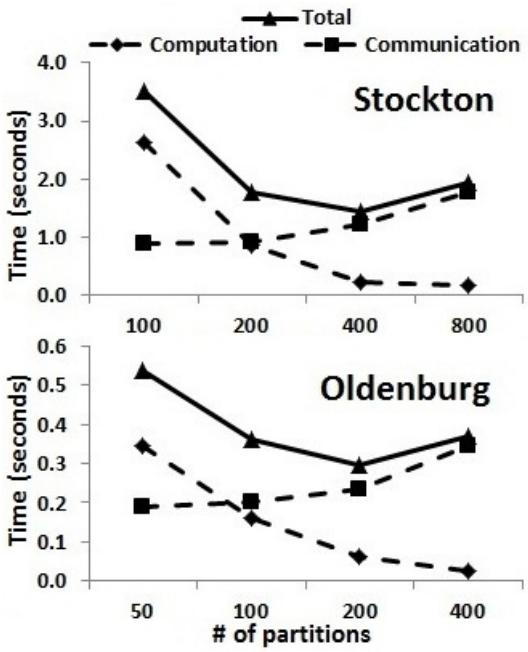

Figure 5.10: Communication and computation costs with different number of partitions

expected, the computation time decreases with the increase of the number of partitions since the amount of work done by the aggregation SPs also decreases. Conversely, the communication time increases with more partitions since more fake samples are injected into the system as explained above. Globally, the near-optimal aggregation time is obtained with a number of partitions that minimizes the cumulated degradation of the computation and communication costs as discussed in Section 5.3.

Discussion. It is worth mentioning that the aggregation time can be greatly improved by increasing the processing power and the communication bandwidth of the SSI. For example, increasing the server bandwidth from 100Mbit to 1 GBit, makes the maximum aggregation time (i.e., median function with the large Stockton dataset) to drop from 14 seconds to less than 7 seconds. Also, in some scenarios, pushing the computation in the user devices may be problematic (e.g., battery powered devices, other applications running at the same time in the device). However, PAMPAS minimizes this type of problem due to its high efficiency. For instance, in our tests, a user participating in the system for one hour, has a probability between 10.6% and 26.6% to participate once to an aggregate computation assuming that aggregates results are produces every 10 seconds, and a probability between 0.004% and 0.02% to do a repartitioning assuming that the probes partitioning is checked every 1 minute. In all cases, the computation is done in a few seconds at most and requires only modest resources.

5.6 Conclusion

This part proposes PAMPAS, a privacy-aware mobile participatory sensing system based on a distributed architecture and personal secure hardware. This combination allows PAMPAS to achieve the same level of privacy as cryptographic solutions without having to sacrifice generality, scalability, and accuracy. The proposed aggregation solution is, to the best of our knowledge, the first proposal of a distributed protocol that is secure, efficient, and scalable and that fits both the strict hardware constraints of secure personal devices and the real-time constraints of participatory sensing applications. The experimental evaluation based on representative hardware for secure platforms validates the proposed solution.

Part III

Implementation and Demonstration

Chapter 6

The General Context of the KISS Project

The work in this thesis is strongly related to the ANR KISS (Keep your personal Information Safe and Secure) project. Therefore, in this Chapter, we present an overview of the KISS project [3] and describe our contributions in this project.

6.1 Overview of the KISS project

An increasing amount of personal data is automatically gathered and stored on servers by administrations, hospitals, insurance companies, etc. Smart appliances surrounding individuals also accumulate spatio-temporal sensitive information (e.g., healthcare monitoring, geolocation) on servers. Meanwhile, an increasing amount of digitized personal data is sent to citizens (salary forms, insurance forms, invoices, phone call sheets, banking statements, etc), who themselves often count on service providers to reliably store this data and make it available through the Cloud. However, these benefits must be weighed against privacy risks incurred by centralizing data on servers. Indeed, there are many examples of privacy violations arising from negligence, abusive use or attacks, and even the most secured servers are not spared.

The KISS project draws a radically different vision of the management of personal data. It builds upon the emergence of new portable and secure devices known as *secure tokens* (e.g., mass storage SIM cards, secure USB sticks, smart sensors) combining the security of smart cards and the storage capacity of NAND Flash chips. The idea promoted in KISS is to embed, in such devices, software components capable of acquiring, storing and managing securely personal data. These software components form a full-fledged Personal Data Server (PDS) which can remain under holder's control. However, our approach does not sum up to a simple secure repository of personal data. The ambition is threefold. The

first objective is to allow the development of new, powerful, user-centric applications thus requiring a well organized, structured and queryable representation of user's data. Second, KISS wants to provide the data holder with a friendly control over the sharing conditions related to her data and to provide the data recipient with certified information related to their provenance. Third, to give sense to this vision, PDSs must provide traditional database services like durability, query facilities, transactions and must be able to interoperate with external data sources.

Compared to an approach where all personal data is gathered on traditional servers, the benefit provided by PDS is manifold: (1) the PDS holder becomes his own Storage Provider thereby precluding abusive usages from external service providers, (2) secure tokens provides tangible elements of trust (i.e., tamper-resistance, holder's ownership) which cannot be provided by any traditional server, (3) privacy principles can be enforced for the data externalized by the holder provided the recipient of this data is another PDS and (4) the holder's data remains available in disconnected mode.

Converting the Personal Data Server vision into reality introduces however several scientific challenges. First, the secure token, central element of the approach, exhibits strong hardware constraints (e.g., little RAM, NAND Flash storage). Traditional core database techniques (storage and indexing, query and transaction processing) need then to be fully revisited to design an embedded database engine that provides acceptable performance whatever the form of the embedded data (regular or spatio-temporal) and of the queries (SQL-like, key-value like, spatio-temporal). Second, the PDS approach aims at helping individuals to better protect their privacy. Hence, a unified data model must be provided to organize their personal space and to express how data is shared and protected at high abstraction level. Third, the traditional functions provided by a central server must be re-established in a rather atypical distributed environment combining a large number of highly secure but low power secure tokens with a powerful but unsecured server infrastructure, all this without sacrificing data privacy.

6.2 Thesis' contributions in KISS

This section presents first the architecture of the embedded part of the Personal Data Server framework, i.e., the software components running on each secure token to manage the personal data of a single individual. Then, we sketch the thesis' contributions in this global architecture.

Figure 6.1 pictures the embedded architecture of a PDS. We detail it below in a bottom-up way.

Raw access layer: this layer manages the physical images of all objects belonging to the digital space of an individual. Physical images are managed differently depending on the

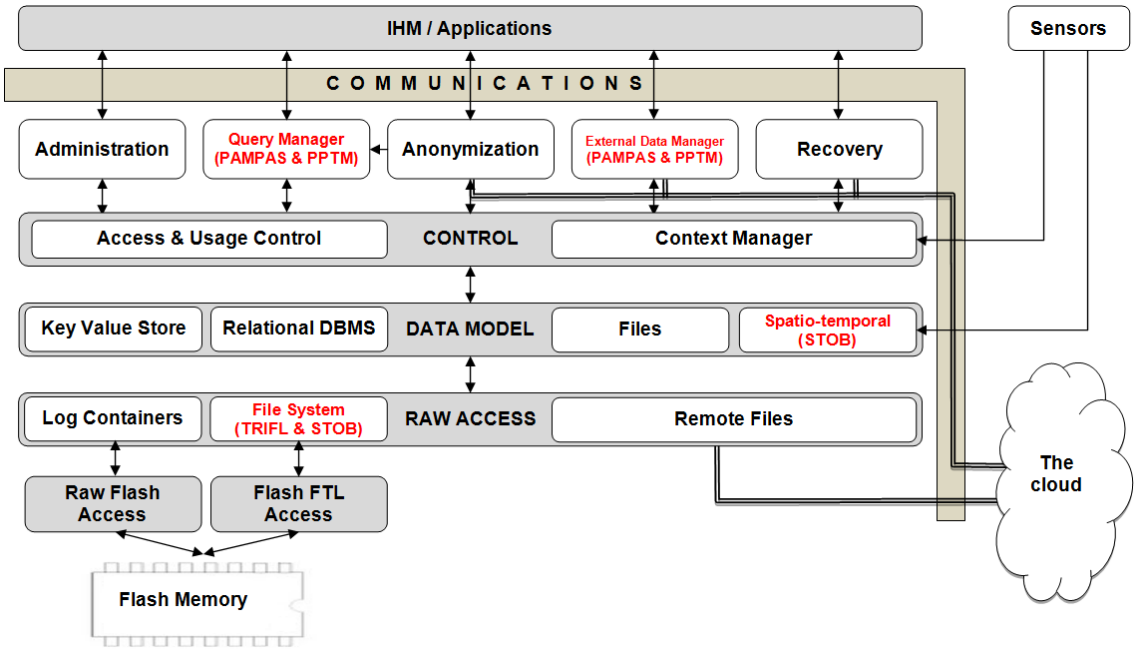

Figure 6.1: KISS architecture and our contributions

medium used to store them. Files stored locally (e.g., documents hosted by the secure token to make them available even in disconnected mode) are stored in an on-board Flash memory. These files are managed through a traditional file system, built on top of an FTL. Fine grain objects like tuples of a relational database are much more challenging to manage because NAND Flash and FTL badly accommodate fine-grain insert/updates. Hence, fine-grain objects are managed through ad-hoc structures called Log Containers [10] which bypass the FTL firmware. So, on-board Flash memory can be physically made of two separate devices or be logically split in a raw partition and an FTL partition. Remote files are links to files physically stored outside of the secure token by a storage supporting service, typically in the Cloud.

Data model layer: This layer provides a structure and the associated query facilities to the raw data. The objective is to give the user a mean to organize his digital space in a unified way. This unified model may mix objects of various forms, namely: files, relational tables and tuples, spatio-temporal data (e.g., sensed geolocalized data) and key-values lists. The data model may allow the user to associate user-defined tags to the objects and to define virtual objects by means of logical expressions over existing objects of the user's digital space (e.g., a relational view, a folder grouping files sharing similar tags, etc).

Control layer: the objective of this layer is to define and enforce user-defined access control rules. The target of these rules can be any object (regular or virtual) of the digital space. The rules can be associated to users, to roles or to groups of individuals sharing same characteristics (friends, people belonging to the same community, working to the same

company, etc). Subjects can then be abstracted by logical expressions expressed over a set of credentials managed by a Context manager. For each individual, credentials take the form of objects of his digital space which are certified by their issuers and the integrity of which is protected by the secure token of the holder.

Other components can be seen as Core applications on top of the other layers. *Recovery* implements the tasks required to restore a consistent Personal Data Server in case of crash/loss of a Secure Token. It is a charge of (1) exporting local updates to an external supporting service, named Event Logger, by sending a message containing those updates to himself and (2) downloading and replaying locally all events logged beforehand to recover a crash. *External Data Manager* performs the tasks required (1) to export/publish objects (and metadata) from the local digital space to the Cloud and (2) to import objects published by others from the Cloud to the local digital space. *Anonymizer* and *Query manager* allow the PDS to participate to the distributed computation of a global query or of an anonymized release. Finally, *Administration* implements the task linked to the administration of the PDS.

Communication manager: this manager takes in charge all communications with the outside world. It sends requests to the Communication supporting service to anonymously send/receive new messages to be delivered by/to this PDS and encrypt/decrypt them.

Applications will communicate with the embedded PDS through a standardized API (e.g., SQL/ODBC/JDBC-like) in order to implement specific services. Typically, while the data model and control layers allow the user to get a global unified view of his digital space and to fix basic user-defined privacy policies on this view, applications can provide on their own more specialized services to structure and protect part of this digital space (e.g., a tool dedicated to build picture albums, tag them and define sharing policies).

In Figure 6.1, we highlighted our contributions in the KISS architecture. Specifically, these contributions are detailed in this thesis as following:

- A spatio-temporal storage and indexing method adapted to the constraints of Flash storage, i.e., TRIFL, was presented in Chapter 3.

- A secure protocol allowing users to share sensing spatio-temporal data and to distributively and securely compute in real time spatio-temporal aggregates, i.e., PAMPAS, was introduced in Chapter 5. In addition, based on PAMPAS, we implemented an application for privacy preserving traffic monitoring (PPTM), which is presented in Chapter 7.

- An extension of the relational data model of the PDS to permit the storage and querying of spatio-temporal data in the secure token. To this end, we implemented spatio-temporal data types and related operations, which are tightly integrated with

the classical relational data types and operations. The spatio-temporal model is detailed in Chapter 8.

Chapter 7

PPTM: Privacy-Aware Participatory Traffic Monitoring Using Mobile Secure Probes

Privacy became one of the main concerns in location-based services in general and in community-based traffic monitoring in particular. This Chapter we present an adaptation of our PAMPAS in a real-application, Traffic Monitoring using Mobile secure probes. This work was implemented in the real token devices and evaluated with the generated dataset together with a large real network map. The experimental results show that our work satisfy the real-time constraints and totally match the privacy requirements. Without loss the generality, this work confirms that PAMPAS could be applied in any kinds of participatory sensing applications that use secure probes.

7.1 Introduction

Monitoring traffic has become a fundamental concern in the transportation domain for both professionals and users. Accurate and global traffic information is essential to avoid heavy congestions and therefore, reduce travel times, accidents, fuel consumption and pollution. Many works deal with the problem of traffic monitoring such as [36], [48], [92], [110] to name but a few. In particular, using vehicles as probes for traffic monitoring is now common (mostly based on mobile applications), due to the better coverage at lower costs than using road-side infrastructure. A typical participatory traffic monitoring (PTM) system collects, aggregates and disseminates the localized traffic volume and speed information. However, most current PTM solutions require users to reveal their locations to trusted monitoring servers. This raises serious privacy concerns and prevents a wide adoption of the system since, by knowing the user locations, an attacker can easily identify

the participants and infer their personal habits and activities [48].

Several approaches have been proposed to solve the PTM problem. In [110], the authors propose the VTrack system having a typical centralized architecture. In VTrack, the cars periodically send their locations to a centralized server that computes the global traffic density, while the privacy issue is not considered. Many commercial systems, such as Waze, INRIX and Navigon, are based on similar solutions. The virtual trip lines (VTLs) proposed in [48] deal with the privacy issue by distributing the PTM service implementation across several specialized servers and by providing a spatio-temporal cloaking of the users under the VTLs. Although the attack of a single system component prevents linking the identity and location of the users, choosing privacy-insensitive locations for VTLs is tricky and limits the traffic information to a part of the road network. Also, the problem of multi-component attack (or collusion) remains, as well as the high cost of building such a complex system distributed over several components. SpotMe [92] proposes a different approach consisting in mixing the real user's location with fake locations before posting them to a central server. Then, the server estimates the aggregated user locations by using the probability theory. However, the traffic estimation errors can be important (around 20%). Also, SpotMe involves higher communication costs because of the large number of fake locations, while linkability may still be a problem for users that send many consecutive location updates.

Hence, providing a high quality PTM service, while protecting the users' privacy, is still a challenge. Recently, the works in [8], [111] propose a user-centric architecture based on a secure device also called secure portable token. The main idea is to transfer the computation process at the users' side. Therefore, there is no need to trust any central server. In this model, the server is only used for communication and temporary storage of encrypted intermediary results. However, these works were designed for privacy-preserving data publishing or privacy-preserving query execution, and do not account for the real-time constraint or the continuous nature of the computations in a PTM system. In this Chapter, we introduce PPTM, a Privacy-aware Participatory Traffic Monitoring system, adopting a user-centric architecture. We first describe our system foundations. Then, we present the demonstration scenarios.

7.2 System design and secure protocols

7.2.1 System architecture

To tackle the challenges of PPTM, we propose a combined hardware and software solution. Figure 7.1 shows the general architecture of our system. In particular, the system consists of two types of hardware components: the personal secure tokens (ST) at the user-side and one supporting server infrastructure (SSI). The software stack implements three privacy

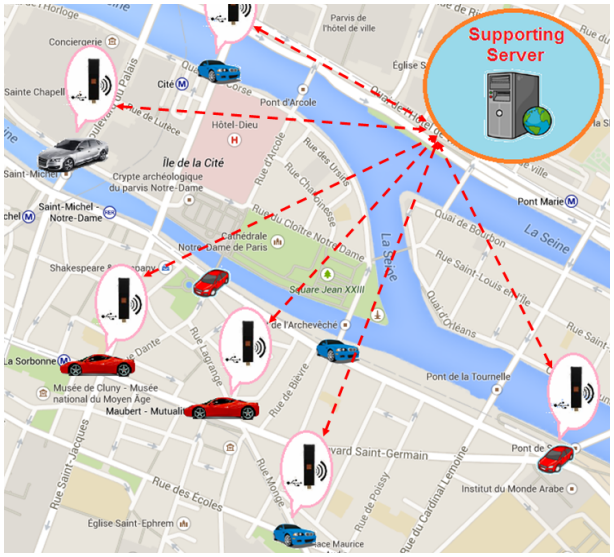

Figure 7.1: System architecture

preserving protocols required to securely send the location updates, to compute the traffic aggregation results, and to partition the road network.

Secure tokens: Any user (here a driver) using our system holds a secure portable token, which can be implemented by any type of tamper-resistant devices flourishing today, e.g., Mobile Security Card[1] (Giesecke & Devrient), Personal Portable Security Device[2] (Gemalto and Lexar), Multimedia SIM card or Secure Portable Token [111]. Whatever its commercial name and form factor, a tamper-resistant device, called *secure token* (ST) hereafter, embeds a secure microcontroller (e.g., a smart card chip) connected to a large NAND Flash memory (e.g., a SD card) and can communicate with a host through a USB, Bluetooth or Ethernet port. In our system, an ST plays three roles: (i) a mobile probe, (ii) a processing node, and (iii) a querier. Indeed, the ST sends encrypted location updates to the SSI, participates in the data aggregation, and receives the final results. Given their high level of security, the STs are considered trusted in our system.

Supporting server infrastructure (SSI): Different from the typical server-centric architecture, the SSI in our system is only used for exchanging messages between the STs and for temporary storage purposes. However, all the temporary results stored in the SSI are encrypted using non-deterministic encryption, since the SSI is not trusted. We assume the server obeys to the protocol it is supposed to do, but may try to infer anything it can from the data or behaviors it sees (i.e., an honest-but-curious or weakly-malicious threat model).

[1] http://www.gi-de.com/en/products_and_solutions/products/ strong_authentication/Mobile-Security-Card-31488.jsp

[2] "Smart Guardian" http://cardps.com/product/gemalto-smart-guardian or "Smart Enterprise Guardian" http://cardps.com/product/gemalto-smart-enterprise-guardian

7.2.2 System requirements

A PPTM system based on this user-centric architecture has to deal with the following challenges:

- *Efficiency*: the system should be highly efficient to be able to continuously compute the traffic density in real-time with very limited resources. Indeed, for economic and security reasons, the STs used for data processing have low resources (see Section 7.3.1.1). Hence, it is necessary to minimize the computation and communication costs of the PPTM protocols.
- *Scalability*: the system should scale to a large number of participants (e.g., up to millions of vehicles) to be applicable to very large networks and dense traffic.
- *Security*: the system should protect the users' privacy. By keeping all the sensitive data in the STs, the adopted user-centric architecture matches this requirement in contrast with a server-centric architecture.
- *Accuracy*: PPTM should continuously reflect the status of the traffic with good precision. In other words, protecting the users' privacy should not impact the accuracy of the traffic computation.

7.2.3 Privacy preserving protocols

Our PPTM system implements three privacy preserving protocols. First, each vehicle participating in the system periodically sends location updates to the SSI. Each update is encrypted using non-deterministic encryption so that the SSI cannot gain any knowledge from the updates. All the STs share a secret key to encrypt/decrypt the updates. Note that, although an ST can decrypt the updates, a user is not allowed to access the decrypted data in her ST and that the tamper-resistant Hardware (HW) protects the transiting data also from the user herself. Then, the remaining two protocols concern the network partitioning and the traffic aggregation, which are both processed at the user-side.

To overcome the problem of large computation and communication costs and to achieve the traffic aggregation result in real-time in spite of the limited resources of the STs, we divide the network map into many parts. Then, the aggregation is calculated in parallel for each part of the map by different STs. By doing so, the communication and computation costs will decrease by a factor equal to the number of partitions. Moreover, to avoid any information leakage to the SSI, the partitions have to be balanced with respect to the number of vehicles in each partition. This also guarantees an upper bound for the amount work of done by an ST in the aggregation phase and for the computation duration of the aggregation phase. Therefore, the partitioning process is triggered periodically in our system, following the spatio-temporal evolution of the traffic in the road network.

As with the partitioning, the aggregation phase is also done at the user-side. In contrast with the partitioning, the aggregation is processed in parallel by many STs, such that each ST computes the traffic aggregates for a specific partition. This limits the communicated and processed data to/by an ST to the updates in a partition. The output of this process is the encrypted data about the traffic volume or the average speed for all the roads in a partition. We divide the aggregation phase into three periods as following:

- *Collection period*: in this phase, the SSI collects all the encrypted update messages from the participant STs. Each message contains an additional field representing the cyphered value of the partition, but using deterministic encryption so that the SSI can classify the messages according to their partition.

- *Processing period*: The SSI randomly selects an ST per partition and sends it all the updates related to the respective partition. The ST decrypts the messages and computes the aggregation before encrypting the results and sending it back to the SSI.

- *Delivery period*: Once it received all the partial results from the aggregation STs, the SSI delivers the global result to all the users. To improve the communication costs, our system also implements an incremental computation of the aggregation traffic map.

7.3 Demonstration

In this demonstration, we present an application of our PAMPAS in a real scenario, i.e., Traffic Monitoring using Mobile secure probes called PPTM. The main aims of our demonstration are to provide the audiences many point of views of PPTM in order to show them how PAMPAS can be used to collect data, aggregate the traffic density as well as how PAMPAS partition the network in PPTM by offering three different scenarios with two running modes (i.e., normal mode and step-by-step mode).

7.3.1 Demonstration platform

7.3.1.1 The hardware platform

In our demonstration we use two kinds of devices. A PC plays the role of the SSI and also displays the demonstration graphical interfaces (see next section). The second kind of device are the secure tokens (see Figure 7.2). Our secure tokens are produced by the ZED company and have an architecture representative for STs. In particular, these devices are equipped with a microcontroller with a 32-bit RISC CPU clocked at 120MHz, a crypto-coprocessor implementing AES and SHA in hardware, 128KB of static RAM and 1MB of NOR Flash to store the software stack. The ST also includes a smartcard chip hosting the cryptographic material and a microSD card reader allowing for a large storage capacity.

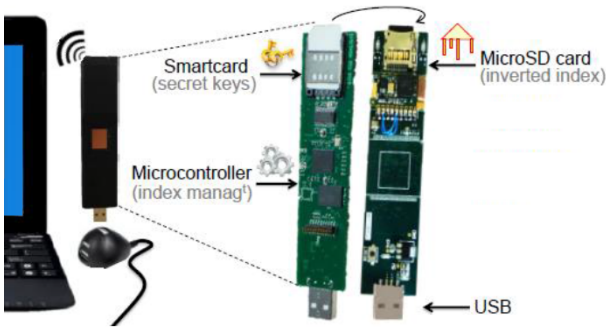

Figure 7.2: Secure token

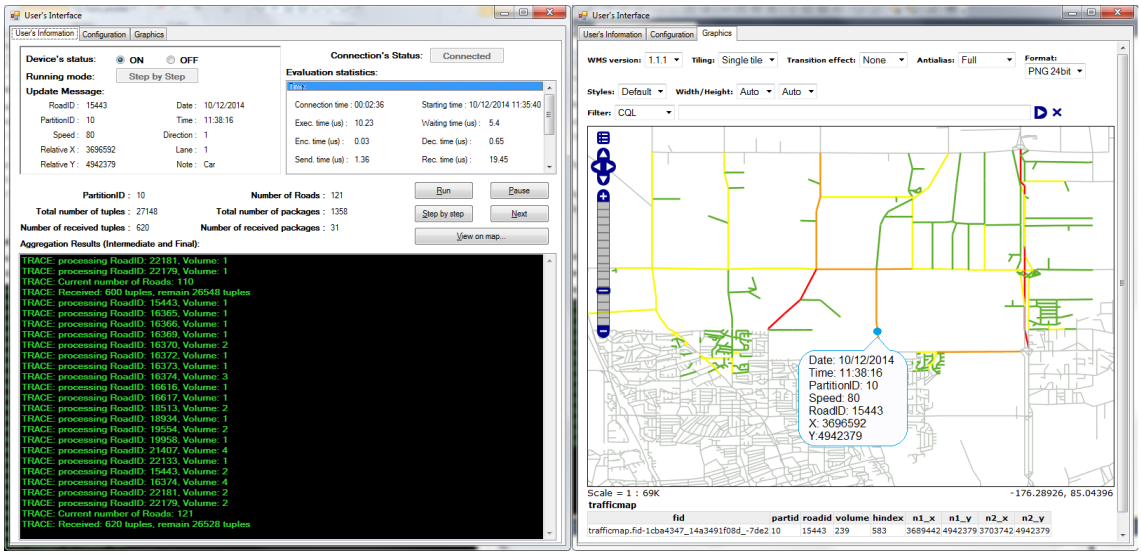

Figure 7.3: User-side graphical interface

The ST comunicates with a host either through a USB port or the Bluetooth protocol. We use four STs in the demonstration.

7.3.1.2 The graphical interface

The demonstration is based on two interfaces: a user-side interface (see Figure 7.3) and a server-side interface (which is similar to the user-side interface). The former reveals the information processing on the STs, while the latter reflects the information that can be seen by the SSI. In both interfaces, we try to show the audience our protocol in a didactic way, by supporting two kinds of running modes: "step-by-step" mode and "run" mode. The audience can also pause the process or move to next step of the demonstration.

In Figure 7.3, we show two panels of the user-side interface. In the first panel (called "information panel", see the left part of Figure 7.3), we detail all the information related to the user's ST over the time. In particular, we indicate the status information of the user's

device in the top-left corner. The statistics related to the computation and communication costs are shown in the top-right corner of this panel. We also propose a tracing window in the lower part of this panel in order to print all the processed information in the user's ST. The interface provides four buttons to deal with the running modes (i.e., run, step-by-step, pause and next). Furthermore, in order to better illustrate the partitioning and the traffic aggregation to the audience, the interface also offers a "Graphics" panel. In this panel, we show on a map the information about the user as well as the aggregation information of the current partition. This panel also displays the information about the traffic volume or the average speed of vehicles in the current partition by using a colored heat map (see the right part of Figure 7.3). We use four colors (red, orange, yellow and green) to indicate the current traffic status on a road: very dense, dense, normal or sparse respectively (the gray color corresponds to the roads outside the current partition).

The graphical interface at the server-side shows the detail of the operations executed by the SSI as well as global statistics of the secure protocols. The aim of this interface is twofold: (i) to show how our system works in parallel as well as the costs of the protocols; (ii) to demonstrate how secure our method is w.r.t. the SSI. The interface allows changing the network map by selecting a new map file and also launching the partitioning algorithm in order to demonstrate the partitioning process. The overall costs of our aggregation protocol for a time window is also displayed in the interface. A tracing window and control buttons with the same functions as in the user-side interface are equally present. The only difference is that we trace the messages that are exchanged between the STs and that transit in the SSI. However, since all the messages are encrypted, the printed contents are only cipher texts

7.3.2 Demonstration scenarios

In this section, we detail three main scenarios of this implementation.

7.3.2.1 Location updates

This scenario corresponds to the collection period and details the updates at both the user-side and the server-side. In this scenario, we demonstrate the subscription of four tokens that represent for four distinct users. After subscribing to the system, these users send update messages to the SSI. The content of the messages are printed in the information panel in the user interface, while the graphic panel shows the user's current location on the map. At the server-side, the cipher text of the update messages is printed in the tracing window together with several statistics of the update protocol (i.e. execution time, transfer time, etc.). Furthermore, in the test, we allow the audience to run this scenario in step-by-step mode in order to see each snapshots of the collection period.

7.3.2.2 Secure aggregation protocol

The purpose of our second scenario is to show in detail the way our system calculates the traffic density and delivers the results. In this scenario we run the calculation processes on four different secure tokens acting as four different users located in different network partitions. First, the SSI groups the messages by the partition cipher-text value, then she sends the data to four users in parallel. The trace messages are shown in the tracing window of the interface at the server-side. The overall costs are also recorded in the top-right corner of this panel. Meanwhile, the information at the user-side for the four users are shown in the corresponding interfaces. In each of these interfaces, the tracing window shows the current status of the user's ST, while the time consumption and the size of the sent/received data are shown in the top-right corner. Similar to the previous scenario, we also allow the audience to execute the program in the step-by-step mode and see snapshots at any point in the execution. Once the process is finished, the final results are reported both in the tracing window, and on a heat map in the graphic panel, to display the traffic volume in the partitions. Note that the demonstration scenarios use simulated traffic data for a large number of users (i.e., up to millions of users) and not only for the four STs.

7.3.2.3 Road network partitioning

The last scenario demonstrates the network partitioning process. Concretely, we use one secure token in order to compute repartitioning process. At first, aggregate results of the entire network are sent to this secure device by the server (i.e., SSI). Then, repartitioning process will be launched in this device. Information related to repartitioning (i.e., messages sent and received) is also shown on both tracing windows of ST and SSI as the further explanations for the audiences. Finally, the overall costs and final results of this process are shown clearly in ST's graphic interface. While, on SSI's side, only the cipher texts are printed, since those are all information could be seen by SSI. We also validated this scenario with different kinds of network maps and datasets (i.e., traffic distributions).

7.3.3 Demonstration results

The objective of this demonstration is to convince the audiences about the following important characteristics of our system:

Privacy and accuracy: As could be seen in these scenarios, any efforts to read participant information from SSI is prevented. Information shown in the ST's interface is to illustrate what happen in secure devices and will not be seen even by the owners of secure devices. Additionally, participants (i.e., STs) could detect easily if the SSI drops too many messages in order to bias the final aggregate results. The explanation is that, participant could know

in advance approximately the amount of messages he should receive from SSI. For this reason, accuracy is also guaranteed in our system.

Real-time execution: even though the system runs in an environment having very limited resources, the demonstration shows that the aggregation process could be obtained in a very short time (i.e. milliseconds to a few seconds), which satisfies the real-time constraints.

Scalability: the further demonstrations with large datasets (i.e. up to millions of users) shows that our system is scalable with respect to all sizes of networks.

7.4 Experimental results

7.4.1 Demonstration settings

Target query. In this demonstration we mainly focus on counting number of vehicles in each road segment. However, any other functions such as calculating average speed, max, min speed or median function could be applied as well. The following query explains detail our target.

SELECT	**Count**(*) – could be **AVG**(speed) or any function
FROM	LocationStream
	[WINDOW x seconds SLIDE x seconds]
GROUP BY	Road_Id

Input Data. Without loss the realistic, we create synthetic datasets to evaluate our systems. We used the well-known Brinkoff generator [17] to generate testing data on Stockton (i.e., a real road network at San Joaquin Country, CA covering nearly $200Km^2$). The dataset contains 1 million of different users and around 110 000 users at the same time send update to server continuously over 1000 time units. These upload messages from users were generated and stored on the server. In order to simulate the system at the run-time, we first pick data at a certain time unit t (e.g., 100th time unit), and select all the update messages belong to the interval $(t, t+1)$ (i.e., the messages that come from around 110 000 users). These messages, then, are considered as the real messages obtained from the real-users at running time.

7.4.2 Results

In this section, we show the results of our demonstration which is related to the time consuming of all the duration in our system (i.e., collection period, aggregation period

and re-partition period). The following table details the costs of these duration.

Collection	Update generation	0.48 ms
	Transfer time	0.51 ms
Aggregation	Processing time	0.58 s
	Communication time	0.194 s
Re-Partition	Processing time	4.2 s
	Communication time	1.7 s

Table 7.1: System performance

Collection period. In order to measure the time consuming for the collection period, we run the application on four different secure tokens that act as four different vehicles moving on the road and generating the upload messages. The results in the *collection rows* in the table 7.1 are obtained by measuring the time consuming for both generating upload message and transfer time on these four tokens over 10 times and calculate the average value and the largest values were reported on the table. As expected, the total time spends for the collection period is very small (i.e., around 1 ms). The explanation is that the collection periods are done in parallel by all participants. Each of participant only need to generate one message and send it to server in encrypted form.

Aggregation period. In aggregation period, instead of computing all the partitions we selected four largest partitions and calculate the aggregation. The longest time among four results is selected and reported at the final result. We also repeated the test with more than 10 times over different partitions and calculate the average time consume for this period. The result on table 7.1 shows that total time for this period is smaller than 1 second. This means we could have an online-traffic monitoring offering the data updated in real-time (i.e., every 1 second).

Re-partition period. Finally, table 7.1 also shows the time consuming for the re-partition process. Even though the total time spends for this process is less than 6 seconds, re-partition could be loaded very several minutes, since it doesn't make sense to launch this process too frequent (i.e., the changes of vehicle distribution on the network is not enough to re-partition). An example of partitioned network (Stockton) is shown in Figure 7.4. In this example, the algorithm is applied to re-partition network into 100 groups which were marked by 9 distinct colors.

Figure 7.4: An example of partitioned network

7.5 Conclusions

In this Chapter, we propose an efficient, privacy-preserving solution for online traffic monitoring by applying our PAMPAS protocol (see Chapter 5) in the context of traffic monitoring. Our system implements a user-centric architecture that matches the security, real-time and scalability requirements of the online traffic monitoring systems.

Chapter 8

Management of Spatio-temporal Data Streams in Embedded Systems

In this Chapter, we present and detail our implementation of an extension of an embedded relational database engine (i.e., PlugDB [82]) to deal with spatio-temporal data. This extension consists in new data types and new operators allowing to locally manage spatio-temporal data in conjunction with the classical data types and operations in the relational model. In particular, we introduce the context in section 8.1. After that, we discuss some related work and background knowledge in sections 8.2 and 8.3 respectively, while the scenarios and queries are presented in section 8.4. In section 8.5 describes our implementation and analyses experimental results of this work. Finally, section 8.6 concludes this Chapter.

8.1 Introduction

8.1.1 Motivation

As discussed in the previous Chapters, integrating mobile technology and positioning devices has led to producing large amounts of spatio-temporal (ST) data every day on the user mobile devices. The generated ST streams are meaningful for many applications such as traffic analysis, customized insurance (pay-as-you-drive), reconstructing the circumstances of an accident in road safety, mobility and population exposure analysis, etc. ST streams may be saved for legal reasons, e.g. to provide an evidence for the insurance, or for individual use, e.g. experience sharing, individual gas consumption monitoring or eco-driving. However, as the best of our knowledge, there is no solution adapted to the

context of mobile devices.

Therefore, it's reasonable to manage the personal ST data and location traces locally. In this Chapter, we address this issue by promoting the use of a trusted Personal Data Server (PDS) for personal data protection, including location data. Many works on embedded databases have been conducted [91], [15], but dealing with the very constraint requirements of embedded devices such as RAM or Flash technology is still a unsolved problem for Spatio-temporal data. Consequently, the requires designing a suitable spatio-temporal data model for the PDS constrained environment. Designing such a data model consists typically in the definition of a set of embedded data types and the definition of basic operators on those data types. These aspects are also developed in this Chapter. Note that, the term PDS used in this Chapter is similar to secure token (ST) in previous Chapters.

8.1.2 Requirements

Our main contribution is to extend the relational database engine embedded in the PDS with ST data stream management capabilities. The main obstacles are the particular features of ST data streams (i.e., big size, high frequency) which are hard to manage in the context of PDS environment. To deal with such complex data, the system should support many kinds of new data types. However, in this Chapter, we present only a few representative data types. The design is based on an algebra for moving objects proposed in a previous work [94], [43], [44], [45]. In addition, since most of the application scenarios are based on constrained moving objects (i.e., objects that move in a transportation network), we focus on this type of movement in this work. Hence, in this Chapter we focus on the implementation of two fundamental data types, i.e., MGPoint and GLine, and some of the related operations. The remaining part of the algebra should be implemented later.

8.1.3 Contributions

The main objectives of this work are:

Implement new data types. We extended the new data types of spatio-temporal data that allow to describe a moving position and a trajectory in a road network. For example, we can use MGPoint to represent the position in time of a car on the road or use GLine to illustrate the trip of a car. However, these ST data types (i.e., MGPoint and GLine) are both complex and large size data types. The size of an MGPoint object depends on the length in time and in space of the trajectory of the moving object. Therefore, an instance of these data types may require many sectors in NAND Flash to store. Hence, loading such a data object in NAND Flash may cost many I/Os. In addition, since large data objects may be stored fragmented on Flash, this may also increase the cost of the object retrieval.

To reduce the access time to Flash, we propose to store continuously these data objects in Flash by using a new dedicated data structure adapted for storing spatio-temporal data. The proposed data structure (called STOB, i.e, Spatio-Temporal Object) consists in two parts. The first part is used to store object description information and has a fixed size similar to the classical DBMS data types. The second part has a variable size and is used to store continuously in NAND Flash the real data. The detail of this implementation is discussed in Section 8.3.2.

Implement new operations. The size of processed data is large in comparison with the RAM memory of a PDS device. Thus, for handling new operations (e.g., Trajectory and MCount) that take extended ST data types as the input data, we have to divide the input data into many small parts (called packages) and then evaluate the operation results package by package. In this way, the main memory consumption can be limited. The downside is that this approach may take much time for processing the result because the number of accesses to NAND Flash can be high. Alternatively, we propose another approach that allows evaluating the operations directly on the streams generated by the input objects when they are retrieved from Flash. These two solutions for operation processing using buffered evaluation or stream evaluation are presented in Section 8.3.2.

8.1.4 Outline

This rest of this Chapter is structured as follows. We first present related work and background knowledge related to this context in Sections 8.2 and 8.3. The scenarios and queries implemented in this Chapter are discussed in Section 8.4. Section 8.5 details our work related to implementation as well as the experimental results. Finally, we conclude this Chapter in Section 8.6.

8.2 Related work

Our work in this part is concerned to Spatio-temporal databases which have been studied for more than decades such as [43], [44], [45]. Indeed, Spatio-temporal databases are the combination of spatial databases which tackle problems of data in two dimensions space and temporal databases dealing with the changes of data in time. Concretely, the study in [42] gave the definitions of spatial database system where most of the aspects of this database were solved. This provided spatial data types, query language, data representation as well as spatial indexing and spatial join methods. Following this work, Guiting et al. then defined in [43] a abstract model where base types, spatial and temporal data types were defined as well as some operations. While [44] and [45] completed this framework by modeling the networks and providing data types for static and moving objects. This framework proposed a set of basic data types and complex types that could

be built from these basic data types. Since, it's still a challenge to adapt this work to specific environment such as PDS, this work follows the set of abstract data types (ADTs) and collection of operations proposed in [43] and [44] (called *algebra*) and apply this *algebra* in the context of embedded environment.

8.3 Background knowledge

In this section, we present some background knowledge about data types and operations that we selected to implement in our system. In particular, we chose GPoint, GLine and MGPoint that are used to define respectively a position in the road network, a line in network and a position in network varying in time. Actually, these data types are initialized in [43], [44]. We also introduce the new operations to apply on MGPoint (i.e., Trajectory), on GLine (i.e., MCount) and an aggregate operation on the number data type (i.e., MSum).

8.3.1 Data types

We introduce in this section three basic spatio-temporal data types that are required to store a location in the road network (GPoint), a trajectory (or route) in the network (GLine) and a spatio-temporal trajectory (i.e., the evolution in time of the position of a moving object) of a vehicle (MGPoint).

8.3.1.1 GLine

GLine is a set of disjoint route intervals used to store a line in the network space [5], [44]. Let $N = N_1...N_k$ be the set of networks present in database. GLine data type is defined by following formula:

$$D_{gline} = \{(i, gl) | 1 \leq i \leq k \wedge gl \in Reg(N_i)\} \tag{8.3.1}$$

Where gl is a *gline* identifier and Reg is the set of all possible regions of a network N_i

Figure 8.1 gives a simple example of *gline* data. This *gline* instance contains 4 road intervals (red lines). Each of intervals has 2 positions: $pos1$ and $pos2$ which belong to the same road.

Figure 8.1 gives a simple example of *gline* data depicted as the red polyline. This *gline* instance contains 4 road intervals (red lines). Each of the intervals has 2 positions: $pos1$ and $pos2$ which belong to the same road. Therefore, a *GLine* is a sequence of disjoint route intervals, i.e., $gl =< (rid_1, pos1, pos2, side), (rid_2, pos1, pos2, side) \ldots (rid_n, pos1, pos2, side) >$,

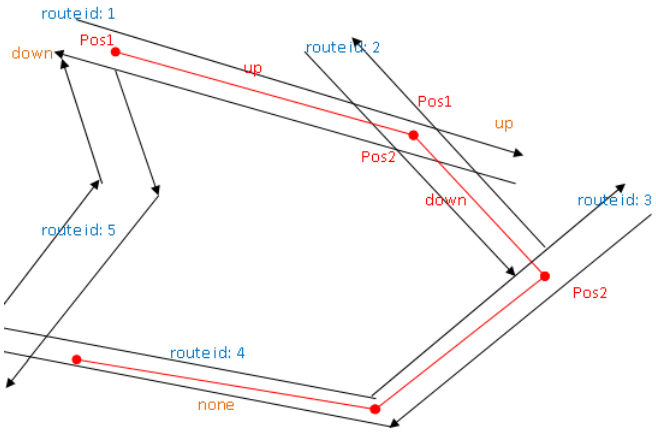

Figure 8.1: An example of GLine

where rid_i is a road identifier and *side* indicates the side of the road that can be *up* or *down*.

8.3.1.2 GPoint

GPoint data type is usually used to represent information of a position on network. Let $N = N_1...N_k$ be the set of networks present in database. *GPoint* data type is defined as:

$$D_{gpoint} = \{(i, gp)|1 \leq i \leq k \wedge gp \in (Loc(N_i) \cup \perp)\} \tag{8.3.2}$$

An instance of *gpoint* is defined as a couple of two values. The first is the identifier of network which position belongs to, and the second is the network location. A network location is defined as $gp = (rid, pos, side)$, where rid is a road identifier, pos is the relative position on the road and side is the side of the road. Loc is set of locations in network N_i and $\perp$ for empty set.

An Example of *gpoint* is shown in Figure 8.2 where the position belongs to road 1 and is on the downside.

8.3.1.3 MGPoint

The *MGPoint* data type is used to describe the position of a moving object that continuously varies in time [43]. In general, a moving value data type can be defined as:

$$A_{moving(\alpha)} = \{f|f : \overline{A}_{instant} \rightarrow \overline{A}_\alpha \, is\, a\, partial\, function \wedge \Gamma(f)\, is\, finite\} \tag{8.3.3}$$

Here, the moving data type is defined as all $fvalues$ that belong to a partial function from time to α with a special condition "$\Gamma(f)\, is\, finite$" to indicate that the number of

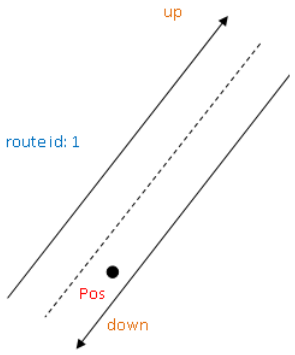

Figure 8.2: A position in network

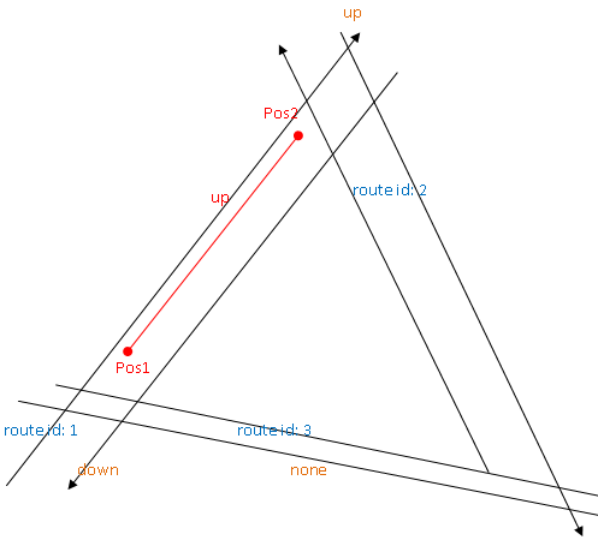

Figure 8.3: An example of Mgpoint unit

continuous components of this function is finite. α can be a typical data type (e.g., integer, float) or it can be a complex data type (e.g., *GPoint*, *GLine*).

MGPoint is the abbreviation of moving(*GPoint*). This special data type can be used to store network position varying in time. In practice, *MGPoint* is defined as an array of *MGPoint-units*. An *MGPoint* unit is presented in Figure 8.3 and is defined as $unitMGPoint = (rid_i, pos1, pos2, side, t1, t2)$. As above, rid_i indicates the road, $pos1$ and $pos2$ are two relative positions on the road measured at two consecutive time instants $t1$ and $t2$. The length of an *MGPoint* object is the number of *GPoint* units of the object.

8.3.2 Operations

In this section, we introduce three basic operations over the extended data types. Specifically, **Trajectory** computes the spatial projection of the spatio-temporal trace of a moving object, **MCount** aggregates a *GLine* object into the number of roads contained in the

GLine object, and **MSum** is a typical relational aggregate can be applied in many kinds of data types.

8.3.2.1 Trajectory

Trajectory is an operation which receives only one *MGPoint* value as the input and returns a *GLine* value.

$$Trajectory(MGPoint) \longrightarrow GLine \tag{8.3.4}$$

Since trajectory returns the network path of a trip and *MGPoint* is an array of *MGPoint* units, the operation trajectory must be applied to *MGPoint* unit by unit. This operation takes a unit of *MGPoint* value and eliminates time intervals information from this unit value to obtain a *gLine* unit.

8.3.2.2 MCount

We also support another operation called **MCount** which takes *GLine* as an input parameter. Given a *GLine* value, **MCount** operation counts the number of roads in this input parameter. The detail of the output and input of this operation is described as follow:

$$MCount(GLine) \longrightarrow Number \tag{8.3.5}$$

8.3.2.3 MSum

MSum is a basic aggregate function which receives a number-data column and returns the aggregate sum value of this column.

$$MSum(Col\_identifier) \longrightarrow Number \tag{8.3.6}$$

8.4 Application scenarios and queries

To illustrate the use of spatio-temporal embedded data management, we consider typical queries related to the location dependent scenarios. The PDS ensures that GPS traces are safely stored for *multi-purpose usage* (e.g. Pay-As-You-Drive (PAYD), insurance or Toll pricing, trip share...).

More precisely, let's consider Bob, a driver who wants to benefit from the PAYD schema for both his insurance and toll payment, while protecting his personal data. Bob uses a PDS combined with a location-aware mobile device. The system records all his GPS

traces on the PDS. Once a month, he sends the minimal aggregated information to both the insurance and the toll service provider for billing. However, the insurance company and the Toll service provider can challenge Bob to provide his exact location Moreover, Bob selects his touristic trips and publishes them on *OpenStreetMap* or chooses to share them with his friend Alice. The insurance should verify the data correctness in case of dispute or fraud suspicion directly or via a trustworthy authority.

Such queries and computations require that the embedded DBMS in the PDS is capable of managing spatio-temporal data, whence the necessity of an adapted spatio-temporal data model, query language and access methods. Based on this, we propose a benchmark comprising a typical database schema and queries. We use an extended relational model. We manage to distinguish traditional relations (using conventional data types) from the spatial and spatio-temporal relations (using the extended data types defined in the previous section).

The database schema is as the following:

Vehicle (	VehicleId:	int
	VehicleType:	int
	VehicleNum:	char (20)
	VehicleDesc:	char (50)
	trip:	mgpoint
)		
ForbiddenRoad (	VehicleType:	int
	Sections:	gline
)		
Road (	Rid:	int
	Name:	char (50)
	Fee:	int
)		

The first relation reports the trips of a driver, defined as a Moving *GPoint*. The second illustrates the use of *GLine* (sections refers to the part of the road that is forbidden for a given type of vehicle). The third relation is a lookup table describing the fee by road (e.g. used by the insurance or the toll service provider).

The purpose of this benchmark is to evaluate the implementation of the new data types and their basic functions. It does not target the set operations like selection and join,

which will necessitate further optimization techniques, such as tailored indexes. Indexing spatio-temporal data will be addressed in the future work.

Hence, we devise some following types of queries:

- Projection on attributes of the extended types, but without any function
- Projection involving a single function on the spatio-temporal data, which returns numerical results
- Projection involving a single function on the spatio-temporal data, that returns spatio-temporal results
- Projection involving a combination of functions on the spatio-temporal data.
- Projection involving an aggregate function.

Question 1: How has the car (id: 4) travelled?	
SELECT	Trip
FROM	Vehicle
WHERE	VehicleId

Question 2: What is the journey of the car number 3456?	
SELECT	**Trajectory**(Trip)
FROM	Vehicle
WHERE	VehicleNum = 3456

Trajectory is an operation that returns a *GLine* value for a given *MGPoint* parameter (*Trip*).

Question 3: How many roads don't allow car (type = 1)?	
SELECT	**MCount**(Sections)
FROM	ForbiddenRoad
WHERE	VehicleType = 1

This query uses **MCount** function, which returns the number of road ids composing a given *GLine* parameter (the paying road section).

The above query uses a combination of functions: **MCount** and **Trajectory**.

Question 4: How many roads has the car number 3456 travelled?	
SELECT	**MCount**(**MCount**(Sections))
FROM	Vehicle
WHERE	VehicleId = 3456

Question 5: How much does a vehicle have to pay if it travels all over the roads of network?	
SELECT	**MSum**(Fee)
FROM	Road

This query used an aggregate function **MSum** to calculate the total fee from table road. In summary, we have some different kind of queries that can be used to solve some different needs of applications when retrieving data from PDSs. Based on these kind of queries, we designed a set of queries that are used for testing our implementation. These queries are used in Section 8.5 for testing. The detail of these queries could be found in Section 8.7.

8.5 Implementation and experimental results

8.5.1 System architecture

Our Secured Personal Tokens (SPTs) or Secure Personal Devices (PDSs) can be appeared in many form of devices: smart phones, USB sticks, or smart card. In all cases, the architecture of our system always includes hardware and software parts (see Figure 8.4) [115].

Hardware environment

The hardware characteristics of PDSs are slightly difference between PDSs and may be changed in future, it's depended on the version and the cost of devices. PDSs could be a mobile phone, a smart badge, a smart device in a vehicle, a USB stick or some other kind of personal devices and in each of case PDSs can have some different characteristics in shape, size, cost and hardware configuration. However, in general all PDSs have a microcontroller which includes a 32-bit Reduced Instruction Set Computer (RISC) microprocessor, RAM (around 64KB), internal NOR FLASH (about 1MB), various communication modules in order to communicate with outside components (e.g., USB 2.0, Bluetooth, 802.11, SPI...) and some other peripheral modules. Specially, this microprocessor is connected to a very large NAND FLASH storage (GBs of NAND Flash) allows PDSs to store much data [9], [7]. Nevertheless, having limited resources and using external memory for storing data in PDSs bring us many challenges.

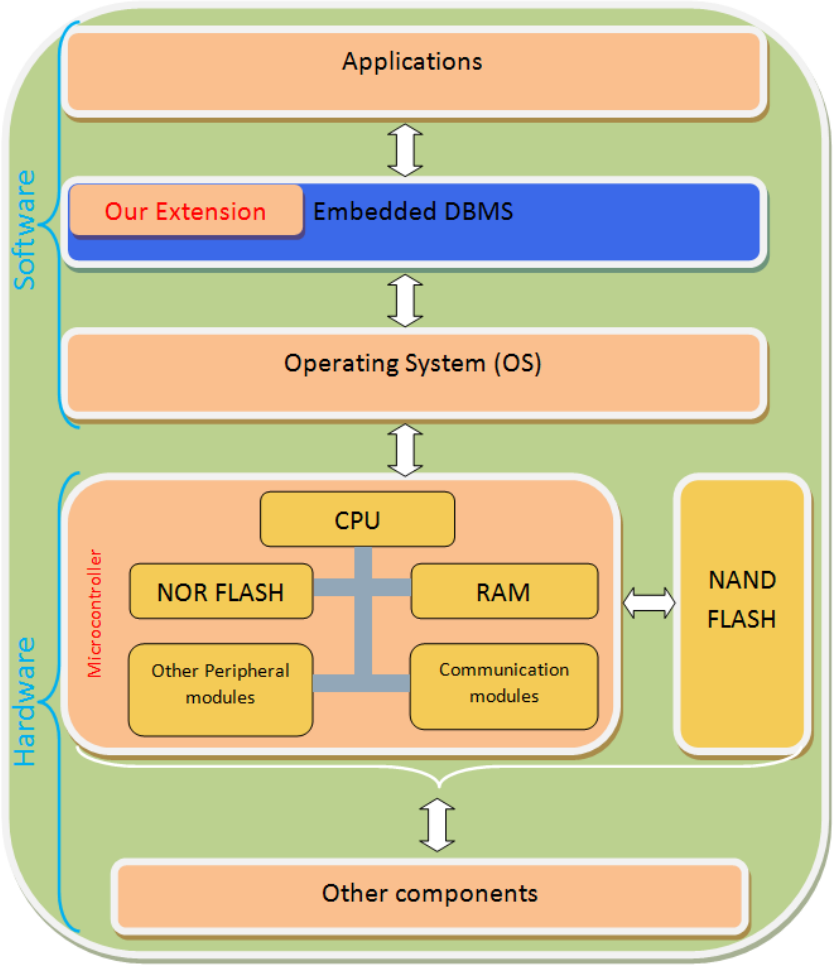

Figure 8.4: System architecture

First of all, small capacity of RAM memory is able to slow down the system when processing or dealing with new type of large data such as: Moving GPoint (MGPoint) or GLine data. The limited capacity of RAM doesn't allow us to load all instances of these kinds of data into main memory. Hence, it consumes more time for all of the processing these data.

Secondly, throughputs between external storage and microprocessor are limited and slow (compare to internal memory). Storing data incoherently in NAND Flash may increase the number of accesses and take more time when retrieving data. Besides, accessing to NAND Flash memory many times not only delays throughput but also makes harmful to NAND Flash because the number of read/write times from/to NAND Flash is also limited.

Software environment

The software of system contains three small parts: application, DBMS and operating system (OS). However, application level is user level while operating system manages and monitors hardware devices both of them are not related to our studies. Therefore, in this

section we only focus on DBMS module.

DBMS is a software package that controls the creation, management and use of database in PDS. DBMS itself has many modules, each of modules has specific task. For example, DBMS manages hardware resources such as RAM, Nor Flash, Nand Flash... by using HwApi module. And NetDBMS is a TCP/IP communication module that allows DBMS to connect to other devices for sharing data.

On server's side, there is another application for communication with PDS devices. In particular, we used JDBC (Java Database connectivity) module which acts as a server to take data from PDS. JDBC is also a software package that also has inside TCP/IP or JNI bridge module to enable connections to DBMS for creating database, sending or receiving data. Moreover, JDBC uses some application programming interfaces (API) which are provided by DBMS. Thus, it can provide methods for creating, querying and updating database in DBMS. And, these methods can be used by some testing modules inside JDBC.

To reduce query processing that may consume much time and memory usage of PDSs, all the SQL queries are pre-compiled into Query Execution Plan form (QEP) before sending to DBMS. QEP is an order set of operations that are used to describe the meaning of SQL queries. This plan can describe exactly how a SQL query will be executed. In our system, we use QEP to perform all the SQL queries. And, QEPGen and QEPGenerator are two versions of QEP compiler that are used to transfer queries from SQL format to QEP string.

8.5.2 Implementation

In the previous Chapters, we discuss about the demands to extend the current DBMS and some briefs about new kind of data types and operations as well as the solutions for the very constraint system. This Chapter presents detail about system architecture, our solutions and implementation.

8.5.2.1 Overview

Here we recalled the definition of DBMS, JDBC and QEP compiler. However, we don't present detail about these modules. The detail descriptions of these modules can be found our technical report [112].

- DBMS, a software package, is downloaded in PDSs and used for controls database in PDSs. DBMS plays the role to monitor PDSs hardware resources, receive requests from servers (JDBC), execute the requests then send feedback to servers.

- JDBC (Java database connectivity) is also a java software package but can be placed in servers. It uses TCP/IP communication to connect to DBMS for sending requirements and receiving information from PDSs.
- QEP compiler is a compiler used to pre-compiler all the queries in SQL to QEP code (Query Execute Plan code). QEPGen and QEPGenerator are two versions of QEP compiler that we used in our implementation. QEPGenerator is the old version while the other is the new one.

Implementation of our extensions is a work related to not only DBMS but also JDBC and QEP compiler (see Figure 8.5). Because queries used between servers and PDSs should be in QEP code (Query Execution Plan code) to reduce the CPU and memory resources in PDSs. Therefore, adding new queries for new data types and operations, we have to have new QEPs code. In addition, new keywords used for new data types and operations must be defined in QEP compiler level. In other side, DBMS and JDBC have to have new functions to handle new data types and operations.

Figure 8.5 shows the general view of how our system works. First of all, SQL queries and file description of database relational schema must be compiled into QEP string by our QEP compiler (QEPGen or QEPGenerator). The output of QEP compiler (in QEP format) is a queries.java file and schema.src file. Queries.java, a file in java, contains all SQL query prototypes (in QEP format) that we want to use for our system. While schema.src is a set of QEPs string that is used to define database model.

Once received queries.java and schema.src, they should be integrated into JDBC project. Based on all the QEPs string provided by queries.java and schema.src, JDBC can compose queries (in QEP) then send them to DBMS for creating database, getting data from DBMS or modifying data in DBMS. Receiving the requirements (in QEP) from JDBC, DBMS executes the queries then sends feedback to JDBC by TCP/IP communication.

To add new data types and new operations, we have to make some modifications from both QEP compiler, JDBC and DBMS. The steps in red color in Figure 8.5 are all the steps we have to modify so that the system can understand new kind of data types and operations. In particular, these steps are:

QEP compiler level:

1. Modify parser.lex in order to recognize new kind of data type and operation keywords.
2. Explain semantic meaning of new operations and data types by changing parser.y
3. Describe database relational schema (in SQL) in schema.sql
4. List all the queries (in SQL) that may be used in our system in queries.sql

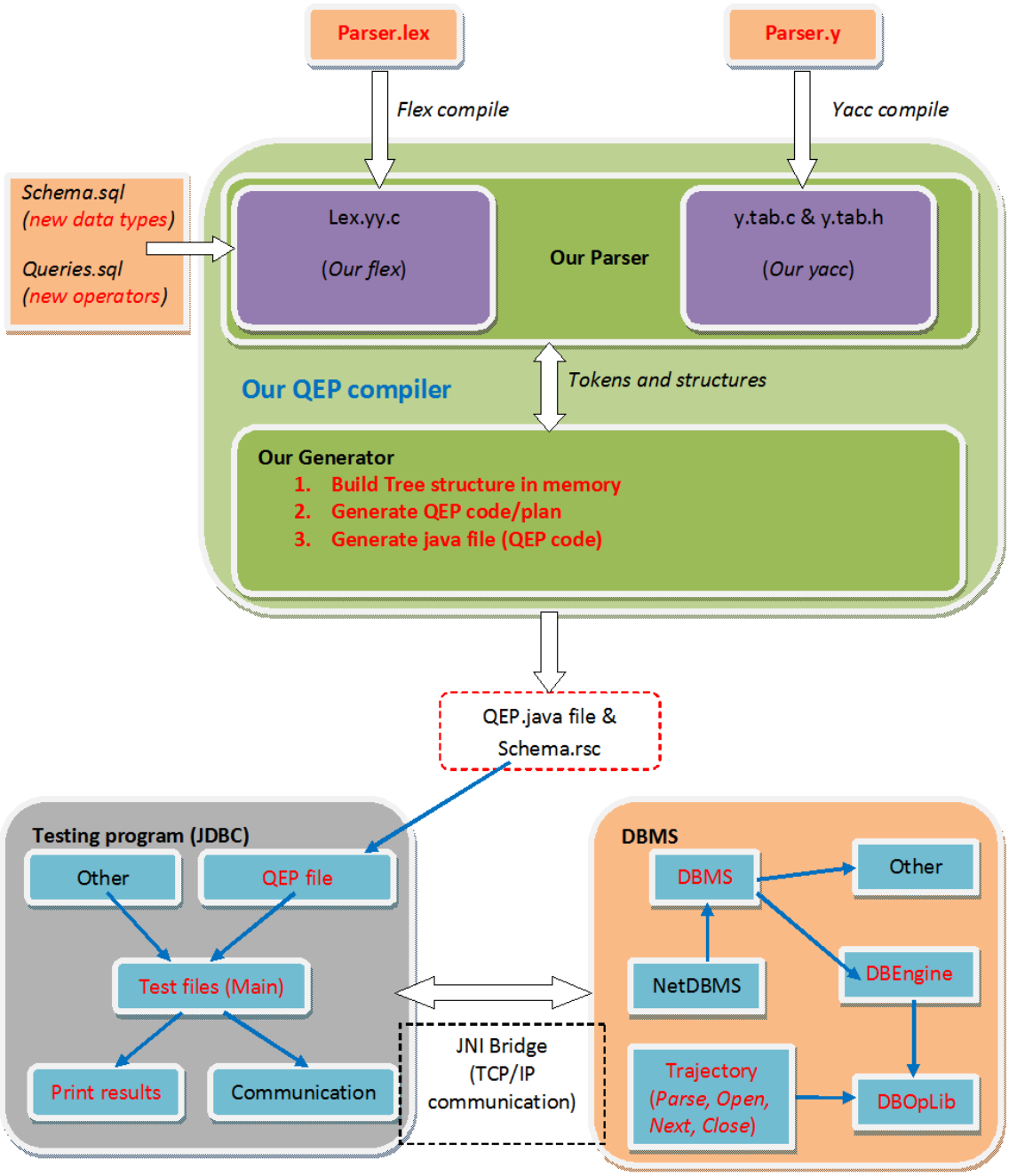

Figure 8.5: The overview of implementation process

5. Design new QEPs code for new operators

6. Modify QEPGen (or QEPGenerator) so that QEP compiler can build and generate QEP code for the new queries (recognize new queries, build queries plan (tree) in memory, generate QEPs code)

DBMS level:

8. Design new structure and new/get/set methods for new data types
9. Make some small changes in DBMS and DBEngine module in order to understand new QEPs code
10. Implement new operator functions to handle new operator in DBOpLib module (add new handle functions: parse, open, next, close)

JDBC level:

11. Update the new QEPs (new data types and operations) code for JDBC
12. Build new testing program for testing new kind of data types and operations
13. Compose new function for receiving and handling new kind of data types

8.5.2.2 The storage model

As we discussed in previous section, *MGPoint* and *GLine* are large size data types which length is variable. They also have an intrinsic semantic that should be captured by the data structure. In this section, we describe the physical implementation for *GLine* and *MGPoint*. In order to cope with the length variability, the data are split into small units. Thus, a *GLine* is an array of road intervals while an *MGPoint* is an array of *MGPoint* units (i.e. *UGPoint*).

GLine:

```
GLine record {
    rints: DBArray of record {                          //set of intervals
                rid:        int                         //road id
                side:       {up, down, none}            //side
                pos1:       real;                       //interval positions
                pos2:       real;
                };
        }
```

The units' structure of these data types can be described by following figure (see Figure 8.6):

Referring to the above structures of *GLine* and *MGPoint* unit, we need 21 bytes to store a unit of *GLine* (4 bytes for storing *rid*, 1 byte for *side* and 16 bytes for the two relative positions), and 37 bytes for a unit of *MGPoint* (see Figure 8.7).

MGPoint Unit:

```
UGPoint record {
        rid:        int                 //road id
        side:       {up, down, none}    //side
        t1:         instant;            //time interval
        t2:         instant;
        pos1:       real;               //interval positions
        pos2:       real;
    }
```

MGPoint:

```
MGPoint record {
        ugps:    UGPoint [LENGTH];    //set of UGPoints
    }
```

Route id	side	Pos1	Pos2
4	1	8	8

UGLine

Route id	side	Time1	Time2	Pos1	Pos2
4	1	8	8	8	8

UGPoint

Figure 8.6: Data units of MGPoint and GLine

To implement a collection (here an array of units), different alternatives exist. It could be a chained list or a contiguous array. The contiguous storage is more suitable to Flash memory. At this end, we define a new data structure called STOB.

STOB (Spatio-temporal Object) is a collection of binary data that is stored in a single entity. This structure can be used to store images, audio files, etc. A STOB is composed of two parts. The first part is a metadata, called STOB descriptor. And the second is used to store actual data, called STOB data[1].

STOB descriptor contains 3 fields (Figure 8.7): head field, size field and hash field. *Head* field has 4 bytes and indicates the physical address of STOB data in Flash memory. *Size* field (4 bytes) gives the length of STOB data. And, *hash* field (8 bytes) is reserved

[1]This storage model can apply to any Binary Long Object (BLOB).

head	size	hash
4	4	8

Figure 8.7: STOB descriptor

unit 1	unit 2	...	unit n
size	size	...	size

Figure 8.8: STOB data

for future use. This field could be used for storing hash number to retrieve a STOB data.

STOB data is a long variables length data units stored contiguously (Figure 8.8). STOB descriptors and STOB data are stored separately in the Flash memory. For retrieving actual data, first we have to read the STOB descriptor to get metadata and then based on this metadata we can access the STOB data.

8.5.2.3 The processing model

Processing a large size data in the PDS poses a problem since the small capacity of RAM memory prevents loading them. Thus, a solution for handling the new functions, such as **Trajectory** and **MCount** operations, consists in processing them in pipeline. Hence, the data are loaded and processed unit by unit, instead of entirely.

Another problem comes from the fact that the temporary results may be too large to be stored in main memory. In addition, the buffer size of TCP/IP packet (between the application level and the DBMS) is also limited. To cope with this problem, we divide the input data into many small parts (up to 256 bytes) and then evaluate the operation results by portion, which limits the main memory consumption. We divide the implementation into two classes: stream-based and buffer-based implementations.

Buffer based implementation (see Figure 8.9): In this solution, the DBMS keeps a (large size) temporary segment in NAND Flash to buffer intermediate results of the operation. This solution can take a longer time at the beginning of the operation processing, but once the temporary data is stored on Flash, the consumed time to respond to the server is constant.

Stream data (see Figure 8.10): For some operations, the results can be processed in stream and buffering the intermediate results in these cases is unnecessary. The DBMS reads a few small packages, it computes a part of the result and then sends back the results to the output. This process is repeated until the entire input data are consumed.

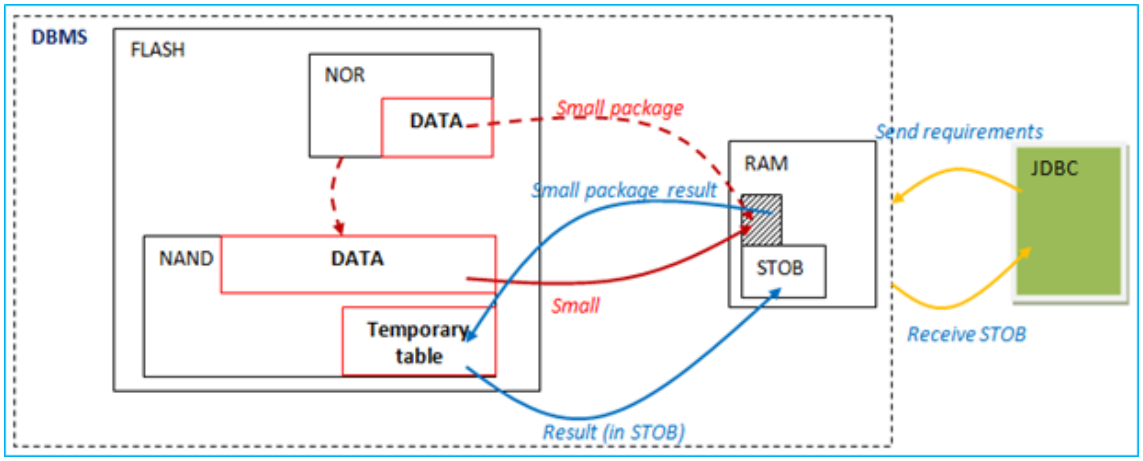

Figure 8.9: Buffer data

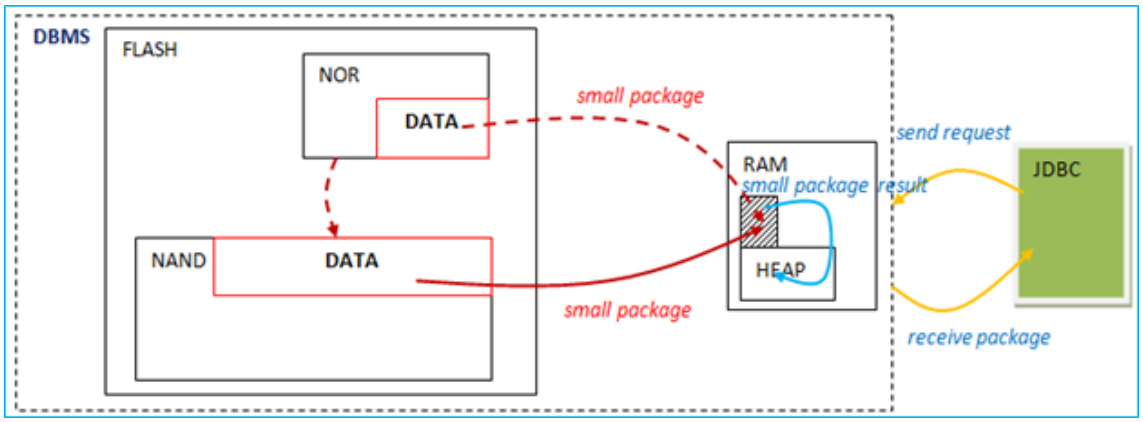

Figure 8.10: Stream data

8.5.3 Evaluation

8.5.3.1 Test case descriptions

In this section, we present the performance tests of our system and then compare result performance of buffer and streaming techniques that was described in previous sections. All of our test cases were run and simulated on Visual studio 2008 and Eclipse installed on an Intel core i5 3.1 GHz machine having 2 GB of Ram running windows XP (Virtual Machine)

The characteristic of system performance, which we aim at, is the response time of the system. Our goal is to reduce the consuming time as small as possible. On the DBMS side, the consuming time comes from CPU processing time and I/O accessing time. Because most of queries are simple or don't usually take too much execution operation. Secondly, CPU power is quite good compare to the Flash accessing time. Thus, time consumes for CPU processing is more less than time used for I/O accessing. Otherwise, the I/O

accessing time is not only depending on amount of data read/write form/to flash memory but also the number of times access to flash memory. However, in our implementation, we access directly to Flash without using Flash Transfer Layer (FTL) library. Each time we access to Flash memory, amount of data transmitted between Flash and main memory is one sector (512 bytes). Therefore, in our case, the number of I/O accesses determines the performance of system. Because of these reasons, in our performance tests we will focus on both the correctness of the implementation and the performance of system based on the number of times accessing to I/O.

In particular, we divide the set of queries used in our schema into 5 groups that are shown at the end of Section 8.4. The cases are also classified into 5 levels depending on the tested functions and the input data as follows

- Level 1 (small size of input data and single function):
  - Input data: the content of data and the size of data are fixed. The number of units for MGPoint and GLine data is up to 20 units
  - Tested function: queries are divided into groups and each of group is tested independently.
- Level 2 (medium size of input data, single function):
  - Input data: the content of data and the size for special data (MGPoint and GLine) are generated randomly. However, in this level, the upper bound of the number of units is 200 units
  - Tested function: queries are divided into groups and each of group is tested independently.
- Level 3 (large size of input data, single function):
  - Input data: the content of data and the size for special data (MGPoint and GLine) are generated randomly. The lower bound and the upper bound of the number of units are 200 units and 1000 units respectively.
  - Tested function: queries are divided into groups and each of group is tested independently.
- Level 4 (medium size of input data, multiple functions)
  - Input data: the content of data and the size for special data (MGPoint and GLine) are generated randomly. However, in this level, the upper bound of the number of units is 200 units
  - Tested function: all the queries are tested at the same time

- Level 5 (large/very large size of input data, multiple functions)
  - Input data: the content of data and the size for special data (MGPoint and GLine) are generated randomly.
    - Large size of data: 200 ÷ 1000 units
    - Very large size of data: 1000 ÷ 3000 units
  - Tested function: all the queries are tested at the same time.

Level 1, level 4 and a part of level 2 are used to check the correctness of the implementation while level 3, level 5 and a part of level 2 are used for the system performance measure. In all cases of our tests the number of tuples in each table that contains special data is always 20 tuples.

Although input data is generated randomly, the methods to generate data have some rules so that it can automatically generate the data with strict constraints. The reasons for these data constraints are come from the fact that all our operations which apply on special data are limited operations. In trajectory operation, we don't support ability to concatenate many parts of a same road and sort output result (in GLine). Therefore, the MGPoint data that we used for testing is non-overlap, sorted by both road id and time and continuous in time. Furthermore, the GLine data used for MCount operation is also sorted by road id.

Finally, in our current tests we only focus on inserting and reading. Deleting and updating performance are not tested. These functions will be tested in near future when we are implementing indexing techniques for our system.

8.5.3.2 Results

Insertion. To see the different between inserting with and without special data type, we divide the queries in our schema into 3 kinds of insert queries. One is inserting without any special data type (Query 3). The others are inserting with MGPoint (Query 1) and GLine (Query 2) data type.

Q1 (MGPoint):	INSERT INTO Vehicle VALUES (vehicleId, vehicleType, vehicleNum, vehicleDesc, Trip)
Q2 (GLine):	INSERT INTO ForbiddenRoad VALUES (vehicleType, sections)
Q3 (Basic types):	INSERT INTO Road VALUES (rid, name, fee)

Figure 8.11 shows information about the time DBMS accesses to memory for inserting data in three levels, while Figure 8.12 indicates the type of accessing to memory (including:

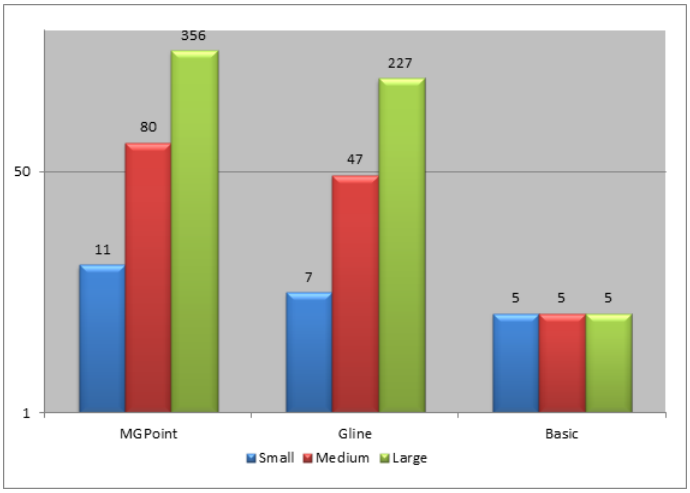

Figure 8.11: Number of I/O accesses in insert query

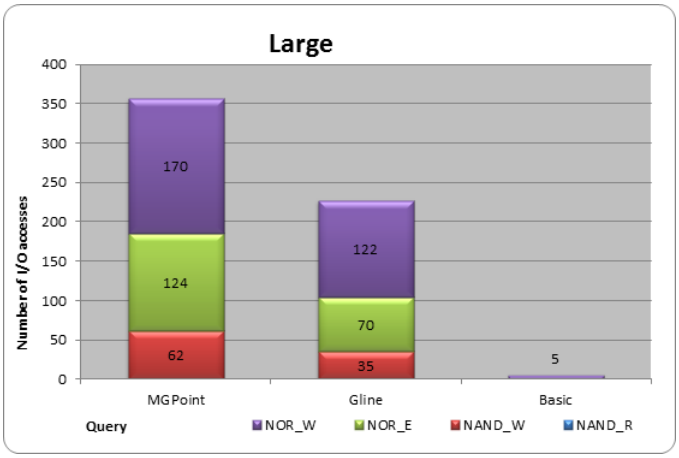

Figure 8.12: Number of I/O accesses (level 3)

NOR write, NOR erase and NAND write) in level 3 (large size of input data).

Selection. The below figures show the comparison of number I/O accesses of following set of queries between levels of performance tests. This information indicates the relation between the I/O access and the query types.

There are five queries (Q1 to Q5) which stand for five groups of queries (see Section 8.7). Because in our current implement we don't implement any indexing methods yet, we only report queries that have no where clause. We also compare the different in performance between stream and buffer techniques by using two couples of queries which are exactly the same in SQL but have different QEP codes. One is streaming and buffer trajectory QEP code, while another using buffer and stream nested operations QEP code as follows:

Query 1 is used to obtain data from DBMS. It gets many kinds of data including number (vehicleId, vehicleType, vehicleNum), character (vehicleDesc) and specially MGPoint data

Q1:	SELECT Trip, vehicleId, vehicleType, vehicleNum, vehicleDesc FROM Vehicle
Q2a:	SELECT Trajectory (Trip), VehicleId, VehicleType, VehicleNum, VehicleDesc FROM Vehicle
Q2b:	SELECT Trajectory (Trip), VehicleId, VehicleType, VehicleNum, VehicleDesc FROM Vehicle
Q3:	SELECT MCount (sections), VehicleType FROM ForbiddenRoad
Q4a:	SELECT MCount (Trajectory (Trip)), VehicleId, VehicleType, VehicleNum, VehicleDesc FROM Vehicle
Q4b:	SELECT MCount (Trajectory (Trip)), VehicleId, VehicleType, VehicleNum, VehicleDesc FROM Vehicle
Q5:	SELECT MSum (fee) FROM road

Trajectory operation:

Q2a: "# s 1 1 2 r0 0 4 # " /* Stream trajectory */

Q2b: "# t 1 1 2 r0 0 4 # " /* buffer trajectory */

Nested operation:

Q4a: "# s 1 1 2 r0 0 4 # " /* Stream trajectory */
"# i 0 0 1 1 r1 0 4 # " /* Stream MCount */

Q4b: "# t 1 1 2 r0 0 4 # " /* Buffer trajectory */
"# i 0 0 1 1 r1 0 4 # " /* Stream MCount */

(Trip). Query 2 (Q2) is tested to see the difference in the number of I/O access when buffer and stream trajectory operation are executed. Query 3 (Q3) and Query 5 (Q5) represent for MCount and MSum operations respectively. Nested operations are also covered by query 4 (Q4) where MCount and trajectory are nested together. To see the difference between stream and buffer used in nested operations, we also compare the results of them by using a couple of query (Q4a and Q4b).

The data that we used for testing is generated randomly depend on the level of tests as we presented at the beginning of this section. In level 1 amount of data is small, while the data in level 2 and 3 are quite large. In particular, the average number of units per MGPoint in level 2 and 3 are 95.5 and 614.7 respectively. Other information about date, time and ID are generated randomly. The bar charts below shows more detail about number of I/O accessing.

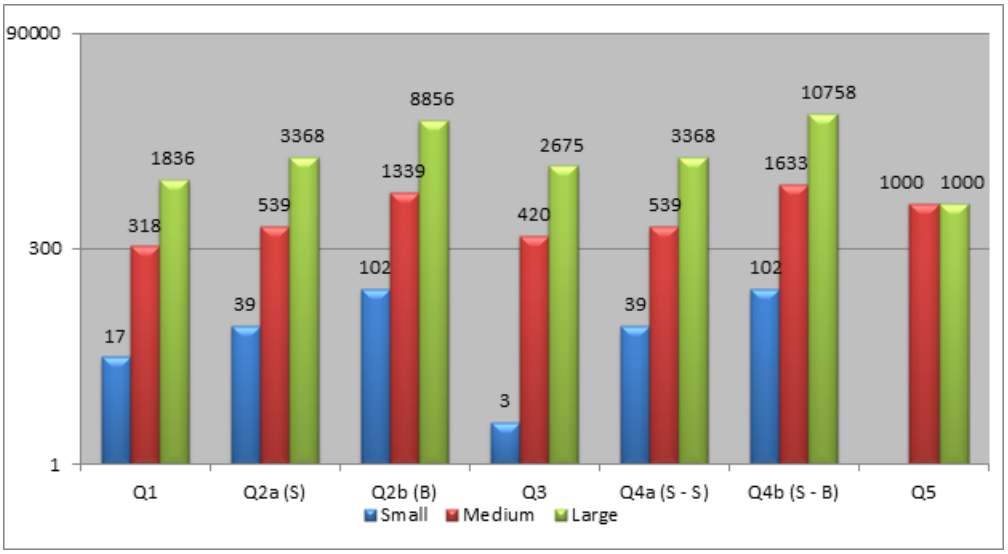

Figure 8.13: Number of I/O accesses of queries

The Figure 8.13 shows information about the number of I/O accesses of the set of queries. The x axis is the number of accesses and the y axis is the kind of queries. Each of queries, we tested in all of three levels (blue, red and green). The highest number of I/O accesses is 10758 times coming from Q4b which has nested operations (stream MCount and buffer trajectory). The detail about the kind of I/O accessing can be clearly seen in Figure 8.14, 8.15 and 8.16. In these figures we classify the I/O access into four kinds of access: NAND read, NAND write, NOR erase and NOR write. For example, the number I/O accesses in Q4b in Figure 8.14 are more than 100 times including 39 times reading from NAND and just 12 times writing to NAND.

From the charts, we can clearly see the different in number of I/O accesses between stream trajectory and buffer trajectory. When the input data is small (Figure 8.14), they are not very different (39 times versus 102 times) but both of them are the same (39 times) in NAND read. However, when the data is large (Figure 8.15 and 8.16), the cost for buffer trajectory is much higher than the cost for stream trajectory. Furthermore, while stream trajectory only reads NAND Flash, buffer trajectory has to read/write to both NAND and NOR memory.

One more thing that can be inferred from these charts is the cost for buffer nested operations (see Q4b in Figure 8.16) in case the input data is large. Its cost is very high (over 10 thousand accessing times) and three times higher than those with stream nested operations (using stream technique for both MCount and Trajectory operation).

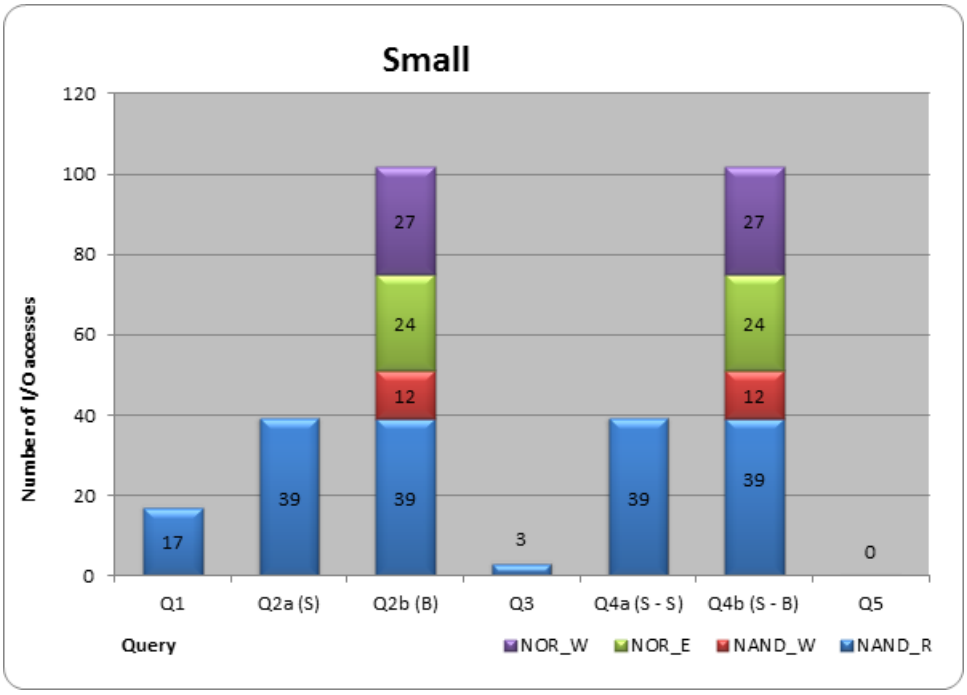

Figure 8.14: Number of I/O accesses of set of queries

8.6 Conclusion

This section has focused on the support of spatio-temporal data in the context of the PDS. We have presented the abstract data model, and pointed out the problems of its implementations due to the specificities of the underlying architecture. Then, we proposed some solutions to deal with its storage and its manipulation. This approach can be easily extended to other BLOBs.

The next steps will be the experimental validation of this model. As mentioned before, this section only addresses the basic data storage and operations (i.e. primitives). The higher level set operators, e.g. spatio-temporal selection, k-NN, etc., are also complex and costly, especially in the context of the PDS. Their optimization is mainly based on the access method. So, we will also study the problem of spatio-temporal data indexing in the context of the PDS.

8.7 Appendix

We detail five groups of queries that are used in our implementation as following. These queries are designed with respect to the scheme presented in Section 8.4.

Group 1:
Q1 : SELECT Trip FROM vehicle

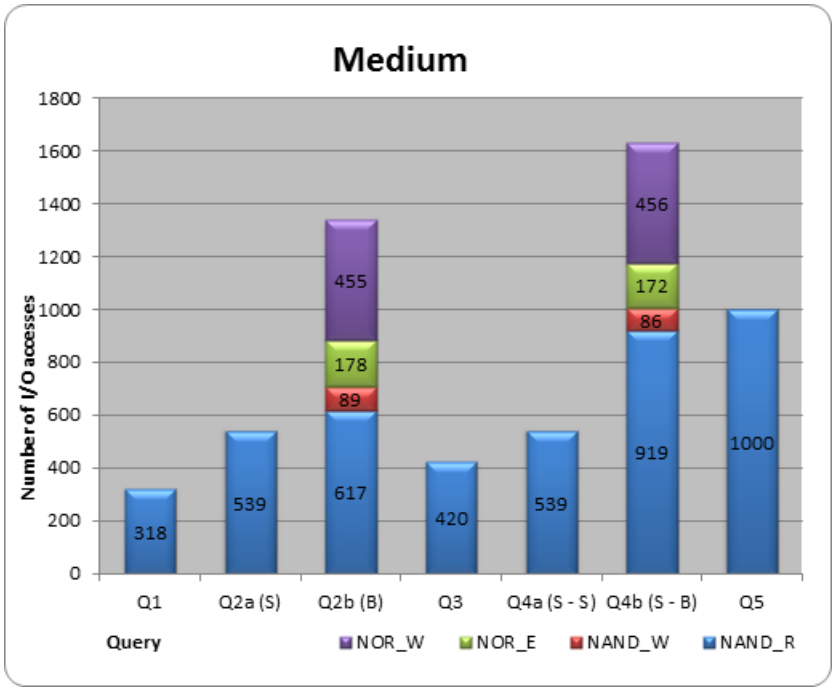

Figure 8.15: Number of I/O accesses of set of queries

Q2 : SELECT Trip FROM vehicle WHERE VehicleId = 4
Q3 : SELECT Trip, VehicleId, VehicleType, VehicleNum, VehicleDesc FROM vehicle
Q4 : SELECT VehicleId, VehicleType, VehicleNum, VehicleDesc, Trip FROM vehicle
Q5 : SELECT VehicleId, Trip, VehicleType, VehicleNum, VehicleDesc FROM vehicle
Q6 : SELECT Trip, VehicleId, VehicleType, VehicleNum, VehicleDesc FROM vehicle
WHERE VehicleId = 4
Q7 : SELECT VehicleId, VehicleType, VehicleNum, VehicleDesc, Trip FROM vehicle
WHERE VehicleId = 4
Q8 : SELECT VehicleId, Trip, VehicleType, VehicleNum, VehicleDesc FROM vehicle
WHERE VehicleId = 4

Group 2:
Q9 : SELECT **Trajectory** (Trip) FROM vehicle
Q10 : SELECT **Trajectory** (Trip) FROM vehicle WHERE VehicleNumber = 3456
Q11 : SELECT **Trajectory** (Trip), VehicleId, VehicleType, VehicleNum, VehicleDesc
FROM vehicle
Q12 : SELECT VehicleId, VehicleType, VehicleNum, VehicleDesc, **Trajectory** (Trip)
FROM vehicle
Q13 : SELECT VehicleId, **Trajectory** (Trip), VehicleType, VehicleNum, VehicleDesc
FROM vehicle

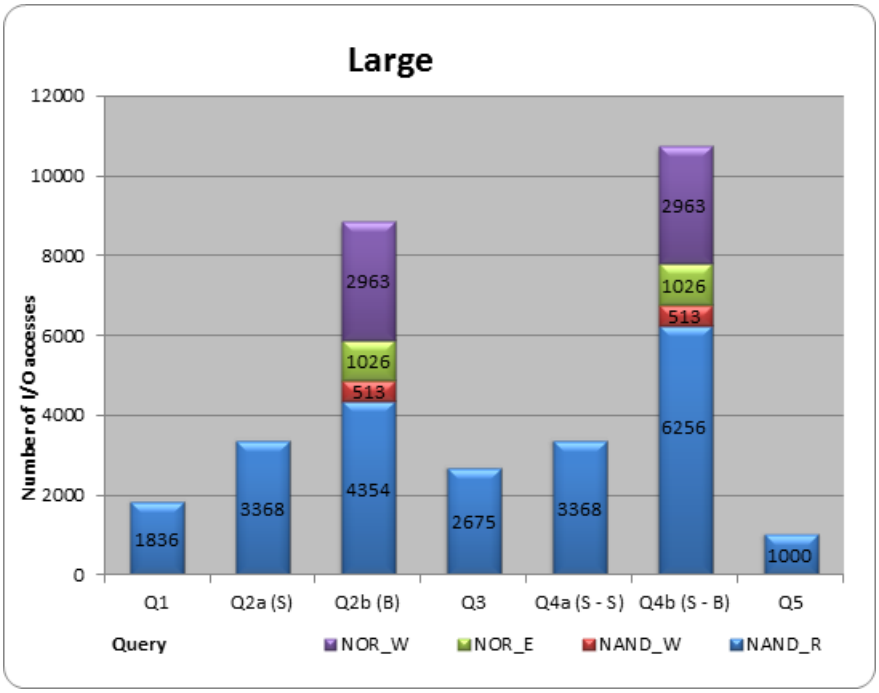

Figure 8.16: Number of I/O accesses of set of queries

Q14 : SELECT **Trajectory** (Trip), VehicleId, VehicleType, VehicleNum, VehicleDesc

FROM vehicle WHERE VehicleId = 4

Q15 : SELECT VehicleId, VehicleType, VehicleNum, VehicleDesc, **Trajectory** (Trip)

FROM vehicle WHERE VehicleId = 4

Q16 : SELECT VehicleId, **Trajectory** (Trip), VehicleType, VehicleNum, VehicleDesc

FROM vehicle WHERE VehicleId = 4

Group 3:

Q17 : SELECT **MCount** (sections) FROM ForbiddenRoad

Q18 : SELECT **MCount** (sections) FROM ForbiddenRoad WHERE VehicleType = 1

Q19 : SELECT **MCount** (sections), VehicleType FROM ForbiddenRoad

Q20 : SELECT VehicleType, **MCount** (sections) FROM ForbiddenRoad

Q21 : SELECT **MCount** (sections), VehicleType

FROM ForbiddenRoad WHERE VehicleType = 1

Q22 : SELECT VehicleType, **MCount** (sections)

FROM ForbiddenRoad WHERE VehicleType = 1

Group 4:

Q23 : SELECT **MCount** (**Trajectory** (Trip)) FROM vehicle

Q24 : SELECT **MCount** (**Trajectory** (Trip)) FROM vehicle WHERE VehicleId = 3456

Q25 : SELECT **MCount** (**Trajectory** (Trip)), VehicleId, VehicleType, VehicleNum,

VehicleDesc FROM vehicle

Q26 : SELECT VehicleId, VehicleType, VehicleNum, VehicleDesc,
MCount (**Trajectory**(Trip)) FROM vehicle

Q27 : SELECT VehicleId, **MCount** (**Trajectory** (Trip)), VehicleType, VehicleNum,
VehicleDesc FROM vehicle

Q28 : SELECT **MCount** (**Trajectory** (Trip)), VehicleId, VehicleType, VehicleNum,
VehicleDesc FROM vehicle WHERE VehicleId = 3456

Q29 : SELECT VehicleId, VehicleType, VehicleNum, VehicleDesc,
MCount (**Trajectory**(Trip)) FROM vehicle WHERE VehicleId = 3456

Q30 : SELECT VehicleId, **MCount** (**Trajectory** (Trip)), VehicleType, VehicleNum,
VehicleDesc FROM vehicle WHERE VehicleId = 3456

Group 5:

Q31 : SELECT **MSum** (fee) FROM road

Q32 : SELECT **MSum** (fee) FROM road WHERE id = 1

Part IV

Conclusion

Chapter 9

Conclusion

9.1 Summary of thesis' contributions

The tight integration of (secure) mobile computing, wireless communication and sensors in small, portable devices has lead to an unprecedented production and consumption of personal mobility data. These data are of great value for many applications and businesses, and hold the promise of allowing society a more sustainable development (e.g., in areas such as smart cities or smart home) through a better usage and understanding of the data. However, many challenges, in particular related to data management, still need to be solved before transforming this vision into reality. This thesis focuses on two such challenges. The first challenge concerns the efficient management of large amounts of spatio-temporal data flows on specific storage devices. The second challenge is related to personal mobile data aggregation which preserves the users' privacy. Specifically, there are three main contributions in this thesis as following:

- *We have proposed TRIFL, an efficient and generic TRajectory Index for FLash.* The change in storage environment from traditional magnetic disk to flash storage, due to the advantages of flash memory such as high performance, low power consumption and shock resistance, has brought new constraints (e.g., erase-before-write mechanism, asymmetric read/write, fast large granularity IO and etc.) and therefore indexing techniques need to be reconsidered. Among these changes, we introduced TRIFL dedicated for trajectory object. TRIFL is built based on an Append-Only B^+-Tree (B^{AO+}-Tree) and Time Interval Index (TII) structures combined with an intelligent buffer management technique (Hot-buffer). These ideas allow TRIFL to reconcile the features of trajectory objects and flash environment and therefore achieve much better performance in both Flash and magnetic storage compared to its competitors. Moreover, we also offered a self-turning machine for TRIFL that allows adapting the index structure to the different types of workloads as well as the

specific characteristics of storage devices.

- *We have proposed PAMPAS, a Privacy-Aware Mobile PArticipatory Sensing system based on secure mobile probes.* Mobile participatory sensing has been applied in many applications ranging from environmental monitoring to traffic monitoring. However, its success depends on finding a solution for querying large numbers of users which protects users location privacy and works in real-time. Due to these reasons, we pursued and proposed PAMPAS, a privacy-aware mobile distributed system for efficient data aggregation in mobile participatory sensing. The main ideas behind are to use secure hardware solutions incorporate a secure-centric privacy-aware protocol and therefore aggregation processes are delegated to a list of secure devices and executed in parallel without any privacy leakage. Furthermore, we also offer an embedded partitioning algorithm and aggregate algorithms that give PAMPAS further strength in order to be easily applied in any real-time and large scalability participatory sensing system.

- *We implemented a prototype in the context of the secure Personal Data Server.* This prototype has two major components, which are embedded in a secure portable token. The first component is an extension of an embedded relational database engine (i.e., PlugDB [82]) to deal with spatio-temporal data. This extension consists in new data types and new operators allowing to locally manage spatio-temporal data in conjunction with the classical data types and operations in the relational model. The second component is a secure global aggregation protocol based on PAMPAS and applied to the context of traffic monitoring using secure devices. We call this component PPTM (Privacy-aware Participatory Traffic Monitoring). The implementation of PPTM can be executed in many secure tokens in parallel and shows the feasibility of doing privacy-preserving participatory sensing using secure mobile devices.

9.2 Perspective work

As smart objects become more and more popular nowadays, managing personal data in such devices as well as caring also the advantages of their security (i.e., secure processing and also secure storage of cryptographic material) has become an important research topic during the last decade. For this reason, there are many research directions that could be developed following the work in this thesis such as spatio-temporal personal data servers, distributed computation on top of secure devices infrastructure or privacy-preserving in participatory sensing systems with secure devices (e.g., Trustzone supported devices [12], [108]). However, we believe that the following research directions are particularly promising:

- *Efficient stream data management for smart objects.* The advent of smart objects and Internet of Things has a strong impact on several aspects of people's daily life in terms of data acquisition. For instance, smart meters are used to monitoring, billing, regulating and even saving the electrical, gas or water consumption with the help of high quality and real-time sensors. Similarly, quantified-self applications uses related personal physical data in order to improve the one's health and well-being by tracking different external factors (e.g., consumed food, air quality, temperature or surrounding noise) or internal states (e.g., heart pulse, breath rate or oxygen levels) [65], [102], [119]. In many cases, personal information and sensing data are collected locally on the smart devices before reporting or sharing the information to a central server or a Personal Cloud. As a result, stream data management in smart objects is of prime importance. In this context, several problems have to be taken into account. Firstly, there are considerable differences in the hardware resources between smart devices (e.g., a smart meter using a smart card versus an application running using Trustzone in a smartphone) and therefore, an embedded database engine needs to be adapted to each specific scenario. Secondly, in order to avoid flooding the network bandwidth and given the requirements of the above mentioned applications, the data recorded by the devices is not necessary sent in real-time. Instead, it could be sent in batch daily or after longer time durations. To this end, the stream of recorded data needs to be stored locally in smart devices at least for some period of time. Keeping the data locally in smart objects has also the prominent advantage of increasing the security due to the data distribution. Therefore, we need to design and implement embedded database engines that allow to efficiently buffer, store and query streams of sensed data in smart devices.

- *Data compression/aggregation/degradation/aging/cleaning techniques.* Given the large amounts of data generated and shared by the users, there is a high risk of waste of resources and large maintenance costs incurred by data transfer and storage. Locally managing data in the smart devices comes as a smart solution. However, dealing with large amounts of data in constrained environments is still challenging. Complementary to studying efficient data management techniques in smart objects, we also need to consider the problem of embedded data compression. For instance, the precision needed by an application often decreases as the data gets older, leading to study embedded data compression/aggregation/degradation/aging techniques to minimize the data volume (and thereby the privacy risk) while preserving the usages on the data. Also, indexing could be selectively applied to the data as they are queried, in the same spirit as Database Cracking [49], [46] or Data Virtualization principles in order to concentrate the resources on the useful part of the data.

- *Gossip-like computation protocols in the PAMPAS architecture.* Currently, the proposed PAMPAS system is built on the assumption that the secure tokens can preclude all the

attacks coming from the token holder. This is a reasonable assumption given the tamper-resistance protection of the secure MCU and the fact that the NAND Flash persistent storage is cryptographically protected. However, with sufficient effort and time, any security can be broken, i.e., a laboratory attack. In PAMPAS, if a secure token is compromised, the attacker can gain access to the secret information protected by the token (e.g., the encryption key used to encrypt the data that is exchanged between the users). In this case, the attacker could collude with the SSI to gain access to all the raw sensed data, and completely compromise the privacy of the other users.

In order to minimize the risks of a laboratory attack, we plan to use a Gossip-like protocol [54] in PAMPAS. Concretely, the improvements in our protocol are twofold: (1) We eliminate the collection phase (in which the SSI gathers encrypted samples from users) and we only use the SSI for broadcasting aggregate results which do not contain any personal sensitive information. (2) We aggregate the information based on a Gossip-based computation protocol. This Gossip approach requires all secure probes (SPs) to participate to the aggregation processes. SPs need to send their samples randomly to other SPs in their partitions and compute partial aggregate results once they receive samples from other SPs. Consequently, partial aggregate results are propagated within partitions and final aggregate results should be obtained after a number of rounds at all SPs [54]. Finally, these final results are broadcast to the other participants in different partitions via SSI.

Globally, higher privacy guarantees are achieved by trading off the efficiency of PAMPAS. Compared to our original architecture, applying a Gossip-based computation protocol incurs higher communication and computation costs and therefore execution time to finish the data aggregation is longer than in the original PAMPAS protocol. The explanations for this are as following. First, the computation process based on Gossip requires approximately $n = log(users)$ rounds instead of one. Second, the samples and the partial results are propagated randomly between the users during these rounds and therefore this leads to a larger communication cost. However, with the Gossip-based protocol, there is no shared secret key used to encrypt all the exchanged data. In the Gossip-based protocol, all the communications between the users are point-to-point and can be secured using a typical asymmetric, public/private key encryption. Hence, in the unlikely event of a secure token being compromised, the attacker can only gain access at most to the data that transits that token (i.e., a few raw data and some partial aggregate data). This makes the Gossip-based protocol a better candidate for scenarios in which stronger security guarantees are required.

Bibliography

[1] D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990.

[2] S. Aflaki, N. Meratnia, M. Baratchi, and P. J. M. Havinga. Evaluation of incentives for body area network-based healthcare systems. In *IEEE ISSNIP*, 2013.

[3] T. F. N. R. Agency. Project Keep your Information Safe and Secure (KISS). `http://www.agence-nationale-recherche.fr/?Project=ANR-11-INSE-0005`, 2015. [Online; accessed 17-Sep-2015].

[4] C. C. Aggarwal, N. Ashish, and A. Sheth. *The Internet of Things: A Survey from the Data-Centric Perspective*. Springer US, 2013.

[5] D. Agrawal, A. El Abbadi, and S. Wang. Secure data management in the cloud. In *Proceedings of the 7th International Conference on Databases in Networked Information Systems*, DNIS'11, pages 1–15, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, Aug. 2009.

[7] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. Le Folgoc, B. Nguyen, P. Pucheral, I. Ray, I. Ray, and S. Yin. Secure Personal Data Servers: A vision paper. *Proc. VLDB Endow.*, 3(1-2):25–35, Sept. 2010.

[8] T. Allard, B. Nguyen, and P. Pucheral. METAP: revisiting privacy-preserving data publishing using secure devices. *Distributed and Parallel Databases*, 32(2):191–244, 2014.

[9] N. Anciaux, L. Bouganim, Y. Guo, P. Pucheral, J.-J. Vandewalle, and S. Yin. Pluggable personal data servers. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1235–1238, New York, NY, USA, 2010. ACM.

[10] N. Anciaux, L. Bouganim, P. Pucheral, Y. Guo, L. Le Folgoc, and S. Yin. MILo-DB: A personal, secure and portable database machine. *Distrib. Parallel Databases*, 32(1):37–63, Mar. 2014.

[11] N. Anciaux, S. Lallali, I. Sandu Popa, and P. Pucheral. A scalable search engine for mass storage smart objects. *Proc. VLDB Endow.*, 8(9):910–921, May 2015.

[12] ARM. *ARM Security Technology - Building a Secure System using TrustZone Technology*. ARM Technical White Paper, 2009.

[13] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3 – 15, 1997.

[14] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283, 2014.

[15] C. Bolchini, F. Salice, F. A. Schreiber, and L. Tanca. Logical and physical design issues for smart card databases. *ACM Trans. Inf. Syst.*, 21(3):254–285, July 2003.

[16] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. PIST: An efficient and practical indexing technique for historical spatio-temporal point data. *Geoinformatica*, 12(2):143–168, June 2008.

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

[18] J. W. S. Brown, O. Ohrimenko, and R. Tamassia. Haze: privacy-preserving real-time traffic statistics. In *ACM SIGSPATIAL*, pages 540–543, 2013.

[19] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.

[20] X. Chen and J. Pang. Protecting query privacy in location-based services. *Geoinformatica*, 18(1):95–133, Jan. 2014.

[21] J. T. Child and S. Petronio. Unpacking the paradoxes of privacy in cmc relationships: The challenges of blogging and relational communication on the internet. In *In Computer-mediated communication in personal relationships*, pages 21–40, 2011.

[22] D. Christin, A. Reinhardt, S. S. Kanhere, and M. Hollick. A survey on privacy in mobile participatory sensing applications. *J. Syst. Softw.*, 84(11):1928–1946, Nov. 2011.

[23] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos. AnonySense: Privacy-aware people-centric sensing. In *MobiSys*, 2008.

[24] P. Cudré-Mauroux, E. Wu, and S. Madden. TrajStore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120. IEEE, IEEE, 2010.

[25] V. T. de Almeida and R. H. Guting. Indexing the trajectories of moving objects in networks. *GeoInformatica*, 9:33–60, 2005.

[26] Y.-A. de Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3, 2013.

[27] E. D'Hondta, M. Stevensb, and A. Jacobs. Participatory noise mapping works! an evaluation of participatory sensing as an alternative to standard techniques for environmental monitoring. *Pervasive and Mobile Computing*, 9(5):681–694, October 2013.

[28] G. Drosatos, P. S. Efraimidis, I. N. Athanasiadis, and M. Stevens. A privacy-preserving cloud computing system for creating participatory noise maps. In *COMPSAC*, pages 581–586, 2012.

[29] T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert, and M. Thoma. On the impact of Flash SSDs on spatial indexing. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 3–8, New York, NY, USA, 2010. ACM.

[30] M. Faezipour, M. Nourani, A. Saeed, and S. Addepalli. Progress and challenges in intelligent vehicle area networks. *Magazine Communications of the ACM*, 55(2):90–100, 2012.

[31] C. Faloutsos. Multiattribute hashing using gray codes. *SIGMOD Rec.*, 15(2):227–238, June 1986.

[32] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. Softw. Eng.*, 14(10):1381–1393, Oct. 1988.

[33] C. Faloutsos and Y. Rong. DOT: A spatial access method using fractals. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 152–159, Washington, DC, USA, 1991. IEEE Computer Society.

[34] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 247–252, New York, NY, USA, 1989. ACM.

[35] M. G. M. *A computer oriented geodetic data base and a new technique in file sequencing*. Ottawa : International Business Machines Co., 1966, 1990.

[36] R. K. Ganti, N. Pham, H. Ahmadi, S. Nangia, and T. F. Abdelzaher. GreenGPS: A participatory sensing fuel-efficient maps application. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 151–164, New York, NY, USA, 2010. ACM.

[37] R. K. Ganti, N. Pham, Y.-E. Tsai, and T. F. Abdelzaher. PoolView: Stream privacy for grassroots participatory sensing. In *SenSys*, 2008.

[38] R. K. Ganti, F. Ye, and H. Lei. Mobile crowdsensing: Current state and future challenges. *IEEE Communications Magazine*, 49(11):32–39, Nov. 2011.

[39] H. Gao, C. H. Liu, W. Wang, J. Zhao, Z. Song, X. Su, J. Crowcroft, and K. K. Leung. A survey of incentive mechanisms for participatory sensing. *IEEE Comm. Surveys and Tutorials*, 17(2):918–943, 2015.

[40] G. Ghinita. Private Queries and Trajectory Anonymization: A dual perspective on location privacy. *Trans. Data Privacy*, 2(1):3–19, Apr. 2009.

[41] glaros.dtc.umn.edu. METIS-Family of Multilevel Partitioning Algorithms. `http://glaros.dtc.umn.edu/gkhome/views/metis`, 2014. [Online; accessed 7-May-2014].

[42] R. H. Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, Oct. 1994.

[43] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, Mar. 2000.

[44] R. H. Güting, T. de Almeida, and Z. Ding. Modeling and querying moving objects in networks. *The VLDB Journal*, 15(2):165–190, June 2006.

[45] R. H. Guting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.

[46] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, Feb. 2012.

[47] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE Pervasive Comp.*, 5(4):38–46, 2006.

[48] B. Hoh, T. Iwuchukwu, Q. Jacobson, D. Work, A. M. Bayen, R. Herring, J. C. Herrera, M. Gruteser, M. Annavaram, and J. Ban. Enhancing privacy and accuracy in probe vehicle-based traffic monitoring via virtual trip lines. *IEEE Tran. on Mobile Computing*, 11(5):849–864, 2012.

[49] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *In CIDR*, 2007.

[50] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2):332–342, May 1990.

[51] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik. Towards a streaming sql standard. In *PVLDB 1(2)*, pages 1379–1390, 2008.

[52] S. S. Kanhere. Participatory sensing: Crowdsourcing data from mobile smartphones in urban spaces. *Distributed Computing and Internet Technology*, 7753:19–26, 2013.

[53] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[54] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, pages 482–, Washington, DC, USA, 2003. IEEE Computer Society.

[55] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. on Knowl. and Data Eng.*, 13(5):758–777, Sept. 2001.

[56] M. Kornacker and D. Banks. High-concurrency locking in R-Trees. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 134–145, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[57] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48(9):140–150, Sept. 2010.

[58] S.-W. Lee and B. Moon. Design of Flash-based DBMS: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 55–66, New York, NY, USA, 2007. ACM.

[59] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[60] Q. Li and G. Cao. Efficient and privacy-preserving data aggregation in mobile sensing. In *IEEE ICNP*, 2012.

[61] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, Sept. 2010.

[62] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. FD-Tree: A tree index on solid state drivevs. `http://pages.cs.wisc.edu/~yinan/fdtree.html`, 2014. [Online; accessed 7-May-2014].

[63] D. Lin, C. S. Jensen, B. C. Ooi, and S. Šaltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *Proceedings of the 6th International Conference on Mobile Data Management*, MDM '05, pages 59–66, New York, NY, USA, 2005. ACM.

[64] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Efficient indexing data structures for flash-based sensor devices. *Trans. Storage*, 2(4):468–503, Nov. 2006.

[65] LiveScience. The quantified self. `http://www.livescience.com/topics/quantified-self/`, 2015. [Online; accessed 7-Oct-2015].

[66] F. Manola and J. A. Orenstein. Toward a general spatial data model for an object-oriented DBMS. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB'86, pages 328–335, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[67] M. F. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel. MNTG: An extensible web-based traffic generator. In *Proceedings of the 13th International Conference on Advances in Spatial and Temporal Databases*, SSTD'13, pages 38–55, Berlin, Heidelberg, 2013. Springer-Verlag.

[68] M. F. Mokbel and W. G. Aref. Location-aware query processing and optimization. In *IEEE MDM*, page 229, 2007.

[69] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26:40–49, 2003.

[70] G. E. Moore. Moore's law. `https://en.wikipedia.org/wiki/Moore%27s_law`, 2015. [Online; accessed 20-Aug-2015].

[71] M. E. Nergiz, M. Atzori, Y. Saygin, and B. Guc. Towards Trajectory Anonymization: A generalization-based approach. *Trans. Data Privacy*, 2(1):47–75, Apr. 2009.

[72] L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-temporal access methods: Part 2 (2003-2010). *IEEE Data Engineering Bulletin*, 33(2):46–55, 2010.

[73] H. Nissenbaum. *Privacy in Context Technology, Policy, and the Integrity of Social Life*. Stanford University Press, 2009.

[74] S. Nittel, J. C. Whittier, and Q. Liang. Real-time spatial interpolation of continuous phenomena using mobile sensor data streams. In *ACM SIGSPATIAL*, pages 530–533, 2012.

[75] ocz.com. OCZ, SSD OCZ Vertext 4 SATA 3, Specifications. `http://ocz.com/consumer/vertex-4-sata-3-ssd/specifications`, 2014. [Online; accessed 7-May-2014].

[76] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[77] J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. *SIGMOD Rec.*, 19(2):343–352, May 1990.

[78] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of SIGMOD'89*, pages 295–305, 1989.

[79] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '84, pages 181–190, New York, NY, USA, 1984. ACM.

[80] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 7(4):513–532, Aug. 1995.

[81] B. Palanisamy and L. Liu. MobiMix: Protecting location privacy with mix-zones over road networks. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 494–505, Washington, DC, USA, 2011. IEEE Computer Society.

[82] I. Paris. PlugDB. `https://project.inria.fr/plugdb/`, 2015. [Online; accessed 17-Sep-2015].

[83] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An efficient index for predicted trajectories. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 635–646, New York, NY, USA, 2004. ACM.

[84] M. Pelanis, S. Šaltenis, and C. S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Syst.*, 31(1):255–298, Mar. 2006.

[85] M. Penza. COST action TD1105: New sensing technologies for environmental sustainability in smart cities. In *IEEE SENSORS*, 2014.

[86] PewResearchCenter. Cell Phones in Africa: Communication lifeline. http://www.pewglobal.org/2015/04/15/cell-phones-in-africa-communication-lifeline/, 2015. [Online; accessed 8-Oct-2015].

[87] PewResearchCenter. Communications technology in emerging and developing nations. http://www.pewglobal.org/2015/03/19/1-communications-technology-in-emerging-and-developing-nations/, 2015. [Online; accessed 8-Oct-2015].

[88] PewResearchCenter. Device ownership over time. http://www.pewinternet.org/data-trend/mobile/device-ownership/, 2015. [Online; accessed 17-Sep-2015].

[89] PewResearchCenter. Emerging nations embrace internet, mobile technology. http://www.pewglobal.org/2014/02/13/emerging-nations-embrace-internet-mobile-technology/, 2015. [Online; accessed 8-Oct-2015].

[90] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and accountability for location-based aggregate statistics. In *CCS*, pages 653–666, 2011.

[91] P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. PicoDBMS: Scaling down database techniques for the smartcard. *The VLDB Journal*, 10(2-3):120–132, Sept. 2001.

[92] D. Quercia, I. Leontiadis, L. Mcnamara, C. Mascolo, and J. Crowcroft. Spotme if you can: Randomized responses for location obfuscation on mobile phones. In *ICDCS*, pages 363–372, 2011.

[93] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.*, 5(4):286–297, Dec. 2011.

[94] I. Sandu Popa. *Modeling, Querying and Indexing Moving Objects with Sensors on Road Networks*. PhD thesis, University of Versailles-Saint-Quentin, 2005.

[95] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial. PARINET: A tunable access method for in-network trajectories. In *ICDE*, pages 177–188, 2010.

[96] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial. Indexing in-network trajectory flows. *The VLDB Journal*, 20(5):643–669, Oct. 2011.

[97] M. Sarwat, M. F. Mokbel, X. Zhou, and S. Nath. FAST: A generic framework for flash-aware spatial trees. In *Proceedings of the 12th International Conference*

on Advances in Spatial and Temporal Databases, SSTD'11, pages 149–167, Berlin, Heidelberg, 2011. Springer-Verlag.

[98] M. Sarwat, M. F. Mokbel, X. Zhou, and S. Nath. Generic and efficient framework for search trees on flash memory storage systems. *Geoinformatica*, 17(3):417–448, July 2013.

[99] P. Schimid and A. Roos. SDXC/SDHC memory cards, rounded up and benchmarked. `http://tinyurl.com/tom-sdxc`, 2014. [Online; accessed 7-May-2014].

[100] I. Schweizer, C. Meurisch, J. Gedeon, R. Bartl, and M. Muhlhauser. Noisemap: multi-tier incentive mechanisms for participative urban sensing. In *PhoneSense*, 2012.

[101] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[102] Q. Self. Quantified self. `http://quantifiedself.com/`, 2015. [Online; accessed 7-Oct-2015].

[103] J. Shi, R. Zhang, Y. Liu, and Y. Zhang. PriSense: Privacy-preserving data aggregation in people-centric urban sensing systems. In *IEEE INFOCOM*, 2010.

[104] R. Shokri, G. Theodorakopoulos, C. Troncoso, J. pierre Hubaux, and J. yves Le Boudec. Protecting location privacy: optimal strategy against localization attacks. In *CCS*, pages 617–627, 2012.

[105] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[106] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 431–440, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[107] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 790–801. VLDB Endowment, 2003.

[108] Texas Instruments. Get Into the Zone: Building secure systems with arm trustzone technology. *White Paper*, Feb 2013.

[109] A. Thiagarajan, J. Biagioni, T. Gerlich, and J. Eriksson. Cooperative transit tracking using smart-phones. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 85–98, New York, NY, USA, 2010. ACM.

[110] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. VTrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *ACM SenSys*, pages 85–98, 2009.

[111] Q.-C. To, B. Nguyen, and P. Pucheral. Privacy-preserving query execution using a decentralized architecture and tamper resistant hardware. In *EDBT*, pages 487–498, 2014.

[112] D. H. Ton That, I. Sandu Popa, and K. Zeitouni. Management of spatial-temporal data stream in embedded systems. Technical report, PRISM Laboratory, University of Versailles, 2012.

[113] D.-H. Ton-That, I. Sandu-Popa, and K. Zeitouni. PPTM: Privacy-aware participatory traffic monitoring using mobile secure probes. In *IEEE MDM*, 2015. Demo paper.

[114] D. H. Ton That, I. Sandu Popa, and K. Zeitouni. TRIFL: A generic trajectory index for flash storage. *ACM Transaction on Spatial Algorithms and Systems*, 1(2), 2015.

[115] A. Troussov. General DBMS architecture. Technical report, INRIA-Rocquencourt, 2012.

[116] N. Tsiftes and A. Dunkels. A database in every sensor. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 316–332, New York, NY, USA, 2011. ACM.

[117] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Rec.*, 29(2):331–342, May 2000.

[118] B. Wang and J. S. Baras. Hybridstore: An efficient data management system for hybrid flash-based sensor devices. In *Proceedings of the 10th European Conference on Wireless Sensor Networks (EWSN)*, EWSN'13, pages 50–66, 2013.

[119] Wikipedia. Quantified self. `https://en.wikipedia.org/wiki/Quantified_Self`, 2015. [Online; accessed 7-Oct-2015].

[120] World Economic Forum. *Personal Data: The Emergence of a New Asset Class*. The World Economic Forum, 2011.

[121] World Economic Forum. *Rethinking Personal Data: Strengthening Trust*. The World Economic Forum, 2012.

[122] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[123] S. Yin, P. Pucheral, and X. Meng. A sequential indexing scheme for flash-based embedded systems. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 588–599, New York, NY, USA, 2009. ACM.

[124] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 791–800, New York, NY, USA, 2009. ACM.